A Semantic Framework for Functional Logic Programming with Algebraic Polymorphic Types*

(Extended Abstract)

P. Arenas-Sánchez and M. Rodríguez-Artalejo

Universidad Complutense de Madrid, Departamento de Informática y Automática
Facultad de CC. Matemáticas, Av. Complutense s/n, 28040 Madrid, Spain
email:{puri,mario}@dia.ucm.es

Abstract. We propose a formal framework for functional logic programming, supporting lazy functions, non-determinism and polymorphic datatypes whose data constructors obey a given set $\mathcal{C}$ of equational axioms. On top of a given $\mathcal{C}$, we specify a program as a set $\mathcal{R}$ of $\mathcal{C}$-based conditional rewriting rules for defined functions. We argue that equational logic does not supply the proper semantics for such programs. Therefore, we present an alternative logic which includes $\mathcal{C}$-based rewriting calculi and a notion of model. We get soundness and completeness for $\mathcal{C}$-based rewriting w.r.t. models, existence of free models for all programs, and type preservation results.

1 Introduction

The interest in multiparadigm declarative programming has grown up during the last decade, giving rise to different approaches to the integration of functions into logic programming; see [10] for a survey. In particular, some *lazy functional logic languages* such as K-LEAF [6] and BABEL [17] have been designed to combine lazy evaluation and unification. This is achieved by presenting programs as rewriting systems and using *lazy narrowing* (a notion introduced in [19]) as a goal solving mechanism.

Classical equational logic does not supply an adequate semantics for functional logic languages, since equations between terms that are intended to denote the same infinite data structure are often not deducible. Recently, a *constructor based rewriting logic* has been proposed as an alternative semantic framework for lazy functional logic languages [7]. This approach includes rewriting calculi, a model-theoretic semantics and a strongly complete lazy narrowing calculus for goal solving. The aim of the present paper is to extend the approach in [7] by introducing *algebraic polymorphic datatypes*, similar to those used in modern functional languages (see e.g. [21]), but allowing to specify a set $\mathcal{C}$ of equational axioms for the data constructors[2]. For instance, we can define a datatype $Set(\alpha)$ for polymorphic sets with constructors $\{\ \} :\rightarrow Set(\alpha)$ and $\{\cdot|\cdot\} : (\alpha, Set(\alpha)) \rightarrow Set(\alpha)$, governed by the equational axioms $\{x|\{y|zs\}\} \approx \{y|\{x|zs\}\}$ and $\{x|\{x|zs\}\} \approx \{x|zs\}$. Simply by omitting the second equation, we obtain a datatype $MSet(\alpha)$ for polymorphic multisets.

Data structures based on non-free constructors, specially sets and multisets, play an important role in several recent proposals for extended logic programming and multiparadigm declarative programming; see e.g. [13, 4, 8, 15]. As a

* This research has been partially supported by the the Spanish National Project TIC95-0433-C03-01 "CPD" and the Esprit BRA Working Group EP-22457 "CCLII".

[2] Note that user-defined datatypes are also called "algebraic" in Haskell. In spite of this terminology, Haskell's data constructors are free.

novel point, we combine non-free constructors with lazy functions and parametric polymorphism. We view a program as a set of $\mathcal{C}$-based conditional rewrite rules to define the behaviour of lazy functions on top of a given finite set $\mathcal{C}$ of equational axioms for data constructors. As in [7], defined functions can be partial and/or non-deterministic, in the spirit of [12]. For instance, a non-deterministic partial function select : $Set(\alpha) \to \alpha$ that selects an arbitrary element from a non-empty set, can be defined by the single rewrite rule $select(\{x|xs\}) \to x$.

We present a semantic framework for this kind of programs, following the lines of [7], but with two major modifications. Firstly, our model-theoretic semantics uses algebras with two carriers (for data and types, respectively), inspired by the polymorphically order-sorted algebras from [18]. Secondly, the constructor-based rewriting calculi from [7] have been modified to incorporate a set $\mathcal{C}$ of equational axioms for constructors, while respecting the intended behaviour of lazy evaluation. To achieve this aim, we give an *inequational calculus* which interprets each equational axiom in $\mathcal{C}$ as a scheme for generating inequalities between *partial data terms* (built from constructors and a bottom symbol $\bot$). Inequalities are thought of as defining an approximation ordering.

The rest of the paper is organized as follows: Sect. 2 sets the basic formalism, defining polymorphic signatures and expressions. In Sect. 3, we introduce equational axioms for data constructors along with the calculus needed to deduce approximation inequalities from them. In Sect. 4 we present $\mathcal{C}$-based rewrite rules and rewriting calculi for defining lazy functions on top of a given set $\mathcal{C}$ of equational axioms. This section also includes some type preservation results. Sect. 5 deals with model theory, showing the existence of free models for programs and soundness and completeness results for the rewriting calculi w.r.t. models. The concluding Sect. 6 summarizes our results and points to some lines for future research.

Proofs have been omitted due to lack of space. They can be found in a Technical Report, available at http://mozart.mat.ucm.papers/1996/TR96-39.ps.gz.

2 Signatures, Expressions and Types

We assume a countable set $TVar$ of *type variables* α, β etc., and a countable ranked alphabet $TC = \bigcup_{n \geq 0} TC^n$ of *type constructors* C. *Polymorphic types* $\tau, \tau' \in T_{TC}(TVar)$ are built as $\tau ::= \alpha | C(\tau_1, \ldots, \tau_n), C \in TC^n$. The set of type variables occurring in τ is written $tvar(\tau)$. We define a *polymorphic signature* Σ over TC as a triple $\langle TC, DC, FS \rangle$, where DC is a set of type declarations for *data constructors*, of the form $c : (\tau_1, \ldots, \tau_n) \to \tau_0$ with $\bigcup_{i=1}^{n} tvar(\tau_i) \subseteq tvar(\tau_0)$ (so-called *transparency* property), and FS is a set of type declarations for *defined function symbols*, of the form $f : (\tau_1, \ldots, \tau_n) \to \tau_0$. We require that Σ does not include multiple type declarations for the same symbol. The types given by declarations in $DC \cup FS$ are called *principal types*. We will write $h \in DC^n \cup FS^n$ to indicate the arity of a symbol according to its type declaration. In the following, $DC_\bot$ will denote DC extended by a new declaration $\bot :\to \alpha$. The bottom constant constructor $\bot$ is intended to represent an undefined value. Analogously, $\Sigma_\bot$ will denote the result of replacing DC by $DC_\bot$ in Σ.

Assuming another countable set $DVar$ of *data variables* x, y, etc., we build *expressions* $e, r, l \ldots \in Expr_\Sigma(DVar)$ as $e ::= x | h(e_1, \ldots, e_n), h \in DC^n \cup FS^n$. The set $Expr_{\Sigma_\bot}(DVar)$ of *partial expressions* is defined in the same way, but

using $DC_\perp$ in place of DC. *Total data terms* $Term_\Sigma(DVar) \subseteq Expr_\Sigma(DVar)$ and *partial data terms* $Term_{\Sigma_\perp}(DVar) \subseteq Expr_{\Sigma_\perp}(DVar)$ are built by using variables and data constructors only. In the sequel, we reserve t, s to denote possibly partial data terms, and we write $dvar(e)$ for the set of all data variables occurring in a expression e.

We define *type substitutions* $\theta \in TSub$ as mappings from $TVar$ to $T_{TC}(TVar)$, and possibly partial data substitutions $\delta \in DSub_\perp$ as mappings from $Dvar$ to $Term_{\Sigma_\perp}(DVar)$. *Total data substitutions* $\delta \in DSub$ are mappings from $DVar$ to $Term_\Sigma(DVar)$. Pairs (θ, δ), with $\theta \in TSub$ and $\delta \in DSub_\perp$ are called *substitutions.* We will use postfix notation for the result of applying substitutions to types and expressions. We will say that $\delta \in DSub_\perp$ is *allowed* for a data term t if $\delta(x)$ is a total term for every variable x having more than one occurrence in t. The notions of *instance*, *renaming* and *variant* have the usual definitions; see e.g. [3].

An *environment* is defined as any set V of type-annotated data variables $x : \tau$, such that V does not include two different annotations for the same variable. The set of *well-typed expressions* w.r.t. an environment V is defined as $Expr_{\Sigma_\perp}(V) = \bigcup_{\tau \in T_{TC}(TVar)} Expr^\tau_{\Sigma_\perp}(V)$, where $e \in Expr^\tau_{\Sigma_\perp}(V)$ holds iff the type judgment $V \vdash_{\Sigma_\perp} e : \tau$ is derivable by means of the following type inference rules:

- $V \vdash_{\Sigma_\perp} x : \tau$ if $x : \tau \in V$;
- $V \vdash_{\Sigma_\perp} h(e_1, \ldots, e_n) : \tau$ if $V \vdash_{\Sigma_\perp} e_i : \tau_i$, where $h : (\tau_1, \ldots, \tau_n) \to \tau$ is an instance of the unique declared principal type associated to h in $DC_\perp \cup FS$.

$Expr^\tau_{\Sigma_\perp}(V)$ has subsets $Expr^\tau_\Sigma(V)$, $Term^\tau_{\Sigma_\perp}(V)$, $Term^\tau_\Sigma(V)$ that are defined in the natural way. It is easy to prove that every well-typed expression has a a most general principal type, which is unique up to renaming.

3 Equations for Algebraic Constructors

We will specify the behaviour of data constructors by means of a set $\mathcal{C}$ of *equational axioms* $s \approx t$, where s, t are *total data terms.* Such an axiom is called *regular* iff $dvar(s) = dvar(t)$; *non-collapsing* iff neither s nor t is a variable; and *strongly regular* iff regular and non-collapsing. $\mathcal{C}$ will be called (strongly) regular iff it consists of (strongly) regular equations. In the rest of the paper, we focus on strongly regular axioms because strong regularity is needed for our current type preservation results; see Theorem 2 in section 4 below.

We say that a strongly regular axiom $c(t_1, \ldots, t_n) \approx d(s_1, \ldots, s_m)$ is *well-typed* iff the principal type declarations for c, d have variants $c : (\tau_1, \ldots, \tau_n) \to \tau$ and $d : (\tau'_1, \ldots, \tau'_m) \to \tau$ such that $c(t_1, \ldots, t_n), d(s_1, \ldots, s_n) \in Term^\tau_\Sigma(V)$, for some environment V. A set $\mathcal{C}$ of strongly regular axioms is called *well-typed* iff each axiom in $\mathcal{C}$ is well-typed.

Example 1. Assume that Σ includes the constructor declarations $True, False :\to Bool$; $Zero :\to Nat$; $Suc : Nat \to Nat$; $\{\ \} :\to Set(\alpha)$; and $\{\cdot|\cdot\} : (\alpha, Set(\alpha)) \to Set(\alpha)$. Then, the two equational axioms $\{x|\{y|zs\}\} \approx \{y|\{x|zs\}\}$ and $\{x|\{x|zs\}\} \approx \{x|zs\}$ are strongly regular and well-typed, by means of $V = \{x : \alpha, y : \alpha, zs : Set(\alpha)\}$. On the contrary, the strongly regular equation $\{Zero|\{y|zs\}\} \approx \{y|\{Zero|zs\}\}$ is not well-typed, since it does not conform to the most general type of the set constructors. □

In subsequent examples, we will use abbreviations such as $\{x, y|zs\}$, $\{x, y\}$, and $\{x\}$ for the terms $\{x|\{y|zs\}\}$, $\{x|\{y|\{\ \}\}\}$ and $\{x|\{\ \}\}$, respectively.

Given a set $\mathcal{C}$ of equational axioms, the following *inequational calculus* allows to derive inequalities $s \sqsupseteq t$, with s, t possibly partial data terms:

Inequational calculus

Bottom: $\frac{}{t \sqsupseteq \bot}$ Reflexivity: $\frac{}{t \sqsupseteq t}$ Transitivity: $\frac{t \sqsupseteq t', t' \sqsupseteq t''}{t \sqsupseteq t''}$

Monotonicity: $\frac{t_1 \sqsupseteq s_1, \ldots, t_n \sqsupseteq s_n}{c(t_1, \ldots, t_n) \sqsupseteq c(s_1, \ldots, s_n)}$ $\mathcal{C}$-inequation: $\frac{}{s \sqsupseteq t}$ if $s \sqsupseteq t \in [\mathcal{C}]_{\sqsupseteq}$

where $t, t', t'', c(t_1, \ldots, t_n), c(s_1, \ldots, s_n) \in Term_{\Sigma_\bot}(DVar)$, and:

$[\mathcal{C}]_{\sqsupseteq} = \{s\delta \sqsupseteq t\delta, t\delta' \sqsupseteq s\delta' \mid s \approx t \in \mathcal{C}, \delta, \delta' \in DSub_\bot,$
δ and δ' are allowed for s and t respectively$\}$

In the rest of the paper, the notation $s \sqsupseteq_{\mathcal{C}} t$ denotes the formal derivability of $s \sqsupseteq t$ using the above inequational calculus for $\mathcal{C}$. Moreover, we write $s \approx_{\mathcal{C}} t$ iff $s \sqsupseteq_{\mathcal{C}} t$ and $t \sqsupseteq_{\mathcal{C}} s$. Thinking of partial data terms as approximations of data, $s \sqsupseteq_{\mathcal{C}} t$ can be read as "t approximates s". Note that the formulation of $\mathcal{C}$-inequation forbids to use the axiom $\{x, x|zs\} \approx \{x|zs\}$ from Example 1 to derive the inequality $\{\bot, \bot\} \sqsupseteq_{\mathcal{C}} \{\bot\}$, which would have undesirable consequences (see Example 3 in Sect. 4 below).

Remark that $\sqsupseteq_{\mathcal{C}}$ and $\approx_{\mathcal{C}}$ are, respectively, the least precongruence and the least congruence over $Term_{\Sigma_\bot}(DVar)$ that contain $[\mathcal{C}]_{\sqsupseteq}$. Furthermore, if $\mathcal{C}$ is regular then for any $s, t \in Term_{\Sigma_\bot}(DVar)$, if $s \sqsupseteq_{\mathcal{C}} t$ and t is total then s is also total and $s \approx_{\mathcal{C}} t$.

4 Defining Rules, Programs and Rewriting Calculi

On top of a given set $\mathcal{C}$ of equational axioms for data constructors, we introduce constructor-based rewrite rules for defined functions. More precisely, assuming a principal type declaration $f : (\tau_1, \ldots, \tau_n) \to \tau \in FS$, a *defining rule* for f must have the form: $f(t_1, \ldots, t_n) \to r \Leftarrow a_1 \bowtie b_1, \ldots, a_m \bowtie b_m$, where the left-hand side is *linear* (i.e. without multiple occurrences of variables), $t_i \in Term_\Sigma(DVar)$, $1 \leq i \leq n$, and $a_j, b_j, r \in Expr_\Sigma(DVar)$, $1 \leq j \leq m$. *Joinability conditions* $a_j \bowtie b_j$ are intended to hold iff a_j, b_j can be reduced to some common *total* $t \in Term_\Sigma(DVar)$, as in [7]. A formal definition will be given below.

A defining rule is called *regular* iff all variables occurring in r occur also in the left-hand side. Extra variables in the conditions are allowed, as well as the inconditional case $m = 0$. We define *programs* as triples $\mathcal{P} = \langle \Sigma, \mathcal{C}, \mathcal{R} \rangle$, where Σ is a polymorphic signature, $\mathcal{C}$ is a finite set of equational axioms for constructors in Σ, and $\mathcal{R}$ is a finite set of defining rules for defined functions symbols in Σ. We will say that a program $\mathcal{P}$ is *strongly regular* iff $\mathcal{C}$ is strongly regular and all rules in $\mathcal{R}$ are regular.

Programs are intended to solve *goals* composed of joinability conditions; i.e. goals will have the same form as conditions for defining rules. The expressive power of algebraic constructors in our programs can be used to model *action and change* problems declaratively, avoiding the so-called *frame problem* [8]. In [8], it has been already shown that *planning problems* can be modeled by means of *equational logic programs*, using a binary (ACI) operation $\circ$, to represent situations as multisets of facts $\mathsf{fact_1} \circ \ldots \circ \mathsf{fact_n}$, and a ternary predicate

execPlan(initialSit, plan, finalSit) to model the transformation of an initial situation into a final situation by the execution of a plan. In our framework we can follow the same idea even more naturally, using multisets of facts to represent situations, and a non-deterministic function execPlan : $(List(Action), Mset(Fact)) \rightarrow MSet(Fact)$ to represent the effect of plan execution.

Next example, adapted from [8], shows a little program which solves a very simple planning problem in our setting. More complicated action and change problems could be treated analogously.

Example 2. A thirsty person named Bert wants to get a lemonade from a vending machine which only accepts quarters. The lemonade costs 75 cents and Bert has a one-dollar note. There is a cashier which changes a dollar into four quarters. The possible facts we have are D (a one dollar-note), Q (a quarter) and L (a lemonade). The available actions are GetChange and GetLemonade whose intended meaning can be easily deduced from function execAction.

The problem of getting the lemonade can be described in our framework by means of the following program:

datatypes $Fact, Action, Mset(\alpha), List(\alpha)$
constructors

$D, Q, L :\rightarrow Fact$	$\{\!\{\cdot\mid\cdot\}\!\} : (\alpha, Mset(\alpha)) \rightarrow Mset(\alpha)$
$GetChange, GetLemonade :\rightarrow Action$	$[\,] :\rightarrow List(\alpha)$
$\{\!\{\,\}\!\} :\rightarrow Mset(\alpha)$	$[\cdot\mid\cdot] : (\alpha, List(\alpha)) \rightarrow List(\alpha)$

equations $\{\!\{x, y|xs\}\!\} \approx \{\!\{y, x|xs\}\!\}$
functions

execPlan : $(List(Action), Mset(Fact)) \rightarrow MSet(Fact)$
$execPlan([\,], sit) \rightarrow sit$
$execPlan([act|restAct], sit) \rightarrow execPlan(restAct, execAction(act, sit))$
execAction : $(Action, Mset(Fact)) \rightarrow Mset(Fact)$
$execAction(GetChange, \{\!\{D|otherFacts\}\!\}) \rightarrow \{\!\{Q, Q, Q, Q|otherFacts\}\!\}$
$execAction(GetLemonade, \{\!\{Q, Q, Q|otherFacts\}\!\}) \rightarrow \{\!\{L|otherFacts\}\!\}$

A possible goal would be execPlan(plan, {!D!}) $\bowtie$ {!L, Q!}, for which we expect plan = [GetChange, GetLemonade] as a computed answer[3]. □

Some of our subsequent results refer to *well-typed programs*. A strongly regular program $\mathcal{P} = \langle \Sigma, \mathcal{C}, \mathcal{R} \rangle$ is well-typed iff $\mathcal{C}$ is well-typed and every defining rule $f(t_1, \ldots, t_n) \rightarrow r \Leftarrow C \in \mathcal{R}$ is well-typed in the following sense: there is some environment V such that $t_i \in Term^{\tau_i}_{\Sigma}(V)$, $1 \leq i \leq n$, $r \in Expr^{\tau}_{\Sigma}(V)$ and for all $a \bowtie b \in C$ there is some type τ' such that $a, b \in Expr^{\tau'}_{\Sigma}(V)$. For instance, if we extend Example 1 with the new declaration union : $(Set(\alpha), Set(\alpha)) \rightarrow Set(\alpha)$, the defining rule $union(\{x|xs\}, ys) \rightarrow \{x|union(xs, ys)\}$ is well-typed, while $union(\{Zero|xs\}, ys) \rightarrow \{Zero|union(xs, ys)\}$ is not, because the type of $\{Zero|xs\}$ is too particular.

In the rest of this section we present constructor-based rewriting calculi which are intended as a proof-theoretical specification of programs' semantics. As in [7], our calculi are designed to derive two kinds of statements: *reduction statements* $e \rightarrow e'$, intended to mean that e can be reduced to e', and *joinability statements*

[3] Computing answers for goals will require a suitable narrowing calculus, whose development is left for future work.

$e \bowtie e'$, intended to mean that e and e' can be reduced to some common total data term. Reduction statements of the form $e \to t$, where t is a possibly partial data term, will be called *approximation statements*. For a given program $\mathcal{P} = \langle \Sigma, \mathcal{C}, \mathcal{R} \rangle$, the basic rewriting calculus (BRC) and the goal-oriented rewriting calculus ($GORC$) are defined as follows:

Basic Rewriting Calculus *BRC*

Bottom: $\dfrac{}{e \to \bot}$ Reflexivity: $\dfrac{}{e \to e}$ Transitivity $\dfrac{e \to e', e' \to e''}{e \to e''}$

Monotonicity: $\dfrac{e_1 \to e_1', \ldots, e_n \to e_n'}{h(e_1, \ldots, e_n) \to h(e_1', \ldots, e_n')}$

$\mathcal{R}$-reduction: $\dfrac{C}{l \to r}$ if $l \to r \Leftarrow C \in [\mathcal{R}]_{\to}$

$\mathcal{C}$-mutation: $\dfrac{}{s \to t}$ if $s \sqsupseteq t \in [C]_{\sqsupseteq}$

Join: $\dfrac{e \to t, e' \to t}{e \bowtie e'}$ if $t \in Term_{\Sigma}(DVar)$ is a *total* data term

where $e, e', e'', h(e_1, \ldots, e_n), h(e_1', \ldots, e_n') \in Expr_{\Sigma_\bot}(DVar)$, and

$$[\mathcal{R}]_{\to} = \{(l \to r \Leftarrow C)\delta \mid l \to r \Leftarrow C \in \mathcal{R}, \delta \in DSub_{\bot}\}$$

Goal-Oriented Rewriting Calculus *GORC*

Bottom: $\dfrac{}{e \to \bot}$ Restricted Reflexivity: $\dfrac{}{x \to x}$

Decomposition: $\dfrac{e_1 \to t_1, \ldots, e_n \to t_n}{c(e_1, \ldots, e_n) \to c(t_1, \ldots, t_n)}$

Outer $\mathcal{C}$-mutation: $\dfrac{e_1 \to t_1, \ldots, e_n \to t_n, s \to t}{c(e_1, \ldots, e_n) \to t}$ if $t \neq \bot, c(\bar{t}) \sqsupseteq s \in [C]_{\sqsupseteq}$

Outer $\mathcal{R}$-reduction: $\dfrac{e_1 \to t_1, \ldots, e_n \to t_n, C, r \to t}{f(e_1, \ldots, e_n) \to t}$ if $t \neq \bot, f(\bar{t}) \to r \Leftarrow C \in [\mathcal{R}]_{\to}$

Join: $\dfrac{e \to t', e' \to t'}{e \bowtie e'}$ if $t' \in Term_{\Sigma}(DVar)$ is a *total* data term

where $e, e', c(e_1, \ldots, e_n), f(e_1, \ldots, e_n) \in Expr_{\Sigma_\bot}(DVar)$, $t, c(t_1, \ldots, t_n) \in Term_{\Sigma_\bot}(DVar)$ and $x \in DVar$.

Note that the construction of $[\mathcal{R}]_{\to}$ does not require δ to be allowed for l, in contrast to the construction of $[C]_{\sqsupseteq}$ in the inequational calculus. This is because l is known to be linear. Neither of the two calculi specifies rewriting in the usual sense. Rule Bottom shows that $e \to t$ is intended to mean "t approximates e", and the construction of $[\mathcal{R}]_{\to}$, $[C]_{\sqsupseteq}$ reflects a "call-time choice" treatment of non-determinism, as explained in [12]. As the main novelty w.r.t. [7], we find the *mutation* rules $\mathcal{C}$-mutation (respect. Outer $\mathcal{C}$-mutation) to deal with equations between constructors. We have presented the two calculi because BRC is closer to the intuition, while the goal-oriented format of the $GORC$-like calculus in [7] was found useful as a basis for designing a complete lazy narrowing calculus. The next result ensures that both calculi are essentially equivalent. Moreover, they are compatible with the inequational calculus presented in Sect. 3.

Theorem 1. (Calculi equivalence) *Let $\mathcal{P} = \langle \Sigma, \mathcal{C}, \mathcal{R} \rangle$ be a program.*

(a) For strongly regular $\mathcal{C}$, $e, e' \in Expr_{\Sigma_\perp}(DVar)$ and $t \in Term_{\Sigma_\perp}(DVar)$: $e \to t$ (respect. $e \bowtie e'$) is derivable in GORC iff $e \to t$ (respect. $e \bowtie e'$) is derivable in BRC;

(b) For any $t, t' \in Term_{\Sigma_\perp}(DVar)$, $t \sqsupseteq_{\mathcal{C}} t'$ iff $t \to t'$ is derivable in BRC.

(c) If $\mathcal{C}$ is regular, then for any $s, t \in Term_{\Sigma_\perp}(DVar)$, $s \bowtie t$ is derivable in BRC iff $s \approx_{\mathcal{C}} t$ and $s, t \in Term_{\Sigma}(DVar)$. ∎

In the rest of the paper, when we write $e \to_{\mathcal{P}} t$ (respect. $e \bowtie_{\mathcal{P}} e'$) we mean that $e \to t$ (respect. $e \bowtie e'$) is derivable in *BRC* or *GORC*. At this point, we can give an example that justifies why we require left-linear defining rules and allowed data substitutions for the construction of $[C]_{\sqsupseteq}$ in the inequational calculus.

Example 3. Let $\mathcal{P}$ be the program obtained by extending Example 1 with the following function type declarations and defining rules:

$\mathsf{eq} : (\alpha, \alpha) \to Bool$	$\mathsf{unit}, \mathsf{duo} : Set(\alpha) \to Bool$	$\mathsf{om} :\to \alpha$
$eq(x, x) \to True$	$unit(\{x\}) \to True$	$om \to om$
	$duo(\{x, y\}) \to True$	

Note that the defining rule for eq is not left-linear and thus illegal. If it were allowed, we would obtain $eq(e, e') \to_{\mathcal{P}} True$ for arbitrary e, e' (by using $e \to_{\mathcal{P}} \perp$, $e' \to_{\mathcal{P}} \perp$ and $eq(\perp, \perp) \to_{\mathcal{P}} True$).

On the other hand, if we would define $\sqsupseteq_{\mathcal{C}}$ in such a way that $\{\perp, \perp\} \sqsupseteq_{\mathcal{C}} \{\perp\}$ could be derived as some instance of $\{x, x | zs\} \approx \{x | zs\}$, we could use $True \to_{\mathcal{P}} \perp$, $False \to_{\mathcal{P}} \perp$ and $unit(\{\perp\}) \to_{\mathcal{P}} True$ for obtaining $unit(\{True, False\}) \to_{\mathcal{P}} True$, which is not expected as a reasonable consequence from unit's defining rule.

Finally, note that the inequational calculus permits $\{\perp\} \sqsupseteq_{\mathcal{C}} \{\perp, \perp\}$. We can combine this with $om \to_{\mathcal{P}} \perp$ and $duo(\{\perp, \perp\}) \to_{\mathcal{P}} True$ to obtain $duo(\{om\}) \to_{\mathcal{P}} True$ which does not contradict our intuitive understanding of the program. □

To conclude this Section, we give a type preservation result.

Theorem 2. (Type preservation) *Let $\mathcal{P} = \langle \Sigma, \mathcal{C}, \mathcal{R} \rangle$ be a well-typed strongly regular program. Let V be an environment. If $e \to_{\mathcal{P}} e'$ and $e \in Expr^{\tau}_{\Sigma_\perp}(V)$ then $e' \in Expr^{\tau}_{\Sigma_\perp}(V)$.* ∎

The last Theorem fails in general if non-regular equations or collapsing regular equations are allowed in $\mathcal{C}$:

Example 4. Let us consider the signature Σ from Example 1 and the empty environment V. Assuming the non-regular axiom $Suc(x) \approx Suc(y)$, we obtain $Suc(Zero) \to_{\mathcal{P}} Suc(True)$, where $Suc(Zero) \in Term^{Nat}_{\Sigma}(V)$ but $Suc(True) \notin Term^{Nat}_{\Sigma}(V)$. Taking the collapsing regular axiom $x \approx Suc(x)$, we get $True \to_{\mathcal{P}} Suc(True)$, where $True \in Term^{Bool}_{\Sigma}(V)$ but $Suc(True) \notin Term^{Bool}_{\Sigma}(V)$. □

5 Model-theoretic Semantics

In this section we will present a model-theoretic semantics, showing also its relation to the rewriting calculi from Section 4. First, we recall some basic notions from domain theory [20].

A *poset* with bottom $\perp$ is any set S partially ordered by $\sqsubseteq$, with least element $\perp$. Def(S) denotes the set of all maximal elements $u \in S$, also called *totally*

defined. Assume $X \subseteq S$. X is a *directed set* iff for all $u, v \in X$ there exists $w \in X$ s.t. $u, v \sqsubseteq w$. X is a *cone* iff $\bot \in X$ and X is downwards closed w.r.t. $\sqsubseteq$. X is an *ideal* iff X is a directed cone. We write $\mathcal{C}(S)$ and $\mathcal{I}(S)$ for the sets of cones and ideals of S, respectively. $\mathcal{I}(S)$ ordered by set inclusion $\subseteq$ is a poset with bottom $\{\bot\}$, called the *ideal completion* of S. Mapping each $u \in S$ into the principal ideal $\langle u \rangle = \{v \in S | v \sqsubseteq u\}$ gives an order preserving embedding. It is known (see e.g. [16]) that $\mathcal{I}(S)$ is the least cpo D s.t. S can be embedded into D. Due to these results, our semantic constructions below could be reformulated in terms of Scott domains [20]. In particular, totally defined elements $u \in \mathsf{Def(S)}$ correspond to finite and maximal elements $\langle u \rangle$ in the ideal completion.

As in [7], to represent non-deterministic lazy functions we use models with posets as carriers, interpreting function symbols as monotonic mappings from elements to cones. The elements of the poset are viewed as finite approximations of possibly infinite values. For given posets D and E, we define the set of all *non-deterministic functions* from D to E as

$$[D \to_{nd} E] = \{f : D \to \mathcal{C}(E) \mid \forall u, u' \in D : (u \sqsubseteq_D u' \Rightarrow f(u) \subseteq f(u'))\}$$

and the set of all *deterministic functions* from D to E as

$$[D \to_{d} E] = \{f \in [D \to_{nd} E] \mid \forall u \in D : f(u) \in \mathcal{I}(E)\}$$

Note that, a deterministic function f computes a directed set of partial values; hence, after performing an ideal completion, such functions become continuous mappings between algebraic cpos. Notice also, that a non-deterministic function f can be extended to a monotonic mapping $f^* : \mathcal{C}(D) \to \mathcal{C}(E)$ defined as $f^*(C) = \bigcup_{c \in C} f(c)$. Abusing of notation, we will identify f with its extension f^*.

We are now prepared to introduce our algebras, combining ideas from [7, 18].

Definition 3. (Polymorphically Typed algebras) Let Σ be a polymorphic signature. A Polymorphically Typed algebra (PT-algebra) $\mathcal{A}$ has the following structure:

$$\mathcal{A} = \langle D^{\mathcal{A}}, T^{\mathcal{A}}, :^{\mathcal{A}}, \{C^{\mathcal{A}} \mid C \in TC\}, \{c^{\mathcal{A}} \mid c : \vec{\tau} \to \tau \in DC_{\bot}\}, \{f^{\mathcal{A}} \mid f : \vec{\tau_*} \to \tau_* \in FS\} \rangle$$

where:

(1) $D^{\mathcal{A}}$ (data universe) is a poset with partial order $\sqsubseteq^{\mathcal{A}}$ and bottom element $\bot^{\mathcal{A}}$ and $T^{\mathcal{A}}$ (type universe) is a set;
(2) $:^{\mathcal{A}} \subseteq D^{\mathcal{A}} \star T^{\mathcal{A}}$ is a binary relation such that for all $\ell \in T^{\mathcal{A}}$, the extension of ℓ in $\mathcal{A}$, defined as: $\mathcal{E}^{\mathcal{A}}(\ell) = \{u \in D^{\mathcal{A}} \mid u :^{\mathcal{A}} \ell\}$ is a cone in $D^{\mathcal{A}}$;
(3) For each $C \in TC^n$, $C^{\mathcal{A}} : (T^{\mathcal{A}})^n \to T^{\mathcal{A}}$ (simply $C^{\mathcal{A}} \in T^{\mathcal{A}}$ if $n = 0$);
(4) for all $c : (\tau_1, \ldots, \tau_n) \to \tau \in DC_{\bot}$, $c^{\mathcal{A}} \in [(D^{\mathcal{A}})^n \to_d D^{\mathcal{A}}]$ satisfies: For all $u_i \in D^{\mathcal{A}}$, there exists $v \in D^{\mathcal{A}}$ such that $c^{\mathcal{A}}(u_1, \ldots, u_n) = \langle v \rangle$. Moreover, if $u_i \in \mathsf{Def}(\mathsf{D}^{\mathcal{A}})$ then $v \in \mathsf{Def}(\mathsf{D}^{\mathcal{A}})$;
(5) for all $f : (\tau'_1, \ldots, \tau'_m) \to \tau' \in FS$, $f^{\mathcal{A}} \in [(D^{\mathcal{A}})^m \to_{nd} D^{\mathcal{A}}]$. □

Note that as in [18], $:^{\mathcal{A}}$ relates the elements of $D^{\mathcal{A}}$ (carrier for data) to the elements of $T^{\mathcal{A}}$ (carrier for types). Note also that the preservation of finite and maximal elements in the ideal completion of $D^{\mathcal{A}}$ is ensured in item (4).

In order to interpret expressions in an algebra $\mathcal{A}$ we use *valuations* $\xi = (\mu, \eta)$, where $\mu : TVar \to T^{\mathcal{A}}$ is a *type valuation* and $\eta : DVar \to D^{\mathcal{A}}$ is a *data valuation*. ξ is called *totally defined* iff $\eta(x) \in \mathsf{Def}(\mathsf{D}^{\mathcal{A}})$, for all $x \in DVar$; and ξ is called *allowed* for a given $t \in Term_{\Sigma_{\bot}}(DVar)$ iff $\eta(x) \in \mathsf{Def}(\mathsf{D}^{\mathcal{A}})$, for all $x \in dvar(t)$

s.t. x has more than one occurrence in t. $Val(\mathcal{A})$ denotes the set of all valuations over $\mathcal{A}$.

For a given $\xi = (\mu, \eta) \in Val(\mathcal{A})$, *type denotations* $[\![\tau]\!]^{\mathcal{A}}\xi = [\![\tau]\!]^{\mathcal{A}}\mu \in T^{\mathcal{A}}$ and *expression denotations* $[\![e]\!]^{\mathcal{A}}\xi = [\![e]\!]^{\mathcal{A}}\eta \in \mathcal{C}(D^{\mathcal{A}})$ are defined recursively:

- $[\![\alpha]\!]^{\mathcal{A}}\mu = \mu(\alpha)$;
- $[\![C(\tau_1, \ldots, \tau_n)]\!]^{\mathcal{A}}\mu = C^{\mathcal{A}}([\![\tau_1]\!]^{\mathcal{A}}\mu, \ldots, [\![\tau_n]\!]^{\mathcal{A}}\mu)$, $C \in TC^n$, $\tau_i \in Expr_{\Sigma_\perp}(DVar)$.
- $[\![\perp]\!]^{\mathcal{A}}\eta = \{\perp^{\mathcal{A}}\}$ and $[\![x]\!]^{\mathcal{A}}\eta = \langle\eta(x)\rangle$, for all $x \in DVar$;
- $[\![h(e_1, \ldots, e_n)]\!]^{\mathcal{A}}\eta = h^{\mathcal{A}}([\![e_1]\!]^{\mathcal{A}}\eta, \ldots, [\![e_n]\!]^{\mathcal{A}}\eta)$, for all $h : (\tau_1, \ldots, \tau_n) \to \tau \in DC \cup FS$, $e_i \in Expr_{\Sigma_\perp}(DVar)$.

As in [7], it is easy to prove that $[\![t]\!]^{\mathcal{A}}\eta$ is a principal ideal $\langle u\rangle$ for each term $t \in Term_{\Sigma_\perp}(DVar)$. Moreover, $u \in \mathsf{Def}(\mathsf{D}^{\mathcal{A}})$ if η is totally defined.

We are particularly interested in those PT-algebras that are well-behaved w.r.t. types. We say that algebra $\mathcal{A}$ is *well-typed* iff for all $h : (\tau_1, \ldots, \tau_n) \to \tau_0 \in DC_\perp \cup FS$ we have $h^{\mathcal{A}}(\mathcal{E}^{\mathcal{A}}([\![\tau_1]\!]^{\mathcal{A}}\mu), \ldots, \mathcal{E}^{\mathcal{A}}([\![\tau_1]\!]^{\mathcal{A}}\mu)) \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau_0]\!]^{\mathcal{A}}\mu)$ for every type valuation μ. Also, for given $\xi = (\mu, \eta) \in Val(\mathcal{A})$, we say that ξ is *well-typed* w.r.t an environment V iff $\eta(x) \in \mathcal{E}^{\mathcal{A}}([\![\tau]\!]^{\mathcal{A}}\mu)$ for every $x : \tau \in V$. Reasoning by structural induction, we can prove that expression denotations behave as expected w.r.t. well-typed algebras and valuations:

Theorem 4. *Let V be an environment. Let $\mathcal{A}$ be a well-typed PT-algebra and $\xi = (\mu, \eta) \in Val(\mathcal{A})$ well-typed w.r.t. V. For all $e \in Expr^{\tau}_{\Sigma_\perp}(V)$, $[\![e]\!]^{\mathcal{A}}\eta \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau]\!]^{\mathcal{A}}\mu)$.* ∎

Next, we define the notion of *model*. Note that reduction/approximation is interpreted as inclusion, while joinability is interpreted as existence of a common maximal approximation.

Definition 5. (Models of a program) Let $\mathcal{A}$ be a PT-algebra.

(i) Let $\xi = (\mu, \eta)$ be a valuation over $\mathcal{A}$. $(\mathcal{A}, \eta) \models e \bowtie e'$ iff $[\![e]\!]^{\mathcal{A}}\eta \cap [\![e']\!]^{\mathcal{A}}\eta \cap \mathsf{Def}(\mathsf{D}^{\mathcal{A}}) \neq \emptyset$. And $(\mathcal{A}, \eta) \models e \to e'$ iff $[\![e']\!]^{\mathcal{A}}\eta \subseteq [\![e]\!]^{\mathcal{A}}\eta$.

(ii) $\mathcal{A}$ satisfies a defining rule $l \to r \Leftarrow C$ iff every $\xi = (\mu, \eta) \in Val(\mathcal{A})$ such that $(\mathcal{A}, \eta) \models C$ verifies that $(\mathcal{A}, \eta) \models l \to r$.

(iii) $\mathcal{A}$ satisfies an equation $s \approx t$ iff for every $\xi = (\mu, \eta) \in Val(\mathcal{A})$: $[\![s]\!]^{\mathcal{A}}\eta \supseteq [\![t]\!]^{\mathcal{A}}\eta$ if ξ is allowed for s and $[\![t]\!]^{\mathcal{A}}\eta \supseteq [\![s]\!]^{\mathcal{A}}\eta$ if ξ is allowed for t.

(iv) Let $\mathcal{P} = \langle\Sigma, \mathcal{C}, \mathcal{R}\rangle$ be a program. $\mathcal{A}$ is a model of $\mathcal{P}$ ($\mathcal{A} \models \mathcal{P}$) iff $\mathcal{A}$ satisfies every defining rule in $\mathcal{R}$ and every equation in $\mathcal{C}$. □

The rest of the section is devoted to the construction of free term models, which allow to prove soundness and completeness of the rewriting calculi from Sect. 4.

Definition 6. (Free term models) Given a program $\mathcal{P} = \langle\Sigma, \mathcal{C}, \mathcal{R}\rangle$ and an environment V, we build the term model $\mathcal{M}_{\mathcal{P}}(V)$ as follows:

- Data universe: Let $X = \{x \in DVar \mid \text{x } \textit{occurs in } V\}$. Then the *data universe* of $\mathcal{M}_{\mathcal{P}}(V)$ is $Term_{\Sigma_\perp}(X)/\approx_{\mathcal{C}}$. For all $t \in Term_{\Sigma_\perp}(X)$, $[t]$ denotes the equivalence class $\{t' \in Term_{\Sigma_\perp}(X) \mid t \approx_{\mathcal{C}} t'\}$;
- Type universe: Let $A = \{\alpha \in TVar \mid \alpha \textit{ occurs in } V\}$ and $T_{TC}(A) = \{\tau \in T_{TC}(TVar) \mid tvar(\tau) \subseteq A\}$. Then the type universe of $\mathcal{M}_{\mathcal{P}}(V)$ is $T_{TC}(A)$;
- For all $[t] \in Term_{\Sigma_\perp}(X)/\approx_{\mathcal{C}}$, $\tau \in T_{TC}(A)$, we define $[t] :^{\mathcal{M}_{\mathcal{P}}(V)} \tau$ iff $t \in Term^{\tau}_{\Sigma_\perp}(V)$.

- For all $C \in TC^n$ and $\tau_1, \ldots, \tau_n \in T_{TC}(A)$: $C^{\mathcal{M}_\mathcal{P}(V)}(\tau_1, \ldots, \tau_n) = C(\tau_1, \ldots, \tau_n)$;
- For all $c : (\tau_1, \ldots, \tau_n) \to \tau \in DC$, $[t_i] \in Term_{\Sigma_\perp}(X)/_{\approx_\mathcal{C}}$:
$$c^{\mathcal{M}_\mathcal{P}(V)}([t_1], \ldots, [t_n]) = \langle [c(t_1, \ldots, t_n)] \rangle$$
- For all $f : (\tau_1, \ldots, \tau_n) \to \tau \in FS$, $[t_i] \in Term_{\Sigma_\perp}(X)/_{\approx_\mathcal{C}}$:
$$f^{\mathcal{M}_\mathcal{P}(V)}([t_1], \ldots, [t_n]) = \{[t] \in Term_{\Sigma_\perp}(X)/_{\approx_\mathcal{C}} \mid f(t_1, \ldots, t_n) \to_\mathcal{P} t\}$$
- $\perp^{\mathcal{M}_\mathcal{P}(V)} = [\perp]$ is the bottom element, whereas the partial order is defined as follows: for all $[s], [t] \in Term_{\Sigma_\perp}(X)/_{\approx_\mathcal{C}}$, $[s] \sqsupseteq^{\mathcal{M}_\mathcal{P}(V)} [t]$ iff $s \sqsupseteq_\mathcal{C} t$. □

It can be proved that for any program $\mathcal{P} = \langle \Sigma, \mathcal{C}, \mathcal{R} \rangle$ s.t. $\mathcal{C}$ is strongly regular and well-typed, $\mathcal{M}_\mathcal{P}(V)$ is a PT-algebra. Moreover, if all rules in $\mathcal{R}$ are regular and well-typed then $\mathcal{M}_\mathcal{P}(V)$ is a well-typed PT-algebra.

All valuations over the term algebra $\mathcal{M}_\mathcal{P}(V)$ can be represented by means of substitutions. Any substitution $\rho = (\theta, \delta)$ s.t. $\delta : DVar \to Term_{\Sigma_\perp}(X)$ and $\theta : TVar \to T_{TC}(A)$, represents the valuation $[\rho] = (\theta, [\delta])$, where $[\delta](x) = [\delta(x)]$. It is easy to check that $[\![\tau]\!]^{\mathcal{M}_\mathcal{P}(V)}\theta = \tau\theta$ for all $\tau \in T_{TC}(TVar)$, and $[\![t]\!]^{\mathcal{M}_\mathcal{P}(V)}[\delta] = \langle [t\delta] \rangle$ for all $t \in Term_{\Sigma_\perp}(DVar)$. Moreover, the relationship between semantic validity in $\mathcal{M}_\mathcal{P}(V)$ and $GORC$-derivability (which allows to prove the adequateness theorem below) can be characterized as follows:

Lemma 7. (Characterization lemma) *Let $\mathcal{P} = \langle \Sigma, \mathcal{C}, \mathcal{R} \rangle$ be a program where $\mathcal{C}$ is strongly regular and well-typed. Consider $[\rho] = (\theta, [\delta]) \in Val(\mathcal{M}_\mathcal{P}(V))$, represented by a substitution $\rho = (\theta, \delta)$. Then for all $e, a, b \in Expr_{\Sigma_\perp}(DVar)$, $t \in Term_{\Sigma_\perp}(X)$:*

$[t] \in [\![e]\!]^{\mathcal{M}_\mathcal{P}(V)}[\delta]$ *iff* $e\delta \to_\mathcal{P} t$ *and* $(\mathcal{M}_\mathcal{P}(V), [\delta]) \models a \bowtie b$ *iff* $a\delta \bowtie_\mathcal{P} b\delta$. ■

Theorem 8. (Adequateness of $\mathcal{M}_\mathcal{P}(V)$) *Let $\mathcal{P} = \langle \Sigma, \mathcal{C}, \mathcal{R} \rangle$ be a program such that $\mathcal{C}$ is strongly regular and well-typed. Then:*

(1) $\mathcal{M}_\mathcal{P}(V) \models \mathcal{P}$.

(2) For any $\varphi = e \to t$ or $\varphi = e \bowtie e'$, where $e, e' \in Expr_{\Sigma_\perp}(X)$ and $t \in Term_{\Sigma_\perp}(X)$, the following statements are equivalent:

(2.1) φ is derivable in GORC (or equivalently, in BRC);

(2.2) $(\mathcal{A}, \eta) \models \varphi$, for all PT-algebra $\mathcal{A}$ such that $\mathcal{A} \models \mathcal{P}$ and for all totally defined $\xi = (\mu, \eta) \in Val(\mathcal{A})$;

(2.3) $(\mathcal{M}_\mathcal{P}(V), [id]) \models \varphi$, where id is the identity partial data substitution defined as $id(x) = [x]$, for all $x \in X$. ■

To conclude, we show that $\mathcal{M}_\mathcal{P}(V)$ admits a categorical characterization as a free object. To this end, suitable morphisms are needed.

Definition 9. (Homomorphism) Let $\mathcal{A}$ and $\mathcal{B}$ be two PT-algebras. A *homomorphism* $h : \mathcal{A} \to \mathcal{B}$ is any pair of mappings (h_0, h_1), where $h_0 : T^\mathcal{A} \to T^\mathcal{B}$ and $h_1 \in [D^\mathcal{A} \to_d D^\mathcal{B}]$ which satisfies the following conditions:

- For all $C \in TC^n, \ell_1, \ldots \ell_n \in T^\mathcal{A}$, $h_0(C^\mathcal{A}(\ell_1, \ldots \ell_n)) = C^\mathcal{B}(h_0(\ell_1), \ldots, h_0(\ell_n))$;
- For all $u \in D^\mathcal{A}$, there is $v \in D^\mathcal{B}$ such that $h_1(u) = \langle v \rangle$;
- h_1 is strict, i.e. $h_1(\perp^\mathcal{A}) = \langle \perp^\mathcal{B} \rangle$;
- For all $c : \bar{\tau} \to \tau \in DC$, $u_i \in D^\mathcal{A}$: $h_1(c^\mathcal{A}(u_1, \ldots, u_n)) = c^\mathcal{B}(h_1(u_1), \ldots, h_1(u_n))$;
- For all $f : \bar{\tau} \to \tau \in FS$, $u_i \in D^\mathcal{A}$: $h_1(f^\mathcal{A}(u_1, \ldots, u_n)) \subseteq f^\mathcal{B}(h_1(u_1), \ldots, h_1(u_n))$.

Morover, h is called a *well-typed* homomorphism iff $h_1(\mathcal{E}^{\mathcal{A}}(\ell)) \subseteq \mathcal{E}^{\mathcal{B}}(h_0(\ell))$ for all $\ell \in T^{\mathcal{A}}$. □

PT-algebras of signature Σ are the objects of a category $PTAlg_\Sigma$ whose arrows are the homomorphisms from Definition 9. The models of any given program $\mathcal{P} = \langle \Sigma, \mathcal{C}, \mathcal{R} \rangle$ determine a full subcategory $Mod_{\mathcal{P}}$ of $PTAlg_\Sigma$. We can prove:

Theorem 10. ($\mathcal{M}_{\mathcal{P}}(V)$ is free) *Let $\mathcal{P} = \langle \Sigma, \mathcal{C}, \mathcal{R} \rangle$ be a program s.t. $\mathcal{C}$ is strongly regular and well-typed. $\mathcal{M}_{\mathcal{P}}(V)$ is freely generated by V in $Mod_{\mathcal{P}}$, that is, given any $\mathcal{A} \models \mathcal{P}$ and any totally defined $\xi = (\mu, \eta) \in Val(\mathcal{A})$, there exists a unique homomorphism $h : \mathcal{M}_{\mathcal{P}}(V) \to \mathcal{A}$ extending ξ, i.e. such that $h_0(\alpha) = \mu(\alpha)$, for all $\alpha \in A$ and $h_1([x]) = \langle \eta(x) \rangle$, for all $x \in X$. Moreover, if $\mathcal{A}$ and ξ are well-typed then h is a well-typed homomorphism.* ■

6 Conclusions and Future Work

We have presented a semantic framework for functional logic programming with *algebraic polymorphic datatypes*, whose data constructors can be governed by a specified set of equational axioms. Since equational logic does not reflect the expected behaviour of lazy functions, we have given rewriting calculi and models which provide an adequate declarative semantics for our programs. This is shown by the existence of free models for programs (Theorem 10), the adequateness of the rewriting calculi w.r.t. models (Theorem 8), and type preservation results (Theorems 2, 4 and 10).

Related works dealing with non-free data constructors in declarative programming languages include [13, 4, 8, 15]. The main novelty here has been to include polymorphic data types and lazy (possibly non-deterministic) defined functions. The combination of algebraic constructors and lazy defined functions precludes a direct use of equational reasoning to deal with the equational theories for constructors. This problem has been discussed and solved in sections 3 and 4. Related work includes also some approaches to functional logic programming with polymorphic types such as [9, 1], using free constructors and more complicated algebras with one carrier for each type and multiple interpretations for polymorphic function symbols. The language in [1] is more expressive in an orthogonal direction, since it supports inclusion polymorphism.

The development of a constructor-based *lazy narrowing calculus* for goal solving has been left outside the scope of this paper. It is an important problem for future research, whose solution will presumably combine known techniques for E-unification [14, 2] with known lazy narrowing calculi for functional logic programming [7]. Another open problem is to obtain more general type preservation results, so that collapsing regular axioms for constructors and extra variables in the right-hand sides of defining rules can be allowed in programs. Last but not least, we are interested in enriching our framework with constraints, coming from a constraint system given as a suitable extension of the equational axioms $\mathcal{C}$ for constructors. For instance, if $\mathcal{C}$ specifies constructors for sets or multisets, the constraint system should provide constraints for disequality, membership, etc. In fact, set constraints are already in use,with various semantics, in different approaches to programming and program analysis [4, 5, 11].

Acknowledgments: We are indebted to Ana Gil-Luezas for her wise advices and comments to the development of this work.

References

1. Almendros-Jiménez J.M., Gavilanes-Franco A., Gil-Luezas A.: *Algebraic Semantics for Functional Logic Programming with Polymorphic Order-Sorted Types.* In Proc. ALP'96. Springer LNCS 1139, pp. 299–313, 1996.
2. Arenas-Sánchez P., Dovier A.: *Minimal Set Unification.* In Proc. PLILP'95. Springer LNCS 982, pp. 397–414, 1995.
3. Derschowitz N., Jouannaud J.P.: *Rewrite Systems.* In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Vol. B, Chapter 6. Elsevier North-Holland, 1990.
4. Dovier A., Rossi G.: *Embedding Extensional Finite Sets in CLP.* In Proc. ILPS'93, the MIT Press, pp. 540–556, 1993.
5. Gervet C.: *Conjunto: Constraint Logic Programming with Finite Set Domains.* In Proc. ILPS'94, the MIT Press, pp. 339–358, 1994.
6. Giovannetti G., Levi G., Moiso C.,Palamidessi C.: *Kernel K-LEAF: A Logic plus Functional Language.* JCSS 42 (2), pp. 139–185, 1991.
7. González-Moreno J.C., Hortalá-González T., López-Fraguas F.J, Rodríguez-Artalejo M.: *A Rewriting Logic for Declarative Programming.* In Proc. ESOP'96, Springer LNCS 1058, pp. 156–172, 1996. Full version available as TR DIA95/10, http://mozart.mat.ucm.es/papers/1996/full-esop96.ps.gz
8. Große G., Hölldobler J., Schneeberger J., Sigmund U., Thielscher M.: *Equational Logic Programming, Actions, and Change.* In Proc. ICLP'92, the MIT Press, pp. 177–191, 1992.
9. Hanus M.: *A Functional and Logic Language with Polymorphic Types (Extended Abstract).* In Proc. Int. Symposium on Design and Implementation of Symbolic Computation Systems, Springer LNCS 429, pp.215–224, 1990.
10. Hanus M.: *The Integration of Functions into Logic Programming.* A Survey. JLP (19:20). Special issue *Ten Years of Logic Programming*, pp. 583–628, 1994.
11. Heintze N., Jaffar J.: *Set Constraints and Set-Based Analysis.* In Proc. PPCP'94, Springer LNCS 874, pp. 281–298, 1994.
12. Hussmann H.: *Non-determinism Algebraic Specifications and Nonconfluent Term Rewriting.* JLP 12, pp. 237–255, 1992.
13. Jayaraman B., Plaisted D.A.: *Programming with Equations, Subsets, and Relations.* In Proc. ICLP'89, Vol. 2, the MIT Press, pp. 1051–1068, 1989.
14. Jouannaud J.P., Kirchner C.: *Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification.* Computational Logic: Essays in Honor of Alan Robinson. J.L. Lassez and G. Plotkin (Eds.). The MIT Press, pp. 257–321, 1991.
15. Meseguer J.: *A Logical Theory of Concurrent Objects and Its Realization in the Maude Language.* In Agha A., Wegner P. and Yonezawa A. (Eds), Research Directions in Concurrent Object-Oriented Programming, the MIT Press, 1993.
16. Möller B.: *On theAlgebraic Specification of Infinite Objects - Ordered and Continuous Models of Algebraic Types.* Acta Informatica 22, pp. 537–578, 1985.
17. Moreno-Navarro J.J., Rodríguez-Artalejo M.: *Logic Programming with Functions and Predicates: The Language BABEL.* JLP 12, pp. 191–223, 1992.
18. Smolka G.: *Logic Programming over Polymorphically Order-Sorted Types.* PhD Thesis, Fachbereich Informatik, Universität Kaiserslautern, 1989.
19. Reddy U.: *Narrowing as the Operational Semantics of Functional Languages.* In Proc. IEE Symposium on Logic Programming, pp. 138–151, 1985.
20. Scott D.S.: *Domains for Denotational Semantics.* In Proc. ICALP'82. SpringerLNCS 140, pp. 567–613, 1982.
21. Peterson J., Hammond K. (eds.): *Report on the Programming Language Haskell. A Non-strict, Purely Functional Language.* Version 1.3., May 1, 1996.