Can Logic Programming Be Liberated from Predicates and Backtracking?

Michael Hanus

Institut für Informatik, Kiel University, Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. Logic programming has a long history. The representative of logic programming in practice, the language Prolog, has been introduced more than 50 years ago. The main features of Prolog are still present today: a Prolog program is a set of predicate definitions executed by resolution steps with a backtracking search strategy. The use of backtracking was justified by efficiency reasons when Prolog was invented. However, its incompleteness destroys the elegant connection of logic programming and the underlying Horn clause logic and causes difficulties to teach logic programming. Moreover, the restriction to predicates hinders an adequate modeling of real world problems, which are often functions from input to output data, and leads to unnecessarily inefficient executions. In this paper we show a way to overcome these problems. By transforming predicates and goals into functions and nested expressions, one can evaluate them with a demand-driven strategy which might reduce the number of computation steps and avoid infinite search spaces. Replacing backtracking by complete search strategies with new implementation techniques closes the gap between the theory and practice of logic programming. In this way, we can keep the ideas of logic programming in future programming systems.

1 Introduction

Logic programming was developed as a restriction of the general resolution principle [34] to Horn clauses so that efficient linear (SLD-resolution) proofs can be constructed (see also [14] for some historical background). It became popular when concrete implementations in the form of interpreters (and later compilers) for the programming language Prolog were available. Horn clauses and SLD-resolution are tightly connected to mathematical logic. The soundness and completeness of SLD-resolution establish the foundation of logic programming [28]. Unfortunately, the memory restrictions of computers at that time caused a gap between these theoretical foundations and the practice of logic programming in Prolog: non-deterministic computations are evaluated by backtracking so that the theoretical completeness of SLD-resolution is lost. For instance, consider the definition of a Prolog predicate relating a list and its last element:

```
last([H|T],E) :- last(T,E).
last([E],E).
```

This definition works when the list is known:

```
?- last([1,2,3],E).
E = 3
```

One of the advantages of logic programming is the absence of fixed input and output parameters. Instead of providing a known value for an argument of a predicate, one can also call the predicate with a free variable for this argument (as `E` above) so that a result is computed by binding this variable to some value. In practice, this advantage is often lost when non-deterministic search is implemented by backtracking, since infinite branches in a search tree might preclude the computation of valid answers. For instance, Prolog does not compute any result for the definition of `last`, as shown above, when the list is unknown, e.g., for the goal `last(L,3)`: the backtracking strategy causes an infinite chain of applications of the first rule. This shows the gap between the theory of logic programming, where the complete SLD-resolution method yields an infinite set of answers to this goal, and the practice of logic programming implemented with the language Prolog.

Compared to functional programming, logic programming is often considered as the more flexible and expressive programming paradigm [32]. This is no longer true if we consider functional logic languages [6], such as Curry [23], which amalgamates features of functional and logic programming and does not force the programmer to model all knowledge in the form of predicates. Actually, many real world problems can be modeled in a more adequate format in the form of functions mapping input data to output data. With a functional logic language, one has the same expressiveness as in logic programming since any (pure) logic program can be transformed into a functional logic program so that the same solutions are computed, as we discuss in this paper. Moreover, the equivalent functional logic programs behave more efficiently and can avoid infinite search spaces.

Example 1. Consider the following Prolog program which defines the well-known predicate `app` relating two lists to their concatenation and a predicate `app3` relating three lists to their concatenation:

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

app3(Xs,Ys,Zs,Ts) :- app(Xs,Ys,Rs), app(Rs,Zs,Ts)
```

When evaluating the goal `app3(Xs,Ys,Zs,[])`, a Prolog system yields the answer {`Xs`↦`[]`, `Ys`↦`[]`, `Zs`↦`[]`}, but it does not terminate when searching for more answers. Similarly, Prolog systems do not terminate when evaluating the goal `app3(Xs,[1],Zs,[])`. This is different if we translate the predicates into functions by considering the last argument as output, as often intended when formulating functional knowledge as predicates. For instance, the tool described in [18] translates these definitions into the following Curry program:

```
app []     ys = ys
app (x:xs) ys = x : app xs ys

app3 xs ys zs = app (app xs ys) zs
```

Since Curry exploits functional dependencies between input and output data to implement a demand-driven strategy, the equations (which are equivalent to the goals above) `app3 xs ys zs =:= []` and `app3 xs [1] zs =:= []` have finite search spaces so that the evaluation in Curry terminates.

In summary, we can see that the basis of Prolog, i.e., predicates and backtracking, has various disadvantages:

- The theoretical completeness of SLD-resolution is lost.
- Backtracking hinders teaching the ideas of logic programming since beginners are often faced with the influence of the search strategy.
- Programmers have to think about the influence of backtracking to the success of computations—a contradiction to the idea of *declarative programming*.
- The use of predicates instead of functions yields a flat structure of goals so that functional dependencies cannot be exploited to avoid useless search.

In this paper we argue that all these problems can be avoided without loosing the flexibility of logic programming by using *functional* logic programming instead of pure logic programming. Functions are helpful to reduce the number of computation steps and avoid infinite search spaces. Contemporary functional logic languages, such as Curry, do not fix a deterministic backtracking strategy for search but support complete search strategies.[1] Thus, abandoning backtracking in logic programming is similar to the removal of the von Neumann bottleneck by functional programming [10]: it supports a higher, declarative programming style which frees the programmer from thinking about low-level control details.

In the following, we sketch[2] methods to get rid of predicates and backtracking. This can be done in a systematic way by transforming logic programs into functional logic programs on which efficient, often optimal, and complete evaluation strategies can be applied. To explain this method, we review the basics of logic and functional logic programming in the next section. Then we show how to transform pure logic programs into functional logic programs and how to apply efficient and complete evaluation strategies on the transformed programs.

The message of this paper is to show that functional logic languages are always preferable to pure logic languages. Transforming logic into functional logic programs is the formal justification. If one accepts this message, one should directly implement the desired application in a functional logic language and exploit all useful features of such languages, like declarative I/O [37], functional patterns [4], strategy-independent encapsulated search [5], default rules [7], etc.

2 Logic and Functional Logic Programming

We briefly review some notions and features of logic and functional logic programming. More details can be found in [28] and in surveys on functional logic programming [6,17].

[1] Note that this is not the case for all such languages. For instance, the functional logic language Verse [8] fixes a deterministic, backtracking-like search strategy.

[2] More details can be found in [11,18] on which this paper is partially based.

We use Prolog syntax to present logic programs. *Terms* in logic programs are constructed from variables ($X, Y, \ldots$), numbers, atom constants ($c, d, \ldots$), and functors or term constructors ($f, g, \ldots$) applied to a sequence of terms, like $f(t_1, \ldots, t_n)$. A *literal* $p(t_1, \ldots, t_n)$ is a predicate p applied to a sequence of terms, and a *goal* $L_1, \ldots, L_k$ is a sequence of literals, where $\square$ denotes the empty goal ($k = 0$). *Clauses* `L :- B` define predicates, where the *head* L is a literal and the *body* B is a goal (a *fact* is a clause with an empty body $\square$, otherwise it is a *rule*). A *logic program* is a sequence of clauses.

Logic programs are evaluated by SLD-resolution steps, where we consider the leftmost selection rule here. Thus, if $G = L_1, \ldots, L_k$ is a goal and `L :- B` is a variant of a program clause (with fresh variables) such that there exists a most general unifier[3] (*mgu*) σ of L_1 and L, then $G \vdash_\sigma \sigma(B, L_2, \ldots, L_k)$ is a *resolution* step. A *computed answer* for a goal G is a substitution σ (restricted to the variables occurring in G) which is composed of all unifiers of a sequence of resolution steps from G to $\square$.

Example 2. Consider the predicates of Example 1 and the list reversal

```
rev([],[]).
rev([X|Xs],Zs) :- rev(Xs,Ys), app(Ys,[X],Zs).
```

The predicate `pali` relates a palindrome with its middle element:

```
pali(Zs,X) :- app3(Xs,[X],Ys,Zs), rev(Xs,Ys).
```

Prolog computes for the goal `pali([1,2,3,2,1],M)` the answer $\{$`M` $\mapsto$ `3`$\}$ but then it does not terminate, since it enumerates arbitrary large values for `Xs`. Similarly, it does not terminate on `pali([1,2],M)`.

Functional logic programming [6,17] integrates the most important features of functional and logic languages, such as higher-order functions and lazy (demand-driven) evaluation from functional programming and non-deterministic search and computing with partial information from logic programming. The declarative multi-paradigm language Curry [23], which we use in this paper, is a functional logic language with advanced programming concepts. Its syntax is close to Haskell [31], i.e., variables and names of defined operations start with lowercase letters and the names of data constructors start with an uppercase letter. The application of an operation f to e is denoted by juxtaposition ("$f\ e$").

In addition to Haskell, Curry allows *free* (*logic*) *variables* in program rules (equations) and initial expressions. Function calls with free variables are evaluated by a possibly non-deterministic instantiation of arguments. Similarly to Prolog and in contrast to Haskell, Curry evaluates operations defined by rules with overlapping left-hand sides in a non-deterministic manner by applying all possible rules. The archetype of an operation defined by overlapping rules is the non-deterministic choice, defined in Curry [23] as the infix operator "`?`" by

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

[3] Substitutions, variants, and unifiers are defined as usual [28].

Hence, "`0 ? 1`" yields two values: `0` and `1`. In contrast to Prolog, the concrete strategy to compute these values, i.e., the search strategy, is not fixed in Curry so that implementations of Curry can provide various search strategies.

Example 3. The following Curry program[4] defines the predicates of Example 2 in a functional manner, where logic features (like the free variables `xs` and `x`) are exploited to define `pali`:

```
rev []     = []
rev (x:xs) = app (rev xs) [x]

pali zs | zs =:= app3 xs [x] (rev xs)
        = x
```

"`|`" introduces a condition, and "`=:=`" denotes semantic unification, i.e., the expressions on both sides are evaluated before unifying them.

Since `app` and `app3` can be called with free variables in arguments, the condition in the definition of `pali` is solved by instantiating `xs` and `x` to appropriate *values* (i.e., expressions without defined functions) before reducing a function call. This corresponds to narrowing [33,35]. $t \leadsto_\sigma t'$ is a *narrowing step* if there is some non-variable position p in t, an equation (program rule) l `=` r, and an mgu σ of $t|_p$ and l such that $t' = \sigma(t[r]_p)$,[5] i.e., t' is obtained from t by replacing the subterm $t|_p$ by the equation's right-hand side and applying the unifier. Conditional equations l `|` c `=` r are considered as syntactic sugar for the unconditional equation l `=` c `&>` r, where "`&>`" is defined by `True &> x = x`.

Curry is based on the *needed narrowing strategy* [2] which uses non-most-general unifiers in narrowing steps to ensure the optimality of computations. Needed narrowing is a demand-driven evaluation strategy, i.e., it supports computations with infinite data structures [26] and can avoid superfluous computations so that it is optimal w.r.t. the number of computed solutions and the length of derivation [2]. This is our motivation to transform logic programs into Curry programs, since it can reduce infinite search spaces to finite ones. For instance, the evaluation of the expression `pali []` has a finite computation space: the generation of larger lists for the first argument of `app3` is avoided since there is no demand for such numbers.

Curry has many more features which are useful to implement applications, like *set functions* [5] to encapsulate search, and standard features from functional programming, like modules or monadic I/O [37]. However, the kernel of Curry described so far should be sufficient to understand the remaining contents.

Early implementations of functional logic languages, like PAKCS [3] or TOY [29], used Prolog as a target language due to its built-in support for non-determinism. A drawback of this approach is that they inherit the incompleteness of Prolog's backtracking strategy. In order to get rid of this fixed search strategy, subsequent implementations are based on the idea to represent non-deterministic

[4] The concrete syntax is simplified by omitting the declaration of free variables, like `x` and `xs`, which is required in Curry programs to enable consistency checks by the compiler.

[5] We use common notations from term rewriting [9].

choices as data. Instead of directly evaluating non-deterministic branches, the alternatives are returned as a tree structure so that search strategies can be defined as tree traversals, which supports an easy switch between different strategies. For instance, the Curry system KiCS2 [12] and Curry2Go [11] have options to select different search strategies, like depth-first, breadth-first, or fair search.

3 From Predicates to Functions

This section discusses methods to transform logic programs into functional logic programs by mapping predicates and goals into functions and nested expressions. Since predicates can be viewed as Boolean functions, the simplest transformation maps each predicate into a Boolean function and each clause into a (conditional) equation. For instance, the clauses of predicate `app` shown in Example 1 can be transformed into

```
app [] ys ys = True
app (x:xs) ys (x:zs) | app xs ys zs = True
```

This *conservative transformation* [18] does not change the structure of derivations since narrowing steps on Boolean functions correspond to resolution steps. Thus, there is no real advantage to perform this transformation.

To exploit the computational power of functional logic languages, predicates should be transformed into non-Boolean functions by selecting some arguments as results and generating function definitions according to this selection.

Example 4. Consider again predicate `app` of Example 1. If the third argument is selected as a result argument (as often intended in logic programs), the clauses of `app` can be transformed into the following functional logic program:

```
app [] ys = ys
app (x:xs) ys | zs =:= app xs ys = x:zs
```

Although any set of argument positions can be selected as results, there are heuristics to select result arguments so that optimal evaluations are ensured for large classes of programs, as discussed in [18].

It is shown in [18] that, even if this *functional transformation* is used, there is a strong one-to-one correspondence, independent of the selection of result arguments, between resolution derivations w.r.t. the original logic program and narrowing derivations w.r.t. the transformed program. To improve this situation and get some computational advantage, one has to replace the unification occurring in conditions by *let* bindings whenever possible[6] and inline these bindings if reasonable. For instance, one can transform the rule

```
app (x:xs) ys | zs =:= app xs ys = x:zs
```

into

```
app (x:xs) ys = let zs = app xs ys in x:zs
```

[6] This is possible when the variable in the left-hand side of the unification has no occurrences in result arguments of other goal literals, see [18] for a precise discussion.

and inline the binding of `zs` into

```
app (x:xs) ys = x : app xs ys
```

This *demand functional transformation* is described in detail in [18]. If the transformed program is eagerly evaluated, i.e., the arguments of a function call are evaluated before replacing the function call by its body ("call by value"), there is no operational difference between programs transformed by the functional and the demand functional transformation. This situation changes when the arguments are evaluated "by need," as in Haskell or Curry and discussed in [25,26].

Example 5. Consider the predicate `siglist` defined by the clauses

```
siglist([],zero).
siglist([_],one).
siglist([_,_|_],many).
```

The demand functional transformation yields

```
siglist [] = Zero
siglist [_] = One
siglist (_:_:_) = Many
```

Now consider the evaluation of the expression `siglist (app` xs ys`)`, where xs is a long list with n elements. An eager evaluation requires $n + 1$ rewrite steps, whereas a non-strict language needs only three steps.

Although it seems that the demand functional transformation is the way to go, there is one potential problem of this transformation: it might change the semantics, i.e., the set of computed solutions. This could be the case if the evaluation of some subexpression is not demanded and its evaluation would fail to yield a value. This failure would be propagated in the original logic program, but it might be "hidden" in the transformed program. For instance, consider a predicate relating a non-empty list with its tail

```
tail([_|Xs],Xs).
```

and its application in the predicate

```
sigtail(S) :- tail([],Xs), app([0,1],Xs,Ys), siglist(Ys,S).
```

Due to the failure of the first subgoal, the goal "`?- sigtail(S).`" fails. However, the demand functional transformation yields

```
tail (_:xs) = xs

sigtail = siglist (app [0,1] (tail []))
```

The demand-driven or lazy evaluation of `sigtail`, which performs only necessary reductions, yields the value `Many`. This is conform to the mathematical principle of "replacing equals by equals" but it changes the set of answers w.r.t. the original logic program.

It is possible to modify the transformation so that the transformed functional logic program computes only more general answers than the original logic program, i.e., each answer of the functional logic program is a generalization of an answer computed by the logic program. Computing more general answers is also preferable in pure logic programming since it results in smaller search spaces.

As shown in [19], the demand functional transformation yields programs so that needed narrowing is sound and complete w.r.t. the logical consequences of the logic program, where soundness requires that all functions are totally defined. Hence, if there are also partially defined functions, one has to ensure that every occurrence of such a function in a computation will eventually be evaluated. This can be obtained by a slight modification of the demand functional transformation which is called *fail-sensitive functional transformation*. If a rule's right-hand side contains an application `(`f e`)` and the evaluation of the expression e *might fail* to compute a value, i.e., it contains a partially defined function, then this application is replaced by

`(`f `$!` e`)`

"`$!`" denotes function application with a strict evaluation of argument e. Thus, if the evaluation of e fails, the evaluation of `(`f `$!` e`)` fails.

For instance, the fail-sensitive functional transformation maps the definition of predicate `sigtail` into

```
sigtail = siglist $! (app [0,1] $! tail [])
```

Then the evaluation of `sigtail` leads to a failure due to the enforced evaluation of `(tail [])`. Note that the operator "`$!`" must be inserted at all places where a potentially failing expression occurs and not only where the failing expressions occurs first.

The fail-sensitive functional transformation requires information whether operations are totally defined. Since this is undecidable in general, one can approximate this property by splitting it into two parts: termination and non-occurrence of failures due to incomplete patterns, as visible in the definition of `tail`.

Termination of rewrite systems or functional programs is well-studied so that various techniques are available to approximate this property, e.g., [16,27]. To approximate absence of failures, one could simply mark a function as failing if it is defined with an incomplete set of patterns or call a failing function in its right-hand side. This results in a fixpoint computation of this property. This simple approximation can be improved by considering the context of using failing functions in right-hand sides. For instance, the following function uses the failing function `tail` but it is totally defined since `tail` is called with a non-empty list:

```
tailOrEmpty []     = []
tailOrEmpty (x:xs) = tail (x:xs)
```

The tool described in [20] approximate the failing property of functions by approximating call types for functions, which ensures a fail-free evaluation, and using call types to approximate the failure status of functions. For instance, the call type of `tail` are all non-empty lists so that the call of `tail` in the second rule of `tailOrEmpty` does not cause a failure. Hence, `tailOrEmpty` is totally defined. In practice, only a few operations of larger programs have non-trivial call types, i.e., might fail on specific arguments.[7]

[7] This requires also the consideration of intended types. For instance, `app` is totally defined on lists, which are the intended arguments, although `app` fails on the unintended argument `42`. The consideration of type information is discussed in [19].

Language:	Prolog	Prolog	Curry
System:	SWI 9.0.4	SICStus 4.9.0	KiCS2 3.1.0
`rev_4096`	0.23	0.22	0.10
`tak_27_16_8`	6.97	3.23	0.74
`ackermann_3_9`	2.13	8.72	0.07
`pali_[]`	∞	∞	0.01
`siglist_app_0`	∞	∞	0.01
`numleaves_7`	∞	∞	0.01
`permsort_10`	1.43	0.28	0.03
`permsort_11`	16.16	1.38	0.08
`permsort_12`	206.34	15.23	0.28

Table 1. Execution times (in seconds) of Prolog and generated Curry programs

Exploiting such tools, one can implement the fail-sensitive functional transformation in three steps. First, the logic program is transformed with the demand functional transformation as described in [18]. Then, the generated program is analyzed with the failure-inference tool described in [20] (automated termination checks are currently omitted since it is seldom that operations generated from Prolog are completely defined but non-terminating). Based on the failure information, the transformed program is modified by replacing function applications `(`$f\ e$`)` by `(`f `$!` e`)` whenever the expression e might fail.

This transformation produces functional logic programs which compute the same or more general answers compared to the original logic programs. In the worst case (if all functions are possible failing), the same number of evaluation steps are performed, but in many other cases, the transformation reduces the number of computation steps (due to the optimality of needed narrowing) so that infinite search spaces might be reduced to finite ones. Thus, the transformation has no disadvantage but in some cases one gets considerable improvements.

To evaluate this transformation, we have implemented a tool performing the fail-sensitive functional transformation as described above.[8] Table 1 contains the results of executing various Prolog programs with SWI-Prolog and SICStus-Prolog and the Curry programs obtained by applying the fail-sensitive functional transformation with the Curry system KiCS2 [12]. KiCS2 compiles Curry programs into Haskell and uses the Glasgow Haskell Compiler (GHC 9.4.5) to generate machine code.[9] The examples, which can be found in the appendix, are small programs since larger Prolog programs are seldom logic programs—they often use non-declarative features. Such non-declarative features are either not necessary in functional logic programs (e.g., cuts are replaced by exploiting

[8] The tool, available at `https://cpm.curry-lang.org/pkgs/prolog2curry-1.2.0.html`, is implemented as a Curry package for easy installation. A script together with all required tools is available as a docker image at `https://hub.docker.com/r/currylang/prolog2curry`.

[9] The benchmarks were executed on a Linux machine running Ubuntu 22.04 with an Intel Core i7-1165G7 (2.80GHz) processor with eight cores. The time is the total run time of executing a binary generated with the Prolog/Curry systems.

functional dependencies) or can be reformulated in a declarative manner (e.g., declarative monadic I/O, state monads).

The first three benchmarks are purely deterministic computations. `rev_4096` is the naive list reversal applied to a list of 4096 elements. `tak_27_16_8` applies the highly recursive `tak` function [30] to the values `(27,16,8)` in Peano representation. The Ackermann function, defined on Peano numbers, is applied to the Peano representation of `(3,9)`. For these functions, the demand-driven evaluation strategy has no real advantage since the values of all subexpressions are required. The situation is different in the next three benchmarks where the original logic program has an infinite search space and the transformed functional logic program has a finite search space, similarly to Example 1. `pali_[]` denotes the evaluation of `pali([],M)` (see Examples 2 and 3), `siglist_app_0` denotes the evaluation of the goal "`app(Xs,[],Zs), siglist(Zs,zero)`", and `numleaves_7` denotes the generation of all binary trees with seven leaves. Since Table 1 shows the time to compute *all* answers to the given goals, the Prolog systems do not terminate due to the infinite search spaces. The final benchmarks, permutation sort applied to lists containing 10, 11, and 12 decreasing Peano numbers, demonstrates the advantage of demand-driven evaluation even if the search space is finite. As discussed at various places [6,17], the functional logic version explores permutations in a demand-driven manner so that not all permutations are actually generated. Thus, our transformation maps a "generate-and-test" algorithm into a more efficient "test-of-generate-as-demanded" algorithm with a lower complexity, as apparent from the benchmarks.

4 From Backtracking to Complete Search Strategies

As already mentioned, Prolog is based on backtracking to deal with "don't know" non-deterministic resolution steps. This was a way to deal with limited hardware resources when Prolog was invented. Since this strategy is fixed for Prolog [15], it has the unfortunate consequence that many non-logical features, like input/output, arithmetic, search-space pruning (cut), depend on this strategy so that it is not easy to change it in real-world applications of Prolog. However, in order to close the gap between theory and practice of logic programming, support a higher-level understanding of programs, and improve the situation when teaching logic programming, complete search strategies are necessary.

Curry does not fix backtracking or depth-first search so that implementations can support other search strategies—there are no language features depending on backtracking. For instance, PAKCS [3,21] compiles into Prolog so that backtracking search is used. KiCS2 [12] compiles Curry programs into Haskell programs and represent the search space as a tree structure on which search strategies are defined so that one can switch between depth-first (DFS) or breadth-first search (BFS), among others. Curry2Go [11] compiles Curry programs into Go[10] programs. Go is a statically typed language with garbage collection and direct

[10] `https://golang.org/`

	PAKCS	KiCS2		Curry2Go		
Example		DFS	BFS	DFS	BFS	FS
`nrev_4096`	6.29	0.10	0.10	0.85	0.85	0.85
`takPeano_24_16_8`	56.78	0.12	0.12	8.05	7.98	7.76
`primesHO_1000`	29.46	0.04	0.04	3.51	3.58	3.55
`psort_13`	18.92	0.35	2.32	7.11	7.25	9.51
`addNum_2`	0.18	0.24	0.57	0.28	0.29	0.28
`addNum_5`	0.20	2.01	4.36	0.67	0.67	0.35
`addNum_10`	0.24	11.83	16.84	1.53	1.54	0.54
`select_50`	0.09	0.19	0.27	0.02	0.02	0.02
`select_100`	0.27	4.13	4.80	0.06	0.06	0.03
`select_150`	0.56	25.10	32.42	0.13	0.13	0.06
`isort_primes4`	9.56	0.02	0.02	1.15	1.14	1.11
`psort_primes4`	112.38	0.02	0.02	1.11	1.11	0.71

Table 2. Comparing Curry system with search strategies

support for CSP-like concurrency [24] and lightweight threads (*goroutines*). The latter feature is used to provide, in addition to DFS and BFS, a fair search strategy. For instance, consider the following contrived example:

```
idND :: a → a
idND n = loop ? n ? loop
```

where `loop` is non-terminating. Semantically, `idND` is the identity function but, operationally, it is non-deterministically defined with looping alternatives. Both DFS and BFS loop on the expression `idND 0` instead of returning the value `0`, since there is no choice when evaluating `loop`. However, the *fair search* (FS) strategy of Curry2Go returns this value since FS evaluates non-deterministic branches concurrently as goroutines and collects the computed results in a channel [11].

To show that the efficiency of advanced search strategies is not really worse than backtracking, we compared these Curry implementations and their search strategies. Table 2 shows the run times (in seconds as the average of three runs) of various examples[11] and search strategies. The first three benchmarks are typical purely functional programs. `nrev_4096` is the quadratic naive reverse algorithm applied to a list with 4096 elements, `takPeano` is a highly recursive function on naturals [30] applied to arguments (24,16,8) in Peano representation, and `primesHO_1000` computes the 1000th prime number by constructing an infinite list of all primes via the sieve of Eratosthenes (using higher-order functions). For these examples, Curry2Go is much faster than PAKCS but less efficient than KiCS2, which is not surprising since Haskell/GHC is a highly optimized functional programming system.

The remaining non-deterministic benchmark programs show that KiCS2 and Curry2Go are competitive with PAKCS (which exploits Prolog's built-in support for non-determinism). `psort_13` is the naive permutation sort applied to a list of

[11] The examples are available at `https://github.com/curry-language/curry2go`.

13 elements. `addNum_`n non-deterministically chooses a number (out of 2000) and adds it n times, and `select_`n non-deterministically selects an element in a list of length n and sums up the element and the list without the selected element. The considerable slowdown in KiCS2 with increasing values for n is caused by the duplication of choices in pull-tab steps [1] when non-deterministic expressions are shared, as discussed in [22]. Curry2Go avoids this problem by adding a kind of memoization for choices, as described in [11,22].

Apart from the fact that the fair search strategy of Curry2Go is the only operationally complete strategy (e.g., it is able to compute a value of `idND 0`), there are also other interesting differences between the search strategies. For instance, KiCS2 shows some overhead of BFS compared to DFS (possibly due to the additional structures used to implement breadth-first tree search), whereas there is almost no overhead in Curry2Go (since the difference between BFS and DFS is just a different scheduling of tasks). Moreover, the fair search (FS) strategy is sometimes faster than BFS and DFS thanks to the use of goroutines possibly scheduled on different processors. This is also visible in the last two lines of Table 2 which show the time to sort

```
[primes!!303, primes!!302, primes!!301, primes!!300]
```

with the deterministic insertion sort (`isort`) and the non-deterministic permutation sort (`psort`) algorithm, respectively, where `primes` defines the infinite list of all prime numbers. Due to backtracking, identical computations might be repeated if they occur in different non-deterministic branches. Thus, `primes` is re-evaluated by PAKCS several times when the list is passed to the non-deterministic operation `psort`. This is not the case in implementations which represent choices in a graph structure so that the results of deterministic computations are shared across non-deterministic evaluations [13].

5 Conclusions

We have shown the advantage of using functions instead of predicates by presenting a systematic method to transform logic programs into functional logic programs so that the transformed functional logic programs always computes the same or more general answers than the original programs. This transformation does not introduce any operational disadvantage: in the worst case, the number of computation steps in the original and the transformed programs are identical, but in many other cases the number of computation steps is reduced and infinite search spaces are transformed into finite ones. Furthermore, we showed that applying complete search strategies on functional logic programs is competitive to backtracking search so that one get rid of the usual problems caused by backtracking. This closes the gap between theory and practice of logic programming and could lead to a higher, really declarative programming style. With these techniques, we can keep the ideas and advantages of logic programming in future programming systems beyond the restriction to predicates and backtracking.

References

1. S. Antoy. On the correctness of pull-tabbing. *Theory and Practice of Logic Programming*, 11(4-5):713–730, 2011.
2. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
3. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.
4. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
5. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
6. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
7. S. Antoy and M. Hanus. Default rules for Curry. *Theory and Practice of Logic Programming*, 17(2):121–147, 2017.
8. L. Augustsson, J. Breitner, K. Claessen, R. Jhala, S.L. Peyton Jones, O. Shivers, G.L. Steele, and T. Sweeney. The Verse calculus: A core calculus for deterministic functional logic programming. In *Proc. ACM International Conference on Functional Programming (ICFP 2023)*, pages 203:1–203:31, 2023.
9. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
10. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. of the ACM*, 21(8):613–641, 1978.
11. J. Böhm, M. Hanus, and F. Teegen. From non-determinism to goroutines: A fair implementation of Curry in Go. In *Proc. of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021)*, pages 16:1–16:15. ACM Press, 2021.
12. 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.
13. 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.
14. J. Cohen. A view of the origins and development of Prolog. *Communications of the ACM*, 31(1):26–36, 1988.
15. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog - the standard: reference manual*. Springer, 1996.
16. J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automatic termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2):Article 7, 2011.
17. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
18. M. Hanus. From logic to functional logic programs. *Theory and Practice of Logic Programming*, 22(4):538–554, 2022.

19. M. Hanus. Improving logic programs by adding functions. In *Proceedings of the 34th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2024)*. Springer LNCS (to appear), 2024.
20. M. Hanus. Inferring non-failure conditions for declarative programs. In *Proc. of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024)*, pages 167–187. Springer LNCS 14659, 2024.
21. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner, and F. Teegen. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2023.
22. M. Hanus and F. Teegen. Memoized pull-tabbing for functional logic programming. In *Proc. of the 28th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2020)*, pages 57–73. Springer LNCS 12560, 2021.
23. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-lang.org`, 2016.
24. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
25. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–443. MIT Press, 1991.
26. J. Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison Wesley, 1990.
27. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 81–92, 2001.
28. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
29. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
30. W. Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202. Springer, 1992.
31. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
32. U.S. Reddy. Transformation of logic programs into functional programs. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 187–196, Atlantic City, 1984.
33. U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.
34. J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
35. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.
36. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 2nd edition, 1994.
37. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

A Source Code of Benchmarks

This appendix shows the Prolog source code of some predicates used in the benchmarks in Sect. 3 and the Curry code generated by our tool implementing the fail-sensitive functional transformation.

A.1 `rev`

The predicate `rev` is the well-known naive reverse with a quadratic complexity:

```
app([],Xs,Xs).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

rev([],[]).
rev([X|Xs],R) :- rev(Xs,Zs), app(Zs,[X],R).
```

Both predicates are translated into totally defined functions:

```
app [] xs = xs
app (x : xs) ys = x : app xs ys

rev [] = []
rev (x : xs) = app (rev xs) [x]
```

A.2 `tak`

The function `tak` is defined in Prolog on Peano numbers, where `o` represents zero and `s` represents the successor of a natural number:

```
tak(X,Y,Z,A) :- leq(X,Y,XLEQY), takc(XLEQY,X,Y,Z,A).

takc(true,X,Y,Z,Z).
takc(false,X,Y,Z,A) :-
        dec(X,X1),
        tak(X1,Y,Z,A1),
        dec(Y,Y1),
        tak(Y1,Z,X,A2),
        dec(Z,Z1),
        tak(Z1,X,Y,A3),
        tak(A1,A2,A3,A).

dec(s(X),X).

leq(o,_,true).
leq(s(_),o,false).
leq(s(X),s(Y),R) :- leq(X,Y,R).
```

Due to the definition of `dec`, the generated function `takc` contains occurrences of "`$!`" in its right-hand side:

```
tak x y z = takc (leq x y) x y z

takc True x y z = z
takc False x y z =
  ((tak $! (tak $! (dec x)) y z) $! (tak $! (dec y)) z x) $!
     (tak $! (dec z)) x y

dec (S x) = x

leq O _ = True
leq (S _) O = False
leq (S x) (S y) = leq x y
```

A.3 ackermann

The Ackermann function is also defined as a Prolog predicate on Peano numbers, as presented in [36]:

```
ackermann(o,N,s(N)).
ackermann(s(M),o,Val) :- ackermann(M,s(o),Val).
ackermann(s(M),s(N),Val) :-
  ackermann(s(M),N,Val1), ackermann(M,Val1,Val).
```

It is translated into the Curry function

```
ackermann O n = S n
ackermann (S m) O = ackermann m (S O)
ackermann (S m) (S n) = ackermann m (ackermann (S m) n)
```

A.4 numleaves

The predicate `numleaves` relates a binary tree with the number of its leaves in Peano representation:

```
plus(o,Y,Y).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).

numleaves(leaf(_),s(o)).
numleaves(node(M1,M2),s(N)) :-
        numleaves(M1,N1), numleaves(M2,N2), plus(N1,N2,N).
```

This is translated into the Curry functions

```
plus O y = y
plus (S x) y = S (plus x y)

numleaves (Leaf _) = S O
numleaves (Node m1 m2) = S (plus (numleaves m1) (numleaves m2))
```

A.5 permsort

The permutation sort example computes permutations by non-deterministically inserting an element into a list.

```
% Non-deterministic list insertion:
insert(X,[],[X]).
insert(X,[Y|Ys],[X,Y|Ys]).
insert(X,[Y|Ys],[Y|Zs]) :- insert(X,Ys,Zs).

% Permutations:
perm([],[]).
perm([X|Xs],Zs) :- perm(Xs,Ys), insert(X,Ys,Zs).

% less-or-equal relation
leq(o,_).
leq(s(X),s(Y)) :- leq(X,Y).

% Is the argument list sorted?
sorted([]).
sorted([_]).
sorted([X,Y|Ys]) :- leq(X,Y), sorted([Y|Ys]).

% Permutation sort: search for some sorted permutation
psort(Xs,Ys) :- perm(Xs,Ys), sorted(Ys).
```

The generated operations `insert` and `perm` are totally defined, whereas `leq`, `sorted`, and `psort` might fail. Due to the strict left-to-right semantics of the predefined conjunction operator "`&&`", insertions of the strict application operator "`$!`" in the third rule of `sorted` are not necessary.

```
insert x [] = [x]
insert x (y : ys) = x : (y : ys)
insert x (y : ys) = y : insert x ys

perm [] = []
perm (x : xs) = insert x (perm xs)

leq O _ = True
leq (S x) (S y) | leq x y = True

sorted [] = True
sorted [_] = True
sorted (x : y : ys) | leq x y && sorted (y : ys) = True

psort xs | sorted ys = ys
  where ys = perm xs
```