In *ACM Computing Surveys*, Vol. 28, No. 2, 1996, pp. 306–308

Integration of Functional and Logic Programming

Michael Hanus   Herbert Kuchen
RWTH Aachen*

During the last decade, many proposals have been made to combine the most important declarative programming paradigms, namely functional and logic programming (see [2] for a survey). Functional logic languages offer features from functional programming (nested expressions, higher-order functions, lazy evaluation) and logic programming (logical variables, partial data structures, built-in search). They subsume purely functional languages as well as pure Prolog. An important advantage of an integrated language is that it can help to bring the functional and the logic programming communities together, enabling them to share their developments and avoid a duplication of work on implementation techniques, program analysis, program transformation, graphical user interfaces, and many other tools. Moreover, this integration should lead to an increased acceptance of declarative programming in general.

Compared to purely functional languages, functional logic languages mainly provide a *built-in search* mechanism which is capable of handling *partial information.* An expression containing logic variables represents an arbitrarily large set of values. If the basic structure of such an expression allows to infer that there are no solutions no matter how the logic variables are bound, then a whole subspace of the search space can be ignored (namely the subspace providing instantiations of the logic variables). The purely functional approach to searching by (lazily) producing a *list of successes* [5] will consider all values of this subspace one by one, since an expression in a purely functional language represents only one value (regardless of the evaluation strategy!). Note that a simulation of expressions containing logic variables by λ-abstractions, e.g., a representation of $(cons\ X\ Y)$ by $\lambda X.\lambda Y.(cons\ X\ Y)$ does not always work, since this would fix the order in which the logic variables have to be bound. Moreover, in constraint logic programming, an important application domain of logic programming, logic variables are seldom bound to a distinct value but typically constrained to smaller and smaller subsets of the entire domain. This incremental constraint solving is hard to describe by functional programs but a natural feature of functional logic languages.

Compared to purely logic languages, functional logic languages provide more efficient evaluation mechanisms due to the (*deterministic*!) reduction of functional expressions. Thus, *impure features* of Prolog to restrict the search space, like the *cut* operator, can be avoided. Moreover, a simulation of higher-order functions by the (impure) `call` predicate is no longer necessary, since such functions are directly available. Further, *lazy evaluation* allows an elegant style of programming, including the treatment of *infinite data structures.*

Unfortunately, functional logic languages did not have the desired success up to now. Reasons for this are the lack of a "standard" functional logic language and the fact that existing functional logic languages and their implementations are mainly experimental systems but not yet mature for

*Informatik II, RWTH Aachen, D-52056 Aachen, Germany, {hanus,herbert}@informatik.rwth-aachen.de

real applications. The development of a standard language is complicated by the fact that there is no agreement on the operational semantics. There are mainly two approaches, namely residuation and narrowing.

Residuation is based on the idea to delay function calls until they are ready for deterministic evaluation. Since the residuation principle evaluates function calls by deterministic reduction steps, nondeterministic search must be explicitly encoded by predicates or disjunctions. The residuation principle is a reasonable integration of the functional and the logic paradigm, since it combines the deterministic reduction of functions with partial data structures (logical variables). Moreover, it allows concurrent computation with synchronization on logical variables. Unfortunately, it is incomplete, since it is unable to compute solutions if arguments of functions are not sufficiently instantiated during the computation.

Narrowing is a combination of unification for parameter passing and reduction as evaluation mechanism. It is *complete* in the sense of functional programming (normal forms are computed if they exist) as well as logic programming (solutions are computed if they exist). In order to get an efficient implementation, sophisticated *narrowing strategies* are required. The strategy *needed narrowing* [1] interleaves the evaluation of *demanded* arguments and an indexing mechanism to select applicable rules. It is optimal w.r.t. the length of derivations and the number of computed solutions. This clearly shows the advantages of integrating functions into logic programs: by transferring results from functional programming to logic programming, we obtain better and, for particular classes of programs, optimal evaluation strategies without loosing the search facilities. Defining functions is not a burden to the programmer, since most predicates of (logic) application programs are functions. Moreover, the knowledge about functional dependencies can avoid useless computations (of arguments which are not needed) and increase the number of deterministic evaluation steps.

A future functional logic language could offer a combination of residuation and narrowing: if the user does not restrict the applicability of a function, a complete strategy will be used; however, the user can add annotations which allow to residuate for some arguments and/or to specify another evaluation order.

The second problem for the currently low acceptance of functional logic programming, namely the toy character of existing systems, has to be solved by including features which are required for programming serious applications. This includes a *polymorphic type system*, a module system, and purely declarative I/O. One possible choice for the latter is to use *monadic I/O* [6] like in Haskell. This forces the main computation thread to be a *sequence* of monad operations. Thus, in a functional logic setting, nondeterministic search has to be encapsulated such that unlimited backtracking is excluded. One possibility for *encapsulating search* has been proposed in Oz [4]. The user gets explicit access to the search space as a data structure and can explore it step by step in the desired direction (e.g., depth-first). Frequently used search strategies like depth-first and breadth-first can be described by higher-order functions and used by simply calling these functions with the problem as parameter. Encapsulated search provides also control over the explored search space.

Another desirable feature is a full integration of higher-order functions (including some sort of higher-order unification rather than their limited use as in purely functional languages). "Higher-order narrowing" is especially convenient for handling applications where the notion of scope is important, like formulae and programs (see [3] for references to such applications). Moreover, *implications* and *explicit quantifiers* (as in λ-Prolog [3]) can be used to avoid (many) occurrences of the impure Prolog-features *assert* and *retract*. While evaluating the succedent e of an implication ($clause \Rightarrow e$), the clause in the antecedent is added to the program.

Logic languages have the advantage that they can easily be extended by constraint solving and thus allow to use domain specific, efficient search strategies. Since functional logic languages also allow an easy integration of constraint solvers, important application areas of logic programming are also covered by such integrated languages.

Existing implementations of functional logic languages show that the integration need not cause a (serious) performance penalty [2]. If logic variables do not occur in a program, the evaluation is identical to functional languages. Similarly, it is identical to logic languages if functions are not used, i.e., there is no overhead due to the presence of the additional features. On the contrary, if logic variables are used in combination with functions, the deterministic behavior of functions is used to increase the efficiency w.r.t. purely logic languages and the handling of partial information enables more efficient search than in purely functional languages.

References

[1] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.

[2] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[3] G. Nadathur and D. Miller. An overview of λProlog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 810–827. MIT Press, 1988.

[4] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Current Trends in Computer Science*. Springer LNCS 1000, 1995.

[5] P. Wadler. How to replace failure by a list of successes. In *Functional Programming and Computer Architecture*. Springer LNCS 201, 1985.

[6] P. Wadler. The essence of functional programming. In *Symposium on Principles of Programming Languages*, pages 1 – 14. ACM, 1992.