In *Proc. of the Joint International Symposium PLILP/ALP'98*,
Pisa (Italy). Springer LNCS 1490, pp. 374–390, 1998

Controlling Search in Declarative Programs*

Michael Hanus and Frank Steiner
RWTH Aachen, Informatik II, D-52056 Aachen, Germany
{hanus,steiner}@i2.informatik.rwth-aachen.de

Abstract. Logic languages can deal with non-deterministic computations via built-in search facilities. However, standard search methods like global backtracking are often not sufficient and a source of many programming errors. Therefore, we propose the addition of a single primitive to logic-oriented languages to control non-deterministic computation steps. Based on this primitive, a number of different search strategies can be easily implemented. These search operators can be applied if the standard search facilities are not successful or to encapsulate search. The latter is important if logic programs interact with the (non-backtrackable) outside world.
We define the search control primitive based on an abstract notion of computation steps so that it can be integrated into various logic-oriented languages, but to provide concrete examples we also present the integration of such a control primitive into the multi-paradigm declarative language Curry. The lazy evaluation strategy of Curry simplifies the implementation of search strategies, which also shows the advantages of integrating functions into logic languages.

1 Introduction

Computing solutions to partially instantiated goals and dealing with non-deterministic computations via built-in search facilities is one of the most important features of logic languages. Standard logic languages like Prolog use a global backtracking strategy to explore the different alternatives of a computation. This is often not sufficient and a source of many problems:

– If a top-level predicate fails, all alternatives of previously called predicates are also explored. This may lead to an unexpected behavior and makes the detection of programming errors difficult (e.g., if the backtracking is caused by a missing alternative in the top-level predicate). This problem is often solved by inserting "cuts" which, however, decreases the readability of programs.
– Depth-first search is an incomplete strategy. Although this drawback can be managed by experienced programmers, it causes difficulties for beginners (who frequently use predicates like commutativity or left-recursive clauses in the beginning). As a consequence, one is forced to talk about Prolog's depth-first search strategy too early in logic programming courses. This can have a negative impact on the declarative style of programming.
– In larger application programs (e.g., combinatorial problems), other strategies than the standard depth-first search are often necessary. In such cases the programmer is forced to program her own strategies (e.g., by using meta-

* This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1.

programming techniques). The possible interaction with the standard strategy can lead to errors which are difficult to find.

These problems can be solved if there is a simple way to replace the standard search strategy by other strategies and to implement new search strategies fairly easy. In this paper we show that this is possible by adding a single primitive operation to control non-deterministic computation steps. This primitive, which is a generalization of Oz's search operator [15], evaluates the program as usual but immediately stops if a non-deterministic step occurs. In the latter case, the different alternatives are returned so that the programmer can determine the way to proceed the computation. Based on this primitive, a number of different search operators, like depth-first search, breadth-first search, `findall`, or the Prolog shell, can be easily implemented. These operators also allow the encapsulation of possible search in local predicates. This feature is important if logic programs interact with the (non-backtrackable) outside world, like file accesses or Internet applications.

In contrast to Oz's search operator [15], which is directly connected to a syntactic construct of the language (disjunctions), our control operator is based on an abstract notion of basic computation steps. Thus, it can be considered as a meta-level construct to control (don't know) non-deterministic computation steps which could be added to logic-oriented languages provided that they offer constraints or equations to represent variable bindings and existential quantification to distinguish variables which can be bound in a local computation. Moreover, we provide a formal connection between the search trees of the base language and the results computed by our search operators. Hence, soundness and completeness results for the base language carry over to corresponding results for particular search strategies based on our control operator.

The next section introduces our notion of computation steps of the base language. The primitive to control non-deterministic computations is described in Section 3. Based on this primitive, we show the implementation of different search strategies in Section 4. The relations of these search strategies with the search trees of the base language are established in Section 5. We show the advantages of a base language with lazy evaluation to provide a simple implementation of search strategies in Section 6. Section 7 compares our techniques with related work, and Section 8 contains our conclusions. Due to lack of space, we omit some details and the proofs of the theorems which can be found in [4, 5].

2 Operational Semantics of the Base Language

As mentioned above, the search primitive should control the different non-deterministic steps occurring in a derivation. To abstract from the operational model of the concrete base language, we only assume that a computation step of the base language reduces an expression (goal) to a disjunction consisting of a sequence of pairs of substitutions (bindings) and expressions (goals), i.e, the *operational semantics of the base language* is defined by a one step relation

$$e \;\Rightarrow\; \sigma_1, e_1 \mid \cdots \mid \sigma_n, e_n$$

where $n \geq 0$, e, e_1,...,e_n are expressions, $\sigma_1, \ldots, \sigma_n$ are substitutions on the free variables in e, and "|" joins different alternatives to a disjunction. A *substitution* is a mapping from variables into terms and we denote it by $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. $\mathcal{D}om(\sigma) = \{x_1, \ldots, x_n\}$ is the *domain* of σ and $\mathcal{VR}an(\sigma) = \mathcal{V}ar(t_1) \cup \ldots \cup \mathcal{V}ar(t_n)$ is its *variable range*, where $\mathcal{V}ar(e)$ denotes the set of all free variables occurring in an expression e. The identity substitution (i.e., the substitution id with $\mathcal{D}om(id) = \emptyset$) is often omitted in computation steps. We call the evaluation step *deterministic* if $n = 1$ and *non-deterministic* if $n > 1$. The case $n = 0$ corresponds to a *failure* and is also written as $e \Rightarrow \mathit{fail}$.

This notion of a computation step makes the possible don't know non-determinism of the base language explicit which will be controlled by our search primitive. A possible don't care non-determinism (e.g., in a concurrent base language) corresponds to an indeterminate definition of "$\Rightarrow$" and will not be controlled by our search primitive. Furthermore, note that this notion of a computation step covers a variety of declarative languages. In functional programming, n is at most 1 (i.e., no non-deterministic step occurs) and all substitutions are the identity since unbound variables do not occur during a computation. In logic programming, e is a goal, $e_1, \ldots, e_n$ are all resolvents of this goal and $\sigma_1, \ldots, \sigma_n$ are the corresponding unifiers restricted to the goal variables (for constraint logic programming, the notion of substitutions must be replaced by constraints).

Since our search control operator will be based on this abstract notion of a computation step of the base language (in contrast to Oz [15]), it is applicable to a variety of (functional, constraint) logic languages. To provide concrete examples and to show the advantages of integrating lazily evaluated functions into a logic language, we present the addition of the search control operator to Curry [3, 5], a multi-paradigm declarative language aiming to amalgamate functional, logic, and concurrent programming paradigms. Therefore, we outline in the rest of this section Curry's computation model (details can be found in [3, 5]).

Values in Curry are, similarly to functional or logic languages, *data terms* constructed from constants and data constructors. These are introduced through *data type declarations* like[1]

```
data bool    = true | false
data nat     = z    | s(nat)
data list(A) = []   | [A|list(A)]
```

`true` and `false` are the Boolean constants, `z` and `s` are the zero value and the successor function to construct natural numbers,[2] and polymorphic lists (`A` is a type variable ranging over all types) are defined as in Prolog.

A *data term* is a well-typed[3] expression containing variables, constants and data constructors, e.g., `s(s(z))`, `[true|Z]` etc. *Functions* (*predicates* are considered as Boolean functions for the sake of simplicity) operate on data terms. Their mean-

[1] In the following we use a Prolog-like syntax which is slightly different from the actual Curry syntax.

[2] Curry has also built-in integer values and arithmetic functions. We use here the explicit definition of naturals only to provide some simple and self-contained examples.

[3] The current type system of Curry is a Hindley/Milner-like system with parametric polymorphism, e.g., a term like `s(true)` is ill-typed and thus excluded.

ing is specified by *rules* (or *equations*) of the form l `|` $\{c\}$ `=` r (the condition part "`|` $\{$`c`$\}$" is optional) where l is a *pattern*, i.e., l has the form $f(t_1, \ldots, t_n)$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once, and r is a well-formed *expression* containing function calls, constants, data constructors and variables from l and c. The *condition* c is a *constraint* which consists of a conjunction of equations and optionally contains a list of locally declared variables, i.e., a constraint can have the form `let` $v_1, \ldots, v_k$ `free in` $\{eq_1, \ldots, eq_n\}$ where the variables v_i are only visible in the equations $eq_1, \ldots, eq_n$. If a local variable v of a condition should be visible also in the right-hand side, the rule is written as l `|` $\{c\}$ `=` r `where` v `free`. A rule can be applied if its condition is satisfiable. A *head normal form* is a variable, a constant, or an expression of the form $c(e_1, \ldots, e_n)$ where c is a data constructor. A *Curry program* is a set of data type declarations and equations.

Example 1. The addition on natural numbers (type `nat` above) is defined by

```
add(z   ,Y) = Y
add(s(X),Y) = s(add(X,Y))
```

The following rules define the concatenation of lists and functions for computing the first and the last element of a list ("`_`" denotes an anonymous variable):

```
append([]    ,Ys) = Ys
append([X|Xs],Ys) = [X|append(Xs,Ys)]
first([X|_]) = X
last(Xs) | {append(_,[X])=Xs} = X  where X free
```

If the equation `append(_,[X])=Xs` is solvable, then `X` is the last element of `Xs`. □

From a functional point of view, we are interested in computing the *value* of an expression, i.e., a data term which is equivalent (w.r.t. the program rules) to the initial expression. The value can be computed by applying rules from left to right. For instance, to compute the value of `add(s(z),s(z))`, we apply the rules for addition to this expression:

`add(s(z),s(z))` $\Rightarrow$ `s(add(z,s(z)))` $\Rightarrow$ `s(s(z))`

A *strategy* selects a single function call for reduction in the next step. Curry is based on a *lazy* (*leftmost outermost*) strategy. This also allows the computation with infinite data structures and provides more modularity, as we will see in Section 6. Thus, to evaluate the expression `add(add(z,s(z)),z)`, the first subterm `add(z,s(z))` is evaluated to head normal form (in this case: `s(z)`) since its value is required by all rules defining `add` (such an argument is also called *demanded*). On the other hand, the evaluation of the subterm `append([z],[])` is not needed in the expression `first([z|append([z],[])])` since it is not demanded by `first`. Therefore, this expression is reduced to `z` by one outermost reduction step.

Since Curry subsumes logic programming, it is possible that the initial expression may contain variables. In this case the expression might not be reducible to a single value. For instance, a logic programming system should find values for the variables in a goal such that it is reducible to `true`. Fortunately, it requires only a slight extension of the lazy reduction strategy to cover non-ground expressions and variable instantiation: if the value of a variable argument is demanded by the left-hand sides of program rules in order to proceed the computation, the variable is non-deterministically bound to the different demanded values.

Example 2. Consider the function `f` defined by the rules

```
f(a) = c
f(b) = d
```

(`a`, `b`, `c`, `d` are constants). Then the expression `f(X)` with the variable argument `X` is evaluated to `c` or `d` by binding `X` to `a` or `b`, respectively. Thus, this non-deterministic computation step can be denoted as follows: `f(X)` $\Rightarrow$ $\{$`X` $\mapsto$ `a`$\}$ `c` | $\{$`X` $\mapsto$ `b`$\}$ `d`. □

A single *computation step* in Curry performs a reduction in exactly one (unsolved) expression of a disjunction. For inductively sequential programs [1] (these are, roughly speaking, function definitions without overlapping left-hand sides), this strategy, called *needed narrowing* [1], computes the shortest possible successful derivations (if common subterms are shared) and a minimal set of solutions, and it is fully deterministic if variables do not occur.[4]

Functional logic languages are often used to solve equations between expressions containing defined functions. For instance, consider the equation $\{$`add(X,z)=s(z)`$\}$ w.r.t. Example 1. It can be solved by evaluating the left-hand side `add(X,z)` to the answer expression $\{$`X` $\mapsto$ `s(z)`$\}$`s(z)` (here we omit the other alternatives). Since the resulting equation is trivial, the equation is valid w.r.t. the computed answer $\{$`X` $\mapsto$ `s(z)`$\}$. In general, an *equation* or *equational constraint* $\{e_1\texttt{=}e_2\}$ is satisfied if both sides e_1 and e_2 are reducible to the same data term. Operationally, an equational constraint $\{e_1\texttt{=}e_2\}$ is solved by evaluating e_1 and e_2 to unifiable data terms where the lazy evaluation of the expressions is interleaved with the binding of variables to constructor terms [10]. Thus, an equational constraint $\{e_1\texttt{=}e_2\}$ without occurrences of defined functions has the same meaning (unification) as in Prolog.[5] Note that constraints are solved when they appear in conditions of program rules in order to apply this rule or when a search operator is applied (see Section 3). Conjunctions of constraints can also be evaluated concurrently but we omit this aspect here (see [3, 5] for more details).

3 Controlling Non-deterministic Computation Steps

Most of the current logic languages are based on global search implemented by backtracking, i.e., disjunctions distribute to the top-level (i.e., a goal $A \wedge B$, where A is defined by $A \leftrightarrow A_1 \vee A_2$, is logically replaced by $(A_1 \wedge B) \vee (A_2 \wedge B)$). As discussed in Section 1, this must be avoided in some situations in order to control the exploration of the search space.

For instance, consider the problem of doing input/output. I/O is implemented in most logic languages by side effects. To handle I/O in a declarative way, as done

[4] These properties also show some of the advantages of integrating functions into logic programs, since similar properties for purely logic programs are not known.

[5] We often use the general notion of a *constraint* instead of equations since it is conceptually fairly easy to add other constraint structures than equations over Herbrand terms.

in Curry, one can use the *monadic I/O* concept [18] where an interactive program is considered as a function computing a sequence of actions which are applied to the outside world. An *action* changes the state of the world and possibly returns a result (e.g., a character read from the terminal). Thus, actions are functions of the type

$$World \quad \to \quad pair(\alpha, World)$$

(where $World$ denotes the type of all states of the outside world). This function type is also abbreviated by $IO(\alpha)$. If an action of type $IO(\alpha)$ is applied to a particular world, it yields a value of type α and a new (changed) world. For instance, `getChar` of type $IO(Char)$ is an action which reads a character from the standard input whenever it is executed, i.e., applied to a world. The important point is that values of type $World$ are not accessible to the programmer — she/he can only create and compose actions on the world. For instance, the action `getChar` can be composed with the action `putChar` (which writes a character to the terminal) by the sequential composition operator `>>=`, i.e., "`getChar >>= putChar`" is a composed action which prints the character typed in the keyboard to the screen (see [18] for more details).

An action, obtained as a result of a program, is executed when the program is executed. Since the world cannot be copied (note that the world contains at least the complete file system or the complete Internet in web applications), an interactive program having a disjunction as a result makes no sense. Therefore, all possible search must be encapsulated between I/O operations. In the following, we describe a primitive operation to get control over non-deterministic computations so that one can encapsulate the necessary search for solving goals.

3.1 Search Goals and a Control Primitive

Since search is used to find solutions to some constraint, we assume that search is always initiated by a constraint containing a *search variable* for which a solution should be computed.[6] A difficulty is that the search variable may be bound to different solutions (by different alternatives in non-deterministic steps) which should be represented in a single expression for further processing. As a solution, we adapt the idea of Oz [15] and abstract the search variable w.r.t. the constraint to be solved, which is possible in a language providing functions as first-class objects.[7] Therefore, a *search goal* has the function type $\alpha \to Constraint$ where α is the type of the values which we are searching for and $Constraint$ is the type of all constraints (e.g., an equation like {`add(X,z)=s(z)`} is an expression of type $Constraint$, cf. [5]). In particular, if c is a constraint containing a variable x and we are interested in solutions for x, i.e., values for x such that c is satisfied, then the corresponding search goal has the form `\`x`->`c (this is the notation for lambda

[6] The generalization to more than one search variable is straightforward by the use of tuples.

[7] If the base language does not provide functions as first-class objects, one has to introduce a special language construct to denote the search variable, like in Prolog's `setof` or `findall` predicate.

abstractions, e.g., `\X->3*X` denotes an anonymous function which multiplies its argument with 3). For instance, if we are interested in values for the variable `X` such that the equation `append([1],X)=[1,2]` holds, then the corresponding search goal is `\X->{append([1],X)=[1,2]}`.

To control the non-deterministic steps performed to find solutions to search goals, we introduce a function[8] `try` of type

$$(\alpha \;\rightarrow\; \mathit{Constraint}) \quad\rightarrow\quad \mathit{list}(\alpha \;\rightarrow\; \mathit{Constraint})$$

i.e., `try` takes a search goal as an argument and produces a list of search goals. The idea is that `try` attempts to evaluate the constraint of the search goal until the computation finishes or does a non-deterministic step. In the latter case, the computation is immediately stopped and the different constraints obtained by the non-deterministic step are returned. Thus, an expression of the form `try(`g`)` can have the following outcomes:

`try(`g`)` = `[]`: The empty list indicates that the search goal g has no solution. For instance, the expression

```
try(\X -> {1=2})
```

reduces to `[]`. Note that a failure of the search can now be handled explicitly because it does not lead to a failure of the whole computation as it would do without the search operator.

`try(`g`)` = `[`g'`]`: A one-element list indicates that the search goal g has a single solution represented by the element of this list. For instance, the expression

```
try(\X -> {[X]=[0]})
```

reduces to `[\X->{X=0}]`. Note that a solution, i.e., a binding for the search variable like a substitution $\{x \mapsto t\}$, can always be represented as an equational constraint $\{x\texttt{=}t\}$. In the following, we denote by $\overline{\sigma}$ the *equational representation* of the substitution σ.

`try(`g`)` = `[`g_1`,...,`g_n`]`, $n > 1$: If the result list contains more than one element, the evaluation of the search goal g requires a non-deterministic computation step. The different alternatives immediately after this non-deterministic step are represented as elements of this list, where the different bindings of the search variable are added as constraints. For instance, if the function `f` is defined as in Example 2, then the expression

```
try(\X -> {f(X)=d})
```

reduces to the list `[\X->{X=a,c=d}, \X->{X=b,d=d}]`. This example also shows why the search variable must be abstracted: the alternative bindings cannot be actually performed (since a variable is only bound to at most one value in each computation thread) but are represented as equational constraints in the search goal. Note that the search goals in the result list are not further evaluated. The further evaluation can be done by a subsequent application of `try` to the list elements. This allows the explicit control of the strategy to explore the search tree. It will be discussed in more detail in Section 4.

[8] If the base language does not provide functions, like Prolog, we can also implement `try` as a binary predicate where the second argument denotes the result.

$$
\texttt{try}(g) = \begin{cases}
\texttt{[]} & \text{if } \{c\} \Rightarrow \mathit{fail} \\
\texttt{[}g'\texttt{]} & \text{if } \{c\} \Rightarrow \sigma\,\{\} \text{ (i.e., } \sigma \text{ is a mgu for all equations in } c\text{) with} \\
 & \mathcal{D}om(\sigma) \subseteq \{x, x_1, \ldots, x_n\},\ g' = \texttt{\textbackslash}x\texttt{->let}\, x_1, \ldots, x_n\, \texttt{free in}\, \{\overline{\sigma}\} \\
\texttt{try(}g'\texttt{)} & \text{if } \{c\} \Rightarrow \sigma\,\{c'\} \text{ with } \mathcal{D}om(\sigma) \subseteq \{x, x_1, \ldots, x_n\} \\
 & \text{and } g' = \texttt{\textbackslash}x\texttt{->let }\ x_1, \ldots, x_n, y_1, \ldots, y_m\ \texttt{ free in }\ \{\overline{\sigma}, c'\} \\
 & \text{where } \{y_1, \ldots, y_m\} = \mathcal{VR}an(\sigma) \setminus (\{x, x_1, \ldots, x_n\} \cup \mathcal{V}ar(g)) \\
\texttt{[}g_1\texttt{,}\ldots\texttt{,}g_k\texttt{]} & \text{if } \{c\} \Rightarrow \sigma_1\,\{c_1\} \mid \cdots \mid \sigma_k\,\{c_k\},\ k > 1, \text{ and, for } i = 1, \ldots, k, \\
 & \mathcal{D}om(\sigma_i) \subseteq \{x, x_1, \ldots, x_n\} \text{ and} \\
 & g_i = \texttt{\textbackslash}x\texttt{->let }\ x_1, \ldots, x_n, y_1, \ldots, y_{m_i}\ \texttt{ free in }\ \{\overline{\sigma_i}, c_i\} \\
 & \text{where } \{y_1, \ldots, y_{m_i}\} = \mathcal{VR}an(\sigma_i) \setminus (\{x, x_1, \ldots, x_n\} \cup \mathcal{V}ar(g)) \\
\mathit{suspend} & \text{otherwise}
\end{cases}
$$

Fig. 1. Operational semantics of the `try` operator for $g = \texttt{\textbackslash}x\texttt{->let}\, x_1, \ldots, x_n\, \texttt{free in}\, \{c\}$

3.2 Local Variables

Some care is necessary if free variables occur in a search goal, as in

```
\E -> {append(L,[E])=[3,4,5]}                                   (*)
```

To compute the last element `E` of the list `[3,4,5]` with this goal, the variable `L` must be instantiated which is problematic since `L` is free. There are different possibilities to deal with this case. In Prolog's `bagof/setof` predicates, free variables are (possibly non-deterministically!) instantiated and then remain instantiated with this value, which does not help to really encapsulate all search and sometimes leads to unexpected results. Another ad-hoc method is to consider a `try` application to a search goal containing free variables as a run-time error. Since Curry as well as most Prolog systems is equipped with coroutining facilities, we take another solution and require that the `try` operator *never* binds free variables of its search goal. If it is necessary to bind a free variable in order to proceed a `try` evaluation, the computation suspends. Thus, a `try` application to the search goal $(*)$ cannot be evaluated and suspends until the variable `L` is bound.

To allow possible bindings of unbound variables during a local search, they can be declared as local to the constraint so that they might have different bindings in different branches of the search. For instance, we start the evaluation of

```
try(\E -> let L free in {append(L,[E])=[3,4,5]})                (**)
```

to compute the last element of the list `[3,4,5]`. Now the variable `L` is only visible inside the constraint (i.e., existentially quantified) and can be bound to different values in different branches. Therefore, the expression $(**)$ evaluates to

```
[\E -> let L        free in {L=[], [E]=[3,4,5]},
 \E -> let L,X,Xs free in {L=[X|Xs], [X|append(Xs,[E])]=[3,4,5]}]
```

The new variables `X` and `Xs` (introduced by unification) are also added to the list of local variables so that they can be further instantiated in subsequent steps.

The exact behavior of the `try` operator is specified in Figure 1. Thus, the search goal is solved (second case) if the constraint is solvable without bindings of global variables. In a deterministic step (third case), we apply the `try` operator again after adding the newly introduced variables to the list of local variables. Note that the free variables $\mathcal{V}ar(g)$ occurring in g must not be declared as local because they can appear also outside of g, and therefore they have to be removed from $\mathcal{VR}an(\sigma)$. In a non-deterministic step (fourth case), we return the different alternatives as the result. If a computation step on the constraint tries to bind a free variable, the evaluation of the `try` operator suspends. In order to ensure that an encapsulated search will not be suspended due to necessary bindings of free variables, the search goal should be a closed expression when a search operator is applied to it, i.e., the search variable is bound by the lambda abstraction and all other variables are existentially quantified by local declarations.

Note that the operational semantics of the `try` operator depends only on the meaning of computation steps of the underlying language. Thus, it can be introduced in a similar way to other logic-oriented languages than Curry.[9] Although the management and testing of local variable bindings look complicated, it can be efficiently implemented by decorating each variable with a declaration level and checking the level in binding attempts (similarly to the implementation of scoping constructs in logic languages [12]). Moreover, the equational representations $\overline{\sigma_i}$ of the substitutions need not be explicitly implemented but can be implicitly represented by binding lists for the variables.

4 Search Strategies

The search control operator `try` introduced in the previous section is a basis to implement powerful and easily applicable search strategies. This section demonstrates the use of `try` to implement some search strategies in Curry. These strategies can be defined in a similar way in other declarative languages. However, we will show in Section 6 that Curry's lazy evaluation strategy can be exploited to simplify the application of search operators.

The following function defines a **depth-first search** strategy which collects all solutions of a search goal in a list:

```
all(G) = collect(try(G))
         where collect([])          = []
               collect([G])         = [G]
               collect([G1,G2|Gs]) = concat(map(all,[G1,G2|Gs]))
```

The auxiliary function `collect` applies recursively `all` to all resulting alternatives of a non-deterministic step and appends all solutions in a single list (`concat` concatenates a list of lists to a single list and `map` applies a function to all elements of a list). Thus, the expression

```
all(\L->{append(L,[1])=[0,1]})
```

reduces to `[\L->{L=[0]}]`.

Due to the laziness of Curry, search goals with infinitely many solutions cause no problems if one is interested only in finitely many of them. A function which

[9] For concurrent languages, one could modify the definition of `try` such that non-deterministic steps lead to a suspension as long as a deterministic step might be enabled by another computation thread. This corresponds to *stability* in AKL [8] and Oz [15].

computes only the first solution w.r.t. a depth-first search strategy can be simply defined by

```
once(G) = first(all(G))
```

Note that `once` is a partial function, i.e., it is undefined if `G` has no solution.

The value computed for the search variable in a search goal can be easily accessed by applying it to an unbound variable. For instance, the evaluation of the applicative expression

```
once(\L->{append(L,[1])=[0,1]})@X
```

(`F@E` denotes the application of a function `F` to some `E`, where `F` and `E` can be arbitrary expressions) binds the variable `X` to the value `[0]`, since the first subexpression evaluates to `\L->{L=[0]}` and the constraint `{X=[0]}` obtained by the application of this expression to `X` can only be solved by this binding. Based on this idea, we can define a function `unpack` that takes a list of solved search goals and computes the list of the corresponding values for the search variable:

```
unpack([]) = []
unpack([G|Gs]) | {G@X} = [X|unpack(Gs)]  where X free
```

Now it is simple to define a function similarly to Prolog's `findall` predicate:

```
findall(G) = unpack(all(G))
```

For a search goal without free variables, `findall` explores the search tree (depth first) and collects all computed values for the search variable in a list.

A **bounded search** strategy, where search is performed only up to a given depth n in the search tree, can also be easily implemented when we consider search trees containing only the nodes for non-deterministic steps. This means that search will not end after n arbitrary reduction steps but only after n non-deterministic steps. The following function is very similar to the function `all` but explores the search goal `G` only up to depth `N`.

```
all_bounded(N,G) = if N>1 then collect(try(G)) else []  where
  collect([])         = []
  collect([G])        = [G]
  collect([G1,G2|Gs]) = concat(map(all_bounded(N-1),[G1,G2|Gs]))
```

Note that the algorithm may not terminate if an infinite deterministic reduction occurs (which is seldom in practical search problems) because the search operator will never return a result in this case. The same can happen with the next algorithm implementing a **breadth-first search** strategy that traverses the search tree level by level and each level from left to right, regarding as level n all goals obtained from the search goal after n non-deterministic steps.

```
all_bfs(G) = trygoals([G])  where
   trygoals([])     = []
   trygoals([G|Gs]) = splitgoals(map(try,[G|Gs]),[])
   splitgoals([]              ,Ugs) = trygoals(Ugs)
   splitgoals([[]|Gs]         ,Ugs) = splitgoals(Gs,Ugs)
   splitgoals([[G]|Gs]        ,Ugs) = [G|splitgoals(Gs,Ugs)]
   splitgoals([[G1,G2|G3]|Gs],Ugs) = splitgoals(Gs,
                                          append(Ugs,[G1,G2|G3]))
```

The function `trygoals` applies the search operator to the list of remaining alternatives and scans the result (a list of lists) using the function `splitgoals`, which removes failures and returns all solutions computed so far. Then the remaining goals, which result from non-deterministic steps, are recursively explored further. Similarly, one can also implement other search strategies like depth-first iterative deepening or best solution search with branch and bound [15]. Moreover, a parallel fair search for the first or all solutions can be implemented with our search primitive and a committed choice [15] (which is also available in Curry). To show the use of encapsulated search to control the failure of computations, we define a function on constraints which implements **negation as finite failure** known from logic programming:

```
naf(C) = {all(\_->{C}) = []}
```

Thus, if `C` is a constraint where all variables are existentially quantified, then `naf(C)` is solvable iff the search space of solving `C` is finite and does not contain any solution.

5 Search Trees and Search Operators

In this section we sketch the connection between the search trees of the base language and the results computed by some of the search operators defined above. More details can be found in [4].

The notion of a *search tree w.r.t.* $\Rightarrow$ can be defined as in logic programming [9], i.e., each node is marked with a constraint, and if an *inner node* N is marked with c and $c \Rightarrow \sigma_1, c_1 \mid \cdots \mid \sigma_k, c_k$ is a computation step of the base language, then N has k sons $N_1, \ldots, N_k$ where N_i is marked with c_i and the edge from c to c_i is marked with σ_i ($i = 1, \ldots, k$). In case of logic programming, where $\Rightarrow$ denotes a resolution step with all resolvents for a goal, search trees w.r.t. $\Rightarrow$ are similar to SLD-trees [9]. *Leaves* are nodes marked with a constraint c that cannot be further derived. The leaf is *successful* if c is the empty constraint (in this case we call the composition of all substitutions marked along the branch from the root to this leaf a $\Rightarrow$*-computed answer* for the constraint at the root of the tree). The leaf is *failed* if $\{c\} \Rightarrow \mathit{fail}$. All other leaves are *suspended*.

The following theorems relate search trees w.r.t. $\Rightarrow$ to results computed by some of the search operators (here we assume the functional definition as given in the previous section, but these properties can be also transferred to other definitions, e.g., in a relational style). To simplify the formulation of the theorems, we represent a *search goal* as a triple (V, σ, c) where $V = \{x_1, \ldots, x_n\}$ is a set of variables, σ is a substitution and c is a constraint. This corresponds to `\_->let` $x_1, \ldots, x_n$ `free in` $\{\overline{\sigma}, c\}$ in the representation introduced in Section 3, i.e., here we ignore the special rôle of the search variable since it is not important for the results in this section. In order to avoid the problem of suspension due to necessary bindings of free variables, we consider only initial search goals where all variables are existentially quantified.

The first theorem states the soundness of the `all` operator.

Theorem 1 (Soundness of "all"). *Let c be a constraint and $g = (\mathcal{V}ar(c), id, c)$. If* `all`$(g)$ *evaluates to a list* `[`(V_1,σ_1,c_1)`,`(V_2,σ_2,c_2)`,...]`*, then each c_i is an empty constraint, each σ_i is a $\Rightarrow$-computed answer for c and $\mathcal{V}ar(c) \cup \mathcal{VR}an(\sigma_i) \subseteq V_i$.*

The converse result does not hold in general due to infinite branches in the search tree, since `all` implements a depth-first search through the tree. However, we can state a completeness result for the case of finite search trees.

Theorem 2 (Completeness of "all" for finite search trees). *Let c be a constraint and σ be a $\Rightarrow$-computed answer for c. If the search tree with root c is finite, then* `all(`$(\mathcal{V}ar(c), id, c)$`)` *evaluates to a list* `[`(V_1,σ_1,c_1)`,...,`(V_n,σ_n,c_n)`]`*, where $\sigma_i = \sigma$ for some $i \in \{1,\ldots,n\}$.*

A corollary of this theorem is the completeness of the negation-as-failure operator.

Corollary 1 (Completeness of "naf" for finite search trees). *Let c be a constraint. If the search tree with root c is finite and contains only failed leaves, then* `naf(`c`)` *is a solvable constraint.*

A further interesting result is the completeness of the breadth-first search strategy `all_bfs`. As already discussed, this strategy may be incomplete in case of infinite deterministic evaluations. Therefore, we call a search tree *deterministically terminating* if there is no infinite branch where each inner node has exactly one successor. Excluding this case, which is seldom in practical search problems, we can state the following completeness result.

Theorem 3 (Completeness of "all_bfs"). *Let c be a constraint and σ be a $\Rightarrow$-computed answer for c. If the search tree with root c is deterministically terminating, then* `all_bfs(`$(\mathcal{V}ar(c), id, c)$`)` *evaluates to a (possibly infinite) list* `[`(V_1,σ_1,c_1)`,`(V_2,σ_2,c_2)`,...]`*, where $\sigma_i = \sigma$ for some $i > 0$.*

6 Exploiting Laziness

We already exploited the advantages of Curry's lazy evaluation strategy by defining the search for the first solution (`once`) based on the general depth-first search strategy `all`. This shows that lazy evaluation can reduce the programming efforts. Furthermore, it is well known from functional programming that lazy evaluation provides more modularity by separating control aspects [7]. We want to emphasize this advantage by an implementation of Prolog's top-level shell with our search operator.

The interactive command shell of a Prolog interpreter roughly behaves as follows. If the user types in a goal, a solution for this goal is computed by the standard depth-first search strategy. If a solution is found, it is presented to the user who can decide to compute the next solution (by typing '`;`' and `<return>`) or to ignore further solutions (by typing `<return>`). This behavior can be easily implemented with our search operator:

```
prolog(G) = printloop(all(G))

printloop([])     = putStr("no") >> newline
printloop([A|As]) = browse(A) >> putStr("? ")
                              >> getChar >>= evalAnswer(As)

evalAnswer(As,';')  = newline >> printloop(As)
evalAnswer(As,'\n') = newline >> putStr("yes") >> newline
```

Here we make use of the monadic I/O concept discussed at the beginning of Section 3. The result of `browse(A)` is an action which prints a solution on the screen. Similarly, `putStr` and `newline` are actions to print a string or an end-of-line. `>>` and `>>=` are the sequential composition operators for actions [18]. The second argument of `>>=` must be a function which takes the result value of the first action and maps this value to another action. The expression "`evalAnswer(As)`" is a partially applied function call, i.e., it is a function which takes a further argument (a character) and produces an action: if the character is `';'`, the next solution is computed by a call to `printloop(As)`, and if the character is a `<return>` (`'\n'`), then the computation finishes with an action to print the string `"yes"`. Note that disjunctions do not occur in the `printloop` evaluation since potential non-deterministic computation steps of the goal `G` are encapsulated with `all(G)`.

Since the solutions for the goal are evaluated by `all` in a lazy manner, only those solutions are computed which are requested by the user. This has the advantage that the user interface to present the solutions (`printloop`) can be implemented independently of the mechanism to compute solutions. In an eager language like Prolog, the computation of the next solution must be interweaved with the print loop, otherwise all solutions are computed (which may not terminate) before the print loop is called, or only one standard strategy can be used. Our implementation is independent of the particular search strategy, since the following functions use the same top-level shell but bounded and breadth-first search to find the solutions:

```
prolog_bounded(N,G) = printloop(all_bounded(N,G))
prolog_bfs(G)       = printloop(all_bfs(G))
```

7 Related Work

This section briefly compares our operator for controlling non-deterministic computations with some related methods.

Prolog provides built-in predicates for computing all solutions, like `bagof`, `setof`, or `findall`. As shown in Section 4, they can be easily implemented with our control primitive, provided that all variables are existentially quantified. On the other hand, the search strategy in these predicates is fixed to Prolog's depth-first search and they always compute all solutions, i.e., they do not terminate if there are infinitely many solutions. In particular, they cannot be used in situations where not all solutions are immediately processed, like in an interactive shell where a demand-driven computation becomes important (cf. Section 6).

The lazy functional language **Haskell** [6] supports the use of list comprehensions to deal with search problems. List comprehensions allow the implementation of many generate-and-test programs, since logic programs with a strict data flow ("well-moded programs") can be translated into functional programs by the use of list comprehensions [17]. On the other hand, list comprehensions are much more restricted than our search operators, since purely functional programs do not allow the use of partially instantiated structures, and list comprehensions fixes a particular search strategy (diagonalization of the generators) so that other strategies (like best solution search) cannot be applied.

The higher-order concurrent constraint language **Oz** [16] provides a primitive operator to control search [15] similarly to ours. Actually, our operator `try` generalizes Oz's operator since `try` is not connected to a construct of the language (like `or` expressions in Oz) but its semantics is defined on the meaning of computation steps of the base language. This has an important consequence of the programming style and causes a significant difference between both concepts which should be explained in the following. An Oz programmer must explicitly specify in the program whether a search operator should later be applicable or not. A non-deterministic step can be performed in Oz only if an explicit disjunction (`or` or `choice`, see [14, 15]) occurs in a procedure. As a consequence, programs must be written in different ways depending on the use of search operators. The following simplified example explains this fundamental difference to our approach in more detail. Consider the multiplication with zero defined by the following rules:

```
mult(X,z) = z
mult(z,X) = z
```

Then expressions like `mult(z,z)` or `mult(add(z,z),z)` can be reduced to `z` with one deterministic reduction step using the first rule.[10]

In Oz, there are two implementation choices by using a conditional (`multc`) or a disjunction (`multd`):

```
proc {multc A B C}                proc {multd A B C}
  if B=z then C=z                     or B=z then C=z
  [] A=z then C=z                     [] A=z then C=z
  fi                                  ro
end                                 end
```

Conditionals commit to single computation branches, e.g., `{multc z z X}` reduces to the constraint `X=z`. However, we cannot use `multc` if we want to compute solutions to a goal like `{multc X Y z}` since the conditions in an `if` are only checked for entailment. Thus, we have to take the disjunctive formulation `multd` where we can compute a solution using some search operator [15]. On the other hand, the advantages of deterministic reductions are lost in `multd`, since the expression `{multd z z X}` is only computable with a search operator (a disjunction is not reduced until all but at most one clause fails [15]). Therefore, one has to implement `mult` twice to combine the deterministic reduction and search possibilities.

[10] Although one could also apply the second rule in this situation, sophisticated operational models for functional logic programming exploit the determinism property of functions: if a function call is reducible (i.e., a rule is applicable without instantiating arguments), then all other alternative rules can be ignored [2, 11].

In contrast to Oz, the definition of our control operator is based on the meaning of computation steps, i.e., the possible application of search operators does not influence the way how the basic functions or predicates are defined. This property keeps the declarative style of programming, i.e., function definitions describe the meaning of functions and control aspects are covered by search operators. Thus, functions or predicates can be defined independently of their later application, and explicit disjunctions are not necessary. The latter property also allows to write more predicates as functions which leads to potentially more efficient executions of programs. Furthermore, the laziness of Curry allows the implementation of search strategies independently of their application, e.g., demand-driven variants of search strategies (see [15]) are not necessary in our framework since the user interface can be implemented independently of the search strategy, as shown in Section 6.

8 Conclusions

We have presented a new primitive which can be added to logic languages in order to control the exploration of the search tree. This operator, which can be seen as a generalization of Oz's search operator [15], can be added to any logic-oriented language which supports equational constraints and existential quantification. In this paper, we have added it to the multi-paradigm language Curry and we have shown the advantages of Curry's lazy evaluation strategy to simplify the implementation of the different search operators. Since the search operators can be applied to any expression (encapsulated in a constraint), there is no need to translate functions into predicates or to use explicit disjunctions as in other approaches.

Since the definition of our control primitive is only based on an abstract view of computation steps (deterministic vs. non-deterministic steps), it can be applied to a variety of programming languages with a non-deterministic computation model, like pure logic or constraint logic languages (extended by existential quantifiers like in Prolog's `bagof/setof` construct), higher-order logic languages like λProlog [13] which already has explicit scoping constructs for variables, or the various functional logic languages which often differ only in the definition of a computation step. The general connection between search trees of the base language and the results computed by the search operators, which is also provided in this paper, supports the transfer of soundness and completeness results for the base language to corresponding results for the search operators.

The use of search operators supports the embedding of logic programs into other application programs where backtracking is not possible or too complicated (e.g., programs with side effects, input/output) since search operators allow the local encapsulation of search. Furthermore, they contribute to an old idea of logic programming by separating logic and control: the specification of functions or predicates becomes more independent of their use since the same function can be used for evaluation (computing values) or for searching (computing solutions) with various strategies without the necessity to define them in different ways. As shown in Section 6, this feature enables to simply replace the standard depth-first search by a bounded or breadth-first search in the user interface. This is quite useful to teach logic programming without talking about backtracking too early.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, Portland, 1994.
2. M. Hanus. Lazy Narrowing with Simplification. *Computer Languages*, Vol. 23, No. 2–4, pp. 61–85, 1997.
3. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
4. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. Technical Report, RWTH Aachen, 1998
5. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1998.
6. P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell (Version 1.2). *SIGPLAN Notices*, Vol. 27, No. 5, 1992.
7. J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topcis in Functional Programming*, pp. 17–42. Addison Wesley, 1990.
8. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proc. 1991 International Logic Programming Symposium*, pp. 167–183. MIT Press, 1991.
9. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
10. R. Loogen, F. Lopez 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*, pp. 184–200. Springer LNCS 714, 1993.
11. R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. *Theoretical Computer Science 142*, pp. 59–87, 1995.
12. G. Nadathur, B. Jayaraman, and K. Kwon. Scoping Constructs in Logic Programming: Implementation Problems and their Solution. *Journal of Logic Programming*, Vol. 25, No. 2, pp. 119–161, 1995.
13. G. Nadathur and D. Miller. An overview of λProlog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 810–827. MIT Press, 1988.
14. C. Schulte. Programming Constraint Inference Engines. In *Proc. of the Third International Conference on Principles and Practice of Constraint Programming*, pp. 519–533. Springer LNCS 1330, 1997.
15. C. Schulte and G. Smolka. Encapsulated Search for Higher-Order Concurrent Constraint Programming. In *Proc. of the 1994 International Logic Programming Symposium*, pp. 505–520. MIT Press, 1994.
16. G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pp. 324–343. Springer LNCS 1000, 1995.
17. P. Wadler. How to Replace Failure by a List of Successes. In *Functional Programming and Computer Architecture*. Springer LNCS 201, 1985.
18. P. Wadler. How to Declare an Imperative. In *Proc. of the 1995 International Logic Programming Symposium*, pp. 18–32. MIT Press, 1995.