A Higher Order Rewriting Logic for Functional Logic Programming

J. Carlos González-Moreno, M. Teresa Hortalá-González and
Mario Rodríguez-Artalejo

DIA, Universidad Complutense de Madrid
Av. Complutense s/n, Madrid (SPAIN), E-28040
Email:jcmoreno,teresa,mario@mozart.mat.ucm.es

Abstract

According to a well known conception, programs in a declarative programming language can be viewed as theories in some suitable logic, while computations can be viewed as deductions. In our opinion, there is yet no general assent on the logic to be viewed as the foundation of higher order, lazy functional logic languages. In this paper, we argue that a specific *rewriting logic* can play this role, and we justify the adequacy of our proposal by means of proof-theoretical and model-theoretical results. Moreover, we present a sound and complete lazy narrowing calculus for goal solving, and we discuss a circuit synthesis problem that illustrates the expressiveness of our approach. This example has been tested in an implemented system.
KEYWORDS: Functional logic programming, non-deterministic functions, higher-order rewriting logic, lazy narrowing.

1 Introduction

The interest in multiparadigm declarative programming has grown over the last two decades, giving rise to different approaches to the integration of functions into logic programming ([9]). In particular, some *lazy functional logic languages* such as K-LEAF [5] 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 [21]) as a goal solving mechanism. To model the proper behaviour of partial non-strict functions, lazy functional logic languages use *constructor-based rewrite rules* and so-called *strict equality*, which considers an equality to hold iff its two sides can be reduced to a common *data term*, involving only constructors.

It turns out that classical equational logic does not supply an adequate semantics for such languages, since the equivalence between two terms which are intended to denote the same infinite data structure may not be captured by equational reasoning. Recently, a *constructor based rewriting logic* has been proposed as an alternative semantic framework for first-order lazy functional logic languages ([6]). This approach includes rewriting calculi, a model-theoretic semantics and a complete lazy narrowing calculus for goal solving, whose completeness requires neither termination nor confluence of the rewrite system acting as a program. Since confluence is not compulsory, expressivity becomes enriched by the possibility of programming with *non-deterministic functions*, in the spirit of [1, 11].

le X zero	$\to$	true	while P []	$\to$	[]
le zero (s X)	$\to$	false	while P [X | Xs]	$\to$	[] $\Leftarrow$ P X $\bowtie$ false
le (s X) (s Y)	$\to$	le X Y	while P [X | Xs]	$\to$	[X | while P Xs] $\Leftarrow$ P X $\bowtie$ true
			iterate F X	$\to$	[X | iterate F (F X)]

Figure 1: A simple example

In the present paper we extend the approach from [6] to a higher-order setting. As in some other approaches to higher-order functional logic programming ([7, 18]), we use *applicative rewrite rules* to gain most of the expressivity of higher-order functions, but avoiding λ-abstractions and higher-order unification. As in [6], we allow lazy, possibly partial and/or non-deterministic functions, and to formulate conditions and goals, we replace strict equality by the more general notion of *joinability*: two expressions a, b are regarded as joinable (in symbols $\mathsf{a} \bowtie \mathsf{b}$) iff they can be reduced to some common *pattern*, where patterns are defined as a higher-order generalization of data terms. For example, a program in our framework can include the rewrite rules shown in Figure 1. Given such a program, we can consider the goal: $\mathsf{while\ P\ (iterate\ s\ zero)} \bowtie \mathsf{[zero,\ s\ zero]}$ for wich higher-order narrowing is expected to get $\mathsf{P} = \mathsf{le\ (s\ zero)}$, as a computed answer. Note the *partial application* of the defined function symbol le to a number of arguments less than that occurring in the left-hand sides of its rewrite rules. In our framework, all such partial application are going to behave as the application of free constructors.

Other recent approaches to functional logic programming using higher-order narrowing include [7, 18, 20, 10]. Our use of a rewriting logic with a model theory and without confluence/termination requirements is a novel point w.r.t. all these papers. Another improvement w.r.t. [7] can be found in the fact that the narrowing calculus in [7] was not lazy, and much less suitable as basis for building implementations. The approach in [18] uses applicative orthogonal rewrite systems, while our applicative rewrite systems are pattern-based and conditional. In [20, 10] higher-order features are more powerful in the sense that simply-typed λ-terms and higher-order unification modulo $\beta\eta$ are allowed. Moreover [10] establishes optimality results of so-called *needed* higher-order narrowing for inductively sequential rewrite systems (a particular kind of unconditional systems). Both [18] and [20, 10] consider strict equality between first order terms, which is treated as a boolean function defined by rewrite rules. Our view of joinability conditions as a built-in construction is more expressive; in addition, joinability between higher-order patterns is allowed in our framework.

The rest of the paper is organized as follows. After this Introduction, Section 2 recalls some technical preliminaries. In Section 3 we define a higher-order rewriting logic, showing two equivalent rewriting calculi and summarizing some model-theoretic results. Section 4 presents a sound and complete lazy narrowing calculus for goal solving. In Section 5 we discuss a circuit synthesis problem that we have used to perform some experiments with an implemented system. Section 6 concludes. Proofs have been omitted because of lack of space. They will appear in [8].

2 Preliminaries

We fix here some basic notions, terminology and notations, needed for the rest of the paper. A *poset* with bottom is a set S equipped with a partial order $\sqsubseteq$ and a least element $\perp$ (w.r.t. $\sqsubseteq$). We say that an element $\mathsf{x} \in \mathsf{S}$ is *totally defined* (written $\mathsf{def}(\mathsf{x})$) iff x is maximal w.r.t. $\sqsubseteq$. The set of all totally defined elements of S will be noted $\mathsf{Def}(\mathsf{S})$. $\mathsf{D} \subseteq \mathsf{S}$ is a *directed set* iff for all $\mathsf{x},\ \mathsf{y} \in \mathsf{D}$ there exists $\mathsf{z} \in \mathsf{D}$ with $\mathsf{x} \sqsubseteq \mathsf{z},\ \mathsf{y} \sqsubseteq \mathsf{z}$. A subset $\mathsf{A} \subseteq \mathsf{S}$ is a *cone* iff $\perp \in \mathsf{A}$, and A is *downwards closed*, i.e., $\mathsf{y} \sqsubseteq \mathsf{x} \Rightarrow \mathsf{y} \in \mathsf{A}$, for all $\mathsf{x} \in \mathsf{A}, \mathsf{y} \in \mathsf{S}$. An *ideal* $\mathsf{I} \subseteq \mathsf{S}$ is a directed cone. We write $\mathcal{C}(\mathsf{S}), \mathcal{I}(\mathsf{S})$ for the sets of cones and ideals of S respectively. The set $\bar{\mathsf{S}} =_{def} \mathcal{I}(\mathsf{S})$, equipped with the set-inclusion $\subseteq$ as ordering, is a poset with bottom called the *ideal completion* of S. There is a natural, order-preserving embedding of S into $\bar{\mathsf{S}}$, which maps each $\mathsf{x} \in \mathsf{S}$ into the principal ideal generated by x, $< \mathsf{x} > =_{def} \{y \in \mathsf{S} : \mathsf{y} \sqsubseteq \mathsf{x}\} \in \bar{\mathsf{S}}$. A poset with bottom C is a *cpo* iff D has a least upper bound $\sqcup \mathsf{D}$ (also called *limit*) for every directed set $\mathsf{D} \subseteq \mathsf{C}$. $\mathsf{u} \in \mathsf{C}$ is a *finite* element (written $\mathsf{fin}(\mathsf{u})$) if whenever $\mathsf{u} \sqsubseteq \sqcup\ \mathsf{D}$ for a directed $\mathsf{D} \subseteq \mathsf{C}$, there exists $\mathsf{x} \in \mathsf{D}$ with $\mathsf{u} \sqsubseteq \mathsf{x}$. A cpo C is called *algebraic* if any element of C is the limit of a directed set of finite elements. For any poset with bottom S, its ideal completion $\bar{\mathsf{S}}$ turns to be the least cpo including S. Furthermore, $\bar{\mathsf{S}}$ is an algebraic cpo whose finite elements are precisely the principal ideals $< \mathsf{x} >, \mathsf{x} \in \mathsf{S}$; see e.g. [16]. Note that elements $\mathsf{x} \in \mathsf{Def}(\mathsf{S})$ correspond to finite and maximal elements $< \mathsf{x} >$ in the ideal completion.

A *signature with constructors* is a countable set $\Sigma = \mathsf{DC}_\Sigma \cup \mathsf{FS}_\Sigma$, where $\mathsf{DC}_\Sigma = \bigcup_{n \in \mathbb{N}} \mathsf{DC}^n_\Sigma$ and $\mathsf{FS}_\Sigma = \bigcup_{n \in \mathbb{N}} \mathsf{FS}^n_\Sigma$ are disjoint sets of *constructors* and *defined function symbols* respectively, each of them with associated *arity*. We write $\mathsf{ar}(\mathsf{c}) = \mathsf{n}$ resp. $\mathsf{ar}(\mathsf{f}) = \mathsf{n}$ to indicate the arities of constructors resp. function symbols. We assume a countable set $\mathcal{V}$ of *variables*, and we omit explicit mention of Σ in the sequel.

We write Exp for the set of expressions built up with aid of Σ and $\mathcal{V}$ and defined as:

$$\begin{aligned} \mathsf{Exp} ::= \mathsf{X}\ \%\ \mathsf{X} \in \mathcal{V}\ &|\ \mathsf{h}\ \%\ \mathsf{h} \in \mathsf{DC} \bigcup \mathsf{FS} \\ &|\ (\mathsf{e}\ \mathsf{e}'\ )\ \%\ \mathsf{e},\ \mathsf{e}' \in \mathsf{Exp} \end{aligned}$$

Expression $(\mathsf{e}\ \mathsf{e}')$ stands for the application of e (acting as a function) to e' (acting as an argument). As usual, we assume that application associates to the left. Therefore, $(\mathsf{e}_0\ \mathsf{e}_1 \ldots \mathsf{e}_n)$ abbreviates $((\ldots(\mathsf{e}_0\ \mathsf{e}_1) \ldots)\ \mathsf{e}_n)$. We distinguish an important kind of expressions called *patterns* ($\mathsf{Pat} \subseteq \mathsf{Exp}$) defined as:

$$\begin{aligned} \mathsf{Pat} ::= \mathsf{X}\ \%\ \mathsf{X} \in \mathcal{V}\ &|\ \mathsf{c}\ \mathsf{t}_1 \ldots \mathsf{t}_m\ \%\ \mathsf{t}_i \in \mathsf{Pat}\ \mathsf{and}\ \mathsf{c} \in \mathsf{DC}^n,,\ 0 \leq m \leq n \\ &|\ \mathsf{f}\ \mathsf{t}_1 \ldots \mathsf{t}_m\ \%\ \mathsf{t}_i \in \mathsf{Pat}\ \mathsf{and}\ \mathsf{f} \in \mathsf{FS}^n,,\ 0 \leq m < n \end{aligned}$$

For some purposes we need to enhance Σ with a new constant (0-arity constructor) $\perp$, obtaining a new signature $\Sigma_\perp$. We will write $\mathsf{Exp}_\perp$ and $\mathsf{Pat}_\perp$ for the corresponding sets of expressions and patterns in this extended signature, the so-called *partial* expressions and *partial* patterns respectively. Expressions and patterns without occurrences of $\perp$ are called *total*. As frequent notational conventions we will also use $\mathsf{c},\mathsf{d} \in \mathsf{DC}$; $\mathsf{f},\mathsf{g} \in \mathsf{FS}$; $\mathsf{s},\mathsf{t} \in \mathsf{Pat}_\perp$; $\mathsf{a},\mathsf{b},\mathsf{e} \in \mathsf{Exp}_\perp$.

In the rest of the paper we assume the following classification of expressions: $\mathsf{X}\ \mathsf{e}_1 \ldots \mathsf{e}_m$, with $\mathsf{X} \in \mathcal{V}$ and $m \geq 0$, is called a *flexible expression*, while $\mathsf{h}\ \mathsf{e}_1 \ldots \mathsf{e}_m$ with $\mathsf{h} \in \mathsf{DC}^n \cup \mathsf{FS}^n$ is called a *rigid expression*. Moreover, a rigid expression is called *junk* iff $\mathsf{h} \in \mathsf{DC}^n$ and $m > n$, *active* iff $\mathsf{h} \in \mathsf{FS}^n$ and $m \geq n$, and *passive* otherwise. The intuition behind this classification is the following one: leftmost outermost reduction makes sense only for active expressions; junk expressions have no sensible meaning. These ideas will reflect in subsequent formal definitions.

A natural *approximation ordering* " $\sqsubseteq$ " for partial patterns can be defined as follows: $\sqsubseteq$ is the least partial ordering over $\mathsf{Pat}_\perp$ satisfying the following properties: $\perp \sqsubseteq \mathsf{t}$, for all $\mathsf{t} \in \mathsf{Pat}_\perp$; $X \sqsubseteq X$, for all $X \in \mathcal{V}$; and $\mathsf{h}\ \mathsf{t}_1 \ldots \mathsf{t}_m \sqsubseteq \mathsf{h}\ \mathsf{s}_1 \ldots \mathsf{s}_m$ whenever these two expressions are *rigid patterns* and $\mathsf{t}_i \sqsubseteq \mathsf{s}_i$ for all $1 \leq i \leq m$.

Substitutions are mappings $\theta : \mathcal{V} \rightarrow \mathsf{Pat}$ which have a unique natural extension $\hat{\theta}{:}\mathsf{Exp} \rightarrow \mathsf{Exp}$, also noted as θ. The set of all substitutions is noted as Subst. The bigger set $\mathsf{Subst}_\perp$ of all *partial substitutions* $\theta : \mathcal{V} \rightarrow \mathsf{Pat}_\perp$ is defined analogously. We note as $\mathsf{e}\theta$ the result of applying the substitution θ to the expression e, and we define the composition $\sigma\theta$ such that $\mathsf{e}(\sigma\theta) \equiv (\mathsf{e}\sigma)\theta$. As usual, $\theta = \{\ X_1\ /\ \mathsf{t}_1, \ldots, X_n\ /\ \mathsf{t}_n\ \}$ stands for the substitution that satisfies $X_i\theta \equiv \mathsf{t}_i$ $(1 \leq i \leq n)$ and $\mathsf{Y}\theta \equiv \mathsf{Y}$ for all $Y \in \mathcal{V} \setminus \{X_1, \ldots, X_n\}$. A substitution θ such that $\theta\theta = \theta$ is called *idempotent*. The approximation ordering over $\mathsf{Pat}_\perp$ induces a natural approximation ordering over $\mathsf{Subst}_\perp$, defined by the condition: $\theta \sqsubseteq \theta'$ iff $X\theta \sqsubseteq X\theta'$, for all $X \in \mathcal{V}$. We will use also the *subsumption ordering* over $\mathsf{Subst}_\perp$, defined by: $\theta \leq \theta'$ iff $\theta' = \theta\sigma$ for some σ. Finally, the notation $\theta \leq \theta'[\mathcal{U}]$, where $\mathcal{U} \subseteq \mathcal{V}$, means that $X\theta \equiv X(\theta\sigma)$ for some σ and for all $X \in \mathcal{U}$ (i.e, θ is more general than θ' over the variables in $\mathcal{U}$).

3 A Higher Order Rewriting Logic

In this section we extend the Constructor Based Conditional Rewriting Logic (in what follows $CRWL$) from [6], in order to deal with applicative rewrite rules. In contrast to Meseguer's rewriting logic [14], which aims at modelling change caused by concurrent actions at a very high abstraction level, our rewriting logic intends to model the evaluation of expressions in a constructor-based language involving lazy functions. As in [6], we do not impose non-ambiguity conditions. This means that non-deterministic functions are allowed. The higher order extension $HOCRWL$ of $CRWL$ is designed to derive the following kinds of statements: *Reduction statements* $\mathsf{e} \rightarrow \mathsf{e}'$, whose intended meaning is that e can be reduced to e'. *Approximation statements* $\mathsf{e} \rightarrow \mathsf{t}$, whose intended meaning is that the possibly partial pattern t approximates the denotation of the expression e. *Joinability statements* $\mathsf{a} \bowtie \mathsf{b}$, which hold iff $\mathsf{a} \rightarrow \mathsf{t}$ and $\mathsf{b} \rightarrow \mathsf{t}$ can be derived for some *total pattern* t (with no occurrences of $\perp$).

We allow for partial patterns in $HOCRWL$-statements because we want to model the behaviour of non-strict functions which may work with potentially infinite data structures. $HOCRWL$-theories, which will be called simply *programs* in the rest of the paper, are defined as sets $\mathcal{R}$ of conditional rewrite rules of the form:

$$\underbrace{\mathsf{f}\ \mathsf{t}_1 \ldots \mathsf{t}_n}_{\textit{l}\text{eft }\textit{h}\text{and }\textit{s}\text{ide (l)}} \rightarrow \underbrace{\mathsf{r}}_{\textit{r}\text{ight }\textit{h}\text{and }\textit{s}\text{ide}} \Leftarrow \underbrace{\mathsf{C}}_{\text{Condition}}$$

where $\mathsf{f} \in \mathsf{FS}^n$, l must be linear, t_i must be patterns (this allows us the use of partial applications of function symbols in left hand sides) and the condition C must consist of finitely many (possibly zero) joinability statements $\mathsf{a} \bowtie \mathsf{b}$ where $\mathsf{r},\ \mathsf{a},\ \mathsf{b}$ are expressions. In the sequel we use the following notation for possibly partial instances of rewrite rules:

$$[\mathcal{R}]_\perp = \{\ (\mathsf{l} \rightarrow \mathsf{r} \Leftarrow \mathsf{C})\ \theta \mid (\mathsf{l} \rightarrow \mathsf{r} \Leftarrow \mathsf{C}) \in \mathcal{R},\ \theta \in \mathsf{Subst}_\perp\ \}$$

Note that joinabilty statements are the natural generalization of strict equations in a non-deterministic setting. Using $HOCRWL$-derivability, we can define the denotation $[\![\mathsf{e}]\!]^{\mathcal{R}}$ of an expression e w.r.t. a program $\mathcal{R}$ as the set of approximations $\{\mathsf{t} \mid \mathcal{R} \vdash_{\text{HOCRWL}} \mathsf{e} \rightarrow \mathsf{t}\}$. In fact, we can characterize $HOCRWL$-derivability $\vdash_{\text{HOCRWL}}$ by means of two different rewriting calculi. The first one, called Higher Order Basic Rewriting Calculus ($HOBRC$), is given by the following inference rules:

B$_0$**:** $\mathsf{e} \rightarrow \perp$ ***RF:*** $\mathsf{e} \rightarrow \mathsf{e}$ ***TR:*** $\dfrac{\mathsf{e} \rightarrow \mathsf{e}' \quad \mathsf{e}' \rightarrow \mathsf{e}''}{\mathsf{e} \rightarrow \mathsf{e}''}$

MN: $\dfrac{\mathsf{e}_0 \rightarrow \mathsf{e}'_0 \quad \mathsf{e}_1 \rightarrow \mathsf{e}'_1}{(\mathsf{e}_0\ \mathsf{e}_1) \rightarrow (\mathsf{e}'_0\ \mathsf{e}'_1)}$ ***R :*** $\dfrac{\mathsf{C}}{\mathsf{l} \rightarrow \mathsf{r}}$ for $(\mathsf{l} \rightarrow \mathsf{r} \Leftarrow \mathsf{C}) \in [\mathcal{R}]_\perp$

J : $\dfrac{\mathsf{a} \rightarrow \mathsf{t} \quad \mathsf{b} \rightarrow \mathsf{t}}{\mathsf{a} \bowtie \mathsf{b}}$ if t is a total pattern.

Note that $HOCRWL$-reduction is related to the idea of approximation, as shown by rule ***B***$_0$. In rule ***J***, the pattern t must be total, since we wish to specify joinability as a generalization of strict equality, and total patterns in the higher-order framework play the same role as total constructor terms in the first-order framework; see [6]. In rule ***R*** we use rule instances from $[\mathcal{R}]_\perp$ to reflect the so-called "*call-time-choice*" for non-determinism; this makes a difference w.r.t. Boudol's semantics for left-linear term rewrite systems [2]. See the "coin example" in [6], Section 3.

Following the spirit of the *uniform proofs* approach to logic programming [15], we introduce also a Goal-oriented Rewriting Calculus ($HOGRC$) whose proofs are closer to the goal-solving calculus which will be presented in Section 4.

B : $\mathsf{e} \rightarrow \perp$ ***RR:*** $\mathsf{X} \rightarrow \mathsf{X}$, if $\mathsf{X} \in \mathcal{V}$

OR: $\dfrac{\mathsf{e}_1 \rightarrow \mathsf{t}_1 \ldots \mathsf{e}_n \rightarrow \mathsf{t}_n \quad \mathsf{C} \quad \mathsf{r}\ \mathsf{a}_1 \ldots \mathsf{a}_m \rightarrow \mathsf{t}}{\mathsf{f}\ \mathsf{e}_1 \ldots \mathsf{e}_n\ \mathsf{a}_1 \ldots \mathsf{a}_m \rightarrow \mathsf{t}}$,
if $\mathsf{t} \not\equiv \perp$ is a pattern, $m \geq 0$, and $\mathsf{f}\ \mathsf{t}_1 \ldots \mathsf{t}_n \rightarrow \mathsf{r} \Leftarrow \mathsf{C} \in [\mathcal{R}]_\perp$

DC: $\dfrac{\mathsf{e}_1 \rightarrow \mathsf{t}_1 \ldots \mathsf{e}_n \rightarrow \mathsf{t}_n}{\mathsf{h}\ \mathsf{e}_1 \ldots \mathsf{e}_n \rightarrow \mathsf{h}\ \mathsf{t}_1 \ldots \mathsf{t}_n}$
if $\mathsf{h}\ \mathsf{t}_1 \ldots \mathsf{t}_n$ is a rigid pattern

J : $\dfrac{\mathsf{a} \rightarrow \mathsf{t} \quad \mathsf{b} \rightarrow \mathsf{t}}{\mathsf{a} \bowtie \mathsf{b}}$
if t is a total pattern

By induction on the structure of proofs, we can prove:

Proposition 3.1 *(Calculi Equivalence).- For any program $\mathcal{R}$, the calculi $HOBRC$ and $HOGRC$ derive the same approximation and joinability statements.* ∎

In the rest of the paper, the notation $\mathcal{R} \vdash_{HOCRWL} \varphi$ will mean provability of φ (an approximation or joinability statement) in any of the calculi $HOBRC$ or $HOGRC$. For instance, if $\mathcal{R}$ is the program showed in Figure 1, the following statements are derivable:

$$\begin{array}{rcl} \mathsf{iterate\ s\ zero} & \rightarrow & [\mathsf{zero},\ \mathsf{s\ zero} \mid \perp] \\ \mathsf{while\ (le\ (s\ zero))\ (iterate\ s\ zero)} & \bowtie & [\mathsf{zero},\ \mathsf{s\ zero}] \end{array}$$

In addition to the proof calculi presented above, we have developped a model theory for $HOCRWL$. Models are applicative algebras similar to those used in [7], modified to account for non-determinism along the lines from [6]. We use posets as carriers, and we view non-deterministic functions as monotonic mappings from elements to cones. The poset's elements must be thought as finite approximations of possibly infinite values. By considering ideal completions ([16]), it can be shown that this approach is equivalent to working with algebraic cpo's and Hoare's powerdomains ([22]), but we will not dwell on this issue here. The technical definitions are as follows:

Definition 3.1 *(Non-deterministic and deterministic functions).- Given two posets with bottom* D, E, *we define:*

- *The set of all non-deterministic functions from* D *to* E *as:*
  $[\mathsf{D} \rightarrow_{\mathsf{n}} \mathsf{E}] =_{\mathsf{def}} \{\mathsf{f} : \mathsf{D} \rightarrow \mathcal{C}(\mathsf{E}) \mid \forall \mathsf{u}, \mathsf{u}' \in \mathsf{D} : (\mathsf{u} \sqsubseteq \mathsf{u}' \Rightarrow \mathsf{f}(\mathsf{u}) \subseteq \mathsf{f}(\mathsf{u}'))\}$.
- *The set of all deterministic functions from* D *to* E *as:*
  $[\mathsf{D} \rightarrow_{\mathsf{d}} \mathsf{E}] =_{\mathsf{def}} \{\mathsf{f} \in [\mathsf{D} \rightarrow_{\mathsf{n}} \mathsf{E}] \mid \forall \mathsf{u} \in \mathsf{D} : \mathsf{f}(\mathsf{u}) \in \mathcal{I}(\mathsf{E})\}$. □

For a fixed argument $u \in D$, a deterministic function f computes a directed set $\mathsf{f}(\mathsf{u})$ of partial values; thus, such functions become continuous mappings between algebraic cpos, after performing an ideal completion. Note also that any non-deterministic function $\mathsf{f} \in [\mathsf{D} \rightarrow_{\mathsf{n}} \mathsf{E}]$ can be extended to the monotonic mapping $\hat{\mathsf{f}} : \mathcal{C}(\mathsf{D}) \rightarrow \mathcal{C}(\mathsf{E})$ defined by $\hat{\mathsf{f}}(\mathsf{C}) =_{\mathsf{def}} \bigcup_{\mathsf{u} \in \mathsf{C}} \mathsf{f}(\mathsf{u})$. By an abuse of notation, we will note $\hat{\mathsf{f}}$ also as f in the sequel.

In [6], $CRWL$-algebras $\mathcal{A}$ interpret functions symbols $\mathsf{f} \in \mathsf{FS}^n$ as non-deterministic functions $\mathsf{f}^{\mathcal{A}} \in [\mathsf{D}^n_{\mathcal{A}} \rightarrow \mathsf{D}_{\mathcal{A}}]$. In the present higher-order setting, applicative algebras will interpret function symbols as first-class elements, and an explicit "apply" operation will be available. More formally:

Definition 3.2 *($HOCRWL$-Algebras).- Let Σ be any given signature.* $HOCRWL$-algebras *of signature Σ are algebraic structures of the form:*

$\mathcal{A} = (\mathsf{D}^{\mathcal{A}}, \mathsf{@}^{\mathcal{A}}, \{\ \mathsf{c}^{\mathcal{A}}\ \}_{c \in DC_\Sigma}, \{\ \mathsf{f}^{\mathcal{A}}\ \}_{f \in FS_\Sigma})$ *where:*

- $\mathsf{D}^{\mathcal{A}}$ *is a poset with bottom element* $\perp^{\mathcal{A}}$.
- $@^{\mathcal{A}} \in [\mathsf{D}^{\mathcal{A}} \times \mathsf{D}^{\mathcal{A}} \rightarrow_n \mathsf{D}^{\mathcal{A}}]$ *is a non-deterministic binary function (apply operation), written in infix notation such that* $\perp^{\mathcal{A}}\ \mathsf{@}^{\mathcal{A}}\ \mathsf{v} = <\perp^{\mathcal{A}}>$ *for all* $\mathsf{v} \in \mathsf{D}^{\mathcal{A}}$. *For given* $u_0, u_1, \ldots, u_k$, *we will use the notation "*$u_0\ @^{\mathcal{A}}\ u_1\ @^{\mathcal{A}} \ldots @^{\mathcal{A}}\ u_k$*" to denote a cone that is defined as:*
  $< \mathsf{u}_0 >$, *if* $k = 0$, *and* $\bigcup \{\ \mathsf{u}\ \mathsf{@}^{\mathcal{A}}\ \mathsf{u}_k\ /\ \mathsf{u} \in \mathsf{u}_0\ \mathsf{@}^{\mathcal{A}} \ldots \mathsf{@}^{\mathcal{A}}\ \mathsf{u}_{k\perp 1}\ \}$, *if* $k > 0$.
- $\mathsf{c}^{\mathcal{A}}, \mathsf{f}^{\mathcal{A}} \in \mathsf{D}^{\mathcal{A}}$, *if 0 <* ***ar***$(\mathsf{f})$, *and* $\mathsf{f}^{\mathcal{A}} \in \mathcal{C}(\mathsf{D}^{\mathcal{A}})$, *if 0 =* ***ar***$(\mathsf{f})$.
- *For all* $\mathsf{u}_1, \ldots, \mathsf{u}_k \in \mathsf{D}^{\mathcal{A}}$: $\mathsf{c}^{\mathcal{A}}\ \mathsf{@}^{\mathcal{A}}\ \mathsf{u}_1\ \mathsf{@}^{\mathcal{A}} \ldots \mathsf{@}^{\mathcal{A}}\ \mathsf{u}_k = <\perp^{\mathcal{A}}>$, *if* $k >$ ***ar***(c).
- *For all* $\mathsf{u}_1, \ldots, \mathsf{u}_k \in \mathsf{D}^{\mathcal{A}}$ *there is* $\mathsf{v} \in \mathsf{D}^{\mathcal{A}}$ *such that* $\mathsf{h}\ ^{\mathcal{A}}\mathsf{@}^{\mathcal{A}}\ \mathsf{u}_1\mathsf{@}^{\mathcal{A}} \ldots \mathsf{@}^{\mathcal{A}}\ \mathsf{u}_k = < v >$, *where* $\mathsf{h} \in \mathsf{DC}^n_\Sigma \cup \mathsf{FS}^m_\Sigma$, *and* $0 \leq \mathrm{k} \leq \mathrm{n}$, $0 \leq \mathrm{k} < \mathrm{m}$. *Moreover,* v *is maximal in case that all* u_i *are also maximal.* □

It is convenient to simplify the notation by writing $\mathsf{h}^{\mathcal{A}}\ \mathsf{u}_1\ \ldots\ \mathsf{u}_k$ in place of $\mathsf{h}^{\mathcal{A}}\ @^{\mathcal{A}}\ \mathsf{u}_1\ @^{\mathcal{A}}\ \ldots @^{\mathcal{A}}\ \mathsf{u}_k$. For $\mathsf{f} \in \mathsf{FS}^n$, we will say that $\mathsf{f}^{\mathcal{A}}$ is *deterministic* iff for all $\mathsf{u}_1\ \ldots\ \mathsf{u}_n \in \mathsf{D}^{\mathcal{A}}$ the cone $\mathsf{f}^{\mathcal{A}}\ \mathsf{u}_1\ \ldots\ \mathsf{u}_n$ is an ideal. Note that $\mathsf{c}^{\mathcal{A}}$ is always deterministic for all $\mathsf{c} \in \mathsf{DC}^n$.

A *valuation* over a structure $\mathcal{A}$ is any mapping η: $\mathcal{V} \to \mathsf{D}^{\mathcal{A}}$, and we say that η is *totally defined* iff $\eta(\mathsf{X}) \in \mathsf{Def}(\mathsf{D}^{\mathcal{A}})$ for all $\mathsf{X} \in \mathcal{V}$. We denote by $\mathsf{Val}(\mathcal{A})$ the set of all valuations, and by $\mathsf{DefVal}(\mathcal{A})$ the set of all totally defined valuations. The *evaluation* of a partial expresion $\mathsf{e} \in \mathsf{Exp}_\perp$ in $\mathcal{A}$ under η yields a cone $[\![\ \mathsf{e}\ ]\!]^{\mathcal{A}}\eta \in \mathcal{C}(\mathsf{D}^{\mathcal{A}})$ which is defined recursively as follows:

- $[\![\ \perp\ ]\!]^{\mathcal{A}}\eta =_{def} <\perp^{\mathcal{A}}>$; $[\![\ \mathsf{X}\ ]\!]^{\mathcal{A}}\eta =_{def} <\eta(\mathsf{X})>$, for $\mathsf{X} \in \mathcal{V}$.
- $[\![\ \mathsf{h}\ ]\!]^{\mathcal{A}}\eta =_{def} <\ \mathsf{h}^{\mathcal{A}}\ >$, for $\mathsf{h} \in \mathsf{DC} \cup \mathsf{FS}^n$, with $n > 0$; $[\![\ \mathsf{f}\ ]\!]^{\mathcal{A}}\eta =_{def} \mathsf{f}^{\mathcal{A}}$, for $\mathsf{f} \in \mathsf{FS}^0$.
- $[\![\ \mathsf{e}\ \mathsf{e}'\ ]\!]^{\mathcal{A}}\eta =_{def} [\![\ \mathsf{e}\ ]\!]^{\mathcal{A}}\eta\ @^{\mathcal{A}}\ [\![\ \mathsf{e}'\ ]\!]^{\mathcal{A}}\eta$.

Using $HOCRWL$-algebras, we are able to extend the model-theoretic results from [6] to a higher-order setting. In particular, our notion of model is:

Definition 3.3 *(Models).- Assume a program $\mathcal{R}$ and an $HOCRWL$-algebra $\mathcal{A}$. We define:*

- *$\mathcal{A}$ satisfies a reduction statement $\mathsf{e} \to \mathsf{e}'$ under a valuation η (in symbols, $(\mathcal{A}, \eta) \models \mathsf{e} \to \mathsf{e}'$) iff $[\![\ \mathsf{e}\ ]\!]^{\mathcal{A}}\eta \supseteq [\![\ \mathsf{e}'\ ]\!]^{\mathcal{A}}\eta$.*
- *$\mathcal{A}$ satisfies a joinability statement $a\ \bowtie\ b$ under a valuation η (in symbols, $(\mathcal{A}, \eta) \models \mathsf{a} \bowtie \mathsf{b}$) iff $[\![\ \mathsf{a}\ ]\!]^{\mathcal{A}}\eta \bigcap [\![\ \mathsf{b}\ ]\!]^{\mathcal{A}}\eta \bigcap \mathsf{Def}(\mathsf{D}^{\mathcal{A}}) \neq \emptyset$.*
- *$\mathcal{A}$ satisfies a rule $\mathsf{l} \to \mathsf{r} \Leftarrow \mathsf{C}$ iff every valuation η such that $\mathcal{A} \models C\eta$ verifies: $(\mathcal{A},\eta) \models \mathsf{l} \to \mathsf{r}$.*
- *$\mathcal{A}$ is a* model *of $\mathcal{R}$ (in symbols $\mathcal{A} \models \mathcal{R}$) iff $\mathcal{A}$ satisfies all the rules in $\mathcal{R}$.* $\square$

For each program $\mathcal{R}$, we can build a *pattern model* $\mathcal{M}_{\mathcal{R}}$ whose carrier $\mathsf{D}^{\mathcal{M}_{\mathcal{R}}}$ is the set $\mathsf{Pat}_\perp$ of all partial patterns, in such a way that the following result holds:

Theorem 3.1 *(Adequateness of $\mathcal{M}_{\mathcal{R}}$) .- $\mathcal{M}_{\mathcal{R}}$ is a model of $\mathcal{R}$. Moreover, for any approximation or joinability statement φ, the following conditions are equivalent:*

a) $\mathcal{R} \vdash_{HOCRWL} \varphi$.

b) *$(\mathcal{A}, \eta) \models \varphi$ for every $\mathcal{A} \models \mathcal{R}$, and every $\eta \in \mathsf{DefVal}(\mathcal{A})$.*

c) $(\mathcal{M}_{\mathcal{R}}, id) \models \varphi$*, where id is the identity valuation, $id(X) = X$.* ■

In particular, for all $\mathsf{f} \in \mathsf{FS}^n$ and $\mathsf{t}_1\ \ldots\ \mathsf{t}_n,\ \mathsf{t} \in \mathsf{Pat}_\perp$, $\mathsf{f}^{\mathcal{M}_{\mathcal{R}}}\ \mathsf{t}_1\ \ldots\ \mathsf{t}_n \to \mathsf{t}$ holds iff $\mathcal{R} \vdash_{HOCRWL} \mathsf{f}\ \mathsf{t}_1\ \ldots\ \mathsf{t}_n \to \mathsf{t}$. This shows that $\mathcal{M}_{\mathcal{R}}$ can be seen as a canonic semantics for $\mathcal{R}$, analogous to the $\mathcal{C}$-semantics [4] for Horn clause logic programs. In fact, it is easy to prove that any Horn clause logic program can be translated into a $HOCRWL$-program in such a way that the $\mathcal{C}$-semantics is preserved. As done in [6] for the first-order case, we can also show that $\mathcal{M}_{\mathcal{R}}$ is free in the category of all models of $\mathcal{R}$.

4 Lazy Narrowing Calculus

In this Section we present a Lazy Narrowing Calculus that allows to solve goals w.r.t. a given $HOCRWL$ program. Initial goals are simply multisets of joinability statements, but we need to allow also a more general class of goals which arise in the course of narrowing derivations. The next definition extends a similar one from [6] to the present higher-order framework:

Definition 4.1 *(Admissible goals).- An* admissible goal *for a given program $\mathcal{R}$ must have the form $G \equiv \exists\overline{U}.\ S \,\Box\, P \,\Box\, E$, where:*

- $\mathsf{evar}(G) \equiv \overline{U}$ *is the set of so-called* existential variables *of the goal G.*
- $\mathsf{S} \equiv \mathsf{X}_1 = \mathsf{s}_1, \ldots, \mathsf{X}_n = \mathsf{s}_n$ *is a set of equations, called* solved part. *Each s_i must be a total pattern, and each X_i must occur exactly once in the whole goal.* (Intuition: Each s_i is a computed answer for X_i.)
- $\mathsf{P} \equiv \mathsf{e}_1 \to \mathsf{t}_1, \ldots, \mathsf{e}_k \to \mathsf{t}_k$ *is a multiset of* approximation conditions, *with* $\mathsf{t}_i \in \mathsf{Pat}$. $\mathsf{pvar}(P) =_{def} \mathsf{var}(\mathsf{t}_1) \cup \ldots \cup \mathsf{var}(\mathsf{t}_k)$ *is called the set of* produced variables *of the goal G. The* production relation *between G-variables is defined by* $\mathsf{X} \gg_P \mathsf{Y}$ *iff there is some $1 \le i \le k$ such that* $\mathsf{X} \in \mathsf{var}(\mathsf{e}_i)$ *and* $\mathsf{Y} \in \mathsf{var}(\mathsf{t}_i)$.
  (Intuition: $\mathsf{e}_i \to \mathsf{t}_i$ demands narrowing e_i to match t_i. This may produce bindings for variables in t_i.)
- $\mathsf{E} \equiv \mathsf{a}_1 \bowtie \mathsf{b}_1, \ldots, \mathsf{a}_l \bowtie \mathsf{b}_l$ *is a multiset of* joinability conditions. $\mathsf{dvar}(E) =_{def} \{\mathsf{X}{\in}\mathcal{V} / (\mathsf{X}\ \mathsf{e}_1 \ldots \mathsf{e}_m){\equiv}\mathsf{a}_i$ or $(\mathsf{X}\ \mathsf{e}_1 \ldots \mathsf{e}_m){\equiv}\mathsf{b}_i$, for some $1 \le i \le l$, and $m \ge 0\}$ *is called the set of* demanded variables *of the goal G.*
  (Intuition: Due to the semantics of joinability, goal solving must bind demanded variables to patterns different from $\bot$.)

Additionally, any admissible goal must fulfil the following conditions:

- *The tuple* $(\mathsf{t}_1, \ldots, \mathsf{t}_k)$ *must be* linear.
  (Intuition: Each produced variable is produced only once.)
- *All the produced variables must be existential, i.e.* $\mathsf{pvar}(P) \subseteq \mathsf{evar}(G)$.
  (Intuition: Produced variables are used to compute intermediate results.)
- *The transitive closure of the production relation $\gg_P$ must be irreflexive, or equivalently, a strict partial order.*
  (Intuition: Bindings for produced variables are computed hierarchically.)
- *The solved part contains no produced variables.*
  (Intuition: The solved part includes no reference to intermediate results.) □

We assume by convention that in an *initial goal* G only the joinability part E is present, and there are no existential variables in G.

A solution for a goal G is essentially a substitution θ that renders $G\theta$ provable in the rewriting logic $HOCRWL$. The next definition allows θ to give partial patterns as bindings for produced variables, since these will be used to capture approximate denotations in intermediate stages of narrowing derivations.

Definition 4.2 *(Solutions).- Let $G \equiv \exists\overline{U}.\ S \,\Box\, P \,\Box\, E$ be an admissible goal, and θ a partial substitution.*

- *θ is* allowable *for G iff $X\theta$ is a total pattern for every $X \notin pvar(P)$.*
- *θ is a* solution *for G iff θ is allowable for G, $X_i\theta \equiv s_i\theta$ for all $(X_i = s_i) \in S$, and $(P \,\Box\, E)\theta$ has a* "witness" *$\mathcal{M}$.*
- *A witness is defined as a multiset containing a $HOGRC$-proof (see Section 3) for each condition $e\theta \to t\theta \in P\theta$ and $a\theta{\bowtie}b\theta \in E\theta$.*
- *We write $Sol(G)$ for the set of all solutions for G.* □

Due to the Adequateness Theorem 3.1, it is immediate to give a model-theoretic characterization of solutions, equivalent to the proof-theoretic definition.

We present now a *Higher Order Constructor-based Lazy Narrowing Calculus* ($HOCLNC$) for solving initial goals, obtaining *solutions* in the sense of Definition

4.2. The calculus $HOCLNC$ consists of a set of *transformation rules* for goals. Each transformation rule takes the form $G \Vdash G'$, specifying one of the possible ways of performing one step of goal solving. Derivations are sequences of $\Vdash$-steps. For writing failure rules we use $FAIL$, representing an irreducible inconsistent goal. We recall that in a goal $\exists\overline{U}.\ S \square P \square E$, S is a set while P, E are multisets. Consequently, in the transformation rules no particular selection strategy (e.g. "sequential left-to-right") is assumed for the conditions in P or E. The following notational conventions are assumed below: $\mathsf{a} \asymp \mathsf{b}$ stands for $\mathsf{a}\bowtie\mathsf{b}$ or $\mathsf{b}\bowtie\mathsf{a}$; $\overline{\mathsf{a}}_\mathsf{m}$ stands for $\mathsf{a}_1 \ldots \mathsf{a}_\mathsf{m}$; $\mathsf{h}\overline{\mathsf{a}}_\mathsf{m} \downarrow \mathsf{h}\overline{\mathsf{b}}_\mathsf{m}$ stands for $\mathsf{a}_1\bowtie\mathsf{b}_1, \ldots, \mathsf{a}_\mathsf{m}\bowtie\mathsf{b}_\mathsf{m}$ if $\mathsf{h}\overline{\mathsf{a}}_\mathsf{m},\ \mathsf{h}\overline{\mathsf{b}}_\mathsf{m}$ are rigid and passive expressions; and $svar(\mathsf{e})$ stands for the set of all variables X occurring in e at some position whose ancestor positions are all occupied by "constructors". More formally, given $\mathsf{e} \in \mathsf{Exp}$ we define its set of safe variables $svar(\mathsf{e})$ as follows: for a flexible expression $\mathsf{e} \equiv \mathsf{X}\mathsf{e}_1 \ldots \mathsf{e}_\mathsf{m}$, $svar(\mathsf{e}) = \{\mathsf{X}\}$ if $m = 0$; for a rigid and passive expression $\mathsf{e} \equiv \mathsf{h}\mathsf{e}_1 \ldots \mathsf{e}_\mathsf{m}$, $svar(\mathsf{e}) = \bigcup_{i=1}^{m} svar(\mathsf{e}_\mathsf{i})$; in any other case, $svar(\mathsf{e}) = \emptyset$.

The $HOCLNC$ calculus

Rules for $\bowtie$

Identity: (ID) $\exists\overline{U}.\ S \square P \square X\bowtie X, E \Vdash \exists\overline{U}.\ S \square P \square E$ if $X \notin pvar(P)$.
Decomposition: (DC1) $\exists\overline{U}.\ S \square P \square h\overline{a}_m \bowtie h\overline{b}_m, E \Vdash$
$\exists\overline{U}.\ S \square P \square \ldots, a_i \bowtie b_i, \ldots, E$ if $h\overline{a}_m,\ h\overline{b}_m$ are rigid and passive expressions.
Binding & Decomposition: (BD) $(k \geq 0)$ $\exists\overline{U}.\ S \square P \square X\overline{a}_k \asymp s\overline{b}_k, E \Vdash$
$\exists\overline{U}.\ X = s, (S \square P \square \ldots, a_i \asymp b_i, \ldots, E)\sigma$
if ($s\overline{b}_k$ is a rigid and passive expression or ($k = 0$, $s \in \mathcal{V}$)), $X \notin pvar(P)$, $X \notin var(s)$, $s \in Pat$, $var(s) \cap pvar(P) = \emptyset$. Where $\sigma = \{X/s\}$.
Imitation & Decomposition: (IM) $(k \geq 0)\exists\overline{U}.\ S \square P \square X\overline{a}_k \asymp h\overline{e}_m\overline{b}_k, E \Vdash$
$\exists\overline{V}_m, \overline{U}.\ X = h\overline{V}_m, (S \square P \square \ldots, V_i \asymp e_i, \ldots, a_j \asymp b_j, \ldots, E)\sigma$
if $h\overline{e}_m\overline{b}_k$ is a rigid and passive expression, $X \notin pvar(P)$, $X \notin svar(h\overline{e}_m)$, $h\overline{e}_m \notin Pat$ or $var(h\overline{e}_m) \cap pvar(P) \neq \emptyset$. Where $\sigma = \{X/h\overline{V}_m\}$, and $\overline{V}_m$ are fresh variables.
Outer Narrowing: (NR1) $(k \geq 0)$ $\exists\overline{U}.\ S \square P \square f\overline{e}_n\overline{a}_k \asymp b, E \Vdash$
$\exists\overline{V}, \overline{U}.\ S \square \ldots, e_i \to t_i, \ldots, P \square C, r\overline{a}_k \asymp b, E$
if $f\overline{e}_n\overline{a}_k$ is a rigid and active expression. Where $R : f\overline{t}_n \to r \Leftarrow C$ is a variant of a rule in $\mathcal{R}$, with $\overline{V} = var(R)$ fresh variables.
Guess & Outer Narrowing: (GN) $(k \geq 0)$ $\exists\overline{U}.\ S \square P \square X\overline{e}_q\overline{a}_k \asymp b, E \Vdash$
$\exists\overline{V}, \overline{U}.X = f\overline{t}_p, (S \square \ldots, e_j \to s_j, \ldots, P \square C, r\overline{a}_k \asymp b, E)\sigma$
if $q > 0$, $X \notin pvar(P)$. Where $R : f\overline{t}_p\overline{s}_q \to r \Leftarrow C$ is a variant of a rule in $\mathcal{R}$, with $\overline{V} = var(R)$ fresh variables, and $\sigma = \{X/f\overline{t}_p\}$.
Guess & Decomposition: (GD) $\exists\overline{U}.\ S \square P \square X a_{p+1} \ldots a_m \bowtie Y b_{q+1} \ldots b_m, E \Vdash$
$\exists\overline{V}_p, \overline{W}_q, \overline{U}.\ X = h\overline{V}_p,\ Y = h\overline{W}_q, (S \square P \square (h\overline{V}_p a_{p+1}..a_m \downarrow h\overline{W}_q b_{q+1}..b_m), E)\sigma$
if $(m \perp p) + (m \perp q) > 0$, $X, Y \notin pvar(P)$, $h\overline{V}_p a_{p+1} \ldots a_m$, $h\overline{W}_q b_{q+1} \ldots b_m$ are rigid and passive expressions. Where $\sigma = \{X/h\overline{V}_p, Y/h\overline{W}_q\}$, $\overline{V}_p, \overline{W}_q$ fresh variables.

Rules for $\to$

Decomposition: (DC2) $\exists\overline{U}.\, S\Box h\overline{e}_m \to h\overline{t}_m, P\Box E \Vdash$
$\exists\overline{U}.\, S\Box \ldots, e_i \to t_i, \ldots, P\Box E.$ if $h\overline{e}_m, h\overline{t}_m$ are rigid and passive expressions.
Output Binding & Decomposition: (OB) ($k \geq 0$)
(OB1) $\exists\overline{U}.\, S\Box X\overline{e}_k \to h\overline{t}_m\overline{s}_k, P\Box E \Vdash$
$\exists\overline{U}.\, X = h\overline{t}_m, (S\Box \ldots e_i \to s_i \ldots, P\Box E)\sigma$ if $X\notin pvar(P)$. Where $\sigma = \{X/h\overline{t}_m\}$.
(OB2) $\exists X, \overline{U}.\, S\Box X\overline{e}_k \to h\overline{t}_m\overline{s}_k, P\Box E \Vdash$
$\exists\overline{U}.\, (S\Box \ldots e_i \to s_i \ldots, P\Box E)\sigma$ if $X \in pvar(P)$, Where $\sigma = \{X/h\overline{t}_m\}$.
Input Binding: (IB) $\exists X, \overline{U}.\, S\Box t \to X, P\Box E \Vdash \exists\overline{U}.\, S\Box(P\Box E)\sigma$
if $t \in Pat$, where $\sigma = \{X/t\}$.
Input Imitation: (IIM) $\exists Y, \overline{U}.\, S\Box h\overline{e}_m \to Y, P\Box E \Vdash$
$\exists\overline{V}_m, \overline{U}.\, S\Box(\ldots, e_i \to V_i, \ldots, P\Box E)\sigma$
if $h\overline{e}_m\notin Pat$, $h\overline{e}_m$ is a rigid and passive expression, $Y \in dvar(E)$. Where $\sigma = \{Y/h\overline{V}_m\}$, $\overline{V}_m$ are fresh variables.
Elimination: (EL) $\exists X, \overline{U}.\, S\Box e \to X, P\Box E \Vdash \exists\overline{U}.\, S\Box P\Box E$ if $X\notin var(P\Box E)$.
Outer Narrowing: (NR2) ($k \geq 0$)$\exists\overline{U}.\, S\Box f\overline{e}_n\overline{a}_k \to t, P\Box E \Vdash$
$\exists\overline{V}, \overline{U}.\, S\Box \ldots, e_i \to t_i, \ldots, r\overline{a}_k \to t, P\Box C, E$
if $t\notin Var$ or $t \in dvar(E)$, where $R : f\overline{t}_n \to r \Leftarrow C$ is a variant of a rule in $\mathcal{R}$, with $\overline{V} = var(R)$ fresh variables.
Output Guess & Outer Narrowing: (OGN) ($k \geq 0$)$\exists\overline{U}.\, S\Box X\overline{e}_q\overline{a}_k \to t, P\Box E \Vdash$
$\exists\overline{V}, \overline{U}.\, X = f\overline{t}_p, (S\Box \ldots, e_i \to s_i, \ldots, r\overline{a}_k \to t, P\Box C, E)\sigma$
if $q > 0$, $X\notin pvar(P)$, $t\notin Var$ or $t \in dvar(E)$. Where $R : f\overline{t}_p\overline{s}_q \to r \Leftarrow C$ is a variant of a rule in $\mathcal{R}$, with $\overline{V} = var(R)$ fresh variables, and $\sigma = \{X/f\overline{t}_p\}$
Output Guess & Decomposition: (OGD)
(OGD1) $\exists Y, \overline{U}. S\Box X\overline{e}_q \to Y, P\Box E \Vdash$
$\exists\overline{U}, \overline{V}_p, \overline{W}_q. X = h\overline{V}_p, (S\Box ..e_i \to W_i.., P\Box E)\sigma$
if $q > 0$, $Y \in dvar(E)$, $h\overline{V}_p\overline{W}_q$ is a rigid and passive expression, $X\notin pvar(P)$. Where $\sigma = \{X/h\overline{V}_p, Y/h\overline{V}_p\overline{W}_q\}$, with $\overline{V}_p, \overline{W}_q$ fresh variables.
(OGD2) $\exists X, Y, \overline{U}.\, S\Box X\overline{e}_q \to Y, P\Box E \Vdash$
$\exists\overline{U}, \overline{V}_p, \overline{W}_q.(S\Box \ldots e_i \to W_i \ldots, P\Box E)\sigma$
if $q > 0$, $Y \in dvar(E)$, $h\overline{V}_p\overline{W}_q$ is a rigid and passive expression, $X \in pvar(P)$. Where $\sigma = \{X/h\overline{V}_p, Y/h\overline{V}_p\overline{W}_q\}$, with $\overline{V}_p, \overline{W}_q$ fresh variables.

Failure rules

Conflict: (CF1) $\exists\overline{U}.\, S\Box P\Box h\overline{a}_p \bowtie h'\overline{b}_q, E \Vdash FAIL$
if $h \not\equiv h'$ or $p \neq q$, and $h\overline{a}_p$, $h'\overline{b}_q$ are rigid and passive expressions.
Cycle: (CY) $\exists\overline{U}.\, S\Box P\Box X\bowtie a, E \Vdash FAIL$ if $X \not\equiv a, X \in svar(a)$.
Junk: (JK1) $\exists\overline{U}.\, S\Box P\Box a\bowtie b, E \Vdash FAIL$ if a or b are junk expressions.
Conflict: (CF2) $\exists\overline{U}.\, S\Box h\overline{a}_p \to h'\overline{t}_q, P\Box E \Vdash FAIL$
if $h \not\equiv h'$ or $p \neq q$, and $h\overline{a}_p$, $h'\overline{t}_q$ are rigid and passive expressions.
Junk: (JK2) $\exists\overline{U}.\, S\Box a \to t, P\Box E \Vdash FAIL$
if a is a junk expression, and ($t\notin\mathcal{V}$, or $t \in dvar(E)$).

In spite of the complex appearence of this narrowing calculus, there is a quite natural way to discover its rules, taking as a guide the fact that any solution θ of a goal G must be witnessed by rewriting proofs, as required by Definition 4.2. The following remarks attempt to clarify some other relevant aspects of the $HOCLNC$ calculus:

- In relation with statements of the forms $f\overline{e}_m \rightarrow X$ and $F\overline{e}_m \rightarrow X$ where $f\overline{e}_m$ is active, the following possibilities are considered:
  - (i) The rule *(EL)* deletes an approximation stament whose produced variable X does not appear elsewhere, because in this case any value (even $\perp$) is valid for the (existential) variable X to satisfy the goal. As a consequence, the evaluation of $f\overline{e}_m$ or $F\overline{e}_m$ is not needed and is indeed not performed. Hence, the rule (EL) is crucial for respecting the non-strict semantics of functions.
  - (ii) Rules *(NR2)* and *(OGN)* use a rule of the program for reducing $f\overline{e}_m$ and $F\overline{e}_m$ respectively, but only if X is detected as a *demanded* variable, which implies that X's value in a solution cannot be $\perp$, and therefore requires the evaluation of $f\overline{e}_m$, or $F\overline{e}_m$. After one or more applications of *(NR2)* and/or *(OGN)* it will be the case (if the computation is going to succeed) that *(IB)* or *(IIM)* or *(OGD)* are applicable, thus propagating (partially, in the case of *(IIM)* and *(OGD)*) the obtained value to all the occurrences of X. As a result, *sharing* is achieved, and computations are lazy.
  - (iii) If none of the rules *(EL), (NR2), (OGN), (OGD)* is applicable, nothing can be done with $f\overline{e}_m \rightarrow X$ resp. $F\overline{e}_m \rightarrow X$, but waiting until some of them becomes applicable. This will eventually happen, as our completeness results show.
- The absence of cycles of produced variables implies that no occur check is needed in *(OB), (IB), (IIM)*.
- The rules *(BD), (OB), (IB)* correspond to safe cases for eager variable elimination (see [19]). Special care must be taken with produced variables. For instance, the goal $\exists N.\ \square X \rightarrow (s\ N) \square X \bowtie N$ is admissible, but if *(BD)* would be applied (which is not allowed in $HOCLNC$, since N is a produced variable) we would obtain $\exists N.\ X = N \square N \rightarrow (s\ N) \square$, which is not admissible due to the presence of the produced variable N in the solved part and, more remarkably, the creation of a cycle $N \gg_P N$, with the subsequent need of occur check to detect unsolvability of $N \rightarrow (s\ N)$.
- Narrowing rules *(NR1), (NR2), (GN), (OGN)* include a *don't know* choice of the rule R of the program $\mathcal{R}$ to be used. Moreover, don' t know choice between rules (GN) and (GD), as well as between rules (OGN) and (OGD), is also needed for completeness.

Successful $HOCLNC$-derivations must end up with a goal of the form $\exists\overline{U}.\ S \square\ \square$. Such *solved forms* are trivially satisfiable; assumming $S \equiv X_1 = t_1, .., X_n = t_n$, we can define the *answer substitution* $\sigma_S = \{X_1/t_1, \ldots, X_n/t_n\}$, which is idempotent. Notice that $\sigma_S \in Sol(\exists\overline{U}.\ S \square\ \square)$.

We can prove soundness and completeness of $HOCLNC$ w.r.t. $HOCRWL$ semantics. Soundness relies on a lemma that ensures the correctness of one-step goal transformations.

Lemma 4.1 *(Correctness lemma).- Goal transformations preserve admissible goals. If* $G \Vdash FAIL$ *then* $Sol(G) = \emptyset$*. Moreover, If* $G \Vdash G'$ *and* $\theta' \in Sol(G')$ *then there exists* $\theta \in Sol(G)$ *with* $\theta = \theta'[\mathcal{V} \setminus evar(G)]$ ■

Theorem 4.1 *(Soundness of* $HOCLNC$*).- Assume* $G \Vdash^* \exists \overline{U}.\ S \square\ \square$*, where* G *is an initial goal. Then* $\sigma_S \in Sol(G)$*.*
Proof.- Repeatedly backwards apply Lemma 4.1, taking into account that $\sigma_S \in Sol(\exists \overline{U}.\ S)$ *and that none of the possible existential variables intervening in the computation belongs to* $var(G)$*.* ■

For the completeness proof, we use a well-founded multiset ordering to compare witnesses of solutions. Given a program $\mathcal{R}$ and two multisets $\mathcal{M} \equiv \{\{\Pi_1, \ldots, \Pi_n\}\}$ and $\mathcal{M}' \equiv \{\{\Pi'_1, \ldots, \Pi'_m\}\}$, each containing $HOGRC$-proofs of approximation and joinability statements, we define

$$\mathcal{M} \lhd \mathcal{M}' \Leftrightarrow \{\{\mid \Pi_1 \mid, \ldots, \mid \Pi_n \mid\}\} \prec \{\{\mid \Pi'_1 \mid, \ldots, \mid \Pi'_m \mid\}\}$$

where $\mid \Pi \mid$ is the *size* (i.e., the number of inference steps) of Π, and $\prec$ is the *multiset extension* [3] of the usual ordering over $\mathbb{N}$. Now, completeness of $HOCLNC$ can be established with the help of a progress lemma. This auxiliary result ensures that as long a goal is not solved, it can be transformed in such a way that it comes closer to the sought solution.

Lemma 4.2 *(Progress Lemma).- Let* $G \not\equiv FAIL$ *be any goal for a given program* $\mathcal{R}$*, and assume* $\theta \in Sol(G)$ *with witness* $\mathcal{M}$*. Then, we can choose some* $HOCLNC$*-transformation rule* T *applicable to* G*, so that for some* G' *and* θ' *we have:* $G \Vdash G'$ *by means of* T*;* $\theta' \in Sol(G')$ *with witness* $\mathcal{M}' \lhd \mathcal{M}$*; and* $\theta = \theta'[\mathcal{V} \setminus (evar(G) \cup nvar(G'))]$*, where* $nvar(G') = var(G') \setminus var(G)$ $(\subseteq evar(G'))$ ■

Theorem 4.2 *(Completeness of* $HOCLNC$*) .- Let* $\mathcal{R}$ *be a program,* G *an initial goal, and* $\theta \in Sol(G)$*. Then there exists a solved form* $\exists \overline{U}.\ S \square\ \square$ *such that* $G \Vdash^* \exists \overline{U}.\ S \square\ \square$ *and* $\sigma_S \leq \theta[var(G)]$*.*
Proof.-Repeatedly apply the progress lemma 4.2 until a solved form is reached. This will eventually happen, since $\lhd$ *is a well-founded ordering.* ■

5 An Experiment: Circuit Synthesis

In this Section we describe an experiment that has been performed using TOY, a functional logic programming language based on the *Higher Order Rewriting Logic* presented in this paper. The TOY system uses so-called *definitional trees* and translation techniques borrowed from [12] to compile source programs into Sicstus Prolog code. TOY supports a polymorphic type system with product types, and Haskell-like syntax. These facilities will be used in what follows.

Circuit synthesis is the problem of finding a logical circuit that exhibits some desired behaviour. This problem is a combinatorially hard one that is probably best tackled using constraint solving techniques [23]. Not pretending to offer a more efficient solution to this problem, we think that it provides a very natural illustration on how higher-order functions, lazy evaluation and non-determism can be combined to obtain a declaratively clean program that improves the naïve generate-and-test approach.

We represent circuits as functions mapping input lists of boolean values into output lists of the same type. Also, we represent circuit behaviours as lists of input-output pairs. These ideas lead to the following function definition (TOY's concrete syntax for defining rules is `f t1..tn = r <== a1 == b1, ..  , ak == bk`):

```
satisfies []                C = true
satisfies [(In,Out) | Rest] C = satisfies Rest C <= C In == Out
```

Moreover, we have programmed several higher-order functions that can be used as *circuit combinators*; i.e, they can be combined in many ways to build patterns of type circuit that serve as natural representations of circuits. Our circuit combinators are:

```
input I Xs = [Xs !! I]
%  Xs !! I returns the I-th element of list Xs.
andGate I J Xs  = [(Xs !! I) /\ (Xs !! J)]
% analogously for other gates.
(C2 . C1) Xs = C2 (C1 Xs)
(C1 # C2) Xs = C1 Xs ++ C2 Xs
% Ys ++ Zs is the concatenation of lists Ys and Zs.
```

For instance, the pattern

```
andGate 1 2 . ((notGate 1 . andGate 1 2) # orGate 1 2)
```

represents a circuit that realizes the behaviour of an XOR-gate, using NOT, AND and OR gates.

We have completed a TOY program that solves circuit synthesis problems using an optimization of the generate-and-test scheme that is programmed using higher-order functions:

```
findSolution Generate Test X = check (Generate X) Test
check Y Test = Y <== Test Y
```

Typically, `Generate` will be a nondeterministic function and `(Generate X)` will return several candidate solutions, even if `X` is bound to a ground pattern and no narrowing is involved. Due to lazy evaluation, `check` can reject a partially evaluated candidate solution `Y`. Thus, the scheme achieves incremental generation interleaved with testing in a very concise declarative formulation. Experiments based on the permutation sort algorithm ([13]) have shown significant time speedups of `findSolution` w.r.t. the naïve "first generate, then test" approach. Note that the "list of successes" approach used for search problems in lazy functional languages ([24]), would compute the list consisting of all the candidate solutions (possibly using a list comprehension), and pass it as a parameter to another function which would filter the actual solutions. In contrast to this, our approach relies on backtracking to compute the candidate solutions one by one. In both cases, the computation of candidates is lazy, and we conjecture that the pruning effect is similar (though there might be some differences in efficiency).

Our program uses `findSolution` with a convenient instantiation of the parameters `Generate`, `Test` and `X` in order to find a circuit with a given number of inputs and outputs that satisfies a given behaviour. Our circuit generators perform a very naïve nondeterministic construction of patterns of increasing syntactic size, built from circuit combinators. Different circuit generators can be designed, according to the kind of logical gates one wants to consider as primitive building blocks. The complete TOY program for circuit synthesis (along with the TOY system and some documentation) is available from our server at: http://mozart.mat.ucm.es/incoming/comprimidos/toy.tar.gz

6 Conclusions

We have extended the first-order approach from [6], obtaining a higher-order rewriting logic $HOCRWL$ that is proposed as a logical foundation for functional logic programming. We have shown that $HOCRWL$ enjoys nice proof-theoretical and model-theoretical properties, and we have presented a sound and complete lazy narrowing calculus $HOCLNC$ for goal solving. Completeness of $HOCLNC$ requires neither higher-order unification nor termination/confluence hypothesis; therefore, we allow non-deterministic lazy functions, and we are able to use first-order unification to work with higher-order patterns intensionally. Programming in $HOCRWL$ has been realized in the TOY language; we have discussed a circuit synthesis example that illustrates some of the adventages of higher-order programmming and non-deterministic functions in our system.

Future work will deal with the extension of our system to incorporate constraint solving capabilities.

Acknowledgements

We are grateful to the implementors of the TOY system: Rafa Caballero-Roldán, Paco López-Fraguas, and Jaime Sánchez-Hernández. Paco merits special thanks for his help in improving and testing the TOY program for circuit synthesis. We are also indebted to the anonymous referees for their constructive comments.

The authors have been partially supported by the spanish CICYT(project TIC 95-0433-C03-01 "CPD"), and the ESPRIT Working Group 22457 (CCL-II).

References

[1] S.Antoy. *Non-Determinism and Lazy Evaluation in Logic Programming.* Proc. LOPSTR' 91, pp. 318-331, 1991.

[2] G.Boudol. *Computational semantics of term rewriting systems.* In M. Nivat and J.C. Reynolds (Eds.), *Algebraic methods in semantics*, Cambridge University Press, Chapter 5, pp. 169-236, 1985.

[3] N.Dershowitz and Z.Manna. *Proving Termination with Multiset Orderings.* Comm. of the ACM 22(8), 1979, 465-476.

[4] M.Falaschi, G.Levi, M.Martelli and C.Palamidessi. *A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs.* Information and Computation, 102(1), pp. 86-113, 1993.

[5] E.Giovannetti, G.Levi, C.Moiso and C.Palamidessi. *Kernel-LEAF: A Logic plus Functional Language*. JCSS 42 (2), pp. 139-185, 1991.

[6] J.C.González-Moreno, M.T.Hortalá-González, F.J.López-Fraguas and M.Rodríguez-Artalejo. *A Rewriting Logic for Declarative Programming. Proc. ESOP'96*, Springer LNCS, Vol. 1058, 1996, pp. 156-172.

[7] J.C.González-Moreno, M.T.Hortalá-González and M.Rodríguez-Artalejo. *On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming*. Procs. CSL'92, Springer LNCS 702, pp. 216-230, 1993.

[8] J.C.González-Moreno, M.T.Hortalá-González and M.Rodríguez-Artalejo. *Declarative Programming in a HO-Rewriting Logic* . Tech. Report, Comput. Sci. Dpt. UCM, 1997 (Forthcoming).

[9] M.Hanus. *The Integration of Functions into Logic Programming: A Survey*. JLP (19&20). Special issue "*Ten Years of Logic Programming*", pp. 583-628, 1994.

[10] M.Hanus, C.Prehofer. *Higher-Order Narrowing with Definitional Trees*. Proc. RTA' 96, Springer LNCS 1103, pp. 138-152, 1996.

[11] H.Hussmann. *Non-determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[12] R.Loogen, F.J.López-Fraguas and M.Rodríguez-Artalejo. *A Demand Driven Computation Strategy for Lazy Narrowing*. Proc. PLILP' 93, Springer LNCS 714, pp.184-200, 1993.

[13] F. J. López-Fraguas. *Personal Communication*. 1996. (In Spanish)

[14] J.Meseguer. *Conditional rewriting logic as a unified model of concurrency*. TCS 96, pp. 73-155, 1992.

[15] D.Miller, G.Nadathur, F.Pfenning, and A.Scedrov. *Uniform Proofs as a foundation for logic programming*. Annals of Pure and Applied Logic 51, pp. 125-157, 1991.

[16] B.Möller. *On the Algebraic Specification of Infinite Objects - Ordered and Continuous Models of Algebraic Types*. Acta Informatica 22, pp. 537-578, 1985.

[17] J.J.Moreno-Navarro and M.Rodríguez-Artalejo. *Logic Programming with Functions and Predicates: The Language BABEL*. JLP 12, pp. 191-223, 1992.

[18] K.Nakahara, A.Middeldorp and T.Ida. *A Complete Narrowing Calculus for Higher Order Functional Logic Programming*. Proc. PLILP'95, Springer LNCS 982, pp. 97-114, 1995.

[19] S.Okui, A.Middeldorp and T.Ida. *Lazy Narrowing: Strong Completeness and Eager Variable Elimination*. Proc. CAAP'95, Springer LNCS 915, pp. 394-408, 1995.

[20] C.Prehofer. *A Call-by-Need Strategy for Higher-Order Functional Logic Programming.* Proc. ILPS'95, MIT Press, pp. 147-161, 1995.

[21] U.Reddy. *Narrowing as Operational Semantics of Functional Languages*. Proc. IEEE Symposium on Logic Programming' 85, pp. 138-151, 1985.

[22] D.S.Scott. *Domains for Denotational Semantics*. Proc. ICALP' 82, Springer LNCS 140, pp. 577-613, 1982.

[23] H.Simonis, and T.Graf. *Technology mapping in CHIP*. Tech. Report TR-LP-44, ECRC, Munich, Germany, 1990.

[24] P.Wadler. *How to Replace Failure by a List of Successes.* Proc. IFIP Int. Conf. on Funct. Progr. Lang. and Computer Architectures, Springer LNCS 201, pp.113-128, 1985.