FUNCTIONAL PROGRAMMING

INTRODUCTION

Functional programs are written by composing expressions that can have values of any type, including functions and large unbounded data structures. The functional programming paradigm avoids the complications of imperative programming language features, such as mutable variables and statements, in favor of a small set of highly orthogonal language constructs. The simplicity of the functional computation model assists in writing modular programs, that is, programs that separate concerns, reuse code, and provide clear interfaces between modules.

A *functional programming language* encourages and supports the functional programming style. Languages such as Haskell (1,2) and Clean (3) are called *purely functional*. Many widely used functional languages, such as Lisp (4), Scheme (5,6), ML (7,8), and Erlang (9), still include imperative language features that conflict with functional programming ideals. Other languages, such as APL and Python, include constructs that support functional programming but are not considered functional programming languages because they do not strongly encourage a functional programming style. The programming language of the Mathematica system and the XML transformation language XSLT are functional languages. Popular spreadsheet languages, such as Microsoft Office Excel, are restricted functional languages.

Like logic programming, functional programming is a *declarative programming paradigm*. This paradigm comprises programming on a high level, expressing tasks directly in terms of the problem domain, without having to deal with implementation details such as memory allocation. Nonetheless, functional programming is not about specifying a problem without knowing a constructive solution; instead, functional programming allows the programmer to describe algorithms without distraction by unnecessary, machine-related details.

Historically, functional languages have been used intensively for artificial intelligence and symbolic computations. More generally, functional languages are often chosen for rapid prototyping and the implementation of complex algorithms, possibly working on complex data structures, that are hard to "get right" in other programming paradigms.

CHARACTERISTIC FEATURES

Functional programming is characterised by a small set of language and programming style features.

Expressions have Values—No Side Effects

In imperative programming, every procedure or method is defined by a sequence of statements. A computation consists of sequential execution of one statement after the other; each statement changes the global state of the computation, that is, changes the values of variables, writes output, or reads input. Such changes of a global state are called side effects.

In contrast, a functional program consists of a set of function definitions. Each function is defined by an expression. Expressions are formed by applying functions to other expressions; otherwise, constants, variables, and a few special forms are also expressions. Every expression has a value. A computation consists of determining the value of an expression, that is, evaluating it. This evaluation of an expression only determines its value and has no side effects. A variable represents a fixed value, not a memory location whose content can be modified.

Unless otherwise stated, all examples in this article will be written in Haskell (1,2), a purely functional programming language. The following program defines two functions, `isBinDigit` and `max`. Each definition consists of a line that declares the type of the function, that is, its argument and result types, and the actual definition is given in the form of a mathematical equation. The function `isBinDigit` takes a character as argument and decides whether this character is a binary digit, that is, whether it is the character `'0'` or (operator `||`) the character `'1'`. The function max takes two integer arguments and returns the greater of them. In contrast to imperative languages, the conditional if then else is an expression formed from three expressions, here `x > y`, `x` and `y`, *not* a statement with statements after `then` and `else`.

```
isBinDigit :: Char -> Bool
isBinDigit x = (x == '0') || (x == '1')

max :: Integer -> Integer -> Integer
max x y = if x > y then x else y
```

In standard mathematics and most (also functional) programming languages, the arguments of a function are surrounded by parenthesis and separated by commas. In Haskell, however, functions and their arguments are separated by blanks. So `isBinDigit 'a'` evaluates to False and `max 7 4` evaluates to `7`, whereas `max (7,4)` is not a valid expression. Parenthesis are needed to group subexpressions; for example, `max 7 (max 4 9)` evaluates to `9`, whereas `max 7 max 4 9` is an invalid expression. This syntax is convenient when higher-order functions are used (see the section below titled "Higher-Order Functions").

Iteration Through Recursion

Functional programming disallows or at least discourages a modification of the value of a variable. So how shall an iterative process be programmed? Imperative languages use loops to describe iterations. Loops rely on mutable variables so that both the loop condition changes its value and the desired result is obtained accumulatively. For example, the following program in the imperative language C computes the product of all numbers from 1 to a given number `n`.

```
int factorial(int n){
int res = 1;
while (n > 1){
  res = n * res;
n = n-1;
}
  return res;
}
```

Wiley Encyclopedia of Computer Science and Engineering, edited by Benjamin Wah.

Functional programming implements iteration through recursion. The following functional program is a direct translation of the C program. The iteration is performed by the recursively defined function `facWork`; that is, the function is defined in terms of itself; it calls itself.

```
factorial :: Integer -> Integer
factorial n = facWork n 1

facWork ::Integer -> Integer ->Integer
facWork n res =
 if n > 1 then facWork (n-1) (n*res)
else res
```

In every recursive call of the function `facWork`, the two parameter variables have new values; the value of a variable is never changed, but every function call yields a new instance of the parameter variables. The parameter variable `res` is called an *accumulator*. In general, a parameter variable is an *accumulator* if in recursive calls it accumulates the result of the function, which is finally returned by the last, nonrecursive call.

The evaluation of an expression can be described as a sequence of reduction steps. In each step, a function is replaced by its defining expression, or a primitive function is evaluated:

```
factorial 3
= facWork 3 1
= if 3 > 1 then facWork (3—1) (3*1) else 1
= if True then facWork (3—1) (3*1) else 1
= facWork (3—1) (1 * 3)
= facWork 2 (1*3)
= facWork 2 3
= if 2 > 1 then facWork (2—1) (2*3) else 3
= if True then facWork (2—1) (2*3) else 3
= facWork (2—1) (2*3)
= facWork 1 (2*3)
= facWork 1 6
= if 1 > 1 then facWork (1—1) (1*6) else 6
= if False then facWork (1—1) (1*6) else 6
= 6
```

For the same expression, several different reduction step sequences exist, as the section below titled "Nonstrict Versus Strict Semantics" will show, but all finite sequences yield the same value.

In imperative languages, programmers usually avoid recursion because of its high performance costs, including its use of space on the run-time stack. The lack of side effects enables compilers for functional programming languages to translate simple recursion schemes as present in `facWork` easily into code that is as efficient as that obtained from the imperative loop.

The following is a simpler definition of the `factorial` function that does not use an accumulator. It resembles the common mathematical definition of the function.

```
factorial :: Integer -> Integer
factorial n =
 if n > 1 then factorial (n-1)
 * n else 1
```

Data Structures

Functional programming languages directly support unbounded data structures such as lists and trees. Such data structures are first-class citizens; that is, they are used like built-in primitive types, such as characters and numbers. They do not require any explicit memory allocation or indirect construction via pointers or references.

A list is a sequence of elements, for example, `[4,2,2,5]`. It can have any length. In statically typed languages, all elements must be of the same type; `[Integer]` is the type of lists whose elements are of type `Integer`. The function `enumFromTo` constructs a list:

```
enumFromTo :: Integer -> Integer ->[Integer]
enumFromTo m n = if m == n then [ ]
          else m : else m : (enumFromTo (m+1) n)
```

The value of the expression `enumFromTo 3 7` is the list of integers `[3,4,5,6,7]`. In the function definition, `[]` denotes the empty list and : is an operator that combines a value and a list to a list, such that the value is the first element. `[]` and : are constants and operators for lists similar to `0` and + for numbers.

The list is the most frequently used data structure in functional programming. Lists can be used to represent many other data structures, such as sets and bags. Lists are also frequently used as intermediate data structures that replace and modularize iterative processes:

```
factorial :: Integer -> Integer
factorial n = product (enumFromTo 1 n)
```

This definition expresses clearly that the factorial of n is the product of the numbers from 1 to n. Both functions product and `enumFromTo` have clear meanings and are likely to be reused elsewhere. Some optimizing compilers will remove the intermediate list and produce the same efficient code as for the imperative loop (cf. Chapter 7.6 of Ref. 2).

In several functional languages, the definition of some tree-structured data type looks similar to a context-free grammar:

```
data Expr = Val Bool
          | And Expr Expr
          | Or Expr Expr
```

`Expr` is a new type whose values are built from the *data constructors* `Val`, `And`, and `Or`. Hence, `And (Val True) (Or (Val False) (Val True))` is an expression that constructs the syntax tree of `True && (False || True)`.

Many functional languages also provide *pattern matching* as a mechanism that simultaneously tests the top data constructor of a value and gives access to its components:

```
eval :: Expr -> Bool
eval (Val b) = b
```

```
eval (And e1 e2) = eval e1 && eval e2
eval (Or e1 e2) = eval e1 || eval e2
```

So, the value of `eval (And (Val True) (Or (Val False) (Val True)))` is `True`. Data structures as first-class citizens and pattern matching together enable clear and succinct definitions of complex algorithms on unbounded data structures, for example, standard algorithms on balanced ordered trees (10). Functional programming encourages the programmer to view a large data structure as a single value instead of concentrating on its many constituent parts.

Besides data constructors and variables, patterns may also contain values of built-in types, such as numbers. If the patterns of several defining equations overlap, then the first matching equation defines the function result. In the next definition of the function `factorial`, the first equation defines the result value for the argument 0, and the second equation defines it for all other arguments.

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Because functional languages are often used for symbolic computations, many functional languages provide a large set of numeric types, including arbitrary-size integers, rationals, and complex numbers, and aim for precise and efficient implementations of basic numeric operations.

Higher-Order Functions

In functional programming, functions are first-class citizens. The value of an expression may be a function, and functions can be passed as arguments to other functions and returned as results from functions. A function that takes another function as argument or that returns a function is called a *higher-order function*.

A standard higher-order function is the function map:

```
map :: (a -> b) -> [a] -> [b]
```

It takes a function and a list as arguments, applies the function to all list elements, and returns the list of the results. For example, the value of `map even [1,2,3,4]` is `[False,True,False,True]`. The type variables `a` and `b` in the type of `map` will be discussed in the section below titled "Types".

Although the function `map` is usually defined recursively and, hence, iteratively consumes its argument list and produces its result list, the programmer can view a higher-order function such as `map` as processing a whole large data structure in a single step.

Many traditional imperative programming languages, such as C, also allow passing functions as arguments and results, hence the definition of a higher-order function such as `map`, but the definition of new functions through composing existing functions is rather limited. For example, such limitations make it impossible to define the function `scale` by composing the existing functions `map` and `*` (or a multiplication function; in C, operators are different from functions). Here, the function `scale` shall take a list of numbers (e.g., prices) and multiply all of them by the same given factor; for example, `scale 1.25 [2,0,4]` yields `[2.5,0,5]`. The difficulty in defining the function `scale` in terms of `map` and `*` lies in the fact that the function to be mapped over the list is not `*`, which requires two arguments, but a function that takes only one argument and multiplies it with the given factor. A functional language provides at least one of two ways to solve this task:

```
scale :: Float -> [Float] -> [Float]
scale factor prices = map scaleOne prices
  where
  scaleOne :: Float ->Float
  scaleOne p = factor * p
```

The preceding, first solution *defines a function* `scaleOne` *locally* so that the local definition can use the variable `factor` because it is in scope. The second definition below builds the function that is to be mapped over the list by *partially applying* the function `*` to one argument. So `(*) factor` is an expression that denotes the function that takes one argument and multiplies it with `factor`.

```
scale :: Float -> [Float] -> [Float]
scale factor prices = map ((*) factor) prices
```

For functional programming, not just the presence of higher-order functions but also the language support for composing arbitrary functions to generate an unbounded number of functions at execution time are essential.

Many higher-order functions are included in the definitions of functional languages or their standard libraries. For lists, besides the function `map`, the function `foldr` (or `reduce`) is the most commonly used higher-order function. This function combines all list elements with a given binary function and uses a given constant for processing the empty list:

```
product ::[Integer] -> Integer
product xs = foldr (*) 1 xs
```

So,

```
product [3,2,4]
= 3 * (2 * (4 * 1))
= 24
```

Although our examples only show higher-order functions that take simple (first-order) functions as arguments, functions that take functions as arguments that take functions as arguments and so forth are used frequently (11).

Point-Free Programming

Below is a shorter definition of the function `product` as the functional value of the expression `foldr (*) 1`:

```
product ::[Integer] -> Integer
product = foldr (*) 1
```

The factorial function can also be defined using the function composition operator (.):

```
factorial :: Integer -> Integer
factorial = product . (enumFromTo 1)
```

Expressions or *function* definitions without argument variables are called *point-free*. Often they are shorter and simplify program transformation, but they can be harder to understand and to modify. Most functional programs are written in a mixture of point-free and "point-full" style.

Embedded Domain-Specific Languages

Identifying the right abstractions is a key component of designing a program. In functional programming, the reuse of existing, mostly higher-order functions or, especially for new data structures and problem domains, the identification of new higher-order functions is central. Several functional programming languages such as Lisp (4) and Scheme (5,6) also provide an elaborate *macro mechanism* for extending the language by new constructs. Thus, the design of a solution to a problem and especially the design of a general library for a problem domain often leads to the design of an *embedded domain-specific language*. This language is a collection of higher-order functions or new language constructs that together substantially simplify programming solutions in a given domain. The embedded language hides domain-specific algorithms and data structures behind an easy-to-use interface.

As a simple example for an embedded domain-specific language, the following interface outlines an embedding of propositional logic. The implementation of the type of propositional formulas `Formula`, is hidden.

```
true :: Formula
false :: Formula
variable :: String -> Formula
(/\) :: Formula -> Formula -> Formula
(\/) :: Formula -> Formula -> Formula
negate :: Formula -> Formula


satisfiable :: Formula -> Bool
tautology :: Formula -> Bool
```

Logical formulas can be constructed and checked for whether they are satisfiable or even tautologies. For example, `tautology (negate (variable "a") \/ variable "a")` yields `True`.

More complex are the widely studied and used embedded domain-specific languages for describing parsers through context-free grammars. The following example is a simple parser for fully bracketed Boolean expressions, using Swierstra's parser interface (12).

```
pExpr :: Parser Expr
pExpr = Val True <$  pStr "True"
  <|> Val False <$  pStr "False"
  <|> And       <$  pSym '(' <*> pExpr <*
  pStr"&&"<*> pExpr <* pSym ')'
  <|> Or        <$  pSym '(' <*> pExpr <*
  pStr"||"<*> pExpr <* pSym ')'

pStr :: String ->Parser ()
pStr[] = pSucceed ()
pStr (x:xs) = () <$ pSym x <* pStr xs
```

The definition of the parser `pExpr` looks like a context-free grammar. The operator `<|>` combines alternative parsers. The operators `<*>` and `<*` concatenate two parsers. Only `<$` does not relate to a construct of a context-free grammar; it turns a function for constructing the desired result into a parser and concatenates it with another parser. All operators associate to the left. For the operators `<*` and `<$`, only the left argument yields the parser's result, whereas for the operator `<*>`, both arguments contribute to the parser's result. The function `pStr` constructs a parser that accepts the given string and returns a required but superfluous empty tuple `()`. Parsing `"(True&&(False||True))"` with the parser `pExpr` will yield `And (Val True) (Or (Val False) (Val True))`.

Simple implementations use backtracking and define the parser type as a function that maps the input string to a list of possible parse results and remaining input:

```
data Parser a = P (String ->[(a,String)])
```

Here, a is a type variable as will be discussed in the section titled "Types". More efficient parser implementations use more sophisticated parser representations (13).

An embedding of the logical language Prolog in Haskell is described in Ref. 14. Pretty printing, graphics, simulation, and music composition are more domain examples (15).

No clear boundary can be found between an abstract data type and an embedded domain-specific language, but the latter gives the programmer the illusion of a new programming language for a specific domain. An embedded domain-specific language strives to hide some features of the host language, give domain-specific compiler errors, and enable domain-specific debugging.

Program Algebra

Because in pure functional programming, evaluation of an expression only determines its value but does not cause any side effects, functional programs have a rich algebra. For example, the law

$$\texttt{map}\ f.\ \texttt{map}\ g = \texttt{map}\ (f.g)$$

holds for any expressions f and g (whose values must be functions). If the functions f and g modified a common variable, then this equation would be unlikely to hold. Hence, in imperative programming languages, hardly any nontrivial semantic equalities hold. The term *referentially transparent* is often used synonymously with *side effect free* in functional programming. By definition, a language is referentially transparent if a subexpression can be replaced by an equal subexpression without changing the meaning of the whole expression or program. This definition, however, is just the definition of what it means for two subexpressions to be equal. Relevant and useful is that many equations with arbitrary unknown subexpressions hold; that is, the equational algebra is rich.

Standard higher-order functions such as `map` and `foldr` come with well-known laws. In a new problem domain, functional programmers strive for identifying functions with rich algebraic properties. Such functions are highly versatile and thus reusable.

Program algebra has already been used to describe the evaluation of `factorial 3` as a sequence of single reduction steps. So, evaluation can be described within the language, without any reference to, for example, the memory locations of a computer.

Functional programming cultivates a school of program development by algebraic derivation. The programmer starts with a set of desired properties expressed as equations or a highly inefficient implementation. These equations are then transformed step by step using equational reasoning until an efficient implementation is obtained. Only using program algebra guarantees that specification and implementation are semantically equal. Reaching an efficient implementation is not automatic but requires the ingenuity of the programmer. However, many strategies and heuristics for deriving programs have been developed (2,16).

Compilers for functional programming languages use program algebra for optimizations. For example, standard evaluation of `map (f . g)` is more efficient than the evaluation of `map f . map g` because the later produces an intermediate list. A compiler optimization, hence, may replace the latter by the former expression. Compilers usually perform many very simple transformations, but altogether they may change a program substantially (17). In contrast, optimizing compilers for imperative languages require sophisticated analyses to detect side effects that invalidate most optimizations.

Algebraic laws also prove to be useful for testing. A law such as

```
reverse (reverse (xs)) = xs
```

is a partial specification of the function `reverse`, which returns a list with all elements in reverse order. A correct implementation of `reverse` should meet this property for all finite lists `xs`. A simple tool automatically can test the law for many lists (18).

In a language without side effects, program components can be tested separately, and test cases can be set up more easily. Equational properties are both documentation and expressive test cases. They encourage the programmer to identify functions that meet nontrivial equational properties.

TYPES

Functional programming languages support both avoidance and localization of program faults by having *strong type systems*. The type systems guarantee that all execution errors, such as the application of a function to unsuitable arguments, are trapped before they occur. There are both functional languages with dynamic type systems (e.g., Lisp and Erlang) that provide flexibility by performing all type checks at run time and that often do not include a fixed syntax for types, and functional languages with static type systems (e.g., ML and Haskell).

Most static type systems of functional languages are based on the Hindley–Milner type system (19). This type system is flexible in that it allows *(parametrically) polymorphic* functions and data structures. That is, a function may take arguments of arbitrary type if its definition does not depend on that type. For example, the function `reverse` that takes a list and returns a list of all elements in reversed order has the type `[a] ->[a]`. Here, `a` is a type variable that represents an arbitrary type. The function `reverse` can be applied to a list with elements of any type. The reoccurrence of `a` in the type of the result states clearly that the elements of the result list are of the same arbitrary type as the elements of the argument list. A more complex type is that of `map` given before in the section titled "Higher-Order Functions":

```
map :: (a -> b) ->[a] ->[b]
```

The repeated occurrences of the type variables `a` and `b` clearly state that (*1*) the type of the argument list elements has to agree with the argument type of the function, and (*2*) the result type of the function has to agree with the type of the result list elements, but these two types can be different, as in the case of `map even [1,2,3,4]`, which evaluates to `[False,True,False,True]`.

Another feature of the Hindley–Milner type system is that types can be inferred automatically. Hence, type declarations such as

```
reverse ::[a] ->[a]
```

are optional, and many programmers only add them when program development has stabilized after an initial phase of rapid prototyping.

Several functional languages extend the Hindley–Milner type system substantially. ML (7) is renowned for its expressive *module system*. Types describe the interfaces of modules, how modules can be combined, and how abstract data types can be defined. The Haskell (1,2) *class system* uses classes, which are similar to types and remind one of the object-oriented paradigm, to describe interfaces of smaller pieces of code (e.g., a few functions that express an ordering) than modules and to enable their combination with little syntactic overhead. OCaml (20,8) has a *subtyping* relationship between its class types to enable object-oriented programming. Clean (3) annotates standard types with *uniqueness* information to express that certain values are used only in a single-threaded way, which enables a form of purely function input/output (see the section below titled “Necessary Side Effects") and compilation to more efficient code. Clean also supports *generic*, also called *polytypic,* programming. Polytypic language features enable the programmer to define a function by induction on the structure of values of types. Like a parametrically polymorphic function, such a function works on all types, but its definition depends on the structure of the values. Example applications are pretty printers, parsers, and equality functions.

More extensions of type systems in many other directions are a major topic of research. A type describes a property of an expression or a piece of code. Types can describe nonstandard properties, such as how much time

or space evaluation of the expression will cost (mainly for applications in embedded systems), or whether evaluation of the expression may create an exception or cause a side effect. Type inference is then a form of *automatic program analysis* (21). *Dependent type systems* allow types to be parameterized not just by other types but also by values. For example, such a type system can express that a vector addition function takes two vectors of any size and returns a vector, but the sizes of all these vectors have to be the same. Dependent type systems realize the Curry–Howard isomorphism that states that types are logical formulas and the typed expressions are proofs of the formulas. Thus, a program and proofs of its properties can be written within the same advanced programming language. The type systems of current functional programming languages already allow a limited amount of dependent typing, usually based on nontrivial encodings of values in types. The development of functional languages with dependent type systems that are easy to use is a long-standing research topic (22). In general, most research on type systems concentrates on functional programming languages with their simple and well-defined semantics (23).

NONSTRICT VERSUS STRICT SEMANTICS

A function is *strict* if its result is undefined (error or evaluation does not terminate) whenever any of its arguments is undefined. Like imperative languages, many functional programming languages (e.g., Lisp, ML, and Erlang) have a strict semantics, that is, allow only the definition of strict functions. This situation follows directly from their *eager evaluation order*: In a function application, first the arguments are fully evaluated, and then the function is applied to the argument values.

In contrast, languages with a nonstrict semantics (e.g., Haskell and Clean) allow the definition of nonstrict functions and infinite data structures. A function `enumFrom` yields an infinitely long list and is used in the definition of the infinite list of factorial numbers, `factorials`. The `factorial` function then just takes the n-th element of this list (list index numbers start at `0`):

```
enumFrom :: Integer -> [Integer]
enumFrom n = n : enumFrom (n+1)

factorials ::[Integer]
factorials =
  1 : (zipWith (*) factorials
(enumFrom 1))

factorial :: Integer -> Integer
factorial n = genericIndex factorials n
```

The expression `zipWith (*)` takes two lists and combines their elements pairwise by multiplication (`*`). The idea underlying the recursive definition of the list of factorial numbers is expressed by the following table:

```
factorials} =   1 1 2  6   24   120 ...

                * * *  *   *    *
enumFrom1 =   1 2 3  4   5    6   ...

              || || ||  ||   ||   ||
factorials = 1 1 2  6  24 120 720  ...
```

Although semantically several infinite lists are defined, the evaluation of a factorial number is finite:

```
factorial 3
= genericIndex factorials 3
= genericIndex (1:(zipWith (*) (1:...)
    (enumFrom 1))) 3
= genericIndex (1:(zipWith (*) (1:...)
    (1: (enumFrom (1+1))))) 3
= genericIndex (1:(1*1): (zipWith (*) ...
    (enumFrom (1+1)))) 3
= genericIndex (1:(1*1): (zipWith (*)
    ((1*1): ...) ((1+1): (enumFrom
    ((1+1)+1))))) 3
= genericIndex (1: (1*1): ((1*1)*(1+1)):
    (zipWith (*) ... (enumFrom
    ((1+1)+1)))) 3
= genericIndex (1: 1: (1*(1+1)): (zipWith
    (*) ... (enumFrom ((1+1)+1)))) 3
= genericIndex (1: 1: (1*2): (zipWith (*)
    ... (enumFrom (2+1)))) 3
= genericIndex (1: 1: 2: (zipWith (*) ...
    (enumFrom (2+1)))) 3
= genericIndex (1: 1: 2: (zipWith (*)
    (2: ...) ((2+1): (enumFrom
    ((2+1)+1))))) 3
= genericIndex (1: 1: 2: (2*(2+1)): (zipWith
    (*) (...) (enumFrom ((2+1)+1)))) 3
= 2*(2+1)
= 2*3
= 6
```

Implementations of nonstrict functional languages usually use *lazy evaluation*, which passes arguments in unevaluated form to functions but avoids duplicated evaluation through sharing of unevaluated expressions. Nonstrict semantics enables modular solutions to many programming problems (24). Recursive definitions of constants are important for defining parsers (12). The programmer can also define new control structures like `if then else`, which is nonstrict in the two last arguments in all programming languages:

```
isPositive:: Integer -> a -> a -> a
isPositive n yes no =
  if n > 0 then yes else
no

factorial:: Integer -> Integer
factorial n =
  isPositive n (n * factorial
(n−1)) 1
```

Nonstrict languages have a simpler program algebra than strict languages because in the latter, many equations do not hold for expressions with undefined values. However, the time and especially the space behavior of nonstrict

functional programs is much harder to predict than that of strict ones.

NECESSARY SIDE EFFECTS

Functional programming aims to minimize or eliminate side effects. However, an executing program usually does not just transform an input into an output but also has to communicate with users, other processes, the file system, and so forth; in short, it has to perform I/O. Many functional languages, such as Lisp and ML, use simple side-effecting functions for I/O, but some languages use I/O models that perform the side effects required by I/O such that the program algebra remains unaffected, as if no side effects were present.

Nonstrict languages, such as Miranda (25) and early versions of Haskell, use the *lazy stream model*. The program transforms a list of input events into a list of output events. The nonstrict semantics ensures that part of the output list can already be produced after processing only part of the input list, and hence, earlier output events can influence later input events (26). Using this I/O model strengthens the intuition for nonstrict semantics. All other I/O models work for both strict and nonstrict languages.

The *uniqueness model* is used in Clean (3). This model is based on the idea that there is a special token, the world value, that every I/O function requires as an argument and returns as part of its result. The world value can be used only in a single-threaded way; that is, the world value cannot be duplicated nor an old value be used twice. A uniqueness type system ensures the single-threaded use of the world value (cf. the section entitled "Types," above).

Early versions of Haskell also used the *continuation model* (26). The idea of the continuation model is that a function that performs I/O never returns; instead, it takes an additional argument, the continuation function, and after performing the side-effecting I/O operation, calls this continuation function and passes any result of the I/O operation as argument to the continuation function. In general, a program written such that functions do not return but instead pass their results to other functions is said to be in *continuation passing style*. Continuation passing style enables the programmer to control the evaluation order tightly (27) and thus ensure the required sequential execution of I/O operations.

Later versions of Haskell use the *monad model*. The monad model is similar to the continuation model but allows easier composition of I/O computations. Every I/O operation returns an element of the abstract monad type, and monadic values can only be composed by a sequence operator, which thus enforces the sequential order of I/O operations. The following Haskell I/O operation reads characters from standard input until the new line character is read and returns the list of read characters.

```
readLine:: IO[Char]
readLine = do
  c <— getChar
  if c == '\n'
     then return[]
  else do
    rest <— readLine
    return (c:rest)
```

`IO` is the monad, and the type of `readline` is `IO[Char]` because this operation returns a list of characters, just as the type of `getChar` is `IO Char`. The `do` construct is syntactic sugar that makes monadic programs look very similar to imperative ones. The keyword `do` is followed by a number of I/O operations, all of monadic `IO` type, which are executed sequentially. The <- notation gives access to normal values computed by monadic operations.

In general, monads are useful for embedding various operations that must be executed in a specific order. For example, they can be used to add mutable references to a pure functional language or to implement backtracking as used by many parser libraries (28).

The algebra for monadic expressions is more complex and, for arbitrary monads, more limited compared with nonmonadic expressions; by definition, the compositionality of monadic code is restricted.

Programmers use side effects also for purposes other than I/O. Many well-known algorithms rely on the modification of data structures to achieve their efficiency, especially those that transform graph-structured data. In principle, a mutable memory can be simulated by a balanced tree in a functional program with a logarithmic loss of time complexity. Nicholas Pippenger showed (29) that certain problems can be solved in linear time in an imperative language but can be solved in a strict, eagerly evaluated functional programming language only with a logarithmic slowdown. However, this theoretical argument does not apply to nonstrict languages that use lazy evaluation (30).

In practice, many efficient, purely functional algorithms exist (10). Arrays are processed most efficiently by operations that construct whole new arrays from existing ones instead of emphasizing individual elements (26). Finally, mutable references can be embedded into pure functional languages using monads, but most functional programmers prefer to use the expressibility of functional programming to develop new algorithms or tackle problems that are too complex for imperative languages.

IMPLEMENTATION TECHNIQUES

In contrast to imperative languages, functional languages are not based on standard computer architecture, and hence many different implementation models have been explored. Backus (31) suggested that functional languages could inspire new computer architectures, and during the 1970s, specially designed computers for running Lisp, Lisp machines, were popular. However, Backus also noted that only when functional languages "have proved their superiority over conventional languages will we have the economic basis to develop the new kind of computer that can best implement them." The speed of mass-produced processors grew far faster than that of specially designed hardware. Backus still saw the efficient and correct implementation of the lambda calculus as a major obstacle (31),

and graph reduction machines that reduced combinators (top-level functions) were devised to circumvent this problem. Nowadays, the compilation of functional programs into code on standard hardware that is comparable in speed to that of imperative programs is well understood, and although many variations are found, compilation is surprisingly similar to the compilation of imperative languages (32,33). Two main additional issues pertain to implementation techniques. First, a functional language allocates most data objects on the heap and has to use a garbage collector (34) because the lifetimes of data objects are not determined by the program structure. Second, to implement functions as first-class citizens, they have to be represented as *closures*. The standard representation of a closure is a pointer to the function code plus an environment, a data structure that maps variables to their values.

Additionally, implementations of nonstrict functional languages have to pass unevaluated expressions as arguments; these are represented as *thunks* that can be implemented identically to closures. Strictness analysis is used to reduce the number of unnecessary and costly thunks. Compiler optimizations mostly work on the level of the functional language and use the rich program algebra for program transformations (cf. the section above titled "Program Algebra"). The implementation model of a functional language is usually described by an abstract machine. The first and best known, but not the most simple or most efficient, is Peter Landin's SECD machine.

Pure functional languages lend themselves naturally to parallel evaluation. In principle, all arguments of a function could be evaluated in parallel. Hence, especially the 1980s saw substantial research into parallel implementations of functional languages. The main problem was that the implicit parallelism of functional languages is of fine granularity, and hence process creation and communication overheads are high.

THEORETICAL FOUNDATIONS

The main theoretical foundation of functional programming is the *lambda calculus* (23,35), which was developed by Alonzo Church in the 1930s not as a programming language but as a small mathematical calculus for describing the operational behavior of mathematical functions. The syntax of the lambda calculus consists of only three different kinds of expressions: variables, applications, and abstractions. An abstraction, written $\lambda x.e$, where x is a variable and e an expression, denotes a function with parameter variable x and body e. An application $(e_1\, e_2)$ applies a function e_1 to its argument e_2.

To evaluate expressions, only a single reduction rule called β-*reduction* is needed:

$$(\lambda x.e_1)e_2 \rightarrow e_1[e_2/x]$$

All occurrences of the parameter variable x in the function body e_1 are replaced by the argument e_2. β-reduction can be applied anywhere in an expression. Evaluation is a sequence of β-reduction steps:

$$(\lambda x.x)((\lambda y.y)(\lambda z.z)) \rightarrow (\lambda x.x)(\lambda z.z) \rightarrow (\lambda z.z)$$

Usually, an expression can be evaluated in many ways. An alternative to the previous one is

$$(\lambda x.x)((\lambda y.y)(\lambda z.z)) \rightarrow (\lambda y.y)(\lambda z.z) \rightarrow (\lambda z.z)$$

An important property of the lambda calculus is its *confluence*, which ensures that all evaluation sequences for an expression that terminate will yield the same final value.

The restriction of the lambda calculus to functions with one argument is not a limitation because the result of an application can be another function that then is applied to its argument. For example, in $((e_1\, e_2)\, e_3)$ the expression e_1 can be viewed as a function that takes two arguments, namely e_2 and e_3. The function e_1 is said to be *curried*. We usually write $(e_1\, e_2\, e_3)$. Many functional languages have adopted this notation for function application instead of the more familiar $e_1(e_2, e_3)$.

The power of the lambda calculus stems from the fact that functions can be applied to themselves. This fact allows functions that are usually defined recursively to be defined in a nonrecursive form. Hence, the lambda calculus is Turing-complete even without having a recursion construct. However, nearly all functional programming languages include explicit recursion for convenience. Many typed variants of the lambda calculus are found; without an additional recursion construct, most of them are strongly normalizing; that is, the evaluation of any expressions terminates, and thus they are not Turing-complete but can still be very expressive.

All data structures such as natural numbers, Booleans, and lists can be represented in the lambda calculus via their *Church-encodings*. For most practical purposes, these Church-encodings are too inefficient, but they prove that built-in data structures are not strictly required.

The lambda calculus forms the core of most functional programming languages and, thus, also provides the foundation for their semantics and implementation. The theory of *term rewriting systems* (36) provides a similar foundation. A term rewriting system is basically a functional program, but most of the theory of term rewriting systems does not cover higher-order functions.

Besides the operational semantics given by sequences of reduction steps, functional programs also have useful *denotational semantics* (37). First, denotational semantics associates every type with a set, the set of values of this type. For example, the set of type `Int` is the set of integers, and the set of type `Int -> Int` is a set of functions that take an integer and return an integer. Second, each expression is interpreted as an element of the set of values of its type. This interpretation is defined by a simple induction on the structure of the expression. For example, from knowing that the semantic value of a function identifier `add` is the addition function and knowing the values of the expressions `3` and `4`, we conclude that the expression `add 3 4` has the value 7, without any reduction sequence expanding the definition of `add`. Thus, denotational semantics is compositional and also less dependent on the syntax of the programming language than operational semantics. Denotational semantics proved particularly useful as foundation for numerous static program analysis methods (21).

COMBINATIONS WITH OTHER PROGRAMMING PARADIGMS

Most functional programming languages are impure and, thus, include an *imperative programming* language. Input and output are realized by side effects, and the values of variables can be modified. In some languages, such as ML (7) and Caml (8), mutable variables have different types from nonmutable ones. So, these languages enable and encourage the functional programming style but do not require it.

The *object-oriented programming* paradigm comprises several features that can be combined with a functional programming language in various ways. OCaml (20) and some Lisp dialects provide features as they are familiar to object-oriented programmers. Most functional programming languages achieve the modularity and code-reuse aimed for by object-oriented programming by related, but different, means, often through their flexible module and type systems.

Both functional and *logic programming* languages are *declarative*; that is, they abstract from many implementation details and concentrate on describing the problem. Several functional logic research languages combine both paradigms; Mercury (38) augments logic programming by functional programming, and Curry (39) augments a Haskell-like functional language by logic programming features.

Several extensions of standard functional programming languages with constructs for *concurrent programming* are found. Erlang (9) was designed from the start as a concurrent functional programming language in which any nontrivial program defines numerous processes. Processes do not share data but communicate via message passing. Process creation and communication are the only side effects in the language. Limitation of side effects simplifies the language and enables an Erlang system to provide code updating at run time.

A BRIEF HISTORY OF FUNCTIONAL LANGUAGES

Lisp (4) was the first functional programming language and is one of the oldest programming languages still in use. John McCarthy started developing Lisp in the late 1950s as an algebraic list-processing language for artificial intelligence research. A central feature of Lisp is the construction of dynamic lists from simple cons cells and the use of a garbage collector for reclaiming unused cells. Lisp provides many higher-order functions over lists, and more higher-order functions can be defined easily. Lisp is not a pure functional language: List structures can be modified, and already defining a function is implemented through side effects. Lisp has a very simple prefix syntax that represents both program and data alike. Thus, Lisp programs require numerous parentheses, but it is very simple to extend the language within itself. The development of Lisp and Lisp applications thrive on this easy extensibility. Although Lisp adopted the lambda abstraction for defining functions from the lambda calculus, otherwise, it was originally little influenced by the lambda calculus. Hence, most Lisp dialects still use *dynamic binding*, in which the scope of local identifiers is based on the call structure of the program, instead of *static binding*, in which local identifiers are bound by their enclosing definitions in the program text. *Scheme* is a small modern dialect of Lisp (with static binding) that has become particularly popular in teaching functional and imperative programming concepts (5,6). The following definition in Scheme demonstrates its simple syntax:

```
(define (factorial n)
   (if (> n 1) (*
(factorial (- n 1)) n) 1))
```

One of the most cited papers on functional programming is John Backus' 1977 Turing Award lecture (31). Backus' arguments have particular authority because he received the Turing Award for his pioneering work on developing Fortran and significant influence on Algol. Backus criticizes existing imperative programming languages as being too tightly bound to the conventional von Neumann machine architecture. The assignment statement directly reflects memory access in the von Neumann architecture. Thus, programming is dominated by a word-at-a-time sequential programming style instead of thinking in terms of larger conceptual units. Furthermore, Backus attacks the "division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs." Backus argues that an algebra of programs is far more useful than the logics designed for reasoning about imperative programs. Backus identifies two main problems of functional languages that were found at that time. First, the substitution operation required for implementing the lambda calculus was difficult to implement efficiently; therefore, his language FP is completely point-free and defines new functions by combining existing ones. Second, functional languages are not history sensitive, and they cannot easily store data beyond the run time of a single program; hence, he defines a traditional state transition system on top of his functional FP system.

The language *ML* was originally developed at the end of the 1970s as command language for a theorem prover but soon developed into a popular stand-alone language. Its main new feature is its advanced static type system, based on the Hindley–Milner type system, and an expressive system for defining and combining modules. ML is not pure because its I/O system is based on side effects, but modification of variables is limited to the use of separate reference types. Besides Standard ML (7), the Caml dialect (8) is used widely. The following definition of the factorial function in Standard ML leaves it to the system to infer the function type:

```
fun factorial x =
  if x = 0 then 1
  else x * factorial (x−1)
```

In the 1970s and 1980s, David Turner developed a series of influential functional languages, SASL (40), KRC (41),

and *Miranda* (25), which in contrast to previous languages, have purely nonstrict semantics. Similar to ML, a program is a system of equations but the syntax is even closer to common mathematical notation. Miranda also uses the Hindley–Milner type system. Miranda is purely functional; the I/O system uses lazily evaluated lists. In the late 1980s and early 1990s, Miranda was widely used in university teaching.

In the late 1970s and the 1980s, many similar nonstrict, purely functional languages appeared, and hence at the end of the 1980s, a committee was formed to define a common language: *Haskell*. Its main novelties are the class system that extends its Hindley–Milner type system and, in later revisions, the use of a monad to support purely functional I/O. Haskell is widely used in teaching, and its application outside the academic community is growing (1,2). The purely functional language Clean (3) is similar to Haskell but has a uniqueness type system to enable purely functional I/O and generation of efficient code.

In the late 1980s, Ericsson started the development of *Erlang*, a concurrent functional programming language (9). Erlang was designed to support the development of distributed, fault-tolerant, soft-real-time systems.

The proceedings of the three ACM SIGPLAN conferences on the history of programming languages (HOPL I,II,III) give historic details about many functional programming languages.

SUMMARY

Functional programs are built from simple but expressive expressions. User-defined unbound data structures substantially simplify most symbolic applications. Features such as higher-order functions and the lack of side effects support writing and composing reusable program components. Program components cannot interact via hidden side effects but only via their visible interface. Thus, all aspects of program development from rapid prototyping, testing, and debugging to program derivation and verification are simplified. Ideas developed within functional programming, such as garbage collection and several type system features, have been adopted by many other programming languages. Several modern compilers produce efficient code. The abstraction from machine details allows short and elegant formulation of algorithms. The regular Programming Pearls in the *Journal of Functional Programming* (42) provide numerous small examples. Writing solutions that are both elegant and efficient for applications that perform substantial I/O or transform graph-structured data is still a challenge.

BIBLIOGRAPHY

1. S. Thompson, *Haskell: The Craft of Functional Programming*, Boston, MA: Addison-Wesley, 1999.
2. R. Bird, *Introduction to Functional Programming Using Haskell*, Upper Saddle River, NJ: Prentice Hall, 1998.
3. R. Plasmeijer and M. van Eekelen, The Concurrent Clean language report, version 2.0, Available: http://www.cs.kun.nl/~~clean, 2001.
4. G. L. Steele, *Common Lisp the Language*, 2nd ed., Newton, MA: Digital Press, 1990.
5. H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, Cambridge, MA: MIT Press, 1996.
6. M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs: an Introduction to Programming and Computing*, Cambridge, MA: MIT Press, 2001.
7. L. C. Paulson, *ML for the Working Programmer*, Cambridge, UK: Cambridge University Press, 1996.
8. G. Cousineau and M. Mauny, *The Functional Approach to Programming*, Cambridge, UK: Cambridge University Press, 1998.

J. Armstrong, *Programming Erlang, Software for a Concurrent World*, Raleigh, NC: Pragmatic Bookshelf, 2007.

C. Okasaki, *Purely Functional Data Structures*, Cambridge, UK: Cambridge University Press, 1998.

C. Okasaki, Functional pearl: Even higher-order functions for parsing or Why would anyone ever want to use a sixth-order function? *J. Functional Prog.*, **8** (2): 195–199, 1998.

D. Swierstra, Combinator parsers: From toys to tools, in G. Hutton, (ed.), *Haskell Workshop 2000*, Electronic Notes in Theoretical Computer Science, Vol. 41, New York: Elsevier Science Publishers, 2001.

J. Hughes and D. Swierstra, Polish parsers, step by step, *ICFP '03: Proc. of the eighth ACM SIGPLAN international conference on Functional programming*, New York, 2003. pp. 239–248.

J. Spivey and S. Seres, Embedding Prolog in Haskell, *Proc. Haskell'99,* Technical Report UU CS 1999-28, Department of Computer Science, University of Utrecht., 1999.

P. Hudak, Building domain-specific embedded languages, *ACM Comput. Survey*, **28** (4es), 1996.

J. Hughes, The design of a pretty-printing library, in J. Jeuring and E. Meijer, (ed.), *Advanced Functional Programming*, LNCS 925, New York: Springer Verlag, 1995.

S. L. Peyton Jones and A. L. M. Santos, A transformation-based optimiser for Haskell, *Sci. Comput. Prog.*, **32** (1–3): 3–47, 1998.

K. Claessen and J. Hughes, QuickCheck: A lightweight tool for random testing of Haskell programs, *ACM SIGPLAN Notices*, **35** (9): 268–279, 2000.

J. C. Mitchell, *Foundations for Programming Languages*, Cambridge, MA: The MIT Press, 1996.

X. Leroy, et al., The Objective Caml system, documentation and user's manual, Available: http://caml.inria.fr, 2007.

F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*, New York: Springer, 1999.

T. Altenkirch, C. McBride, and J. McKinna, Why dependent types matter, Available: http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf, 2005.

B. C. Pierce, *Types and Programming Languages*, Cambridge, MA: The MIT Press, 2002.

J. Hughes, Why functional programming matters, *Compu. J.*, **32** (2): 98–107, 1989.

D. A. Turner, An overview of Miranda, *SIGPLAN Notices*, **21** (12): 158–166, 1986.

P. Hudak, Conception, evolution, and application of functional programming languages, *ACM Comput. Surv.*, **21** (3): 359–411, 1989.

J. C. Reynolds, Definitional interpreters for higher-order programming languages, *Higher Order Symbol. Comput.*, **11** (4): 363–397, 1998.

P. Wadler, Monads for functional programming, in M. Broy, (ed.), *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*, New York: Springer-Verlag, 1993.

N. Pippenger, Pure versus impure Lisp, *ACM Trans. Prog. Lang. Syst.*, **19** (2): 223–238, 1997.

R. Bird, G. Jones, and O. DeMoor, More haste, less speed: Lazy versus eager evaluation, *J. Funct. Program.*, **7** (5): 541–547, 1997.

J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Commun. ACM*, **21** (8): 613–641, 1978.

D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen, *Modern Compiler Design*, New York: 2000.

S. L. P. Jones, *The Implementation of Functional Programming Languages*, Upper Saddle River, NJ: Prentice Hall, 1987.

R. Jones, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, New York: Wiley 1999.

J. R. Hindley and J. P. Seldin, *Lambda-Calculus and Combinators: An Introduction*, Cambridge, UK: Cambridge University Press, 2008.

F. Baader and T. Nipkow, *Term Rewriting and All That*, Cambridge, UK: Cambridge University Press, 1998.

G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*, Cambridge, MA: MIT Press, 1993.

F. Henderson, T. Conway, Z. Somogyi, et al., The Mercury project, Available: http://www.mercury.csse.unimelb.edu.au/.

M. Hanus, et. al., Curry: A truly integrated functional logic language, Available: http://www.informatik.uni-kiel.de/~curry.

D. A. Turner, SASL language manual, Technical Report, St. Andrews University, Department of Computational Science Technical Report, 1976.

D. A. Turner, The semantic elegance of applicative languages, *FPCA '81: Proc. of the 1981 conference on Functional programming languages and computer architecture*, New York, 1981, pp. 85–92.

J. Func. Prog., Cambridge, UK: Cambridge University Press.

OLAF CHITIL
University of Kent
Canterbury, United Kingdom