The Imperative and Functional Programming Paradigm

Henk Barendregt[a], Giulio Manzonetto[a,1], Rinus Plasmeijer[a]

[a] *Faculty of Science, Radboud University, Nijmegen, The Netherlands*

1. Models of computation

In Turing (1937) a characterization is given of those functions that can be computed using a mechanical device. Moreover it was shown that some precisely stated problems cannot be decided by such functions. In order to give evidence for the power of this model of computation, Turing showed in the same paper that machine computability has the same strength as definability via λ-calculus, introduced in Church (1936). This model of computation was also introduced with the aim to show that undecidable problems exist.

Turing Machine computability forms a very simple model that is easy to mechanize. Lambda calculus computability, on the other hand, is *a priori* more powerful. Therefore it is not obvious that it can be executed by a machine. As Turing showed the equivalence of both models also λ-calculus computations can be performed by a machine and also Turing Machine computations are powerful. This gave rise to the combined

Church-Turing Thesis *The notion of intuitive computability is exactly captured by λ-definability or by Turing computability.*

Computability via Turing machines gave rise to Imperative Programming. Computability described via λ-calculus gave rise to Functional Programming. As imperative programs are more easy to run on hardware, this style of software became predominant. We present major advantages of the functional programming paradigm over the imperative one, that are applicable, provided one is has the mental capacity to explicitly deal with simple abstractions.

Email addresses: henk@cs.ru.nl (Henk Barendregt), g.manzonetto@cs.ru.nl (Giulio Manzonetto), rinus@cs.ru.nl (Rinus Plasmeijer)

[1]Supported in part by NWO Project 612.000.936 CALMOC.

2. Functional programming

Features from lambda calculus

Rewriting. Lambda terms form a set of formal expressions subjected to possible *rewriting* (or *reduction*) steps. For each term there are in general several parts that can be rewritten. However, if there is an eventual outcome, in which there is no more possibility to rewrite, it necessarily is unique.

Application. An important feature of the syntax of λ-terms is *application*. Two expressions can be applied to each other: if F and A are λ-terms, then so is FA, with as intended meaning the function F applied to the argument A. Both the function and the argument are given the same status as λ-terms. This implies that functions can be applied to functions, obtaining higher order functions.

Abstraction. Next to application there is *abstraction*. This feature allows to create complex functions. For example given terms F and G intended as functions, then one may form $F \circ G$ and $G \circ F \circ G$ with the rewriting rules

$$\begin{array}{rcl} (F \circ G)\ a & \rightarrow & F(G\ a); \\ (G \circ F \circ G)\ a & \rightarrow & G(F(G\ a)). \end{array}$$

It is interesting to note that there is one single mechanism, λ-abstraction, that can capture both examples and much more. Given a λ-term M in which the variable x may occur, one can form the abstraction $\lambda x.M$. It has as intended meaning the function that assigns to x the value M. More generally $\lambda x.M$ assigns to N the value $M[x{:=}N]$, where the latter denotes the expression obtained by substituting N for x in M. Then one has

$$\begin{array}{rcl} F \circ G & \triangleq & \lambda x.F(Gx); \\ G \circ F \circ G & \triangleq & \lambda x.G(F(G\ x)). \end{array}$$

β-reduction. Corresponding to this abstraction with its intended meaning, there is a single rewriting mechanism. It is called β-reduction and is

$$(\lambda x.M)N \rightarrow M[x{:=}N],$$

giving the two rewrite examples mentioned above from the definition of $F \circ G$ and $G \circ F \circ G$. One can iterate the procedure, and introduce the higher order function C 'composition' as follows.

$$C \triangleq \lambda f \lambda g \lambda x.f(g\ x).$$

Having C one can write $F \circ G \triangleq C\ F\ G$, where in the absence of parentheses one should read this $C\ F\ G$ as $(C\ F)\ G$. Dually the iterated abstraction $\lambda f \lambda g \lambda x.f(g\ x)$ should be read as $\lambda f(\lambda g(\lambda x.f(g\ x)))$.

Instead of λ-abstraction, it is convenient to define functions by their applicative behavior[2]. One then writes `comp f g x = f (g x)`, obtaining 'composition' `comp f g` $= \mathtt{f} \circ \mathtt{g}$. One can even give definitions that are 'looping', like `L x` $= (\mathtt{x}, \mathtt{L}(\mathtt{x}+1))$, so that `L 0 = (0,(1,(2,(3,...))))`. Similarly one can construct the list of all prime numbers. Infinite lists are easier to describe than a list of say 28 elements.

Lazy evaluation. Expressions are evaluated as little as possible. Consider the program `f (L 4)`: the expression `L 4` is computed only when `f` tries to read some input, and is just evaluated for long enough to return a value to `f`. This enables dealing with 'infinite objects', mentioned above.

Features beyond lambda calculus

For the pragmatics of functional programming several features are added to the basic system of λ-calculus.

Data. Although integers and 'scientific' real numbers can be represented as λ-terms, for efficiency reasons they are given by special constants, together with the primitives for standard operations.

Names. The original λ-calculus formalism does not have names *by design*, as arbitrary λ-abstractions can be made. However, for the pragmatics of using and reusing software components, it is useful introduce a naming construction `let`. E.g. `let comp = λfλgλx.f(g x)` means that composition $C \triangleq \lambda f \lambda g \lambda x.f(g\ x)$ is now called `comp` and can be used later to define

$$F \circ G \triangleq \mathtt{comp}\ FG.$$

3. Types

In physics constants have a 'dimension', e.g. speed is measured in km/h. When we bike at $\mathtt{v} = \mathtt{12km/h}$ and we do this for $\mathtt{t} = \mathtt{3h}$, we have gone $\mathtt{vt} = \mathtt{12} \cdot \mathtt{3} = \mathtt{36km}$. Dimensions prevent that we want to consider e.g. $\mathtt{vt}^2$ to compute the distance.

Similarly functional programming languages come with a type system helping to ensure correctness. A program expecting a number (`: Int`) should

[2]This method is called *heuristic application principle* by Böhm.

not receive a judgement (`: Bool`). Giving module `F` a type `A` is denoted by `F : A` (read '`F` in `A`'). One starts typing the data, e.g. `3 : Int`, `True : Bool`. Functions with behavior `G x = y` get as type `A → B`, where `x : A` and `y : B`. An application `F a` is only allowed if the types match, i.e. `F : A → B` and `a : A`.

The functional programmer indicates the types of the data structures and basic functions and the machine performs *type-inferencing* at compile-time. This is a major help for combining software modules in a correct way: many bugs are caught as the result will be an untypable program.

Algebraic Data Types. Next to basic data types, like `Int`, `Bool`, one likes to use common data structures, like lists and trees of elements of type `A`. Such structures start small and grow. There is the empty list `Nil` and one can extend a list `tl` by chaining an element `hd` of type `A`, obtaining `Cons hd tl`. The function that counts the number of elements in a list (of arbitrary type) can be defined by specifying that on the empty list it is 0 and on an enlarged list it is the length of the previous list plus 1.

```
count : List A → Int
count Nil = 0
count (Cons hd tl) = 1 + count tl
```

Using so called Generalized Abstract Data Types one introduces several such types simultaneously, mutually depending on each other, keeping type-inferencing possible, see Schrijvers et al. (2009).

Generic types. One can 'code' algebraic types enabling to write uniformly functions on these. For example, there is a map on lists and on trees of data (hanging at the leaves), whereby a given function `f` is applied to each element. The two functions can be obtained by specializing one program with a generic type, including possible exceptions. With this feature one can embed Domain Specific Languages within a functional language, see Plasmeijer et al. (2007).

Dynamic types. A functional program `P` having type `A` is compiled to machine code and evaluated. During this process the original term, its type and context of definitions are forgotten. Using 'dynamic types' one may keep track of this syntactic data. This enables dynamic code, e.g. for writing typed Operating System in a pure functional language, and dealing with unknown plug-ins or database specifications.

Efficiency. Considerable effort has been put into the compiler technologies for functional languages. The code generated by the state of art compilers for Haskell, OCaML, and Clean is so good, that efficiency is no longer an issue, see the URL `<shootout.alioth.debian.org>`.

An example. Using abstraction, application, algebraic data types and

functions as arguments one can write compact software that can easily be modified. One can construct `foldr` with the following specification.
`foldr` $(\bullet)$ $[a_1, \ldots, a_n]$ `start` $= (a_1 \bullet (a_2 \bullet (\ldots (a_n \bullet \texttt{start}) \ldots))$
Here '$\bullet$' is a binary operation used in infix position and '$(\bullet)$' stands for $\bullet$ as argument. The program `foldr` works on every kind of lists, and subsequently can be applied to particular functions and data-structures, for example to obtain `sumlist`. As a comparison also the imperative code is given. The function `foldr` has the 'schematic type' $(A \to B \to B) \to B \to \texttt{List}\ A \to B$, which does not need to be given by the programmer.

working Functional (Haskell-like)	Imperative (C-like)

`foldr f e Nil = e`
`foldr f e (Cons hd tl) =`
`  f hd (foldr f e tl)`
 	
`sumlist = foldr (+) 0`
	`int sumlist(node *l){`
	`  node *cur = l;`
	`  int res = 0;`
	`  while not (curr = Nil){`
	`  res = res + curr->num;`
	`  curr = curr->next;`
	`  }`
	`  return res;`
	`}`

If one needs the product of the elements of a list, then in the functional case one just adds a new line `prodlist` $=$ `foldr` $(*)$ `1`, whereas in the imperative case one needs to write another procedure exactly like `sumlist`, except for the product that now replaces the sum, thus doubling the lines of code.

4. Input/output

In applications one needs to perform I/O to manipulate information available in peripherals (e.g. keyboard and mouse for input and the file system and screen for output). These characterize the 'state' of the machine and platform running a program. This may effect the content of the file, which is imperative by nature. In *pure*[3] functional languages one has to deal with I/O in a special way, while maintaining modularity, readability, and typability.

In Haskell the state has a particular type `State`. For output one modifies the state ('writing') and for input one makes available to the main functional program a value from the state ('reading', thereby also possibly changing the state). The programming environment of Haskell comes with a collection of

[3] A pure functional language is one without assignments, i.e. statements like `[x:=x+1]`.

write and read operations having the following effect.
`write a state = state'`
`read state = (a,state')`
Together these are called 'actors', with the following type and action.
$\texttt{actor} : \texttt{State} \to \texttt{A} \times \texttt{State}$
$\texttt{actor state} = (\texttt{a}, \texttt{state}')$
Here '`write a`' is interpreted as an actor, for which the `A` is not used. Actors can have an erasing effect on the state. Therefore the programmer has access to the actors, but not to the state. Otherwise one could write
$\texttt{f state} = (\texttt{write 1 state}, \texttt{write 2 state})$ (!)
having a not well-defined effect (depending on execution order). The type `State` remains hidden to the programmer, but not the actors having type $\texttt{Monad A} = \texttt{State} \to (\texttt{A} \times \texttt{State})$, parametrized with a type `A`, on which the program can operate. The entire program is seen as a state modifying function of type `Monad B`, for some type `B`. Composing such functions, using a kind of composition[4], one preserves modularity and compactness. In the meantime any possible computations can take place, by interleaving these with the actors. The monadic approach in Haskell has as advantage that it does not require a special type system to deal with I/O.

In Clean dealing with I/O is more general. Monads are used as well, but also the state on which they act is given to the programmer. This is possible because a *uniqueness type system* warrants safe usage that avoids situations like (!) above. As the state is available, one can split it into different components, like files, the keyboard, and whatever one needs. These can be modified separately, as long as they are not duplicated. Explicit access to the state allow to write the actors within Clean itself.

We see that both in Haskell and in Clean, dealing with I/O comes at a certain price. But this is well worth the advantages of pure functional programming languages: having arbitrarily high meaningful information density, with modules that can be combined easily in a safe way.

5. Current research

Parallelism. Pure functional languages are better equipped for programming multi-cores than imperative languages, as the result of a function is indepen-

[4]As reading actors use the information obtained by modifying the state, it is a serial action with giving over a token, like in a relay race.

dent of its evaluation order. Therefore modules in a functional program can be safely evaluated in parallel. As it costs overhead to send data between processors, one should restrict parallel evaluation to those functions having time consuming computations. Research on parallel evaluation of functional programs, see Hammond and Michaelson (1999), has been revived by the advent of new multi-core machines, Marlow et al. (2009).

Certification. In languages like AGDA, Bove et al. (2009), or Coq, Coq Development Team (2010), one can fully specify a program and prove correctness. This demands even more skills from the programmer than pure functional languages already do: programming becomes proving. But the ideal of 'formal methods' (fully specifying software together with a proof of correctness) has become feasible. For example Leroy (2009) gives a full certification of an optimized compiler for the kernel of the (imperative) language C.

6. History and perspective

The first functional language was Lisp, McCarthy et al. (1962). There is no type system and I/O is done imperatively. By contrast the language ML, Milner et al. (1997), and its modern variant F#, Syme (2007), are impure as well, but strongly typed. Miranda, Turner (1985), was one of the first pure functional programming languages, with lazy evaluation and type inference. Clean, Plasmeijer and van Eekelen (2002), and Haskell, Peyton Jones (2003), are modern variants of Miranda. Haskell has become the *de facto* standard pure functional language, widely used in academia.

Pure functional programming has not yet become mainstream, despite its expressive power and increased safety. To make use of the power, one needs understanding the type systems and the use of the right abstractions. Once mastered, functional programming enables writing applications in a fraction of the usual development and debugging time.

References

Bove, A., Dybjer, P., Norell, U., 2009. A brief overview of Agda - a functional language with dependent types. Vol. 5674 of Lecture Notes in Computer Science. Springer, pp. 73–78.

Church, A., 1936. An unsolvable problem of elementary number theory. American Journal of Mathematics 58, 345–363.

Coq Development Team, 2010. The Coq Proof Assistant Reference Manual – Version V8.3. `http://coq.inria.fr`.

Hammond, K., Michaelson, G., 1999. Research Directions in Parallel Functional Programming. Springer.

Leroy, X., 2009. A formally verified compiler back-end. Journal of Automated Reasoning 43 (4), 363–446.

Marlow, S., Jones, S. L. P., Singh, S., 2009. Runtime support for multicore Haskell. In: Hutton, G., Tolmach, A. P. (Eds.), Proc. Int. Conf. on Functional programming 2009. ACM, pp. 65–78.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., Levin, M. I., 1962. LISP 1.5 Programmer's Manual. MIT Press.

Milner, R., Tofte, M., Harper, R., McQueen, D., 1997. The Definition of Standard ML. The MIT Press.

Peyton Jones, S. (Ed.), 2003. Haskell 98 language and libraries: the revised report. Cambridge University Press.

Plasmeijer, R. M., Achten, P., Koopman, P. W. M., 2007. iTasks: executable specifications of interactive work flow systems for the web. In: Hinze, R., Ramsey, N. (Eds.), Proc. Int. Conf. on Functional Programming 2007. ACM, pp. 141–152.

Plasmeijer, R. M., van Eekelen, M. C. J. D., 2002. Clean Language Report. University of Nijmegen, software Technology Group.

Schrijvers, T., Jones, S. L. P., Sulzmann, M., Vytiniotis, D., 2009. Complete and decidable type inference for GADTs. In: Hutton, G., Tolmach, A. P. (Eds.), Proc. Int. Conf. on Functional programming 2009. ACM, pp. 341–352.

Syme, D. and Granicz, A. and Cisternino, A., 2007. Expert F#. Apress.

Turing, A. M., 1937. Computability and lambda-definability. The Journal of Symbolic Logic 2 (4), 153–163.

Turner, D. A., 1985. Miranda: A non-strict functional language with polymorphic types. In: FPCA. pp. 1–16.