TPLP **22** (4): 538–554, 2022. © The Author(s), 2022. Published by Cambridge University Press.

doi:10.1017/S1471068422000187 First published online 4 July 2022

From Logic to Functional Logic Programs

MICHAEL HANUS
Institut für Informatik, CAU Kiel, Kiel, Germany
(*e-mail:* mh@informatik.uni-kiel.de)

submitted 17 May 2022; accepted 8 June 2022

Abstract

Logic programming is a flexible programming paradigm due to the use of predicates without a fixed data flow. To extend logic languages with the compact notation of functional programming, there are various proposals to map evaluable functions into predicates in order to stay in the logic programming framework. Since amalgamated functional logic languages offer flexible as well as efficient evaluation strategies, we propose an opposite approach in this paper. By mapping logic programs into functional logic programs with a transformation based on inferring functional dependencies, we develop a fully automatic transformation which keeps the flexibility of logic programming but can improve computations by reducing infinite search spaces to finite ones.

KEYWORDS: functional logic programming, transformation, resolution, narrowing strategies

1 Motivation

Functional and logic programming are the most prominent declarative programming paradigms. Functional programming provides a compact notation, due to nested expressions, and demand-driven (optimal) evaluation strategies, whereas logic programming provides flexibility due to free variables, unification, and built-in search. Thus, both paradigms have their advantages for application programming so that it is reasonable to offer their features in a single language. One option to achieve this is to extend a logic language with functional syntax (equations, nested functional expressions) and transform evaluable functions into predicates and flatten nested expressions (Barbuti *et al.* 1984; Casas *et al.* 2006; Naish 1991). Hence, logic programming can be considered as the more general paradigm. On the other hand, functional programming supports efficient, in particular, optimal evaluation strategies, by exploiting functional dependencies during evaluation so that it provides more modularity and new programming concepts, like programming with infinite data structures (Hughes 1990). In Prolog systems which support coroutining, that is, delaying the evaluation of literals when arguments are not sufficiently instantiated, one can exploit coroutining to implement the basic idea of lazy evaluation (Casas *et al.* 2006; Naish 1991). However, the use of coroutines might add new problems. For instance, computations with coroutining introduce the risk of incompleteness by floundering (Lloyd 1987). Furthermore, coroutining might yield infinite search spaces due to delaying literals generated by recursively defined predicates (Hanus 1995).

Although the operational behavior of particular programs can be improved with coroutines, there are no general results characterizing classes of programs where this always leads to an improved operational behavior.

Amalgamated *functional logic languages* (Antoy and Hanus 2010) are an approach to support flexible as well as efficient evaluation strategies. For instance, the language Curry (Hanus 2016) is based on an optimal and logically sound and complete strategy (Antoy *et al.* 2000; Antoy 1997). The motivation of this paper is to show that functional logic languages are actually superior to pure logic languages. Instead of transforming functions into predicates, we present a sequence of transformations which map logic programs into functional logic programs. Using a sophisticated mapping based on inferring functional dependencies, we obtain a fully automatic transformation tool which can reduce the computational efforts. In particular, infinite search spaces w.r.t. logic programs can be reduced to finite ones w.r.t. the transformed functional logic programs.

This paper is structured as follows. The next section reviews basic notions of logic and functional logic programming. Section 3 presents a simple embedding of logic into functional logic programs, which is extended in Sections 4 and 5 by considering functional dependencies. Section 6 discusses the inference of such dependencies and Section 7 adds some extensions. This final transformation is implemented in a tool sketched in Section 8. We evaluate our transformation in Section 9 before we conclude. Due to lack of space, some details and proofs are omitted. They are available in a longer version of this paper.[1]

2 Logic and functional logic programming

In this section, we fix our notation for logic programs and briefly review some relevant features of functional logic programming. More details are in the textbook of Lloyd (1987) and in surveys on functional logic programming (Antoy and Hanus 2010; Hanus 2013).

In logic programming (we use Prolog syntax for concrete examples), *terms* 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 ($k = 0$). Predicates are defined by *clauses* L `:-` B, 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[2] (*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 $\sigma = \sigma_{m-1} \circ \cdots \circ \sigma_1$. A *computed answer* for a goal G is a substitution σ (usually restricted to the variables occurring in G) with $G \vdash^*_\sigma \square$.

[1] `https://arxiv.org/abs/2205.06841`.

[2] Substitutions, variants, and unifiers are defined as usual (Lloyd 1987).

Example 1
The following logic program defines the well-known predicate `app` for list concatenation, a predicate `app3` to concatenate three lists, and a predicate `dup` which is satisfied if the second argument occurs at least two times in the list provided as the first argument:

```
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).

dup(Xs,Z) :- app3(_,[Z|_],[Z|_],Xs).
```

The computed answers for the goal `dup([1,2,2,1],Z)` are $\{\texttt{Z} \mapsto \texttt{1}\}$ and $\{\texttt{Z} \mapsto \texttt{2}\}$. They can be computed by a Prolog system, but after showing these answers, Prolog does not terminate due to an infinite search space. Actually, Prolog does not terminate for the goal `dup([],Z)` since it enumerates arbitrary long lists for the first argument of `app3`.

Functional logic programming (Antoy and Hanus 2010; Hanus 2013) integrates the most important features of functional and logic languages in order to provide a variety of programming concepts. Functional logic languages support higher order functions and lazy (demand-driven) evaluation from functional programming as well as nondeterministic search and computing with partial information from logic programming. The declarative multiparadigm language Curry (Hanus 2016), which we use in this paper, is a functional logic language with advanced programming concepts. Its syntax is close to Haskell (Peyton Jones 2003), that is, 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 nondeterministic instantiation of arguments.

Example 2
The following Curry program[3] defines the operations of Example 1 in a functional manner, where logic features (free variables `z` and `_`) are exploited to define `dup`:

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

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

dup xs | xs =:= app3 _ (z:_) (z:_)
       = z
```

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

Since `app` can be called with free variables in arguments, the condition in the definition of `dup` is solved by instantiating `z` and the anonymous free variables "`_`" to appropriate *values* (i.e., expressions without defined functions) before reducing the function calls. This corresponds to narrowing (Reddy 1985; Slagle 1974). $t \leadsto_\sigma t'$ is a *narrowing step* if there is some nonvariable position p in t, an equation (program rule) l = r, and an

[3] We simplify the concrete syntax by omitting the declaration of free variables, like `z`, which is required in concrete Curry programs to enable some consistency checks by the compiler.

mgu σ of $t|_p$ and l such that $t' = \sigma(t[r]_p)$,[4] that is, t' is obtained from t by replacing the subterm $t|_p$ by the equation's right-hand side and applying the unifier. This definition also applies to conditional equations l `|` c `=` r which are considered as syntactic sugar for the unconditional equation l `=` c `&>` r, where the operation "`&>`" is defined by `True &> x = x`.

Curry is based on the *needed narrowing strategy* (Antoy *et al.* 2000) which also uses non-most-general unifiers in narrowing steps to ensure the optimality of computations. Needed narrowing is a demand-driven evaluation strategy, that is, it supports computations with infinite data structures (Hughes 1990) and can avoid superfluous computations. 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 `dup []` has a finite computation space: the generation of longer lists for the first argument of `app3` is avoided since there is no demand for such lists.

`dup` is a *nondeterministic operation* since it might deliver more than one result for a given argument, for example, the evaluation of `dup [1,2,2,1]` yields the values `1` and `2`. Nondeterministic operations, which can formally be interpreted as mappings from values into sets of values (González-Moreno *et al.* 1999), are an important feature of contemporary functional logic languages. For the transformation described in this paper, this feature has the advantage that it is not important to transform predicates into purely mathematical functions (having at most one result for a given combination of arguments).

Curry has many more features that are useful for application programming, like *set functions* (Antoy and Hanus 2009) to encapsulate search, and standard features from functional programming, like modules or monadic I/O (Wadler 1997). However, the kernel of Curry described so far should be sufficient to understand the transformation described in the following sections.

3 Conservative transformation

Functional logic programming is an extension of pure logic programming. Hence, there is a straightforward way to map logic programs into functional logic programs: map each predicate into a Boolean function and transform each clause into a (conditional) equation. We call this mapping the *conservative transformation* since it keeps the basic structure of derivations. Since narrowing-based functional logic languages support free variables as well as overlapping rules, this mapping does not change the set of computed solutions (in contrast to a purely functional target language where always the first matching rule is selected).

As a first step to describe this transformation, we have to map terms from logic into functional logic notation. Since terms in logic programming (here: Prolog syntax) have a direct correspondence to data terms (as used in Curry), the mapping of terms is just a change of syntax (e.g., uppercase variables are mapped into lowercase, and lowercase constants and constructors are mapped into their uppercase equivalents). We denote this *term transformation* by $[\![\cdot]\!]_{\mathscr{T}}$, that is, $[\![\cdot]\!]_{\mathscr{T}}$ is a syntactic transformation which maps a

[4] We use common notations from term rewriting (Baader and Nipkow 1998; TeReSe 2003).

(Prolog) term, written inside the brackets, into a (Curry) data term. It is defined by a case distinction as follows:

$$\begin{array}{rcll} [\![X]\!]_{\mathscr{T}} & = & x & \text{(variable)} \\ [\![n]\!]_{\mathscr{T}} & = & n & \text{(number constant)} \\ [\![c]\!]_{\mathscr{T}} & = & C & \text{(atom constant)} \\ [\![f(t_1,\ldots,t_n)]\!]_{\mathscr{T}} & = & F\ [\![t_1]\!]_{\mathscr{T}}\ \ldots\ [\![t_1]\!]_{\mathscr{T}} & \text{(constructed term)} \end{array}$$

Based on this term transformation, we define a mapping $[\![\cdot]\!]_{\mathscr{C}}$ from logic into functional logic programs where facts and rules are transformed into unconditional and conditional equations, respectively (the symbol "`&&`" is an infix operator in Curry denoting the Boolean conjunction):

$$\begin{array}{rcll} [\![p(t_1,\ldots,t_n)]\!]_{\mathscr{C}} & = & p\ [\![t_1]\!]_{\mathscr{T}}\ \ldots\ [\![t_n]\!]_{\mathscr{T}} & \text{(literal)} \\ [\![l_1,\ldots,l_k]\!]_{\mathscr{C}} & = & [\![l_1]\!]_{\mathscr{C}}\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ [\![l_k]\!]_{\mathscr{C}} & \text{(goal)} \\ [\![l\,.]\!]_{\mathscr{C}} & = & [\![l]\!]_{\mathscr{C}}\ \texttt{= True} & \text{(fact)} \\ [\![l\ \texttt{:-}\ b.]\!]_{\mathscr{C}} & = & [\![l]\!]_{\mathscr{C}}\ \texttt{|}\ [\![b]\!]_{\mathscr{C}}\ \texttt{= True} & \text{(rule)} \\ [\![clause_1\ \ldots\ clause_m]\!]_{\mathscr{C}} & = & [\![clause_1]\!]_{\mathscr{C}}\ \ldots\ [\![clause_m]\!]_{\mathscr{C}} & \text{(program)} \end{array}$$

Example 3
Consider the following program to add two natural numbers in Peano representation, where `o` represents zero and `s` represents the successor of a natural (Sterling and Shapiro 1994):

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

The conservative transformation produces the following Curry program:

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

Note that the first rule would not be allowed in functional languages, like Haskell, since the left-hand side is not linear due to the two occurrences of the pattern variable `y`. For compatibility with logic programming, such multiple occurrences of variables in patterns are allowed in Curry, where they are considered as syntactic sugar for explicit unification constraints. Thus, the first rule is equivalent to

```
plus O y y' | y =:= y' = True
```

Apart from small steps to handle conditions and conjunctions, there is a strong correspondence between the derivations steps in the logic programs and the functional logic programs obtained by the conservative transformation. Therefore, the following result can be proved by induction on the length of the resolution and narrowing derivations, respectively.

Theorem 1 (*Correctness of the conservative transformation*)
Let P be a logic program and G a goal. There is a resolution derivation $G \vdash_{\sigma}^{*} \Box$ w.r.t. P if and only if there is a narrowing derivation $[\![G]\!]_{\mathscr{C}} \overset{*}{\rightsquigarrow}_{\sigma}$ `True` w.r.t. $[\![P]\!]_{\mathscr{C}}$.

4 Functional transformation

The conservative transformation simply maps n-ary predicates into n-ary Boolean functions. In order to exploit features from *functional* logic programming, one should mark at least one argument as a *result argument* with the intended meaning that the operation maps values for the remaining arguments into values for the result arguments. For instance, consider the predicate `plus` defined in Example 3. Here, the third argument could be considered as a result argument since `plus` maps values for the first two arguments into a value for the third argument. Hence, the definition of `plus` can also be transformed into the following functional logic program:

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

We call this the *functional transformation* and denote it by $[\![\cdot]\!]_{\mathscr{F}}$. At this point we do not replace the occurrence of `z` in the right-hand side by `plus x y` since this might lead to a different semantics, as we will see later.

It is interesting to note that, without such a replacement, it is not really relevant which arguments are considered as results. For instance, if the first two arguments of `plus` are marked as result arguments, we obtain the following program by the functional transformation:

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

This specifies a nondeterministic operation which returns, for a given natural number n, 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))
```

Later we will show how to prefer the transformation into deterministic operations, since they will lead to a better operational behavior. For the moment we should keep in mind that the selection of result arguments could be arbitrary. In order to fix it, we assume that, for each n-ary predicate p, there is an assignment $\mathscr{R}(p/n) \subseteq \{1,\ldots,n\}$ which defines the result argument positions, for example, $\mathscr{R}(\texttt{plus}/3) = \{3\}$ or $\mathscr{R}(\texttt{plus}/3) = \{1,2\}$. In practice, one can specify the result argument positions for a predicate by a specific directive, for example,

```
:- function plus/3: 3.
```

or

```
:- function plus/3: [1,2].
```

If the set of result argument positions for an n-ary predicate is $\{n\}$, it can also be omitted in the directive, as in

```
:- function plus/3.
```

Our tool, described below, respects such directives or tries to infer them automatically.

With these prerequisites in mind, we denote the *functional transformation w.r.t. a result argument position mapping* $\mathscr{R}$ by $[\![\cdot]\!]^{\mathscr{R}}_{\mathscr{F}}$. To define this transformation, we use the following notation to split the arguments of a predicate call $p(t_1,\ldots,t_n)$ into the result and the remaining arguments. If $\mathscr{R}(p/n) = \{\pi_1,\ldots,\pi_u\}$ and $\{\pi'_1,\ldots,\pi'_v\} = \{1,\ldots,n\} \setminus$

$\mathscr{R}(p/n)$ (where $\pi_i < \pi_{i+1}$ and $\pi'_j < \pi'_{j+1}$), then the result arguments are $r_i = t_{\pi_i}$, for $i \in \{1,\dots,u\}$, and the remaining arguments are $a_j = t_{\pi'_j}$, for $j \in \{1,\dots,v\}$. Then $[\![\cdot]\!]^{\mathscr{R}}_{\mathscr{F}}$ is defined as follows:

$$\begin{array}{rll}
[\![p(t_1,\dots,t_n)]\!]^{\mathscr{R}}_{\mathscr{F}} & = \texttt{(}\,[\![r_1]\!]_{\mathscr{T}}\,,\dots,[\![r_u]\!]_{\mathscr{T}}\,\texttt{)} \ \texttt{=:=} \ p\ [\![a_1]\!]_{\mathscr{T}} \dots [\![a_v]\!]_{\mathscr{T}} & \text{(literal, } u>0)\\
[\![p(t_1,\dots,t_n)]\!]^{\mathscr{R}}_{\mathscr{F}} & = p\ [\![t_1]\!]_{\mathscr{T}} \dots [\![t_n]\!]_{\mathscr{T}} & \text{(literal, } u=0)\\
[\![l_1,\dots,l_k]\!]^{\mathscr{R}}_{\mathscr{F}} & = [\![l_1]\!]^{\mathscr{R}}_{\mathscr{F}}\ \texttt{\&\&}\ \dots\ \texttt{\&\&}\ [\![l_k]\!]^{\mathscr{R}}_{\mathscr{F}} & \text{(goal)}\\
[\![p(t_1,\dots,t_n)\texttt{.}]\!]^{\mathscr{R}}_{\mathscr{F}} & = p\ [\![a_1]\!]_{\mathscr{T}} \dots [\![a_v]\!]_{\mathscr{T}}\ \texttt{=}\ \texttt{(}\,[\![r_1]\!]_{\mathscr{T}}\,,\dots,[\![r_u]\!]_{\mathscr{T}}\,\texttt{)} & \text{(fact, } u>0)\\
[\![p(t_1,\dots,t_n)\texttt{.}]\!]^{\mathscr{R}}_{\mathscr{F}} & = p\ [\![t_1]\!]_{\mathscr{T}} \dots [\![t_n]\!]_{\mathscr{T}}\ \texttt{=}\ \texttt{True} & \text{(fact, } u=0)\\
[\![p(t_1,\dots,t_n)\ \texttt{:-}\ b.]\!]^{\mathscr{R}}_{\mathscr{F}} & = p\ [\![a_1]\!]_{\mathscr{T}} \dots [\![a_v]\!]_{\mathscr{T}}\ \texttt{|}\ [\![b]\!]^{\mathscr{R}}_{\mathscr{F}} & \\
 & \quad \texttt{=}\ \texttt{(}\,[\![r_1]\!]_{\mathscr{T}}\,,\dots,[\![r_u]\!]_{\mathscr{T}}\,\texttt{)} & \text{(rule, } u>0)\\
[\![p(t_1,\dots,t_n)\ \texttt{:-}\ b.]\!]^{\mathscr{R}}_{\mathscr{F}} & = p\ [\![t_1]\!]_{\mathscr{T}} \dots [\![t_n]\!]_{\mathscr{T}}\ \texttt{|}\ [\![b]\!]^{\mathscr{R}}_{\mathscr{F}}\ \texttt{=}\ \texttt{True} & \text{(rule, } u=0)\\
[\![cls_1 \dots cls_m]\!]^{\mathscr{R}}_{\mathscr{F}} & = [\![cls_1]\!]^{\mathscr{R}}_{\mathscr{F}} \dots [\![cls_m]\!]^{\mathscr{R}}_{\mathscr{F}} & \text{(program)}
\end{array}$$

As already mentioned, the actual selection of result positions is not relevant so that we have the following result, which can be proved similarly to Theorem 1:

Theorem 2 (*Correctness of the functional transformation*)
Let P be a logic program, $\mathscr{R}$ a result argument position mapping for all predicates in P, and G a goal. There is a resolution derivation $G \vdash^*_\sigma \Box$ w.r.t. P if and only if there is a narrowing derivation $[\![G]\!]^{\mathscr{R}}_{\mathscr{F}} \overset{*}{\rightsquigarrow}_\sigma$ `True` w.r.t. $[\![P]\!]^{\mathscr{R}}_{\mathscr{F}}$.

5 Demand functional transformation

Consider the result of the functional transformation of `plus` w.r.t. the result argument position mapping $\mathscr{R}(\texttt{plus}/3) = \{3\}$:

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

Since the value of `z` is determined by the expression `plus x y`, we could be tempted to replace the unification in the condition by a local binding for `z`:

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

Since `z` is used only once, we could inline the definition of `z` and obtain the purely functional definition

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

Although this transformation looks quite natural, there is a potential problem with this transformation. In a strict language, where arguments are evaluated before jumping into the function's body ("call by value"), there is no difference between these versions of `plus`. However, there is also no operational advantage of this transformation. An advantage could come from the nonstrict or demand-driven evaluation of functions, as used in Haskell or Curry and discussed by Hughes (1990) and Huet and Lévy (1991). For instance, consider the predicate `isPos` which returns `True` if the argument is nonzero

```
isPos O     = False
isPos (S x) = True
```

and the expression `isPos (plus` n_1 n_2`)`, where n_1 is a big natural number. A strict language requires $n_1 + 1$ rewrite steps to evaluate this expression, whereas a nonstrict language needs only two steps w.r.t. the purely functional definition of `plus`.

The potential problem of this transformation comes from the fact that it does not require the evaluation of subexpressions which do not contribute to the overall result. For instance, consider the functions

```
dec (S x) = x                    const x y = x
```

and the expression $e =$ `const O (dec O)`. Following the mathematical principle of "replacing equals by equals", e is equivalent to `O`, but a strict language does not compute this value.

Hence, it is a matter of taste whether we want to stick to purely equational reasoning, that is, ignore the evaluation of subexpressions that do not contribute to the result, or strictly evaluate all subexpressions independent of their demand.[5] Since nonstrict evaluation yields reduced search spaces (as discussed below), we accept this slight change in the semantics and define the *demand functional transformation* $[\![\cdot]\!]^{\mathscr{R}}_{\mathscr{D}}$ as follows. Its definition is identical to $[\![\cdot]\!]^{\mathscr{R}}_{\mathscr{F}}$ except for the translation of a literal in a goal. Instead of a unification, $[\![\cdot]\!]^{\mathscr{R}}_{\mathscr{D}}$ generates a local binding `let/where (`$[\![r_1]\!]_{\mathscr{T}}, \ldots, [\![r_u]\!]_{\mathscr{T}}$`) =` $p\ [\![a_1]\!]_{\mathscr{T}} \ldots [\![a_v]\!]_{\mathscr{T}}$ if the result arguments $r_1, \ldots, r_u$ are variables which do not occur in the rule's left-hand side or in result arguments of other goal literals. The latter restriction avoids inconsistent bindings for a variable. Since such bindings are evaluated on demand in nonstrict languages, this change has the effect that the transformed programs might require fewer steps to compute a result.

In order to produce more compact and readable program, local bindings of single variables, that is, x `=` e, are *inlined* if possible, that is, if there is only a single occurrence of x in the rule, this occurrence is replaced by e and the binding is deleted. This kind of inlining is the inverse of the normalization of functional programs presented by Launchbury (1993) to specify a natural semantics for lazy evaluation.

Example 4

Consider the usual definition of naive reverse:

```
:- function app/3.
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

:- function rev/2.
rev([],[]).
rev([X|Xs],Zs) :- rev(Xs,Ys), app(Ys,[X],Zs).
```

The demand functional transformation translates this program into the Curry program

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

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

Thanks to the functional *logic* features of Curry, one can use the transformed program similarly to the logic program. For instance, the equation `app xs ys =:= [1,2,3]` computes all splittings of the list `[1,2,3]`, and `rev ps =:= ps` computes palindromes `ps`.

[5] Note that every reasonable programming language requires the nonstrict evaluation of conditional (if-then-else) expressions so that there is no completely strict language.

An advantage of this transformation becomes apparent for nested applications of recursive predicates. For instance, consider the concatenation of three lists, as shown in Example 1:

```
:- function app3/4.
app3(Xs,Ys,Zs,Ts) :- app(Xs,Ys,Rs), app(Rs,Zs,Ts).
```

The demand functional transformation maps it into

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

The Prolog goal `app3(Xs,Ys,Zs,[])` has an infinite search space, that is, it does not terminate after producing the solution `Xs=[],Ys=[],Zs=[]`. In contrast, Curry has a finite search space since the demand-driven evaluation avoids the superfluous generation of longer lists. This shows the advantage of transforming logic programs into functional logic programs: the operational behavior is improved, that is, the size of the search space could be reduced due to the demand-driven exploration of the search space, whereas the positive features, like backward computations, are kept.

A slight disadvantage of the demand functional transformation is the fact that it requires the specification of the result argument position mapping $\mathscr{R}$, for example, by explicit `function` directives. In the next section, we show how it can be automatically inferred.

6 Inferring result argument positions

As already discussed above, the selection of result arguments is not relevant for the applicability of our transformation. For instance, the predicate

```
p(a,c).
p(b,c).
```

could be transformed into the function

```
p A = C
p B = C
```

as well as into the nondeterministic operation

```
p C = A
p C = B
```

or just kept as a predicate:

```
p A C = True
p B C = True
```

In general, it is difficult to say what is the best representation of a predicate as a function. One could argue that deterministic operations are preferable, but there are also examples where nondeterministic operations lead to reduced search spaces. For instance, the complexity of the classical permutation sort can be improved by defining the computation of permutations as a nondeterministic operation (González-Moreno *et al.* 1999; Hanus 2013).

A possible criterion can be derived from the theory of term rewriting (Huet and Lévy 1991) and functional logic programming (Antoy *et al.* 2000). If the function definitions are *inductively sequential* (Antoy 1992), that is, the left-hand sides of the rules of each function contain arguments with a unique case distinction, the demand-driven evaluation (needed narrowing) is optimal in the number of computed solutions and the length of successful derivations (Antoy *et al.* 2000). In the following, we present a definition of this criterion adapted to logic programs.

In many logic programs, there is a single argument which allows a unique case distinction between all clauses, for example, the first argument in the predicates `app`, `rev`, or `plus` shown above. However, there are also predicates requiring more than one argument for a unique selection of a matching rule. For instance, consider Ackermann's function as a logic program, as presented by Sterling and Shapiro (1994):

```
ackermann(o,N,s(N)).
ackermann(s(M),o,V) :- ackermann(M,s(o),V).
ackermann(s(M),s(N),V) :- ackermann(s(M),N,V1), ackermann(M,V1,V).
```

The first argument distinguishes between the cases of an atom `o` and a structure `s(M)`, but, for the latter case, two rules might be applicable. Hence, the second argument is necessary to distinguish between these rules. Therefore, we call $\{1, 2\}$ a set of *inductively sequential argument positions* for this predicate.

A precise definition of inductively sequential argument positions is based on the notion of definitional trees (Antoy 1992). The following definition is adapted to our needs.

Definition 1 (*Inductively sequential arguments*)
A *partial definitional tree* $\mathcal{T}$ with a literal l is either a *clause node* of the form *clause*(l `:-` b) with some goal b or a *branch node* of the form *branch*$(l, p, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where p is a position of a variable x in l, $f_1, \ldots, f_k$ are pairwise different functors, $\sigma_i = \{x \mapsto f_i(x_1, \ldots, x_{a_i})\}$ where $x_1, \ldots, x_{a_i}$ are new pairwise distinct variables, and, for all i in $\{1, \ldots, k\}$, the child $\mathcal{T}_i$ is a partial definitional tree with literal $\sigma_i(l)$.

A *definitional tree* of an n-ary predicate p defined by a set of clauses cs is a partial definitional tree $\mathcal{T}$ with literal $p(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are pairwise distinct variables, such that a variant of each clause of cs is represented in exactly one clause node of $\mathcal{T}$. In this case, p/n is called *inductively sequential*.

A set $D \subseteq \{1, \ldots, n\}$ of argument positions of p/n is called *inductively sequential* if there is a definitional tree $\mathcal{T}$ of p/n such that all positions occurring in branch nodes of $\mathcal{T}$ with more than one child are equal or below a position in D.

The predicate `ackermann` shown above has the inductively sequential argument sets $\{1, 2, 3\}$ and $\{1, 2\}$. For the predicates `app` and `rev` (see Example 4), $\{1\}$ is the minimal set of inductively sequential argument positions.

Our inference of result argument positions for a n-ary predicate p is based on the following heuristic:

1. Find a minimal set D of inductively sequential argument positions of p/n.
2. If D exists and the set $R = \{1, \ldots, n\} \setminus D$ is not empty, select the maximum value m of R as the result argument, that is, $\mathcal{R}(p/n) = \{m\}$, otherwise $\mathcal{R}(p/n) = \varnothing$.

Thus, a predicate is transformed into a function only if there are some inductively sequential arguments and some other arguments. In this case, we select the maximum argument position, since this is usually the intended one in practical programs. Moreover, a single result argument allows a better nesting of function calls which leads to a better demand-driven evaluation.

Minimal sets of inductively sequential argument positions can be computed by analyzing the heads of all clauses. There might be different inductively sequential argument sets for a given set of clauses. For instance, the predicate `q` defined by the clauses

```
q(a,c).
q(b,d).
```

has two minimal sets of inductively sequential arguments: {1} and {2}. The actual choice of arguments is somehow arbitrary. If predicates are inductively sequential, the results of Antoy *et al.* (2000) ensure that the programs obtained by our transformation with the heuristic described above can be evaluated in an optimal manner.

A special case of our heuristic to infer result argument positions are predicates defined by a single rule. Since such a definition is clearly nonoverlapping and inductively sequential, all such predicates might be considered as functions. However, this might lead to unintended function definitions, for example, if a predicate is defined by a single clause containing a conjunction of literals. Therefore, we use the following heuristic. A predicate defined by a single rule is transformed into a function only if the last argument of the head is not a variable or a variable which occurs in a result argument position in the rule's body. The first case is reasonable to transform predicates which define constants, as

```
two(s(s(o))).
```

into constant definitions, as

```
two = S (S O)
```

An example for the second case is the automatic transformation of `app3` into a function, as shown in Section 5.

Although these heuristics yield the expected transformations in most practical cases (they have been developed during the experimentation with our tool, see below), one can always override them using an explicit `function` directive in the logic program.

7 Extensions

Our general objective is the transformation of pure logic programs into functional logic ones. Prolog programs often use many impure features which cannot be directly translated into a language like Curry. This is intended, because functional logic languages are an approach to demonstrate how to avoid impure features and side effects by concepts from functional programming. For instance, I/O operations, offered in Prolog as predicates with side effects, can be represented in functional (logic) languages by monadic operations which structure effectful computations (Wadler 1997). Encapsulated search (`findall`) or cuts in Prolog, whose behavior depends on the search strategy and ordering of rules, can be represented in functional logic programming in a strategy-independent manner as set functions (Antoy and Hanus 2009) or default rules (Antoy and Hanus 2017). Thus, a complete transformation of Prolog programs into Curry programs might have to distinguish between "green" and "red" cuts, which is not computable. Nevertheless, it is possible to transform some Prolog features which we discuss in the following.

A useful feature of Prolog is the built-in arithmetic which avoids to compute with numbers in Peano arithmetic. For instance, consider the definition of the predicate `length` to relate a list with its number of elements:

```
length([],0).
length([X|Xs],L) :- length(Xs,L1), L is L1+1.
```

Since the first argument position is inductively sequential, it is reasonable to transform `length` into a function with $\mathscr{R}(\texttt{length}/2) = \{2\}$. Furthermore, the predicate "`is`" evaluates its second argument to a number and returns this result by unifying it with its first argument. Thus, $\mathscr{R}(\texttt{is}/2) = \{1\}$. Using this result argument position mapping, the demand functional transformation yields the program

```
length []     = 0
length (x:xs) = length xs + 1
```

where the occurrence of `is` is omitted since it behaves as the identity function.

Arbitrary Prolog cuts cannot be translated (or only into awkward code). However, Prolog cuts can be avoided by using Prolog's if-then-else construct. If the condition is a simple predicate, like a deterministic test or an arithmetic comparison, it can be translated into a functional if-then-else construct. For instance, consider the following definition of the factorial function as a Prolog predicate:

```
fac(N,F) :- (N=0  → F=1 ; N1 is N - 1, fac(N1, F1), F is F1 * N).
```

Translating this into a function (note that variable `F` is used in a result argument position in the body) with the arithmetic operations transformed as discussed above, we obtain the functional definition

```
fac n = if n == 0 then 1 else fac (n - 1) * n
```

With these extensions, many other arithmetic functions are automatically transformed into their typical functional definition.

8 Implementation

In order to evaluate our approach, we have implemented the transformation described in this paper as a tool `pl2curry` in Curry so that it can easily be installed by Curry's package manager.[6] `pl2curry` has various options to influence the transformation strategy (e.g., conservative, without let bindings, without result position inference, etc). In the default mode, the demand functional transformation is used where result arguments are inferred if they are not explicitly specified by `function` directives. The tool assumes that the logic program is written in standard Prolog syntax. It reads the Prolog file and transforms it into an abstract representation,[7] from which a Curry program is generated that can be directly loaded into a Curry system.

A delicate practical issue of the transformation is the typing of the transformed programs. Curry is a strongly typed language with parametric types and type classes (Wadler and Blott 1989). Since logic programs and standard Prolog do not contain type information, our tool defines a single type `Term` containing all atoms and functors occurring in the logic program. Although this works fine for smaller programs, it could be improved by using type information. For instance, CIAO-Prolog (Hermenegildo *et al.* 2012) supports the definition of regular and Hindley-Milner types, which could be translated into algebraic data types, or Barbosa *et al.* (2021) describe a tool to infer similar types from logic programs. Although there is some interest toward adding types to Prolog programs (Schrijvers *et al.* 2008), there is no general agreement about its syntax and

[6] `https://www-ps.informatik.uni-kiel.de/~cpm/pkgs/prolog2curry.html`.
[7] `https://www-ps.informatik.uni-kiel.de/~cpm/pkgs/prolog.html`.

Table 1. *Execution times of Prolog, Haskell, and Curry programs*

Language: System:	Prolog SWI 8.0.2	Prolog SICStus 4.7.0	Haskell GHC 8.4.4	Curry KiCS2 3.0.0
`rev_4096`	0.57	0.27	0.09	0.13
`takInt_27_16_8`	0.85	0.29	0.07	0.54
`takPeano_27_16_8`	5.81	0.79	0.17	0.47
`ackermann_3_9`	231.10	13.27	0.09	0.07

structure. Therefore, the translation of more refined types is omitted from the current implementation but it could be added in the future.

9 Evaluation

The main motivation for this work is to show that functional logic programs have concrete operational advantages compared to pure logic programs. This has been demonstrated by defining transformations for logic programs into functional logic programs. The simplest transformations (conservative and functional) keeps the structure of computations, whereas the demand functional transformation has the potential advantage to reduce the computation space by evaluating fewer subexpressions.

The practical comparison of original and transformed programs is not straightforward since it depends on the underlying implementation to execute these programs. Compilers for functional languages might contain good optimizations since they must not be prepared for nondeterministic computations (although Prolog systems based on Warren's Abstract Machine (Aït-Kaci 1991; Warren 1983) implement specific indexing techniques to support deterministic branching when it is possible). This can be demonstrated by some typical examples: the naive reverse of list structures (see Example 4), the highly recursive `tak` function used in various benchmarks (Partain 1993) for logic and functional languages, and the Ackermann function (see Section 6). Since these logic programs are automatically transformed into purely functional programs using our demand functional transformation, we can execute the original logic programs with Prolog systems and the transformed programs with Haskell (GHC) and Curry (KiCS2 Braßel *et al.* 2011) systems (since the functional kernel of Curry use the same syntax as Haskell). KiCS2 compiles Curry programs to Haskell programs by representing nondeterministic computations as search trees, that is, the generated Haskell functions return a tree of all result values. Table 1 contains the average execution times in seconds[8] of reversing a list with 4096 elements, the function `tak` applied to arguments $(27, 16, 8)$, implemented with built-in integers (`takInt`) and Peano numbers (`takPeano`), and the Ackermann function applied to the Peano representation of the numbers $(3, 9)$.

Note that the demand strategy has no real advantage in these examples. The values of all subexpressions are required so that the same resolution/rewrite steps, possibly in a

[8] The benchmarks, which are contained in the Curry package `prolog2curry`, were executed on a Linux machine running Debian 10 with an Intel Core i7-7700K (4.2Ghz) processor. The time is the total run time of executing a binary generated with the various Prolog/Haskell/Curry systems.

different order, are performed in Prolog and Haskell/Curry. Therefore, the results show the dependency on the actual language implementations. Although the table indicates the superiority of the functional programs (in particular, GHC seems to implement recursion quite efficiently), one might also obtain better results for a logic programming system by sophisticated implementation techniques, for example, by specific compilation techniques based on mode information, as done in Mercury (Somogyi *et al.* 1996), or by statically analyzing programs to optimize the generated code (Van Roy and Despain 1990). For instance, the large execution times of the Prolog version of the Ackermann function are probably due to the fact that the function is defined by pattern matching on two arguments whereas typical Prolog systems implement indexing on one argument only. Nevertheless, the results for the Curry system KiCS2 show a clear improvement without loosing the flexibility of logic programming, since the same Curry program can compute result values as well as search for required argument values. Note that all systems of these benchmarks use unbounded integers, whereas KiCS2 has a more complex representation of integers in order to support searching demanded values for free integer variables (Braßel *et al.* 2008).

Because it is difficult to draw definite conclusions from the absolute execution times, we want to emphasize the qualitative improvement of our transformation. The demand functional transformation might reduce the number of evaluation steps and leads to a demand-driven exploration of the search space. In the best case, it can reduce infinite search spaces to finite ones. We already discussed such examples before. For instance, our transformation automatically maps the logic program of Example 1 into the functional logic program of Example 2 so that the infinite search space of the predicate `dup` applied to an empty list is cut down to a small finite search space for the function `dup`. As a similar example, the goal `plus(X,Y,R), plus(R,Z,o)` (w.r.t. Example 3) has an infinite search space, whereas the transformed expression `plus (plus x y) z =:= O` has a finite search space w.r.t. the demand functional transformation.

These examples show that our transformation has a considerable advantage when goals containing several recursive predicates are used to search for solutions. Such goals naturally occur when complex data structures, like XML structures, are explored. The good behavior of functional logic programs on such applications is exploited by Hanus (2011) to implement a domain-specific language for XML processing 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 (Bry and Schaffert 2002) which uses a specialized unification procedure to ensure finite matching and unification w.r.t XML terms.

10 Conclusions

We presented methods to transform logic programs into functional logic programs. By specifying one or more arguments as results to transform predicates into functions and evaluating them with a demand-driven strategy, we showed with various examples that this transformation is able to reduce the computation space. Although this effect depends on the concrete examples, our transformation never introduces new or superfluous steps in successful computations. We also discussed a heuristic to infer result arguments for predicates. It is based on detecting inductively sequential argument positions so that the programs transformed by our method benefit from strong completeness and optimality results of functional logic programming (Antoy *et al.* 2000; Antoy 1997).

In principle, it is not necessary to switch to another programming language since demand-driven functional computations can be implemented in Prolog systems supporting coroutining. However, one has to be careful about the precise evaluation strategy implemented in this way. For instance, Naish (1991) 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 which can be avoided by complete strategies (Hanus 1995). Casas *et al.* (2006) use coroutining to implement lazy evaluation and offer a notation for functions which are mapped into Prolog predicates. Although this syntactic transformation might yield the same values and search space as functional logic languages, there are no formal results justifying this transformation. On the other hand, there are various approaches to implement lazy narrowing strategies in Prolog (Antoy and Hanus 2000; Jiménez-Martin *et al.* 1992; Loogen *et al.* 1993; López-Fraguas and Sánchez-Hernández 1999). In this sense, our results provide a systematic method to improve computations in logic programs by mapping predicates into functions and applying sound and complete evaluation strategies to the transformed programs. In particular, if predicates are defined with inductively sequential arguments (as all examples in this paper), the needed narrowing strategy is optimal, that is, the set of computed solutions is minimal and successful derivations have the shortest possible length (Antoy *et al.* 2000). This does not restrict the flexibility of logic programming but might reduce the computation space. Although our implemented tool maps logic programs into Curry programs, one could also map them back into Prolog programs by compiling the demand-driven evaluation strategy into appropriate features of Prolog (e.g., coroutining).

There is a long history to improve the execution of logic programs by modified control rules (Bruynooghe *et al.* 1989; Narain 1986). However, these proposals usually consider the operational level so that a declarative justification (soundness and completeness) is missing. In this sense, our work provides a justification for specific control rules used in logic programming, since it is based on soundness and completeness results for functional logic programs.

For future work it is interesting to use a refined representation of types (as discussed in Section 8) or to consider other methods to infer result positions, for example, by a program analysis taking into account the data flow between arguments of literals in goals.

References

Aït-Kaci, H. 1991. *Warren's Abstract Machine.* MIT Press.

Antoy, S. 1992. Definitional trees. In *Proceedings of of the 3rd International Conference on Algebraic and Logic Programming.* LNCS, vol. 632. Springer, 143–157.

Antoy, S. 1997. Optimal non-deterministic functional logic computations. In *Proceedings of International Conference on Algebraic and Logic Programming (ALP'97).* LNCS, vol. 1298. Springer, 16–30.

Antoy, S., Echahed, R. and Hanus, M. 2000. A needed narrowing strategy. *Journal of the ACM 47*, 4, 776–822.

Antoy, S. and Hanus, M. 2000. Compiling multi-paradigm declarative programs into Prolog. In *Proceedings of International Workshop on Frontiers of Combining Systems (FroCoS'2000)*. LNCS, vol. 1794. Springer, 171–185.

Antoy, S. and Hanus, M. 2009. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*. ACM Press, 73–82.

Antoy, S. and Hanus, M. 2010. Functional logic programming. *Communications of the ACM 53*, 4, 74–85.

Antoy, S. and Hanus, M. 2017. Default rules for Curry. *Theory and Practice of Logic Programming 17*, 2, 121–147.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That*. Cambridge University Press.

Barbosa, J., Florido, M. and Santos Costa, V. 2021. Data type inference for logic programming. In *Proceedings of the 31st International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2021)*. LNCS, vol. 13290. Springer, 16–37.

Barbuti, R., Bellia, M., Levi, G. and Martelli, M. 1984. On the integration of logic programming and functional programming. In *Proceedings of IEEE International Symposium on Logic Programming*. Atlantic City, 160–166.

Brassel, B., Fischer, S. and Huch, F. 2008. Declaring numbers. *Electronic Notes in Theoretical Computer Science 216*, 111–124.

Brassel, B., Hanus, M., Peemöller, B. and Reck, F. 2011. KiCS2: A new compiler from Curry to Haskell. In *Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*. LNCS, vol. 6816. Springer, 1–18.

Bruynooghe, M., De Schreye, D. and Krekels, B. 1989. Compiling control. *Journal of Logic Programming 6*, 135–162.

Bry, F. and Schaffert, S. 2002. 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)*. LNCS, vol. 2401. Springer, 255–270.

Casas, A., Cabeza, D. and Hermenegildo, M. 2006. A syntactic approach to combining functional notation, lazy evaluation, and higher-order in LP systems. In *Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, LNCS, vol. 3945. Springer, 146–162.

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

Hanus, M. 1995. Analysis of residuating logic programs. *Journal of Logic Programming 24*, 3, 161–199.

Hanus, M. 2011. Declarative processing of semistructured web data. In *Technical Communications of the 27th International Conference on Logic Programming*, vol. 11. Leibniz International Proceedings in Informatics (LIPIcs), 198–208.

Hanus, M. 2013. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*. LNCS, vol. 7797. Springer, 123–168.

Hanus, M. (ed.) 2016. Curry: An integrated functional logic language (vers. 0.9.0). URL: http://www.curry-lang.org.

Hermenegildo, M., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. and Puebla, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12*, 1-2, 219–252.

Huet, G. and Lévy, J.-J. 1991. Computations in orthogonal rewriting systems. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. MIT Press, 395–443.

Hughes, J. 1990. Why functional programming matters. In *Research Topics in Functional Programming*, D. Turner, Ed. Addison Wesley, 17–42.

Jiménez-Martin, J., Marino-Carballo, J. and Moreno-Navarro, J. 1992. Efficient compilation of lazy narrowing into Prolog. In *Proceedings of International Workshop on Logic Program Synthesis and Transformation (LOPSTR'92)*. Springer Workshops in Computing Series, 253–270.

Launchbury, J. 1993. A natural semantics for lazy evaluation. In *Proceedings of 20th ACM Symposium on Principles of Programming Languages (POPL'93)*. ACM Press, 144–154.

Lloyd, J. 1987. *Foundations of Logic Programming*, 2nd extended ed. Springer.

Loogen, R., López Fraguas, F. and Rodríguez Artalejo, M. 1993. A demand driven computation strategy for lazy narrowing. In *Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming*. LNCS, vol. 714. Springer, 184–200.

López-Fraguas, F. and Sánchez-Hernández, J. 1999. TOY: A multiparadigm declarative system. In *Proceedings of RTA'99*. LNCS, vol. 1631. Springer, 244–247.

Naish, L. 1991. Adding equations to NU-Prolog. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*. LNCS, vol. 528. Springer, 15–26.

Narain, S. 1986. A technique for doing lazy evaluation in logic. *Journal of Logic Programming 3*, 259–276.

Partain, W. 1993. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer, 195–202.

Peyton Jones, S., Ed. 2003. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press.

Reddy, U. 1985. Narrowing as the operational semantics of functional languages. In *Proceedings of IEEE International Symposium on Logic Programming*, Boston, 138–151.

Schrijvers, T., Santos Costa, V., Wielemaker, J. and Demoen, B. 2008. Towards Typed Prolog. In *24th International Conference on Logic Programming (ICLP 2008)*. Springer. LNCS, vol. 5366, 693–697.

Slagle, J. 1974. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM 21*, 4, 622–642.

Somogyi, Z., Henderson, F. and Conway, T. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29*, 1-3, 17–64.

Sterling, L. and Shapiro, E. 1994. *The Art of Prolog*, 2nd ed. MIT Press.

TeReSe 2003. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press.

Van Roy, P. and Despain, A. 1990. The benefits of global dataflow analysis for an optimizing prolog compiler. In *Proceedings of the 1990 North American Conference on Logic Programming*. MIT Press, 501–515.

Wadler, P. 1997. How to declare an imperative. *ACM Computing Surveys 29*, 3, 240–263.

Wadler, P. and Blott, S. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of POPL'89*, 60–76.

Warren, D. 1983. An abstract Prolog instruction set. Technical note 309, SRI International, Stanford.