In *Proc. of the 1996 Joint Conference on Declarative Programming* (APPIA-GULP-PRODE'96), pp. 259–270, San Sebastian, 1996

A denotational semantics for needed narrowing

Michael Hanus[*] Salvador Lucas[†]

Abstract

Needed narrowing [2] is currently the best (lazy) narrowing strategy for functional logic programs. In order to automatically improve compilation, it is essential to rely on an adequate semantic framework. The denotational semantics of a programming language is its standard semantics and can be used as a formal basis to improve implementations. In this work we introduce a denotational semantics which is adequate to express needed narrowing.[1]

Keywords: denotational semantics, functional logic languages, lazy narrowing strategies.

1 Introduction

Lazy narrowing strategies in functional logic programming are important in avoiding unnecessary computations and enabling the use of infinite data structures. In order to have a good framework to implement lazy languages and automatically improve compilation, we need to rely on an adequate semantic definition. The denotational semantics of a programming language maps syntactic constructs in the program to the abstract values which they denote [15]. It is often considered as the *standard semantics* of programming languages and can be used as a suitable basis for analyses and implementations.

The denotational definition of a programming language may be more or less close to the operational principle of the language. In the setting of functional logic languages, this is to say that the particular narrowing strategy used in the operational semantics can be more or less reflected in the denotational description of the language. Some denotational approaches to the semantics of functional logic languages can be found in [11, 13]. However, these semantic definitions do not reflect the strategy used for narrowing and, therefore, they are not considered here.

In this paper we introduce a denotational semantics for term rewriting systems which allows us express the needed narrowing strategy. The needed narrowing strategy is considered as a combination of program transformation and specialization of the general narrowing mechanism.

[*]Informatik II, RWTH Aachen, Germany, hanus@informatik.rwth-aachen.de.

[†]DSIC, U.P. de Valencia, Spain, slucas@dsic.upv.es. Work partially supported by Bancaixa (Bancaja-Europa grant) and CICYT (under grant TIC 95-0433-C03-03).

[1]This work is a short version of [6] which we refer to find out the missing details and proofs.

In the semantic description of the computation, we work with finite substitutions in representing the solutions of narrowing evaluations thus borrowing the treatment of [3] from the logic programming setting. The formalization here is slightly different. We define a suitable notion of finite substitution and use them to deal with renaming as well as for the solutions of evaluations.

We give the main properties of the semantic definition and prove the adequacy of this semantics to express needed narrowing.

In Section 2, we briefly recall the technical concepts and results used in the remainder of the paper. In Section 3, we describe the notion of finite substitution that we deal in the semantics. In Section 4, we summarize the notion of needed narrowing. In Section 5, the denotational semantics and its basic properties are given. We also give the adequacy result. Finally, Section 6 points to conclusions and future work.

2 Preliminaries

We introduce the most important notations used in the paper. For full definitions we refer to [4].

The set of *terms* $\mathcal{T}(\Sigma, V)$ is constructed w.r.t. a given many-sorted *signature* Σ and variables from V. We denote by $ar(f)$ the number of argument positions of a symbol $f \in \Sigma$. We write $\mathcal{V}ar(t)$ for the set of variables occurring in a term t, and $\overline{t_n}$ for a tuple $t_1, \ldots, t_n$ of terms (where we sometimes omit the subscript n). Functional logic programs are generally *constructor-based* [5], i.e., the signature is the disjoint union $\Sigma = \mathcal{C} \uplus \mathcal{F}$ of two classes of symbols: *constructors* $c \in \mathcal{C}$ that construct data terms, and *defined functions* or *operations* $f \in \mathcal{F}$ that operate on data terms. We denote by HNF_Σ the set of terms in $\mathcal{T}(\Sigma, V)$ which are in *head normal form* (*hnf*), i.e., terms where the root symbol is not a defined function.

Terms are viewed as labelled trees in the usual way. An *occurrence* or *position* p is a path identifying a subterm in a term. $t|_p$ denotes the subterm of t at position p, and $t[s]_p$ denotes the result of *replacing* $t|_p$ with s in t.

A *term rewriting system* (*TRS*) is a pair $\mathcal{R} = (\Sigma, R)$ where R is a set of rewrite rules. Since functional logic programs are constructor-based, we assume that a *program* $\mathcal{R}$ is a *constructor-based term rewriting system* consisting of *rewrite rules* $l \to r$, where l is a *pattern*, i.e., the root of l is an operation and the arguments of l do not contain any operation symbols.

A substitution is a mapping $\sigma : V \to \mathcal{T}(\Sigma, V)$ which is the identity mapping at all but finitely many points. We denote by ϵ the identity substitution. We write substitutions in the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Terms t, s unify if there exists a substitution σ such that $\sigma(t) = \sigma(s)$. In this case, σ is called a unifier of t and s. If W is a set of variables, we denote by $\sigma =_W \sigma'$ that $\sigma(x) = \sigma'(x)$ for all $x \in W$. A term t is a *variant* of s if it is obtained from s by a unique replacement of variables by new variables.

Functional logic programs compute with partial information, i.e., a functional expression may contain logical variables. The goal is to compute values for these variables such that the expression is evaluable to a particular normal form, e.g., a constructor term [11]. This is done by narrowing. A term t is *narrowable* to a term s if there exist a non-variable position p in t (i.e., $t|_p$ is not a variable), a variant $\alpha : l \to r$ of a rewrite rule in R with $\mathcal{V}ar(t) \cap \mathcal{V}ar(l \to r) = \emptyset$ and a unifier σ of $t|_p$ and l such that $s = \sigma(t[r]_p)$. In this case we write $t \rightsquigarrow_{p,\alpha,\sigma} s$. If σ is a most general unifier of $t|_p$ and l, the narrowing step is called *most general.*[2] We write $t_0 \rightsquigarrow^*_\sigma t_n$ if there is a narrowing derivation $t_0 \rightsquigarrow_{p_1,\alpha_1,\sigma_1} t_1 \rightsquigarrow_{p_2,\alpha_2,\sigma_2} \cdots \rightsquigarrow_{p_n,\alpha_n,\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_2 \circ \sigma_1$. The renamings in the i-th narrowing step must be also performed apart of all variables appearing in the previous narrowing steps $j < i$.

Narrowing solves equations, i.e., computes values for the variables in an equation such that the equation becomes true, where an *equation* is a pair $t \approx t'$ of terms of the same sort. Since we do not require terminating term rewriting systems, normal forms may not exist. Hence, we define the validity of an equation as a *strict equality* on terms in the spirit of functional logic languages with a lazy operational semantics such as *BABEL* [11]. Thus, a substitution σ is a *solution* for an equation $t \approx t'$ iff $\sigma(t)$ and $\sigma(t')$ are reducible to a same ground constructor term. Equations can also be interpreted as terms by defining the symbol $\approx$ as a binary operation symbol, more precisely, one operation symbol for each sort. Therefore, all notions for terms, such as substitution, narrowing etc., will also be used for equations. The semantics of $\approx$ is defined by the following rules, where $\wedge$ is assumed to be a right-associative infix symbol, and c is a constructor of arity 0 in the first rule and arity $k > 0$ in the second rule.

$$\begin{array}{rcl} c \approx c & \to & \mathtt{true} \\ c(x_1,\ldots,x_k) \approx c(y_1,\ldots,y_k) & \to & x_1 \approx y_1 \wedge \cdots \wedge x_k \approx y_k \\ \mathtt{true} \wedge x & \to & x \end{array}$$

These are the *equality rules* of a signature. By adding the equality rules to the rewrite system, equation solving can be done by narrowing equations to "`true`" [2, 11].

3 Finite substitutions

3.1 Formalization of finite substitutions

In this section we formalize the notion of finite substitution which we need in this work. The definition is very close to the one in [3]. Nevertheless, some differences arise in the detail and meaning of involved operations.

Definition 3.1 (Finite substitution) *Let V be an infinite set of variables and $D \subset V$ a finite subset of variables. A* finite substitution *(f.s.) θ is a mapping $\theta : D \to \mathcal{T}(\Sigma, V)$. The set of all finite substitutions is denoted as $FSubst$. Some basic functions in $FSubst \to V$ are used to describe the finite substitutions.*

$$\mathcal{D}om(\theta) = D \qquad \mathcal{R}ng(\theta) = \cup_{x \in \mathcal{D}om(\theta)} \mathcal{V}ar(\theta(x)) \qquad \mathcal{V}ar(\theta) = \mathcal{D}om(\theta) \cup \mathcal{R}ng(\theta)$$
$$\mathcal{I}de(\theta) = \{x \in \mathcal{D}om(\theta) \mid \theta(x) = x\} \qquad \mathcal{DOM}(\theta) = \mathcal{D}om(\theta) \backslash \mathcal{I}de(\theta)$$

[2]Narrowing is often identified with most general narrowing. However, it is shown in [2] that dropping the requirement for most general narrowing steps is crucial for optimal evaluation strategies.

A f.s. θ such that $\mathcal{I}de(\theta) = \emptyset$ is called a strict *finite substitution (s.f.s.).*

In [3] is called $\mathcal{D}om(\theta)$ what we refer as $\mathcal{V}ar(\theta)$. In [3] $\mathcal{D}om(\theta)$ is actually the domain of the substitution, but it is (implicitly) also the range and contains (necessarily, to be consistent with the definition) identity bindings for some variables. Therefore, implicit assumptions on the identity bindings in the $f.s.$ are taken. We feel that this does not aid the formal treatment of the $f.s.$ Our definition is based on the standard intuitive notion of (partial) function, where domain and range are not necessarily overlapping.

We apply a finite substitution θ (which is not defined for some variables) to any term $t \in \mathcal{T}(\Sigma, V)$ by means of the lifting $^{-} : FSubst \rightarrow Subst$ which is defined as $\bar{\theta}(x) = \theta(x)$ if $x \in \mathcal{D}om(\theta)$ and $\bar{\theta}(x) = x$ otherwise. Then $\theta(t) = \bar{\theta}(t)$. This is necessary if we use finite substitutions to 'collect' the computed answers in the narrowing process. The following example shows that the composition of finite substitutions based on the composition of *partial* functions is not adequate to express the composition of partial solutions.

Example 3.2 *Let $\phi = \{x' \mapsto 0\}$ and $\theta = \{x \mapsto f(x', y')\}$ be f.s's. Then, if we consider the application of f.s.'s to terms in the standard treatment of partially defined functions, we can not obtain the composite substitution, since $f(x', y')$ is not in $\mathcal{T}(\Sigma, \{x'\})$ which is the (extended to terms) domain of ϕ. Therefore, the binding for x will not be correctly established. With the previous definition of application we would obtain the binding $x \mapsto f(0, y')$, as expected.*

The (infix) restriction operation ($\downarrow$: $FSubst \times \wp(V) \rightarrow FSubst$) combines the *projection* and *extension* operations in [3] as follows.

Definition 3.3 (Restriction of finite substitutions) *Let ϕ, θ be f.s's. and W be a finite set. $\theta\downarrow_W$ is a f.s such that $\mathcal{D}om(\theta\downarrow_W) = W$, $\theta\downarrow_W(x) = \theta(x)$ if $x \in \mathcal{D}om(\theta) \cap W$ and $\theta\downarrow_W(x) = x$ if $x \in W \backslash \mathcal{D}om(\theta)$.*

The Example 3.2 points out that, in spite of the fact that the $f.s.$'s are partially defined functions w.r.t. the set of variables V, it is desirable to have a mechanism to compose two $f.s.$'s. We define the composition operation by lifting $f.s.$'s to the space of substitutions.

Definition 3.4 (Composition of finite substitutions) *Let ϕ, θ be f.s's. Define $\phi \circ \theta = (\bar{\phi} \circ \bar{\theta})\downarrow_{\mathcal{D}om(\phi) \cup \mathcal{D}om(\theta)}$. This amounts to say that $\mathcal{D}om(\phi \circ \theta) = \mathcal{D}om(\phi) \cup \mathcal{D}om(\theta)$.*

3.2 Narrowing and finite substitutions

Managing the renaming in the narrowing process leads to new considerations concerning finite substitutions. By following [3], the renaming process is guided in part by the 'current' computed $f.s.$ The requirement of *joining* domains in the composition of $f.s.$'s (see Definition 3.4) is needed if we use $f.s$'s to propagate the renaming-apart restrictions in the narrowing process [14]. We rename terms apart of the set of variables appearing in a given *f.s.* Therefore, we must distinguish *even* between the *f.s.* θ and $\theta \circ \epsilon$ or $\theta \circ \epsilon'$ provided that ϵ and ϵ' are different identity *f.s.* (i.e. $\mathcal{D}om(\epsilon) \neq \mathcal{D}om(\epsilon')$ and not contained in θ), since rename a term l' out of $\mathcal{V}ar(\theta)$ is not the same that renaming it out of $\mathcal{V}ar(\theta) \cup \mathcal{V}ar(\epsilon)$. Thus, the domains of *f.s.*'s must be carefully managed in the operations concerning them.

The following proposition expresses an important property of partial solutions in a narrowing derivation.

Proposition 3.5 *Let* $t \leadsto^{+}_{\sigma''} t'' \leadsto^{*}_{\sigma'} t'$ *be a narrowing derivation. Then* $\sigma{\downarrow}_{\mathcal{V}ar(t)} = (\sigma' \circ \sigma''){\downarrow}_{\mathcal{V}ar(t)} = (\sigma'{\downarrow}_{\mathcal{V}ar(t'')} \circ \sigma''){\downarrow}_{\mathcal{V}ar(t)}$.

4 Needed narrowing

4.1 Definitional trees

Needed narrowing relies on the concept of definitional trees of [1]. A definitional tree can be used as a representation of the rules defining a given function symbol. $\mathcal{T}$ is a partial definitional tree (*pdt*) iff one of the following cases holds:[3]

$\mathcal{T} = rule(\pi \to r)$ where $\pi \to r$ is a variant of a rule in $\mathcal{R}$.

$\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ where π is a pattern, o is the occurrence of a variable of π, $c_1, \ldots, c_n$ are constructors of the sort of $\pi|_o$ for some $n > 0$, and for all i in $\{1, \ldots, n\}$, $\mathcal{T}_i$ is a *pdt* with pattern $\pi[c_i(x_1, \ldots, x_{ar(c_i)})]_o$ where $x_1, \ldots, x_{ar(c_i)}$ are new variables. Moreover, $\mathcal{T}_i$ must be of finite depth.

A *definitional tree* of a k-ary function f is a *pdt* $\mathcal{T}$ with pattern $f(\overline{x})$, where $\overline{x}$ is an $ar(f)$-tuple of distinct variables. A function f with definitional tree $\mathcal{T}$ is called *inductively sequential* if $\mathcal{T}$ contains all and only the rules defining f in the term rewriting system $\mathcal{R}$ which we call the 'program'.

Example 4.1 *Let us consider the following program:*

```
from(N) → [N|from(s(N))]
first(0,L) → [ ]
first(s(N),[E|L]) → [E|first(N,L)]
```

Then

```
rule(from(N) → [N|from(s(N))])
```

is a definitional tree for the function `from`*, and*

```
branch(first(X,Y),1,
       rule(first(0,Y) → [ ]),
       branch(first(s(N),Y),2,
              rule(first(s(N),[E|L]) → [E|first(N,L)])))
```

is a definitional tree for the function `first`.

[3]We ignore the *exempt* nodes which are in the original definition of [1].

A TRS is called *inductively sequential* if all function symbols are inductively sequential. An inductively sequential TRS can be viewed as a set of definitional trees, each defining a function symbol.

A definitional tree determines a narrowing strategy, namely the *needed narrowing* strategy. Roughly speaking, given a term $t = f(\bar{t})$, the needed narrowing strategy looks in the definitional tree of f for a rule node which applies to t directly or after reducing some of its subterms to head normal form.

More formally, if $\mathcal{T}$ is a node $rule(l \to r)$, then we apply the rule $l \to r$ to t. If $\mathcal{T}$ is a node $branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$, then we consider the subterm $t|_o$. If $t|_o$ has a function symbol at the top, this subterm is reduced to a head normal form by applying recursively the strategy to $t|_o$. If $t|_o$ has a constructor symbol at the top, we narrow t with a subtree $\mathcal{T}_i$ whose pattern unifies with t, or fail otherwise. If $t|_o$ is a variable, we (nondeterministically) select a subtree $\mathcal{T}_i$, unify t with the pattern of $\mathcal{T}_i$, and narrow this instance of t with $\mathcal{T}_i$.

4.2 Term Rewriting Systems and case expressions

It is possible to integrate the definitional trees into the term rewriting system by using *case expressions*. In this way, each function symbol has just one associated rule which completely defines the meaning of the function. Using case expressions as a represention of the equations which define a function is a common transformation technique in functional programming. The syntax of a case expression is as follows:

$$\begin{array}{ll} case\ X\ of & c_1(\overline{x}) : \mathcal{X}_1 \\ & \ldots \\ & c_n(\overline{y}) : \mathcal{X}_n \end{array}$$

where X is a variable, $c_1 \ldots c_n$ are different constructors of the sort of X, $\overline{x}, \ldots, \overline{y}$ are tuples of new variables, and $\mathcal{X}_1 \ldots, \mathcal{X}_n$ are terms, possibly containing case expressions. We interpret such a case expression with n patterns and n actions as a $2n+1$-ary function $case(X, c_1(\overline{x}), \mathcal{X}_1, \ldots, c_n(\overline{y}), \mathcal{X}_n)$.

In this way, we can put together the rules defining a function f in the original program into a single (case) rule in a transformed TRS. A more complete treatment of case expressions in the setting of the definitional trees can be found in [7].

Example 4.2 *The TRS of the Example 4.1 is transformed into the following TRS:*

```
from(N) → [N|from(s(N))]
first(X,Y) → case(X,0,[ ],s(N),case'(Y,[E|L],[E|first(N,L)]))
```

The semantics of the functions `case` *and* `case'` *can be simply defined by the following rewrite rules:*

```
case(0    ,0,X,s(N),Y) → X
case(s(N),0,X,s(N),Y) → Y
case'([E|L],[E|L],X)  → X
```

In general, a case expression with patterns $\pi_1, \ldots, \pi_n$ can be viewed as a function $\texttt{case}_{\pi_1,\ldots,\pi_n}$ which is defined by the following set of rules:

$\texttt{case}_{\pi_1,\ldots,\pi_n}(\pi_1, \pi_1, \texttt{X}, \_, \ldots, \_) \;\rightarrow\; \texttt{X}$
$\ldots$
$\texttt{case}_{\pi_1,\ldots,\pi_n}(\pi_n, \_, \ldots, \_, \pi_n, \texttt{X}) \;\rightarrow\; \texttt{X}$

We often omit the index $\pi_1, \ldots, \pi_n$ and abbreviate a *'pattern/action'* pair $(\pi, \mathcal{X})$ as Ξ and a tuple of these pairs as $\overline{\Xi}$.

4.3 Narrowing with case expressions

Given a term rewriting system $\mathcal{R} = (\Sigma, R)$, let $\mathcal{R}' = (\Sigma \cup \Sigma_c, R' \cup R_c)$ be the transformed TRS where Σ_c is the set of case function symbols, R' contains the case version of rules in R, and R_c is the set of additional case rules which must be considered.

Remark 4.3 *The transformation of $\mathcal{R}$ into $\mathcal{R}'$ have some consequences:*

1. *For each defined function f there is only one rule $f(\overline{x}) \rightarrow r \in \mathcal{R}'$. The right-hand side r of the rule is a (normal) term or a case expression.*
2. *Given a term $t = f(\overline{t})$, it is always possible to apply a rule. No evaluation of arguments in t is required for this application.*
3. *All rules in R_c have the form $case(c_i(\overline{x}), X_1, X'_1, \ldots, c_i(\overline{x}), X'_i, \ldots, X_n, X'_n) \rightarrow X'_i$, where $\overline{x}, X_1, X'_1, \ldots, X_n, X'_n$ are distinct variables.*
4. *Given a term $t = case(s, \pi_1, \mathcal{X}_1, \ldots, \pi_n, \mathcal{X}_n)$,* only the first argument *of this case term must be evaluated up to a head normal form in order to apply some case rule. Evaluations in the remaining arguments of a case expressions are not necessary.*

In expressing the definitional trees by means of case expressions, we can evaluate the goals by using a simpler narrowing strategy, namely *leftmost-outermost* narrowing.[4] Evaluating goals by needed narrowing w.r.t. $\mathcal{R}$ is equivalent to evaluating goals by leftmost-outermost narrowing w.r.t. $\mathcal{R}'$ in the following sense.

Theorem 4.4 ([7]) *Let t be a term with a function symbol at the top and $\mathcal{T}$ a definitional tree for this function symbol. For each needed narrowing derivation $t \overset{\mathcal{N}}{\leadsto}{}^*_\sigma\, c$ w.r.t. $\mathcal{R}$, there exists a leftmost-outermost narrowing derivation $t \overset{\mathcal{L}}{\leadsto}{}^*_{\sigma'} c$ w.r.t. $\mathcal{R}'$ with $\sigma =_{Var(t)} \sigma'$ and vice versa.*

[4]A position p is *leftmost-outermost* in a set P of positions if there is no $p' \in P$ with p' prefix of p, or $p' = q \cdot i \cdot q'$ and $p = q \cdot j \cdot q''$ and $i < j$. A narrowing step is *leftmost-outermost* if the selected subterm is the leftmost-outermost one among all possible narrowing positions.

5 Lazy denotational semantics

5.1 Preliminary definitions

In functional logic programming, given a goal t, we are interested in two classes of *observable* information: the evaluated expression(s) corresponding to this goal and the solution (or set of solutions) leading to this computed expression. Therefore, the answer for a goal is a function $\psi : \mathit{Term} \to \wp(\mathit{FSubst})$ which maps (possible evaluated) results to sets of substitutions.[5]

Example 5.1 Consider the rules of Example 4.1 and the term `first(X,from(0))`. Then the answer ψ representing all possible evaluations of this term has the property $\psi(\texttt{[]}) = \{\{\texttt{X} \mapsto \texttt{0}\}\}$, $\psi(\texttt{[0]}) = \{\{\texttt{X} \mapsto \texttt{s(0)}\}\}$, $\psi(\texttt{[0,s(0)]}) = \{\{\texttt{X} \mapsto \texttt{s(s(0))}\}\}$, and so on. Thus, ψ has a non-empty denotation containing a single substitution on lists of ascending natural numbers starting from `0`.

In the following, we use some operations on answers:

- The *bottom answer* $\psi_\perp$ is defined by $\psi_\perp(t) = \emptyset$ for all term t.
  This is the bottom element in the domain of answers.
- The *additive composition* $\diamond : \mathit{Answer} \times \mathit{Answer} \to \mathit{Answer}$ is defined by

  $$(\psi \diamond \psi')(t) = \psi(t) \cup \psi'(t) \ .$$

  Since $\diamond$ is a commutative and associative operation, we extend it to a set of answers Ψ in the obvious way, and write $\diamond\ \Psi$. In particular, if $\Psi = \emptyset$ we define $\diamond\ \Psi = \psi_\perp$.
- The *non-trivial domain* $\Delta : \mathit{Answer} \to \wp(\mathcal{T}(\Sigma, V))$, defined by

  $$\Delta(\psi) = \{t \in \mathcal{T}(\Sigma, V) \mid \psi(t) \neq \emptyset\} \ ,$$

  denotes the set of all terms which are computable by applying some substitution.
- We denote by $t \mapsto \Theta$ a *unique-valued answer* ψ which has the property $\psi(t) = \Theta$ and $\psi(t') = \emptyset$ for all $t' \neq t$.
- The *restriction* (infix operator) $\downarrow$: $\mathit{Answer} \times \wp(V) \to \mathit{Answer}$ is defined by $\psi{\downarrow_W}\ (t) = \psi(t){\downarrow_W}$.
- The right substitution composition $\triangleleft : \mathit{FSubst} \times \mathit{Answer} \to \mathit{Answer}$ is defined by $(\theta \triangleleft \psi)(t) = (\psi(t)) \circ \theta$. Note that $(\phi \circ \theta) \triangleleft \psi = \theta \triangleleft (\phi \triangleleft \psi)$.

[5]Other representations of answers, like sets of pairs of terms and substitutions sets, are also possible. However, our representation leads to a concise denotational semantics.

5.2 Denotational semantics

For a complete presentation of the denotational semantics, we introduce the denotational domains and the signatures of the semantic functions.

- **Domains:**

$Program = Rule^*$	(R)		$Rule = \mathsf{Rule}$	(ρ)
$Den = Term \to FSubst \to Answer$	(δ)		$FSubst = \mathsf{FSubst}$	$(\epsilon, \sigma, \theta)$
$Goal = Term$	(t)		$Term = \mathcal{T}(\Sigma, V)$	(t)
$Answer = Term \to \wp(FSubst)$	(ψ)			

- **Semantic functions:**
  $\mathbf{P}[\![\ ]\!] : Program \to Den \to Den$
  $\mathbf{R}[\![\ ]\!] : Rule \to Den \to Den$
  $\mathbf{G}[\![\ ]\!] : Goal \to Den \to FSubst \to Answer$

- **Definition of semantics functions:**
  $\mathbf{P}[\![\emptyset]\!]\ \delta = \delta_\perp$
  $\mathbf{P}[\![\{\rho\} \cup R]\!]\ \delta = \mathbf{R}[\![\rho]\!]\, \delta \diamondsuit \mathbf{P}[\![R]\!]\, \delta$

  $\mathbf{R}[\![l \to r]\!]\ \delta\, t\, \theta = \mathsf{if}\ \sigma = fail \vee \theta(t) \in V\ \mathsf{then}\ \psi_\perp\ \mathsf{else}\ (\mathbf{G}[\![\sigma(r')]\!]\ \delta\ (\sigma \circ \theta)){\downarrow_{\mathcal{V}ar(\theta)}}$
  $\mathsf{where}\ \ l' \to r' = ren(l \to r, \mathcal{V}ar(\theta) \cup \mathcal{V}ar(t))\ \ \mathsf{and}\ \ \sigma = unif(\theta(t), l')$

  $\mathbf{G}[\![x]\!]\ \delta\ \theta = x \mapsto \{\theta\}$
  $\mathbf{G}[\![c(\overline{t})]\!]\ \delta\ \theta = c(\overline{t}) \mapsto \{\theta\}$
  $\mathbf{G}[\![f(\overline{t})]\!]\ \delta\ \theta = \delta(f(\overline{t}))\ \theta$
  $\mathbf{G}[\![case(s, \overline{\Xi})]\!]\ \delta\ \theta =$
  $\quad \mathsf{let}\ \ \psi_s = \mathbf{G}[\![s]\!]\ \delta\ \theta\ \ \mathsf{in}\ \ \diamond_{\substack{s' \in \Delta(\psi_s) \\ \theta' \in \psi_s(s')}} (\delta(case(s', \overline{\Xi}))\ (\theta' \circ \theta)){\downarrow_{\mathcal{V}ar(\theta)}}$

A program is a sequence of rewrite rules (of the special form discussed in Section 4.2). The meaning of a program is a function which maps denotations into denotations. A denotation $\delta \in Den$ maps each term instantiated by some substitution into an answer. It describes the 'transformation power' of a program. Hence, $\delta(t)\ \theta$ expresses the (direct) evaluation (by means of the program) of t in the context of the substitution θ.

The meaning of an *empty* program is the *bottom* denotation $\delta_\perp$: $\mathbf{P}[\![\emptyset]\!]\ \delta = \delta_\perp$ where $\delta_\perp$ is defined as follows: $\delta_\perp(t)\ \theta = \psi_\perp$ for all t and θ. Thus, an empty program cannot compute any value for all possible input terms and substitutions.

The meaning of a non-empty program is the combination of all answers obtained by the different rules of the program. For this purpose, we have to define the composition of denotations which can be done by extending the additive composition $\diamond$ of answers to the corresponding operation on denotations. $\diamondsuit : Den \times Den \to Den$ is defined as $(\delta \diamondsuit \delta')(t)\ \theta = (\delta(t)\ \theta) \diamond (\delta'(t)\ \theta)$. Now, the meaning of a (non-empty) program is obtained by adding the denotations assigned to the rules contained in the program: $\mathbf{P}[\![\{\rho\} \cup R]\!]\ \delta = \mathbf{R}[\![\rho]\!]\, \delta \diamondsuit \mathbf{P}[\![R]\!]\, \delta$.

A rewrite rule is an evaluation component of the program. If a rule $l \to r$ applies (with unifier σ) to an (instantiated) term $\theta(t)$, then, to evaluate $\theta(t)$, just evaluate $\sigma(r)$:[6]

$$\begin{array}{l}\mathbf{R}[\![l \to r]\!]\ \delta\, t\, \theta = \mathsf{if}\ \sigma = fail\ \vee\ \theta(t) \in V\ \ \mathsf{then}\ \ \psi_\perp\ \ \mathsf{else}\ \ (\mathbf{G}[\![\sigma(r')]\!]\ \delta\ (\sigma \circ \theta)){\downarrow_{\mathcal{V}ar(\theta)}} \\ \mathsf{where}\ \ l' \to r' = ren(l \to r, \mathcal{V}ar(\theta) \cup \mathcal{V}ar(t))\ \ \mathsf{and}\ \ \sigma = unif(\theta(t), l')\end{array}$$

The meaning of a goal or term is given as a function that, for some denotation δ of the program and accumulated partial solution θ, yields the answer function for this goal. However, in lazy languages, we are not interested in a total evaluation of the arguments of a given function symbol. Since the arguments in the left-hand sides of rules are flat constructor terms or variables (see Section 4.3), we evaluate the expressions up to head normal form in order to apply some rule. Therefore, when a goal is a head normal form, we stop the (lazy) evaluation: $\mathbf{G}[\![t]\!]\ \delta\ \theta = t \mapsto \{\theta\}$ if t is a variable or a constructor-rooted term.

If the goal is an *operation-rooted* term $f(\overline{t})$, then a rule directly applies and we use the denotation of the program to obtain the meaning of the goal: $\mathbf{G}[\![f(\overline{t})]\!]\ \delta\ \theta = \delta(f(\overline{t}))\ \theta$.

If the goal is a case expression $case(s, \overline{\Xi})$, we may need some evaluation of the selection argument s. In this case, we evaluate s up to a *hnf*, thus obtaining the answer $\psi_s = \mathbf{G}[\![s]\!]\ \delta\ \theta$. Next, we apply the denotation of the program to the case expression with s replaced by all its possible evaluations $s' \in \Delta(\psi_s)$:

$$\mathbf{G}[\![case(s, \overline{\Xi})]\!]\ \delta\ \theta = \diamond_{\substack{s' \in \Delta(\psi_s) \\ \theta' \in \psi_s(s')}}\ (\delta(case(s', \overline{\Xi}))\ (\theta' \circ \theta)){\downarrow_{var(\theta)}}$$

As usual, the denotation δ_0 of a program is defined as the least fixpoint[7] of the defining equations, i.e., $\delta_0 = \mathsf{fix}\ \mathbf{P}[\![R \cup E]\!]$ where R contains the rewrite rules for operations and case functions and E is the set of equality rules of the given signature. Now, we can directly obtain the meaning of a given equation e by means of the function $\mathbf{G}[\![\ \ ]\!]$ as follows: $\psi = \mathbf{G}[\![e]\!]\ \delta_0\ \epsilon$, where ϵ is the identity substitution with $\mathcal{D}om(\epsilon) = \mathcal{V}ar(e)$. We can also obtain the result ψ of the *complete* evaluation of a given term t as follows: let $\psi' = \mathbf{G}[\![X \approx t]\!]\ \delta_0\ \epsilon$, where X is a fresh variable and ϵ is the identity substitution with $\mathcal{D}om(\epsilon) = \mathcal{V}ar(t) \cup \{X\}$. If $\psi' = \psi_\perp$, then $\psi = \psi_\perp$. Otherwise, ψ' must be a unique answer of the form $\mathtt{true} \mapsto \{\theta_1\{X \mapsto t_1\}, \ldots, \theta_n\{X \mapsto t_n\}, \ldots\}$, since equalities can only be derived to the constant $\mathtt{true}$. Then $\psi = t_1 \mapsto \{\theta_1\} \diamond \cdots \diamond t_n \mapsto \{\theta_n\} \diamond \cdots$.

5.3 Properties of the semantic functions

The following proposition allows us to express a solution θ obtained from the evaluation of either an operation rooted term or a case expression with evaluated selection

[6]$ren(s, W)$ denotes the renaming of s so that the new variant of s has no variables in common with W. By introducing a total ordering on V, it is easy to define ren as a function.

[7]We consider the standard ordering in the domains, e.g., $\emptyset$ is the least element in sets and functions are ordered pointwise on their domains.

argument, as a combination of the first *mgu* σ used to narrow the term and the answer which corresponds to the evaluation of the *rhs* instantiated by σ.

Proposition 5.2 *Let t be either an operation rooted term $t = f(\overline{t})$ or $t = case(s, \overline{\Xi})$ where s is a* hnf *such that there is a variant $\alpha_\sigma : l' \to r'$ of a rule which can be applied to t with mgu σ. Let Θ be the set of all of such unifiers. Let $\psi = \mathbf{G}[\![t]\!]\ \delta\ \epsilon$ with $\mathcal{V}ar(\epsilon) = \mathcal{V}ar(t)$ and $\psi'_\sigma = \mathbf{G}[\![\sigma(r')]\!]\ \delta\ \epsilon'$ with $\mathcal{V}ar(\epsilon') = \mathcal{V}ar(\sigma(r'))$. Then $\psi = \diamond_{\sigma \in \Theta}(\sigma \triangleleft \psi'_\sigma)\downarrow_{\mathcal{V}ar(t)}$.*

Roughly speaking, the following proposition amounts to say that we can express a solution θ of a case expression $case(s, \overline{\Xi})$ with a non evaluated selection argument s as a combination of the solutions ϕ of the evaluation s' of s and the answers corresponding to the partially evaluated goal $case(s', \overline{\Xi})$.

Proposition 5.3 *Let $t = case(s, \overline{\Xi})$, where s is not a* hnf*. Let ϵ_s such that $\mathcal{V}ar(\epsilon_s) = \mathcal{V}ar(s)$. Then*

$$\psi = \mathbf{G}[\![case(s, \overline{\Xi})]\!]\ \delta\ \epsilon = \diamond_{\substack{s' \in \Delta(\psi'_s) \\ \phi \in \psi'_s(s')}} (\phi \triangleleft \psi_\phi)\downarrow_{\mathcal{V}ar(\epsilon)}$$

where $\psi'_s = \mathbf{G}[\![s]\!]\ \delta\ \epsilon_s$, $\psi_\phi = \mathbf{G}[\![\phi(case(s', \overline{\Xi}))]\!]\ \delta\ \epsilon'$, $\mathcal{V}ar(\epsilon') = \mathcal{V}ar(\phi(case(s', \overline{\Xi})))$.

These propositions express the compositional behavior of the semantic evaluations. Note the similarities with the result shown in Proposition 3.5 for the operational setting.

The main result of the paper is given now. It expresses that the semantic definition reflects the operational behavior of the needed narrowing strategy.

Theorem 5.4 (Adequacy of the semantics w.r.t. needed narrowing) *Let $\mathcal{R} = (\Sigma, R)$ be an inductively sequential TRS and $\mathcal{R}' = (\Sigma \cup \Sigma_c, R' \cup R_c)$ be the case version of $\mathcal{R}$. Let $t \in \mathcal{T}(\Sigma, V)$ and $t' \in \mathsf{HNF}_\Sigma$. Let ϵ be the identity substitution with $\mathcal{D}om(\epsilon) = \mathcal{V}ar(t)$. If $\psi = \mathbf{G}[\![t]\!]\ \delta\ \epsilon$, then*

$$t \overset{\mathcal{N}}{\leadsto}{}^*_\sigma\ t' \Leftrightarrow \sigma\downarrow_{\mathcal{V}ar(t)} \in \psi(t').$$

6 Conclusions and further work

We have presented a denotational semantics for lazy functional logic languages based on the needed narrowing strategy. Our semantics is the first denotational definition of needed narrowing.

Some abstract interpretation frameworks reinforce the use of denotational semantics as the formal basis for data-flow program analysis by giving an intermediate meta-language able to couch the program as a set of semantic equations [10, 12]. In this way, a given analysis can be defined as a particular interpretation of the language. Therefore, this is a reusable framework for program analysis. We plan to apply this kind of strategy for the analysis of functional logic programs based on needed narrowing strategy.

References

[1] S. Antoy. Definitional Trees. In H. Kirchner and G. Levi, editors, *Proc. of Int'l Conf. on Algebraic and Logic Programming, ALP'92*, LNCS 632:143-157, Springer-Verlag, Berlin, 1992.

[2] S. Antoy R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc of the 21st ACM Symposium on Principles of Programming Languages, POPL'94*, pages 268-279, ACM Press, 1994.

[3] S.K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming* 5:61-91, 1988.

[4] 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 and The MIT Press, Cambridge, MA, 1990.

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

[6] M. Hanus and S. Lucas. Definition and analysis of a denotational semantics for needed narrowing. Technical Report, DSIC II/2/96, Universidad Politécnica de Valencia, 1996. Also available by URL http://www.dsic.upv.es/users/elp/papers.html.

[7] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Proc. of RTA'96.* To appear in Springer LNCS.

[8] 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.

[9] N.D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for PROLOG. In *Proc. of the 1984 International Symposium on Logic Programming*, pages 289-298, IEEE Computer Soc., Atlantic City, N.J., 1988.

[10] K. Marriott. Frameworks for abstract interpretation. *Acta Informatica* 30:103-129, 1993.

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

[12] F. Nielson. Towards a denotational theory of abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis Horwood Ltd., John Wiley and sons, pages 218-245, 1987.

[13] U.S. Reddy. Functional Logic Languages Part I (Preliminary Report). In J.H. Fasel and R.M. Keller, editors, *Proc. of a Workshop on Graph Reduction*, LNCS 279:401-425, Springer-Verlag, Berlin, 1987.

[14] J.C. Shepherdson. Mistakes in Logic Programming: the Role of Standarising Apart, manuscript, University of Bristol, 1991.

[15] J.E. Stoy. Denotational semantics: the Scott-Strachey approach to programming language theory. The MIT Press, Cambridge MA, 1977.

[16] R.D. Tennent. Denotational Semantics. In S. Abramsky, D.M. Gabbay and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, Volume 3, pages 169-322, Oxford University Press, Oxford, 1992.