Search Strategies for Functional Logic Programming

Michael Hanus Björn Peemöller Fabian Reck

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh|bjp|fre}@informatik.uni-kiel.de

Abstract: In this paper we discuss our practical experiences with the use of different search strategies in functional logic programs. In particular, we show that complete strategies, like breadth-first search or iterative deepening search, are a viable alternative to incomplete strategies, like depth-first search, that have been favored in the past for logic programming languages.

1 Introduction

Functional logic languages combine the most important features of functional and logic programming in a single language (see [AH10, Han07] for recent surveys). In particular, they provide higher-order functions and demand-driven evaluation from functional programming together with logic programming features like non-deterministic search and computing with partial information (logic variables). This combination led to new design patterns [AH02, AH11] and better abstractions for application programming, e.g., as shown for programming with databases [BHM08, Fis05], GUI programming [Han00], web programming [Han01, Han06, HK10], or string parsing [CLF99]. Moreover, it is a good basis to teach the ideas of functional and logic programming, or declarative programming in general, with a single computation model and programming language [Han97]. The operational principles of functional logic languages have also been used for other computation tasks, like inverse computations [AGK06], partial evaluation [AFV98], or generation of test cases [FK07, RNL08].

An important feature of logic programming languages is non-deterministic search. In Prolog, which is still the standard language for logic programming, non-deterministic search is implemented via backtracking, which corresponds to a depth-first search traversal through the SLD proof tree [Llo87]. Due to this feature of Prolog, the idea of logic programming is often reduced to the combination of unification and backtracking, as shown by approaches to add logic programming features to existing functional languages (e.g., [CL00, Hin01]). This limited "backtracking" view of logic programming is also harmful to beginners, e.g., when newbies define their family relationships using a Prolog rule like

```
sibling(X,Y) :- sibling(Y,X).
```

In such cases, one has to explain from the beginning the pitfalls of backtracking which harms the understanding of declarative programming. From a declarative point of view,

a logic program defines a set of rules and a logic programming system tries to find a solution to a query w.r.t. this set of rules. In order to abstract from operational details, the search strategy has to be complete. Due to these considerations, the functional logic language Curry [He06] does not fix a particular search strategy so that different Curry implementations can support different (or also several) search strategies. Moreover, Curry implementations also support encapsulated search where non-deterministic computations are represented in a data structure so that different search strategies can be implemented as tree traversals [BHH04, HS98, Lux99].

In this paper, we present our practical results with different search strategies implemented in a new Curry system called KiCS2 [BHPR11]. KiCS2 compiles Curry programs into Haskell programs where non-deterministic values and computations are represented as tree structures so that flexible search strategies can be supported. Although the incomplete depth-first search strategy is the most efficient one (provided that it is able to find a result value), we show that complete strategies, like breadth-first search or iterative deepening search, are a reasonable alternative that does not force the programmer to consider the applied search strategy in his program.

In the next section, we briefly recall some principles of functional logic programming and the programming language Curry that are necessary to understand the remaining part of the paper. The encapsulation of search and the implementation of different search strategies are discussed in Section 3. These strategies are evaluated with a number of benchmarks in Section 4 before we conclude in Section 5.

2 Functional Logic Programming and Curry

Integrated functional logic programming languages combine features from functional programming and logic programming. Recent surveys are available in [AH10, Han07]. Curry [He06] is a functional logic language which extends lazy functional programming as to be found in Haskell [PJ03] and additionally supports logic programming features. Another functional logic language based on similar principles is $\mathcal{TOY}$ [LFSH99]. However, $\mathcal{TOY}$ does not offer flexible search strategies by a concept of encapsulating search (although it provides a concept of nested computation spaces in order to deal with failures in functional logic programming [LFSH04, SH06]). Therefore, we use Curry throughout this paper, although the concepts presented here could be also integrated in other functional logic languages.

A Curry program consists of the definition of data types and operations on these types. Since the syntax of Curry is close to Haskell, variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$"). In addition, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. Note that in a functional logic language operations might yield more than one result on the same input due to the logic programming features. For instance, Curry contains a *choice* operation defined by:

```
x ? _ = x
_ ? y = y
```

The choice operation can be used to define other non-deterministic operations like

```
coin = 0 ? 1
```

Thus, the expression "`coin`" has two values: `0` and `1`. If expressions have more than one value, one wants to select intended values according to some constraints, typically in conditions of program rules. A *rule* has the form

$f\ t_1 \ldots t_n$ `|` c `=` e

where the (optional) condition c is a *constraint*, i.e., an expression of the built-in type `Success`. For instance, the trivial constraint `success` is a value of type `Success` that denotes the always satisfiable constraint. Thus, we say that a constraint c is *satisfied* if it can be evaluated to `success`. An *equational constraint* e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifiable values.

As a simple example, consider the following Curry program which defines a polymorphic data type for lists and operations to compute the concatenation of lists and the last element of a list:[1]

```
data List a = [] | a : List a    -- [a] denotes "List a"

-- "++" is a right-associative infix operator
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last :: [a] → a
last xs | (ys ++ [z]) =:= xs = z
  where ys, z free
```

Logic programming is supported by admitting function calls with free variables (e.g., `(ys++[z])` in the rule defining `last`) and constraints in the condition of a defining rule. In contrast to Prolog, free variables need to be declared explicitly to make their scopes clear (e.g., "`where ys,z free`" in the example). A conditional rule is applicable if its condition is satisfiable. Thus, the rule defining `last` states in its condition that `z` is the last element of a given list `xs` if there exists a list `ys` such that the concatenation of `ys` and the one-element list `[z]` is equal to the given list `xs`.

Curry also offers standard features of functional languages, like modules or monadic I/O which is identical to Haskell's I/O concept [Wad97]. Thus, "`IO` α" denotes the type of an I/O action that returns values of type α.

The operational semantics of Curry is based on an optimal evaluation strategy [AEH00] which is a conservative extension of lazy functional programming and (concurrent) logic programming. A big-step and a small-step operational semantics of Curry can be found

[1] Note that lists are a built-in data type with a more convenient syntax, e.g., one can write `[x,y,z]` instead of `x:y:z:[]` and `[a]` instead of the list type "`List a`".

in [AHH+05]. Curry's semantics is sound in the sense of logic programming, i.e., each computed result is correct and for each correct result there is a more general computed one [AEH00]. In order to achieve completeness, one has to take *all* possible non-deterministic derivation paths into account. In contrast to Prolog, which fixes a (potentially incomplete) depth-first search strategy to find solutions, Curry does not fix a particular search strategy. Actually, descriptions of the model-theoretic [GMHGLFRA99] or operational [AHH+05] semantics of functional logic languages do not take a search strategy into account. Thus, different Curry implementations can support various search strategies. For instance, the Curry implementation PAKCS [HAB+10], which compiles Curry programs into Prolog programs, supports only a depth-first search strategy. MCC [Lux99] compiles Curry programs into C programs and uses also a depth-first search strategy to find the solutions of a given top-level goal. In addition, MCC offers the encapsulation of search (see below) so that other search strategies, like a complete breadth-first search strategy, can be used inside a Curry program. The Curry implementation KiCS [BH07, BH09], which compiles Curry programs into Haskell programs, offers depth-first and breadth-first search strategies for top-level goals as well as the encapsulation of search with user-definable strategies. In this paper we consider the Curry implementation KiCS2 [BHPR11], which is based on similar ideas than KiCS but uses a different compilation model avoiding side effects to enable better optimizations for target programs. KiCS2 also offers different search strategies for top-level goals and the encapsulation of search with user-definable strategies, which is described next.

3 Search Strategies

As mentioned above, Curry does not enforce a particular search strategy. A Curry implementation can provide various search strategies to find solutions or values for a given constraint or expression. The most advanced system in this respect is KiCS2 [BHPR11], which supports the evaluation of top-level expressions with depth-first search, breadth-first search, iterative deepening search, or parallel search strategies. Curry supports this flexibility since all operations with side effects are collected in monadic I/O operations [Wad97]. As a consequence of this computation model, all non-deterministic computations between I/O operations must be encapsulated since one can not apply two alternative I/O operations to an existing "world" and non-deterministically proceed with two alternative worlds ("one can not duplicate the world"). Therefore, Curry offers the encapsulation of search by representing non-deterministic computations in a data structure so that the computation of different solutions (one solution or all solutions) can conceptually be implemented as traversals on this data structure.

An early approach [HS98, Lux99] to encapsulating search in Curry is based on a primitive search operator

```
try :: (a → Success)  →  [a → Success]
```

that takes a constraint abstraction, e.g., `(\x->x=:=coin)`, as input, evaluates it until the first non-deterministic step occurs, and returns the result: the empty list in case of fail-

ure, a list with a single element in case of success, or a list with at least two elements representing a non-deterministic choice. For instance, `try (\x->x=:=coin)` evaluates to `[\x->x=:=0,\x->x=:=1]`. Based on this primitive, one can define various search strategies to explore the search space and return its solutions. [Lux99] shows an implementation of this primitive.

Although typical search operators of Prolog, like `findall`, `once`, or negation-as-failure, can be implemented using `try`, it became also clear that the combination of encapsulated search and demand-driven evaluation and sharing causes further complications [BHH04]. For instance, in an expression like

```
let y = coin in try (\x → x =:= y)
```

it is not obvious whether the non-determinism caused by the evaluation of `coin` (introduced outside but demanded inside the search operator) should be encapsulated or not. Hence, the result of this expression might depend on the evaluation order. For instance, if `coin` is evaluated before the `try` expression, it results in two computations where `y` is bound to `0` in one computation and to `1` in the other computation. Hence, `try` does not encapsulate the non-determinism of `coin` (this is the semantics of `try` implemented in [Lux99]). However, if `coin` is evaluated inside the capsule of `try` (because it is not demanded before), then the non-determinism of `coin` is encapsulated. These and more peculiarities are discussed in [BHH04]. Furthermore, the order of the solutions might depend on the textual order of program rules or the evaluation time (e.g., in parallel implementations). Hence, it is difficult to define a search operator as a pure function.

Due to these considerations, [BHH04] contains a proposal for another primitive search operator:

```
getSearchTree :: a  →  IO (SearchTree a)
```

It takes an expression and delivers a search tree representing the search space when evaluating the input:

```
data SearchTree a = Value a
                  | Fail
                  | Or (SearchTree a) (SearchTree a)
```

`(Value v)` and `Fail` represent a single value or a failure (i.e., no value), respectively, and `(Or t1 t2)` represents a choice (i.e., a non-deterministic value) between two search trees `t1` and `t2`. Since `getSearchTree` is an I/O action, its result (in particular, the order of subtrees) depends on the current environment, e.g., time of evaluation. To avoid the complications w.r.t. shared variables, `getSearchTree` implements a *strong encapsulation view*, i.e., conceptually, the argument of `getSearchTree` is cloned before the evaluation starts in order to cut any sharing with the environment. Furthermore, the structure of the search tree is computed lazily so that an expression with infinitely many values does not cause the nontermination of the search operator if one is interested in only one solution.

This primitive has been implemented for the first time in KiCS [BH07, BH09] and it is also provided in KiCS2 [BHPR11] considered in this paper. With this primitive, the programmer can define its own search strategy as `SearchTree` traversals in order to collect

all non-deterministic values into a list structure. For instance, a depth-first search strategy can be easily defined as follows:

```
allValuesDFS :: SearchTree a → [a]
allValuesDFS Fail      = []
allValuesDFS (Value x) = [x]
allValuesDFS (Or  x y) = allValuesDFS x ++ allValuesDFS y
```

Note that the lazy evaluation of traversal operations like `allValuesDFS` has the advantage that the search strategy is decoupled from the control. For instance, we can define the following operation to print a single value of a non-deterministic expression:

```
printFirstValueDFS x =
  getSearchTree x >>= print . head . allValuesDFS
```

Thus, `(printFirstValueDFS` exp`)` can print some value even if the non-deterministic expression exp has infinitely many values. This is in contrast to Prolog's constructs for controlling search where different operators are necessary to compute one or all solutions.

It is well known that depth-first search lacks completeness, i.e., it might not be able to compute all existing values. For instance, consider the following operation that non-deterministically returns all increasing numbers from a given number `n`:

```
f n = f (n+1) ? n
```

Although `0` is a value of `(f 0)`, the evaluation of `(printFirstValueDFS (f 0))` does not terminate (provided that the primitive `getSearchTree` explores the non-determinism of "`?`" in left-to-right order). This problem can be avoided with complete search strategies, like breadth-first search strategy, which can be defined on `SearchTree` structures as follows:

```
allValuesBFS :: SearchTree a → [a]
allValuesBFS t = collect [t]

collect []     = []
collect (t:ts) = values (t:ts) ++ collect (children (t:ts))

values []             = []
values (Fail    : ts) = values ts
values (Value x : ts) = x : values ts
values (Or  _ _ : ts) = values ts

children []             = []
children (Fail    : ts) = children ts
children (Value _ : ts) = children ts
children (Or  x y : ts) = x : y : children ts
```

The operation `values` extracts the values in each level of the tree and the operation `children` extracts all direct successors of a level in order to recursively collect their values.

Using `allValuesBFS`, we can print a value of the expression `(f 0)` by

```
getSearchTree (f 0) >>= print . head . allValuesBFS
```

In order to abstract from the details of the evaluation order, it would be preferable to use complete search strategies. However, complete strategies are often neglected due to performance reasons. For example, the breadth-first search strategy stores all child nodes of a tree level in a list to be explored later, which, as the search space potentially doubles on each level, may lead to an exponentially growing memory usage.

Another complete search strategy with a superior memory behavior is iterative-deepening search. Basically, it is a depth-first search strategy with a depth-bound which is incremented in each iteration. Thus, we compute in each iteration a list of values together with some information whether we have aborted (due to the depth-bound) the computation of further possible values. For this purpose, we define list structures that can also end with an `Abort`:

```
data AbortList a = Nil | Cons a (AbortList a) | Abort
```

and define the concatenation on such lists:

```
concA :: AbortList a  → AbortList a  → AbortList a
concA Abort       Abort       = Abort
concA Abort       Nil         = Abort
concA Abort       (Cons x xs) = Cons x (concA Abort xs)
concA Nil         ys          = ys
concA (Cons x xs) ys          = Cons x (concA xs ys)
```

Now we define an operation to collect values in a search tree within some level bounds (to avoid the repeated collection of values found in each iteration):

```
collectInBounds :: Int  → Int  → SearchTree a  → AbortList a
collectInBounds oldbound newbound st = collectLevel newbound st
 where
  collectLevel _ Fail      = Nil
  collectLevel d (Value x) = if d <= newbound - oldbound
                             then Cons x Nil
                             else Nil
  collectLevel d (Or  x y) =
    if d > 0
    then concA (collectLevel (d-1) x) (collectLevel (d-1) y)
    else Abort
```

Now, the entire search strategy consists of repeated calls to `collectInBounds` as long as the result list is aborted. In order to experiment with different parameters, the initial depth bound and the method to increase the depth bound in each iteration are passed as parameters to the main operation:

```
allValuesIDS :: Int  → (Int  → Int)  → SearchTree a  → [a]
allValuesIDS initdepth incrdepth st =
  iterIDS initdepth (collectInBounds 0 initdepth st)
 where
  iterIDS _ Nil = []
```

```
iterIDS n (Cons x xs) = x : iterIDS n xs
iterIDS n Abort =
  let newdepth = incrdepth n
   in iterIDS newdepth (collectInBounds n newdepth st)
```

The key advantage of depth-first search in comparison to breadth-first search is its memory behavior: whereas breadth-first search has to store all nodes in a level of the search tree in parallel, depth-first search only needs to store the nodes in the branch from the root to the current node under investigation. Since iterative deepening uses depth-first search in each iteration, it should have a memory behavior similarly to depth-first search and, in case of wide search trees, better than breadth-first search. The price for this behavior is the recomputation of the initial goal in each iteration. However, since we defined iterative deepening on a search tree, recomputation is not required. Instead, the already evaluated part of the search tree is kept in memory, sacrificing the better memory behaviour. Therefore, we also implemented the iterative deepening strategy for top-level goals without an explicit search tree but with a recomputation of the initial goal in each iteration (see Section 4).

The various operators to encapsulate search can also be used to implement an interactive top-level search to print all values of an expression as requested by the user. For instance, the following I/O operation interactively prints all elements of a given list:

```
printResults :: [a]  → IO ()
printResults []     = putStrLn "No more values"
printResults (x:xs) = do print x
                         putStr "More values? "
                         inp <- getLine
                         if inp == "yes" then printResults xs
                                         else done
```

Hence, an interactive Prolog-like top-level behavior to show the values of an expression exp in depth-first order can be obtained by

```
getSearchTree exp >>= printResults . allValuesDFS
```

In addition, we can print the results in breadth-first order by using `allValuesBFS` instead of `allValuesDFS`. Based on these ideas, KiCS2 provides an interactive top-level search where the user can select various search strategies, e.g., depth-first, breadth-first, iterative deepening, or an experimental implementation of parallel search. However, the top-level search in KiCS2 is not implemented via the primitive encapsulation operators but in a monadic style (see also [BHPR11]) in order to avoid the explicit construction of the `SearchTree` structure. Thus, in the next section we both compare the different search strategies introduced above as well as the top-level search with the encapsulated search, in order to evaluate the potential overhead caused by abstracting and programming with search structures.

4 Benchmarks

In this section we evaluate the practical behavior of the various search strategies discussed so far. Since they are only supported by the Curry implementation KiCS2, we use this system for our evaluation. A general comparison of KiCS2 and other Curry implementations can be found in [BHPR11]. Since KiCS2 compiles Curry programs into Haskell programs, we used the Glasgow Haskell Compiler (GHC 7.0.4, option -O2) to compile and execute the generated target programs. All benchmarks were executed on a 32bit Linux machine running Ubuntu 11.10 with an Intel Core 2 Duo (2.13 GHz) processor and 4 GiB RAM. The timings were performed with the time command measuring the execution time (in seconds) of a compiled executable for each benchmark. Due to the lack of precise measurements of the space behavior, we measured only the execution times.

Program	DFS	IDS(+1)	IDS(*2)	eDFS	eBFS	eIDS(+1)	eIDS(*2)
PermSort	13.12	841.58	32.77	38.48	66.82	72.64	46.79
Last	0.10	660.08	0.26	0.34	0.19	6.59	0.39
Half	0.67	561.45	1.62	1.43	1.22	1.86	1.71
Graph	1.76	52.53	2.40	3.51	4.22	4.26	3.83
HorseMan	2.64	639.64	6.63	3.23	3.29	3.20	3.17
MAC	6.80	544.05	7.84	21.19	18.34	18.59	21.03
Queens	24.74	374.64	33.62	42.71	43.98	44.01	42.65

Figure 1: Benchmarks: computing all values

Figure 1 shows the benchmark results when all values of a given expression are computed with various search strategies. All benchmark programs are non-deterministic programs. "PermSort" sorts a list containing 15 elements by enumerating all permutations and selecting the sorted ones, "Last" computes the last element `x` of a list `xs` containing 10,000 elements by solving the equation "`ys++[x] =:= xs`" (see Section 2), "Half" computes the half `y` of a natural number `x` (in Peano representation) by solving the equation `y+y=:=x`, "Graph" computes some path in a graph where the edges are represented by a non-deterministic "successor" operation, "HorseMan" computes the numbers of horses and men from given numbers of heads and feet by searching for appropriate numbers in Peano representation, "MAC" solves the "Missionaries and Cannibals" puzzle (see [AH02, Sect. 3.1]) for reasonable numbers of missionaries and cannibals to obtain measurable run times, and "Queens" computes safe placements of queens on a chess board.

The columns in Figure 1 are the various search strategies considered in this paper. "DFS" denotes the top-level depth-first search strategy which is similar to `allValuesDFS` but implements this strategy in a monadic style without the explicit construction of a search tree. The encapsulated search strategies are prefixed by "e", i.e., "eDFS" and "eBFS" correspond to `allValuesDFS` and `allValuesBFS`, respectively. "eIDS(+1)" corresponds to `allValuesIDS` with an initial depth bound of `10` and the depth increment operation `(+1)`, whereas "eIDS(*2)" uses the same initial depth bound but the depth increment operation `(*2)`, i.e., to depth bound is doubled in each iteration. Finally, "IDS(+1)" and "IDS(*2)" are iterative deepening strategies with similar parameters but recompute the

Program	DFS	IDS(+1)	IDS(*2)	eDFS	eBFS	eIDS(+1)	eIDS(*2)
PermSort	12.57	804.82	32.09	34.75	67.02	70.47	46.66
Last	0.10	661.92	0.24	0.35	0.19	6.57	0.39
Half	0.34	72.64	0.58	0.69	0.59	0.78	0.78
Graph	0.00	0.10	0.04	0.00	0.04	0.04	0.03
HorseMan	2.06	636.62	6.64	2.52	2.60	2.61	2.57
MAC	0.14	38.75	0.94	0.94	2.81	2.75	2.79
Queens	1.40	370.26	33.65	2.44	8.95	8.86	3.04
NDNums	oom	47.89	0.14	oom	oom	oom	0.47

Figure 2: Benchmarks: computing a single value

initial expression in each iteration in order to trade space for run time, as discussed in Section 3.

The table entries in Figure 1 contain the run times in seconds to compute all values of the initial expression. As one can see, the top-level depth-first search is the most efficient search strategy. The encapsulated version of depth-first search (eDFS) with the explicit construction of the search tree causes some overhead, since the search strategy is encoded in the source program rather than in the run-time system as in the top-level search (DFS). The benchmarks also show that complete search strategies, like "eBFS" and "eIDS(*2)" are viable alternatives to the incomplete depth-first search strategy. Their overhead (sometimes they are even faster than "eDFS", but they are always slower than top-level search) is acceptable taking into account the fact that one need not to reason about the details of exploring the search space. The behavior of iterative deepening is largely influenced by its parameterization. A constant increment of the depth bound causes a big overhead compared to doubling the depth bound in each iteration, in particular, when the search tree is slim as in "Last". This is even worse for top-level iterative deepening which recomputes the initial expression in each iteration, see "IDS(+1)".

The benchmarks indicate that breadth-first search seems to be a good strategy taking into account the size of memory provided by modern computers. However, if the search tree is wide (i.e., the nodes in each level increases with the depth of the tree), then iterative deepening is superior to breadth-first search. To examine such a case, we added a benchmark "NDNums" which defines a non-deterministic operation with a high branching factor that returns all inreasing numbers

```
g n = g (n+1) ? n ? g (n+1)
```

and solve the equation "`g 0 =:= 29`". Obviously, the computation of all values will never terminate. Therefore, we executed the benchmarks to compute only a first value of an expression via different search strategies. Figure 2 contains the corresponding results where "oom" denotes a memory overflow in a computation. As one can see, iterative deepening is the only search strategy that is able to find a solution in all examples.

5 Conclusions

We discussed the use of different search strategies in functional logic programs. In order to enable user-programmable search strategies, modern functional logic languages like Curry provide primitives to represent non-deterministic computations or values as data structures that can be traversed like any term structure. The Curry implementation KiCS2, considered in this paper, provides a primitive `getSearchTree` that returns a tree representation of a non-deterministic computation. This representation can be used to define various search strategies, like depth-first search, breadth-first search, or iterative deepening search.

We have practically evaluated and compared the efficiency of these strategies. The benchmarks indicated that complete strategies are a viable alternative to incomplete strategies, that have been favored in the past due to limited memory requirements. Decoupling non-deterministic programs from their search strategy could lead to a more declarative programming style (instead of the use of predicates with side effects as in Prolog) and enable more potential for optimization, e.g., parallel search strategies. The explicit definition of search strategies has also been advocated for combining logic programs with different constraint solvers [FHPR06, Sch97].

Our results indicate that it could be reasonable to make complete strategies the default in functional logic languages. This would have a good impact on teaching declarative programming to beginners. Furthermore, if efficiency and memory limitations are important, one can still use an efficient strategy, like depth-first search, provided that it is able to find all solutions (e.g., in case of a finite search space). It is an interesting topic for future work to statically approximate situations where the use of theoretically incomplete strategies is sufficient.

References

[AEH00] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[AFV98] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[AGK06] S. Abramov, R. Glück, and Y. Klimov. An Universal Resolving Algorithm for Inverse Computation of Lazy Languages. In *Perspectives of Systems Informatics (PSI 2006)*, pages 27–40. Springer LNCS 4378, 2006.

[AH02] S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.

[AH10] S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, 53(4):74–85, 2010.

[AH11] S. Antoy and M. Hanus. New Functional Logic Design Patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 19–34. Springer LNCS 6816, 2011.

[AHH+05] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

[BH07] B. Braßel and F. Huch. On a Tighter Integration of Functional and Logic Programming. In *Proc. APLAS 2007*, pages 122–138. Springer LNCS 4807, 2007.

[BH09] B. Braßel and F. Huch. The Kiel Curry System KiCS. In *Applications of Declarative Programming and Knowledge Management*, pages 195–205. Springer LNAI 5437, 2009.

[BHH04] B. Braßel, M. Hanus, and F. Huch. Encapsulating Non-Determinism in Functional Logic Computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

[BHM08] B. Braßel, M. Hanus, and M. Müller. High-Level Database Programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.

[BHPR11] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.

[CL00] K. Claessen and P. Ljunglöf. Typed Logical Variables in Haskell. In *Proc. ACM SIGPLAN Haskell Workshop*, Montreal, 2000.

[CLF99] R. Caballero and F.J. López-Fraguas. A Functional-Logic Perspective of Parsing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 85–99. Springer LNCS 1722, 1999.

[FHPR06] S. Frank, P. Hofstedt, P. Pepper, and D. Reckmann. Solution Strategies for Multi-domain Constraint Logic Programs. In *Perspectives of Systems Informatics (PSI 2006)*, pages 209–222. Springer LNCS 4378, 2006.

[Fis05] S. Fischer. A Functional Logic Database Library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 54–59. ACM Press, 2005.

[FK07] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 63–74. ACM Press, 2007.

[GMHGLFRA99] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

[HAB+10] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2010.

[Han97] M. Hanus. Teaching Functional and Logic Programming with a Single Computation Model. In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, pages 335–350. Springer LNCS 1292, 1997.

[Han00] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.

[Han01] M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[Han06] M. Hanus. Type-Oriented Construction of Web User Interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.

[Han07] M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

[He06] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.

[Hin01] R. Hinze. Prolog's control constructs in a functional setting - Axioms and implementation. *International Journal of Foundations of Computer Science*, 12(2):125–170, 2001.

[HK10] M. Hanus and S. Koschnicke. An ER-based Framework for Declarative Web Programming. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, pages 201–216. Springer LNCS 5937, 2010.

[HS98] M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.

[LFSH99] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

[LFSH04] F.J. López-Fraguas and J. Sánchez-Hernández. A Proof Theoretic Approach to Failure in Functional Logic Programming. *Theory and Practice of Logic Programming*, 4(1):41–74, 2004.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[Lux99] W. Lux. Implementing Encapsulated Search for a Lazy Functional Logic Language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.

[PJ03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[RNL08] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proc. of the 1st ACM SIGPLAN Symposium on Haskell*, pages 37–48. ACM Press, 2008.

[Sch97] C. Schulte. Programming Constraint Inference Engines. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 519–533. Springer LNCS 1330, 1997.

[SH06] J. Sánchez-Hernández. Constructive Failure in Functional-Logic Programming: From Theory to Implementation. *Journal of Universal Computer Science*, 12(11):1574–1593, 2006.

[Wad97] P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.