Preliminary version. Final version in Information Processing Letters (Elsevier), Vol. 85, No. 1, pp. 19–25, 2003

A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs [⋆]

Elvira Albert [a] Michael Hanus [b] Germán Vidal [a,∗]

[a] *DSIC, U. Politécnica de Valencia, Camino de Vera s/n, E-46022 Valencia, Spain*

[b] *Institut für Informatik, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany*

Abstract

Recent proposals for multi-paradigm declarative programming combine the most important features of functional, logic and concurrent programming into a single framework. The operational semantics of these languages is usually based on a combination of narrowing and residuation. In this note, we introduce a non-standard, *residualizing* semantics for multi-paradigm declarative programs and prove its equivalence with a standard operational semantics. Our residualizing semantics is particularly relevant within the area of program transformation where it is useful, e.g., to perform computations during partial evaluation. Thus, the proof of equivalence is a crucial result to demonstrate the correctness of (existing) partial evaluation schemes.

Key words: Programming languages, formal semantics, program transformation

1 Introduction

Multi-paradigm declarative languages (like Curry [6,7]) integrate features from functional, logic and concurrent programming (e.g., lazy evaluation, partial data structures, non-deterministic computations, and concurrent evaluation with synchronization on logical variables) into a single programming paradigm. The operational semantics of these languages is usually based on a combination of two different operational principles: narrowing and residuation [6]. The *residuation* principle is based on the idea of delaying function calls until they are ready for a deterministic evaluation (by rewriting). On the other hand, the *narrowing* mechanism allows the instantiation of variables in input expressions and, then, applies reduction steps to the function calls of the instantiated expressions. Each function has an *evaluation annotation* (see [6,7]) indicating whether it should be eval-

[⋆] Some preliminary material of this note appeared in LPAR 2000 [1].
This work has been partially supported by CICYT TIC 2001-2705-C03-01, by Acción Integrada HA2001-0059, and by the DFG under grant Ha 2457/1-2.
[∗] Corresponding author.

uated by residuation (for functions annotated as *rigid*) or by narrowing (for *flexible* functions). Due to its optimality properties w.r.t. the length of derivations and the number of computed solutions, *needed narrowing* [5] is currently the best narrowing strategy for multi-paradigm functional logic programs. The formulation of needed narrowing is based on the use of *definitional trees* [4], which define a strategy to evaluate functions by applying narrowing steps. Recently, [8] introduced a *flat* representation for functional logic programs in which definitional trees are embedded in the rewrite rules by means of case expressions. The interest in using the flat representation arises because it provides more explicit control (hence the associated calculus is simpler than needed narrowing), while source programs can be still automatically translated to the new representation.

In this article, we define a new, *residualizing* version of the operational semantics for flat programs: the RLNT calculus. We call it "residualizing" because it does not compute bindings but encodes them by means of case expressions (which are considered "residual" code), which contrasts with the previous semantics. This property is essential in the area of program transformation and, particularly, to perform computations during *partial evaluation* [10]. Indeed, recent approaches to the partial evaluation of multi-paradigm functional logic languages are defined at the level of the flat representation and use the RLNT calculus to perform partial evaluations (see, e.g., [1–3]).

2 The Flat Language

This section briefly introduces a *flat* representation for multi-paradigm functional logic programs and its operational semantics. Similar representations are considered in [8,9,11]. Unlike them, we consider two kinds of case expressions in order to represent *flexible/rigid* evaluation annotations of source programs as expressions. Since inductively sequential programs [4] (with evaluation annotations) can be automatically translated into the flat representation, our approach covers recent proposals for multi-paradigm functional logic programming. The syntax for programs in the flat representation is:

$$\begin{aligned}
\mathcal{R} &::= \mathcal{D}_1 \ldots \mathcal{D}_m \\
\mathcal{D} &::= f(\overline{x_n}) = e \\
e &::= x \ \mid\ c(\overline{e_n}) \ \mid\ f(\overline{e_n}) \\
&\quad\ \mid \mathit{case}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\} \\
&\quad\ \mid \mathit{fcase}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\} \\
p &::= c(\overline{x_n})
\end{aligned}$$

Here, we write $\overline{o_n}$ for the sequence of objects $o_1, \ldots, o_n$. Thus, a program $\mathcal{R}$ consists of a sequence of function definitions D such that the left-hand side is linear and has only variable arguments, i.e., pattern matching is compiled into case expressions. The right-hand side of each function definition is an expression e composed by variables ($\mathcal{X}$), constructors ($\mathcal{C}$), function calls ($\mathcal{F}$), and case expressions for pattern matching. Variables are denoted by $x, y, z \ldots$, constructors by $a, b, c \ldots$, and defined functions by $f, g, h \ldots$ The general form of a case expression is:

$$\begin{array}{rl}(f)case\ e\ of\ \{ & c_1(\overline{x_{n_1}}) \to e_1 \ ; \\ & \quad \ldots \quad ; \\ & c_k(\overline{x_{n_k}}) \to e_k \ \}\end{array}$$

where e is an expression, $c_1, \ldots, c_k$ are different constructors of the type of e, and $e_1, \ldots, e_k$ are expressions (possibly containing case structures). The variables $\overline{x_{n_i}}$ are local variables which occur only in the corresponding expression e_i. The difference between *case* and *fcase* shows up when the argument e is a free variable: *case* suspends (which corresponds to residuation) whereas *fcase* nondeterministically binds this variable to the pattern in a branch of the case expression and proceeds with the appropriate branch (which corresponds to narrowing). Functions defined only by *fcase* or *case* expressions are called *flexible* or *rigid*, respectively.

For instance, the (flexible) function "app" to concatenate two lists can be written in the flat representation by the following single rule:

```
app (x, y)  =
  fcase x of {
        [] → y;
        (z : zs) → z : app (zs, y) }
```

The operational semantics of flat programs is based on the LNT (Lazy Narrowing with definitional Trees) calculus [8]. In Figure 1, we present a slight extension of this calculus in order to cope with case expressions including evaluation annotations; nevertheless, we still use the name "LNT calculus" for simplicity. First, note that the symbols "⟦" and "⟧" in an expression like $[\![e]\!]$ are purely syntactical (i.e., they do not denote "the value of e"). Indeed, they are only used to mark subexpressions where the inference rules may be applied. LNT steps are labeled with the substitution computed in the step. The empty substitution is denoted by *id*. Let us briefly describe the LNT rules:

Case Select. It is used to select the appropriate branch of the current case expression. If there is no matching branch, the evaluation of the current expression *fails*.

Case Guess. It non-deterministically selects a branch of a flexible case expression and instantiates the variable at the case argument to the appropriate constructor pattern. The step is labeled with the computed substitution σ. Rigid case expressions with a variable argument *suspend*, giving rise to an abnormal termination.

Case Eval. It is used to evaluate case expressions with a function call or another case expression in the argument position. Here, $root(e)$ denotes the outermost symbol of e. This rule initiates the evaluation of the case argument by creating a (recursive) call for this subexpression. If it cannot be evaluated, we stop (unsuccessfully).

Function Eval. This rule performs the unfolding of a function call. If there is no defining rule, the evaluation of the current expression *fails*. As in logic programming, we assume that rules are renamed so that they always contain fresh variables.

Case Select

$$[\![(f)case\ c(\overline{e_n})\ of\ \{\overline{p_k \to e'_k}\}]\!] \Rightarrow_{id} [\![\sigma(e'_i)]\!] \quad \text{if } p_i = c(\overline{x_n}) \text{ and } \sigma = \{\overline{x_n \mapsto e_n}\}$$

Case Guess

$$[\![fcase\ x\ of\ \{\overline{p_k \to e_k}\}]\!] \Rightarrow_{\sigma} [\![\sigma(e_i)]\!] \quad \text{if } \sigma = \{x \mapsto p_i\},\ i = 1, \ldots, k$$

Case Eval

$$[\![(f)case\ e\ of\ \{\overline{p_k \to e_k}\}]\!] \Rightarrow_{\sigma} [\![\sigma((f)case\ e'\ of\ \{\overline{p_k \to e_k}\})]\!]$$
$$\text{if } [\![e]\!] \Rightarrow_{\sigma} [\![e']\!],\ e \notin \mathcal{X}, \text{ and } root(e) \notin \mathcal{C}$$

Function Eval

$$[\![f(\overline{e_n})]\!] \Rightarrow_{id} [\![\sigma(e')]\!] \quad \text{if } f(\overline{x_n}) = e' \in \mathcal{R} \text{ and } \sigma = \{\overline{x_n \mapsto e_n}\}$$

Fig. 1. The LNT calculus

Arbitrary LNT *derivations* are denoted by $e \Rightarrow^*_\sigma e'$ which is a shorthand for the sequence of steps $e \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_n} e'$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). We say that an LNT derivation $e \Rightarrow^*_\sigma e'$ is *successful* when e' is in *head normal form*, i.e., it is rooted by a constructor symbol or a variable; in this case, we say that e evaluates to e' with *answer* σ. Our calculus can be easily extended to evaluate expressions to normal form, but we keep the above presentation for simplicity. Furthermore, the calculus is intended to perform partial evaluations where computations proceed at most to head normal form (although this is not strictly necessary to preserve correctness results, current partial evaluators follow this convention; see, e.g., [2]).

3 The Residualizing Semantics

The first framework to perform partial evaluation in functional logic languages (see [3] for a survey) considered the same mechanism (narrowing) for both execution and partial evaluation. Basically, the idea is to compute a *residual rule* of the form $\sigma(e) = e'$ for each partial computation $e \Rightarrow^*_\sigma e'$. However, the backpropagation of bindings to the left-hand sides of residual rules poses several problems in the context of the flat representation (see, e.g., [1]). For instance, the variables in the left-hand sides of residual rules may become instantiated, which is not allowed in our flat syntax (cf. Section 2). Therefore, we propose a new, *residualizing* version of the LNT calculus in which variable bindings are encoded by (flexible) case expressions (and are considered "residual" code). This avoids the backpropagation of bindings and makes our calculus suitable for being used in partial evaluation. The inference rules of the Residualizing LNT calculus (RLNT calculus in the following) are shown in Fig. 2.

The main difference with the LNT formulation is in the Case Guess rule. In particular, the new defini-

Case Select

$$[\![(f)case\ c(\overline{e_n})\ of\ \{\overline{p_k \to e'_k}\}]\!] \;\Rightarrow\; [\![\sigma(e'_i)]\!] \quad \text{if } p_i = c(\overline{x_n}) \text{ and } \sigma = \{\overline{x_n \mapsto e_n}\}$$

Case Guess

$$[\![(f)case\ x\ of\ \{\overline{p_k \to e_k}\}]\!] \;\Rightarrow\; (f)case\ x\ of\ \{\overline{p_k \to [\![\sigma_k(e_k)]\!]}\}$$
$$\text{if } \sigma_i = \{x \mapsto p_i\},\ i = 1, \ldots, k$$

Case Eval

$$[\![(f)case\ e\ of\ \{\overline{p_k \to e_k}\}]\!] \;\Rightarrow\; [\![(f)case\ e'\ of\ \{\overline{p_k \to e_k}\}]\!]$$
$$\text{if } [\![e]\!] \Rightarrow [\![e']\!],\ e \notin \mathcal{X},\ root(e) \notin \mathcal{C}, \text{ and}$$
$$e \neq (f)case\ x\ of\ \{\ldots\}$$

Function Eval

$$[\![f(\overline{e_n})]\!] \;\Rightarrow\; [\![\sigma(e')]\!] \quad \text{if } f(\overline{x_n}) = e' \in \mathcal{R} \text{ and } \sigma = \{\overline{x_n \mapsto e_n}\}$$

Case-of-Case

$$[\![(f)case\ ((f)case\ x\ of\ \{\overline{p_k \to e_k}\})\ of\ \{\overline{p'_j \to e'_j}\}]\!]$$
$$\Rightarrow\; [\![(f)case\ x\ of\ \{\overline{p_k \to (f)case\ e_k\ of\ \{\overline{p'_j \to e'_j}\}}\}]\!]$$

Fig. 2. The RLNT calculus

tion "residualizes" the case structure and continues with the evaluation of the different branches (by applying the corresponding substitution in order to propagate bindings forward in the computation). It imitates the instantiation of variables in the standard evaluation of a flexible case but keeps the case structure. Due to this modification, no distinction between flexible and rigid case expressions is needed. Moreover, the resulting calculus does not compute "answers". Rather, they are represented in the derived expressions by means of case expressions with variable arguments. Also, the calculus becomes deterministic, i.e., there is no don't know nondeterminism involved in the computations (thus only one derivation can be issued from a given expression).

An undesirable effect of the Case Guess rule is that nested case expressions may suspend unnecessarily. Consider the expression

$$[\![\ \mathtt{case}\ (\mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \{[] \to \mathtt{T};\ (\mathtt{y} : \mathtt{ys}) \to \mathtt{F}\})\ \mathtt{of}\ \{\mathtt{T} \to \mathtt{F}\}]\!]$$

The evaluation of this expression suspends since the outer case can be evaluated only if the argument is a variable (Case Guess), a function call (Case Eval) or a constructor-rooted term (Case Select). To avoid such premature suspensions, we introduce the Case-of-Case rule, which moves the outer case inside the branches of the inner one and, thus, the evaluation of the branches can now proceed (similar rules can be found in the Glasgow Haskell Compiler as well as in Wadler's deforestation [13]). By using the Case-of-Case rule, the above expression can be reduced to:

$[\![$`case x of {`

  `[] → case T of {T → F}`

  `(y : ys) → case F of {T → F} }`$]\!]$

which can be further simplified by applying the Case Guess and Case Select rules. Rigorously speaking, the Case-of-Case rule can be expanded into four rules (with the different combinations for *case* and *fcase*), but we keep the above (less formal) presentation for simplicity. Observe that the outer case expression may be duplicated several times, but each copy is now (possibly) scrutinizing a known value, and so the Case Select rule can be applied to eliminate some case constructs.

The same considerations of Section 2 about the symbols "$[\![$" and "$]\!]$", derivations, etc., apply here. In contrast to the LNT calculus, expressions like $[\![e]\!]$ can also occur at inner positions. In this case we also allow RLNT steps at these positions (which can be formally defined by a congruence rule for case expressions). In the RLNT calculus, the relation $\Rightarrow$ is not labeled with a substitution since the new calculus does not compute bindings. This allows us to use the same arrow "$\Rightarrow$" for the formalization of both calculi without confusion. The following theorem establishes a precise equivalence between the (nondeterministic) LNT calculus (Fig. 1) and its residualizing, deterministic version: the RLNT calculus (Fig. 2). The correctness of a partial evaluation scheme based on our residualizing semantics [1,2] relies on this result. First, we need the auxiliary relation $\hookrightarrow$, which is defined by

$$fcase\ x\ of\ \{\overline{p_k \to e_k}\} \hookrightarrow_\sigma e_i$$

where $\sigma = \{x \mapsto p_i\}$, $i = 1, \ldots, k$. This (nondeterministic) relation is used to extract the bindings encoded by residualized case expressions.

Theorem 1 *Let e be an expression, e' a head normal form, and $\mathcal{R}$ a flat program. For each LNT derivation $[\![e]\!] \Rightarrow^*_\sigma e'$ in $\mathcal{R}$, there exists an RLNT derivation $[\![e]\!] \Rightarrow^* e''$ in $\mathcal{R}$ such that $e'' \hookrightarrow^*_\sigma e' \not\hookrightarrow$, and vice versa.*

Informally speaking, for each LNT derivation from $[\![e]\!]$ to a head normal form e', computing σ, there is an RLNT derivation from $[\![e]\!]$ to some e'' in which the computed substitution σ is encoded in e'' by case expressions and can be obtained by a (finite) sequence of $\hookrightarrow$ steps (deriving the same expression e').

PROOF. Our proof proceeds by relating the application of a rule in one calculus to the application of one or more rules in the other calculus.

($\Rightarrow$) Let us consider an LNT derivation of the form $[\![e]\!] \Rightarrow^*_\sigma e'$. We proceed by induction on the length n of this derivation.

Base case ($n = 0$). Trivial.

Inductive case ($n > 0$). Assume that the LNT derivation has the form $[\![e]\!] \Rightarrow_\theta e^a \Rightarrow^*_\gamma e'$ where $\sigma = \gamma \circ \theta$. Now, we distinguish several cases depending on the rule applied in the first step:

Case Select. Since its definition is the same in both calculi, the claim follows trivially by induction.

Case Guess. Then, $[\![e]\!]$ has the form $[\![fcase\ x\ of\ \{\overline{p_k \to e_k}\}]\!]$ and, thus, $e^a = [\![\sigma_j(e_j)]\!]$, where $\sigma_j = \theta = \{x \mapsto p_j\}$ for some $j \in \{1,\ldots,k\}$. In the RLNT calculus, we can also apply the equivalent rule and obtain

$$e^{a'} = fcase\ x\ of\ \{\overline{p_k \to [\![\sigma_k(e_k)]\!]}\}$$

where $\sigma_i = \{x \mapsto p_i\}$ for all $i = 1,\ldots,k$. Since $e^a \Rightarrow^*_\gamma e'$, by the induction hypothesis, there exists an RLNT derivation $e^a \Rightarrow^* e^b$ with $e^b \hookrightarrow^*_\gamma e' \not\hookrightarrow$. Consider $e^{a'}$ represented as $C[e^a]$ (i.e., a context C which contains the subterm e^a). Let $C[e^b]$ be the expression which results from $e^{a'} \equiv C[e^a]$ by replacing the occurrence of e^a by e^b. Since $e^a \Rightarrow^* e^b$, we have that $C[e^a] \Rightarrow^* C[e^b]$, and, thus, $[\![e]\!] \Rightarrow C[e^a] \Rightarrow^* C[e^b]$. Finally, since $C[e^b] \hookrightarrow_{\sigma_j} e^b$ and $e^b \hookrightarrow^*_\gamma e' \not\hookrightarrow$ (by the induction hypothesis), we have $C[e^b] \hookrightarrow^*_\sigma e' \not\hookrightarrow$ and the claim follows.

Case Eval. This case is immediate except when $[\![e]\!]$ has the form $[\![(f)case\ e^x\ of\ \{\overline{p_m \to e_m}\}]\!]$ with $e^x = fcase\ x\ of\ \{\overline{p'_k \to e'_k}\}$ (since, in the remaining cases, rule Case Eval behaves identically in both calculi when we only consider derivations to head normal form). In this case, we get $e^a = [\![\sigma_j((f)case\ \sigma_j(e'_j)\ of\ \{\overline{p_k \to e_k}\})]\!] \equiv [\![\sigma_j((f)case\ e'_j\ of\ \{\overline{p_k \to e_k}\})]\!]$ with $\theta = \sigma_j = \{x \mapsto p'_j\}$ and $j \in \{1,\ldots,k\}$ by one application of rule Case Eval which, recursively, demands the application of rule Case Guess. In the RLNT calculus, we can first apply rule Case-of-Case to obtain the new expression $[\![fcase\ x\ of\ \{\overline{p'_k \to e''_k}\}]\!]$ with $e''_i = (f)case\ e'_i\ of\ \{\overline{p_m \to e_m}\}$ for all $i = 1,\ldots,k$. Then, by applying rule Case Guess, we get

$$e^{a'} = fcase\ x\ of\ \{\overline{p'_k \to [\![\sigma_k(e''_k)]\!]}\}$$

with $\sigma_i = \{x \mapsto p'_i\}$ for all $i = 1,\ldots,k$. Similarly to the previous case, we can consider $e^{a'}$ of the form $C[e^a]$ and the inductive hypothesis can be applied as in the Case Guess.

Function Eval. As for the Case Select rule, this case follows trivially by induction, since its definition is the same in both calculi.

($\Leftarrow$) Let us consider an RLNT derivation $[\![e]\!] \Rightarrow^* e''$ with $e'' \hookrightarrow^*_\sigma e' \not\hookrightarrow$. We proceed by induction on the sum n of the length of the RLNT derivation plus the length of the $\hookrightarrow$ derivation. It is obvious that the length of $[\![e]\!] \Rightarrow^* e''$ could not be zero when $e'' \hookrightarrow^*_\sigma e' \not\hookrightarrow$ has a positive length (since $\hookrightarrow$ only applies when there are residualized case expressions produced by relation $\Rightarrow$). We will assume this property in the following.

Base case ($n = 0$). Trivial.

Inductive case ($n > 0$). Assume that the RLNT derivation has the form $[\![e]\!] \Rightarrow e^a \Rightarrow^* e''$ with $e'' \hookrightarrow^*_\sigma e' \not\hookrightarrow$. Now, we distinguish several cases depending on the rule applied in the first step:

Case Select. Since its definition is the same in both calculi, the claim follows trivially by induction.

Case Guess. Then, $[\![e]\!]$ has the form $[\![fcase\ x\ of\ \{\overline{p_k \to e_k}\}]\!]$. Note that, if the case expression is rigid, e' could not be a head normal form (since $\hookrightarrow$

cannot remove the outer case expression). Thus, e must be rooted by a flexible case. By applying rule Case Guess, we get

$$e^a = fcase\ x\ of\ \{\overline{p_k \to [\![\sigma_k(e_k)]\!]}\}$$

where $\sigma_i = \{x \mapsto p_i\}$ for all $i = 1, \ldots, k$. Now, we look at the $\hookrightarrow$ derivation, since it will determine the corresponding LNT step. Let us assume $e'' \hookrightarrow_\theta e^b \hookrightarrow^*_\gamma e'$ with $\sigma = \gamma \circ \theta$. Since the outermost case has been residualized, e'' is of the form $e'' = fcase\ x\ of\ \{\overline{p_k \to e^c_k}\}$. Thus, we have that $\theta = \sigma_j$ for some $j \in \{1, \ldots, k\}$. By definition of the "$\hookrightarrow$" relation, if $e^a \Rightarrow^* e'' \hookrightarrow_{\sigma_j} e^b$, then $[\![\sigma_j(e_j)]\!] \Rightarrow^* e^b$ for some $j \in \{1, \ldots, k\}$. Now, $[\![e]\!] \Rightarrow_{\sigma_j} [\![\sigma_j(e_j)]\!]$, by applying once rule Case Guess. Finally, by the induction hypothesis, we have $[\![\sigma_j(e_j)]\!] \Rightarrow^*_\gamma e'$ and the claim follows.

Case Eval and Function Eval. As for the Case Select rule, the proof follows trivially by induction, since these steps can be also done with the corresponding LNT rules.

Case-of-Case. Then, $[\![e]\!]$ has the form $[\![(f)case\ e^x\ of\ \{\overline{p_m \to e_m}\}]\!]$ with $e^x = (f)case\ x\ of\ \{\overline{p'_k \to e'_k}\}$. By applying Case-of-Case, we get $[\![(f)case\ x\ of\ \{\overline{p'_k \to e''_k}\}]\!]$ with $e''_i = (f)case\ e'_i\ of\ \{\overline{p_m \to e_m}\}$ for all $i = 1, \ldots, k$. The same considerations of the previous case apply, thus the outer case expression must be flexible. Then, the only possibility is to apply rule Case Guess, thus obtaining $e^a = fcase\ x\ of\ \{\overline{p'_k \to [\![\sigma_k(e''_k)]\!]}\}$ with $\sigma_i = \{x \mapsto p'_i\}$ for all $i = 1, \ldots, k$. As in the previous case, we look at the $\hookrightarrow$ derivation, since it will determine the corresponding LNT step. We assume $e'' \hookrightarrow_\theta e^b \hookrightarrow^*_\gamma e'$ with $\sigma = \gamma \circ \theta$. Since the outermost case has been residualized, e'' is of the form $e'' = fcase\ x\ of\ \{\overline{p_k \to e^c_k}\}$. Thus, we have that $\theta = \sigma_j$ for some $j \in \{1, \ldots, k\}$. By definition of the "$\hookrightarrow$" relation, if $e^a \Rightarrow^* e'' \hookrightarrow_{\sigma_j} e^b$, then $[\![\sigma_j(e_j)]\!] \Rightarrow^* e^b$ for some $j \in \{1, \ldots, k\}$. Now, $[\![e]\!] \Rightarrow_{\sigma_j} [\![\sigma_j(e''_j)]\!]$ by applying once rule Case Eval which demands the application of rule Case Guess. Finally, by the induction hypothesis, we have $[\![\sigma_j(e''_j)]\!] \Rightarrow^*_\gamma e'$ and the claim follows.

4 Discussion

This note presents a novel, non-standard, residualizing semantics for functional logic programs and demonstrates its equivalence with a standard semantics for such programs. In the field of program transformation, the residualizing semantics has been shown especially well suited to perform computations during partial evaluation (see, e.g., [1,2]). Thus, our proof of equivalence is the basis for the correctness of this partial evaluation scheme. Moreover, we think that our calculus is of independent interest itself and can be used for other applications in the field of program transformation.

In the literature, many different calculi have been defined for the purpose of program transformation. Among them, the closest to our work are the driving mechanism to perform positive supercompilation [12] and

Wadler's calculus for deforestation [13]. In particular, the main difference with driving is the use of logic variables by means of flexible case constructs. Indeed, our underlying semantics is based on a combination of narrowing and residuation, while driving is defined for purely functional expressions. On the other hand, an important difference w.r.t. deforestation is revealed in the Case Guess rule, where the patterns are substituted in the different branches, like in the driving transformation.

References

[1] E. Albert, M. Hanus, G. Vidal, Using an Abstract Representation to Specialize Functional Logic Programs, in: Proc. of the Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR 2000), Springer LNAI 1955, 2000, pp. 381–398.

[2] E. Albert, M. Hanus, G. Vidal, A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages, Journal of Functional and Logic Programming 2002 (1).

[3] E. Albert, G. Vidal, The Narrowing Driven Approach to Functional Logic Program Specialization, New Generation Computing 20 (1) (2002) 3–26.

[4] S. Antoy, Definitional trees, in: Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92), Springer LNCS 632, 1992, pp. 143–157.

[5] S. Antoy, R. Echahed, M. Hanus, A Needed Narrowing Strategy, Journal of the ACM 47 (4) (2000) 776–822.

[6] M. Hanus, A Unified computation model for functional and logic programming, in: Proc. of ACM Symp. on Principles of Programming Languages (POPL '97), ACM, New York, 1997, pp. 80–93.

[7] M. Hanus, Curry: An Integrated Functional Logic Language, `http://www.informatik.uni-kiel.de/~curry` (2000).

[8] M. Hanus, C. Prehofer, Higher-Order Narrowing with Definitional Trees, Journal of Functional Programming 9 (1) (1999) 33–75.

[9] T. Hortalá-González, E. Ullán, An Abstract Machine Based System for a Lazy Narrowing Calculus, in: Proc. of the 5th Int'l Symp. on Functional and Logic Programming (FLOPS 2001), Springer LNCS 2024, 2001, pp. 216–232.

[10] N. Jones, C. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[11] W. Lux, H. Kuchen, An Efficient Abstract Machine for Curry, in: Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99), 1999, pp. 171–181.

[12] M. Sørensen, R. Glück, N. Jones, A Positive Supercompiler, Journal of Functional Programming 6 (6) (1996) 811–838.

[13] P. Wadler, Deforestation: Transforming Programs to Eliminate Trees, Theoretical Computer Science 73 (1990) 231–248.