Programmierparadigma: Funktionale Programmierung

Christoph Klotz

Inhaltsverzeichnis

1 Introduction

Since there was programming there was functional programming, not to say even longer. It began with mathematican who wanted to formalize mathematics with functions and ended with a lot of different realisations of implementations. Functional programming is not that popular but has some important aspects, which made it survive its counterpart, the imperative programming style.

2 Functional programming

This type of programming is so called because its programs consist only of functions. Typically the main function consists of other functions, which are also seperated in more functions until at the bottom level the functions are primitives. This programs are then executed by evaluating these functions and expressions.

2.1 Characteristics and advantages

2.1.1 Higher-order functions

Higher-order functions are functions, which take as their argument other functions or are returning functions as their result. They are very useful to avoid simple loops respectively recursive functions. One of these Higher-order function is `map` which executes another function on every instance of a list. An example with Ocaml [4]:

recursive function:

```
#let rec doubleAll=function
  [] -> []
  |(x::xs)->(2*x)::(doubleAll xs);;
```

Higher-order function:

```
# let double x=2*x;;

# let doubleAll lst=map double lst;;
```

2.1.2 Purity

Pure functional programms normally operate on unchangeable data, to prevent side effects. The function only computes the expression and does nothing but return the result. Typical side-effects would be incremented count variables or printed strings.
That also means, everytime an expression is computed with a certain argument, it returns the same result. This is called *referential transparency* which would make it possible to replace two functions like $y = fx$ and $g = hyy$ with one like $g = h(fx)(fx)$.
With these two properties there can be found the conclusion, that pure computations can be performed any time and will return the same result. So it is possible to postpone computations until they are needed. This is called textit-Lazy evaluation and avoids unnecessary evaluations and makes it possible to use infinite data structures like the fibonacci numbers, the primes or any other number type [3].

2.1.3 Recursion

A recursive function is an expression which is computed by calling itself, just like the code in 2.1.1. Recursion is often used in functional programming because it is basically the only way to iterate. Recursive functions sometimes need a lot of memory and to prevent a memory overflow the functions are implemented tail-recursive. This form of recursion needs almost no memory. The tail-recursive code to the example of 2.1.1:

```
#let doubleAll xs =
   let rec doubleAll acc = function
     [] -> acc
     |y::ys -> doubleAll (acc@[2*y]) ys
 in
   doubleAll [] xs;;
```

2.2 Types

At first functional programming languages[1] only supported dynamic typing, which means functions and values got their types not until runtime. But with the discovery of the Hindley-Milner type inference for the lambda calculs it was possible to implement languages which were able to do static typing. These programming languages already typed their functions at compiletime which can prevent failures at runtime. Programmers didn't need to type their functions and values because this was from now on the job of the compiler.

3 The Lambda Calculus

A main foundation of the functional programming is Alonzo Church's Lambda Calculus. Because this calculus was type-free it became small and simple and also had a very interessting characteristic which gives the lambda calculus its power, functions could be applied to themselves [1]. This made it possible to use recursion without defining a recursion explicitly.

3.1 The Lambda Calculus syntax

The abstract syntax of the lambda calculus is called lambda expressions. They are defined by the following BNF:

$x \in Id$	Identifiers
$t \in Exp$	Lambda expressions/Lambda terms
	where $t ::= x \mid t_1 t_2 \mid \lambda x.t$

Expressions in the form $\lambda x.t$ (somehow equivalent to a function definition like $f(x) = t$) are called abstractions and of the form $(t_1 t_2)$ are called applications (these are representing the functions). The evaluation steps of the lambda calculus are mostly substitutions of a term t_1 for all free appearences of an identifier x in another term t_2 ($[t_1/x]t_2$). To understand this we must know what a free variable of such an expression is. We name this free variable FVar(t) and define it as followed [2]:

$FVar(x) = x$
$FVar(t_1 t_2) = FVar(t_1) \cup FVar(t_2)$
$FVar(\lambda x.e) = FVar(e) - x$

so x is free in t if $x \in FVar(t)$

3.2 Evaluation of Lambda Expressions

After knowing the syntax of the lambda calculus it would be interesting to know how to do computations with it. Surprisingly we only need three rules to

[1] like Lisp and its variants

unleashe the full computational power of this calculus.

1. α-conversion (renaming)hudak:
   $\lambda x_i.t \Leftrightarrow \lambda x_j.[x_j/x_i]t$, where $x_j \notin FVar(t)$.
2. β-conversion (application):
   $(\lambda x.t_1)t_2 \Leftrightarrow [t_2/x]t_1$.
3. η-conversion:
   $\lambda x.(ex) \Leftrightarrow t$, if $x \notin fVar(t)$.

3.3 The Normal Forms

A Lambda Term is in normal form if it is no possible to apply any β- or η-reduction. Some lambda expressions, like $(\lambda x.xx)(\lambda x.xx)$, have no normal form. The result of a β-reduction would be the same expression we wanted to compute. In two theorems Church and John Barkley Rosser proved that if a normal form exists, it can be found.

3.4 Encoding datatypes

Church didn't need any arithmetic or boolean values and operators, he was able to emulate them in lambda terms. For example [2]:

0	$:= \lambda fx.x$	$TRUE$	$:= \lambda x.\lambda y.x$
1	$:= \lambda fx.fx$	$FALSE$	$:= \lambda x.\lambda y.y$
ADD	$:= \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$	AND	$:= \lambda p.\lambda q.pqp$

This can be accomplished for every datatype, even more complex like pairs, lists and it is even possible to write recursive functions.

4 Conclusion

Functional programming is not an unimportant programming style, in fact it is used quite often, but the popular style is the very imperative programming.

Literatur

[1] Hudak, Paul: Conception, evolution, and application of functional programming languages (1989)

[2] Sternagel, Christian and Zankl, Harald: Functional Programming (with Ocaml) (2011)

[3] `http://www.haskell.org/haskellwiki/Functional_programming`

[4] `http://www.csc.villanova.edu/~dmatusze/resources/ocaml/ocaml.html#Higher-order%20functions`