Narrowing the Narrowing Space

Sergio Antoy
Zena M. Ariola

CIS-TR-97-04
April 1997

Department of Computer and Information Science
University of Oregon

Narrowing the Narrowing Space

Sergio Antoy
Portland State University

Zena M. Ariola
University of Oregon

Abstract

We introduce a framework for managing as a whole the space of a narrowing computation. The aim of our framework is to find a finite representation of an infinite narrowing space. This, in turn, allows us to replace an infinite enumeration of computed answers with an equivalent finite representation. We provide a semidecidable condition for this result. Our framework is intended to be used by implementations of functional logic programming languages. Our approach borrows from the memoization technique used in the implementation of functional languages. Since narrowing adds non-determinism and unifiers to functional evaluation, we develop a new approach based on graphs to memoize the outcome of a goal. We demostrate our framework with some examples and briefly discuss an experimental implementation.

1 Introduction

A fundamental problem in the integration of functional and logic programming is how to deal with the fact that the execution of a program may lead to the evaluation of a functional expression containing uninstantiated logic variables. Both narrowing and residuation have been proposed for this problem. Residuation may delay the evaluation of functional expressions that contain uninstantiated logic variables. Residuation is conceptually simple and relatively efficient, but incomplete, i.e., unable to compute the results of a computation in some cases. By contrast, narrowing evaluates functional expressions containing uninstantiated logic variables and is complete if an appropriate strategy is chosen. For this reason narrowing has been widely proposed [4, 9, 20] and implemented [5, 14, 17, 19, 31, 34] for the integration of the two paradigms. However, narrowing has the propensity to generate infinite search spaces. When this situation arises, narrowing becomes incomplete in practice in the sense that it cannot compute, with finite resources, the complete solution of a goal. This paper partially fixes this problem.

Recent years have seen the discovery of many narrowing strategies, e.g., [2, 3, 6, 8, 11, 12, 13, 14, 16, 18, 23, 24, 28, 29, 30, 31, 32, 34, 35]. In particular, the *needed narrowing* strategy [2] is both complete and optimal for the inductively sequential rewrite systems, a class that encompasses the first order component of functional languages such as ML [22] and Haskell [25]. The *parallel narrowing* strategy [3] is complete and locally optimal for an even larger

This work has been supported in part by the National Science Foundation under grants CCR-9406751, CCR-9410237, CCR-9624711.

Authors' addresses: Sergio Antoy, Department of Computer Science, Portland State University, P.O. Box 751, Portland, OR 97207, U.S.A., antoy@cs.pdx.edu; Zena M. Ariola, Department of Computer and Information Science, University of Oregon, Eugene, OR 97405, U.S.A., ariola@cs.uoregon.edu.

and more declarative class of systems. These results seem to suggest that the contribution of narrowing strategies alone to the efficient execution of functional logic computations has reached its theoretical limit. Yet, in some situations easily arising in practice, the application of these strategies produces outcomes that leaves much to desire, as we will show in some examples. The focus of this note is in "managing" a narrowing strategy by aiming at a finite representation, in the form of a possibly cyclic graph, of the narrowing space of a goal. While there is no guarantee that we will succeed, if we do we are able to provide a simple finite representation of the set of the goal's computed expressions.

The paper is organized as follows. Section 2 recalls some notations and concepts. Section 3 describes our framework. More specifically, we discuss how to finitely represent the possibly infinite narrowing space of a goal, and prove soundness and completeness of our representation. We also investigate known and new techniques for increasing the chances of obtaining a finite representation of a narrowing space. We continue by showing how to obtain a finite representation of an infinite set of computed expressions from a finite representation of a narrowing space. Section 4 discusses a prototypical tool that we implemented to experiment with our framework. Section 5 contains our conclusions. There is no explicit comparison with previous work, as we are not aware of any previous effort in this area.

2 Preliminaries

Rewriting, see [10, 27] for tutorials, is a computational paradigm convenient to study functional and functional logic computations. A rewrite program is a set of rewrite rules or oriented equations, pairs of terms denoted by $l := r$, where l is a pattern and all the variables of r are in l. Rewriting a term t into u, written as $t \to u$, is the operation of obtaining u by replacing in t an instance of some rule left-hand side l with the corresponding instance of the right-hand side r. For example, consider the program that defines addition on the natural numbers represented in unary notation,

```
0 + y   := y
s x + y := s (x + y)
```

and term t defined as $s\ (s\ 0) + 0$. According to the second rule, instantiated by $\{x \mapsto s\ 0, y \mapsto 0\}$, term t rewrites into $s\ (s\ 0 + 0)$. Additional rewrite steps eventually yield $s\ (s\ 0)$ which is a normal form, i.e., cannot be rewritten and is understood as the result of the computation.

Narrowing differs from rewriting by using unification instead of pattern matching, but is identical to rewriting in most other aspects. For example, term t defined as $u + 0$ is narrowed into 0, written as $u + 0 \leadsto_{\{u \mapsto 0\}} 0$, as follows. First, term t is instantiated to $0 + 0$ by $\{u \mapsto 0\}$. Then, $0 + 0$ is rewritten as usual. Choosing instantiations and rewrites is the task of a narrowing strategy. Narrowing is often used in functional logic programming for its ability to solve equations, i.e., computing unifiers with respect to an equational theory [13]. For example, consider the equation $u + s\ 0 = s\ (s\ 0)$. The second rule is applied to the equation by instantiating u to $s\ w$, obtaining $s\ (w + s\ 0) = s\ (s\ 0)$. Then, the first rule is applied by instantiating w to 0. The resulting equation, $s\ (s\ 0) = s\ (s\ 0)$, is trivially (syntactically) true. Thus, the composition $\{u \mapsto s\ w\} \circ \{w \mapsto 0\}$, i.e., $\{u \mapsto s\ 0\}$, is the equation's solution.

The functional expressions narrowed in the examples presented in this note are boolean

expressions and are referred to as *goals*. Considering only goals is not a limitation. To evaluate a functional expression t, regardless of its type, we solve the equation (goal) $t = x$, where x is a new variable. The equality symbol "=" is an overloaded operator defined by a few rules for each type. Below we show these rules for the natural numbers.

```
0 = 0     := true
s _ = 0   := false
0 = s _   := false
s x = s y := x = y
```

This definition, known as *strict equality*, is more appropriate than syntactic equality when computations may not terminate. It is easy to generalize the above rules to other types.

Our framework is largely independent of the narrowing strategy. However, in presenting our examples we will employ the *needed narrowing* strategy [2], due to its soundness, completeness, and optimality with respect to a class of rewrite systems underlying many functional logic programs.

3 The Framework

Memoization [33] is a technique aimed at improving the performance of functional languages. A memoized function remembers the arguments to which it has been applied together with the result it generates on them. If it is applied to the same arguments again it returns the stored result rather than repeating the computation. Although memoization entails an overhead, it may provide substantial benefits. For example, consider the program that defines the Fibonacci function:

```
fib 0         := 0
fib (s 0)     := s 0
fib (s (s x)) := fib (s x) + fib x
```

It is easy to see that the computational complexity of *fib* is exponential in its argument. The culprit is the third rule, since for any $n > 1$ it requires twice the computation of *fib* $(n-2)$. Memoization has a dramatic effect on the complexity of *fib*. Once either one of the two addends originating from the right-hand side of the third rule is computed, the other addend is computed in constant time. Thus, the computational complexity of *fib* changes from exponential to linear.

Extending memoization to narrowing is not straightforward, since the result of a narrowing computation can be an infinite collection of substitutions or computed expressions. This situation creates new problems, but also the opportunity for benefits greater than those arising in purely functional computations.

We introduce our framework with an example. Let us extend the definition of addition given earlier with the rules defining the usual "less than or equal to" relational operator.

```
0 <= y     := true
s x <= 0   := false
s x <= s y := x <= y
```

Consider the goal $u \leq u + v$, where u is an uninstantiated variable. The narrowing compu-

tation of this goal non-deterministically takes either of two paths:

$$u \leq u + v \overset{+}{\leadsto}_{\{u \mapsto 0\}} \mathit{true} \qquad \text{or} \qquad u \leq u + v \overset{+}{\leadsto}_{\{u \mapsto s\ w\}} w \leq w + v$$

(the superscript + stands for one or more narrowing steps). The second path yields a goal equal to the original one, except for a renaming of w, hence the goal's narrowing space is infinite. It is easy to verify that a complete narrowing strategy computes on $u \leq u + v$ the infinite set of substitutions $\{0, s\ 0, s\ (s\ 0), \ldots\}$. If, during this computation, we recognize that $w \leq w + v$ is (a variant of) a problem that has been already tackled, we achieve two major advantages: We save a good deal of computation, and we obtain a finite representation of the narrowing space. This representation is shown in the left part of Figure 1.

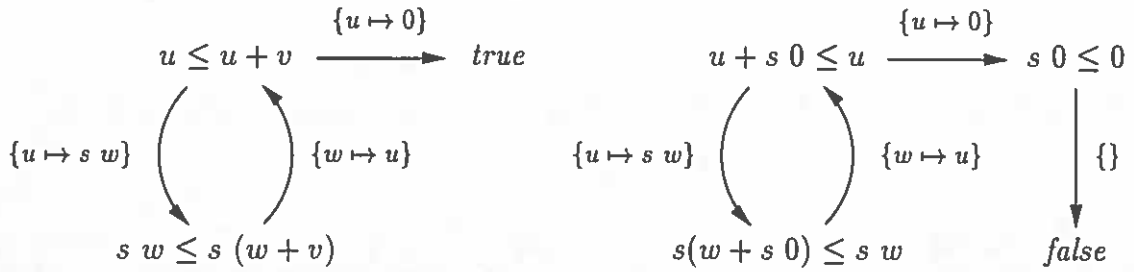

Figure 1: Graph representation of the narrowing space of two goals. An edge is associated to a narrowing step and is labeled by the step substitution composed with a permutation.

A finite representation of the narrowing space has the potential of replacing an infinite computation with a finite one. By analyzing the finite representation of the narrowing space we infer that our goal is satisfied for all u. This allows us to replace an infinite enumeration of the goal's solutions with a single, simpler, more general solution. Equally important, we may discover that a goal has no solutions. For example, if we apply the same reasoning to the goal $u + s\ 0 \leq u$, we obtain the finite representation shown in the right part of Figure 1. From it we infer that the goal has no solutions. By contrast, the direct application of a narrowing strategy to this goal keeps looking forever for a solution that does not exist.

3.1 Space Representation

There are two differences between the functional evaluation of an expression and narrowing which affect how to memoize functional logic computations. The first difference concerns the outcome of a narrowing computation. In functional logic programming one is interested in narrowing computations of the form $t_0 \leadsto_{\sigma_1} t_1 \leadsto_{\sigma_2} \cdots \leadsto_{\sigma_n} t_n$, where t_n is a constructor term. The result of this computation is the *computed expression* $\sigma_1 \circ \cdots \circ \sigma_n \llbracket t_n$, i.e., t_n is the normal form of $\sigma_n(\cdots \sigma_1(t_0) \cdots)$.

The second relevant difference concerns non-determinism. Functional computations are *don't care* non-deterministic, i.e., for a complete evaluation strategy, all choices lead to the same result. Functional logic computations are *don't know* non-deterministic, i.e., different choices lead to possibly different results, as shown in Figure 2. Narrowing computations are not linear sequences of steps, but rather trees of steps whose branches often cannot be joined together.

- *If $g \overset{+}{\leadsto}_\delta g'$, where g' is a data term, is a narrowing derivation computed by $\mathcal{S}$, then there exists a path P in G that connects g to g' and computes a substitution equivalent to δ. (Completeness)*

Proof Sketch (Soundness) The proof is by induction on the length of P. Base case: P consists of single edge. Since g and g' are distinct vertices of G and g' is either *true* or *false*, the claim is immediate. Ind. case: Let P consist of an initial edge (g, t) with label ρ followed by a non-empty path P' that computes substitution ρ'. By definition of graph representation, there exists a permutation μ such that $\rho = \sigma \circ \mu$ and $g \leadsto_\sigma \mu^{-1}(t)$ is a narrowing step. By the induction hypothesis, there exists a substitution $\bar{\rho}'$ equivalent to ρ' such that $t \overset{+}{\leadsto}_{\bar{\rho}'} g'$. $t \overset{+}{\leadsto}_{\bar{\rho}'} g'$ implies that $\mu^{-1}(t) \overset{+}{\leadsto}_{\mu \circ \rho''} g'$, for some substitution ρ'' equivalent to $\bar{\rho}'$. Since the equivalence of substitutions is a transitive relation, ρ' and ρ'' are equivalent as well and there exists a permutation τ such that $\rho' = \rho'' \circ \tau$. Thus there exists a narrowing derivation of g to g' that computes the substitution $\sigma \circ \mu \circ \rho''$. Since $\sigma = \rho \circ \mu^{-1}$, this is $\rho \circ \rho''$, i.e., $\rho \circ \rho' \circ \tau$ which is equivalent to the substitution computed by P. (Completeness) By definition of graph representation, for each step of $g \overset{+}{\leadsto}_\delta g'$ computed by $\mathcal{S}$ there is a corresponding edge in G, hence there is a path P, connecting g to g', associated to the entire derivation. Using a technique similar to that used in the proof of soundness, it can be verified that P computes a substitution equivalent to δ. □

There are a number of options to consider when building the narrowing space of a goal. One is whether to look for a single computed expression or for an enumeration of computed expressions. The other is whether to construct the narrowing space depth-first (for efficiency) or bread-first (for completeness). Some of these options could be left to the programmer via annotations in a program or could be decided from program analysis. An experiment that we performed with a prototypical tool has shown some advantages of iterative deepening. In particular, this strategy finds the first solution relatively quickly without losing completeness and it may find all the solutions of a problem without risking non-termination.

3.2 Space Analysis

The graph representation of the narrowing space of a computation may allow us to infer properties of the entire computation. Every path from a goal to a constructor term gives us a computed expression. Cycles in the graph representation of a narrowing space are particularly interesting, since they finitely represent infinite sets of computed expressions. For example, let G denote the graph representation of the narrowing space of $u \leq u + v$, which is shown in the left part of Figure 1. Any path of G consists of zero or more traversals of the loop followed by the final edge reaching *true*. The substitution computed by this path is $\{u \mapsto s^n\ 0\}$, where n is the number of loop traversals. In the above notation, following a common practice, if f is a function from a type T into T, then $f^n\ x$, $n > 0$, stands for $f^{n-1}\ (f\ x)$, and $f^0\ x = x$, for all $x \in T$. Thus, we conclude that goal $u \leq u + v$ is solved for any natural number u, or in other words that the goal's computed expressions is $\{\} \,[\!]\, \mathit{true}$. By contrast, plain narrowing enumerates an infinite set of ground computed expressions of this goal.

Non-determinism leads to a large, possibly infinite, set of results that is inconvenient or impossible to store directly.

This consideration suggests to initially memoize the results of a single narrowing step of a goal, rather than trying to accumulate the entire set of its computed expressions. We choose to store this information as a graph. The vertices of the graphs are goals, i.e., terms being narrowed, the edges are narrowing steps between these goals. Generally, we are interested only in the substitution of a narrowing step. Thus, we discard the rule and the position of the step and we label an edge with the step's substitution. Terms that differ only for a renaming of variables are considered to be the same vertex. This decision, of course, raises some concerns that we will address shortly.

Definition 1 Let g be a goal. A *graph representation* of the narrowing space of g computed by a strategy $\mathcal{S}$ is a finite rooted directed labeled graph G such that g is the root vertex of G, and if t is a vertex of G, $t \leadsto_\sigma t'$ according to strategy $\mathcal{S}$ iff for some permutation μ, $\mu(t')$ is a vertex of G and there is an edge in G from t to $\mu(t')$ with label $\sigma \circ \mu$. ■

There may exist many graph representations of the narrowing space of a goal. Representations with a smaller number of vertices are more desirable in our framework. A procedure that from a goal g attempts to construct a graph representation of the narrowing space of g is straightforward to implement from Definition 1.

We wish to reason about the narrowing derivations of a goal g by unfolding (traversing paths of) a graph representation, when it exists, of g's narrowing space. Since a graph may identify terms that differ for a renaming of variables, it could happen that the "derivations" that we unfold from the graph do not belong to the narrowing space of g. In fact, in general, this indeed happens. Consider a program that computes the leftmost decoration of a binary tree.

```
leftmost (leaf x)     := x
leftmost (branch l _) := leftmost l
```

A graph representation of the narrowing space of the goal *leftmost* $t = 0$, where t is an uninstantiated variable, has an edge beginning and ending at the goal itself with label $\{t \mapsto$ *branch* t _$\}$. This edge does not correspond to a narrowing step, since the unifier of a narrowing step is an idempotent substitution. However, this is not a problem for derivations ending in a constructor term, which are the only derivations that we care about.

The *narrowing space* of a goal g computed by a strategy $\mathcal{S}$ is the set of the narrowing derivations starting from g whose steps are computed by $\mathcal{S}$. Since every time that we use a rule R in a step we consider a variant of R with new variables, narrowing derivations that differ only for a renaming of these variables compute *equivalent* substitutions and are considered to be the same derivation. If G is a labeled graph whose edges are labeled by substitutions, the substitution *computed* by a path P of G is the composition of the labels of P's edges.

Proposition 1 *Let G be a graph representation of a goal's g narrowing space computed by a strategy $\mathcal{S}$.*

- *If P is a non-empty path of G that connects g to a data term g' and computes δ, then $g \overset{+}{\leadsto}_{\delta'} g'$, for some δ' equivalent to δ. (Soundness)*

Unfortunately, it does not always seem possible to simplify an infinite set of substitutions to a single, more general substitution. For example, consider the following program

```
double 0      := 0
double (s x) := s (s (double x))

half 0          := 0
half (s 0)      := 0
half (s (s x)) := s (half x)
```

and the goal $double\ (half\ u) = u$, where u is an uninstantiated variable. The narrowing space of this goal is shown in Figure 2. Using the notation discussed earlier, we finitely represent the set of computed expressions of this goal with the following two computed expression-like formulas $\{u \mapsto s^{2n}\ 0\} \,[]\, true$ and $\{u \mapsto s^{2n+1}\ 0\} \,[]\, false$, where n ranges over $\mathbb{N}$. These formulas are clear and intuitive, but ad hoc to this example.

To obtain a finite representation of the set of computed expressions of a goal g, when g's narrowing space has a graph representation, we introduce a new concept. We regard a graph representation of a narrowing space as a finite state machine—both are finite rooted directed labeled graphs. If we apply to a graph representation of a narrowing space a standard algorithm for the construction of a regular expression associated to a finite state machine, we obtain expressions, which by analogy with regular expressions, we call *regular substitutions*. Regular substitutions represent possibly infinite sets of substitutions. If we use regular substitutions, instead of plain ones, in computed expressions we get *regular computed expressions*, which represent possibly infinite sets of computed expressions. The notation is familiar and fairly intuitive. For example, the infinite set of computed expressions of the goal of Figure 2 is entirely represented by the two regular computed expressions $\{u \mapsto s\ (s\ u)\}^* \circ \{u \mapsto 0\} \,[]\, true$ and $\{u \mapsto s\ (s\ u)\}^* \circ \{u \mapsto s\ 0\} \,[]\, false$.

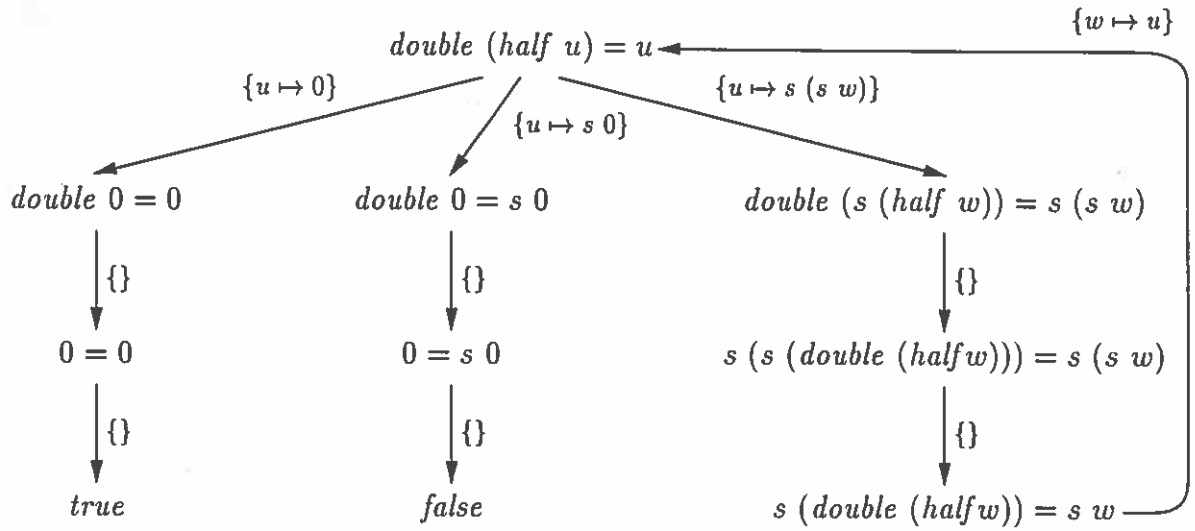

Figure 2: Graph representation of the narrowing space of $double\ (half\ u) = u$.

Definition 2 *Regular substitutions*, *rs* for short, are expressions defining sets of substitutions as follows: If σ is a substitution, then the regular substitution σ denotes the set of substitutions $\{\sigma\}$. It will be clear from the context whether we talk about σ as a regular substitution or a

substitution. If σ and η are regular substitutions, then

$(\sigma) \mid (\eta)$	is an rs denoting the set $\sigma \cup \eta$
$(\sigma)(\eta)$ or $(\sigma) \circ (\eta)$	is an rs denoting the set $\{\sigma' \circ \eta' \mid \sigma' \in \sigma, \eta' \in \eta\}$
$(\sigma)^*$	is an rs denoting the set $\{\{\}\} \cup \sigma \cup \sigma \circ \sigma \cup \cdots$

A *regular computed expression*, *RCE* for short, of a goal g is a pair $\sigma \,[\!]\, t$, where σ is a regular substitution and, for all $\sigma' \in \sigma$, $\sigma' \,[\!]\, t$ is a computed expression of g. The use of parentheses can be reduced using standard conventions on precedence and associativity of regular expression operators. ■

Proposition 2 *If the narrowing space of a goal g has a graph representation, then the set S of computed expressions of g has a finite representation.*

Proof Sketch Regard the graph representation of the narrowing space of g as a finite state machine M where the states are goals, the initial state is g, the final states are constructor normal forms, the moves are narrowing steps between the goals (modulo renaming), and the inputs are the substitutions of the above narrowing steps. The elements of S are all and only the pairs $\sigma \,[\!]\, n$, where n is a final state and σ is a regular expression defining the inputs accepted by M that terminate in n. Since the number of final states of M is finite, the set S is finite as well. □

A semidecision procedure for the existence of a finite representation of the set S of computed expressions of a goal g is trivial. First, attempt to construct a graph representation G of g's narrowing space. If this construction terminates, compute the set S of regular computed expressions accepted by G. Propositions 1 and 2 are the basis for the correctness of this procedure.

3.3 Memoization

Representing the narrowing space of a goal as a graph and regarding this graph as a finite state machine allows us to obtain a finite representation of the goal's computed expressions. This is crucial to memoization in functional logic programming. If we have to solve a goal a second time, it would be unnecessarily wasteful to analyze the graph again to retrieve the goal's computed expressions. To obtain a performance similar to that of memoization in functional languages, we use the graph representation of a goal's narrowing space to compute a table-like structure in which each vertex of the graph is associated to its set of computed expressions. This computation is straightforward and we show the result of its application to the graph of Figure 2. As in the previous section, we use the intuitive notation of RCEs.

We could memoize functional logic computations also without our framework. However, without our framework some difficulties arise. If narrowing a goal yields a sequence of computed expressions that are not all computed at the same time, then the association between a goal and its computed expressions is dynamic. Creating and maintaining a dynamic association is computationally more complicated and expensive. Retrieving the association at different times might produce different results, later results could be more informative than earlier results, since more computed expressions may become known, a situation that we think is undesirable in a declarative environment.

$double\ (half\ w) = w$ $double\ (s\ (half\ w) = s\ (s\ w)$ $s\ (s\ (double\ (half w))) = s\ (s\ w)$ $s\ (double\ (half w)) = s\ w$	$\{\{w \mapsto s^{2n}\ 0\} [\!] true, \{w \mapsto s^{2n+1}\ 0\} [\!] false\}$, for $n \in \mathbb{N}$
$double\ 0 = 0$ $0 = 0$	$\{\{\} [\!] true\}$
$double\ 0 = s\ 0$ $0 = s\ 0$	$\{\{\} [\!] false\}$

Figure 3: Memoization of the goals occurring in the narrowing space of Figure 2.

3.4 Simplification

It is well known [21] that simplification rules reduce the size of the narrowing space of certain goals. A simplification rule is a rewrite rule used to perform deterministic steps during a narrowing derivation. For example, referring to the program of Section 2, we can extend the definition of the addition operator with the following simplification rules.

```
y + 0     := y
y + (s x) := s (y + x)
```

Simplification rules are beneficial to reduce the size of a goal's narrowing space only when, in a narrowing step, they are used in place of, rather than in addition to, standard rules. To preserve the completeness of narrowing when simplification rules are used in this fashion, no term can have an infinite derivation in which only simplification rules are applied, a condition not always easy to ensure in practice. Not surprisingly, it turns out that simplification rules are also useful in our approach.

For example, the narrowing space of the goal $u + v = v + u$ is infinite and does not have a finite representation in our framework. The reason is that goals of the form $s^n\ u = u + s^n\ 0$, for increasing values of n, keep being created. However, if we use the above simplification rules, we obtain a (finite) graph representation of the narrowing space. This graph is small and simple, and it allows us to infer the computed expression $\{\} [\!] true$.

Since our framework thrives on cycles, it is interesting to explore the effects of non-terminating simplification rules such as

```
x+y := y+x                                    (1)
```

that subsumes both the simplification rules for addition proposed earlier, but cannot be used in the classic approach [21]. If we use this rule *in place* of the defining rules of "+" we immediately get a cycle, but nothing else. If we use this rule *in addition* to the defining rules of "+" we increase the out-degree of many vertices when we attempt to compute the graph representation of the narrowing space. Neither alternative seems profitable. To benefit from using the above rule, it is necessary to perform a more sophisticated analysis of the narrowing space. This will be discussed at the end of the next section.

3.5 Fertilization

It is clear from the previous section that techniques that prune the narrowing space are doubly beneficial in our framework, since they may prune portions of space that have no finite representation. Pruning all these portions makes the difference between a finite and an infinite representation of a goal's narrowing space. A powerful technique to this aim has its roots in induction. To introduce this technique, consider again our first goal, $u \leq u + v$. We can prove it by induction on u as follows. There are two cases. Base case: Prove the goal for $u = 0$. Ind. case: Prove the goal for $u = s\ w$ assuming $w \leq w + v$. Both cases are proved directly by rewriting. The analogy with a narrowing computation is striking. During the construction of a graph representation, we use an induction hypothesis when we find in the graph being constructed a variant of the current goal. In the following we show that when the goal is an equation, it is possible to do better.

A *recursive* constructor is a data constructor of a type T that has an argument of type T. For example, *successor* and *cons* are recursive constructors of natural numbers and lists, respectively. Automated theorem provers, e.g., [7, 15], recognize recursive constructors and create induction hypotheses. When the goal is an equation, theorem provers apply an induction hypothesis by replacing in the current goal an instance of the equation's left-hand side with the corresponding instance of the right-hand side or vice versa. This operation is called "fertilization" in [7]. We show, in Figure 4, how fertilization allows us to finitely represent the narrowing space of $u + v = v + u$. Fertilization involves the terms connected by

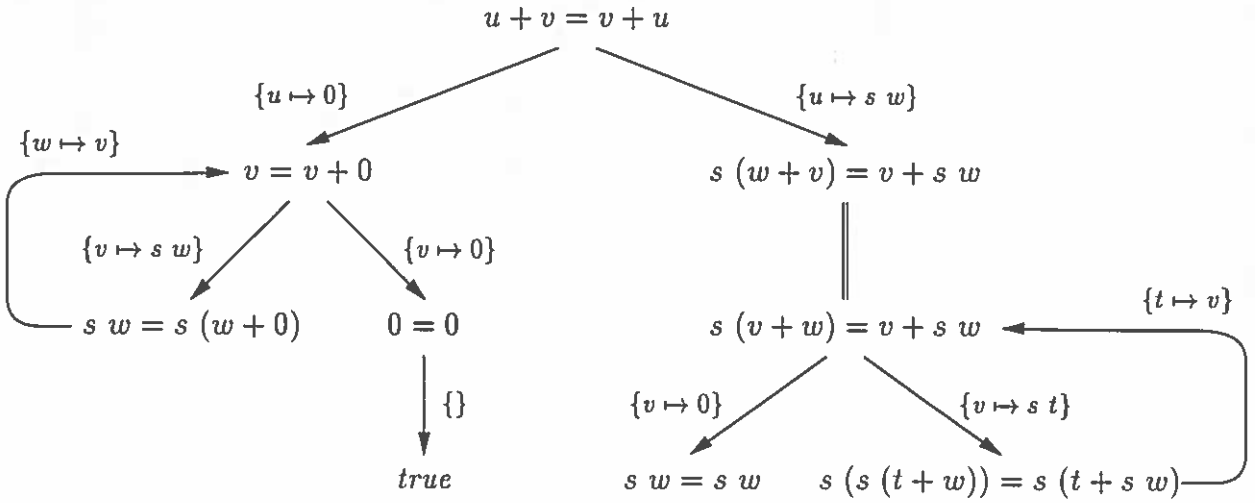

Figure 4: Partial graph representation of the narrowing space of $u + v = v + u$. The narrowing space of $s\ w = s\ w$, omitted from the figure, has a graph representation and computed expression $\{\} \,[\!]\, \mathit{true}$. To contain the size of the figure, some steps are performed in parallel. The double vertical line denotes the fertilization step.

a double vertical line. The equation $s\ (w + v) = v + s\ w$ can be considered the reduct of the inductive case of a proof that $w + v = v + w$. Hence, the latter is an inductive hypothesis. Consequently, we can replace $w + v$ with $v + w$ and we obtain $s\ (v + w) = v + s\ w$, which turns out to be easier to solve. Note that, in the replacement of terms originating from fertilization, variables must be intended as Skolem constants, thus their names matter.

The practicality of fertilization for the execution of functional logic programs will have to be assessed. However, fertilization has some immediately apparent potential advantages with respect to simplification. Simplification rules must be coded by the programmer, whereas fertilization rules originate spontaneously during the execution of a program. When a goal is being narrowed, we expect that only a small number of fertilization rules will originate, a number definitely smaller than the cardinality of a useful set of simplification rules. Furthermore, each simplification rule must be tried on every subgoal, whereas fertilization rules are applied more selectively. In [7] a fertilization rule is used only once and then discarded, a policy that further speeds up a computation. Finally, simplification rules must be terminating, whereas fertilization rules have no such a requirement.

Earlier we considered $x+y = y+x$ as a simplification rule. Using this rule, in addition to the defining rules of "+", to solve $u + v = v + u$, yields a representation, not necessarily a finite one, of the goal's narrowing space that obviously embeds the graph of Figure 4. By analyzing a large enough portion of this representation, we discover, as we did earlier that $\{\} [] \mathit{true}$ is a computed expression. Consequently, we are able to find a finite representation of the set of the goal's computed expressions, even if the goal's narrowing space has no finite representation in our framework. Here is where iterative deepening pays off. We work on the construction of a graph representation of a goal's narrowing space until a predetermined amount of resources has been consumed. Then we analyze the possibly incomplete graph representation of the narrowing space. If we determine that all the goal's computed expressions have been found, there is no need to complete the construction of the graph. In this way, we succeed in finding a finite representation of a goal's set of computed expressions even for some goals that do not have graph representations of their narrowing spaces. For example, this situation happens for $u + v = v + u$ when we use the non-terminating simplification rule of display (1). Note that, according to the rules defining "=", the goal $u + v = u + v$ is not solved, though the sides of the equation are syntactically equal.

3.6 RCE Calculus

Up to this point, for the sake of intuition and readability, we adopted a cavalier attitude toward the computation and presentation of the finite representation of an infinite set of computed expressions. Presenting to the user precise and easy to read RCEs is a challenging problem.

From a graph representation of a goal's narrowing space, we obtain a finite representation of the goal's set of computed expressions with any algorithm that computes the regular expression accepted by a finite state machine. The major difference with respect to standard regular expressions is that the alphabet symbols are replaced by substitutions and the concatenation of symbols is replaced by the composition of substitutions. Regular expressions can be simplified to be more readable. These simplifications improve the readability of RCEs as well. In addition, substitutions are objects much richer than alphabet symbols and their composition gives rise to a whole new class of simplification rules. We discuss some of these rules and show how their application allows us to determine, mechanically and rather easily, that the computed expression of $u + v = v + u$, the goal of Figure 4, is $\{\} [] \mathit{true}$.

Figure 1 shows a common pattern for recursive types, such as natural numbers, lists, and trees. This pattern consists of a loop with an exit. While many goals may have complicated

graph representations of their narrowing spaces, often these graphs embed a loop with an exit. For example, this situation occurs twice in Figure 4 and one more time in its omitted portion. We generalize these loops with exits as follows. Let T be a recursive type. Partition the constructors of T into a set $\{r_1, \ldots r_i\}$ of recursive constructors, a notion discussed in Section 3.5, and a set $\{n_1, \ldots, n_j\}$ of non-recursive constructors. If c is a constructor of T, let $\bar{c}$ denote the *linear* term $c(v_1, \ldots, v_n)$ where n is the arity of c and $v_1, \ldots, v_n$ are distinct variables. It can be verified by induction that the *regular substitution*

$$(\{u \mapsto \bar{r}_1\} \mid \ldots \mid \{u \mapsto \bar{r}_i\})^* \circ (\{u \mapsto \bar{n}_1\} \mid \ldots \mid \{u \mapsto \bar{n}_j\}) \tag{2}$$

defines all the instances of type T and consequently can be simplified to the identity substitution. Referring to the left part of Figure 1, we simplify $(\{u \mapsto s\ w\} \circ \{w \mapsto u\})^* \circ \{u \mapsto 0\}$ to the identity after explicitly composing the two substitutions in parentheses.

A second useful simplification rule, which we use to simplify the RCE computed for the goal of Figure 4, is the following. Let T be a type and u a variable of type T. Consider the regular substitution

$$\{u \mapsto t_1\} \mid \ldots \mid \{u \mapsto t_n\} \tag{3}$$

If the set of terms $\{t_1, \ldots, t_n\}$ is complete for (the constructors of) T in the sense of [26], then the regular substitution of display (3) defines all the instances of type T [26, Lemma 3], and consequently can be simplified to the identity substitution. Referring to Figure 4, we first simplify the substitution of the goal's RCE by repeatedly applying the simplification rule of diplay (2) and obtain $\{u \mapsto 0\} \mid \{u \mapsto s\ w\}$. Since the set $\{0, s\ w\}$ is complete for the type of u, using the simplification rule of display (3) we reduce this regular substitution to the identity.

While finding "the simplest" presentation of an RCE may be impossible or impractical, we obtain good improvements with the above simplification rules.

4 Implementation

We have implemented a prototypical tool to experiment with the ideas presented in this paper. Although rudimentary, this tool highlights both the potential benefits and some difficulties of a practical application of our framework. At the time of the writing, our tool attempts to construct a graph representation of a goal's narrowing space. This allows us to easily discover if goals more complicated than those discussed in this note have graph representations of their narrowing spaces. It also allows us to investigate the effects of both terminating and non-terminating simplification rules.

The tool demonstrates that our framework is extremely effective in some situations. It also points in the direction of a new intriguing area of investigation. In nearly all cases where a goal has no finitely representable narrowing space, we were able to generalize infinite sets of subgoals, such as $s^n\ u = u + s^n\ 0$ which arises in the space of $u + v = v + u$ computed without using simplification and/or fertilization rules. A relatively simple and efficient algorithm for this generalization in described in [1]. This generalization is valuable, since it shows that simply performing further narrowing steps to find a graph representation of the narrowing

space is useless. One could use this information as a trigger for other options, if they are available, e.g., to apply simplification or fertilization rules, to analyze the space constructed up to that point to see whether all the computed expressions have been found already, or simply to give up a hopeless effort and settle for plain narrowing.

5 Conclusions

We have presented a framework for narrowing akin to memoization in functional programming. The key feature of our framework is its ability, in some cases, to provide a finite representation of an infinite narrowing space. This, in turn, allows us to finitely represent the set of a goal's computed expressions and to associate this set to the goal. This association plays the same role of the association of a computation with its result implemented by memoizing functional languages.

A major advantage of our framework is its potential to find computed expressions more efficiently than plain narrowing alone and/or to find computed expressions that are more general than those obtained by plain narrowing alone. These two features are closely interrelated. Referring to the first example of Figure 1, consider the goal

$$u \leq u + v \ \wedge member(u, l)$$

where u is an uninstantiated variable, "$\wedge$" is the conjunction operator, and *member* is a predicate that checks whether its first argument, an element, occurs in its second argument, a list of elements. If l is a long list of big numbers, plain narrowing solves this goal inefficiently, regardless of which goal is selected first, since a large number of narrowing steps are needed in each case. Our framework speeds up this computation tremendously, since only a handful narrowing steps are needed. This speed up is achieved without asking the programmer to supply evaluation annotations or mode declarations or other similar devices that may make a program less general and/or less declarative and/or more difficult to understand and code.

Equally important, our framework may quickly find that a goal cannot be solved (its computed expression is $\{\} \,[\!]\, \mathit{false}$) where plain narrowing alone searches forever. The right part of Figure 1 contains an example that proves this point.

We have shown how our framework accommodates techniques, or even extends them and promotes new ones, intended to reduce the size of the narrowing space.

An interesting aspect of our framework is the language used to finitely represent possibly infinite sets of substitutions. This language is substantially identical to that of regular expressions. Since there are languages more powerful than regular expressions, our framework might fail in situations where a different framework could succeed. However, we believe that these situations are indeed rare in practice and that the simplicity of the language is an asset. We have shown that in addition to the well-know simplification rules of regular expressions, our framework can take advantage of new simplification rules specific of substitutions. We have only superficially discussed how to present RCEs to the user. This appears to be mostly a syntactic, though non trivial, issue. A more substantial problem is how to use RCEs internally when their substitutions cannot be simplified to usual ones, but contain the "or" or the Kleene closure operators. This could be solved by extending the notion of a term.

The implications of this extension on unification and other components of a functional logic language require further study.

References

[1] D. Angluin and C. Smith. Inductive inference: Theory and methods. *Computing Surveys*, 15(3):237–269, 1983.

[2] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994. Extended version available at URL `ftp://ftp.cs.pdx.edu/pub/faculty/antoy/An-Optimal-Narrowing-Strategy.ps.Z`.

[3] S. Antoy, R. Echahed, and M. Hanus. A parallel narrowing strategy. In *14th Int'l Conference on Logic Programming*, Leuven, Belgium, July 1997. (to appear) Extended version available at URL `ftp://ftp.cs.pdx.edu/pub/faculty/antoy/A-Parallel-Narrowing-Strategy.ps.Z`.

[4] M. Bellia and G. Levi. The relation between logic and functional languages: a survey. *Journal of Logic Programming*, 3(3):217–236, 1986.

[5] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *ESOP-86*, pages 119–132. Springer LNCS 213, 1986.

[6] A. Bockmayr, S. Krischer, and A. Werner. An optimal narrowing strategy for general canonical systems. In *Proc. of the 3rd Intern. Workshop on Conditional Term Rewriting Systems*, pages 483–497. Springer LNCS 656, 1992.

[7] R.S. Boyer and J.S. Moore. Proving theorems about LISP functions. *JACM*, 22(1):129–144, Jan. 1975.

[8] J. Darlington and Y. Guo. Narrowing and unification in functional programming - an evaluation mechanism for absolute set abstraction. In *Proc. of the Conference on Rewriting Techniques and Applications*, pages 92–108. Springer LNCS 355, 1989.

[9] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, 1986.

[10] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North Holland, Amsterdam, 1990.

[11] R. Echahed. On completeness of narrowing strategies. In *Proc. CAAP'88*, pages 89–101. Springer LNCS 299, 1988.

[12] R. Echahed. Uniform narrowing strategies. In *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 259–275, Volterra, Italy, September 1992.

[13] M. J. Fay. First-order unification in an equational theory. In *Proc. 4th Workshop on Automated Deduction*, pages 161–167, Austin (Texas), 1979. Academic Press.

[14] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 172–184, Boston, 1985.

[15] S.J. Garland and J.V. Guttag. Inductive methods for reasoning about abstract data types. In *ACM SIGACT-SIGPLAN Symposium of Principles of Programming Languages*, pages 219–228, 1988.

[16] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: a logic plus functional language. *The Journal of Computer and System Sciences*, 42:139–185, 1991.

[17] J. A. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pages 295–363. Prentice Hall, 1986.

[18] W. Hans, R. Loogen, and S. Winkler. On the interaction of lazy evaluation and backtracking. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 355–369. Springer LNCS 631, 1992.

[19] M. Hanus. Compiling logic programs with equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.

[20] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[21] M. Hanus. Lazy narrowing with simplification. *Computer Languages (to appear)*, 1997.

[22] B. Harper. Introduction to Standard ML. Technical report, ECS-LFCS-86-14, Laboratory for the Foundation of Computer Science, Edinburgh University, 1986.

[23] A. Herold. Narrowing techniques applied to idempotent unification. Technical Report SR-86-16, SEKI, 1986.

[24] S. Hölldobler. *Foundations of Equational Logic Programming*. Springer LNCS 353, 1989.

[25] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell. *ACM SIGPLAN Notices*, 27(5):1–64, 1992.

[26] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *JCSS*, 25:239–266, 1982.

[27] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1–112. Oxford University Press, 1992. Previous version: Term rewriting systems, Technical Report CS-R9073, Stichting Mathematisch Centrum, Amsterdam, 1990.

[28] S. Krischer and A. Bockmayr. Detecting redundant narrowing derivations by the LSE-SL reducibility test. In *Proc. RTA '91*. Springer LNCS 488, 1991.

[29] A. Middeldorp and E. Hamoen. Counterexamples to completeness results for basic narrowing (extended abstract). In *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 244–258, Volterra, Italy, September 1992.

[30] J. J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In *Proc. Second International Conference on Algebraic and Logic Programming*, pages 298–317. Springer LNCS 463, 1990.

[31] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[32] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7:295–317, 1989.

[33] S. L. Peyton Jones. *The implementation of Functional Programming Languages*. Prentice-Hall International, Englewood Cliffs, N.J., 1987.

[34] U. S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.

[35] J.-H. You. Unification modulo an equality theory for equational logic programming. *The Journal of Computer and System Sciences*, 42(1):54–75, 1991.