Optimal Non-Deterministic Functional Logic Computations*

Sergio Antoy

Portland State University

Abstract. We show that non-determinism simplifies coding certain problems into programs. We define a non-confluent, but well-behaved class of rewrite systems for supporting non-deterministic computations in functional logic programming. We show the benefits of using this class on a few examples. We define a narrowing strategy for this class of systems and prove that our strategy is sound, complete, and optimal, modulo non-deterministic choices, for appropriate definitions of these concepts. We compare our strategy with related work and show that our overall approach is fully compatible with the current proposal of a universal, broad-based functional logic language.

1 Introduction

Curry [4], a recently proposed, general-purpose, broad-based functional logic language, offers lazy evaluation, higher order functions, non-deterministic choices, and a unified computation model which integrates narrowing and residuation. *Curry* models functions by the defined operations of a constructor-based, almost orthogonal, term rewriting system (CAT). Non-determinism occurs typically in three situations: when variables are instantiated during a narrowing step; when certain arguments of a non-inductively sequential function, e.g., the *parallel or*, are selected for evaluations; and when an alternative of a *choice* operator, a device to encapsulate non-deterministic computations, is selected for execution. This careful combination of features ensures true functionality, i.e., the application of a function to a tuple of arguments yields at most one value. This condition simplifies declarative, non-backtrackable I/O, but, we will show shortly, it sacrifices the expressive power of the language.

Non-determinism and expressiveness are key ingredients of functional logic programming. The contribution of this note is the discovery of the existence of a class of rewrite systems that are more non-deterministic and expressive than CATs without loss of the properties that make CATs appealing for *Curry*. Our approach has the following features: (1) it is compatible with the unified computation model for functional and logic programming based on narrowing and residuation [8]; (2) it is compatible with the mechanism, the *choice* operator, that has been proposed to encapsulate non-deterministic computations [4]; (3) it is sound, depending on an implementation option, with respect to either the call-time or the evaluation-time semantics that may be adopted for non-deterministic computations [10]; (4) on deterministic computations it is as efficient as the best currently known strategy [2] and on inherently non-deterministic computations it has the potential to be superior.

* This work has been supported in part by the NSF grant CCR-9406751.

This paper is organized as follows. We show in a few examples, in Section 2, some limitations of *CAT*s for expressing simple non-deterministic computations. Then, after some preliminaries in Section 3, we define in Section 4 a new class of rewrite systems and re-program our examples using this class. In Section 5 we define a narrowing strategy for the new class of systems. In the following Sections 6 and 7 we prove that our strategy is sound, complete, and optimal. In Sections 8 and 9 we briefly compare our approach with related work and we offer our conclusions. Sections 6 and 7 are more technical than the rest of the paper and require more specialized notions of rewrite systems and narrowing strategies. The other sections only require an introductory knowledge of functional logic programming.

2 Motivation

This section presents three small examples of computations that are not conveniently coded using *CAT*s. The first example, rather benign, proposes a simple abstraction that has to be coded in a less clear, natural, and expressive form than it is commonly stated. The second example, more compelling, shows that traditional logic programmers must give up some convenient, familiar programming techniques when functions become available. An attempt to adapt a couple of well-typed, pure Prolog predicates to functional logic programming gives rise to several problems whose solutions require more convoluted code. The third example presents two computations, conceptually very similar, that should both succeed—however, one may fail.

The common root of these difficulties is non-determinism. Functions modeled by *CAT*s are too deterministic for these problems. The ability to handle non-determinism is a major asset of logic programming, perhaps the single most important factor of its success. A consequence of overly limiting non-determinism is that programs for the problems that we discuss must be coded in a less natural, expressive, and declarative form than a high-level, declarative language would lead us to expect. We will show later that by adopting a new class of rewrite systems we gain back all the expressiveness and declarativeness needed for these problems without any loss of soundness, completeness, and/or efficiency.

Example 1. Consider an abstraction dealing with family relationships: There are people, parents, and a person's attribute, having blue eyes. Since *parent* is not a function, the standard functional logic approach to its definition is to cast parenthood as a binary predicate.

```
parent Alice  Beth   = true
parent Fred   Beth   = true
parent Carol  Dianna = true
...
blue_eyed Alice = true
```

With this program, a goal to find out whether *Beth* has a blue-eyed *parent* is

```
(parent x Beth) && (blue_eyed x)
```

where `&&` is *Curry*'s predefined sequential conjunction operator.

There are several mildly undesirable aspects of this program:

- it is not clear from a program equation whether the first argument of *parent* is the parent or the child, e.g., whether *Alice* is a parent of *Beth* or vice versa.
- *parent* is a function less completely defined than it should, e.g., *parent Carol Beth*, is undefined even though we may know that its value is *false*.
- the goal is clumsy and verbose, it is not a natural formalization of the way one would state it: "Is there a blue-eyed parent of Beth?"

Example 2. Consider a program that computes permutations of a list. A naive translation of the standard Prolog approach, e.g., see [12, p. 38], would be

```
permute []     [] = true
permute (x:xs) y  = (permute xs ys) && (insert x ys y)

insert x xs       (x : xs) = true
insert x (y : ys) (y : z)  = insert x ys z
```

Unfortunately, the above program is incorrect in *Curry* for three independent reasons.

- In the second equation of *permute* the symbol *ys* is an extra variable [7,13], i.e., it occurs in the right hand side of the equation, but not in the left hand side.
- Both equations of *insert* are not left-linear, i.e., some variable, e.g., x in the first equation, occurs twice in the left hand side.
- The equations of *insert* create a non-trivial critical pair, i.e., there exists a unifier of the left hand sides of the two equations that does not unify the right hand sides.

Example 3. To keep small the size of this example, we consider an abstract problem. Suppose that *ok* is a unary function that, for all arguments, evaluates to *true* and *double* is a function of a natural number that doubles its argument.

```
ok _ = true

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

Evaluating whether the "*double* of some expression t is *ok*", i.e., solving the goal

```
ok (double t)
```

succeeds regardless of t, i.e., even if t is undefined.

Suppose now that we extend our program with a mechanism to halve numbers. We call it "mechanism," rather than function, because halving odd numbers rounds non-deterministically. Even numbers are halved as usual. Following the standard practice, we code *half* as a predicate.

```
half 0         0     = true
half (s 0)     0     = true
half (s 0)     (s 0) = true
half (s (s x)) (s y) = half x y
```

Now, if we want to find out whether the "*half* of some expression t is *ok*", we must solve the goal

```
(half t x) && (ok x)
```

which requires to evaluate t and consequently is unnecessarily inefficient and may even fail if t cannot be evaluated to a natural. However, we have shown that the analogous computation for *double* always succeeds.

3 Preliminaries

A *narrowing step*, or *step* for short, of a term t is a two-part computation. *Part one* is an instantiation of t, i.e., the application of a substitution to t. *Part two* is a rewrite step, i.e., the application of a rule to a subterm of t. The substitution in part one of a narrowing step is a constructor substitution, and can be the identity. In this case, part one *has no effect* on the term and we consider it only for uniformity. This is consistent with the viewpoint that narrowing generalizes rewriting. It is convenient to consider steps in which part two *has no effect* on a term as well, i.e., there is no rewriting. These steps, called *degenerate*, are not intended to be performed during the execution of a program. We consider them only as a device to prove some results. A non-degenerate narrowing step is denoted $(p, l \to r, \sigma)$. Its application is denoted $t \rightsquigarrow_{(p,l \to r,\sigma)} t'$, were t' is obtained from $\sigma(t)$ by contracting the subterm at p using rule $l \to r$. A degenerate narrowing step is denoted $(=, =, \sigma)$. Its application to a term t yields $\sigma(t)$.

Traditionally, the substitution of a narrowing step is a most general unifier of a term and the left hand side of a rule. Every time we select a rule to compute a step, the rule's variables are new, i.e., different from the variables of any other term and/or rule used to compute the step. Consequently, the substitution of these variables is usually of no interest and we stipulate the following. In a step $t \rightsquigarrow_{(p,l \to r,\sigma)} t'$, σ is a substitution, rather than the traditional unifier, that makes $t_{|p}$ an instance of l. A most general substitution σ makes a term t an instance of a rule's left hand side l is denoted with $t \lhd l$. In this case, the domain of $t \lhd l$ is contained in the set of the variables of t and over these variables $t \lhd l$ is equal to $mgu(t, l)$, a most general unifier of t and l. Note that "$\lhd$" is not a commutative function.

The *composition* $\sigma_1 \circ \sigma_2$ of substitutions σ_1 and σ_2 is defined for all terms t as $(\sigma_1 \circ \sigma_2)(t) = \sigma_2(\sigma_1(t))$. A substitution σ is *idempotent* iff $\sigma \circ \sigma = \sigma$. A most general unifier of a narrowing step is an idempotent substitution. Two substitutions σ_1 and σ_2 are *unifiable* iff there exists a substitution σ such that $\sigma_1 \circ \sigma = \sigma_2 \circ \sigma$. For a discussion of properties of substitutions and unifications used in this paper, see, e.g., [5].

4 Overlapping Inductively Sequential Systems

Below we reformulate at a higher level of abstraction the notion of definitional tree originally proposed in [1]. We assume that defined operations are not "totally undefined." Symbols of this kind are generally regarded as constructors in constructor based systems.

Definition 4. A *pattern* is a term of the form $f(t_1, \ldots, t_n)$, where f is a defined operation and, for all $i = 1, \ldots, n$, t_i is a constructor term. A *definitional tree* of an operation f is a non-empty set $\mathcal{T}$ of linear patterns partially ordered by subsumption and having the following properties up to renaming of variables.

- [leaves property] The maximal elements, referred to as the *leaves*, of $\mathcal{T}$ are all and only variants of the left hand sides of the rules defining f. Non-maximal elements are referred to as *branches*.

– [root property] The minimum element, referred to as the *root*, of $\mathcal{T}$ is $f(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are new, distinct variables.
– [parent property] If π is a pattern of $\mathcal{T}$ different from the root, there exists in $\mathcal{T}$ a unique pattern π' strictly preceding π such that there exists no other pattern strictly between π and π'. π' is referred to as the *parent* of π and π as a *child* of π'.
– [induction property] All the children of a same parent differ from each other only at the position, referred to as *inductive*, of a variable of their parent.

There exist operations with no definitional tree, and operations with more than one definitional tree, examples are in [1]. The existence of a definitional tree of an operation is decidable. In most practical situations, computing a definitional tree of an operation is a simple task. An operation is called *inductively sequential* if it has a definitional tree. A rewrite system is called *inductively sequential* if all its operations are inductively sequential. It follows from the definition that inductively sequential systems are left linear. So far, inductive sequentiality has been studied only for non-overlapping systems [1,2]. In the following, we extend this study to overlapping systems.

The adjective "inductive" is motivated by the fact that, for completely defined operations, the children of a pattern are obtained by "doing a data type induction" on the inductive variable of their parent. This variable is typeset in boldface in the examples shown later.

The next result shows that the left hand sides of the rules of an inductively sequential rewrite program overlap only if they are *variants* (of each other), i.e., one can be obtained from the other by a renaming of variables. The converse obviously holds for any program. This property of overlapping left hand sides is stronger than that of weakly orthogonal rewrite systems. By contrast, in inductively sequential systems there are no specific restrictions on the right hand sides beside those of general rewrite systems.

Proposition 5. *Let f be an inductively sequential operation and $l \to r$ and $l' \to r'$ defining rules of f. If l and l' overlap, then l and l' are variants.*

Proof. Suppose that l and l' overlap. Since f is inductively sequential, there exists a definitional tree $\mathcal{T}$ of f containing leaf patterns π and π' variants of l and l', respectively. If π and π' are distinct patterns, there exists a maximal pattern $\hat{\pi}$ in $\mathcal{T}$ such that $\hat{\pi} < \pi$ and $\hat{\pi} < \pi'$. Let p be the inductive position of $\hat{\pi}$. The subterms of both π and π' at p are rooted by different constructors, hence l and l' cannot overlap. Thus, $\pi = \pi'$, l and l' are both variants of π, and consequently they are variants of each other.

Proposition 5 suggests to group together all the equations whose left hand sides are variants and to code the left hand side only once with alternative choices for the right hand sides. This notation is reminiscent of the definitions of types. Equations so defined become identical to those of *ML* and *Haskell*, except for multiple choices of right hand sides. As in *ML* and *Haskell*, one could adopt priorities among the equations based on textual ordering. This is merely a syntactic issue that we do not pursue in this paper. We revisit our introductory examples using overlapping inductively sequential systems.

Example 6. (Example 1 revisited) Only the definition of *parent* changes.

```
parent Beth   = Alice | Fred
parent Dianna = Carol
...
blue_eyed Alice = true
```

and the goal is

```
blue_eyed (parent Beth)
```

All the mildly annoying, syntactical problems discussed earlier disappear.

Example 7. (Example 2 revisited) The definitions of both *permute* and *insert* change. *Insert* non-deterministically inserts its first argument, an element, into its second argument, a list of elements, either at the head or anywhere but the head. For the second choice we introduce a new operation, *tail_insert*, that implicitly ensures that the second argument of *insert* is a non-empty list. We will further discuss this example in Section 8.

```
permute []       = []
permute (x : xs) = insert x (permute xs)
insert x xs =  (x : xs) | tail_insert x xs
tail_insert x (y : ys) = (y : insert x ys)
```

Example 8. (Example 3 revisited) The definition of *half* changes as follows.

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

The goal becomes

```
ok (half t)
```

and its behavior with respect to evaluation, efficiency, and termination becomes identical to the analogous goal involving *double*.

5 Narrowing

In this section we define a narrowing strategy, which we call **I***nductively Sequential* **N***arrowing* **S***trategy*, or *INS* for short, for possibly overlapping, inductively sequential rewrite systems. In following sections we prove that *INS* is sound, complete, and optimal. *INS* coincide with the **N***eeded* **N***arrowing Strategy* [2], *NN* for short, on non-overlapping systems, but has some relevant differences in general due its larger domain. *NN* has strong normalization properties. In particular:

- *NN* is hyper-normalizing on ground terms, i.e., if a ground term is reducible to a data term, then there exists no derivation that computes an infinite number of *NN* steps,
- *NN* is optimal in the number of steps, provided that the common subterms of a term are shared,
- *NN* only performs steps that are needed to compute root-stable terms, this property is more fundamental than normalization, since it allows us to compute infinitary normal forms [13].

However, *INS* shares only a weaker form of the last property with *NN*. The first two properties do not hold as shown by the following overlapping, inductively sequential rewrite program

```
f = f | 0
```
(1)

INS computes, among others, the derivation $f \to f \to \cdots$. We will argue later that this difference originates from non-determinism and that inherently non-deterministic computations performed by *NN* are in practice similar to, if not less efficient than, those computed by *INS*.

Throughout the rest of the paper, we make the following assumptions:
- Every rewrite system that we discuss is inductively sequential, possibly overlapping.
- Definitional trees are fixed. We choose a tree for every defined operation once and for all and we use it for all our claims. The choice of the tree does not affect a claim.

Lemma 9. *Let $t = f(t_1, \ldots, t_k)$ be an operation-rooted term and $\mathcal{T}$ a definitional tree of f. There exists a pattern in $\mathcal{T}$ that unifies with t.*

Proof. Witness is the root of $\mathcal{T}$.

Definition 10. Let $t = f(t_1, \ldots, t_k)$ be an operation-rooted term, $\mathcal{T}$ the fixed definitional tree of f, and π a maximal pattern of $\mathcal{T}$ that unifies with t. *INS*(t) is the set of all and only the triples of the form (p, R, σ), where p is a position, R is a rule, and σ is a substitution such that:

$$(p, R, \sigma) = \begin{cases} (\Lambda, R, t \lhd l) & \text{if } \pi \text{ is a } \textit{leaf} \text{ of } \mathcal{T}\text{, where} \\ & \quad R = l \to r \text{ is a variant of a rule} \\ & \quad \text{such that } l = \pi; \\ (q \cdot q', R, \eta \circ \eta') & \text{if } \pi \text{ is a } \textit{branch} \text{ of } \mathcal{T}\text{, where} \\ & \quad q \text{ is the inductive position of } \pi, \\ & \quad \eta = t \lhd \pi, \text{ and} \\ & \quad (q', R, \eta') \in \mathit{INS}(\eta(t_{|q})). \end{cases} \tag{2}$$

If t is not operation-rooted, then a triple $s = (p, R, \sigma)$ is in *INS*(t) iff $s \in \mathit{INS}(t')$ for some maximal operation-rooted subterm t' of t.

If $s = (p, R, \sigma)$ is a triple computed by Equation 2 on some term t, then a representation of s of the form $(p_1 \cdot \ldots \cdot p_n, R, \sigma_1 \circ \ldots \circ \sigma_n)$ is called the *canonical decomposition* of s, and n is called the *length* of s, iff either $n = 1$, $p_1 = p = \Lambda$, and $\sigma_1 = \sigma$, or $n > 1$, p_1 and σ_1 are computed by the second case of Equation 2 on t, and $(p_2 \cdot \ldots \cdot p_n, R, \sigma_2 \circ \ldots \circ \sigma_n)$ is the canonical decomposition of a step of $\sigma_1(t_{|p_1})$.

Lemma 11. *If t is a term such that $s = (p, R, \sigma) \in \mathrm{INS}(t)$, then*
- *s is a narrowing step of t,*
- *σ is a constructor substitution, and*
- *the canonical decomposition of s is unique.*

Proof. By induction on the length of the canonical decomposition of s.

Definition 10 is non-trivial. The following informal account of the computation of a narrowing step may help its understanding. Let $t = f(t_1, \ldots, t_k)$ be an operation-rooted term to narrow. We most-generally unify t with some non-deterministically chosen maximal pattern π in a definitional tree $\mathcal{T}$ of f. Let η be the restriction to the variables of t of a most general unifier of t and π. If π is a leaf of $\mathcal{T}$, $\eta(t)$ is a redex and we contract it. If π is a branch of $\mathcal{T}$, we consider the subterm u of $\eta(t)$ at the inductive position of π. u cannot be a variable. If u is operation-rooted, we attempt to narrow it. If u is constructor-rooted, we fail, since it can be proved that $\eta(t)$ cannot be narrowed at the top, which is necessary to narrow t to a data term.

The substitution of a step computed by *INS* is not the restriction of a most general unifier. This is different from most narrowing strategies, the major exception being Needed Narrowing. The "extra" substitution computed by *INS* ensures that future steps of a derivation are necessary. This claim will be proved later. The following simple example clarifies this point.

Example 12. Consider the inductively sequential, non-overlapping program

```
0 <= _         = true
(s _) <= 0     = false
(s x) <= (s y) = x <= y

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

INS computes the step $\bar{s} = (2, 0 + z \to z, \{x \mapsto s\ x', y \mapsto 0\})$ on the term $x \leq y + y$. Without instantiating x in the substitution of $\bar{s}$, the step is still possible, but it could become superfluous depending on later steps. E.g., one could compute

$$x \leq y + y \leadsto_{\{y \mapsto 0\}} x \leq 0 \leadsto_{\{x \mapsto 0\}} \mathit{true}$$

where the first step is useless. However, including $\{x \mapsto s\ x'\}$ in $\bar{s}$ prevents this derivation.

Lemma 13. *Let t be a term and $s = (p, R, \sigma) \in \mathrm{INS}(t)$ and $s' = (p', R', \sigma') \in \mathrm{INS}(t)$.*

- *σ (and σ') is of the form $\{v_1 \mapsto t_1, \dots v_n \mapsto t_n\}$, where for all i in $1, \dots, n$, t_i is a linear term whose set of variables is disjoint from the set of variables of t_j, for $j \neq i$ and j in $1, \dots, n$;*
- *σ and σ' unify iff, for any variable v in the domains of both σ and σ', $\sigma(v)$ and $\sigma'(v)$ unify.*

Proof. We prove the first claim by induction on the length of the canonical decomposition of s. Base case: $\sigma = t \lhd \pi$, for some pattern π. If a variable v is in the domain of σ, then v is a variable of t and $\sigma(v)$ is a most general unifier of the set of term $\{\pi_{|q}\}_{q \in Q}$, where $q \in Q$ iff q is an occurrence of π and $t_{|q} = v$. Since π is a linear term with all new variables the claim is immediate. Ind. case: Let $\sigma = \eta_1 \circ \dots \circ \eta_n$ the canonical decomposition of σ and let η' abbreviate $\eta_2 \circ \dots \circ \eta_n$. Consider $\eta'(\eta_1(v_1))$ and $\eta'(\eta_1(v_2))$, where v_1 and v_2 are distinct variables of t. The claim holds for η_1, the proof being identical to the base case. The claim holds for η' too, by the induction hypothesis. Thus $\sigma(v_1)$ and $\sigma(v_2)$ are lienar and do not share any variable.

We now prove the second claim. Suppose that for any variable v of t in the domains of both σ and σ', $\sigma(v)$ and $\sigma'(v)$ unify. By the first claim, there exists a most general unifier η of the set of sets of terms $\{\{\sigma(v), \sigma'(v)\}_{v \in \mathcal{V}ar(t)}\}$. It is easy to verify that η is a unifier, actually a most general one, of σ and σ'. The opposite implication is immediate.

6 Soundness and Completeness

Proposed notions of soundness and completeness of a calculus with non-deterministic functions [6,10] have not yet obtained a universal consensus. In this paper we take the following approach. A term rewriting system defines *all and only* the allowed steps of a computation, but it does not say which of the generally many steps that are allowed in term should be performed. The selection of a step is the job of a strategy. Since not every step selection policy is useful, a good strategy should guarantee that all and only the results of a computation are reached and that time and space resources are not unnecessarily consumed.

In constructor based system, constructors define data, whereas operations define computations. Thus, a term t is a *computation* and if $t \stackrel{*}{\to} u$ and u is a normal form, i.e., it cannot be further rewritten, then u is a *result* or *value* of t. In constructor based system, as in most functional programming languages, only normal forms that do not contain operations are interesting or legal and we refer to them as *data terms*.

The notions of soundness and completeness arise in computations performed with incomplete information, i.e., in terms containing uninstantiated variables. Narrowing has the potential to fill in the missing information necessary to compute results. We stipulate, as usual, that a variable of sort S stands for all and only the ground data terms of sort S and we consider only well-typed instantiations of a variables. Thus, if t is a computation with incomplete information, i.e., a term with uninstantiated variables, the results of t are all and only the elements of the set of data terms rewritten from $\sigma(t)$, when $\sigma(t)$ is ground and σ is a ground constructor substitution whose domain is the set of variables of t. The soundness and completeness of a narrowing strategy are thus defined as the properties of the strategy to compute only and all, respectively, the results of a term. A differing viewpoint is discussed in Section 8.

In practice, things are slightly more complicated. When we compute with incomplete information, we care not only for the results, but also, and perhaps even more, for the substitutions, called *computed answers*, that allow us to compute the results. For example, the result of the computation $0 \leq x$, where "$\leq$" is the usual relational operator on the naturals defined in Example 12, are the data term *true* and the computed answer set $\{\{x \mapsto 0\}, \{x \mapsto s\ 0\}, \ldots\}$. Infinite sets of computed answers are not unusual and dealing with them is unconvenient at best, thus we relax the requirement that values and computed answers must be ground. This, for example, allows us to represent the result of the above computation, in *Curry*'s *computed expression* notation, as $\{\}\,[]\,\mathit{true}$. The following definitions capture the intuition that we just discussed.

Definition 14. A narrowing strategy S is *sound* iff for any derivation $t \leadsto_\sigma u$ computed by S, $\sigma(t) \stackrel{*}{\to} u$. A narrowing strategy S is *complete* iff for any derivation $\sigma(t) \stackrel{*}{\to} u$, where u is a data term and σ is a constructor substitution, there exists a derivation $t \stackrel{*}{\leadsto}_{\sigma'} u'$ computed by S such that $\sigma' \leq \sigma$ and $u' \leq u$.

Next we prove the soundness and completeness of *INS*. In passing, we observe that rewriting is unaffected by incomplete information.

Proposition 15. *If $t \stackrel{*}{\to} u$, then, for any substitution σ, $\sigma(t) \stackrel{*}{\to} \sigma(u)$.*

Proof. The rewrite relation is closed under substitution.

Theorem 16. *If* INS *computes a narrowing derivation* $A : t \overset{*}{\rightsquigarrow}_\sigma u$*, where* u *is a data term, then* $\sigma(t) \overset{*}{\to} u$*.*

Proof. By induction on the length of A. The base case is trivial. Let A be $t \rightsquigarrow_\eta t' \overset{*}{\rightsquigarrow}_{\eta'} u$. By the definition of narrowing, $\sigma = \eta \circ \eta'$ and $\eta(t) \to t'$. By the definition of rewriting, $\eta'(\eta(t)) \to \eta'(t')$. By the indcution hypothesis, $\eta'(t') \overset{*}{\to} u$. Thus, $\sigma(t) \to \eta'(t') \overset{*}{\to} u$.

The proof of completeness requires a few auxiliary lemmas that shed some light on the ideas of Definitional Tree and Inductively Sequential Narrowing. Lemma 20 is an extension to narrowing of the *Parallel Moves Lemma* [9]. Lemma 22 shows that a pattern is an abstraction of the set of rules that can narrow a term that unifies with the pattern. Lemmas 23 and 24 address the persistence of a step computed by *INS* in a term t after t undergoes respectively an instantiation or another narrowing step. These properties are crucial since a necessary step of t remains necessary if t is further instantiated or another step is performed on t, provided that these operations are compatible with the necessary step. Lemma 26 shows that a step computed by *INS* must be performed, eventually, to reach a certain class of constructor-rooted terms. These computations are more fundamental than needed computations [11]. Theorem 27 shows that any derivation that narrows a term at the root must perform a step computed by *INS* on t. Its proof further shows that *INS* lays the foundations for a sequence of steps that must be performed to compute a root-stable form. Although *INS* does not compute a minimal (most general) unifier for a step, it computes a minimal unifier for this sequence of steps. Theorem 29 addresses the relationship between any derivation to a data term and the *INS* step implicitly performed by this derivation. Corollary 30 shows that if a term t is narrowable to a data term, then there exists an *INS* derivation that computes a similar term. Termination is a relevant aspect of the proof. Finally, Corollary 31 proves the completeness of *INS*.

It is well-known that a set of disjoint redexes in a term can be contracted simultaneously [9]. We generalize this notion to a sets of narrowing step. In addition to the disjointness of redexes, we also require the unifiability of the substitutions of the steps.

Definition 17. If $S = \{(p_i, l_i \to r_i, \sigma_i)\}_{i=1,\ldots,n}$ is a set of narrowing steps of a term t such that σ is an upper bound of the set of substitutions $\{\sigma_i\}_{i=1,\ldots,n}$, and for all distinct i and j in $1,\ldots,n$, $\sigma_i(t)_{|p_i}$ and $\sigma_j(t)_{|p_j}$ are disjoint redexes, then $t' = \sigma(t)[\sigma(r_1),\ldots,\sigma(r_n)]_{|p_1,\ldots,p_n}$ is well defined and independent of the order in which the redexes are contracted. We call $t \rightsquigarrow t'$ a *narrowing multistep.*

Likewise, we generalize to narrowing the notion of descendant [9], which is of paramount importance for investigating the derivation space of a term. Separating the two parts of a narrowing step pays off here.

Definition 18. Let t be a term and $s_1 = (p_1, R_1, \sigma_1)$ and $s_2 = (p_2, R_2, \sigma_2)$ possibly degenerate narrowing steps of t. Steps s_1 and s_2 are *compatible* iff σ_1 and σ_2 unify and if $p_1 = p_2$ then $R_1 = R_2$. If s_1 and s_2 are compatible, $t \rightsquigarrow_{s_1} t_1$, and σ is a most general unifier of σ_1 and σ_2 restricted to the variables of t_1, then the set of descendants of s_2 by s_1, denoted $s_2 \backslash s_1$, is defined as follows.

$$s_2 \backslash s_1 = \begin{cases} \{(=,=,\sigma)\} & \text{if } p_1 = p_2 \\ \{(p_2, R_2, \sigma)\} & \text{if } p_1 \not\leq p_2 \\ \{(p_1 \cdot p \cdot q, R_2, \sigma) \mid r_{1\,|p} = x\} & \text{if } s_1 \text{ is not degenerate,} \\ & \quad R_1 = l_1 \to r_1, \\ & \quad p_2 = p_1 \cdot p' \cdot q, \text{ and} \\ & \quad l_{1\,|p'} = x \text{ is a variable} \end{cases} \tag{3}$$

The notions of descendant of either a step or multistep by either a multistep or a derivation are defined as for rewriting using Equation 3. The descendants of a term t by a narrowing step (p, R, σ) is defined as the descendant of $\sigma(t)$ by the rewrite step at p.

The above definition is a conservative extension of the definition of descendant (residual) for rewriting in orthogonal systems. Narrowing in inductively sequential systems adds two new dimensions to the problem: instantiations and distinct right hand sides of a same left hand side. Compatibility of steps, which always holds for rewriting in orthogonal systems, ensures (see Lemma 20) that after a narrowing step s_1 of a term t we can perform what remains to be done of a narrowing step s_2 of t. The major novelty in our discussion is when two steps differ only in their substitutions. If, after doing one step, we want to catch up with the other step, we may have to further instantiate the term without any rewriting. This situation is consistent and natural with our viewpoint that a narrowing step is a two-part computation.

Example 19. Consider the program that defines addition and multiplication on the naturals in unary representation

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

and the term $t = y + x * y$. Given the following steps of t (only the substitution is indicated in the steps)

$$\begin{array}{l} y + x * y \rightsquigarrow_{s_1 = \{y \mapsto 0\}} x * 0 \\ y + x * y \rightsquigarrow_{s_2 = \{x \mapsto 0\}} y + 0 \end{array}$$

the step $x * 0 \rightsquigarrow_{\{x \mapsto 0\}} 0$ is a descendant of s_2 by s_1 whereas $x * 0 \rightsquigarrow_{\{x \mapsto s\ z\}} 0 + z * 0$ is not.

Lemma 20. *If $t \rightsquigarrow_{s_1} t_1$ and $t \rightsquigarrow_{s_2} t_2$ are compatible steps, then*
- there exist narrowing steps $t_1 \rightsquigarrow_{s_2 \backslash s_1} u$ and $t_2 \rightsquigarrow_{s_1 \backslash s_2} v$;
- $u = v$;
- $t \rightsquigarrow_{s_1} t_1 \rightsquigarrow_{s_2 \backslash s_1} u$ and $t \rightsquigarrow_{s_2} t_2 \rightsquigarrow_{s_1 \backslash s_2} v$ compute the same substitution;
- for every step s of t compatible with both s_1 and s_2, $s \backslash (s_2 \backslash s_1) = s \backslash (s_1 \backslash s_2)$.

Proof. By case analysis of the possible relative positions of the steps s_1, s_2, and s.

Example 21. Consider the functions *half* and *double* defined in Example 8. The diagram of Fig. 1, where f is a binary function whose rules are irrelevant, illustrates Lemma 20. Let $t = f\ (\mathit{half}\ u)\ (\mathit{double}\ u)$. The top left step narrows t at position 1, whereas the top right step narrows t at position 2. Each bottom step is the descendant of the step at the opposite side of the diagram.

$$\begin{tikzcd}
& f\ (\mathit{half}\ u)\ (\mathit{double}\ u) \arrow[dl, "\{u \mapsto s\ (s\ y)\}"'] \arrow[dr, "\{u \mapsto s\ x\}"] & \\
f\ (s\ (\mathit{half}\ y))\ (\mathit{double}\ (s\ (s\ y))) \arrow[dr, "\{\}"'] & & f\ (\mathit{half}\ (s\ x))\ (s\ (s\ (\mathit{double}\ x))) \arrow[dl, "\{x \mapsto s\ y\}"] \\
& f\ (s\ (\mathit{half}\ y))\ (s\ (s\ (\mathit{double}\ (s\ y)))) &
\end{tikzcd}$$

Fig. 1. Pictorial representation of the Parallel Narrowing Moves Lemma. An arrow represents a narrowing step and its label shows the step's substitution.

Lemma 22. *Let t be an operation rooted term, $\mathcal{T}$ the fixed definitional tree of the root of t, π a maximal pattern of $\mathcal{T}$ that unifies with t, and $\eta = t \lhd \pi$. If $l \to r$ is a rule applied to narrow a descendant of $\eta(t)$, then $\pi \leq l$.*

Proof. Consider any derivation $A : \eta(t) \overset{*}{\rightsquigarrow} t' \rightsquigarrow_{(A,l\to r,\sigma)} t''$, where the last step of A is the only step of A at the root, i.e., it is the step that narrows a descendant of $\eta(t)$. If π and l are incomparable, then there exists in $\mathcal{T}$ a maximal pattern π' strictly smaller than both π and l. If p is the inductive position of π', then $l_{|p}$ and $\pi_{|p}$ are rooted by different constructor symbols and so are $l_{|p}$ and $\eta(t)_{|p}$. The only step of A at a position equal to or above p is the last one, since all symbols at a position equal to or above p in $\eta(t)$, except the root, are constructor symbols, thus the root of $t'_{|p}$ is the same as the root of $\eta(t)_{|p}$. This implies that $l \to r$ cannot be applied to narrow t'. Thus, π and l must be comparable, and $\pi \leq l$.

Lemma 23. *Let t be a term such that $s = (p, R, \eta) \in \mathrm{INS}(t)$, and σ an idempotent substitution whose domain is contained in the set of the variables of t. If σ and η are unifiable, then there exists a substitution σ' such that $(p, R, \sigma') \in \mathrm{INS}(\sigma(t))$.*

Proof. Let $(p_1 \cdot \ldots \cdot p_n, R, \eta_1 \circ \ldots \circ \eta_n)$ be the canonical decomposition of s. Let π_i, for i in $1, \ldots, n$, be the pattern used in the i-th application of the second case of Definition 10 during the computation of s. Let M_t and M_π be the n-tuples of terms $\langle t, t_{|p_1}, t_{|p_1 \cdot p_2}, \ldots\rangle$ and $\langle \pi_1, \pi_2, \ldots\rangle$, respectively. Since *INS* computes step s on t, M_t and M_π unify and have a most general unifier $\hat{\eta}$ such that η and $\hat{\eta}$ coincide on the variables of t. Since σ and η unify and η and $\hat{\eta}$ differ only for variables that are not in the domain of σ, σ and $\hat{\eta}$ unify too. Let ρ an unifier of σ

and $\hat{\eta}$. Since ρ unifies σ and $\hat{\eta}$, $\rho(\sigma(M_t)) = \rho(\hat{\eta}(M_t))$. Since $\hat{\eta}$ unifies M_t and M_π, $\hat{\eta}(M_t) = \hat{\eta}(M_\pi)$. By substitutivity, $\rho(\sigma(M_t)) = \rho(\hat{\eta}(M_\pi))$. Since σ is an idempotent substitution, $\rho(\sigma(\sigma(M_t))) = \rho(\sigma(M_\pi))$, hence $\sigma \circ \rho$ is a unifier of $\sigma(M_t)$ and M_π. Thus a computation of *INS* on $\sigma(t)$ parallels the computation of s, except for the unifiers, and *INS* computes a step (p, R, σ') on $\sigma(t)$, for some σ'.

Lemma 24. *Let t be a term, $t \rightsquigarrow_{s_1} t_1$ any narrowing step, $t \rightsquigarrow_{s_2} t_2$ a narrowing step compatible with s_1 computed by* INS *on t, and $s_3 = s_2 \backslash s_1$. If s_3 is not degenerate, then $s_3 \in \mathrm{INS}(t_1)$.*

Proof. Let $s_1 = (p_1, R_1, \sigma_1)$ and $s_2 = (p_2, R_2, \sigma_2)$. Since s_3 is not degenerate, $p_1 \neq p_2$. Since s_1 and s_2 are compatible and s_2 is computed by *INS*, position p_1 cannot be above position p_2. Thus, we only have to prove the claim for the second case of Equation 3. Using Lemma 23, it can be verified that the computation of s_3 parallels that of s_2, except for the unifiers.

Definition 25. We say that a narrowing step $(p, l \to r, \sigma)$ is *root-needed* for an operation-rooted term t iff in any narrowing derivation $t \overset{+}{\rightsquigarrow}_\eta u$, where u is constructor-rooted and $\eta \geq \sigma$, a descendant of $t_{|p}$ is narrowed.

Lemma 26. INS *computes only root-needed steps.*

Proof. Let t be an operation rooted term, $s = (p, l \to r, \sigma)$ a step computed by *INS* on t, and $A : t \overset{+}{\rightsquigarrow}_\eta u$ a derivation of t into a constructor-rooted term u such that $\eta \geq \sigma$. We prove the claim by induction on the length of the canonical decomposition of s. Base case: $\sigma(t)$, hence $\eta(t)$, is a redex and its descendants remain redexes, regardless of other steps of A, until they are narrowed. Ind. case: Let $s = (q \cdot q', l \to r, \tau \circ \tau')$, where π is a maximal pattern in the fixed definitional tree of the root of t that unifies with t, q is the inductive position of π and $\tau = t \lhd \pi$. By Lemma 22, any descendant of $\tau(t)$, hence of $\eta(t)$, can be narrowed at the root only by rules whose left hand sides are instances of π. All these rules have a constructor symbol at position q. However, $\tau(t)$ has an operation symbol at position q. Thus, no descendant of $\tau(t)$ can be narrowed at the root, unless $t_{|q}$ is narrowed to a constructor-rooted term, i.e., a step at q is needed to narrow $\tau(t)$ at the root. By the induction hypothesis, $(t_{|q})_{|q'}$ is root-needed for any instance of $\tau'(\tau(t)_{|q})$. Since $(t_{|q})_{|q'} = t_{|p}$, by transitivity, s is root-need for t.

INS does not compute every root-needed step of a term, but if an operation-rooted term t can be narrowed to a constructor-rooted term, then *INS* always computes a step of t. If a term t is narrowable, but not to a data term, then *INS* may fail to compute any step. This property is a blessing in disguise. For example, consider

```
f = 0
g 0 = 0
h 0 0 = 0
```

and the term $t = h\ (g\ (s\ 0))\ f$. For some definitional tree of h, e.g., $\{h\ \boldsymbol{x}_1\ y_1,\ h\ 0\ \boldsymbol{y}_2,\ h\ 0\ 0\}$, *INS* does not compute any step on t, although t is narrowable (reducible) at position 2. However, t cannot be narrowed (reduced) to a data term. The early failure of *INS* saves performing useless steps.

Lemma 26 shows that *INS* computes only root-needed steps. We now prove a somewhat complementary result roughly equivalent to the fact that *INS* computes all the steps necessary to reach a data term. Because narrowing in non-confluent systems is more general than rewriting in strongly sequential systems, our formulation takes a different form and is broken into several results.

Theorem 27. *Let t be an operation-rooted term and $A : t \overset{+}{\rightsquigarrow}_{\eta'} t'$ a derivation that performs a step at the root.* INS *computes a step $s = (q, R, \eta)$ on t such that $\eta \leq \eta'$.*

Proof. Let $A : t = t_0 \rightsquigarrow_{s_1} t_1 \rightsquigarrow_{s_2} \cdots \rightsquigarrow_{s_n} t_n = t'$, for all i in $1, \ldots, n$, let $s_i = (p_i, l_i \to r_i, \sigma_i)$, and let k be the minimum index such that $p_k = \Lambda$. If t and l_k unify, then $s = (\Lambda, l_k \to r_k, t \lhd l_k) \in \mathit{INS}(t)$ and it can be verified that $t \lhd l_k \leq \sigma_1 \circ \ldots \circ \sigma_k \leq \eta'$. Otherwise, we construct s by structural induction on t. The base case is vacuous. Ind. case: Since t and t_k have the same root symbol, there exists a maximal pattern π in the fixed definitional tree of the root of t such that π and t unify, $\sigma = t \lhd \pi$, and $\pi < l_k$. Let p be the inductive position of π. Since k is the index of the first step of A at the root and p is the position of an inductive variable of π, for all i in $1, \ldots, k-1$, and all q such that $q < p$ and $q \neq \Lambda$, $t_{i|q}$ is constructor-rooted. Hence, in the initial portion of A up to index k, no step occurs at a position strictly above p. Since $t_{|p}$ is operation-rooted and $t_{k-1|p}$ is constructor-rooted, A defines a derivation, which we denote $A_{|p}$, of $t_{|p}$ to a constructor-rooted term. By the induction hypothesis, *INS* computes a step $s' = (p', l \to r, \tau)$ on $t_{|p}$ and $\tau \leq \sigma_1 \circ \cdots \circ \sigma_{k-1}$. We now prove that τ and σ unify. For each variable v in t, let $H_v = \{\pi_{|p} \mid t_{|p} = v\}$. Informally, H_v is the set of the subterms of π matching instances of v in t. σ restricted to v is a most general unifier of H_v. Since the variables in the terms in H_v are all new, $\sigma(v)$ and $\tau(v)$ unify if and only if the set $H_v \cup \{\tau(v)\}$ unifies. Since v is in the domain of σ, there exists an occurrence p_v of π such that $t_{|p_v} = v$. By assumption, $t_{k-1|p_v} = \sigma_1 \circ \cdots \circ \sigma_{k-1}(v)$. Thus, using the induction hypothesis, $\tau(v) \leq t_{k-1|p_v}$. Let $H'_v = \{l_{k|p} \mid \pi_{|p} \in H_v\}$. Since $\pi < l_k$, for all h in H_v, there exists some h' in H'_v such that $h \leq h'$. The k-th step of A implies that the set $H'_v \cup \{t_{k-1|p_v}\}$ unifies. Thus $H_v \cup \{\tau(v)\}$ unifies too and $\sigma(v)$ and $\tau(v)$ unify. Since $\sigma(v)$ and $\tau(v)$ unify for every variable v, by Lemma 13, σ and τ unify. By Lemma 23, there exists a substitution σ' such that $(p', l \to r, \sigma') \in \mathit{INS}(\sigma(t_{|p}))$ and consequently $s = (p \cdot p', l \to r, \sigma \circ \sigma') \in \mathit{INS}(t)$. To prove that $\eta \leq \eta'$ we need an auxiliary result. Let $(p_1 \cdot \ldots \cdot p_n, R, \eta_1 \circ \ldots \circ \eta_n)$ be the canonical decomposition of s. We show, by induction on n, that for all i in $1, \ldots, n$, there exists a step of A whose position is $p_1 \cdot \ldots \cdot p_{i-1}$, where for $i = 1$ we stipulate that $p_1 \cdot \ldots \cdot p_0 = \Lambda$. Base case: when $n = 1$ we only have to prove that A performs a step at the root. This is already an assumption. Ind. case: by the induction hypothesis, we assume the claim for the derivation that earlier we denoted with $A_{|p}$. In this context $p = p_1$. Since $A_{|p_1}$ is embedded into A, the induction hypothesis guarantees that A performs a step at position $p_1 \cdot \ldots \cdot p_{i-1}$, for i in $2, \ldots, n$. Since we also assume that the k-th step of A is at the root, the claim holds for $i = 1$, too. Now, let $\langle \pi_1, \ldots, \pi_n \rangle$ be the tuple of patterns used in the computation of s and, for all i in $1, \ldots, n$, let π'_i be the left hand side of the rule applied to narrow a descendant of $t_{|p_1 \cdot \ldots \cdot p_{i-1}}$, let x be a variable in t, let M_i be the sets of subterms of π_i matching x, and let M'_i be the set of corresponding subterms of π'_i, i.e., for all positions p, $\pi'_{i|p} \in M'_i$ if and only if $\pi_{i|p} \in M_i$. Let $M = M_1 \cup \ldots \cup M_n$ and $M' = M'_1 \cup \ldots \cup M'_n$. By definition of *INS*, $\pi_n = \pi'_n$. For all i in $1, \ldots, n-1$, by Lemma 22, $\pi_i < \pi'_i$. By convention, let the unifier of an empty set be the identity substitution. $\eta(x)$ is a most general unifier of

M and $(\sigma_1 \circ \ldots \circ \sigma_k)(x)$ is a unifier of M'. For each term m in M there exists a term m' in M' such that $m \leq m'$. Thus, $\eta \leq \sigma_1 \circ \ldots \circ \sigma_k$ and consequently $\eta \leq \eta'$.

The proof of previous theorem associates a step computed by *INS* to any derivation that narrows an operation-rooted term to a constructor-rooted term. Next we formally define this step.

Definition 28. Let t be an operation-rooted term and $A : t = t_0 \rightsquigarrow_{s_1} t_1 \rightsquigarrow_{s_2} \cdots \rightsquigarrow_{s_n} t_n$ a derivation that performs a step at the root. For all i in $1, \ldots, n$, let $s_i = (p_i, l_i \to r_i, \sigma_i)$ and let k be the minimum index such that $p_k = \Lambda$. We define the step s *associated* to A as follows. If t and l_k unify, then $s = (\Lambda, l_k \to r_k, t \lhd l_k)$, else we define s by structural induction on t as follows. The base case of the definition is vacuous. Ind. case of the definition: There exists a non-empty set S of patterns of the fixed definitional tree of the root of t that unify with t. Let π be a maximal pattern in S, $\sigma = t \lhd \pi$, and $\pi < l_k$. Let p be the inductive position of π. Since $t_{|p}$ is operation-rooted and $t_{k-1\,|p}$ is constructor-rooted, A defines a derivation, which we denote $A_{|p}$, of $t_{|p}$ to a constructor-rooted term. By induction, let $(p', l \to r, \tau)$ be the step associated to $A_{|p}$. The proof of Theorem 27 shows that σ and τ unify. Hence, by Lemma 23, there exists a substitution σ' such that $(p', l \to r, \sigma') \in \mathit{INS}(\sigma(t_{|p}))$. We define $s = (p \cdot p', l \to r, \sigma \circ \sigma')$.

Theorem 29. *Let $A = A_1; A_2; \ldots A_n$ be a narrowing derivation of a term t_0 to a data term t_n, and let B_0 be the step associated to A. There exists a commutative diagram (arrows stand for narrowing multisteps)*

$$\begin{tikzcd}
t_0 \arrow[r, "A_1"] \arrow[d, "B_0"'] & t_1 \arrow[r, "A_2"] \arrow[d, "B_1"'] & \cdots \arrow[r, "A_n"] & t_n \arrow[d, "B_n"'] \\
t'_0 \arrow[r, "A'_1"] & t'_1 \arrow[r, "A'_2"] & \cdots \arrow[r, "A'_n"] & t'_n
\end{tikzcd}$$

where $A'_{i+1} = A_{i+1} \backslash B_i$ and $B_{i+1} = B_i \backslash A_{i+1}$, for $i = 0, \ldots, n-1$, and the substitution of step B_n is a permutation.

Proof. For $i = 0, \ldots, n$, η_i will denote the substitution of step B_i, and for $i = 1, \ldots, n$, σ_i will denote the substitution of step A_i. Without loss of generality assume that σ_i is a unifier restricted to the set of variables of t_{i-1} and consequently is an idempotent substitution. We begin by proving that for all i in $1, \ldots, n$, if $\eta_{i-1} \leq \sigma_i \circ \sigma_{i+1} \circ \ldots \circ \sigma_n$, then η_{i-1} and σ_i unify. This result will be used later in the proof. Let σ' abbreviate $\sigma_i \circ \sigma_{i+1} \circ \ldots \circ \sigma_n$. Since $\eta_{i-1} \leq \sigma'$, there exists a substitution ρ_i such that $\eta_{i-1} \circ \rho_i = \sigma'$. Applying both sides to η_{i-1} yields $\eta_{i-1} \circ \eta_{i-1} \circ \rho_i = \eta_{i-1} \circ \sigma'$. Since η_{i-1} is an idempotent substitution, $\eta_{i-1} \circ \rho_i = \eta_{i-1} \circ \sigma'$. Since $\eta_{i-1} \circ \rho_i = \sigma'$, $\sigma' = \eta_{i-1} \circ \sigma'$. Since σ_i is an idempotent substitution, $\sigma_i \circ \sigma' = \eta_{i-1} \circ \sigma'$. Thus, σ' is a unifier of η_{i-1} and σ_i. Now we prove that for all i in $1, \ldots, n$, $\eta_{i-1} \leq \sigma_i \circ \sigma_{i+1} \circ \ldots \circ \sigma_n$. The proof is by induction on i. Base case: Immediate from Theorem 27. Ind. case: Assume the claim for $i > 0$. This assumes the existence of the diagram up B_{i-1}. Let y be a variable in the domain of η_i. If $y \in \mathcal{V}ar(t_{i-1})$, then $\sigma_i(y) = y$, then $\eta_{i-1}(y) \leq (\sigma_{i+1} \circ \ldots \circ \sigma_n)(y)$ and $\eta_{i-1}(y) = \eta_i(y)$, then $\eta_i \leq \sigma_{i+1} \circ \ldots \circ \sigma_n$. If $y \notin \mathcal{V}ar(t_{i-1})$, then there exist a variable $x \in \mathcal{V}ar(t_{i-1})$ and a term t_x such that $\sigma_i(x) = t_x$ and $y \in \mathcal{V}ar(t_x)$. $\eta_i(t_x)$ is a lowest upper bound of $\{\eta_{i-1}(x), \sigma_i(x)\}$. Since $\eta_{i-1}(x) \leq (\sigma_i \circ \ldots \circ \sigma_n)(x)$ and $\sigma_i(x) \leq (\sigma_i \circ \ldots \circ \sigma_n)(x)$ we have $\eta_i(t_x) \leq (\sigma_i \circ \ldots \circ \sigma_n)(x)$. Since $\sigma_i(x) = t_x$ we have $\eta_i(t_x) \leq (\sigma_{i+1} \circ \ldots \circ \sigma_n)(t_x)$, hence $\eta_i(y) \leq (\sigma_{i+1} \circ \ldots \circ \sigma_n)(y)$, hence $\eta_i \leq \sigma_{i+1} \circ \ldots \circ \sigma_n$. Thus the entire diagram exists and commutes. Finally, since the substitution of step B_{n-1} is not greater than the substitution of step A_n, we have that the substitution of step B_n is a permutation.

Corollary 30. *Let $A : t \overset{+}{\leadsto}_\sigma u$ a narrowing derivation of a term t to a data term u.* INS *computes a narrowing derivation $B : t \overset{+}{\leadsto}_{\sigma'} u'$ such that $\sigma' \leq \sigma$ and $u' \leq u$.*

Proof. Consider the commutative diagram

$$\begin{tikzcd}
t_{00} \arrow[r, "A_{01}"] \arrow[d, "B_{10}"'] & t_{01} \arrow[r, "A_{02}"] \arrow[d, "B_{11}"'] & \cdots \arrow[r, "A_{0n}"] & t_{0n} \arrow[d, "B_{1n}"'] \\
t_{10} \arrow[r, "A_{11}"] \arrow[d, "B_{20}"'] & t_{11} \arrow[r, "A_{12}"] \arrow[d, "B_{21}"'] & \cdots \arrow[r, "A_{1n}"] & t_{1n} \arrow[d, "B_{2n}"'] \\
\vdots & \vdots & & \vdots
\end{tikzcd}$$

where $A = A_{01}; A_{02}; \ldots; A_{0n}$. Let A_i abbreviate $A_{i0}; A_{i1}; \ldots$, for $i \geq 0$. Let B_i be the step associated to A_i and let B appreviate $B_{10}; B_{20}; \ldots$ We first prove that B terminates. By Theorem 27 and Lemma 26, for all $i \geq 0$, A_i performs a step at the position of B_{i0}. If d is the index of this step, then $A_{(i+1)d}$ is degenerate, whereas A_{id} is not. Thus, for all $i \geq 0$, the number of degenerate steps of A_{i+1} is greater than the number of degenerate steps of A_i. Since the number of steps, degenerate or non, in A is finite, B must have only a finite number of steps. The condition of minimality of σ' and u' are a consequence of the fact that, by the second claim of Theorem 29, the sequence $B_{1n}; B_{2n}; \ldots$ consists of degenerate steps with a substitution equivalent to the identity.

Corollary 31. *If $A : \sigma(t) \overset{*}{\to} u$, where u is a data term and σ is a constructor substitution, then* INS *computes a narrowing derivation $B : t \overset{*}{\leadsto}_{\sigma'} u$ with $\sigma' \leq \sigma$ and $u' \leq u$.*

Proof. If $\sigma(t) = u$, the claim is trivial. Otherwise, let $A : \sigma(t) \to t' \overset{*}{\to} u$. Consider the narrowing derivation $A' : t \leadsto_\sigma t' \overset{*}{\leadsto}_{\{\}} u$. The claim is a direct consequence of Corollary 30 applied to A'.

7 Optimality

Strategies are intended to avoid or minimize unnecessary computations. When non-determinism is involved, the notion of what is or is not necessary becomes subtle. Often, we use non-determinism when we do not know how to make the "right" steps and we cannot expect the narrowing *strategy* to choose for us. Thus, a realistic measure of the quality of a strategy should discount unnecessary work which is performed only because of a wrong non-deterministic choice.

INS makes three kinds of non-deterministic choices in the computation of a step of a term t: (0) a maximal operation-rooted subterm t' of t; (1) a maximal pattern π that unifies with t' in the fixed definitional tree of the root of t', and (2) when π is leaf, a rule whose left hand side is a variant of π. We refer to these choices as *type-0*, *type-1*, and *type-2* choices, respectively. Type-0 choices are *don't care* choices whereas the other choices are *don't know* choices. Type-2 choices originate from overlapping rules whereas type-1 choices occur in non-overlapping systems too, e.g., the strongly sequential ones. The following results show that according to this viewpoint *INS* does not waste a single step, though it may still make "wrong" choices.

Lemma 32. *Let $s_1 = (p_1, l_1 \to r_1, \sigma_1)$ and $s_2 = (p_2, l_2 \to r_2, \sigma_2)$ be narrowing steps computed by* INS *on some operation-rooted term t. If s_1 and s_2 differ for a type-1 choice, then σ_1 and σ_2 are independent.*

Proof. Let $\langle \pi_{11}, \pi_{12}, \ldots \rangle$ and $\langle \pi_{21}, \pi_{22}, \ldots \rangle$ be the sequences of patterns chosen in the computations of s_1 and s_2 respectively. Suppose that k is the smallest index such that $\pi_{1k} \neq \pi_{2k}$. This implies that $p_{11} \cdots p_{1(k-1)} = p_{21} \cdots p_{2(k-1)}$. By Definition 10, $(\sigma_{11} \circ \ldots \circ \sigma_{1k})(t_{|p_{11} \cdots p_{1(k-1)}}) \lhd \pi_{1k}$ and $(\sigma_{21} \circ \ldots \circ \sigma_{2k})(t_{|p_{21} \cdots p_{2(k-1)}}) \lhd \pi_{2k}$. Since π_{1k} and π_{2k} are distinct incomparable patterns of the same definitional tree, they do not unify, and consequently $\sigma_{11} \circ \ldots \circ \sigma_{1k}$ and $\sigma_{21} \circ \ldots \circ \sigma_{2k}$ are independent. Thus σ_1 and σ_2 are independent, too.

Theorem 33. *Let $A_1 : t \overset{+}{\leadsto}_{\sigma_1} t_1$ and $A_2 : t \overset{+}{\leadsto}_{\sigma_2} t_2$ be narrowing derivations computed by* INS*. If A_1 and A_2 differ for a type-1 choice in some step, then σ_1 and σ_2 are independent.*

Proof. Let $A_1 : t = t_{10} \leadsto_{\sigma_{11}} t_{11} \leadsto_{\sigma_{12}} \ldots$ and $A_2 : t = t_{20} \leadsto_{\sigma_{21}} t_{21} \leadsto_{\sigma_{22}} \ldots$ Let k be the smallest index such that the k-th steps of A_1 and A_2 differ for a type-1 choice. We prove the claim by induction on k. Base case: $k = 1$ and the claim is a direct consequence of Lemma 32. Ind. case: Let $k > 1$. By the induction hypothesis, there exist a variable v in $t_{1k} = t_{2k}$ such that $(\sigma_{1k} \circ \sigma_{1(k+1)} \circ \ldots)(v)$ and $(\sigma_{2k} \circ \sigma_{2(k+1)} \circ \ldots)(v)$ do not unify. It is easy to verify [2, Full version, Prop. 2] that either v is in t_{10} or that there exists a variable v' in t_{10} such that $v \in \mathcal{V}ar((\sigma_{11} \circ \ldots \circ \sigma_{1(k-1)})(v'))$. Since A_1 and A_2 are equal up to the k-th step, either condition ensures the claim.

Corollary 34. *A narrowing derivation to a data term computed by* INS *performs only unavoidable steps, modulo non-deterministic choices.*

Proof. The claim is trivial for empty derivations, thus let $A : t \overset{+}{\leadsto}_{\sigma} u$ a non-empty derivation of a term t to a data term u. Type-0 choices do not play any role, since every maximal operation-rooted subterm of a term must be narrowed to a data term.

By Theorem 33, no other derivation with the same type-1 choices computes σ. By Lemma 26, if any step of A involving a type-2 choice were avoided, then u could not be reached.

Example 35. The previous result, does not imply that *INS* computes "the best" derivation. Referring to the operation defined in display (1), both the following derivations satisfy Corollary 34.

$$f \to 0$$
$$f \to f \to 0$$

This fact is not surprising. It is obvious *a posteriori* that the second derivation makes an inappropriate type-2 choice to minimize the number of steps of the derivation. This is not a specific problem of *INS*. It is well-known [9] that even when the rewrite system does not allow type-2 choices, a derivation that makes only unavoidable steps may not minimize their number.

8 Related Work

INS has similarities with both Needed Narrowing [2] and CRWL [6]. The class of rewrite systems to which *INS* can be applied is wider than the class to which *NN* can be applied. *NN* is limited to confluent systems and consequently does not support the kind of non-deterministic computations discussed in Section 2. By contrast, CRWL does not place any specific limitation on rewrite systems. This "in-between" position of *INS* is an asset for the reasons discussed next.

Needed Narrowing is optimal in the number of steps of a derivations, but is likely to be less efficient than *INS* on inherently non-deterministic computations. For example, suppose that we want to solve the N-queen problem with the well-known technique of trial and error. With limited non-determinism, we would follow the standard functional programming approach, e.g., see [3, p. 133–134], which employs list comprehensions and higher order functions. This approach lazily generates permutations, stores them in a structure, and then tests the generated permutations one after another. By contrast, an inductively sequential program, see Example 7, does not require the explicit potential generation of all the permutations nor a structure to hold them nor higher order functions to process this structure. This approach simplifies the programming task and does not incur the cost of explicitly building and later garbage collecting the structure that holds the permutations. We dare to argue that inductively sequential programs would be welcome in functional programming, if this paradigm were equipped for backtracking and non-deterministic choices.

The lack of restrictions placed by CRWL on rewriting systems is not always advantageous. CRWL does not guarantee that computed answers are independent or that every performed step is unavoidable modulo non-deterministic choices. Furthermore, pattern matching for functions defined by not left-linear rules may become computationally expensive. From a software design perspective, programs in CRWL can be poorly structured. E.g., referring to Example 7 the following definition of *insert*, accepted by CRWL, is not inductively sequential.

```
insert x xs       = (x : xs)
insert x (y : ys) = (y : insert x ys)
```

According to this program, the need to evaluate or instantiate the second argument of *insert* depends on the non-deterministic selection of a rule, rather than on the definition of a function. Reasoning about laziness becomes difficult in this situation. By contrast, the definitions of functions in inductively sequential programs are well-understood and familiar. They are exactly the first order Haskell programs except, possibly, for allowing multiple choices of right hand side expressions. Inductive sequentiality imposes on the definition of functions restrictions weaker than those of functional programming, yet we have shown that it provides considerable benefits.

The notion of soundness that we have proposed differs from that adopted in [6]. CRWL adopts what is referred to as *call-time* choice for non-deterministic computations [10]. By contrast, we have adopted an *evaluation-time* choice. Intuitively, the first freezes at the time of a call the non-deterministic choices that will be made to evaluate the arguments of the call, whereas the second does not. Both approaches are plausible and defensible and we defer any decision on their appropriateness to another arena. However, we observe that evaluation-time choice is more natural when semantics are based on rewriting. The call-time choice approach is sound only if some rewrite steps, legal according to the rewrite semantics, are actually prohibited. From a purely computational and implementative point of view, our approach effortlessly supports call-time choice if the common subterms of a term are shared. Sharing would have no negative effects on our strategy completeness, whose definition for call-time choice differs, and would strengthen its optimality by ensuring derivations of minimum length modulo type-2 non-deterministic choices.

9 Conclusion

We have defined a novel class of programs, modeled by possibly overlapping inductively sequential rewrite systems, which simplifies coding non-deterministic functional logic computations. Our approach has been driven by *Curry*, a universal functional logic language currently discussed by researchers in this field, and has been inspired by CRWL, a rewriting logic for declarative programming. We have shown the existence of a sound, complete, and optimal narrowing strategy for inductively sequential programs.

The unified computation model proposed for *Curry* is based on definitional trees. Thus, it can be applied to inductively sequential programs, whether or not overlapping, without any change. When multiple results of non-deterministic computations are not acceptable, the *choice* operator already provided in *Curry* can be used, explicitly or implicitly, to prune the solution space. An implementation option, based on sharing common subterms of a term, accommodates either call-time or evaluation-time choice as the semantics of non-deterministic computations. Deterministic computations are performed by our strategy as efficiently as theoretically possible. In particular, inductively sequential narrowing is a conservative extension of needed narrowing. Inherently non-deterministic computations are performed more efficiently than needed narrowing computations, since it is no longer necessary to cast non-deterministic algorithms into deterministic ones.

There are two directions along which our work could be extended. One is the addition of conditions to rewrite rules. The other is a mild relaxation of the overlapping requirements by allowing trivial critical pairs between rules with multiple right hand sides.

Acknowledgments

This paper would not have been possible without the fruits of my long lasting collaboration with R. Echahed and M. Hanus on narrowing strategies. I also owe to the ideas and explanations of P. López Fraguas, M. Rodríguez-Artalejo, and others on the *Curry* mailing list.

References

1. S. Antoy. Definitional trees. In *Proc. of the 4th Intl. Conf. on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
2. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994. Full versions at URL `www.cs.pdx.edu/~antoy/publications.html`.
3. R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, New York, NY, 1988.
4. Curry: An integrated functional logic language. M. Hanus (ed.), Draft Dec. 5, 1996.
5. E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.
6. J. C. González Moreno, F. J. López Fraguas, M. T. Hortalá González, and M. Rodríguez Artalejo. A rewriting logic for declarative programming. In *ESOP' 96*, Linköping, Sweden, April, 1996. LNCS 1058, Extended version TR DIA 95/5.
7. M. Hanus. On extra variables in (equational) logic programming. In *Proc. Twelfth International Conference on Logic Programming*, pages 665–679. MIT Press, 1995.
8. M. Hanus. A unified computation model for functional logic programming. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 80–93, Paris, 1997.
9. G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*. MIT Press, Cambridge, MA, 1991. Previous version: Call by need computations in non-ambiguous linear term rewriting systems, Technical Report 359, INRIA, Le Chesnay, France, 1979.
10. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent rewriting. *J. of Logic Programming*, 12:237–255, 1992.
11. A. Middeldorp. Call by need computations to root-stable form. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 94–105, Paris, 1997.
12. R. A. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
13. T. Suzuki, A. Middeldorp, and T. Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In *6th International Conference on Rewriting Techniques and Applications*, pages 179–193, Kaiserslautern, 1995. Lecture Notes in Computer Science 914.