Preliminary version. Final version in Journal of Logic and Computation (Oxford University Press), Vol. 13, No. 2, pp. 287-312, 2003

Uniform Lazy Narrowing

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

[1] DSIC, Univ. Politécnica de Valencia. {alpuente,gvidal}@dsic.upv.es
[2] Dip. di Mat. e Informatica, Univ. di Udine. falaschi@dimi.uniud.it
[3] Departamento de Informática, Univ. de Castilla-La Mancha.
Paseo de la Universidad 4, 13071 Ciudad Real, Spain.
phone: 34.926295300 ext.: 3716; fax: 34.926295354; pjulian@inf-cr.uclm.es

Abstract. Needed narrowing is a complete and optimal operational principle for modern declarative languages which integrate the best features of lazy functional and logic programming. We investigate the formal relation between needed narrowing and another (not so lazy) narrowing strategy which is the basis for popular implementations of lazy functional logic languages. We demonstrate that needed narrowing and lazy narrowing are computationally equivalent over the class of *uniform* programs introduced in [16, 17]. We also introduce a complete refinement of lazy narrowing, called *uniform lazy narrowing*, which is still equivalent to needed narrowing over the aforementioned class. Since actual implementations of functional logic languages are based on the transformation of the original program into a uniform one—which is then executed using a lazy narrowing strategy—our results can be thought of as a formal basis for the correctness of these implementations.

1 Introduction

Functional logic programming [11] allows us to integrate some of the best features of the classical declarative paradigms, namely functional and logic programming. The operational semantics of the integrated languages is usually based on *narrowing*, an evaluation mechanism which combines the reduction principle of functional languages with the instantiation of unbound variables used in logic programming. Lazy evaluation is a significant feature of functional (logic) programming languages since it avoids unnecessary computations and allows programs to deal with infinite data structures. Due to its optimality properties, *needed narrowing* [5] is currently considered the best lazy narrowing strategy. It generalizes Huet and Lévy's [15] call-by-need reduction in order to deal with logical variables and unification.

This work investigates and clarifies the formal relation between needed narrowing and another (not so lazy) narrowing strategy which was introduced in [22]—where it is called *lazy narrowing*—and is the basis for the implementation of several lazy functional logic languages like, e.g., [12, 16, 22]. Our main contributions are the following:

1. Needed narrowing and lazy narrowing are computationally equivalent over the class of *uniform programs* introduced in [16, 17], 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, lazy narrowing may still perform some redundant computations which produce several copies of the same value and answer.
3. Therefore, we formalize a complete refinement of 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 article is organized as follows. After recalling in Section 2 some basic notions about functional logic programming and (lazy) narrowing strategies, we characterize in Section 3

the class of uniform programs as a subclass of the inductively sequential programs of [3]. Section 4 investigates the precise relation between lazy narrowing and needed narrowing. Here, we prove the main theoretical result of this work: the computational equivalence of both narrowing strategies over the class of uniform programs. In Section 5, we present an optimization of the lazy narrowing strategy and prove the correctness of the proposed refinement. Section 6 discusses the usefulness of the new strategy in actual implementations and Section 7 concludes.

2 Preliminaries

We assume familiarity with basic notions of term rewriting [7] and functional logic programming [11]. Throughout this paper, $\mathcal{X}$ denotes a countably infinite set of *variables* and $\mathcal{F}$ denotes a set of *function symbols* with a fixed associated arity (also called the *signature*). 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 $t \not\in \mathcal{X}$, then $\mathcal{H}ead(t)$ denotes the outermost symbol of t. We write $\overline{o_n}$ for the *sequence of objects* $o_1,\ldots,o_n$. A term is *linear* if it does not contain multiple occurrences of the same variable. We let $\mathcal{V}ar(o)$ denote 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. The identity substitution is denoted by *id*. We use the overloaded symbol "$\leq$" for the usual preorder on substitutions, i.e., $\sigma \leq \theta$ iff there is a substitution γ such that $\gamma \circ \sigma = \theta$ (here, "$\circ$" is the *composition* of substitutions). The *restriction* $\sigma_{\upharpoonright V}$ of a substitution σ to a set V of variables is defined by $\sigma_{\upharpoonright V}(x) = \sigma(x)$ if $x \in V$ and $\sigma_{\upharpoonright V}(x) = x$ if $x \not\in V$. 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 a term s is *more general* than a term t (in symbols, $s \leq t$) iff there exists a substitution σ such that $\sigma(s) = t$.

As it is common practice, terms are represented by labeled trees. The 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 there exists a position w such that $p.w = q$. $\mathcal{P}os(t)$ and $\mathcal{FP}os(t)$ denote, respectively, the set of positions and the set of non-variable 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 [22], Curry [14], and Toy [19].

A *rewrite rule* is an ordered pair $l \to r$ such that $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $l \not\in \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Terms l and r are called the *left-hand side* and the *right-hand side* 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 $\mathcal{D} = \{\mathcal{H}ead(l) \mid (l \to r) \in \mathcal{R}\}$ and $\mathcal{C} = \mathcal{F} \backslash \mathcal{D}$. 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* if the left-hand sides of $\mathcal{R}$ are patterns. We say that a TRS is *orthogonal* if it is *left-linear* (i.e., the left-hand sides are linear terms) and *non-ambiguous* [7]. Roughly speaking, a (constructor-based) TRS is

non-ambiguous if there are no unifiable left-hand sides. In the remaining of this work, we consider constructor-based and left-linear term rewriting systems as *programs*.

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ 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 often write $t \to s$ when the position and the rule are clear from the context. We denote by $\to^+$ the transitive closure of $\to$, and by $\to^*$ the transitive and reflexive closure of $\to$. We say that a term is in *head normal form* if it is a variable or a constructor-rooted term.

The equality relation $\approx$ is defined, as in lazy functional languages, as the *strict equality* on terms; note that we do not require terminating rewrite systems and, thus, reflexivity is not desired. More precisely, 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., if there exists a ground constructor term $d \in \mathcal{T}(\mathcal{C})$ 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. Therefore, all notions for terms, such as substitution, rewriting, narrowing, etc., will also be used for equations. The semantics of $\approx$ is defined by the following rules, where $\wedge$ is assumed to be a right-associative infix symbol and c is a constructor of arity 0 in the first rule and arity $n > 0$ in the second rule [10, 22]:

$$\begin{array}{r}
c \approx c \to \mathit{true} \\
c(\overline{x_n}) \approx c(\overline{y_n}) \to x_1 \approx y_1 \wedge \ldots \wedge x_n \approx y_n \\
\mathit{true} \wedge \mathit{true} \to \mathit{true}
\end{array}$$

We assume that the above rules are added to all programs; note that the orthogonality of a rewrite system is not changed by adding these rules. The equivalence between the reducibility to the same ground constructor term and the reducibility to the constant *true* using the equality rules is established, e.g., in [5, Proposition 1].

2.2 Lazy Narrowing Strategies

The operational principle of functional logic languages with a complete semantics is called narrowing [11]. Essentially, narrowing extends the rewriting mechanism of functional languages by replacing pattern matching with unification. 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, i.e., all possible derivations have to be considered in order to guarantee completeness. Since (unrestricted) narrowing can have a huge search space, 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 program rule (with fresh variables), 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

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

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 found by narrowing using φ.

Lazy Narrowing In the following, we formalize the considered *lazy* narrowing strategy in the style of [22]. This narrowing strategy reduces expressions at outermost narrowable positions. Narrowing at inner positions is performed only if it is *demanded* by the left-hand side of some rule. Our formalization [2] was originally introduced for conditional term rewriting systems. It uses a particular kind of unification algorithm [22] which takes advantage of the pattern discipline and the left-linearity of program rules. A *linear unification problem* is a pair of terms: $\Gamma = \langle f(\overline{d_n}), f(\overline{t_n})\rangle$, where $f(\overline{d_n})$ is a linear pattern which does not share variables with $f(\overline{t_n})$. An LU-configuration is a pair (U, σ) where U is a set of pairs $d \downarrow t$, with d a constructor term, and σ is a substitution. Given a linear unification problem $\langle f(\overline{d_n}), f(\overline{t_n})\rangle$, the *initial* LU-configuration is: $(U_0, \sigma_0) = (\{\overline{d_n \downarrow_n t_n}\}, id)$. The following definition presents the unification relation $\rightarrow_{\mathsf{LU}}$ which we use to solve linear unification problems.

Definition 2 (relation $\rightarrow_{\mathsf{LU}}$). *We define the unification relation $\rightarrow_{\mathsf{LU}}$ between LU-configurations as the smallest relation satisfying:*

1. $(\{c(\overline{d_m}) \downarrow_u c(\overline{t_m})\} \cup U, \sigma) \rightarrow_{\mathsf{LU}} (\{\overline{d_m \downarrow_{u.m} t_m}\} \cup U, \sigma)$, *where* $c/m \in \mathcal{C}, m \geq 0$
2. $(\{x \downarrow_u t\} \cup U, \sigma) \rightarrow_{\mathsf{LU}} (\{x/t\}(U), \{x/t\} \circ \sigma)$, *where* $t \notin V$
3. $(\{d \downarrow_u x\} \cup U, \sigma) \rightarrow_{\mathsf{LU}} (\{x/d\}(U), \{x/d\} \circ \sigma)$
4. $(\{c(\overline{d_m}) \downarrow_u c'(\overline{t_n})\} \cup U, \sigma) \rightarrow_{\mathsf{LU}} (\{\mathtt{fail}\}, \sigma)$, *where* $c/m, c'/n \in \mathcal{C}$, $c \neq c'$, *and* $m, n \geq 0$

Let us note that the unification relation $\rightarrow_{\mathsf{LU}}$ is well defined, in the sense that, whenever (U, σ) is a LU-configuration and $(U, \sigma) \rightarrow_{\mathsf{LU}} (U', \sigma')$, then (U', σ') is also a LU-configuration.

We use a function, denoted LU, 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 of positions which demand further evaluation of their arguments. Formally, given a LU-configuration (U, σ), a position u is *demanded*, if $(c(\overline{d_m}) \downarrow_u f(\overline{t_n})) \in U$.

Definition 3 (linear unification). *Let $\Gamma = \langle f(\overline{d_n}), f(\overline{t_n})\rangle$ be a linear unification problem and $(U_0, \sigma_0) = (\{\overline{d_n \downarrow_n t_n}\}, id) \rightarrow^{*}_{\mathsf{LU}} (U, \sigma) \not\rightarrow_{\mathsf{LU}}$ be a LU-derivation to an irreducible LU-configuration. We define the function $\mathsf{LU}(\Gamma)$ as follows:*

$$\mathsf{LU}(\Gamma) = \begin{cases} (\mathtt{Succ}, \sigma) & \mathtt{if}\ U = \emptyset \\ (\mathtt{Fail}, \emptyset) & \mathtt{if}\ U = \{\mathtt{fail}\} \\ (\mathtt{Demand}, P) & \mathtt{otherwise,\ where}\ P \textit{ is the set of demanded positions} \end{cases}$$

As in proof procedures for logic programming, we assume that the program rules always contain fresh variables when they are used in the process of linear unification.

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

Definition 4 (lazy narrowing). *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, $R_k = (l_k \rightarrow r_k)$ is a rule of $\mathcal{R}$ (with fresh variables), and σ is a substitution, as follows:*

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

$$\texttt{where } \lambda'(t,p,R_k) = \texttt{if } \mathcal{H}ead(l_k) = \mathcal{H}ead(t|_p) \texttt{ then case } \mathsf{LU}(l_k, t|_p) \texttt{ of } \begin{cases} (\texttt{Success}, \sigma) : \{\langle p, R_k, \sigma \rangle\} \\ (\texttt{Fail}, \emptyset) : \quad \emptyset \\ (\texttt{Demand}, P) : \bigcup_{q \in P} \bigcup_{k=1}^{m} \lambda'(t, p.q, R_k) \end{cases} \texttt{ else } \emptyset$$

The associated lazy narrowing relation is denoted by $\leadsto_{LN}$. Lazy narrowing is strong complete with respect to strict equations and constructor substitutions as solutions in constructor-based orthogonal programs [20, 22]. Here, *strong completeness* means that, given a conjunction $t_1 \wedge t_2 \wedge \ldots \wedge t_n$, we can non-deterministically select for evaluation just one of the conjuncts, t_i, without losing completeness.[2] It is well-known that lazy narrowing does not define a *pure* lazy evaluation strategy since it might demand the evaluation of expressions which are not really *needed* to compute the outcome [5].

Needed Narrowing In this section, we recall the main notions concerning the needed narrowing strategy of [5]. *Needed Narrowing* reduces the *outermost needed* positions of the input term which are unavoidable to compute the final result. It is defined on *inductively sequential programs* [3], a (strict) subset of constructor-based orthogonal programs. The definition of this class of programs, as well as the needed narrowing strategy, makes use of the notion of a *definitional tree*, first introduced by Antoy [3]; nevertheless, we use the more declarative definition appeared later in [4].

Definition 5 (partial definitional tree). *A* partial definitional tree *of a finite set of linear patterns S is a non-empty set $\mathcal{P}$ of linear patterns partially ordered by the strict subsumption order "$<$"*[3] *and fulfilling the following properties:*

Root property: *There is a minimum element $pattern(\mathcal{P})$—the* pattern *of the partial definitional tree.*

Leaves property: *The maximal elements, called the* leaves*, are the elements of S. Non-maximal elements are also called* branches.

Parent property: *If $\pi \in \mathcal{P}$, $\pi \neq pattern(\mathcal{P})$, there exists a unique $\pi' \in \mathcal{P}$, called the* parent *of π (and π is called a* child *of π'), such that $\pi' < \pi$ and there is no other pattern π'' with $\pi' < \pi'' < \pi$.*

Inductive property: *Given $\pi \in \mathcal{P} \backslash S$, there is a position o in π with $\pi|_o \in \mathcal{X}$ (called the* inductive position*), and constructors $c_1, \ldots, c_n \in \mathcal{C}$ with $c_i \neq c_j$ for $i \neq j$, such that, for all $\pi_1, \ldots, \pi_n$ which have the parent π, $\pi_i = \pi[c_i(\overline{x_{n_i}})]_o$ (where $\overline{x_{n_i}}$ are new distinct variables) for all $1 \leq i \leq n$.*

Given a defined function symbol f/n, we call $\mathcal{P}$ a *definitional tree of f* if $\mathcal{P}$ is a partial definitional tree with $pattern(\mathcal{P}) = f(\overline{x_n})$, 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 may exist more than one definitional tree for a defined function. 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.

Example 1. The following TRS is inductively sequential:[4]

$$f(a, Y) \to a \qquad f(c(X), a) \to b \qquad f(c(X), c(Y)) \to X$$

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

[3] We say that s is strictly more general that t, $s < t$, if there exists a substitution σ such that $t = \sigma(s)$ and σ is not a variable renaming.

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

whereas the following TRS is constructor-based and orthogonal but it is not inductively sequential:

$$f(a,b,X) \to a \qquad f(b,X,a) \to b \qquad f(X,a,b) \to X$$

Needed narrowing can be formalized as follows:[5]

Definition 6 (needed narrowing). *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 $\pi \le t$. We define a mapping 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 $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \le \pi'\}$ be the partial definitional tree where all patterns are instances of π_i. Then, we consider the following cases:*

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

The associated needed narrowing relation is denoted by $\leadsto_{NN}$. Let us remark that each needed narrowing step can be represented as $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_1\rangle$, which is called the *canonical representation* of the step. We assume that the definitional trees always contain fresh variables when they are used in a narrowing step. By abuse, the position p of a needed redex is called "needed narrowing position".

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

Definition 7 (anticipated substitution). *Let $\mathcal{R}$ be an inductively sequential program and R a rule in $\mathcal{R}$. 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 $\theta = \sigma \circ \tau$.*

Let us note that the concept of "anticipated substitution", as informally defined in [5], does not correspond with the more declarative notion introduced above. Technically, the "anticipated substitutions" of [5] are made of those bindings computed by traversing definitional trees which do not correspond to the function which is currently reduced. This notion of "anticipation" has an operational flavor and, as we will show in Section 4.1, it may compute additional bindings that can not be considered as "anticipated" when we use Definition 7. Therefore, in order to distinguish between both concepts, we prefer to refer to the "anticipated substitutions" of [5] as *anticipated bindings*. A formal definition of this concept is given in Definition 9, which constitutes a valuable instrument to prove several results.

Antoy et al. [5] proved that, for inductively sequential programs, needed narrowing is complete w.r.t. strict equations and constructor substitutions as solutions. Moreover, it is optimal w.r.t. the independence of computed solutions and the length of successful derivations (in graph-based implementations).

---

[5] This definition slightly differs from the one appeared in [5], although it computes the same needed narrowing steps if we consider substitutions modulo variable renaming and restricted to $\mathcal{V}ar(t)$.

3 Uniform Programs

Uniform programs were introduced in [16, 17] to improve the implementation of lazy narrowing within the functional logic language Babel [22].

Definition 8 (uniform program). *A uniform program is a TRS fulfilling the following conditions:*

1. Flat constructor patterns*: the arguments in the left-hand sides of program 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 left-hand sides of two rules defining* f*; then,* t_i *is a variable iff* s_i *is a variable, for all* $i \in \{1, \ldots, n\}$.

Basically, functions f/n may be defined in uniform programs by:

- one or more rules whose left-hand sides have 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
- one single rule whose shape is $f(\overline{x_n}) \to r$.

In the former case, for every pair of (distinct) program rules, there exists at least an inductive position in the left-hand sides of these rules where the corresponding arguments are rooted by two different constructors.

Example 2. The following program is uniform

$$f(X, 1, 2, 3) \to 1 \qquad f(X, 2, 2, 3) \to 2 \qquad f(X, 2, 2, 2) \to 3$$

In [23], *simple uniform programs* (i.e., uniform programs whose rules have—at most—one inductive position) were introduced. As an easy consequence of [23, Lemma 2], the equivalence between needed and lazy narrowing holds in simple uniform programs w.r.t. derivations to head normal form.

3.1 Implementation Issues

Moreno et al. showed in [21] that, within a sequential implementation of narrowing using a depth-first search with backtracking, it is difficult to combine lazy evaluation with the use of logical variables. The existence of additional choice points associated with several redexes in the input expression prevents an implementation of lazy narrowing from being efficient. Basically, the reason is that it has to manage these additional choice points together with the usual choice points associated with the selection of different rules (as in logic programming). The following example, taken from [16], illustrates this point.

Example 3. Consider the following (non uniform) program:

$$\begin{array}{rl} f(0, 0) \to 0 & (R_1) \\ f(s(X), 0) \to s(0) & (R_2) \\ f(X, s(s(Y))) \to s(s(0)) & (R_3) \end{array}$$

and the term $t = f(f(X, Y), Z)$. Figure 1 shows the corresponding lazy narrowing tree for t. Selected redexes are underlined and the branches are labeled by the rules of $\mathcal{R}$ used in each step. Initially, the subterm $t|_1 = f(X, Y)$ is demanded by the rules R_1 and R_2, since

the left-hand sides of these rules contain a constructor-rooted term at position 1. Therefore, the application of these rules is delayed until the subterm $t|_1$ is in head normal form. Thus, four possible steps can be proven from t: we can either apply rules R_1, R_2, and R_3 to $t|_1$, or rule R_3 to t. However, once the demanded subterm $t|_1$ is in head normal form, rule R_3 can also be applied, which does not seem very "lazy", since this rule did not demand the evaluation of $t|_1$.

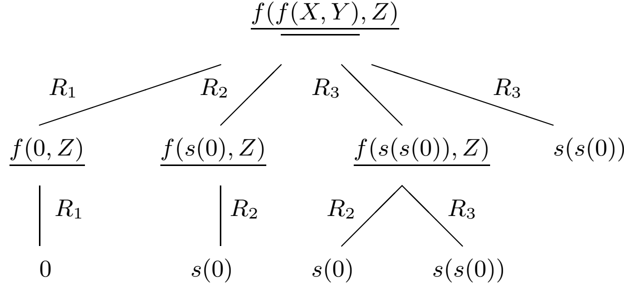

Fig. 1. Lazy Narrowing Tree for $f(f(X,Y),Z)$

As we mentioned above, having rules which demand different redexes is a source of undesired inefficiencies in sequential implementations of lazy languages which may also generate redundant outputs. For instance, the program of Example 3 computes the outputs

$$\{\ \langle s(s(0)), \{X/X_1, Y/s(s(Y_1)), Z/s(s(Y_2))\}\rangle,\ \langle s(s(0)), \{Z/s(s(Y_3))\}\rangle\ \}$$

for the input term $f(f(X,Y),Z)$. Thus, the same result $s(s(0))$ is computed twice, with two different computed answers, where the second answer subsumes (i.e., is more general than) the first one.

Uniform programs were introduced in [16] to avoid these drawbacks. The basic idea is to replace backtracking due to different redexes and rules by backtracking only due to different rules. This is achieved by introducing a new class of programs where there is *exactly one* reducible subterm (which should be demanded by all rules defining a particular function symbol). In [16], a semantics-preserving transformation procedure is also proposed for implementing the functional logic language Babel, which transforms any given inductively sequential program into a uniform one.

Example 4. The transformation procedure proposed in [16] transforms the program $\mathcal{R}$ of Example 3 into the following uniform program:

$$\begin{aligned} f(X,0) &\to h(X) & h(0) &\to 0 \\ f(X,s(Y)) &\to g(Y) & h(s(X)) &\to s(0) \\ g(s(Y)) &\to s(s(0)) \end{aligned}$$

where g and h are new function symbols, different from those in the signature of the original program. In the transformed program, lazy narrowing only computes the output $\langle s(s(0)), \{Z/s(s(Y_3))\}\rangle$ for the input term $f(f(X,Y),Z)$. In general, given an arbitrary term $f(s_1,s_2)$, if s_2 is operation-rooted, then all rules defining f demand the evaluation of s_2. This means that the choice points which are due to different redexes have been removed from the program.

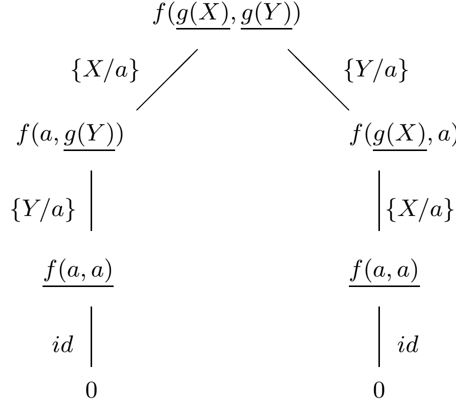

Fig. 2. Lazy Narrowing Tree for $f(g(X), g(Y))$

Unfortunately, even if uniform programs have many advantages (see [16]), they do not generally succeed in removing all additional choice points and getting rid of useless computations, as witnessed by the following example:

Example 5. Let us consider the uniform program:

$$f(a, a) \to 0 \qquad g(a) \to a$$

and the input term $f(g(X), g(Y))$. Lazy narrowing computes the search tree depicted in Figure 2, where several choice points occur which are associated to different demanded redexes of the same term. Note that, positions 1 and 2 of $f(g(X), g(Y))$ are demanded by the first rule, thus giving rise to redundant derivations when the corresponding subterms at these positions are reduced. Hence, the program computes the output $\langle 0, \{X/a, Y/a\}\rangle$ twice.

In uniform programs, different disjoint positions of a term may still be demanded by the rules defining a given function, and this may cause inefficiencies which are not avoided by [16]. In Section 5, we will propose an optimization of lazy narrowing which overcomes this drawback and is still complete for the class of uniform programs.

3.2 Properties and Taxonomy

In this section we prove that uniform programs are inductively sequential. Our proof relies in the fact that, given a uniform program, a definitional tree can be associated to each function f by considering as inductive positions the non-variable positions in the left-hand sides of the rules defining f.

Proposition 1. *Uniform term rewriting systems are inductively sequential.*

Proof. The following procedure builds a set of patterns representing a definitional tree whose root is the term $f(\overline{x_n})$ and whose leaves are the terms of the set $L_f(\mathcal{R})$, which consists of the left-hand sides of the rules defining f:

Input: The set $L_f(\mathcal{R})$ with the left-hand sides of the rules defining f and the set $I = \{k \mid l \in L_f(\mathcal{R}) \wedge l|_k = c_k(\overline{x_{m_k}})\}$ with the inductive positions of f.
Output: A set $\mathcal{P}$ of linear patterns.
Initialization: $\mathcal{P} := \{\pi_0\}$, where $\pi_0 = f(\overline{x_n})$ and $\overline{x_n}$ are new variables.
While $I \neq \emptyset$ do

1. select an arbitrary $k \in I$; $I := I \setminus \{k\}$;
2. $H := \{\pi \mid \pi \text{ is a leaf of } \mathcal{P}\}$;

3. Repeat
   (a) select a leaf $\pi \in H$; $H := H \setminus \{\pi\}$;
   (b) $L_\pi := \{l \mid \ l \in L_f(\mathcal{R}) \wedge \pi \leq l\}$;
   (c) $C := \{c_i(\overline{x_{m_i}}) \mid \ l_i \in L_\pi \wedge \mathcal{H}ead(l_i|_k) = c_i\}$, where $\overline{x_{m_i}}$ are fresh variables;
   (d) for each $c_i(\overline{x_{m_i}}) \in C$ do $\mathcal{P} := \mathcal{P} \cup \{\pi_i\}$, where $\pi_i = \pi[c_i(\overline{x_{m_i}})]_k$.
4. until $H = \emptyset$

Return: $\mathcal{P}$

This algorithm considers all possible cases. In particular, when the function f is defined by a single rule $f(x_1, \ldots, x_n) \to r$, we have $L_f(\mathcal{R}) = \{f(\overline{x_n})\}$ and $I = \emptyset$, thus obtaining immediately the tree $\mathcal{P} = \{f(\overline{x_n})\}$ with a single node.

The set of linear patterns returned by the above algorithm is ordered by the strict subsumption order $<$ and is a definitional tree (according to Definition 5):

- *Root property*: Immediate, since $pattern(\mathcal{P}) = f(\overline{x_n})$ is a minimum element of the set $\mathcal{P}$, since $f(\overline{x_n}) < \pi$ for all $\pi \in \mathcal{P}$.
- *Leaves property*: By construction, the leaves of $\mathcal{P}$ are the elements of the set $L_f(\mathcal{R})$. The elements of $\mathcal{P}$ are obtained from the minimal element $f(\overline{x_n})$, by instantiating each variable which appears in an inductive position, $k \in I$, with the corresponding flat constructor terms (steps 3.b–3.d). Therefore, the elements of $L_f(\mathcal{R})$ are strict instances of their ancestor and, hence, maximal elements of $\mathcal{P}$.
- *Parent property*: By construction, a node $\pi = f(\ldots, x_k, \ldots)$ may have several different children $\pi_i = f(\ldots, c_i(\overline{x_{m_i}}), \ldots)$. Therefore, each node $\pi_i \in \mathcal{P}$ different from $pattern(\mathcal{P})$ has a unique parent $\pi < \pi_i$ and there is no parent π', such that $\pi < \pi' < \pi_i$, since the variable x_k is replaced by a flat constructor $c_i(\overline{x_{m_i}})$ when the pattern π_i is built.
- *Inductive property*: Given a node $\pi \in \mathcal{P} \setminus L_f(\mathcal{R})$, its inductive position is an element of the set I and the step 3.d of the algorithm guarantees that the children π_i of π fulfills the inductive property.

Therefore, we have that uniform programs are a subclass of inductively sequential programs. In general, the following relations hold:

$$\text{simple uniform} \subseteq \text{uniform} \subseteq \text{inductively sequential} \subseteq \text{constructor-based and orthogonal}$$

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:

Example 6. Consider the following (non uniform) program that defines the Boolean function "$\leq$" over natural numbers, represented in the usual Peano's notation, with constructor functions zero "0" and successor "s":

$$0 \ \leq \ N \to true \qquad s(M) \ \leq \ 0 \to false \qquad s(M) \ \leq \ s(N) \to M \ \leq \ N$$

Given the minimal pattern $X_1 \leq X_2$, a definitional tree for $\leq$ can be built if we start by selecting the position 1, but it cannot be built if we first select the position 2.

For uniform programs, the number of possible definitional trees associated to a given function is greater than $n!$, where n is the number of its inductive positions, and the depth of these trees is $n + 1$. Note also that there is a complete one–to–one correspondence between the inductive positions of a function f and the inductive positions in the definitional trees associated to that function. This justifies the name "inductive positions" for those positions in the left-hand sides of a uniform program where flat constructors occur, since they play a similar role as in the definitional trees.

In the following, the *standard* definitional tree of a function is a definitional tree in which the inductive positions 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 [5] and the lazy narrowing strategy of [22].

4.1 Stepwise Equivalence

In order 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 lazy narrowing and needed narrowing steps.

Example 7. Consider the uniform program:

$$\begin{aligned} f(a,b) &\to c \quad (R_1) \\ g(c) &\to b \quad (R_2) \end{aligned}$$

where a, b and c are constructor symbols. Using standard definitional trees for f and g, we have the following needed narrowing derivation:

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

However, the corresponding derivation using lazy narrowing is:

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

In the needed narrowing derivation, the binding X/a is computed in the first step, whereas it is computed in the second step in the corresponding lazy narrowing derivation.

We will formalize the precise correspondence between lazy narrowing and needed narrowing steps in Proposition 2. First, we need some preparatory results.

The following lemma establishes that needed narrowing does not compute anticipated substitutions when the 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-rooted term and $\mathcal{P}$ a definitional tree for the root of t. Then, $\langle \Lambda, R, \sigma \rangle \in \lambda(t, \mathcal{P})$ iff $\langle \Lambda, R, \sigma \rangle \in \lambda_{lazy}(t)$.*

Proof. According to Definition 6, we distinguish two cases:

1. $pattern(\mathcal{P}) = \pi$ is a leaf: Then, the function f is defined by a single rule $R = f(\overline{x_n}) \to r$ and it is immediate to prove that $\lambda(t, \mathcal{P}) = \lambda_{lazy}(t) = \{\langle \Lambda, R, id \rangle\}$, when we consider substitutions restricted to $\mathcal{V}ar(t)$.
2. $pattern(\mathcal{P}) = \pi$ is a branch: First, if $\langle \Lambda, R, \sigma \rangle \in \lambda(t, \mathcal{P})$, for any inductive position o_i of f, we have $t|_{o_i} = x_i$ or $t|_{o_i} = c_i(\overline{s_{n_i}})$; otherwise Λ would not be a position computed by the needed narrowing strategy, λ, which contradicts the initial assumption. Let $R = l \to r$ be one of the rules defining f. By definition of uniform program, each inductive position o_i of l must address a flat constructor term. Moreover, since the step $\langle \Lambda, R, \sigma \rangle \in \lambda(t, \mathcal{P})$ is possible, these flat constructor terms must be rooted by the same constructor symbol (located at position o_i of t, when $t|_{o_i} \neq x_i$). Therefore, $l|_{o_i} = c_i(\overline{x_{n_i}})$ and, by Definition 6, it is easy to prove that $\langle \Lambda, R, \tau_q \circ \ldots \circ \tau_1 \rangle \in \lambda(t, \mathcal{P})$, where q is the number of inductive positions of f and $\tau_i = id$ or $\tau_i = \{x_i / c_i(\overline{x_{n_i}})\}$.
   On the other hand, when we compute $\lambda_{lazy}(t)$, the unification problem is $\Gamma = \langle l, t \rangle$. The corresponding initial LU-configuration is (U_0, σ_0); here we distinguish two kinds of elements: the pairs $l|_{o_i} \downarrow_{o_i} t|_{o_i}$, where i is an inductive position of f, and the remainder pairs where, by definition of uniform program, l_i is a variable. In the following we always

consider substitutions modulo variable renaming and restricted to $\mathcal{V}ar(t)$. Now, using the left-linearity of l, it is easy to prove that the following LU derivation exists:

$$(U_0, \sigma_0) \to^*_{\mathsf{LU}} (\{l|_{o_1} \downarrow_{o_1} t|_{o_1}, \ldots, l|_{o_q} \downarrow_{o_q} t|_{o_q}\}, id) \to^*_{\mathsf{LU}} (\emptyset, \tau_q \circ \ldots \circ \tau_1)$$

Note that the elements of the form $c_i(\overline{x_{n_i}}) \downarrow_{o_i} c_i(\overline{s_{n_i}})$ in the intermediate LU-configurations reduce to id, since the variables coming from the left-hand side of the rules are not considered in the domain of the substitutions. Finally, $\mathsf{LU}(l, t) = (\mathtt{Success}, \tau_q \circ \ldots \circ \tau_1)$ and, therefore, $\langle \Lambda, R, \tau_q \circ \ldots \circ \tau_1 \rangle \in \lambda_{lazy}(t)$.

The proof of the "only if" part is completely analogous.

As we mentioned before, the ability to anticipate some bindings in the needed narrowing 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 head normal form), only the rules which actually demanded the evaluation of $t|_p$ can be applied. Moreover, thanks to the anticipated substitutions, only *strongly sequential redexes* are reduced in needed narrowing derivations [5]. The computation of anticipated substitutions makes the difference between the lazy and needed narrowing strategies. However, in order to state and prove the precise relation between lazy narrowing and needed narrowing steps, we need a more operational and technical notion of "anticipation", called *anticipated bindings*. It is formulated by mimicking the definition of a needed narrowing step. Both notions do coincide for linear terms but they may differ for non-linear terms.

Definition 9 (anticipated bindings). *Let $\mathcal{R}$ be a program, t an operation-rooted term, and $\mathcal{P}$ 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: if π is a leaf, then $\alpha(t, \mathcal{P}) = \{id\}$; otherwise ($\pi$ is a branch), we have*

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

Here, 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 example illustrates the difference between this definition and the notion of anticipated substitution.

Example 8. Consider the uniform program:

$$\begin{array}{llll} f(a,b) \to c & (R_1) & g(a) \to c & (R_3) \\ f(a,c) \to b & (R_2) & g(c) \to b & (R_4) \end{array}$$

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

- Given the term $g(Z)$, $pattern(\mathcal{P}')$ is a branch, and the fourth case of Definition 9(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 9 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 considered 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 by 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

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

and $\langle 2.2, R_3, \{Z/a\}\rangle \in \lambda_{lazy}(f(Z, f(Y, g(Z))))$. 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 lazy narrowing and needed narrowing steps: given a needed narrowing step from t which computes the substitution $\sigma \circ \tau$, where τ are the bindings anticipated in this step, there exists a corresponding lazy narrowing 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, $R \in \mathcal{R}$ a program rule, and t an operation-rooted term. Then, $\langle p, R, \sigma\rangle \in \lambda_{lazy}(\tau(t))$ iff there exists some definitional tree $\mathcal{P}$ for $\mathcal{H}ead(t)$ 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 $\lambda(t, \mathcal{P})$.*

Proof. First, we introduce the auxiliary ordering "$\prec$". Let $n_{\mathcal{F}}(t)$ be the number of defined function symbols in term t, and $\mathcal{DT}(\mathcal{F})$ be the set of definitional trees over a signature $\mathcal{F}$. Then, "$\prec$" denotes the noetherian ordering on $\mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{DT}(\mathcal{F})$ defined by: $\langle t_1, \mathcal{T}_1\rangle \prec \langle t_2, \mathcal{T}_2\rangle$ iff i) $n_{\mathcal{F}}(t_1) < n_{\mathcal{F}}(t_2)$ or ii) $n_{\mathcal{F}}(t_1) = n_{\mathcal{F}}(t_2)$ and $\mathcal{T}_1$ is a proper subtree of $\mathcal{T}_2$ [5]. We prove the claim by complete induction on the order $\prec$.

Base case ($n_{\mathcal{F}}(t) = 1$ and $pattern(\mathcal{P}) = \pi$ is a leaf). Then, $\langle \Lambda, R, id\rangle \in \lambda(t, \mathcal{P})$, $\langle \Lambda, R, id\rangle \in \lambda_{lazy}(t)$, and the anticipated binding is $\tau = id$.

Inductive case. We distinguish two possibilities: If $n_{\mathcal{F}}(t) = 1, \mathcal{P}$ and $pattern(\mathcal{P}) = \pi$ is a branch, then the subterms of t are either variables or constructor terms. Therefore, $\langle p, R, \sigma \circ \tau\rangle \in \lambda(t, \mathcal{P})$ implies $p = \Lambda$. Lemma 1 and Definition 9 guarantee that $\langle p, R, \sigma \circ \tau\rangle \in \lambda_{lazy}(t)$; hence, $\tau = id$ is the anticipated binding and the result follows.

Otherwise ($n_{\mathcal{F}}(t) > 1$), according to the definition of λ, we consider the following possibilities:

1. $pattern(\mathcal{P}) = \pi$ is a leaf. Then, the function f is defined by a single rule $R = f(\overline{x_n}) \to r$, and it is immediate that $\lambda(t, \mathcal{P}) = \lambda_{lazy}(t) = \{\langle \Lambda, R, id\rangle\}$. Therefore, $\langle p, R, id \circ id\rangle \in \lambda(t, \mathcal{P})$ iff $\langle p, R, id\rangle \in \lambda_{lazy}(t)$.
2. $pattern(\mathcal{P}) = \pi$ is a branch. Let o be an inductive position of π. Now, we distinguish three possibilities:

(a) $t|_o = x$. Let $\tau = \{x/c_i(\overline{x_{n_i}})\}$. Then, by Definition 6, $\langle p, R, \sigma' \circ \tau' \circ \tau\rangle \in \lambda(t, \mathcal{P})$ if $\langle p, R, \sigma' \circ \tau'\rangle \in \lambda(\tau(t), \mathcal{P}_i)$, where τ' are the anticipated bindings in $\lambda(\tau(t), \mathcal{P}_i)$. Note that, $n_{\mathcal{F}}(\tau(t)) = n_{\mathcal{F}}(t)$, since τ is a constructor substitution, $\mathcal{P}_i$ is a proper subtree of $\mathcal{P}$ and, thus, the induction hypothesis is applicable in this case. Now, by the inductive hypothesis, we have $\langle p, R, \sigma' \circ \tau'\rangle \in \lambda(\tau(t), \mathcal{P}_i)$ iff $\langle p, R, \sigma'\rangle \in \lambda_{lazy}(\tau'(\tau(t)))$. Finally, by Definition 9, $\tau' \circ \tau$ are the anticipated bindings in $\lambda(t, \mathcal{P})$.

(b) $t|_o = c(\overline{t_n})$. The proof of this case is similar to the previous one.

(c) $t|_o = g(\overline{t_n})$. Then, $\langle o.p, R, \sigma' \circ \tau' \circ id\rangle \in \lambda(t, \mathcal{P})$ if $\langle p, R, \sigma' \circ \tau'\rangle \in \lambda(t|_o, \mathcal{P}')$, where $\mathcal{P}'$ is a definitional tree of g, and τ' are the anticipated bindings in $\lambda(t|_o, \mathcal{P}')$. Since $\mathcal{R}$ is uniform, it is easy to verify that, for some k, $\mathsf{LU}(l_k, t) = (\mathtt{Demand}, P)$ with $o \in P$ and there is no demanded position $u < o$ in t. Therefore, the computation of $\lambda_{lazy}(t)$ first requires to compute $\lambda_{lazy}(t|_o)$; hence, it holds that $\langle p, R, \sigma\rangle \in \lambda_{lazy}(t|_o)$ iff $\langle o.p, R, \sigma\rangle \in \lambda_{lazy}(t)$. By the inductive hypothesis, we have $\langle p, R, \sigma'\rangle \in \lambda_{lazy}(\tau'(t|_o))$ and, then, $\langle o.p, R, \sigma'\rangle \in \lambda_{lazy}(\tau'(t))$. Finally, by Definition 9, $\tau' \circ id = \tau'$ are the bindings anticipated in the computation $\lambda(t, \mathcal{P})$.

As a corollary, we obtain the following result:

$$t \overset{[p,R,\sigma\circ\tau]}{\rightsquigarrow}_{NN} s \quad \text{iff} \quad \tau(t) \overset{[p,R,\sigma]}{\rightsquigarrow}_{LN} s$$

where $\tau \in \alpha(t, \mathcal{P})$ are the bindings anticipated in the computation of $\lambda(t, \mathcal{P})$.

Let us note that Proposition 2 holds for any arbitrarily fixed definitional tree, i.e., for each lazy narrowing step, there exists an appropriate definitional tree such that the corresponding needed narrowing step can be proven.

Finally, we prove a stronger, stepwise equivalence between lazy narrowing and needed narrowing steps over simple uniform programs. This is an easy consequence of Proposition 2 and establishes that, for simple uniform programs, needed narrowing does not produce anticipated bindings.

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

Proof. It follows straightforwardly from the following facts:

- Since $\mathcal{R}$ is a *simple* uniform program (i.e., defined functions have a single inductive position), each function has only one associated definitional tree.
- Let o be the inductive position of $\mathcal{H}ead(t)$. We distinguish two possible cases:
  1. $t|_o \in \mathcal{X}$. Then, by Definition 6, the needed narrowing step is performed at the top position Λ of t. Therefore, by Definition 9, the anticipated substitution for this step is $\tau = id$.
  2. $t|_o \notin \mathcal{X}$. If we assume the existence of either the lazy narrowing and needed narrowing steps then, by Definition 4 and Definition 6, both steps compute the identity substitution; trivially, the anticipated substitution for this step is $\tau = id$. Otherwise, neither the lazy narrowing nor the needed narrowing steps can be proven.
- Finally, by Proposition 2, the result $\lambda_{lazy}(t) = \lambda(t, \mathcal{P})$ follows.

It is worthwhile to note that Corollary 1 is stronger than [23, Lemma 2], which only entails the equivalence for derivations leading to a head normal form. Our result proves a stepwise equivalence and, thus, it holds for derivations leading to arbitrary terms (and not only to head normal form). Moreover, it guarantees that these derivations use the same rules over the same positions at each computation step.

5 Uniform Lazy Narrowing

In this section, we introduce a refinement of lazy narrowing which is equivalent to needed narrowing over uniform programs. Proposition 2 shows that every demanded position of a term (w.r.t. a uniform program) is computed by λ, hence it is needed. On the other hand, 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. Therefore, lazy narrowing can be improved by considering a don't-care selection of demanded positions in uniform programs. This is similar to the *uniform* narrowing strategies introduced in [9] for canonical term rewriting systems, which only exploit one position of the considered term while preserving completeness; these strategies have inspired us the name *uniform lazy narrowing*. We formalize the refined narrowing strategy as follows (again, we assume that the rules of $\mathcal{R}$ are numbered as $R_1, \ldots, R_m$):

Definition 10 (uniform lazy narrowing). *Let $\mathcal{R}$ be a uniform program. We define the uniform lazy narrowing strategy as a set-valued function as follows:*

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

$$\texttt{where } \lambda'_u(t,p,R_k) = \texttt{if } \mathcal{H}ead(l_k) = \mathcal{H}ead(t|_p) \\ \texttt{then case } \mathsf{LU}(l_k, t|_p) \texttt{ of} \\ \begin{cases} (\texttt{Success}, \sigma) : \{\langle p, R_k, \sigma\rangle\} \\ (\texttt{Fail}, \emptyset) : \quad \emptyset \\ (\texttt{Demand}, P) : \bigcup_{k=1}^{m} \lambda'_u(t, p.q, R_k) \\ \qquad\qquad \texttt{with } q = sdc(P) \end{cases} \\ \texttt{else } \emptyset$$

where function $sdc(S)$ selects in a don't-care non-deterministic way an element of S.

The associated uniform lazy narrowing relation is denoted by $\leadsto_{ULN}$. Let us note that, although uniform lazy narrowing only selects one redex position p in a term, there may exist several triples (p, R, σ) associated to this position. Informally speaking, the improved strategy allows us to avoid the computation of redundant steps by exploiting the equivalence with the don't-care non-deterministic selection of a definitional tree in needed narrowing steps. Indeed, one may fix any selection strategy for demanded positions (e.g., left-to-right), similarly to the selection of inductive positions in the construction of a definitional tree. The following example illustrates how the uniform lazy narrowing strategy takes advantage of this improved behavior to disregard some useless lazy narrowing derivations.

Example 9. Consider again the uniform program of Example 5 and the term $f(g(X), g(Y))$. Lazy narrowing computes the search tree depicted in Figure 2. On the other hand, by considering the standard definitional tree for f, needed narrowing computes only the following derivation:

$$f(g(X), g(Y)) \overset{\{X/a\}}{\leadsto}_{NN} f(a, g(Y)) \overset{\{Y/a\}}{\leadsto}_{NN} f(a, a) \overset{id}{\leadsto}_{NN} 0$$

Therefore, the rightmost branch of the lazy narrowing tree is redundant. Similarly, if we fix a different definitional tree for f (the one which first considers the inductive position 2), the leftmost branch of the lazy narrowing tree would be useless. The improved uniform lazy narrowing strategy behaves exactly as the needed narrowing strategy on this example.

5.1 Equivalence of Derivations

Now, we are ready to prove the formal relation between uniform lazy narrowing and needed narrowing derivations over uniform programs. Nevertheless, most of our results could easily be extended to lazy narrowing. First, we state the equivalence between uniform lazy narrowing and needed narrowing derivations leading to a head normal form.

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

1. *$\mathcal{D} = (t \overset{\sigma}{\leadsto}{}^*_{NN}\ s)$ iff $\mathcal{D}' = (t \overset{\sigma}{\leadsto}{}^*_{ULN}\ s)$, where s is in head normal form;*
2. *$\mathcal{D}$ and $\mathcal{D}'$ have the same length and apply the same rules over the same positions on the corresponding steps.*

In order to prove this result, we first need some auxiliary lemmata.

Lemma 2 (independence). *Let $\mathcal{R}$ be a uniform program, t an operation-rooted term, and $\mathcal{P}$ a definitional tree for $\mathcal{H}ead(t)$. If $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_1 \rangle \in \lambda(t, \mathcal{P})$, then*

1. *for all $i, j \in \{1, \ldots, k\}$, with $i \neq j$, $\mathcal{V}ar(\vartheta_i) \cap \mathcal{V}ar(\vartheta_j) = \emptyset$;*
2. *for all $i \in \{1, \ldots, k\}$, if $\vartheta_i = \{x/c_i(x_1, \ldots, x_{n_i})\}$ then $x \in \mathcal{V}ar(t)$.*

Proof. We prove the claim by induction on the number k of bindings in the canonical substitution.

Base case ($k = 1$). Then, $pattern(\mathcal{P}) = \pi$ is a leaf (otherwise $k > 1$, which contradicts the initial assumption). Therefore, $\lambda(t, \mathcal{P}) = \{\langle \Lambda, \pi \to r, id \rangle\}$ and the claims trivially follow.

Inductive case ($k > 1$). Then, $pattern(\mathcal{P}) = \pi$ is a branch. Let o_1 be the inductive position of this branch and consider the three cases of λ in Definition 6. Then, $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_1 \rangle \in \lambda(t, \mathcal{P})$ if one of the following cases hold:

1. $t|_{o_1} = x$, $\vartheta_1 = \{x/c_1(x_1, \ldots, x_{n_1})\}$, and $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_2 \rangle \in \lambda(\vartheta_1(t), \mathcal{P}_1)$. First, we prove claim (1). By the inductive hypothesis, for all $i, j \in \{2, \ldots, k\}$, the bindings ϑ_i and ϑ_j do not share variables. On the other hand, all arguments of the left-hand sides in the rules of a uniform program are flat constructor terms. Thus, for any position o of t which corresponds to an inductive position of f, there is no inductive position o' of f such that $o < o'$. Let o_2 be an inductive position of $\mathcal{P}$, hence disjoint with o_1. Then, it cannot happen that $\vartheta_1(t)|_{o_2} = y$, with $y \in \{x_1, \ldots, x_{n_1}\}$. Hence, $\vartheta_2 = \{y/c_2(y_1, \ldots, y_{n_2})\}$, where $y_1, \ldots, y_{n_2}$ are fresh variables, or $\vartheta_2 = id$, otherwise, and claim (1) follows.
   In order to prove claim (2), we consider that, by the inductive hypothesis, $\vartheta_i = \{y/c_i(y_1, \ldots, y_{n_i})\}$ implies $y \in \mathcal{V}ar(\vartheta_1(t))$ for all $i \in \{2, \ldots, k\}$. Since claim (1) holds, ϑ_i and ϑ_1 do not share variables. Therefore, $y \in \mathcal{V}ar(t)$ and thus ϑ_i satisfies claim (2). Moreover, $x \in \mathcal{V}ar(t)$ and, hence, ϑ_1 satisfies claim (2) too.
2. $t|_{o_1} = c_1(t_1, \ldots, t_n)$, $\vartheta_1 = id$, and $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_2 \rangle \in \lambda(t, \mathcal{P}_1)$. By the inductive hypothesis, the binding ϑ_j satisfies the statements of the lemma, with $j \in \{2, \ldots, k\}$. Now, since $\vartheta_1 = id$, claims (1) and (2) hold by vacuity.
3. $t|_{o_1} = g(t_1, \ldots, t_n)$, $\vartheta_1 = id$, and $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_2 \rangle \in \lambda(t|_{o_1}, \mathcal{P}')$, where $\mathcal{P}'$ is a definitional tree of g. Now, since $\mathcal{V}ar(t|_{o_1}) \subseteq \mathcal{V}ar(t)$, the proof is similar to case (2) above.

As an easy consequence of this lemma, we have that bindings in the canonical representation do not share variables, hence $\vartheta_k \circ \cdots \circ \vartheta_1 = \vartheta_k \cup \cdots \cup \vartheta_1$.

As we showed in Example 8, the notions of anticipated substitution and the anticipated bindings computed by α do not generally coincide. By using the previous lemma, we can split $\tau \in \alpha(t, \mathcal{P})$ as follows: $split(\tau) = \langle \tau_1, \tau_2 \rangle$, where τ_1 is the anticipated substitution and τ_2 are the remaining bindings. The following lemma establishes an interesting relation between the anticipated substitution and the substitution obtained in subsequent needed narrowing steps. Informally speaking, it states that, in a needed narrowing step, the bindings of the anticipated substitution which are not computed in the corresponding lazy narrowing step will be computed anyway in the next needed narrowing step.

Lemma 3. *Let $\mathcal{R}$ be a uniform program, t an operation-rooted term, and $\mathcal{P}$ a definitional tree for $\mathcal{H}ead(t)$. Let $\tau \in \alpha(t,\mathcal{P})$ and $split(\tau) = \langle \tau_1, \tau_2 \rangle$. If $\langle p, R, \sigma \cup \tau \rangle \in \lambda(t,\mathcal{P})$ and $t \overset{[p,R,\sigma\cup\tau_2]}{\leadsto_{LN}} t'$ with $p \neq \Lambda$, then $\langle p', R', \sigma' \cup \tau_1 \rangle \in \lambda(t',\mathcal{P})$ for some position p' and rule R'.*

Proof. It is straightforward, by considering the following facts:

- Given the needed narrowing step $\langle p, R, \sigma \cup \tau \rangle \in \lambda(t,\mathcal{P})$, with $R = l \to r$, the term t is reduced to the term $s = (\sigma \cup \tau_2 \cup \tau_1)(t[r]_p)$. On the other hand, in the corresponding lazy narrowing step, t is reduced to $t' = (\sigma \cup \tau_2)(t[r]_p)$. Let $\mathcal{O}$ be the set of positions o of t such that either $o < p$ or o is disjoint of p, and $t|_o = x$ with $x \in \mathcal{D}om(\tau_1)$. Note that the only difference between s and t' are the anticipated bindings introduced by τ_1 at positions $o \in \mathcal{O}$.
- By Lemma 2, $\mathcal{D}om(\sigma \cup \tau_2) \cap \mathcal{D}om(\tau_1) = \emptyset$ and $\mathcal{D}om(\sigma \cup \tau_2 \cup \tau_1) \subseteq \mathcal{V}ar(t)$.
- If $q \in \mathcal{FP}os(t)$ and either $q < p$ or q is disjoint of p, then $q \in \mathcal{FP}os(t')$ and $\mathcal{H}ead(t'|_q) = \mathcal{H}ead(t|_q)$.
- Since σ and τ_2 are constructor substitutions, which only contain flat constructor terms in their range, they do not introduce new redexes at position q of t such that $t|_q = x$ and $x \in \mathcal{D}om(\sigma \cup \tau_2)$.

Therefore, the context of the subterm $t'|_p$ remains unchanged (up to the possible instantiation of the variables in $\mathcal{D}om(\sigma \cup \tau_2)$). Now, consider the position p' selected for the last needed narrowing step. Then, we should consider the following possibilities for p' [1, Proposition 4.1]:

- $p' \geq p$ and $t'|_p$ is an operator rooted subterm. Now, since needed narrowing revisits from scratch the context of the subterm $t'|_p$ in order to prove the subsequent needed narrowing step $\lambda(t',\mathcal{P})$, then the bindings in τ_1 are recomputed in $\lambda(t',\mathcal{P})$.
- $p' \geq u$, where u is a position immediately above p, and $t'|_p$ is a variable or a constructor rooted subterm. In this case the position p' is determined by the definitional tree of $\mathcal{H}ead(t|_u)$ and, in order to prove the subsequent needed narrowing step $\lambda(t',\mathcal{P})$, all the bindings in τ_1 are recomputed, since each position $o \in \mathcal{O}$ is inspected before the position p' is considered.

The following lemma states a clear relationship between an arbitrary uniform lazy narrowing derivation and another derivation that first applies the portion of the anticipated substitution which uniform lazy narrowing does not compute.

Lemma 4. *Let $\mathcal{R}$ be a uniform program, t an operation-rooted term, and $\mathcal{P}$ a definitional tree for $\mathcal{H}ead(t)$. Let $\langle p_1, R_1, \sigma_1 \cup \tau \rangle \in \lambda(t,\mathcal{P})$ and $\langle p_1, R_1, \sigma_1 \cup \tau_2 \rangle \in \lambda_{ulazy}(t)$, with $\tau \in \alpha(t,\mathcal{P})$ and $split(\tau) = \langle \tau_1, \tau_2 \rangle$. Then, $t \overset{\theta'\,*}{\leadsto}_{ULN} s$ iff $\tau_1(t) \overset{\theta\,*}{\leadsto}_{ULN} s$, where s is in head normal form, $\theta' = \theta \circ \tau_1$, and $\theta = \sigma \circ (\sigma_1 \cup \tau_2)$ for some substitution σ.*

Proof. We prove the claim by induction on the number n of steps in the former derivation.

Base case ($n = 1$). Then, the step $t \overset{[p_1,R_1,\sigma_1\cup\tau]}{\leadsto_{ULN}} s$ must be applied at position $p_1 = \Lambda$ (otherwise, $p_1 > \Lambda$ and the root symbol of s would be a defined function symbol and, thus, not in head normal form). By Lemma 1, $\tau = id$ and the claim follows trivially since $\tau_1 = id$.

Inductive case ($n > 1$). Then, we can split the derivation $t \overset{\theta'\,*}{\leadsto}_{ULN} s$ as follows:

$$t \overset{[p_1,R_1,\sigma_1\cup\tau_2]}{\leadsto_{ULN}} t_1 \overset{\sigma'\,*}{\leadsto_{ULN}} s$$

where $\sigma' = \sigma \circ \tau_1$. The case when $p_1 = \Lambda$ is trivial since, by Lemma 1, $\tau = id$. Hence, let us consider the case when $p_1 \neq \Lambda$. By Lemma 3, we have that $\langle p_2, R_2, \sigma_2 \cup \tau_1 \rangle \in \lambda(t_1,\mathcal{P})$, where p_2 and R_2 are the position and the rule exploited in the second step of the derivation, respectively. Now, we consider two possibilities:

– $\langle p_2, R_2, \sigma_2 \cup \tau_1\rangle \in \lambda_{ulazy}(t_1)$. Then, the anticipated binding is computed in this step and, since $\mathcal{D}om(\tau_1)$ is restricted to $\mathcal{V}ar(t)$, the step $\langle p_2, R_2, \sigma_2\rangle \in \lambda_{ulazy}(\tau_1(t_1))$ can be proved. Now, assume that $t_1 \overset{\sigma'}{\rightsquigarrow}{}^{*}_{ULN}\ s$ has the form

$$t_1 \overset{[p_2,R_2,\sigma_2\cup\tau_1]}{\rightsquigarrow_{ULN}} t_2 \overset{\sigma''}{\rightsquigarrow}{}^{*}_{ULN}\ s,$$

where $\sigma = \sigma'' \circ \sigma_2$. Then, we can also prove the following derivation:

$$\tau_1(t_1) \overset{[p_2,R_2,\sigma_2]}{\rightsquigarrow_{ULN}} t_2 \overset{\sigma''}{\rightsquigarrow}{}^{*}_{ULN}\ s$$

Therefore, the whole derivation $\tau_1(t_1) \overset{\sigma}{\rightsquigarrow}{}^{*}_{ULN}\ s$ can be proven.

– $\langle p_2, R_2, \sigma_2\rangle \in \lambda_{ulazy}(t_1)$. Then, the anticipated binding is computed at some later step. By the inductive hypothesis, since the number of steps in the derivation $t_1 \overset{\sigma'}{\rightsquigarrow}{}^{*}_{ULN}\ s$ (with $\sigma' = \sigma \circ \tau_1$) is strictly lesser than n, the derivation $\tau_1(t_1) \overset{\sigma}{\rightsquigarrow}{}^{*}_{ULN}\ s$ also exists.

On the other hand, since the step $t \overset{[p_1,R_1,\sigma_1\cup\tau_2]}{\rightsquigarrow_{ULN}} t_1 = \sigma_1(\tau_2((t[r_1]_{p_1})))$ can be proven, with $R_1 = l_1 \to r_1$, the corresponding linear unification problem succeeds, i.e., $\mathsf{LU}(\langle l_1, t|_{p_1}\rangle) = (\mathtt{Success}, \sigma_1 \cup \tau_2)$. Now, it is routine to verify that $\mathsf{LU}(\langle l_1, \tau_1(t)|_{p_1}\rangle\rangle) = (\mathtt{Success}, \sigma_1 \cup \tau_2)$ and, hence, $\langle p_1, R_1, \sigma_1 \cup \tau_2\rangle \in \lambda_{ulazy}(\tau_1(t))$. Therefore, the following step can be proven:

$$\tau_1(t) \overset{[p_1,R_1,\sigma_1\cup\tau_2]}{\rightsquigarrow_{ULN}} \sigma_1(\tau_2(\tau_1(t)[r_1]_{p_1}))$$

Since $\mathcal{D}om(\tau_1)$ is restricted to $\mathcal{V}ar(t)$ and the constructor substitutions σ_1, τ_1 and τ_2 are linear and they do not share any variable (Lemma 2), these substitutions can be switched:

$$\sigma_1(\tau_2(\tau_1(t)[r_1]_{p_1})) = \sigma_1(\tau_2(\tau_1(t[r_1]_{p_1}))) = \tau_1(\sigma_1(\tau_2(t[r_1]_{p_1}))) = \tau_1(t_1)$$

Therefore, the following derivation can be proven:

$$\tau_1(t) \overset{[p_1,R_1,\sigma_1\cup\tau_2]}{\rightsquigarrow_{ULN}} \tau_1(t_1) \overset{\sigma}{\rightsquigarrow}{}^{*}_{ULN}\ s$$

For the "only if" direction, the proof is completely analogous.

As a corollary of this lemma, we have the following result: given $\tau \in \alpha(t, \mathcal{P})$, $t \overset{\sigma'}{\rightsquigarrow}{}^{*}_{ULN}\ s$ iff $\tau(t) \overset{\sigma}{\rightsquigarrow}{}^{*}_{ULN}\ s$, where $\sigma' = \sigma \circ \tau$ and s is in head normal form.

Finally, we are ready to prove the main result of this section.

Proof (Theorem 1). We prove each claim separately:

1. This claim is proved by induction on the number n of steps in $\mathcal{D}$. Since the base case ($n = 1$) is perfectly analogous to the base case in the proof of Lemma 4, we proceed with the inductive case ($n > 1$). Let us assume that $\mathcal{D}$ has the following form:

$$\mathcal{D} = (t \overset{[p_1,R_1,\sigma_1\circ\tau_1]}{\rightsquigarrow_{NN}} t_1 \overset{[p_2,R_2,\sigma_2\circ\tau_2]}{\rightsquigarrow_{NN}} \ldots \overset{[p_{n-1},R_n,\sigma_n\circ\tau_n]}{\rightsquigarrow_{NN}} s)$$

where $\theta_i = (\sigma_i \circ \tau_i)$ and $\tau_i \in \alpha(s_{i-1}, \mathcal{P}_{i-1})$, for $i \in \{1, \ldots, n\}$. Then, this derivation can be split as follows: $t \overset{[p_1,R_1,\theta_1]}{\rightsquigarrow_{NN}} t_1 \overset{\theta'}{\rightsquigarrow}{}^{*}_{NN}\ s$, where $\theta' = \theta_n \circ \cdots \circ \theta_2$. Since the needed narrowing step $t \overset{[p_1,R_1,\theta_1]}{\rightsquigarrow_{NN}} t_1$ can be proved, by Proposition 2, there exist the uniform lazy narrowing step $\tau_1(t) \overset{[p_1,R_1,\sigma_1]}{\rightsquigarrow_{ULN}} t_1$. Then, by the inductive hypothesis, we have $t_1 \overset{\theta'}{\rightsquigarrow}{}^{*}_{ULN}\ s$ and, therefore, $\tau_1(t) \overset{[p_1,R_1,\sigma_1]}{\rightsquigarrow_{ULN}} t_1 \overset{\theta'}{\rightsquigarrow}{}^{*}_{ULN}\ s$ Finally, by Lemma 4, we have $t \overset{\theta}{\rightsquigarrow}{}^{*}_{ULN}\ s$, with $\theta = \theta_n \circ \cdots \circ \theta_2 \circ \theta_1$, and the proof concludes.
2. This claim is an easy consequence of Proposition 2.

The following corollary is a direct consequence of Theorem 1.

Corollary 2. *Let $\mathcal{R}$ be a uniform program and e an equation. Then, $e \overset{\sigma}{\rightsquigarrow}{}^{*}_{NN}$ true iff $e \overset{\sigma}{\rightsquigarrow}{}^{*}_{ULN}$ true.*

5.2 Correctness of Uniform Lazy Narrowing

Finally, Corollary 2 together with the soundness and completeness of needed narrowing entail the correctness of uniform lazy narrowing w.r.t. strict equations and constructor substitutions as solutions:

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

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

Furthermore, Theorem 2 implies that uniform lazy narrowing is strongly complete (i.e., demanded positions can be safely selected in a don't-care non-deterministic way without jeopardizing completeness).

6 Improving Needed Narrowing Implementations

The results presented so far can be seen as a theoretical basis for recent implementations of needed narrowing. These implementations usually compile inductively sequential programs to (a Prolog representation of a simple) uniform program. The compiled program is then executed in Prolog *on demand* by means of a suitable predicate which evaluates input arguments to head normal form. Thus, the execution mechanism is basically lazy narrowing. Trivially, for simple uniform programs, lazy narrowing and uniform lazy narrowing are equivalent. In this section, we claim that these implementations could be improved by considering the translation of source programs into (not necessarily simple) uniform programs which are then executed by uniform lazy narrowing. Let us start with a simple example which illustrates the compilation process of PAKCS [13]. Consider the following uniform program:

$$f(a,a) \to a \qquad g(a) \to a \qquad g(b) \to b$$

This program can be transformed into the following simple uniform program:

$$f(a,X) \to f_1(X) \qquad f_1(a) \to a \qquad g(a) \to a \qquad g(b) \to b$$

by considering standard definitional trees. The `curry2prolog` compiler of PAKCS produces the following set of Prolog clauses:

```
f(A,B,C)   :- hnf(A,F), f_1(F,B,C).
f_1(a,A,B) :- hnf(A,E), f_1_a_1(E,B).
f_1_a_1(a,a).

g(A,B) :- hnf(A,E), g_1(E,B).
g_1(a,a).
g_1(b,b).
```

In this program, `f(A,B,C)` is a predicate where `A` and `B` are input arguments and `C` is an output argument which returns the result of a computation; auxiliary predicate `hnf` is used to compute the head normal form of a term. Thus, there is a close correspondence between the execution of the (simple) uniform program by uniform lazy narrowing and the Prolog execution of the compiled program.

Now, let us consider the function call $f(X,g(X))$. It can be evaluated in the original—uniform—program in only two steps, as follows:

$$f(x,\underline{g(x)}) \overset{\{X/a\}}{\leadsto}_{ULN} \underline{f(a,a)} \overset{id}{\leadsto}_{ULN} a$$

However, in the *simple* uniform program, the same computation performs an additional step:

$$\underline{f(x, g(x))} \overset{\{X/a\}}{\leadsto}_{ULN} f_1(\underline{g(a)}) \overset{id}{\leadsto}_{ULN} \underline{f_1(a)} \overset{id}{\leadsto}_{ULN} a$$

In general, when we transform a uniform program into a simple uniform one, we may increase the number of rules proportionally to the number of inductive positions in the corresponding definitional trees. As a consequence, the number of computation steps also grows. Thanks to the strong equivalence between uniform lazy narrowing and needed narrowing over the class of uniform programs, we could compile the original program to a "uniform-like" set of Prolog clauses (rather than to a "simple uniform" Prolog program) and still preserve the correctness of the process. For instance, function f in the previous uniform program could be directly translated into Prolog as follows:

```
f(A,B,C) :- hnf(A,F), hnf(B,G), f_1(F,G,C).
f_1(a,a,a).
```

Let us notice that uniform clauses may require an extra pattern-matching effort in comparison with the corresponding simple uniform rules. Nevertheless, we have performed an experimental evaluation of both compiled Prolog programs and we have measured a speedup of 1.29 (when we execute sufficiently large input terms). Here speedups are the ratio between the execution of the Prolog program obtained by the `curry2prolog` compiler of PAKCS, and the runtimes, for the same input term, of the "uniform-like" set of Prolog clauses.

Regarding failing derivations, the improved behaviour is not always achieved. In particular, it depends on the selection strategy considered in a particular implementation of uniform lazy narrowing (function $sdc(S)$ in Definition 10) and on the definitional tree which is considered to translate the source program into simple uniform form. Nevertheless, the improvement is preserved when the considered selection strategies do agree, i.e., when the inductive positions of a given function are considered in the same order as in the uniform lazy narrowing selection strategy. Moreover, in some cases, uniform lazy narrowing may detect a failing derivation earlier. Consider, for instance, the input term $f(g(X), b)$. While this call immediately fails in the uniform program above, the following (non-deterministic) failing derivations are possible w.r.t. the corresponding simple uniform program:

$$\underline{f(g(X), b)} \overset{\{X/a\}}{\leadsto}_{ULN} f_1(\underline{b})$$
$$\underline{f(g(X), b)} \overset{\{X/b\}}{\leadsto}_{ULN} f_1(\underline{b})$$

A precise comparison would require a deeper study, which goes beyond the scope of this work. More detailed information about the conducted experiments can be found in `http://zeus.inf-cr.uclm.es/www/pjulian/uln.html`.

7 Conclusions

This paper investigates the precise relation between lazy narrowing and needed narrowing. The original contributions are as follows:

- Uniform programs are a subclass of inductively sequential programs (Proposition 1).
- For uniform programs, a formal equivalence between lazy narrowing and needed narrowing derivations (and steps) has been established by using the notion of "anticipated bindings" (Proposition 2).
- An optimized lazy evaluation strategy, called uniform lazy narrowing, has been formulated, which is still complete and strongly equivalent to needed narrowing over the class of uniform programs (Theorem 1).

– Finally, we showed how current compilers for lazy functional programs can be improved by considering a translation of source programs into uniform programs rather than to *simple* uniform programs.

Recent implementations of functional logic languages [6, 19] first transform inductively sequential programs into (a Prolog representation of simple) uniform programs and, then, apply a demand-driven narrowing strategy codified in Prolog. This is the technique most commonly proposed to compute needed narrowing steps in inductively sequential systems even if the correspondence w.r.t. the original needed narrowing computation model was never formalized nor claimed [5]. Since there exist several implementations of functional logic languages which are based on a similar transformation model [6, 12, 16, 18, 19, 22], the results in this work can be seen as a formal demonstration that—under the assumption that the transformation into uniform programs does preserve the semantics—current implementations of needed narrowing indeed fit the intended semantics of the source language. Moreover, the analysis of Section 6 shows that these implementations can be still improved by using uniform programs (instead of simple uniform ones) and uniform lazy narrowing. Nevertheless, uniform lazy narrowing would not improve the implementations described, e.g., in [6, 18, 19], because these implementations already generate Prolog code which does not backtrack over the choice of different demanded positions of the same term.

On the other hand, our results constitute the first proof of the fact that lazy narrowing—and, particularly, the refinement called uniform lazy narrowing—enjoys the optimality properties of needed narrowing over the class of uniform programs. We conjecture that a similar property can be proven for the broader class of *constructor-based orthogonal non-strictly subunifiable* term rewriting systems [8]. This class of programs generalizes the class of uniform programs by allowing nested constructors in the left-hand sides of the rules. However, the proof is not an straightforward extension of our proof scheme, since this class of programs does not fulfill the independence property stated in Lemma 2, which is the key of our proof. Nevertheless, these programs are not used in any current implementation of a functional logic language and, hence, there is no clear practical interest in this result.

Acknowledgements We gratefully acknowledge the anonymous referees for many useful suggestions. This work has been partially supported by CICYT TIC 2001-2705-C03-01, by Acción Integrada Hispano-Italiana HI2000-0161, by Acción Integrada Hispano-Alemana HA2001-0059, by Acción Integrada Hispano-Austriaca HU2001-0019, and the Valencian Research Council under grant GV01-424.

References

1. M. Alpuente, S. Escobar, and S. Lucas. Incremental Needed Narrowing. In *Proc. of the Int'l Workshop on Implementation of Declarative Languages (IDL'99)*, 1999.
2. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, volume 32, 12 of *Sigplan Notices*, pages 151–162. ACM Press, 1997.
3. S. Antoy. Definitional Trees. In *Proc. of the Int'l Conf. on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
4. S. Antoy. Optimal Non-Deterministic Functional Logic Computations. In *Proc. of the Int'l Conf. on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
5. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
6. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the 3rd Int'l Workshop on Frontiers of Combining Systems (FroCoS 2000)*, pages 171–185. Springer LNCS 1794, 2000.

7. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
8. R. Echahed. On Completeness of Narrowing Strategies. In *Proc. of the Int'l Colloquium on Trees in Algebra and Programming (CAAP'88)*, pages 89–101. Springer LNCS 299, 1988.
9. R. Echahed. Uniform Narrowing Strategies. In *Proc. of the Int'l Colloquium on Automata, Languages and Programming (ICALP'92)*, pages 259–275. Springer LNCS 632, 1992.
10. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.
11. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
12. M. Hanus. Efficient Translation of Lazy Functional Logic Programs into Prolog. In *Proc. of the Int'l Workshop on Logic Program Synthesis and Transformation (LOPSTR'95)*, pages 252–266. Springer LNCS 1048, 1995.
13. M. Hanus, S. Antoy, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.3: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2000.
14. M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. of ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.
15. 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.
16. H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. In *Proc. of the Int'l Conf. on Algebraic and Logic Programming (ALP'90)*, pages 298–317. Springer LNCS 463, 1990.
17. H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Implementing a Functional Logic Language with Disequality Constrains. In *Proc. of the Joint Int'l Conference and Symposium on Logic Programming (JICSLP'93)*, pages 207–221. The MIT Press, Cambridge, MA, 1993.
18. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the Int'l Symp. on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714, 1993.
19. F.J. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of the Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
20. A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.
21. J.J. Moreno-Navarro, H. Kuchen, J. Mariño-Carballo, S. Winkler, and W. Hans. Efficient Lazy Narrowing using Demandedness Analysis. In *Proc. of the Int'l Symp. on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 167–183. Springer LNCS 714, 1993.
22. 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.
23. F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In *Proc. of the Int'l Static Analysis Symposium (SAS'97)*, pages 141–159. Springer LNCS 1302, 1997.