Incremental Needed Narrowing*

M. Alpuente, S. Escobar, S. Lucas†

Abstract

Needed narrowing is currently the best complete strategy for executing inductively sequential functional logic programs. Its optimality properties and the fact that inductively sequential programs are a subclass of strongly sequential programs support the claim that needed narrowing must be considered the functional logic couterpart of Huet and Lévy's strongly needed reduction. In this paper, we show how a pre-eminent property of reduction in (a distinguished subclass of) strongly sequential programs, namely the incrementality of the evaluation, can be inherited by needed narrowing. We give an incremental definition of needed narrowing and show that the original optimality properties are kept. Moreover, we experimentally demonstrate that the incremental refinement can lead to substantial improvements in the overall evaluation process.

Keywords: Functional logic languages, Implementation, Incrementality, Needed narrowing.

1 Introduction

For orthogonal term rewriting systems (TRSs), Huet and Lévy's strong sequentiality provides a formal basis for the mechanization of sequential, normalizing rewriting computations [HL79, HL91]. In [HL79, HL91], Huet and Lévy defined the notion of strongly needed redex and showed that, for the class of strongly sequential TRSs (SS), the steady reduction of strongly needed redexes is normalizing. Huet and Lévy showed that the strong sequentiality of a TRS is decidable. They also provided an algorithm to identify a strongly needed redex within a reducible term, thus giving an effective procedure for implementing rewriting computations: given an input expression, first use Huet and Lévy's algorithm to find out a redex; if such a redex does not exist, then stop (the term is in normal form); otherwise, reduce the redex using the (unique) rule that can be applied; repeat these steps as long as possible. Within Huet and Lévy's algorithm, the search for a new redex after each replacement starts from the root of the term. In general, only this way to proceed is safe; nevertheless, Strandh introduced

*This work has been partially supported by CICYT TIC 98-0445-C03-01 and Acción Integrada hispano–alemana HA1997-0073.

†DSIC, UPV. Camino de Vera, E-46022 Valencia, Spain. E-mail: {`alpuente,sescobar,slucas`}`@dsic.upv.es`

the so-called forward branching TRSs (FB) and provided an algorithm which does not only hit upon a strongly needed redex in the term but also performs the corresponding replacement and resumes the reduction process incrementally, that is, *without* (necessarily) turning back to the root of the term in order to start the search for a new redex [Str89]. FB is a subclass of SS [Dur94]. Strandh proved that his *incremental* algorithm preserves the optimality and normalizing properties of Huet and Lévy's reduction of strongly needed redexes. Toyama et al. defined the class of transitive systems (TR) and the notion of *transitive index* which, for transitive systems, can be advantageously used for incremental evaluation of terms also [TSvEP93]. Durand showed that FB = TR [Dur94].

Functional logic languages (see [Han94] for a survey) with a complete operational semantics are usually based on *narrowing*, an operational mechanism which combines reduction (from the functional part) and variable instantiation (from the logic part) [Hul80, Red85, Sla74]. A *narrowing step* instantiates variables of an expression and then applies a reduction step to a redex of the instantiated expression. Unfortunately, simple narrowing can have a huge search space and great effort has been made to develop sophisticated narrowing strategies without losing completeness. *Needed narrowing* [AEH94] is currently the best complete narrowing strategy due to its optimality properties regarding the length of the derivations and the independence of computed answers [AEH94]. Based on the notion of constructor system and inductive sequentiality [Ant92], needed narrowing rebuilds, in the functional logic setting, the unique (peerless) properties of the strongly needed reduction of (pure) equational and functional programs [O'D98, PvE93]. In [AEH94], needed narrowing is formalized by means of a function λ which, for a given term, yields the position, rule, and substitution to be applied in order to reduce the considered expression. Therefore, the function λ accomplishes the role of Huet and Lévy's procedure to implement needed reduction and, similarly to that one, the corresponding calculus is not incremental.

For constructor systems, inductive sequentiality and strong sequentiality coincide [HLM98]. Since it is known that constructor strongly sequential systems are forward branching [DS93], it seems reasonable to consider the possibility to define an incremental needed narrower which is able to efficiently conduct needed functional logic reductions. In this paper, we first give an incremental definition λ^{ι} of needed narrowing and demonstrate that it is strongly equivalent to the original function λ (also concerning its optimality results [AEH94]) and more efficiently implementable thanks to the incremental extent. Our incremental definition covers two different features of the implementation of needed narrowing: first, we provide an incremental definition of the data structure which is used to guide needed narrowing computations, namely, the *definitional trees*, and then we formalize an incremental definition of λ, i.e., an incremental procedure which actually implements needed narrowing computations.

We have applied our techniques to the optimization of the multi-paradigm language Curry [HKMN95, HAK^{+}99], an extension of Haskell [HPW92] which is supported by an international initiative with the aim of becoming a standard in the area and whose operational model is based on needed narrowing. We prove that all the proposed optimizations are actually (and independently) effective by comparing a basic, non-optimized version of UPV-Curry (a novel, public implementation of Curry [EAL98]), and three refined implementations. Finally, we provide an experimental comparison of our implementation and other existing Curry interpreters, namely TasteCurry and PACS-TasteCurry, the most popular (and, the only public currently available) implementations of Curry.

Compiling functional logic programs into another high-level language for which efficient implementations exist is a well known technique for the efficient implementation of integrated languages [Han94]. Compilation into Prolog is the most popular option [BCG+89, LLR93, Han95]. Prolog-based implementations implicitly reap the benefits of incrementality thanks to Prolog's SLD resolution mechanism. Unfortunately, no formal demonstration that these kinds of implementations fit the original (nonincremental) semantics of Curry had been provided before. By having proved the strong equivalence between the standard Curry semantics and the incremental one, our work can be thought of as a formal warranty that compiling functional logic programs into Prolog code is not only simple and practical but also reliable. A similar argument applies to those implementations based on abstract narrowing machines [KLMR90], which also enjoy the incrementality implicitly which is embedded into standard compilation techniques.

On the other hand, programmers working on other implementation techniques can also rely on the present work for achieving incremental evaluation safely. Although it cannot compete with an implementation based on the compilation into low-level (abstract) machine code, our interpreter written in Prolog demonstrates that all optimizations are effective and could generate new insights for further developments in this field.

In Section 2, we give some Preliminares. Section 3 describes the proposed incremental optimization of definitional trees. Section 4 formalizes the incremental needed narrowing calculus. Section 5 gives some experimental results. Section 6 concludes. Benchmark programs are given in Appendix.

2 Preliminaries

We assume familiarity with basic notions of term rewriting [DJ90] and functional logic programming [Han94]. We consider a *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. operation The set of *terms* and *constructor terms* with *variables* (e.g., x, y, z) from $\mathcal{X}$ are denoted by $\mathcal{T}(\Sigma, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. We write $\overline{o_n}$ for the *list of objects* $o_1, \ldots, o_n$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term is *linear* if it does not contain multiple occurrences of one variable. $root(t)$ denotes the symbol at the root of the term t. If $root(t) \in \mathcal{F}$ ($root(t) \in \mathcal{C}$), the term t is said *operation-rooted* (*constructor-rooted*). A *pattern* is a linear term of the form $f(\overline{d_n})$ where $f \in \mathcal{F}$ is n-ary and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$.

A *position* p in a term t is represented by a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). $\mathcal{P}os(t)$ denotes the set of positions of t. Positions are ordered by the *prefix* ordering: $p \leq q$, if there exists p' such that $p.p' = q$. $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing* $t|_p$ by the term s (see [DJ90] for details). $\mathcal{FP}os(t)$ is the set of positions of operation-rooted subterms of t.

We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. The identity substitution is denoted by id. Substitutions are extended to morphisms on terms by $\sigma(f(\overline{t_n})) = f(\overline{\sigma(t_n)})$ for every term $f(\overline{t_n})$. Given terms t, s, we write $t \leq s$ if $\exists \sigma. s = \sigma(t)$ and say that t is a prefix of s (said *strict* if $t < s$). A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$.

A set of rewrite rules $l \to r$ such that $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (*lhs*) and the *right-hand side* (*rhs*) of the rule, respectively. A TRS $\mathcal{R}$ is left-linear if l is linear for all $l \to r \in \mathcal{R}$. A TRS is constructor based (CB) if each lhs l is a pattern. Two (possibly renamed) rules $l \to r$ and $l' \to r'$ *overlap*, if there is a non-variable position $p \in \mathcal{FP}os(l)$ and a most-general unifier σ such that $\sigma(l|_p) = \sigma(l')$. The pair $\langle \sigma(l)[\sigma(r')]_p, \sigma(r) \rangle$ is called a critical pair. A left-linear TRS without critical pairs is called *orthogonal*. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists $p \in \mathcal{P}os(t)$, a rewrite rule $R = l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. The instantiated lhs $\sigma(l)$ is called a *redex*. To evaluate terms containing variables, *narrowing* non-deterministically instantiates the variables such that a rewrite step is possible. Formally, $t \leadsto_{(p,R,\sigma)} t'$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \to_{p,R} t'$. We denote by $t_1 \leadsto^*_\sigma t_n$ a sequence of narrowing steps $t_1 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_{n-1}} t_n$ with $\sigma = \sigma_{n-1} \circ \cdots \circ \sigma_1$.

3 Definitional trees

Given a TRS $\mathcal{R}$, a *definitional tree* $\mathcal{T}$ with pattern π is an expression of the form[1] [Ant92]:

$\mathcal{T} \;=\; rule(\pi = r')$ where $\pi = r'$ is a variant of a rule $l = r \in \mathcal{R}$.

$\mathcal{T} \;=\; branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ where o is a variable position of π (called the inductive position), $c_1, \ldots, c_n$ are different constructors for $n > 0$, and each $\mathcal{T}_i$ is a definitional tree with pattern $\pi[c_i(x_1, \ldots, x_k)]_o$ where k is the arity of c_i and $x_1, \ldots, x_k$ are new variables.

A defined symbol f is called *inductively sequential* if there exists a definitional tree $\mathcal{T}$ with pattern $f(x_1, \ldots, x_k)$ (where $x_1, \ldots, x_k$ are different variables) whose rule nodes contain all (and only) the rules defining f. In this case, we say that $\mathcal{T}$ is a definitional tree for f. A TRS $\mathcal{R}$ is *inductively sequential* if all its

[1] We ignore the *exempt* nodes which are in the original definition of [Ant92]. For simplicity, we also ignore the *or* nodes used in Curry in the development of our technical results, though they have been considered in our implementation since, when using *or* nodes, a disjunctive Curry expression is obtained and, thus the problem essentially boils down to the simple case we have considered in our formalization.

defined function symbols are inductively sequential. An inductively sequential TRS can be viewed as a set of definitional trees, each defining a function symbol.

Example 3.1 *Given the following function definition:*

```
first Z     y      = []
first (S n) (x:xs) = x : (first n xs)
```

The associated definitional tree is:

$$
\begin{array}{l}
\mathit{branch}(\ \texttt{first x y}, 1, \\
\qquad \mathit{rule}(\texttt{first Z y = []}), \\
\qquad \mathit{branch}(\ \texttt{first (S n) y}, 2, \\
\qquad\qquad \mathit{rule}(\texttt{first (S n) (m:ms) = m:(first n ms)})))
\end{array}
$$

Figure 1 shows the graphical representation of this definitional tree. Each branch node shows the associated pattern and its inductive position, which is enclosed by a box.

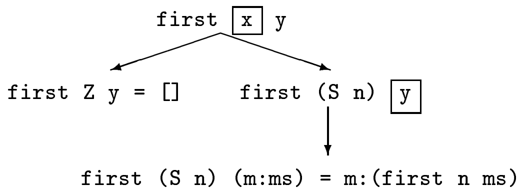

Figure 1: Definitional tree for the function `first`

Definitional trees are used for describing needed narrowing computations as follows. A function λ from terms t and definitional trees to sets of triples (p, R, σ) determines each evaluation step. Here, p is a position in t, R is the rule to be applied, and σ is a substitution which should be applied to t before reducing it. Given an operation-rooted term t and a definitional tree $\mathcal{T}$ with $\mathit{pattern}(\mathcal{T}) = \pi$ such that $\pi \leq t$, then $\lambda(t, \mathcal{T})$ is defined as follows [AHLV99]:

$$
\lambda(t,\mathcal{T}) \ni \begin{cases}
(\Lambda, l \to r, id) & \text{if } \mathcal{T} = \mathit{rule}(l = r); \\
(p, l \to r, \sigma \circ \tau) & \text{if } \mathcal{T} = \mathit{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n),\ t|_o = x \in \mathcal{X}, \\
 & \quad \tau = \{x \mapsto c_i(\overline{x_k})\}, \\
 & \quad \mathit{pattern}(\mathcal{T}_i) = \pi[c_i(\overline{x_k})]_o, \text{ and } (p, l \to r, \sigma) \in \lambda(\tau(t), \mathcal{T}_i); \\
(p, l \to r, \sigma) & \text{if } \mathcal{T} = \mathit{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n),\ t|_o = c_i(\overline{t_k}), \\
 & \quad \mathit{pattern}(\mathcal{T}_i) = \pi[c_i(\overline{x_k})]_o, \text{ and } (p, l \to r, \sigma) \in \lambda(t, \mathcal{T}_i); \\
(o.p, l \to r, \sigma) & \text{if } \mathcal{T} = \mathit{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n),\ t|_o = f(\overline{t_k}) \text{ for } f \in \mathcal{F}, \\
 & \quad \mathcal{T}' \text{ is a definitional tree for } f, \text{ and } (p, l \to r, \sigma) \in \lambda(t|_o, \mathcal{T}')
\end{cases}
$$

This definition ensures that, whenever $(p, l{\to}r, \sigma) \in \lambda(t, \mathcal{T})$, $t \rightsquigarrow_{(p, l \to r, \sigma)} s$ is a valid narrowing step which is called a *needed* narrowing step. Needed narrowing only evaluates subterms which are needed to compute a value and it is the best narrowing strategy for inductively sequential TRSs due to its optimality properties w.r.t. the length of derivations and the independency of computed solutions [AEH94].

3.1 Incremental definitional trees

Definitional trees are used to guide the search for a redex within the computation of λ (also instrumenting the possible variable instantiations). From the definition of λ, it is immediate to see that the pattern π of a branch node $branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ is simply *augmented* by a flat term $c_i(x_1, \ldots, x_k)$, which is placed at the inductive position o of π, in order to attain the patterns of the (root nodes of the) sons $\mathcal{T}_i$, for $1 \leq i \leq n$. On the other hand, the decision about the definitional tree to be considered in the next step, only depends on the constructor symbol c_i rather than on the entire pattern of the son $\mathcal{T}_i$.

Thus, we have developed an optimized representation of the definitional trees which moves the pattern matching information to the exact place where it is needed. Given a TRS $\mathcal{R}$, an *incremental definitional tree* $\mathcal{I}$ with pattern π is an expression of the form:

$\mathcal{I} = irule(\pi = r')$ where $\pi = r'$ is a variant of a rule $l = r \in \mathcal{R}$.

$\mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n))$ where o is a variable position of π, $c_1, \ldots, c_n$ are constructors for $n > 0$, and each $\mathcal{I}_i$ is an incremental definitional tree with pattern $\pi[c_i(x_1, \ldots, x_k)]_o$ where k is the arity of the constructor c_i and $x_1, \ldots, x_k$ are new variables.

The pattern of an incremental definitional tree, which is not explicitly included in its syntactic structure, can be easily obtained as follows:

$$pattern(\mathcal{I}) = \begin{cases} \pi & \text{if } \mathcal{I} = irule(\pi = r); \\ pattern(\mathcal{I}_1)[x]_o & \text{if } \mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n)) \text{ and} \\ & x \notin \mathcal{V}ar(pattern(\mathcal{I}_1)). \end{cases}$$

Standard and incremental definitional trees are related by means of a function ρ which maps standard definitional trees to incremental definitional trees:

$$\rho(\mathcal{T}) = \begin{cases} irule(l = r) & \text{if } \mathcal{T} = rule(l = r); \\ ibranch(o, (c_1, \rho(\mathcal{T}_1)), \ldots, (c_n, \rho(\mathcal{T}_n))) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n), \text{ and} \\ & \quad pattern(\mathcal{T}_i) = \pi[c_i(\overline{x_k})]_o, \text{ for } 1 \leq i \leq n \end{cases}$$

Example 3.2 *The definitional tree of Example 3.1 is represented as follows:*

$ibranch$(1, (`Z`, $irule$(`first Z y = []`)),
(`S`, $ibranch$(2, (`:`, $irule$(`first (S n) (m:ms) = m:(first n ms)`)))))

Figure 2 shows the corresponding graphical representation. The main differences w.r.t. Figure 1 are: a) patterns have been removed from branch nodes, b) a dashed box encloses the inductive position which labels branch nodes, and c) we label arcs with the corresponding constructor symbol.
Note that it is immediate to redefine λ in terms of incremental definitional trees, although we don't include this definition in the paper.

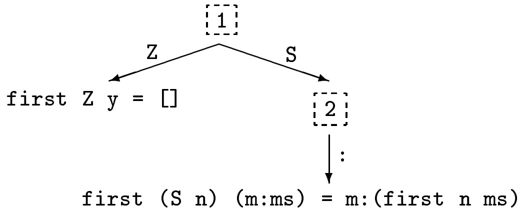

Figure 2: Incremental definitional tree for the function `first`

3.2 Matching dags, index trees, definitional trees, and incremental definitional trees

It has recently been proved that inductively sequential TRSs and Huet and Lévy's strongly sequential CB-TRSs coincide [HLM98]. In [HLM98], Hanus et al. also discuss the precise relation between definitional trees and Strandh's *index trees* [Str89], a data structure which was proved equivalent by Durand [Dur94] to Huet and Lévy's *matching dags* (i.e., the original data structure which serves to implement strongly needed reductions in strongly sequential TRSs [HL79, HL91]).

Roughly speaking, an index tree for an orthogonal TRS $\mathcal{R}$ is (despite its name) a directed acyclic graph whose inner nodes are *index points*, i.e., pairs $\langle \pi, p \rangle$, where π is a strict prefix of a lhs of $\mathcal{R}$ (with the variables replaced by a special symbol Ω) and p is a strong index of π, which satisfy some suitable conditions (see [Dur94]). There is a special root node $\langle \Omega, \Lambda \rangle$. Given a node $\langle \pi, p \rangle$ and a function symbol f, there is an (ordinary) *arc* from $\langle \pi, p \rangle$ to $\langle \pi[f(\overline{\Omega})]_p, q \rangle$ if either $\langle \pi[f(\overline{\Omega})]_p, q \rangle$ is an index point or $\pi[f(\overline{\Omega})]_p$ is a lhs of a rule whose variables have been replaced by Ω's (in the latter case, we discard q and just write $\pi[f(\overline{\Omega})]_p$ to denote this node at the end of an arc). There are also *failure arcs*, which are used to proceed after a failing partial matching without being compelled to start from scratch. Orthogonal TRSs such that a corresponding index tree exists are called *bounded* TRSs. Durand proved that the class of bounded TRSs and strongly sequential TRSs coincide [Dur94]. The proof is based on the existence of a precise and immediate correspondence between Huet and Lévy's matching dags and Strandh's index trees.

According to the definition of index trees, some nodes might *not* be reachable from the initial node $\langle \Omega, \Lambda \rangle$ via ordinary arcs (failure arcs can be necessary to access them). Strandh defined the *forward branching* index trees, which are those index trees such that all nodes can be reached from the root node through an ordinary path [Str89], and introduced the class of forward branching TRSs (FB) to contain those TRSs which admit a forward branching index tree. Strandh proved (actually, that was the main aim of [Str89]) that forward branching index trees are suitable for implementing incremental strongly needed reductions and provided an algorithm which does perform incrementally. Later on, and following a different line of work, Toyama et al. investigated the existence of the *transitive* strongly needed redexes, which can be reduced incrementally also [TSvEP93]. They did not provide any particular data structure (comparable to forward branching index trees) to ease the search for the redexes but only defined the transitive TRSs (TR) as those which grant the transitive strongly needed reductions. Durand showed that FB = TR [Dur94].

Huet and Lévy had previously defined a particular class of strongly sequential TRSs, namely the *simple* systems [HL79, HL91], with a very simple associated matching dag; actually, they showed that the corresponding matching dag is *almost* a tree (that is, the ordinary arcs and the corresponding nodes make up a tree, see [HL91], pages $439-440$). Nevertheless, they did say nothing about the possibility of performing incremental reductions with simple systems; Huet and Lévy only noticed that there is a very simple algorithm for supplying a simple system with a matching dag. Durand and Salinier [DS93] demonstrated that simple systems are forward branching and, correspondingly, the matching dag which is associated to a simple system exactly coincides with the forward branching index tree which would be constructed using the algorithm in [Dur94]. As a consequence, it is possible to perform incremental reductions in simple systems.

By glueing together all these facts, and since strongly sequential CB-TRSs (hence inductively sequential ones) are simple systems [HL79, HL91], we get that the forward branching index tree which corresponds to an inductively sequential TRS is 'almost' a tree. Actually, we can say that it *is* a tree since, as a benefit of the constructor discipline, failing arcs are *not* present in the index trees of inductively sequential TRSs (see Definition 2.21 of [Dur94] for more details). Now, by using the connection between index trees and definitional trees given in [HLM98] we obtain, from the forward branching index tree associated to an inductively sequential system, a forest of definitional trees which consists of the immediate subtrees stemming from the special node $\langle \Omega, \Lambda \rangle$. On the other hand, the nonexistence of failing arcs (in CB-TRSs) allows us a further refinement in the representation, which immediately derives in our incremental definitional trees. This redeems definitional trees from unnecessarily mimicking the structure of index trees when we consider CB-TRSs. Nevertheless, for arbitrary (possible non-constructor) simple systems, the need to record failing arcs prevents one from considering an index tree as a simple collection of definitional trees.

On the other hand, Huet and Lévy and Strandh only considered rewriting computations, wheras our incremental trees do not only ease pattern matching but also support the variable instantiation in input terms which is necessary for the narrowing steps. In the following section, we give an incremental definition of needed narrowing which can be thought of as the functional logic counterpart of Strandh's algorithm for the incremental reduction of terms in forward branching TRSs [Str89].

4 Incremental evaluation

Consider a needed narrowing sequence $e_1 \leadsto e_2 \leadsto \cdots \leadsto e_n$. Since each step $e_i \leadsto_{(p_i, R_i, \sigma_i)} e_{i+1}$ is scheduled by $\lambda(e_i, \mathcal{T}_i)$ (where $\mathcal{T}_i$ is a definitional tree for $root(e_i)$), and it usually changes only small parts of e_i, it is obvious that, if $p_i \in$

$\mathcal{P}os(e_i) - \{\Lambda\}$, then the needed narrowing step for e_{i+1} (i.e., the computation of $\lambda(e_{i+1}, \mathcal{T}_{i+1})$) might *wastefully* examine some of the symbols already considered in the preceding step and take almost the same decisions. In order to avoid this wasting, we look for an *incremental* definition of λ which is able to locally resume the search from $e_{i+1}|_{q_i}$ for some $q_i \leq p_i$. Intuitively, whenever $(p, l \to r, \sigma) \in \lambda(t, \mathcal{T})$ and $t \leadsto_{(p,l\to r,\sigma)} s$, we can be sure that the symbols occurring at the position immediately above p have not been modified. On the other hand, the term $s|_p$ may or may not be operation-rooted. If $root(s|_p) \in \mathcal{F}$, then $s|_p$ should be further narrowed until a constructor-rooted term or a variable is obtained. If $root(s|_p) \in \mathcal{C} \cup \mathcal{X}$, then the next needed narrowing step is determined by the definitional tree $\mathcal{T}$ of $root(s|_q)$, where $q < p$ is the position of the defined symbol occurring at the position immediately above p.

Proposition 4.1 *Let t be a term, such that $t \leadsto_{(p,R,\sigma)} s$ using $(p, R, \sigma) \in \lambda(t, \mathcal{T})$, where $\mathcal{T}$ is a definitional tree such that $pattern(\mathcal{T}) \leq t$. Let $\mathcal{T}'$ be a definitional tree such that $pattern(\mathcal{T}') \leq s$ and $q = max(\{q' \in \mathcal{FP}os(s) \mid q' \leq p\})$. If $(p', R', \sigma') \in \lambda(s, \mathcal{T}')$, then $p' \geq q$.*

Proof. Let us first consider the case when $q \neq p$ (note that, since we impose $pattern(\mathcal{T}') \leq s$, it follows that $\{q' \in \mathcal{FP}os(s) \mid q' \leq p\}$ is not empty; i.e., q does exist.). Then, the computation of $\lambda(t, \mathcal{T})$ has reached a branch node $branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ and there is an operation-rooted subterm $t|_{q.o}$ at the inductive position o. Thus the computation of $\lambda(s, \mathcal{T}')$ first reaches the same branch node and then proceeds by checking whether $s|_{q.o}$ is a constructor-rooted term or a variable, since it cannot be an operation-rooted term unless $q = p$, which contradicts the original assumption. In both cases, a subtree $\mathcal{T}_i$ for $1 \leq i \leq n$ is selected before resuming the computation. If $\mathcal{T}_i$ is a branch subtree, then the computed position p' will be below q; otherwise (that is, $\mathcal{T}_i$ is a rule subtree), $p' = q$.

Now consider the case when $q = p$. Then $s|_p$ is operation-rooted and this position needs to be further narrowed in order to compute a constructor-rooted term. Then, the computed position is trivially q or below q. $\square$

Corollary 4.2 *Let t be a term, such that $t \leadsto_{(p,R,\sigma)} s$ using $(p, R, \sigma) \in \lambda(t, \mathcal{T})$, where $\mathcal{T}$ is a definitional tree such that $pattern(\mathcal{T}) \leq t$. Let $\mathcal{T}'$ be a definitional tree such that $pattern(\mathcal{T}') \leq s$. If $s|_p$ is operation-rooted and $(p', R', \sigma') \in \lambda(s, \mathcal{T}')$, then $p' \geq p$.*

Since each step $e_i \leadsto_{(p_i,R_i,\sigma_i)} e_{i+1}$ in a needed narrowing sequence $e_1 \leadsto e_2 \leadsto \cdots \leadsto e_n$ is performed using $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$, where $\mathcal{T}_i$ is a definitional tree for $root(e_i)$, Corollary 4.2 ensures that, whenever $e_{i+1}|_{p_i}$ is operation-rooted, then $p_{i+1} \geq p_i$. Thus, we take $(p_{i+1}, R_{i+1}, \sigma_{i+1}) = (p_i.p, R, \sigma)$, where $(p, R, \sigma) \in \lambda(e_{i+1}|_{p_i}, \mathcal{T})$ and $\mathcal{T}$ is a definitional tree for $root(e_{i+1}|_{p_i})$. If $e_{i+1}|_{p_i}$ is constructor-rooted or a variable, then it is necessary to resume the evaluation somewhere above p_i. Proposition 4.1 allows us to come back to $e_{i+1}|_q$ by taking the ('most defined') definitional subtree $\mathcal{T}_i'$ of the definitional tree of $root(e_i|_q)$ which has been used in the computation of $\lambda(e_i, \mathcal{T}_i)$ immediately before obtaining $\mathcal{T}_i$. Thus, we let $(p_{i+1}, R_{i+1}, \sigma_{i+1}) = (q.p, R, \sigma)$, where $(p, R, \sigma) \in \lambda(e_{i+1}|_q, \mathcal{T}_i')$.

Thus, we define a data structure $\mathcal{L}$ which contains the information needed to perform the needed narrowing steps incrementally. Given a term t, we let $\mathcal{L}$ denote a list of pairs $(p, \mathcal{I})$ where p is a position and $\mathcal{I}$ is an incremental definitional tree. The *incremental needed narrowing strategy* is denoted by $\lambda^\iota(t, \mathcal{L})$. It takes as arguments a term t and an auxiliary list $\mathcal{L}$ which plays the role of the definitional tree $\mathcal{T}$ in the original strategy $\lambda(t, \mathcal{T})$. Given an operation-rooted term t and a list $\mathcal{L} = [(q, \mathcal{I}), \ldots]$ with $pattern(\mathcal{I}) \leq t|_q$, we compute $(p, R, \sigma, \mathcal{L}') \in \lambda^\iota(t, \mathcal{L})$, where p is the position pointing to the subterm of t to be narrowed, R is the rule to be applied, σ is the computed substitution, and $\mathcal{L}'$ is a list containing the information which allows us to proceed with the subsequent evaluation steps incrementally. Formally,

$$\lambda^\iota(t, [(q,\mathcal{I})|\mathcal{L}]) \ni \begin{cases} (q, l \to r, id, \mathcal{L}') & \text{if } \mathcal{I} = irule(l = r) \text{ and} \\ & \quad \mathcal{L}' = \begin{cases} [(q, \mathcal{I}')|\mathcal{L}] & \text{if } \theta(r) \text{ is operation-rooted} \\ \mathcal{L} & \text{otherwise} \end{cases} \\ & \quad \text{where } t|_q = \theta(l) \text{ and } \mathcal{I}' \text{ is an incremental} \\ & \qquad\qquad \text{definitional tree for } root(\theta(r)) \\ (p, l \to r, \sigma \circ \tau, \mathcal{L}') & \text{if } \mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n)), \\ & \quad t|_{q.o} = x \in \mathcal{X},\ \tau = \{x \mapsto c_i(\overline{x_k})\}, \\ & \quad \text{and } (p, l \to r, \sigma, \mathcal{L}') \in \lambda^\iota(\tau(t), [(q, \mathcal{I}_i)|\mathcal{L}]); \\ (p, l \to r, \sigma, \mathcal{L}') & \text{if } \mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n)), \\ & \quad t|_{q.o} = c_i(\overline{t_k}), \\ & \quad \text{and } (p, l \to r, \sigma, \mathcal{L}') \in \lambda^\iota(t, [(q, \mathcal{I}_i)|\mathcal{L}]); \\ (p, l \to r, \sigma, \mathcal{L}') & \text{if } \mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n)), \\ & \quad t|_{q.o} = f(\overline{t_k}) \text{ for } f \in \mathcal{F}, \\ & \quad \mathcal{I}' \text{ is an incremental definitional tree for } f, \\ & \quad \text{and } (p, l \to r, \sigma, \mathcal{L}') \in \lambda^\iota(t, [(q.o, \mathcal{I}'), (q, \mathcal{I})|\mathcal{L}]); \end{cases}$$

The following notations are helpful. Given a position p and a list $\mathcal{L}$ of pairs position/incremental definitional tree, we define:

$$p.\mathcal{L} = \begin{cases} [\,] & \text{if} \quad \mathcal{L} = [\,] \\ [(p.q, \mathcal{I}) \mid p.\mathcal{L}'] & \text{if} \quad \mathcal{L} = [(q, \mathcal{I}) \mid \mathcal{L}'] \end{cases}$$

We also use the standard operator '++' for concatenating lists. The following auxiliary result is used below.

Lemma 4.3 *Let t be a term and $\mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n))$ be such that $pattern(\mathcal{I}) \leq t$ and $t|_o$ is operation-rooted. Let $\mathcal{I}'$ be an incremental definitional tree such that $pattern(\mathcal{I}') \leq t|_o$. Then,*

$$\begin{aligned} \lambda^\iota(t, [(\Lambda, \mathcal{I})]) = \{(o.p, l \to r, \sigma, (o.\mathcal{L}) \mathbin{++} [(\Lambda, \mathcal{I})]) \mid \\ (p, l \to r, \sigma, \mathcal{L}) \in \lambda^\iota(t|_o, [(\Lambda, \mathcal{I}')])\}. \end{aligned}$$

Proof. Immediate, by definition of λ^ι. $\square$

The following proposition states that λ^ι is able to compute the same needed narrowing step than λ starting from a given term and an initial list of incremental information.

Proposition 4.4 *Let t be a term and $\mathcal{T}$ be a definitional tree such that $pattern(\mathcal{T}) \leq t$. Then, $(p, l{\to}r, \sigma) \in \lambda(t, \mathcal{T})$ if and only if $\exists \mathcal{L}.(p, l{\to}r, \sigma, \mathcal{L}) \in \lambda^\iota(t, [(\Lambda, \rho(\mathcal{T}))])$.*

Proof. Immediately from the definition of λ and λ^ι. $\square$

Now, we have to ensure that the defined function symbols occurring at positions which are above (or at) a previously narrowed position are obtained by the strategy λ^ι and thus considered for the subsequent narrowing steps.

Proposition 4.5 (Completeness) *Let t be a term and $\mathcal{T}$ be a definitional tree such that $pattern(\mathcal{T}) \leq t$. Let $t \leadsto_{(p, l\to r, \sigma)} s$ using $(p, l{\to}r, \sigma) \in \lambda(t, \mathcal{T})$. Let $(p, l{\to}r, \sigma, \mathcal{L}) \in \lambda^\iota(t, [(\Lambda, \rho(\mathcal{T}))])$, $F_s = \{q \in \mathcal{FP}os(s) \mid q \leq p\}$. Then, $q \in F_s$ if and only if $\exists \mathcal{I}.(q, \mathcal{I}) \in \mathcal{L}$.*

Proof. By noetherian induction on pairs $(t, \mathcal{T})$ (strictly) ordered by $(t, \mathcal{T}) < (t', \mathcal{T}')$ iff $n_\mathcal{F}(t) < n_\mathcal{F}(t')$ or $n_\mathcal{F}(t) = n_\mathcal{F}(t')$ and $\mathcal{T} < \mathcal{T}'$, i.e., $\mathcal{T}$ is a subtree of $\mathcal{T}'$ [AEH94]. Here, $n_\mathcal{F}(t)$ is the number of defined function symbols in t.

- (Base) Since $pattern(\mathcal{T}) \leq t$, then t is operation-rooted. Hence, the base case consists in considering those pairs $(t, \mathcal{T})$ such that t is operation-rooted and does not contain any other defined function symbol, and $\mathcal{T}$ is a rule node $\mathcal{T} = rule(l = r)$. By hypothesis, $l \leq t$, hence let $t = \theta(l)$. Thus, only the first case for λ applies, i.e., $t \leadsto_{(\Lambda, l\to r, id)} s$ and we have either

  1. $s = \theta(r)$ is not operation-rooted, hence $F_s = \emptyset$. Then, from the definition of λ^ι, we have that $\mathcal{L} = [\,]$.
  2. $\theta(r)$ is operation-rooted, hence $F_s = \{\Lambda\}$. Then, from the definition of λ^ι, $\mathcal{L} = [(\Lambda, \mathcal{I})]$, where $\mathcal{I}$ is an incremental definitional tree for $root(\theta(r))$.

  On the other hand, if $\mathcal{L} = [\,]$, then from the definition of λ^ι, we have that $s = \theta(r)$ is not operation-rooted and hence $F_s = \emptyset$. If $\mathcal{L}$ is not empty, then $\mathcal{L} = [(\Lambda, \mathcal{I})]$, and thus $s = \theta(r)$ is operation-rooted, i.e., $F_s = \{\Lambda\}$.

- (Induction) We consider the different cases for λ separately.

  1. If $\mathcal{T}$ is a rule node, we reason as in the base case.
  2. If $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ and $t|_o$ is a variable x, then $\lambda(t, \mathcal{T}) = (p, l{\to}r, \sigma \circ \tau)$, where $(p, l{\to}r, \sigma) \in \lambda(\tau(t), \mathcal{T}_i)$ for some $1 \leq i \leq n$ and $\tau = \{x \mapsto c_i(\overline{x_k})\}$. Thus, $\tau(t) \leadsto_{(p, l\to r, \sigma)} s$. Since $n_\mathcal{F}(\tau(t)) = n_\mathcal{F}(t)$ and $\mathcal{T}_i < \mathcal{T}$, by induction hypothesis we have that, for $(p, l{\to}r, \sigma, \mathcal{L}) \in \lambda^\iota(\tau(t), [(\Lambda, \rho(\mathcal{T}_i))])$, $q' \in F_s$ iff $\exists \mathcal{I}.(q', \mathcal{I}) \in \mathcal{L}$. From the definition of λ^ι, the conclusion follows.

3. If $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ and $t|_o$ is constructor-rooted, the conclusion follows by a similar reasoning to case 2.
4. If $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ and $t|_o$ is operation-rooted, then $(p', l \to r, \sigma) \in \lambda(t|_o, \mathcal{T}')$ (where $\mathcal{T}'$ is a definitional tree for $root(t|_o)$ and $p = o.p'$), i.e., $t|_o \leadsto_{(p', l \to r, \sigma)} s'$. Since $n_{\mathcal{F}}(t|_o) < n_{\mathcal{F}}(t)$, by the induction hypothesis $q' \in F_{s'}$ if and only if $\exists \mathcal{I}.(q', \mathcal{I}) \in \mathcal{L}'$, for $(p', l \to r, \sigma, \mathcal{L}') \in \lambda^\iota(t|_o, [(\Lambda, \rho(\mathcal{T}'))])$. Since $F_s = \{\Lambda\} \cup o.F_{s'}$, by Lemma 4.3 the conclusion follows.

□

The following theorem shows that the needed narrowing steps performed by the incremental strategy λ^ι and the computation steps carried out by the original needed narrowing strategy λ are equivalent.

Theorem 4.6 (Correctness) *Let $\mathcal{R}$ be an inductively sequential program, $e_1 \leadsto e_2 \leadsto \cdots \leadsto e_n$ be a needed narrowing derivation where $\mathcal{T}_i$ is a definitional tree such that $pattern(\mathcal{T}_i) \leq e_i$ for all $1 \leq i \leq n$, and $\mathcal{L}_0 = [(\Lambda, \rho(\mathcal{T}_1))]$. Then, for all $1 \leq i \leq n$, $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$ if and only if $(p_i, R_i, \sigma_i, \mathcal{L}_i) \in \lambda^\iota(e_i, \mathcal{L}_{i-1})$.*

Proof. We prove it by induction on the length of the derivation.

- $i = 1$. This case is trivial, since by Proposition 4.4 it amounts to a call to $\lambda^\iota(e, \mathcal{L}_0)$.
- $i > 1$. Let $\mathcal{L}_{i-1} = [(q, \rho(\mathcal{T}))|\mathcal{L}']$ for some definitional tree $\mathcal{T}$. By the induction hypothesis, we have that $(p_{i-1}, R_{i-1}, \sigma_{i-1}) \in \lambda(e_{i-1}, \mathcal{T}_{i-1})$ if and only if $(p_{i-1}, R_{i-1}, \sigma_{i-1}, \mathcal{L}_{i-1}) \in \lambda^\iota(e_{i-1}, \mathcal{L}_{i-2})$. Since $e_{i-1} \leadsto_{(p_{i-1}, R_{i-1}, \sigma_{i-1})} e_i$, by Proposition 4.5, q points to a defined symbol of e_i which is above or at the position p_{i-1}. By the ordering between the positions recorded in $\mathcal{L}_{i-1}$ (see Lemma 4.3), q is the position of the defined symbol immediately above or at the position p_{i-1}. By Proposition 4.1, $(p', R_i, \sigma_i) \in \lambda(e_i|_q, \mathcal{T}) \Leftrightarrow (q.p', R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$, where $q.p' = p_i$. Then, by Proposition 4.4, $(p', R_i, \sigma_i) \in \lambda(e_i|_q, \mathcal{T}) \Leftrightarrow (p', R_i, \sigma_i, \mathcal{L}'_i) \in \lambda^\iota(e_i|_q, [(\Lambda, \rho(\mathcal{T}))])$. Now, since all positions in the first component of the position/incremental definitional tree lists correspond to operation-rooted subterms, by using Lemma 4.3 it is immediate that $(q.p', R_i, \sigma_i, \mathcal{L}_i) \in \lambda^\iota(e_i, [(q, \rho(\mathcal{T}))|\mathcal{L}']) \Leftrightarrow (p', R_i, \sigma_i, \mathcal{L}'_i) \in \lambda^\iota(e_i|_q, [(\Lambda, \rho(\mathcal{T}))])$ if $(q', \mathcal{I}') \in \mathcal{L}'_i \Rightarrow (q.q', \mathcal{I}') \in \mathcal{L}_i$.

  Thus, $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i) \Leftrightarrow (p_i, R_i, \sigma_i, \mathcal{L}_i) \in \lambda^\iota(e_i, [(q, \rho(\mathcal{T}))|\mathcal{L}'])$.

□

Although we only provide in this paper the definition of λ^ι in terms of incremental definitional trees, the function λ^ι can be easily redefined to perform with standard definitional trees, similarly to λ. We make free with this in the following section, where different implementations of needed narrowing are compared.

Benchmark	Original	Incremental def. trees	Incremental evaluation	Both optimizations	Global Speed-up
`iter`	3345	1145	395	344	9.72
`ackermann`	2097	1112	834	653	3.20
`mergesort`	556	466	340	323	1.72
`quicksort`	4137	3114	2668	2658	1.55
`fibonacci`	869	717	678	622	1.40
`horseman`	3342	3180	2762	2751	1.21
`last`	1025	1008	983	968	1.05

Table 1: Comparison between UPV-Curry optimizations (in ms.)

5 Experimental results

Curry [HAK+99] is a functional logic programming language based on needed narrowing which combines the best ideas of declarative languages such as Haskell [HPW92] and SML [MTH90] (functional languages), Gödel [HL94] and λProlog [NM88] (logic languages), and ALF [HS91], Babel [KA96] and $\mathcal{TOY}$ [LS99] (functional logic languages). UPV-Curry is a new interpreter of Curry which provides an almost complete implementation[2] of the language according to [HAK+99] (it lacks Curry modules and encapsulated search). The system is written in SICStus Prolog v3.6 and consists of about 250 clauses (3000 lines of code). The parser is expressed by 95 clauses, the typechecker module by 55 clauses, the algorithm which builds definitional trees by 20 clauses, the execution module by 45 clauses, and the interface and some other utilities by 35 clauses. See [EAL98] for a complete description.

We have performed an empirical comparison of the original needed narrowing strategy and the optimized definition when used in the implementation of Curry (see Table 1). We first consider a basic UPV-Curry implementation which mechanizes the standard λ function, and the original definitional trees. A second UPV-Curry implementation includes the refinement due to the incremental definitional trees. The third UPV-Curry implementation includes the incremental strategy λ^ι, but it considers the original definitional trees. We finally consider a completely refined implementation, including all optimizations. Times were measured on a SUN SparcStation, running under UNIX System V Release 4.0. They are expressed in milliseconds and are the average of 10 executions. The benchmarks used for the analysis are given in Appendix; most of them are standard Curry test programs[3]. The functions `ackermann`, `fibonacci`, `last`, `quicksort` and `mergesort` are the standard ones; `horseman` computes, by means of constraints, the number of men and horses that have a certain number of heads and feet; finally, `iter` produces a sequence of n nested calls to a given function and is used to set the potentiality of our method off.

Runtime input goals were chosen to give a reasonably long overall time, and

[2] UPV-Curry is publicly available from `http://www.dsic.upv.es/users/elp/soft.html`.

[3] Look at the URL `http://www-i2.informatik.rwth-aachen.de/~hanus/curry/examples/` for a list of available Curry test programs.

Benchmark	Goal	UPV-Curry	TasteCurry	PACS
`iter`	`iter 100 sub1 100`	344	1174	909
`ackermann`	`ackermann 20`	653	2505	1203
`mergesort`	`sort intMerge [3,1,2] xs`	323	870	442
`quicksort`	`qsort` L	2658	1829	1511
`fibonacci`	`fibonacci 10`	622	2262	1106
`horseman`	`horseman x y 8 20`	2751	3542	1658
`last`	`last [1]`$_{100}$	977	58736	4256

Table 2: Comparison with other Curry interpreters (in ms.)

Benchmark	Saving	Benchmark	Saving
`ackermann`	43%	`mergesort`	60%
`fibonacci`	25%	`iter`	65%
`horseman`	33%	`quicksort`	0%
`last`	33%		

Table 3: Spatial savings

are shown in Table 2. Natural numbers are implemented using Z/S-terms, and lists are shown within goals by using a subindex which represents its size. L is `[10,9,8,7,6,5,4,3,2,1,0]` in the goal for `quicksort`.

The figures in Table 1 reveal that the proposed refinements are actually (and independently) effective. The goals which require much computation, such as `ackermann`, `fibonacci`, `quicksort`, and `mergesort` get a substained speed-up when incrementality is used (e.g. 3.20 for the `ackermann` example). Indeed, the `iter` benchmark highlights the benefits which are brought by the incremental evaluation when the goal contains some nested calls and the innermost one must be evaluated first. On the other hand, even in the cases when no significant speed-up is obtained (this is the case of `last`), execution is not punished with any appreciable overhead. In other words, the use of incremental definitional trees causes no overload, which substantiate the idea that they are simply an apt shortening of the standard definitional trees. On the other hand, the incremental evaluation leads on its own to considerable speed-ups in program execution while still guaranteeing no appreciable slowdown.

Table 2 compares the fully optimized UPV-Curry implementation and other Curry interpreters. Namely, the figures in Table 2 correspond to the runtimes of the benchmarks for UPV-Curry, TasteCurry, and PACS-TasteCurry interpreters[4]. These results show that the incremental UPV-Curry implementation performs very well in comparison to the other Curry interpreters. In overall, it takes about 80% the time needed by PACS-TasteCurry and about 38% the time needed by TasteCurry to evaluate the queries. These results seem to substantiate the advantages of using the proposed incremental techniques.

Another interesting aspect for the comparison concerns the spatial savings

[4] TasteCurry and PACS-TasteCurry are available from `http://www-i2.informatik.rwth-aachen.de/~hanus/curry/`.

due to the use of incremental definitional trees. For the comparison, we strictly consider the information which is necessary for defining the needed narrowing strategy. Table 3 shows the ratio between the number of symbols of inner nodes of standard definitional trees and incremental definitional trees, respectively. The results of Table 3 clearly evidence that incremental definitional trees can also bring substantial savings in memory space (up to 65% in these tests).

6 Conclusions

We have introduced *incremental* definitional trees, a data structure which is a refinement of the usual definitional trees and redeems them from unnecessarily sticking to the traditional Huet and Lévy's matching dags [HL79, HL91] or Strandh's index trees [Str89]. We have shown that, in CB systems, such a basic redefinition of definitional trees permits a more compact representation which can be used to substantially improve the evaluation of input expressions.

Inspired in the fact that inductively sequential TRSs are forward branching, we have developed an incremental description of needed narrowing which can be thought of as the functional logic counterpart of Strandh's algorithm for the incremental reduction of terms [Str89]. By this means, we have implicitly demonstrated that (incremental) Curry implementations fit the original (nonincremental) semantics of Curry. We have also proved that our incremental needed narrower preserves the optimality properties and is more efficiently implementable than the original procedure thanks to its intrinsic incremental nature. We have shown that the proposed refinements can significantly improve the execution of functional logic languages based on needed narrowing. In particular, we have shown that they can be advantageously applied to optimize the implementation of the basic Curry machinery.

References

[AEH94] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, 1994.

[AHLV99] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Inductively Sequential Functional Logic Programs. In *Proc. of 1999 International Conference on Functional Programming, ICFP'99*, 1999. To appear.

[Ant92] S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming ALP'92*, pages 143–157. Springer LNCS 632, 1992.

[BCG+89] P.G. Bosco, C. Cecci, E. Giovannetti, C. Moiso, and C. Palamidessi. Using resolution for a sound and efficient integration of logic and functional programming. In J. deBakker, editor, *Languages for parallel architectures.* Wiley (UK), 1989.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.

[DS93] I. Durand and B. Salinier. Constructor Equivalent Term Rewriting Systems. *Information Processing Letters*, 47:131–137, 1993.

[Dur94] I. Durand. Bounded, Strongly Sequential and Forward Branching Term Rewriting Systems. *Journal of Symbolic Computation*, 18:319–352, 1994.

[EAL98] S. Escobar, M. Alpuente, and S. Lucas. UPV-Curry User's Manual. Technical Report DSIC-II/38/98, DSIC, Universidad Politécnica de Valencia, 1998. Available at `http://www.dsic.upv.es/users/elp/papers.html`.

[HAK+99] M. Hanus, S. Antoy, H. Kuchen, F.J. López-Fraguas, and F. Steiner. Curry An Integrated Functional Logic Language (version 0.5). Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, February 1999.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[Han95] M. Hanus. Efficient translation of lazy functional logic programs into prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.

[HKMN95] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[HL79] G. Huet and J.-J. Lévy. Call by need computations in nonambiguous linear term rewriting systems. Technical report, IRIA Laboria, LeChesnay, France, 1979.

[HL91] 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*, pages 396–443. The MIT Press, Cambridge, MA, 1991.

[HL94] P. Hill and J.W. Lloyd. *The Gödel Programming Language*. The MIT Press, Cambridge, MA, 1994.

[HLM98] M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.

[HPW92] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Functional Programming Language Haskell: a non–strict, purely functional language. *Sigplan Notices*, 27(5), 1992.

[HS91] M. Hanus and A. Schwab. The Implementation of the Functional-Logic Language ALF. FB Informatik, Univ. Dortmund, 1991.

[Hul80] J.M. Hullot. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.

[KA96] H. Kuchen and J. Anastasiadis. Higher Order Babel: Language and implementation. In *editors Proc. 5th International Workshop on Extensions of Logic Programming ELP'96*, pages 193–207. Springer LNAI 1050, 1996.

[KLMR90] 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*, pages 298–317. Springer LNCS 463, 1990.

[LLR93] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93, Tallinn (Estonia)*, pages 184–200. Springer LNCS 714, 1993.

[LS99] F.J. López Fraguas and J. Sánchez Hernández. Toy: A multiparadigm Declarative System. In P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications, RTA'99*, volume 1631 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1999.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of ML*. The MIT Press, 1990.

[NM88] G. Nadathur and D. Miller. An Overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Proc. of Fifth Int'l Conf. on Logic Programming ICLP'88*, pages 810–827. The MIT Press, Cambridge, MA, 1988.

[O'D98] M. J. O'Donnell. Equational Logic Programming. In *D. Gabbay, ed., Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5 on Logic Programming, Chapter 2*, pages 69–161. Oxford Science Publications, 1998.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[Red85] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE, New York, 1985.

[Sla74] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[Str89] R. Strandh. Classes of Equational Programs that Compile into Efficient Machine Code. In N. Dershowitz, editor, *Proc. of RTA'89*, volume 355 of *Lecture Notes in Computer Science*, pages 449–461. Springer-Verlag, Berlin, 1989.

[TSvEP93] Y. Toyama, S. Smetsers, M.C.J.D. van Eekelen, and R. Plasmeijer. The functional strategy and transitive term rewriting systems. In *Term Graph Rewriting - Theory and Practice*, pages 117–129. John Wiley & sons, 1993.

A Benchmarks code

ackermann

```
data Nat = Z | S Nat

ackermann n = ack (S (S Z)) n

ack Z n = S n
ack (S m) Z = ack m (S Z)
ack (S m) (S n) = ack m (ack (S m) n)
```

fibonacci

```
data Nat = Z | S Nat

sum Z y = y
sum (S x) y = S (sum x y)

fib Z = S Z
fib (S x) = S x
fib (S (S x)) = sum (fib (S x)) (fib x)
```

iter

```
data Nat = Z | S Nat

iter Z f x = x
iter (S y) f x = f (iter y f x)

sub1 (S Z) = Z
sub1 (S (S Z)) = S Z
sub1 (S (S (S Z))) = S (S Z)
sub1 (S (S (S (S x)))) = S (S (S x))
```

last

```
append eval flex
append [] y = y
append (x:r) y = (x:append r y)

last l | append _ [_x] =:= l = _x
```

horseman

```
data Nat = Z | S Nat

int2nat   ::  Int -> Nat
int2nat n = if n<=Z then Z else S (int2nat (n-1))

add Z     n = n
add (S m) n = S (add m n)

horseman m h heads feet = heads =:= add m h & feet =:= add (add m m) (add (add h h) (add h h))
```

quicksort

```
-- quicksort using higher-order functions:
qsort ::  [Int] -> [Int]
qsort [] = []
qsort (x:l) = qsort (filter (<x) l) ++ x :  qsort (filter (>=x) l)
```

mergesort

```
-- merge sort:  sorting two list by merging the sorted first
-- and second half if the list
firsthalf xs = take (length xs `div` 2) xs

secondhalf xs = drop (length xs `div` 2) xs

sort ::  ([a] -> [a] -> [a] -> Constraint) -> [a] -> [a] -> Constraint
sort merge xs ys =
   if length xs < 2 then ys =:= xs
   else sort merge (firsthalf xs) _us
        & sort merge (secondhalf xs) _vs
        & (merge _us _vs ys)

intMerge ::  [Int] -> [Int] -> [Int] -> Constraint
intMerge [] ys zs = zs =:= ys
intMerge (x:xs) [] zs = zs =:= x:xs
intMerge (x:xs) (y:ys) zs =
   if (x > y) then intMerge (x:xs) ys _us & zs =:= y:_us
              else intMerge xs (y:ys) _vs & zs =:= x:_vs
```