FUNCTIONAL AND LOGIC PROGRAMMING

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, A-4040 Linz, Austria, Wolfgang.Schreiner@risc.uni-linz.ac.at.

Keywords: declarative programming, mathematical functions, Haskell, ML, referential transparency, term reduction, strict evaluation, lazy evaluation, higher-order functions, program skeletons, program transformation, reasoning, polymorphism, functors, generic programming, parallel execution, logic formulas, Horn clauses, automated theorem proving, Prolog, SLD-resolution, unification, AND/OR tree, constraint solving, constraint logic programming, functional logic programming, natural language processing, databases, expert systems, computer algebra.

Contents

1. Introduction
2. Functional Programming
2.1 Mathematical Foundations
2.2 Programming Model
2.3 Evaluation Strategies
2.4 Higher Order Functions
2.5 Parallel Execution
2.6 Type Systems
2.7 Implementation Issues
3. Logic Programming
3.1 Logical Foundations
3.2. Programming Model
3.3 Inference Strategy
3.4 Extra-Logical Features
3.5 Parallel Execution
4. Refinement and Convergence
4.1 Constraint Logic Programming
4.2 Functional Logic Programming
5. Impacts on Computer Science
Glossary
Bibliography

Summary

Most programming languages are models of the underlying machine, which has the advantage of a rather direct translation of a program statement to a sequence of machine instructions. Some languages, however, are based on models that are derived from mathematical theories, which has the advantages of a more concise description of a program and of a more natural form of reasoning and transformation. In functional languages, this basis is the concept of a mathematical function which maps a given argument values to some result value. A program is a mathematical term which is evaluated to a normal form by replacing each occurrence of a function symbol by its

corresponding definition. On the other hand, logic languages are built upon the concept of a predicate that relates certain values to each other. A program is a logic formula in which an inference mechanism finds substitutions for the variables such that the formula becomes true. The efficient execution of functional and logic languages has made great progress during the last two decades; further developments have extended the expressiveness of the programming models (constraint logic programming) and unified them in a common framework (functional logic programming). Powerful type systems have been developed which allow to write in a safe way programs that may be applied to a variety of application domains (generic programming). The ideas exemplified by functional and logic languages have essentially influenced the design of other programming languages.

1. Introduction

Functional and logic programming languages are also called *declarative languages*; programs in these languages are said to describe (declaratively) *what* to do and not (operationally) *how* to do it. While this statement may be questioned, declarative languages have certainly their basis in formal models with properties that make programs particularly amenable to precise reasoning and correctness-preserving transformations. This is in contrast to imperative languages which are based on models of the underlying machine; programs written in imperative languages can be thus more directly compiled to efficient machine code, but reasoning and program transformations are comparatively difficult (see *Imperative Programming*).

Declarative programming languages have been developed since the 1970s, but their roots can be traced to the 1930s when mathematicians and logicians began to study the theory of computability. Concise formal calculi were developed in which (supposedly) any calculation can be expressed that a machine can perform and that thus should already suffice as linguistic frameworks for computer programming (Church's thesis, see *Models of Computation*). For instance, although the λ-calculus developed by Alonzo Church consists of just three kinds of expressions and a simple reduction rule, it is believed to be capable of performing every possible computation; a subset of the programming language LISP developed by McCarthy in the late 1950s can be considered as an implementation of this calculus. Nevertheless, it was at that time not believed that a practically useable programming language could be in its entirety based on a simple formal model.

However, in the late 1960s and early 1970s several ideas arose how to write programs in a purely declarative style and still have them executed with reasonable efficiency. Depending on the formal theory, two major schools of thought have subsequently emerged: the *functional programming* community has focused on the concept of the mathematical function as a value-mapping entity; since such a function is typically defined by a set of equations, this yields a style of "programming with recursive equations" (the title of an early paper). The task of the programmer is construct a wanted result value from the given argument values by some basic constructs with a simple mathematical interpretation; reasoning about program correctness thus is immediately reduced to conventional mathematical reasoning and program transformations can be performed like arithmetic calculations. While early functional languages were comparatively slow, especially in the second half of the 1980s compilation techniques were developed that nowadays allow very efficient execution. Functional languages have also considerably contributed to the theory of type systems by concepts such as polymorphic functions (functions applicable to arguments of different types) and functors (parameterized program modules that take modules as arguments and return modules as results) which yielded the idea of generic programming (nowadays en vogue in object-oriented languages). The myriad of functional languages developed in the 1980s has today crystalized into two major representatives: ML (for Meta-Language) developed at the University of Edinburgh in the course of a project in automated theorem proving and Haskell (named after the logician Haskell Curry) which was developed by a joint initiative of various research groups in Europe and the US.

Logic programming is an outcome of research in automated theorem proving. In 1965, Robinson published the resolution method as an efficient decision procedure for logic formulas written in a subset of first-order predicate logic called Horn clause logic. While not every logic formula can be expressed in this language, it is sufficiently rich to serve as the basis of a rule-based programming style where the task of the programmer is to construct a relation between values: those given by the user are considered as input from which the system computes the other ones as output. In the early 1970s, Kowalski elaborated the theory of logic programming with Colmerauer producing the first implementation of the programming language Prolog (Programming in Logic). The language became an instant success and triggered the world-wide interest of many institutions that developed various dialects and (also commercial) implementations. A major break-through was achieved in the second half of the 1980s when the Japanese Research Organization ICOT chose logic programming as the basis of their "5th Generation Computers" project. While this initiative failed to produce a new basis for computer architecture, it helped to widely disseminate expertise in logic programming. In the 1990s, research in logic programming focused on making the basic principle more expressive by including constraints (equations and inequalities) over arithmetic domains, which gave rise to *constraint logic programming*. The resolution mechanism was extended by methods for "constraint solving" which brought mathematics in closer contact to logic programming.

From the very beginning, both functional and logic programming languages have been considered for parallel programming, i.e., the solution of a problem by concurrent execution of multiple tasks on multiprocessors and computer networks. In contrast to imperative languages, declarative languages do not impose a predefined order of execution steps such that a variety of concurrent evaluation/inference strategies can be devised. While efficient automatic parallelization is still out of reach, parallelization annotations in "para-functional" languages and "Guarded Horn Clause Languages" allow with comparatively little effort to write parallel programs in a declarative style.

In the 1990s, new developments have started to blur the distinction between functional programming and logic programming leading to *functional-logic programming:* here a logic formula also has a return value or, vice versa, a function call is also a goal which has to be satisfied by constructing term substitutions for the variables. A new mechanism called "narrowing" unifies the execution strategies "term reduction" for functional programming and “resolution” for logic programming and thus enhances the expressiveness of the declarative style of programming. While research currently focuses more on the theoretical aspects, the next decade will certainly see also further progress on more efficient compilation strategies for this kind of languages. While numerous applications have been developed in declarative languages, their main impact on computer science is an indirect one: the ideas and techniques elaborated in functional and logic programming have found their way to conventional languages, especially to object-oriented languages such as C++ and Java and to the languages used in computer algebra systems such as Mathematica.

2. Functional Programming

2.1 Mathematical Foundations

A mathematical function. $f : A \rightarrow B$ is a mapping *f* from a set of objects *A* called the *domain* to a target set *B* called the *range* such that for every element *a* of *A* the term *f(a)* (the *application of f to a*) uniquely denotes an object of *B*. Typically *f* is defined by an equation $f(x) = T$ where *T* is a term in which only *x* occurs as a free variable; the result of *f(a)* is determined by evaluating *T* after the formal parameter *x* has been replaced by actual argument *a*. For instance,

$$\text{square} : Z \rightarrow Z$$
$$\text{square}(x) = x^* x$$

defines a function *square* on the set Z of integer numbers such that the application *square*(2) denotes the result 2*2 = 4. A function may also take multiple parameters, e.g. defining

$$\text{squarediff} : Z \times Z \rightarrow Z$$
$$\text{squarediff}(a, b) = (a+b)^* (a-b)$$

yields *squarediff*(3, 2)=(3 + 2)*(3 - 2)=5*1 = 5. We may construct the defining term also hierarchically with the help of *local definitions*:

$$\text{squarediff} : Z \times Z \rightarrow Z$$
$$\text{squarediff}(a, b) = c^* d \text{ where}$$
$$c = a + b$$
$$d = a - b$$

A function may be defined by multiple terms guarded by *conditions* on the parameters, e.g.

$$\text{abs} : Z \rightarrow Z$$
$$\text{abs}(x) = \begin{cases} -x & \text{if} \quad x < 0 \\ x & \text{otherwise} \end{cases}$$

defines the absolute value of an integer number. If the condition can be expressed by a syntactic *pattern* of the arguments, then a function definition may consist of multiple equations, e.g.

$$\mathrm{tail} : \mathrm{List}(\mathrm{A}) \to \mathrm{List}(\mathrm{A})$$
$$\mathrm{tail}([\ ]) = [\ ]$$
$$\mathrm{tail}(\mathrm{x} : \mathrm{xs}) = \mathrm{xs}$$

defines a function *tail* on lists of elements from any set *A*: when *tail* is applied to an empty list, it returns the empty list; when it is applied to a list with first element *x* and rest list *xs*, it returns *xs*.

A function may also refer to itself *recursively* on the right hand side of the defining equation, e.g.

$$\mathrm{sum} : \mathrm{List}(\mathrm{Z}) \to \mathrm{Z}$$
$$\mathrm{sum}([\ ]) = 0$$
$$\mathrm{sum}(\mathrm{x} : \mathrm{xs}) = \mathrm{x} + \mathrm{sum}(\mathrm{xs})$$

defines a function *sum* that when applied to a list of integer numbers returns the sum of the list elements, e.g. *sum*([1, 2, 3]) = 1 + *sum*([2, 3]) = 1 + (2 + *sum*([3])) = 1+ (2 + (3 + sum([]))) = 1 + (2 + (3 + 0))) = 6.

Given functions $\mathrm{f} : \mathrm{A} \to \mathrm{B}$ and $\mathrm{g} : \mathrm{B} \to \mathrm{C}$ we can create the *composition* $\mathrm{f\ og} : \mathrm{A} \to \mathrm{C}$ such that for every *a* in *A* we have $(\mathrm{f\ og})(\mathrm{a}) = \mathrm{g}(\mathrm{f}(\mathrm{a}))$. For instance, defining $\mathrm{listsquare} = \mathrm{sum\ o\ square}$, we get $\mathrm{listsquare}[1,2,3] = 36$. Thus we may construct in a modular way from simple functions more complex ones.

2.2 Programming Model

In a functional programming language like *Haskell* or *ML*, the definition of a program function closely resembles a mathematical function definition. For instance, we can define in Haskell

```
square :: Int -> Int
square x = x*x
```

such that the term (square 2) evaluates to 4. Likewise we can define

```
squarediff :: Int -> Int -> Int
squarediff a b = c*d where
c = a + b
d = a - b
```

```
abs :: Int -> Int
abs x | x < 0      = -x
      | otherwise =  x

tail :: [a] -> [a]
tail [ ]      = [ ]
tail (x : xs) = xs
sum :: [Int] -> Int
sum [ ]      = 0
sum (x : xs) = 1 + (sum xs)
```

such that e.g. the term (sum [1, 2, 3]) evaluates to 6. It is illustrative to compare above definition of sum to a corresponding definition in an imperative programming language (see. *Imperative Programming*):

```
fun sum(a : Array[Int], n : Int) : Int
var s,  i : Int
s := 0
for i := 1 to n do
s := s + a[i]
retrun s
end
```

In an imperative language, a program is a sequence of commands (or *statements*) that operate on a hidden state, namely the computer store holding the current values of the program variables (s, i). Program execution proceeds in a sequence of *assignments* which start in some initial state and iteratively modify this state by updating the variable values until some final state is reached which denotes the result of the computation $\left(s = \sum_{i-1}^{n} a[i],\ i = n + 1\right)$.

All imperative languages (and also the object-oriented ones, (see *Object-oriented Programming*) reflect this state-oriented view which has its basis in the von Neumann model of computation which itself represents an abstraction of the underlying hardware (see *Processors*).

On the contrary, in a functional language a program is a mathematical term (or *expression*) and program execution proceeds in a sequence of *reduction steps*: in each step, we select a subterm denoting a function application (the *redex*) and replace it by the defining term after having substituted the formal parameters by the actual arguments. The reduction process terminates when we yield a term that does not contain any more redex (a *normal form*).

A term may have more than one potential redex, e.g. in squarediff(sum(a), sum(b))each of the underlined function symbols denotes the head of a term that may be selected for reduction. However, no matter, which selection strategy we apply, *if* a reduction sequence yields a normal form, this normal form is uniquely determined. This fundamental feature is a consequence of the *Church-Rosser Theorem* in λ-calculus which forms the theoretical basis of functional programming (see *Models of Computation*).

As a consequence, functional programming languages have the following characteristics that distinguish them from imperative or object-oriented languages:

- *Referential transparency.* A functional program is an expression that can be considered as a mathematical term: its result is a unique value that only depends on the results of its subexpressions. Consequently, if the program is executed multiple times, it always returns the same result. Conversely, the only effect of executing a functional program is returning a result: e.g., in program *f(g(a),h(b))* the execution of *g(a)* cannot have a side-effect that affects the result of *h(b).* Furthermore, if we define a name *x* = *T* in a functional program, then any occurrence of *x* in the scope of the definition may be replaced by *T* without changing the overall result.

All this is *not* true for imperative languages: here the definition of a program function may refer to global data that may be destructively updated between calls; actually the function itself may update these data such that two subsequent identical applications give different results.

- *No destructive assignments.* A (purely) functional programming language does not have an assignment statement that updates the value of a variable. A "variable" in a functional language is actually a *constant* i.e. a name that is bound by a definition to a value; this binding is *immutable*, i.e., it cannot be changed in the scope of the definition (*single assignment property*).

This restriction has a major consequence on the functional style of programming: while it is in imperative programming common to have a global data structure that is iteratively updated in the course of computation, in functional programming always a new version of the structure has to be constructed (nevertheless the new structure may share components with the old one or the system may automatically find that the old structure can be safely updated in place).

-
-
-

Bibliography

Apt, K. R. (1996). *From Logic Programming to Prolog.* Englewood Cliffs: Prentice Hall. [An introduction to logic programming in general and Prolog in special.]

Bird, R. (1998). *Introduction to Functional Programming Using Haskell.* 2nd edition, Englewood Cliffs: Prentice Hall. [This book gives a good overview on the functional school of programming using the Haskell language.]

Bowen, J.P. (2000). *Logic Programming.* Web page http://archive.comlab.ox.ac.uk/logic-prog.html.[The best starting point for a Web search, see also the Usenet news group comp.lang.prolog.]

Clocksin, W.F. and Mellish, C.S. (1994) *Programming in Prolog.* 4th edition, Berlin, Springer.

[This is the classical introduction to Prolog programming.]

Field A. J. and Harrison P. G. (1988) *Functional Programming.* Wokingham: Addison-Wesley. [This is a comprehensive book which presents functional programming in general, the implementation of functional languages, and the formal manipulation of functional programs for optimization.]

Hammond, K. and Michelson, G. (1999) *Research Directions in Parallel Functional Programming.* Berlin: Springer. [A state of the art survey on all aspects of parallel functional programming.]

Hartel, P. and Muller, H. (1997). *Functional C.* Wokingham: Addison-Wesley. [This shows that functional programming is not bound to a functional language by using the imperative language C.]

Hutton, G. (2000) *Frequently Asked Questions for comp.lang.functional*. Usenet news group. Also on http://www.cs.nott.ac.uk/~gmh/faq.html [This is the best starting point for a Web search of online materials on functional programming.]

Marriot, K. and Stuckey, P.J. (1999) *Programming with Constraints: An Introduction.* Boston, MIT Press. [An introduction to constraint programming, in particular constraint logic programming.]

Masahiko, S. and Toyama, Y. (1998) *Functional and Logic Programming.* World Scientific Publisher. [Discusses issues on functional and logic programming, and the integration of the two.]

Paulson, L.C (1996). *ML for the Working Programmer*. Cambridge: Cambridge University Press. [This is a practical introduction to the functional language ML.]

Peyton Jones, S. (1987). *The Implementation of Functional Programming Languages.* Englewood Cliffs: Prentice Hall. [This is still the most comprehensive overview on the implementation of lazy programming languages.]

Shapiro, E. (1988) *Concurrent Prolog: Collected Papers*. Volumes I+II. Boston, MIT Press. [This is a collection of research papers on parallel logic programming with a GHC language.]

Sterling, L. and Shapiro E. (1994) *The Art of Prolog: Advanced Programming Techniques.* 2nd edition, Boston: MIT Press. [A comprehensive description from logical foundations to advanced programming techniques; less suitable for a quick overview but essential for serious programming.]