Lazy Narrowing and Needed Narrowing: A Comparison

María Alpuente[1], Moreno Falaschi[2], Pascual Julián[3], and Germán Vidal[1]

[1] DSIC, UPV, {alpuente,gvidal}@dsic.upv.es; [2] DIMI, U. Udine, falaschi@dimi.uniud.it; [3] DI, UCLM, Pascual.Julian@uclm.es

1 Introduction

Functional logic programming [8] allows us to integrate some of the best features of the classical declarative paradigms, namely functional and logic programming. The operational semantics of functional logic languages is usually based on narrowing, an evaluation mechanism which combines the reduction principle of functional languages and the resolution principle of logic languages. Lazy evaluation is a valuable feature of functional (logic) programming languages since it avoids unnecessary computations and allows us to deal with infinite data structures. Recently, Antoy et al. [4] introduced a lazy evaluation strategy for functional logic programs, called *needed narrowing*, which generalizes Huet and Lévy's [11] call by need reduction to deal with logical variables and unification in *inductively sequential* rewrite systems, where functions are defined in a way which is similar to case expressions.

This paper investigates and clarifies the formal relation between needed narrowing and the (not so lazy) *demand-driven* narrowing strategy of [18]. We demonstrate the following results:

1. Needed narrowing and (demand-driven) lazy narrowing are computationally equivalent on a subclass of inductively sequential programs: the *uniform programs* of [12, 13], i.e., both strategies compute the same *answers* and *values* over this class of programs.
2. This is the broadest class of programs where such an equivalence has been proven. However, (demand-driven) lazy narrowing may still perform some redundant computations which produce several copies of the same value and answer.
3. Therefore, we introduce a complete refinement of (demand-driven) lazy narrowing: *uniform lazy narrowing*, which is equivalent to needed narrowing on uniform programs and does not perform redundant computations. This implies that uniform lazy narrowing enjoys the optimal properties of needed narrowing.

This paper is organized as follows: After recalling in Section 2 some basic notions, in Section 3 we characterize the class of uniform programs. Section 4 investigates the precise relation between the needed narrowing and demand-driven lazy narrowing strategies. In Section 5, we present an optimization of the demand–driven narrowing strategy and we prove the correctness of the proposed refinement. Section 6 concludes.

2 Preliminaries

We assume familiarity with basic notions of term rewriting [6] and functional logic programming [8]. Throughout this paper, $\mathcal{X}$ denotes a countably infinite set of *variables* and $\mathcal{F}$ denotes a set of *function symbols* (also called the *signature*), each of which has a fixed associated arity. We often write $f/n \in \mathcal{F}$ to denote that f is a function symbol of arity n. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* built from $\mathcal{F}$ and $\mathcal{X}$. $\mathcal{T}(\mathcal{F})$ denotes the set of *ground terms*. If $t \notin \mathcal{X}$, then $\mathcal{R}oot(t)$ is the function symbol heading the term t, also called the *root symbol* of t. We write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$. A *linear* term does not contain multiple occurrences of the same variable. $\mathcal{V}ar(o)$ is the set of variables occurring in the syntactic object o.

A *substitution* σ is a mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that its *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. We denote the identity substitution by id. We use the overloaded symbol "$\leq$" to denote the usual preorder on substitutions, i.e., $\sigma \leq \theta$ iff $\exists \gamma.\ \gamma \circ \sigma = \theta$. The *restriction* $\sigma_{\restriction V}$ of a substitution σ to a set V of variables is defined by $\sigma_{\restriction V}(x) = \sigma(x)$ if $x \in V$ and $\sigma_{\restriction V}(x) = x$ if $x \notin V$. A *unifier* of two terms s and t is a substitution σ such that $\sigma(s) = \sigma(t)$. A unifier σ is called the *most general unifier (mgu)* if $\sigma \leq \theta$ for all other unifier θ. We say that term s is *more general* than term t (in symbols, $s \leq t$) iff $\exists \sigma$ such that $\sigma(s) = t$.

Positions of a term t are represented by sequences of positive numbers. We let Λ denote the empty sequence, and $p.w$ denote the concatenation of sequences p and w. They are ordered by the *prefix* ordering "$\leq$": $p \leq q$ iff $\exists w.\ p.w = q$. $\mathcal{P}os(t)$ and $\mathcal{FP}os(t)$ denote, respectively, the set of positions and the set of nonvariable positions of the term t. The subterm of t at position p is denoted by $t|_p$ and the result of replacing $t|_p$ with a term s is denoted by $t[s]_p$.

2.1 Programs

In this section, we introduce a functional logic language which can be thought of as a "common core" for some popular lazy functional logic languages such as Babel [18], Curry [10], and Toy [15].

A *rewrite rule* is an ordered pair $l \to r$ such that $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Terms l and r are called the *left-hand side* (lhs) and the *right-hand side* (rhs) of the rule, respectively. A *term rewriting system* (TRS), is a pair $\langle \mathcal{F}, \mathcal{R} \rangle$ where $\mathcal{F}$ is a signature and $\mathcal{R}$ is a set of rewrite rules. Given a TRS $\langle \mathcal{F}, \mathcal{R} \rangle$, we assume that the signature $\mathcal{F}$ is partitioned into two disjoint sets $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ where $\mathcal{D} = \{\mathcal{R}oot(l) \mid (l \to r) \in \mathcal{R}\}$ and $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions* or *operations*. $\mathcal{T}(\mathcal{C}, \mathcal{X})$ denotes the set of *constructor terms*. A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{D}$ and $\overline{d_n}$ are constructor terms. For simplicity, we often identify a TRS with the set of rewrite rules $\mathcal{R}$. We say that a TRS is *constructor–based* (CB) if the lhs's of $\mathcal{R}$ are patterns. We say that a TRS is *orthogonal* if it is *left–linear* and *non–ambiguous* [6]. In this work, TRSs are called *programs*.

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ (or just $t \to s$) if there exists a position $p \in \mathcal{FP}os(t)$, a rewrite rule $R = (l \to r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. We denote by $\to^+$ the transitive closure of $\to$, and by $\to^*$ the transitive and reflexive closure of $\to$. We say that a term t is a *head normal form* (hnf) if t is a variable or $\mathcal{R}oot(t) \in \mathcal{C}$.

The equality predicate $\approx$ is defined, as in functional languages, as the *strict equality* on terms (note that we do not require terminating rewrite systems and thus reflexivity is not desired), i.e., the equation $t_1 \approx t_2$ is satisfied if t_1 and t_2 are reducible to the same ground constructor term (i.e., $\exists d \in \mathcal{T}(\mathcal{C})$ such that $t_1 \to^* d \; {}^*\!\!\leftarrow t_2$). Equations can also be interpreted as terms by defining the symbol $\approx$ as a binary operation symbol.

2.2 Lazy Narrowing Strategies

The operational principle of functional logic languages with a complete semantics is called narrowing [8]. Narrowing was originally proposed as a method to *solve* equations. A *solution* of an equation $s \approx t$ is a substitution σ such that $\sigma(s \approx t)$ can be reduced to *true*.

The narrowing relation is nondeterministic and it can produce a huge search space. Hence, several narrowing strategies which are able to remove some useless derivations have been proposed.

Definition 1 (Narrowing Strategy).
A narrowing strategy is a mapping φ which, given a term t, computes a set of triples $\langle p, R, \sigma\rangle$, where $p \in \mathcal{FP}os(t)$, $R = (l \to r)$ is a (standardized apart) program rule and σ is a unifier of $t|_p$ and l.[1]

We say that $t \overset{[p,R,\sigma]}{\leadsto_\varphi} t'$ is a *narrowing step using the strategy* φ if $\langle p, R, \sigma\rangle \in \varphi(t)$ and $\sigma(t) \to_{p,R} t'$. Analogously, we say that a derivation $t_0 \overset{[p_1,R_1,\sigma_1]}{\leadsto_\varphi} t_1 \overset{[p_2,R_2,\sigma_2]}{\leadsto_\varphi} \ldots \overset{[p_n,R_n,\sigma_n]}{\leadsto_\varphi} t_n$ is a φ-derivation (denoted by $t_0 \overset{\sigma}{\leadsto}_\varphi^* t_n$, where $\sigma = \sigma_n \circ \sigma_{n-1} \ldots \circ \sigma_1$), if each narrowing step in the derivation uses φ. We are mainly interested in those derivations leading to a value (a constructor term). A narrowing derivation $t \overset{\sigma}{\leadsto}_\varphi^* s$ is *successful* iff $s \in \mathcal{T}(\mathcal{C} \cup \mathcal{X})$, where s is the computed *value* and σ is the computed *answer*. The pair "value/answer" computed in a narrowing derivation is often called *output*.

An important property of a narrowing strategy φ is *completeness*: for each solution to a given (equational) goal, a more general answer is still found by narrowing using φ.

Lazy Narrowing. In the following, we specify our (demand driven) *lazy narrowing* (LN) strategy in the style of [18]. Lazy narrowing reduces expressions at

[1] In most narrowing strategies, σ is required to be a *most general unifier* of the two terms l and $t|_p$. This is not the case of needed narrowing, which may compute substitutions which are not most general.

outermost narrowable positions. Narrowing at inner positions is performed only if it is *demanded* by the lhs of some rule. Our formalization of LN first appeared in [1], where it was defined for conditional (CB and weakly orthogonal) TRSs. The formalization of LN uses a particular kind of unification algorithm [18] which takes advantage of the pattern discipline and the left-linearity of program rules. In particular, we use the function LU (Linear Unification) to compute the unification between a linear pattern and an arbitrary term. In contrast to standard unification algorithms, LU considers the case when there is a mismatch between a defined function symbol f and a constructor symbol c as a demand of further evaluation of f. Therefore, LU can either succeed (with outcome `Success`), fail (`Fail`) or suspend (`Demand`); when it suspends, it returns the set P of positions which "demand" further evaluation of their arguments. A formal definition of LU can be found in [1].

Now we define the LN strategy. We assume that the rules of $\mathcal{R}$ are numbered as $R_1, \ldots, R_m$.

Definition 2 (Lazy Narrowing Strategy).
We define the lazy narrowing strategy as a set-valued function λ_{lazy} which, given a term t, computes the set of triples $\langle p, R_k, \sigma\rangle$, where $p \in \mathcal{FP}os(t)$ is a position of term t (called "lazy" position), $R_k = (l_k \to r_k)$ is a (renamed apart) rule of $\mathcal{R}$, and σ is a substitution, as follows:

$$\lambda_{lazy}(t) = \bigcup_{k=1}^{m} \lambda'(t, \Lambda, R_k)$$

where $$\lambda'(t,p,R_k) = \begin{array}{l} \texttt{if } \mathcal{R}oot(l_k) = \mathcal{R}oot(t|_p) \texttt{ then} \\ \quad \texttt{case } \mathsf{LU}(l_k, t_{|p}) \texttt{ of} \\ \qquad \left\{ \begin{array}{ll} (\texttt{Success}, \sigma): & \{\langle p, R_k, \sigma\rangle\} \\ (\texttt{Fail}, \emptyset): & \emptyset \\ (\texttt{Demand}, P): & \bigcup_{q \in P} \bigcup_{k=1}^{m} \lambda'(t, p.q, R_k) \end{array} \right. \\ \texttt{else } \emptyset \end{array}$$

For CB orthogonal programs, LN is (strongly) complete w.r.t. strict equations and constructor substitutions as solutions [16, 18]. Here, *strong completeness* means that, given a conjunction $t_1 \wedge t_2 \wedge \ldots \wedge t_n$, we can nondeterministically select for evaluation just one of the conjuncts, t_i, without losing completeness.[2] It is well-known that LN does not define a *pure* lazy evaluation strategy in the sense that it might demand the evaluation of expressions which are not really *needed* to compute the outcome [4].

Needed Narrowing. Here, we recall the main notions concerning the needed narrowing strategy of [4]. *Needed Narrowing* (NN) reduces the *outermost needed positions* of the input term which are unavoidable to compute the final result. NN is defined on *inductively sequential programs*, a (strict) subset of CB orthogonal programs. The definition of this class of programs, as well as the NN strategy, make use of the notion of a *definitional tree*, first introduced in [2]. From a

[2] Note that, in our context, each t_i is a Boolean expression, in particular an equation.

declarative point of view, a *partial definitional tree* $\mathcal{P}$ can be seen as a set of linear patterns partially ordered by the strict subsumption order "$<$" [3]. $pattern(\mathcal{P})$ denotes the minimum element of $\mathcal{P}$. Given a defined function f/n, we call $\mathcal{P}$ a *definitional tree of* f if $\mathcal{P}$ is a partial definitional tree, $pattern(\mathcal{P}) = f(\overline{x_n})$, where $\overline{x_n}$ are distinct variables and the leaves of $\mathcal{P}$ are all and only variants of the left-hand sides of the rules defining f. Note that there can be more than one definitional tree for a defined function.

Definition 3 (Inductively Sequential Program).
A defined function f is called inductively sequential if it has a definitional tree. A rewrite system $\mathcal{R}$ is called inductively sequential if all its defined functions are inductively sequential.

Needed narrowing can be formulated as follows[3].

Definition 4 (Needed Narrowing Strategy).
Let $\mathcal{R}$ be an inductively sequential program. Let t be an operation-rooted term, and $\mathcal{P}$ a partial definitional tree with $pattern(\mathcal{P}) = \pi$ such that $\pi \leq t$. We define an application λ from terms and partial definitional trees to sets of triples (position, rule, substitution) as the least set satisfying the following properties. We consider two cases for $\mathcal{P}$:

1. *If π is a leaf, i.e., $\mathcal{P} = \{\pi\}$, and $(\pi \to r) \in \mathcal{R}$, $\lambda(t, \mathcal{P}) = \{\langle \Lambda, \pi \to r, id\rangle\}$.*
2. *If π is a branch, consider the inductive position o of π and a child $\pi_i = \pi[c_i(\overline{x_n})]_o \in \mathcal{P}$. Let $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$ be the partial definitional tree where all patterns are instances of π_i. We consider the following cases for the subterm $t|_o$:*

$$\lambda(t,\mathcal{P}) \ni \begin{cases} \langle p, R, \sigma \circ \tau\rangle & \texttt{if } t|_o = x \in \mathcal{X},\ \tau = \{x/c_i(\overline{x_n})\}, \texttt{ and } (p,R,\sigma) \in \\ & \lambda(\tau(t), \mathcal{P}_i); \\ \langle p, R, \sigma \circ id\rangle & \texttt{if } t|_o = c_i(\overline{t_n}) \texttt{ and } (p,R,\sigma) \in \lambda(t,\mathcal{P}_i); \\ \langle o.p, R, \sigma \circ id\rangle & \texttt{if } t|_o = f(\overline{t_n}),\ f \in \mathcal{F} \texttt{ and } (p,R,\sigma) \in \lambda(t|_o,\mathcal{P}') \texttt{ where} \\ & \mathcal{P}' \textit{ is a definitional tree for } f. \end{cases}$$

Note that each NN step can be represented as $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_1\rangle$, which is called the *canonical representation* of a NN step. We assume that the definitional trees always contain fresh variables when they are used in a narrowing step. By abuse, we call "needed narrowing position" to the position p of a needed redex.

This mechanism differs from the *needed reduction* of lazy functional languages only in the instantiation of free variables by means of unifiers. The unifiers computed in NN steps may contain extra bindings which would eventually be performed later in (demand-driven) LN narrowing derivations. We refer to them as *anticipated substitutions*. Roughly speaking, an anticipated substitution is the part of a substitution computed in a NN step which is not computed in a "corresponding" LN step (using the same rule at the same position).

[3] This definition slightly differs from the one appeared in [4], although it computes the same needed narrowing steps if we consider substitutions modulo variable renaming and restricted to $\mathcal{V}ar(t)$. In the sequel we always assume this restriction when we compare needed narrowing and lazy narrowing.

Definition 5 (Anticipated Substitution).
Let $\mathcal{R}$ be an inductively sequential program and R a rule of $\mathcal{R}$. Let t be a term rooted by $f \in \mathcal{F}$, and $\mathcal{P}$ be a definitional tree of f. Let $\langle p, R, \theta \rangle \in \lambda(t, \mathcal{P})$ and $\langle p, R, \sigma \rangle \in \lambda_{lazy}(t)$ (if they exist). We say that τ is the anticipated substitution part of θ computed by λ if $\theta = \sigma \circ \tau$.

It is noteworthy that the concept of "anticipated substitution", as informally defined in [4], does not correspond with the more declarative notion above introduced. Technically, the "anticipated substitutions" of [4] are made of those bindings computed by traversing definitional trees which do not correspond to the function which is actually reduced. This notion of "anticipation" has an operational flavor and as we are going to show in Section 4.1, for some cases, it may compute additional bindings that can not be considered as "anticipated" when we use Definition 5. Therefore, in order to distinguish between both concepts, we prefer to name the "anticipated substitutions" of [4] as *anticipated bindings.* A formal definition of this concept is given in Definition 7, which revels to be a valuable tool to prove several results presented in this paper.

Antoy et al. [4] proved that, for inductively sequential programs, NN is complete w.r.t. strict equations and constructor substitutions as solutions. Moreover, NN is optimal w.r.t. the independence of computed solutions and the length of successful derivations (in graph–based implementations).

3 Uniform Programs

Uniform programs were introduced in [12, 13] to improve the implementation of lazy narrowing within the functional logic language Babel [18].

Definition 6 (Uniform Program).
A uniform program is a set of rewrite rules such that the following conditions hold:

1. Flat constructor patterns*: the arguments of the lhs's of the rules are either variables or linear constructor terms of the form $c(\overline{x_n})$.*
2. Orthogonality*.*
3. Uniformity*: Let $f(\overline{t_n})$ and $f(\overline{s_n})$ be the lhs's of two rules defining f; then, t_i is a variable iff s_i is a variable, for all $i \in \{1, \ldots, n\}$.*

In uniform programs, functions f/n are defined by:

(i) one or more rules whose lhs has the shape:

$f(\ldots, c_{k_1}(\overline{x_{m_{k_1}}}), \ldots, c_{k_p}(\overline{y_{m_{k_p}}}), \ldots)$

where $\{k_1, \ldots, k_p\} \subseteq \{1, \ldots, n\}$ are fixed positions (called *inductive positions* of f) addressing flat constructor terms and the remaining positions contain only variable arguments, or

(ii) one single rule whose shape is $f(\overline{x_n}) \to r$.

In the former case, for every pair of (distinct) program rules, there exist at least an inductive position of the lhs's of these rules where the corresponding arguments are rooted by two different constructors.

Example 1. The following program[4] is uniform

$$f(X,1,2,3) \to 1, \quad f(X,2,2,3) \to 2, \quad f(X,2,2,2) \to 3.$$

In [19], *simple uniform programs* (i.e., uniform programs whose rules have—at most—one inductive position) were introduced and, as an easy consequence of Lemma 2 in [19], the equivalence between NN and LN over this class of programs holds for derivations leading to a hnf.

Implementation Problems and Properties Moreno et al. showed in [17] that, inside a sequential implementation of narrowing, using a deep-first search with backtracking, it is difficult to combine the lazy evaluation strategy with the use of logical variables. Uniform programs were introduced to avoid this drawback. Unfortunately, even if uniform programs have many advantages [12], they do not always suffice to get rid of useless computations (see Example 4).

Proposition 1. *Uniform TRSs are inductively sequential.*

The structure of the proof is very simple: Given a uniform program $\mathcal{R}$, it suffices to consider that, for each defined function f of $\mathcal{R}$, we can construct an associated definitional tree by using as inductive positions to build the tree the inductive positions of the function symbol (i.e., the nonvariable positions of the lhs's of the rules defining the function f).

We have that simple uniform programs $\subseteq$ uniform programs $\subseteq$ inductively sequential programs $\subseteq$ CB orthogonal programs.

It is worthwhile to note that, for uniform programs, it is possible to build the associated definitional tree by considering the set of inductive positions of the function in any possible order. This does not generally hold for (non-uniform) inductively sequential programs. In the following, we call *standard* definitional tree of a function f the tree in which the inductive positions of f are considered from left to right.

4 Equivalence of Lazy Evaluation Strategies

In this section, we establish the precise relation between the needed narrowing strategy of [4] and the (demand-driven) lazy narrowing strategy of [1], first for single steps and then for successful derivations.

4.1 Stepwise Equivalence

To motivate our discussion, let us begin with an example which shows that, even for uniform programs, there is not a direct, one–to–one correspondence between LN and NN steps.

[4] We use capital letters to denote variables in examples.

Example 2. Consider the uniform program:

$$R_1: \ f(a,b) \to c, \qquad R_2: \ g(c) \to b.$$

where a, b and c are constructor symbols. Using standard definitional trees for f and g, there exists the following NN derivation:

$$f(X, g(Y)) \overset{[2,R_2,\{X/a,Y/c\}]}{\leadsto_{NN}} f(a,b) \overset{[\Lambda,R_1,id]}{\leadsto_{NN}} c$$

whereas the corresponding derivation using LN is:

$$f(X, g(Y)) \overset{[2,R_2,\{Y/c\}]}{\leadsto_{LN}} f(x,b) \overset{[\Lambda,R_1,\{X/a\}]}{\leadsto_{LN}} c$$

Note that, in the NN derivation, the binding X/a is computed in the first step, whereas it is computed in the second step in the corresponding LN derivation.

We formalize the precise correspondence between LN and NN steps in Proposition 2. First, we need to prove some preparatory results.

The following lemma establishes that NN does not compute anticipated substitutions when the NN step is performed at the root position of the term.

Lemma 1. *Let $\mathcal{R}$ be a uniform program and $R \in \mathcal{R}$ a program rule. Let t be an operation-term and $\mathcal{P}$ be a definitional tree for the root of t. Then, $\langle \Lambda, R, \sigma \rangle \in \lambda(t, \mathcal{P})$ iff $\langle \Lambda, R, \sigma \rangle \in \lambda_{lazy}(t)$.*

As we mentioned before, the ability to anticipate some bindings in the NN strategy is the key to avoid some unnecessary computations: if the evaluation of a subterm $t|_p$ is demanded by some program rule, after the evaluation of this subterm (to a hnf), only the rules which actually demanded the evaluation of $t|_p$ can be applied. Indeed, thanks to the anticipated substitutions, only *strongly sequential redexes* are reduced in needed narrowing derivations [4]. The computation of anticipated substitutions makes the difference between the LN and NN strategies. However, in order to state and prove the precise relation between LN and NN computation steps for arbitrary input terms, we need a more operational and technical notion of "anticipation", which we formulate by mimicking the definition of a NN step, called *anticipated bindings.* Both notions do coincide for linear terms but they may differ for non-linear terms.

Definition 7 (Anticipated Bindings).
Let $\mathcal{R}$ be a program, t be an operation-rooted term, and $\mathcal{P}$ be a partial definitional tree such that $pattern(\mathcal{P}) = \pi$ and $\pi \leq t$. We define the set of anticipated bindings $\alpha(t, \mathcal{P})$ in $\lambda(t, \mathcal{P})$ as a mapping from terms and partial definitional trees to sets of substitutions as follows:

1. `If` π *is a leaf,* $\alpha(t, \mathcal{P}) = \{id\}$.
2. `If` π *is a branch,*

$$\alpha(t,\mathcal{P}) \ni \begin{cases} \tau' \circ \tau & \texttt{if } t|_o = x \in \mathcal{X},\ \tau = \{x/c_i(\overline{x_n})\},\ \langle \Lambda, R, \sigma\rangle \notin \lambda(\tau(t), \mathcal{P}_i)\texttt{, for} \\ & \texttt{some } \textit{rule } R \texttt{ and } \textit{substitution } \sigma\texttt{, and } \tau' \in \alpha(\tau(t), \mathcal{P}_i)\texttt{;} \\ \tau' \circ id & \texttt{if } t|_o = c_i(\overline{t_n}),\ \langle \Lambda, R, \sigma\rangle \notin \lambda(t, \mathcal{P}_i)\texttt{, for some } \textit{rule } R \texttt{ and} \\ & \textit{substitution } \sigma\texttt{, and } \tau' \in \alpha(t, \mathcal{P}_i)\texttt{;} \\ \tau' \circ id & \texttt{if } \langle \Lambda, R, \sigma\rangle \notin \lambda(t, \mathcal{P}),\ t|_o = g(\overline{t_n}),\ g \in \mathcal{F},\ \langle \Lambda, R, \sigma\rangle \notin \\ & \lambda(t|_o, \mathcal{P}')\texttt{, for some } \textit{rule } R \texttt{ and } \textit{substitution } \sigma\texttt{, and } \tau' \in \\ & \alpha(t|_o, \mathcal{P}') \texttt{ with } \mathcal{P}' \textit{ a definitional tree of } g. \\ id & \texttt{otherwise} \end{cases}$$

where o is the inductive position of π, $\pi_i = \pi[c_i(x_1, \ldots, x_n)]_o \in \mathcal{P}$ is a child of π, and $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$ is a proper subtree of $\mathcal{P}$.

The following examples illustrate the difference between this definition and the notion of anticipated substitution.

Example 3. Consider the uniform program:

$$R_1 : \ f(a,b) \to c \qquad R_2 : \ f(a,c) \to b \qquad R_3 : \ g(a) \to c \qquad R_4 : \ g(c) \to b$$

and the standard definitional trees $\mathcal{P}$ and $\mathcal{P}'$ for f and g, respectively.

- Consider the term $g(Z)$. Then, $pattern(\mathcal{P}')$ is a branch and the fourth case of Definition 7(2) holds, since

  $$\lambda(g(a), \mathcal{P}'_1) = \{\langle \Lambda, R_3, id\rangle\} \quad \text{and} \quad \lambda(g(c), \mathcal{P}'_2) = \{\langle \Lambda, R_4, id\rangle\}.$$

  Therefore, $\alpha(g(Z), \mathcal{P}') = \{id\}$, which does coincide with the anticipated substitution.

- Given the term $f(X, f(Y, g(Z)))$, we have that

  $$\langle 2.2, R_4, id \circ \{Z/c\} \circ id \circ \{Y/a\} \circ id \circ \{X/a\}\rangle \in \lambda(f(X, f(Y, g(Z))), \mathcal{P})$$

  and $\langle \Lambda, R_4, id \circ \{Z/c\}\rangle \in \lambda(f(a, f(a, g(Z)))|_{2.2}, \mathcal{P}')$. Hence, Definition 7 computes the anticipated bindings: $id \circ \{Y/a\} \circ id \circ \{X/a\} = \{Y/a, X/a\} \in \alpha(f(X, f(Y, g(Z))), \mathcal{P})$, which also agree with the anticipated substitution. Roughly speaking, for the above term, the substitution computed by α consists of the bindings computed by λ before the demanded position 2.2 is considered and, hence, before the "non–anticipated" bindings $id \circ \{Z/c\}$ are computed by the call to $\lambda(f(a, f(a, g(Z)))|_{2.2}, \mathcal{P}')$. Note that, the computation of the mapping α stops returning the binding id and disregarding the "non–anticipated" part, when the subterm $g(Z)$ of $f(a, f(a, g(Z)))$ at position 2.2 is reached.

- Finally, consider the non-linear term $f(Z, f(Y, g(Z)))$. Then,

  $$id \circ id \circ \{Y/a\} \circ id \circ \{Z/a\} = \{Y/a, Z/a\} \in \alpha(f(Z, f(Y, g(Z))), \mathcal{P})$$

  Moreover, it holds that

  $$\langle 2.2, R_3, id \circ id \circ id \circ \{Y/a\} \circ id \circ \{Z/a\}\rangle \in \lambda(f(Z, f(Y, g(Z))), \mathcal{P})$$

  but $\langle 2.2, R_3, \{Z/a\}\rangle \in \lambda_{lazy}(f(Z, f(Y, g(Z))))$. This means that the mapping α computes an anticipated binding Z/a which does not belong to the corresponding "anticipated substitution" associated to this step.

The following proposition establishes the precise relation between LN and NN steps: given a NN step from t which computes the substitution $\sigma \circ \tau$, where τ are the bindings anticipated in this step, there exists a corresponding LN step from $\tau(t)$ which computes the substitution σ by reducing the same position using the same program rule (and vice versa).

Proposition 2. *Let $\mathcal{R}$ be a uniform program and $R \in \mathcal{R}$ a program rule. Let t be an operation-rooted term. Then, $\langle p, R, \sigma \rangle \in \lambda_{lazy}(\tau(t))$ iff there exists some definitional tree $\mathcal{P}$ for $\mathcal{R}oot(t)$ such that $\langle p, R, \sigma \circ \tau \rangle \in \lambda(t, \mathcal{P})$, where $\tau \in \alpha(t, \mathcal{P})$ are the anticipated bindings in $\lambda(t, \mathcal{P})$.*

As a corollary, we obtain the following result, which holds for each single narrowing step. A NN step $t \overset{[p,R,\sigma\circ\tau]}{\leadsto}_{NN} s$ can be proved iff there exists a corresponding LN step $\tau(t) \overset{[p,R,\sigma]}{\leadsto}_{LN} s$, where $\tau \in \alpha(t, \mathcal{P})$ are the bindings anticipated in the computation $\lambda(t, \mathcal{P})$.

Note that Proposition 2 holds for an arbitrarily fixed definitional tree; this amounts to say that, for each LN step, there is an appropriate definitional tree such that the corresponding NN step can be proven.

To finish this subsection, let us mention a stronger equivalence, step by step, between NN and LN steps over simple uniform programs. This is an easy consequence of Proposition 2, and establishes that, for simple uniform programs, NN does not produce anticipated bindings.

Corollary 1. *Let $\mathcal{R}$ be a simple uniform program. Let t be an operation-rooted term and $\mathcal{P}$ be the definitional tree for $\mathcal{R}oot(t)$. Then, $\lambda_{lazy}(t) = \lambda(t, \mathcal{P})$.*

It is interesting to note that Corollary 1 is stronger than Lemma 2 in [19], which only entails the equivalence for derivations leading to a hnf while our results demonstrate the equivalence, step by step, between the NN and LN strategies over simple uniform programs (and thus the equivalence of derivations leading to arbitrary terms). Moreover, our result guarantees that these derivations use the same rules over the same positions at each computation step.

5 Uniform Lazy Narrowing

Proposition 2 shows that every lazy position of a term (w.r.t. a uniform program) is computed by λ, hence it is needed. Since all needed positions must be exploited in order to compute the final outcome and, according to the completeness results for needed narrowing, the order in which needed positions are considered is not relevant, it turns out that, for uniform programs, an optimization of LN is possible by a don't care selection among the demanded positions of the input term. This is similar to the *uniform narrowing* strategies introduced in [7] for canonical TRSs, which narrow only one position of the considered term while still preserving completeness. This kind of strategies has inspired us to use the name *Uniform Lazy Narrowing* (ULN) for the optimization of the LN strategy which we formalize as follows. Again, we assume that the rules of $\mathcal{R}$ are numbered as $R_1, \ldots, R_m$.

Definition 8 (Uniform LN Strategy).
Let $\mathcal{R}$ be a uniform program. We define the uniform lazy narrowing strategy as a set-valued function:

$\lambda_{ulazy}(t) = \bigcup_{k=1}^{m} \lambda'_u(t, \Lambda, R_k)$

where $\lambda'_u(t,p,R_k) =$ `if` $\mathcal{R}oot(l_k) = \mathcal{R}oot(t|_p)$ `then`

$$
\begin{array}{l}
\texttt{case LU}(\langle l_k, t|_p\rangle)\texttt{of} \\
\left\{\begin{array}{ll}
(\texttt{Success}, \sigma): & \{\langle p, R_k, \sigma\rangle\} \\
(\texttt{Fail}, \emptyset): & \emptyset \\
(\texttt{Demand}, P): & \bigcup_{k=1}^{m} \lambda'_u(t, p.q, R_k) \\
 & \texttt{with } q = \mathit{select\_don't\_care}(P)
\end{array}\right. \\
\texttt{else } \emptyset
\end{array}
$$

where the function select_don't_care(S) arbitrarily chooses one element of the set S.

Note that the ULN strategy only selects one redex position p in a term (although several triples (p, R, σ) can be associated to that position p). Informally, the above strategy allows us to overcome the loss of efficiency of (unoptimized) LN strategies (w.r.t. NN) which is due to the "don't-know" nondeterministic choice of all demanded positions in a LN reduction, in contrast to the more effective "don't-care" choice of the particular definitional tree for f in NN steps. In ULN computations, note that one can fix any selection strategy, e.g., a left-to-right (Prolog like) selection of demanded positions, just in the same way needed narrowing fixes one of the possible definitional trees for computing. The following example illustrates how our ULN strategy takes advantage of this property to disregard some useless "redundant" LN derivations.

Example 4. Given the program

$R_1:\ f(a,a) \to 0, \qquad R_2:\ g(a) \to a.$

and the term $t = f(g(X), g(Y))$, if we use the standard definitional tree for f, the NN strategy computes only one (successful) derivation:

$$
\begin{array}{ll}
f(g(X), g(Y)) \overset{[1,R_2,\{X/a\}]}{\leadsto_{NN}} & f(a, g(Y)) \\
\overset{[2,R_2,\{Y/a\}]}{\leadsto_{NN}} & f(a,a) \\
\overset{[\Lambda,R_1,id]}{\leadsto_{NN}} & 0.
\end{array}
$$

However, the LN strategy produces a search tree with two branches where the leftmost branch "matches" the former NN derivation and the rightmost branch can be considered a "redundant" LN derivation. On the other hand, if we fix a different definitional tree for f, which first considers the inductive position 2, then the NN strategy only computes the rightmost LN derivation:

$$
\begin{array}{ll}
f(g(X), g(Y)) \overset{[2,R_2,\{Y/a\}]}{\leadsto_{NN}} & f(g(X), a) \\
\overset{[1,R_2,\{X/a\}]}{\leadsto_{NN}} & f(a,a) \\
\overset{[\Lambda,R_1,id]}{\leadsto_{NN}} & 0.
\end{array}
$$

Now the leftmost branch represents the "redundant" LN derivation.

5.1 Equivalence of Derivations

Now, we are ready to state and prove the relation between general LN and NN derivations over uniform programs. Although we restrict the discussion to the ULN strategy, most of our results can be easily extended to rough LN.

The following example reveals that, even for uniform programs, NN and ULN do not compute the same values and answers for arbitrarily fixed definitional trees, unless we require the evaluation to reach a head normal form.

Example 5. Given the uniform program

$$R_1 : \ f(a,b) \to b, \quad R_2 : \ g(c(Y)) \to Y, \quad R_3 : \ h(b) \to b.$$

and the term $f(g(X), h(Z))$, there exists only one ULN derivation for this term (selecting the subset of triples associated to the leftmost lazy position), which outcomes the result $\langle f(Y,b), \{Z/b, X/c(Y)\}\rangle$:

$$f(g(X), h(Z)) \overset{[1,R_2,\{X/c(Y)\}]}{\leadsto_{ULN}} f(Y, h(Z)) \overset{[2,R_3,\{Z/b\}]}{\leadsto_{ULN}} f(Y,b)$$

However, by considering standard definitional trees for f, g and h, we have the NN derivation:

$$f(g(X), h(Z)) \overset{[1,R_2,\{X/c(Y_2)\}]}{\leadsto_{NN}} f(Y_2, h(Z)) \overset{[2,R_3,\{Z/b\}\circ\{Y_2/a\}]}{\leadsto_{NN}} f(a,b)$$

which computes the term $f(a,b)$ and substitution $\{Z/b, X/c(a)\}$. Now, if a subsequent narrowing step (reaching a hnf) is performed, both derivations end with the same values and answers (up to renaming). It is important to note that there exists an additional, redundant LN derivation, whose first step uses rule R_3 and whose second step uses rule R_2. This derivation is cut when we choose the *select_don't_care* function to be a left-to-right selection function.

The following theorem establishes that each NN derivation to a head normal form can be mimicked by ULN and vice versa.

Theorem 1. *Let $\mathcal{R}$ be a uniform program and t an operation-rooted term. Then,*

1. *there exists a NN derivation $\mathcal{D} = (t \overset{\sigma}{\leadsto}_{NN}{}^* \, s)$, where s is a hnf, iff there exists a corresponding ULN derivation $\mathcal{D}' = (t \overset{\sigma}{\leadsto}_{ULN}{}^* \, s)$;*
2. *$\mathcal{D}$ and $\mathcal{D}'$ have the same length, and they use the same rules over the same positions on the corresponding steps.*

The following result is a direct consequence of Theorem 1.

Corollary 2. *Let $\mathcal{R}$ be a uniform program and e be an equation. There exists a NN derivation $e \overset{\sigma}{\leadsto}_{NN}{}^* \, true$ iff there exists the ULN derivation $e \overset{\sigma}{\leadsto}_{ULN}{}^* \, true$.*

5.2 Correctness of Uniform Lazy Narrowing

Corollary 2 together with the soundness and completeness of NN entail the correctness of ULN w.r.t. the strict equality and constructor substitutions as solutions in uniform programs.

Theorem 2. *Let $\mathcal{R}$ be a uniform program and e be an equation.*

1. *(Soundness) if $e \leadsto^{\sigma}_{ULN}{}^* true$ is a ULN derivation, then σ is a solution for e.*
2. *(Completeness) For each constructor substitution σ which is a solution of e, there exists a ULN derivation $e \leadsto^{\sigma'}_{ULN}{}^* true$ with $\sigma' \leq \sigma$ $[\mathcal{V}ar(e)]$.*

It is important to note that Theorem 2 demonstrates that ULN is strongly complete (i.e., all lazy positions which may occur inside a term but one can be disregarded without jeopardizing completeness).

6 Conclusions

This paper investigates the precise relation between the needed narrowing strategy of [4] and the (demand-driven) lazy narrowing strategy of [1]. The original contributions are:

- Uniform programs are a subclass of the inductively sequential programs (Proposition 1).
- For uniform programs, a precise relation between needed narrowing and lazy narrowing derivations (and steps) has been established in terms of "anticipated bindings" (Proposition 2).
- For uniform programs, an optimized lazy evaluation strategy, called uniform lazy narrowing, has been formulated, which is still complete and strongly equivalent to needed narrowing on the class of uniform programs (Theorem 1).

These results offer the first proof that (uniform) lazy narrowing enjoys the optimality properties of needed narrowing over the class of uniform programs.

Recent implementations of functional logic languages [5, 15] which are based on definitional trees first transform inductively sequential programs into (a Prolog representation of simple) uniform programs, and then apply a demand-driven strategy codified in Prolog. This is the technique most commonly proposed to compute NN steps in inductively sequential systems even if the correspondence w.r.t. the original NN computation model was never formalized nor claimed [4]. Our results in this paper demonstrate that, by first using a semantics preserving transformation from inductively sequential programs into uniform programs [12, 19], and then using the uniform lazy narrowing strategy to execute the resulting uniform program, we get the same successful derivations as by running the original inductively sequential program by needed narrowing. Since there exist several implementations of functional logic languages which are based on a similar transformation model [9, 12, 14, 15, 18], the results in this paper can be seen as a formal demonstration that current NN implementations fit the intended (needed narrowing) semantics of the languages.

References

1. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of PEPM'97*, volume 32, 12 of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.
2. S. Antoy. Definitional trees. In *Proc. of ALP'92*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, Berlin, 1992.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of ALP'97*, volume 1298 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, Berlin, 1997.
4. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, July 2000.
5. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into prolog. In *Proc. of FroCoS 2000*, pages 171–185. Springer LNCS 1794, 2000.
6. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
7. R. Echahed. Uniform narrowing strategies. In *Proc. of ICALP'92*, pages 259–275. Springer LNCS 632, 1992.
8. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
9. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc.of LOPSTR'95*, pages 252–266. Springer LNCS 1048, 1995.
10. M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. of ILPS'95*, pages 95–107, 1995.
11. G. Huet and J.J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443, 1992.
12. H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. In *Proc. of ALP'90*, volume 463 of *Lecture Notes in Computer Science*, pages 298–317, 1990.
13. H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Implementing a Functional Logic Language with Disequality Constrains. In K. Apt, editor, *Proc. of JICSLP'93*, pages 207–221. The MIT Press, Cambridge, MA, 1993.
14. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93*, pages 184–200. Springer LNCS 714, 1993.
15. F.J. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
16. A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.
17. J.J. Moreno-Navarro, H. Kuchen, J. Mariño-Carballo, S. Winkler, and W. Hans. Efficient Lazy Narrowing using Demandedness Analysis. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93*, pages 167–183. Springer LNCS 714, 1993.
18. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.
19. F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In P. Van Hentenryck, editor, *Proc. of SAS'97*, pages 141–159. Springer LNCS 1302, 1997.