Improving Logic Programs by Adding Functions

Michael Hanus

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

Abstract. Logic programming is based on defining relations. Functions are often considered as syntactic sugar which can be transformed into predicates so that their logic is not used for computational purposes. In this paper, we present a method to use functions to improve the operational behavior of logic programs without loosing the flexibility of logic programming. For this purpose, predicates and goals are transformed into functions and nested expressions. By evaluating these functions in a demand-driven manner wherever possible and taking potential failures into account, we ensure that the execution of the transformed programs will never require more steps than the original programs but can decrease the number of steps—in the best case reducing infinite search spaces to finite ones. Thus, we obtain a systematic method to improve the operational behavior of logic programs without changing their semantics.

1 Introduction

Logic programming supports flexible programming techniques by built-in non-determinism and free (logic) variables. Predicates can be called with unknown arguments so that there are no fixed input and output positions, in contrast to functional languages. Since functions can be represented as predicates by adding the result as a parameter, logic programming is often considered as the more expressive programming paradigm [36].

Due to these considerations, functions and nested function calls are considered as nice syntactic sugar which can be eliminated by translating them into predicates and flattening nested expressions [8,12,32]. A consequence of this traditional view is the fact that one does not gain any operational advantage by the presence of functions. Since it is well known in functional programming that demand-driven (lazy) evaluation supports new programming techniques, as computing with infinite data structures or modularity [25], there are also approaches to translate functions with a demand-driven evaluation strategy by exploiting coroutining in Prolog [12,32]. Since coroutining might influence completeness due to flounderung, these approaches have an ad-hoc flavor—they are useful for particular examples, but general correctness results (soundness, completeness) are not provided.

To improve this situation, one could also take the opposite way. Instead of transforming functions into predicates, one could transform logic programs into programs of a language with another operational semantics. For instance, Van

Roy and Haridi [42] proposed to translate Prolog programs into Oz programs where advanced features of Oz for concurrent and distributed computations are used. Although this is an interesting approach to move logic programs techniques into the distributed world, an operational improvement is not obtained by this transformation. This is different in [19] where various transformations of logic programs into functional logic programs are proposed. Functional logic languages, such as Curry [22], offer the same flexibility as logic languages (soundness and completeness w.r.t. computing with partial information). In addition, optimality properties are known for well-defined classes of programs (minimal number of computed solutions, minimal number of evaluation steps [2]). The latter properties are due to the demand-driven evaluation of functions, i.e., it is essential to keep nested functional expressions instead of flattening them.

Due to these results, one could transform logic programs into functional logic programs. However, there is still one obstacle. As shown in [19], this transformation yields equivalent computations (identical answers, same number of computation steps) if the functional logic program is evaluated with an eager (strict) strategy. When a lazy strategy is used, the transformed programs might be more efficient but there are also cases where they compute answers which are not justified by the logic program. This could be the case if some subgoal fails in the logic program but they are not evaluated (due to laziness) in the transformed program. Thus, the transformation might obtain more efficient programs but with logically different answers.

In this paper, we want to close this gap by incorporating the failure behavior into the transformation. Due to this improvement, we obtain a method to transform logic programs with the following properties.

- The transformed programs always compute the same or more general answers to a given goal.
- The evaluation of the transformed programs is guaranteed equal or better than the evaluation of the original programs: in the worst case, the same number of steps are performed, but there are also cases where less steps are required to compute the result.
- As a consequence, there are also cases where infinite search spaces are reduced to finite ones.

Sloppily speaking, the improvements introduced by our transformation are comparable to "green" cuts in logic programming.

To obtain this result, we extend the transformation of [19] by deciding the eager or lazy evaluation of operations based on potential failures. For this purpose, we carefully analyze the problem of different answers between an eager or lazy evaluation of transformed programs. Then we use an approximation of the totality property of the transformed functions to explicitly enforce strict evaluations at some points in the generated programs. If a subexpression is ensured to be totally defined, it can be evaluated in a demand-driven manner, otherwise its evaluation is enforced even if the result is not immediately demanded.

This paper is structured as follows. After sketching the basics of logic and functional logic programming, we review in Sect. 3 the existing method to transform logic into functional logic programs. Section 4 discusses the incorporation of failure information to obtain a transformation which is correct w.r.t. the logical consequences of the logic program. In order to improve the transformation, we show in Sect. 5 how the addition of types can help to deduce a more precise approximation of potentially failing computations. Section 6 sketches the implementation of our approach which is evaluated in Sect.7. Section 8 discusses related work before we conclude.

2 Logic and Functional Logic Programming

In the following we briefly review some notions and features of logic and functional logic programming. More details can be found in [29] and in surveys on functional logic programming [5,17].

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[1] (*mgu*) σ of L_1 and L, then $G \vdash_\sigma \sigma(B, L_2, \ldots, L_k)$ is a *resolution* step. We denote by $G_1 \vdash_\sigma^* G_m$ a sequence $G_1 \vdash_{\sigma_1} G_2 \vdash_{\sigma_2} \ldots \vdash_{\sigma_{m-1}} G_m$ of resolution steps with $\sigma = \sigma_{m-1} \circ \ldots \circ \sigma_1$. A *computed answer* for a goal G is a substitution σ (restricted to the variables occurring in G) with $G \vdash_\sigma^* \square$.

Example 1. The following logic program defines a predicate `plus`, which relates two natural numbers in Peano representation, where `o` represents zero and `s` represents the successor of a natural [41] to its sum, a predicate `plus3`, which relates three natural numbers to its sum, and a predicate `thirdof` which is satisfied if the second argument is a third of the first one.

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

plus3(X,Y,Z,R) :- plus(X,Y,XY), plus(XY,Z,R).

thirdof(X,Y) :- plus3(Y,Y,Y,X).
```

For the goal `thirdof(o,T)`, the answer $\{$`T` $\mapsto$ `o`$\}$ is computed but, after showing this answer, Prolog does not terminate due to an infinite search space (since it enumerates arbitrary large values for `Y`).

[1] Substitutions, variants, and unifiers are defined as usual [29].

Functional logic programming [5,17] 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 [22], which we use in this paper, is a functional logic language with advanced programming concepts. Its syntax is close to Haskell [35], 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.

Example 2. The following Curry program[2] defines the operations of Example 1 in a functional manner, where logic features (like the free variable `y`) are exploited to define `thirdof`:

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

plus3 x y z = plus (plus x y) z

thirdof x | x =:= plus3 y y y
          = y
```

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

Since `plus` can be called with arguments containing free variables, the condition in the definition of `thirdof` is solved by instantiating `y` to appropriate *values* (i.e., expressions without defined functions) before reducing a function call. This corresponds to narrowing [37,40]. $t \rightsquigarrow_\sigma t'$ 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 $t' = \sigma(t[r]_p)$,[3] 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 also 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 [25] and can avoid superfluous computations so that it is optimal w.r.t. the number of computed solutions and the length of derivation (see [2] for precise statements). The latter property is our motivation to transform logic programs into Curry programs, since this can reduce infinite search spaces to finite ones. For instance, the evaluation of the expression `thirdof O` has a finite computation space: the generation of larger

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

[3] We use common notations from term rewriting [6].

numbers for the first argument of `plus3` is avoided since there is no demand for such numbers.

Curry has many more features which are useful to implement applications, like *set functions* [4] to encapsulate search, and standard features from functional programming, like modules and monadic I/O [43]. However, the kernel of Curry described so far should be sufficient to understand the remaining contents.

3 From Logic to Functional Logic Programs

Due to the fact that functional logic programming is an extension of pure logic programming, there is a simple way to transform logic into functional logic programs by mapping each predicate into a Boolean function and each clause into a (conditional) equation. This *conservative transformation* [19] 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.

In order to exploit the computational power of functional logic languages, one has to transform predicates into non-Boolean functions by selecting some arguments as results and generating function definitions according to this selection.

Example 3. Consider the predicate `plus` defined in Example 1. If the third argument is selected as a result argument (as often intended in logic programs), the clauses of `plus` can be transformed into the following functional logic program:

```
plus O y = y
plus (S x) y | z =:= plus x y = S z
```

In principle, any set of argument positions can be selected as results, as discussed in [19]. For instance, if the first two arguments of `plus` are selected as result arguments, the clauses of `plus` can be transformed into the following program:

```
plus y = (O, y)
plus (S z) | (x,y) =:= plus z = (S x, y)
```

Although this is not a function in a mathematical sense, it is a valid definition in Curry: it defines a *non-deterministic operation* which might deliver more than one result for a given argument. For a given natural number n, it returns all splittings into two numbers such that their sum is equal to n:

```
>  plus (S (S O))
(S (S O), O)
(S O, S O)
(O, S (S O))
```

Non-deterministic operations, which can formally be interpreted as mappings from values into sets of values [14], are an important feature of contemporary functional logic languages. The admissibility of non-deterministic operations provides more freedom when transforming logic programs into functional logic programs.

It is shown in [19] 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. In order to get a real advantage, one has to replace the unification occurring in conditions by *let* bindings whenever possible[4] and inline these bindings if reasonable. For instance, one can transform the rule

```
plus (S x) y | z =:= plus x y = S z
```

into

```
plus (S x) y = let z = plus x y in S z
```

and inline the binding of `z` into

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

This *demand functional transformation* is described in detail in [19]. 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 [24,25]. For instance, consider the predicate `signat` defined by the clauses

```
signat(o,o).
signat(s(X),s(o)).
```

Its demand functional transformation is[5]

```
signat O     = O
signat (S x) = S O
```

Now consider the evaluation of the expression `signat (plus` n_1 n_2`)`, where n_1 is a big natural number. An eager evaluation requires $n_1 + 1$ rewrite steps, whereas a non-strict language needs only two 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 "decrement" predicate

```
dec(s(X),X).
```

and its application in the predicate

```
sig1(R) :- dec(o,X), plus(s(o),X,Y), signat(Y,R).
```

[4] 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 [19] for a precise discussion.

[5] If it is not explicitly mentioned, the last argument of a predicate is considered as the result argument.

Due to the failure of the first subgoal, the goal "`?- sig1(R).`" fails. However, the demand functional transformation yields

```
dec (S x) = x
sig1 = signat (plus (S O) (dec O))
```

so that a lazy evaluation of `sig1` yields the value `S O`.

One could argue that the latter result value is intended due to the mathematical principle of "replacing equals by equals." This point of view is taken in [19]. However, if we intend to use the transformation of logic programs into functional logic programs to obtain more efficient programs without changing the set of computed answers, this difference is not acceptable. Therefore, we develop in the next sections a slightly different transformation which does not change the answer behavior (except for computing more general answers) but improves the operational behavior in many cases.

4 Incorporating Failure Information

We have seen in the previous section that the demand functional transformation might yield a different answer behavior if functions generated by the transformation are partially defined, i.e., fail to compute a value for particular argument values. We call such functions *failing*. The reason could be a pattern matching failure as well as an infinite loop. Since failures due to infinite loops are seldom, we later concentrate on the approximation of pattern matching failures.

If all functions occurring in an evaluation are totally defined, i.e., always non-failing, then a demand-driven evaluation strategy, like needed narrowing, computes only more general answers compared to an eager strategy, as shown in the following result.

Theorem 1 (Correctness of the demand functional transformation). *Let P be a logic program and G be a goal. Let F be the functional logic program and e the expression obtained by applying the demand functional transformation to P and G, respectively.*

1. *If there is a resolution derivation $G \vdash^*_\sigma \square$ w.r.t. P, then there is a needed narrowing derivation $e \overset{*}{\leadsto}_{\sigma'}$ `True` w.r.t. F and a substitution φ with $\sigma = \varphi \circ \sigma'$.*
2. *If all functions in F are totally defined and there is a needed narrowing derivation $e \overset{*}{\leadsto}_\sigma$ `True` w.r.t. F, then there is a resolution derivation $G \vdash^*_{\sigma'} \square$ w.r.t. P and a substitution φ with $\sigma' = \varphi \circ \sigma$.*

Proof. We sketch the proof which is a direct consequence of the soundness and completeness of needed narrowing [1,2].

If there is a resolution derivation $G \vdash^*_\sigma \square$ w.r.t. P, then there is an innermost narrowing derivation $e \overset{*}{\leadsto}_\sigma$ `True` w.r.t. F computing the same answer [19]. By soundness of narrowing, σ is a solution of the Boolean expression e w.r.t. F.

By completeness of needed narrowing, there is a needed narrowing derivation $e \overset{*}{\leadsto}_{\sigma'}$ `True` w.r.t. F and a substitution φ with $\sigma = \varphi \circ \sigma'$.

If there is a needed narrowing derivation $e \overset{*}{\leadsto}_{\sigma}$ `True` w.r.t. F, then, by soundness of needed narrowing, there is a rewriting of $\sigma(e)$ to `True`. This is not necessarily an innermost (eager) rewriting, i.e., there are some function calls in this rewrite sequence which might not be rewritten since their values are not needed. In an eager evaluation, all these function calls are evaluated. Due to the assumption that all functions are totally defined, they can always be evaluated to some value. An eager narrowing derivation might also bind some argument variables which were not bound in the needed narrowing evaluation. Thus, innermost narrowing might compute a more instantiated answer σ'. By [19], there is a resolution derivation $G \vdash^{*}_{\sigma'} \square$ w.r.t. P. $\square$

For instance, consider the expression `signat (plus x x)` where `x` is a free variable. Needed narrowing computes only the answer substitutions $\{$`x` $\mapsto$ `O`$\}$ for the result `O` and $\{$`x` $\mapsto$ `S _`$\}$ for the result `S O`, whereas innermost narrowing computes an infinite set of answer substitutions, i.e., $\{$`x` $\mapsto n\}$ for every natural number n. The same infinite set of answers is computed for the corresponding Prolog program.

In order to safely use the demand functional transformation w.r.t. the needed narrowing strategy, one has to ensure that all functions occurring in the evaluation steps are totally defined. However, one has not to give up in the presence of possibly failing operations since one only has to enforce the evaluation of such operations if they occur in the right-hand side of a rule, since the corresponding predicates are also evaluated in the logic program. This can be obtained by the following modification of the 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, then this application is replaced by

f `$!` e

We call this modified transformation *fail-sensitive functional transformation*. The predefined infix operator "`$!`" denotes a function application with a strict evaluation of the argument. Thus, if the evaluation of e fails, the evaluation of f `$!` e fails.

For instance, the fail-sensitive functional transformation maps the predicate `sig1` defined in the previous section into

```
sig1 = signat $! (plus (S O) $! (dec O))
```

so that the evaluation of `sig1` leads to a failure due to the enforced evaluation of `(dec O)`. Note that the usage of "`$!`" is necessary at all places where a potentially failing expression occurs and not only where the failing expressions occurs first. For instance, the slightly modified expression

```
signat (S (plus (S O) $! (dec O)))
```

evaluates to `S O` when evaluated by needed narrowing, whereas

```
signat $! (S $! (plus (S O) $! (dec O))
```

fails, similarly to the corresponding logic program.

In order to perform our transformation, we need to know whether an operation is totally defined. Obviously, this is undecidable in general so that some approximation is required. We can split this property into two parts: termination and non-occurrence of failures. Termination of rewrite systems or functional programs is a well-studied topic so that various techniques are available to approximate this property, e.g., [13,28]. Actually, the Curry analysis framework CASS [21] also provides an analysis to approximate the termination of functions. Therefore, we concentrate the subsequent discussion to the problem to approximate fail-freeness.

An operation is called *fail-free* [18] if its evaluation will never run into an explicit failure (due to a pattern-match error). A recent technique to approximate and verify fail-freeness is a type-based approximation of the arguments describing values for which a function call does not fail [20]. For this purpose, sets of concrete argument values are approximated by some abstract type. A simple but practically useful approximation are depth-k abstractions [38]. For the case $k = 1$, depth-1 types are described by subsets of constructors representing all terms having one of these constructors at the root. Moreover, the special value $\top$ describes the set of all constructors occurring in the program (a refinement based on type information will be discussed later). A *call type* of an n-ary function f is sequence $\alpha_1, \ldots, \alpha_n$ of abstract types such that any evaluation of a call $f\, t_1 \ldots t_n$ does not fail if t_i belongs to the set represented by α_i ($i = 1, \ldots, n$). For instance, $\{\texttt{S}\}$ is a call type of the operation `dec` ($\varnothing$ woul be another, less precise call type), and $\{\texttt{O}, \texttt{S}\}, \top$ is a call type of `plus`. Since $\{\texttt{O}, \texttt{S}\}$ are the only constructors in this program, we can also write this call type as $\top, \top$. A *trivial call type* is a sequence of $\top$s.

A method to infer call types is described in [20]. It is based on approximating the input/output behavior of operations and using this information to infer call types by considering the patterns and the operations occurring in the right-hand sides of the rules. Since these operations might demand for refined call types, the entire inference is a global fixpoint computation. For instance, the initial call type of

```
plusD x y = plus x (dec y)
```

is $\top, \top$ (since there are no restrictions in the left-hand side due to patterns). The call `(dec y)` demands that the second argument has type $\{\texttt{S}\}$ so that the initial call type is refined to $\top, \{\texttt{S}\}$. With this call type, `plusD` can be proved as fail-free.

An operation can have a trivial call type even if it calls potentially failing operations. For instance, consider the operation

```
dPlusS n = dec (plus (S n) n)
```

The tool described in [20] infers the trivial call type for `dPlusS` since the first argument of `plus` ensures that `dec` is always called with an `S`-rooted term.

The practical evaluation shows that only a few operations have non-trivial call types in larger programs since most operations are defined by complete pattern matchings on their input type. However, this demands for the consideration of the intended types of operations, as discussed next.

5 Type-Based Translation

Although Curry and other contemporary functional logic languages (e.g., TOY [31]) are strongly typed, type information is not considered in the fail-sensitive functional transformation. In principle, this is not necessary since types of functions are automatically inferred. Since data constructors need to be declared, the transformation adds a single definition of a type `Term` containing all constructors occurring in the program. For instance, the transformation of the previous examples using natural numbers in Peano representation adds the type declaration

```
data Term = O | S Term
```

If we extend the previous logic programs by a definition of a predicate to relate a binary tree with the number of its leaves, like

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

then the following data type is generated:

```
data Term = O | S Term | Leaf Term | Node Term Term
```

Since it is not ensured that `plus` is called only with natural numbers as arguments, the call type inferred for `plus` is no longer trivial. This is correct since the call `plus (Leaf O) O` is allowed and fails.

In order to improve this situation, we extend the transformation by the inclusion of type declarations. Although there is some interest to add types to Prolog programs [39], there is no general agreement about its syntax and structure. For instance, CIAO-Prolog [23] allows the definition of regular and Hindley-Milner types, or [7] describes a tool to infer types from logic programs. For our purpose, we support the explicit definition of polymorphic algebraic data types in Prolog syntax, like

```
:- type nat = o ; s(nat).
:- type tree(A) = leaf(A) ; node(tree(A),tree(A)).
```

These will be translated into definitions in Curry syntax:

```
data Nat = O | S Nat
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

With these definitions, the following types are inferred for the transformed operations `plus` and `numleaves`:

```
plus :: Nat → Nat → Nat
numleaves :: Tree a → Nat
```

Based on these types, both operations have trivial call types so that they are fail-free. This demonstrates that type information is useful to improve the transformation of logic programs into functional logic programs. In principle, it is not necessary. In the worst case, missing type information might have the effect that all operations are potentially failing so that the translated program is eagerly evaluated without any advantage compared to the original logic program.

6 Implementation

To evaluate our approach, we have extended the transformation tool presented in [19] by the inclusion of type and failure information.[6] Types can be declared in the source program as discussed in the previous section. They are directly mapped into corresponding data definitions in Curry. The remaining constructors occurring in the logic program but not mentioned in a type declaration are declared in a single `Term` type, as shown in the previous section.

Predicates are mapped into Curry functions without type annotations so that type signatures are inferred by the Curry system. The entire transformation process is performed in the following steps:

1. The given logic program is translated into a Curry program based on the demand functional transformation. The result argument positions are either inferred using a heuristic discussed in [19] or they can also be explicitly declared in the given logic program.
2. The generated functional logic program is analyzed with the tool described in [20] to check which operations are fail-free.[7]
3. Using the information computed in the previous step, the given logic programs is translated into a Curry program based on the fail-sensitive functional transformation. Thus, the same program structure is generated but the standard function application f e is replaced by f `$!` e whenever the expressions e might fail, i.e., contains an operation which is not fail-free. Note that f might be an expression, e.g., another application. For instance, if f is a binary operation and e_1 as well as e_2 contain operations with non-trivial call types, then an application $f\ e_1\ e_2$ generated in step 1 is replaced by `(`f `$!` e_1`)` `$!` e_2.[8] If all operations in e_2 have trivial call types, the application `(`f `$!` e_1`)` e_2 is generated.

7 Evaluation

The main motivation for this work is to show that functional logic programs have concrete operational advantages compared to pure logic programs. In principle,

[6] 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 with all required tools is also available as a docker image at `https://hub.docker.com/r/currylang/prolog2curry`.

[7] The inclusion of automated termination checks is currently omitted since it is seldom that operations generated from Prolog are completely defined but non-terminating.

[8] Note that this might change the order of evaluation if both arguments are demanded by f, but this order is not relevant in a declarative language without side effects.

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
`thirdof_0`	∞	∞	0.01
`signat_plus_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

this has already been shown in [19] but that approach is based on changing the semantics of programs from strict to demand-driven evaluation. As discussed above, this could have the effect that the transformed program computes *more* answers than the original logic program. By incorporating failure information, our new transformation has not this effect but compute more general answers compared to the answers of the logic program. For instance, the Prolog goal

```
?- plus(X,o,N), signat(N,s(o)).
```

yields an infinite set of answers where `X` and `N` are bound to all non-zero natural numbers, whereas the equivalent Curry expression obtained by our transformation

```
> signat (plus x O) =:= S O
```

has a finite search space and yields the single answer substitution $\{$`x` $\mapsto$ `S _`$\}$.

In the following, we want to show the practical advantages of this transformation by various examples. Note that these examples 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 functional dependencies) or can be reformulated in a declarative manner (e.g., declarative monadic I/O, state monads). Due to these reasons, functional logic programs have advantages from the view of software construction—which is less tangible to formalization. Therefore, we evaluate the advantages of functional logic programming only from the operational point of view.

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 [9].[9] The direct comparison of original and transformed programs is not straightforward

[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.

since the execution times depend on the implementation techniques used in the Prolog and Curry systems. KiCS2 compiles Curry programs into Haskell programs that are compiled to machine code by the Glasgow Haskell Compiler (GHC 9.4.5). GHC generates efficient code for functional computations. This is visible in the first three benchmarks which 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, used in various benchmarks [34] for logic and functional languages, to the values `(27,16,8)` in Peano representation. Similarly, the Ackermann function is defined on Peano numbers and applied to the Peano representation of `(3,9)`. The definition of these functions can be found in the appendix. It is interesting to note that the definition of `tak` decrements some arguments based on the predicate `dec` defined in Sect. 3. Since the generated function `dec` might fail, our transformation inserts the strict application operator "`$!`" in several places in the body of `tak`. These operators cause some overhead[10] and are not really necessary since `tak` is strict in its first argument. With a strictness analysis, one could drop these operators. Although this is not implemented in our tool, a manual optimization of the Curry code results in an execution time of 0.38 seconds. Thus, there is potential to improve the generated Curry code.

For the first three benchmarks, the demand-driven evaluation strategy has no real advantage since the values of all subexpressions are required. Hence, the same steps, possibly in a different order, are performed in the Prolog and Curry programs. The situation is different in the next three benchmarks which use examples already discussed in this paper. `thirdof_O` is the evaluation of the literal `thirdof(o,T)` (see Sect.1), `signat_plus_O` is the evaluation of the goal shown at the beginning of this section, and `numleaves_7` is the evaluation of the literal `numleaves(T,s(s(s(s(s(s(s(o))))))))` having five solutions (compare Sect. 5). 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 of these goals, whereas the transformed Curry programs have a finite search space. 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 [5,17], the functional logic version of permutation sort has the effect that permutations are explored 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.

The fail-sensitive functional transformation produces functional logic programs providing the same or more general answers as the original logic programs. In the worst case, the same number of evaluation steps are performed (apart from a few additional steps to reduce occurrences of "`$!`" introduced due to non-total functions). In the best case, the transformation reduces infinite search spaces to

[10] The operator "`$!`" is not considered by the Curry compiler KiCS2 but implemented by a specific predefined operation.

finite ones. Thus, one does not get any disadvantage of this transformation but in some cases considerable advantages.

We want to remark that this improvement is not only of theoretical interest. As apparent from our examples, logic programs might have infinite search spaces when search is nested, i.e., if two predicates defined on infinite structures, like Peano numbers, lists, or trees, are sequentially called with unknown arguments, as in `plus3` or `numleaves`. When such programs are transformed into functional logic programs, a demand-driven evaluation, like needed narrowing, results in a demand-driven exploration of the search space. This is exploited in [16] to implement a domain-specific language for deep XML matching and transformation as a library in Curry. Without the demand-driven evaluation strategy, many of the library functions would not terminate. Actually, the library has similar features as the logic-based language Xcerpt [11] which uses a specialized unification procedure to ensure finite matching and unification of XML terms.

8 Related Work

The view that algorithms can be seen as declarative logic descriptions combined with appropriate control rules was introduced before decades [27]. Since the standard control rule of Prolog, left-to-right depth-first search implemented via backtracking, has the risk to loop infinitely instead of delivering an answer, there is also long history to modify the standard control rule in order to improve the operational behavior [10,33]. These proposals usually consider the operational level so that a declarative justification (soundness and completeness) is missing. This is in contrast to our work which is motivated to keep the soundness and completeness of logic programming and provide a better operational model.

There are also approaches to add functions and functional notations to logic programs and map them in various ways to logic programs, in particular, to Prolog systems with coroutining in order to have a similar effect as in a demand-driven computation model. However, one has to be careful since there are various ways and pitfalls to implement laziness in this way. For instance, [32] implements lazy evaluation in Prolog by representing closures as terms and use `when` declarations to delay insufficiently instantiated function calls. This might lead to floundering so that completeness is lost when predicates are transformed into functions. Moreover, delaying recursively defined predicates could result in infinite search spaces where a complete narrowing strategy has a finite search space [15]. A notation for functions in Prolog programs is also proposed in [12]. This syntactic extension is mapped into Prolog predicates where coroutining is used to implement lazy evaluation. Although this syntactic transformation might yield the same values and search space as functional logic languages, there are no formal results justifying this transformation.

Other approaches use Prolog as a target language to implement functional logic languages based on demand-driven narrowing strategies [3,26,30,31]. Since these approaches implement narrowing strategies, the soundness and completeness results of narrowing holds also for the generated logic programs. When implementing needed narrowing in this way [3], the resulting programs are optimal, i.e., the set of computed solutions is minimal and successful derivations have the shortest possible length [2].

In this paper we take an opposite approach since we use (pure) Prolog as the source language. This was already proposed in [19] but there it is not ensured that the set of answers is kept by the transformation, as discussed in Sect. 3. By respecting the failure behavior of the source programs, we fixed this problem and showed how to obtain an equal or better operational behavior without changing the answer behavior. The extension of this transformation to Prolog's built-in arithmetic and conditional goals is shown in [19] and also implemented in our current approach. Methods to infer result argument positions so that the transformed program is optimal (if possible) are also discussed in [19]. Since these methods can be used identically in our transformation, we omitted a detailed presentation of them in this paper.

9 Conclusions

We presented a method to transform logic programs into functional logic programs so that the transformed functional logic program always computes the same or more general answers than the original program. For a well-defined class of programs (predicates completely defined by inductively sequential rules), the set of answers and the number of evaluation steps leading to an answer are minimal [1,2]. Thus, the transformation is useful to transfer strong results from functional logic programming into logic programming.

To ensure the correctness of the transformation, we use an approximation of the failure behavior of transformed programs, which is improved by considering types provided for logic programs. As a result, the transformation yields programs that require the same number of evaluation steps in the worst case, but often require less steps and reduces infinite search spaces to finite ones. Thus, the main lesson learned from this work is that the introduction of functions and the usage of these dependencies in functional logic programs has many advantages but no disadvantages in general. We implemented our method in a tool and showed the practical advantages by evaluating a set of small but typical examples.

For future work it might be interesting to improve the presented fail-sensitive functional transformation by taking strictness information into account to avoid the introduction of strict application operations, as discussed in Sect. 7. However, the motivation of this work is not to provide a general applicable tool to transform Prolog programs, since this is difficult due to the use of non-declarative features in larger Prolog programs. Our intention is to show that the execution mechanism of functional logic programs is always equal or better than the resolution principle for pure logic programs thanks to exploiting functional dependencies. Although this could be simulated in Prolog systems with corouting, additional problems might occur due to floundering and incompleteness of executions.

References

1. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
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. 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.
5. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
6. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
7. J. Barbosa, M. Florido, and V. Santos Costa. Data type inference for logic programming. In *Proceedings of the 31st International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2021)*, pages 16–37. Springer LNCS 13290, 2021.
8. R. Barbuti, M. Bellia, G. Levi, and M. Martelli. On the integration of logic programming and functional programming. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 160–166, Atlantic City, 1984.
9. 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.
10. M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming (6)*, pages 135–162, 1989.
11. F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proceedings of the International Conference on Logic Programming (ICLP'02)*, pages 255–270. Springer LNCS 2401, 2002.
12. A. Casas, D. Cabeza, and M.V. Hermenegildo. A syntactic approach to combining functional notation, lazy evaluation, and higher-order in LP systems. In *Proc. of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, pages 146–162. Springer LNCS 3945, 2006.
13. 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.
14. 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.
15. M. Hanus. Analysis of residuating logic programs. *Journal of Logic Programming*, 24(3):161–199, 1995.
16. M. Hanus. Declarative processing of semistructured web data. In *Technical Communications of the 27th International Conference on Logic Programming*, volume 11, pages 198–208. Leibniz International Proceedings in Informatics (LIPIcs), 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. Verifying fail-free declarative programs. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP 2018)*, pages 12:1–12:13. ACM Press, 2018.
19. M. Hanus. From logic to functional logic programs. *Theory and Practice of Logic Programming*, 22(4):538–554, 2022.
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 and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, pages 181–188. ACM Press, 2014.
22. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-lang.org`, 2016.
23. M. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J.F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.
24. 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.
25. J. Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison Wesley, 1990.
26. J.A. Jiménez-Martin, J. Marino-Carballo, and J.J. Moreno-Navarro. Efficient compilation of lazy narrowing into Prolog. In *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'92)*, pages 253–270. Springer Workshops in Computing Series, 1992.
27. R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, 1979.
28. 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.
29. J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.
30. R. Loogen, F. López Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 184–200. Springer LNCS 714, 1993.
31. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
32. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.
33. S. Narain. A technique for doing lazy evaluation in logic. *Journal of Logic Programming*, 3:259–276, 1986.
34. W. Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202. Springer, 1992.

35. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
36. U.S. Reddy. Transformation of logic programs into functional programs. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 187–196, Atlantic City, 1984.
37. U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.
38. T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.
39. T. Schrijvers, V. Santos Costa, J. Wielemaker, and B. Demoen. Towards Typed Prolog. In *24th International Conference on Logic Programming (ICLP 2008)*, pages 693–697. Springer LNCS 5366, 2008.
40. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.
41. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 2nd edition, 1994.
42. P. Van Roy and S. Haridi. Ideas for the future of Prolog inspired by Oz. *CoRR*, abs/2302.00558, 2023.
43. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

A Benchmarks

This appendix shows the Prolog source code of some predicates used in the benchmarks 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, i.e., terms constructed from `o/0` and `s/1`:

```
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 [41]:

```
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 `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
```