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].
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 \(\lambda\)-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 \(a \Join b\)) 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: \(\text{while } P \ (\text{iterate } s \ \text{zero}) \ \Join [\text{zero}, \ s \ \text{zero}]\) for which higher-order narrowing is expected to get \(P = \text{le} \ (s \ \text{zero})\), as a computed answer. Note the partial application of the defined function symbol \(\text{le}\) 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 \(\lambda\)-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].

<table>
<thead>
<tr>
<th>le X zero</th>
<th>(\rightarrow) true</th>
</tr>
</thead>
<tbody>
<tr>
<td>le (s X)</td>
<td>(\rightarrow) false</td>
</tr>
<tr>
<td>le (s X) (s Y)</td>
<td>(\rightarrow) le X Y</td>
</tr>
<tr>
<td>iterate F X</td>
<td>(\rightarrow) [X</td>
</tr>
</tbody>
</table>
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 $\bot$ (w.r.t. $\sqsubseteq$). We say that an element $x \in S$ is totally defined (written $\text{Def}(x)$) if $x$ is maximal w.r.t. $\sqsubseteq$. The set of all totally defined elements of $S$ will be noted $\text{Def}(S)$. $D \subseteq S$ is a directed set iff for all $x, y \in D$ there exists $z \in D$ with $x \sqsubseteq z, y \sqsubseteq z$. A subset $A \subseteq S$ is a cone iff $\bot \in A$, and $A$ is downwards closed, i.e., $y \sqsubseteq x \Rightarrow y \in A$, for all $x \in A, y \in S$. An ideal $I \subseteq S$ is a directed cone. We write $C(S), I(S)$ for the sets of cones and ideals of $S$ respectively. The set $\bar{S} = I(S)$, 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 $\bar{S}$, which maps each $x \in S$ into the principal ideal generated by $x$, $< x > = \{ y \in S : y \subseteq x \} \in \bar{S}$. A poset with bottom $\mathbb{C}$ is a cpo iff $\mathbb{D}$ has a least upper bound $\sqcup \mathbb{D}$ (also called limit) for every directed set $\mathbb{D} \subseteq \mathbb{C}$. $u \in \mathbb{C}$ is a finite element (written $\text{fin}(u)$) if whenever $u \subseteq \sqcup \mathbb{D}$ for a directed $\mathbb{D} \subseteq \mathbb{C}$, there exists $x \in \mathbb{D}$ with $u \subseteq x$. A cpo $\mathbb{C}$ is called algebraic if any element of $\mathbb{C}$ is the limit of a directed set of finite elements. For any poset with bottom $\mathbb{S}$, its ideal completion $\bar{S}$ turns to be the least cpo including $\mathbb{S}$. Furthermore, $\bar{S}$ is an algebraic cpo whose finite elements are precisely the principal ideals $< x >, x \in \mathbb{S}$; see e.g., [16]. Note that elements $x \in \text{Def}(S)$ correspond to finite and maximal elements $< x >$ in the ideal completion.

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

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

$$\text{Exp} ::= X \% X \in V \mid h \% h \in DC \cup FS \mid (e \ e') \% e, e' \in \text{Exp}$$

Expression $(e \ e')$ 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, $(e_0 \ e_1 \ldots \ e_n)$ abbreviates $((\ldots (e_0 \ e_1) \ldots) \ e_n)$. We distinguish an important kind of expressions called patterns (Pat $\subseteq$ Exp) defined as:

$$\text{Pat} ::= X \% X \in V \mid c \ t_1 \ldots \ t_m \% t, \in \text{Pat} \text{ and } c \in DC^n, 0 < m < n \mid f \ t_1 \ldots \ t_m \% t, \in \text{Pat} \text{ and } f \in FS^n, 0 \leq m < n$$

For some purposes we need to enhance $\Sigma$ with a new constant (0-arity constructor) $\bot$, obtaining a new signature $\Sigma_\bot$. We will write $\text{Exp}_\bot$ and $\text{Pat}_\bot$ 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 $\bot$ are called total. As frequent notational conventions we will also use $c, d \in DC; f, g \in FS; s, t \in \text{Pat}_\bot; a, b, e \in \text{Exp}_\bot$. 

In the rest of the paper we assume the following classification of expressions: \(X \cdot e_1, \ldots, e_m\), with \(X \in \mathcal{V}\) and \(m \geq 0\), is called a flexible expression, while \(h \cdot e_1, \ldots, e_m\) with \(h \in \text{DC}^n \cup \text{FS}^n\) is called a rigid expression. Moreover, a rigid expression is called junk iff \(h \in \text{DC}^n\) and \(m > n\), active iff \(h \in \text{FS}^n\) 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 \(\leq\) for partial patterns can be defined as follows: \(\subseteq\) is the least partial ordering over \(\text{Pat}\) satisfying the following properties: \(\bot \subseteq \top\), for all \(t \in \text{Pat}\); \(X \subseteq X\), for all \(X \in \mathcal{V}\); and \(h \cdot t_1 \ldots t_m \subseteq h \cdot s_1 \ldots s_m\) whenever these two expressions are rigid patterns and \(t_i \subseteq s_i\) for all \(1 \leq i \leq m\).

Substitutions are mappings \(\theta : \mathcal{V} \rightarrow \text{Pat}\) which have a unique natural extension \(\hat{\theta} : \mathcal{E} \rightarrow \mathcal{E}\), also noted as \(\theta\). The set of all substitutions is noted as \(\text{Subst}\).

The bigger set \(\text{Subst}\) of all partial substitutions \(\theta : \mathcal{V} \rightarrow \text{Pat}\) is defined analogously. We note as \(e\theta\) the result of applying the substitution \(\theta\) to the expression \(e\), and we define the composition \(\sigma\theta\) such that \(e(\sigma\theta) \equiv (e\sigma)\theta\). As usual, \(\theta = \{X_1 / t_1, \ldots, X_n / t_n\}\) stands for the substitution that satisfies \(X_i\theta \equiv t_i\) (\(1 \leq i \leq n\)) and \(Y\theta \equiv Y\) for all \(Y \in \mathcal{V} \setminus \{X_1, \ldots, X_n\}\). A substitution \(\theta\) such that \(\theta \theta = \theta\) is called idempotent. The approximation ordering over \(\text{Pat}\) induces a natural approximation ordering over \(\text{Subst}\), defined by: \(\theta \subseteq \theta'\) iff \(X\theta \subseteq X\theta'\), for all \(X \in \mathcal{V}\). We will use also the subsumption ordering over \(\text{Subst}\), defined by: \(\theta \leq \theta'\) if \(\theta' = \theta \sigma\) for some \(\sigma\). Finally, the notation \(\theta \leq \theta' [\mathcal{U}\)\], where \(\mathcal{U} \subseteq \mathcal{V}\), means that \(X\theta \equiv X(\theta\sigma)\) for some \(\sigma\) and for all \(X \in \mathcal{U}\) (i.e., \(\theta\) is more general than \(\theta'\) 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 \(\text{CRWL}\)) 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 \(\text{HOCRWL}\) of \(\text{CRWL}\) is designed to derive the following kinds of statements: Reduction statements \(e \rightarrow e'\), whose intended meaning is that \(e\) can be reduced to \(e'\). Approximation statements \(e \rightarrow t\), whose intended meaning is that the possibly partial pattern \(t\) approximates the denotation of the expression \(e\). Joinability statements \(a \bowtie h\), which hold iff \(a \rightarrow t\) and \(b \rightarrow t\) can be derived for some total pattern \(t\) (with no occurrences of \(\bot\)).

We allow for partial patterns in \(\text{HOCRWL}\)-statements because we want to model the behaviour of non-strict functions which may work with potentially infinite data structures. \(\text{HOCRWL}\)-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:
where \( f \in \mathsf{FS}^n \), \( l \) 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 \( a \bowtie b \) where \( r, a, b \) are expressions. In the sequel we use the following notation for possibly partial instances of rewrite rules:

\[
[R]_\bot = \{ (l \to r \leftarrow C) \theta \mid (l \to r \leftarrow C) \in R, \theta \in \mathsf{Subst}_\bot \}
\]

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

\[
\begin{align*}
\text{BC:} & \quad e \to \bot \\
\text{RF:} & \quad e \to e \\
\text{TR:} & \quad e \to e', e' \to e'' \\
\text{MN:} & \quad \frac{e_0 \to e_1}{(e_0 \to e_1)} \\
& \quad \frac{e_0 \to e_1}{(e_0 \to e_1)} \\
\text{R:} & \quad \frac{C}{\Gamma \to r} \\
& \quad \text{for } (l \to r \leftarrow C) \in [R]_\bot \\
\text{J:} & \quad \frac{a \bowtie b}{\to t} \\
& \quad \text{if } t \text{ is a total pattern.}
\end{align*}
\]

Note that \( \mathcal{HOCRWL} \)-reduction is related to the idea of approximation, as shown by rule \( \text{BC} \). In rule \( \text{J} \), 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 \( \text{R} \) we use rule instances from \( [R]_\bot \) 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 (\( \mathcal{HGRC} \)) whose proofs are closer to the goal-solving calculus which will be presented in Section 4.

\[
\begin{align*}
\text{B:} & \quad e \to \bot \\
\text{RR:} & \quad X \to X, \text{if } X \in \mathcal{V} \\
\text{OR:} & \quad \frac{e_1 \to t_1 \ldots e_n \to t_n}{C \to t} \quad \text{if } t \neq \bot \text{ is a pattern, } m \geq 0, \text{ and } f t_1 \ldots t_n \to r \leftarrow C \in [R]_\bot \\
& \quad \frac{e_1 \to t_1 \ldots e_n \to t_n}{C \to t} \\
& \quad \text{if } t \neq \bot \text{ is a pattern, } m \geq 0, \text{ and } f t_1 \ldots t_n \to r \leftarrow C \in [R]_\bot \\
\text{DC:} & \quad \frac{e_1 \to t_1 \ldots e_n \to t_n}{h \to \frac{t_1 \ldots t_n}{h}} \\
& \quad \text{if } h \text{ is a rigid pattern} \\
& \quad \frac{a \to t}{b \to \frac{t}{a \bowtie b}} \\
& \quad \text{if } t \text{ is a total pattern}
\end{align*}
\]

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

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

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

\[
\text{iterate } s \text{ zero } \rightarrow [\text{zero}, \text{zero} | \perp]
\]

\[
\text{while } (\text{le } (s \text{ zero})) \text{ (iterate } s \text{ zero) } \preceq [\text{zero}, \text{zero}]
\]

In addition to the proof calculi presented above, we have developed a model theory for \( \text{HOCHR} \). 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 \( \sqcap \), \( \sqcup \), we define:

- The set of all non-deterministic functions from \( D \) to \( E \) as:
  \[
  [D \to \sqcup E] =_{\text{def}} \{ f : D \to C(E) \mid \forall u, u' \in D : (u \leq u' \Rightarrow f(u) \subseteq f(u')) \}.
  \]

- The set of all deterministic functions from \( D \) to \( E \) as:
  \[
  [D \to \sqcap E] =_{\text{def}} \{ f \in [D \to \sqcap E] \mid \forall u \in D : f(u) \in \mathcal{F}(E) \}.
  \]

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

In [6], CRWL-algebras \( A \) interpret functions symbols \( f \in \mathcal{F}^n \) as non-deterministic functions \( f^A \in [D^A \to^n D^A] \). 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** (HOCHRWL-Algebras).- Let \( \Sigma \) be any given signature. HOCHRWL-algebras of signature \( \Sigma \) are algebraic structures of the form:

\[
A = (D^A, @^A, \{ c^A \}_{c \in DC_\Sigma}, \{ f^A \}_{f \in \mathcal{F}^n \Sigma}) \quad \text{where:}
\]

- \( D^A \) is a poset with bottom element \( \bot^A \).
- \( @^A \in [D^A \times D^A \to \sqcup D^A] \) is a non-deterministic binary function (apply operation), written in infix notation such that \( \bot^A @^A \nu = \bot^A \) for all \( \nu \in D^A \).
- For given \( u_0, u_1, \ldots, u_k \), we will use the notation \( \langle u_0 @^A u_1 @^A \ldots @^A u_k \rangle \) to denote a cone that is defined as:
  \[
  < u_0 >, \quad \text{if } k = 0, \text{and } \bigcup \{ u \in D^A @^A u_k \mid u \in u_0 @^A \ldots @^A u_k \} \text{, if } k > 0.
  \]
- \( c^A \in D^A, @^A \in D^A, \text{if } 0 < \text{ar}(c), \text{and } f^A \in C(D^A), \text{if } 0 = \text{ar}(f). \)
- For all \( u_1, \ldots, u_k \in D^A \), \( c^A @^A u_1 @^A \ldots @^A u_k = < \bot^A >, \text{if } k > \text{ar}(c). \)
- For all \( u_1, \ldots, u_k \in D^A \) there is \( \nu \in D^A \) such that \( h @^A u_1 @^A \ldots @^A u_k = < \nu >, \text{where } h \in D^A @^A u_0 \bigcup D^A, \text{and } 0 < k < n, 0 < k < m. \text{ Moreover, } \nu \text{ is maximal in case that all } u_i \text{ are also maximal}. \)
It is convenient to simplify the notation by writing $h^A u_1 \ldots u_k$ in place of $h^A @^A u_1 @^A \ldots @^A u_k$. For $f \in FS^n$, we will say that $f^A$ is deterministic iff for all $u_1 \ldots u_n \in D^A$ the cone $f^A u_1 \ldots u_n$ is an ideal. Note that $c^A$ is always deterministic for all $c \in DC^n$.

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

- $[ \bot ]^A \eta = \bot < \bot^A$;
- $[ X ]^A \eta = \Delta_f < \eta(X)$, for $X \in \mathcal{V}$;
- $[ h ]^A \eta = \Delta_f < h^A$, for $h \in DC \cup FS^n$, with $n > 0$; $[ f ]^A \eta = \Delta_f f^A$, for $f \in FS^c$.
- $[ e e']^A \eta = \Delta_f [ e ]^A \eta [ e' ]^A \eta$.

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 $A$. We define:

- $A$ satisfies a reduction statement $e \rightarrow e'$ under a valuation $\eta$ (in symbols, $(A, \eta) \models e \rightarrow e'$) iff $[ e ]^A \eta \supseteq [ e' ]^A \eta$.
- $A$ satisfies a joinability statement $a \lor b$ under a valuation $\eta$ (in symbols, $(A, \eta) \models a \lor b$) iff $[ a ]^A \eta \cap [ b ]^A \eta \subseteq \text{Def}(D^A) = \not= \emptyset$.
- $A$ satisfies a rule $l \rightarrow \tau \in \mathcal{C}$ iff every valuation $\eta$ such that $A \models C \eta$ verifies: $(A, \eta) \models l \rightarrow \tau$.
- $A$ is a model of $\mathcal{R}$ (in symbols $A \models \mathcal{R}$) iff $A$ satisfies all the rules in $\mathcal{R}$. \hfill \Box$

For each program $\mathcal{R}$, we can build a pattern model $\mathcal{M}_\mathcal{R}$ whose carrier $D^{\mathcal{M}_\mathcal{R}}$ is the set $\text{Pat}_-$ 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 $\varphi$, the following conditions are equivalent:

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

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

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

In particular, for all $f \in FS^n$ and $t_1 \ldots t_n$, $t \in \text{Pat}_-$, $f^{\mathcal{M}_\mathcal{R}} t_1 \ldots t_n \rightarrow t$ holds iff $\mathcal{R} \vdash_{HOCRWL} f t_1 \ldots t_n \rightarrow t$. This shows that $\mathcal{M}_\mathcal{R}$ can be seen as a canoncanic 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 multiset 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 \( G \equiv \exists \overline{U}, S \Box F \Box E \), where:

- \( \text{evar}(G) \equiv \overline{U} \) is the set of so-called existential variables of the goal \( G \).
- \( S \equiv X_1 = s_1, \ldots, X_n = s_n \) 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 \( X_i \).)
- \( P \equiv e_1 \Rightarrow t_1, \ldots, e_k \Rightarrow t_k \) is a multiset of approximation conditions, with \( t_i \in \text{Pat} \). \( \text{pvar}(P) \equiv \text{def} \var{t_1} \cup \ldots \cup \var{t_k} \) is called the set of produced variables of the goal \( G \). The production relation between \( G \)-variables is defined by \( X \gg_P Y \) iff there is some \( 1 \leq i \leq k \) such that \( X \in \var{e_i} \) and \( Y \in \var{t_i} \).
  (Intuition: \( e_i \Rightarrow t_i \) demands narrowing \( e_i \) to match \( t_i \). This may produce bindings for variables in \( t_i \).
- \( E \equiv a_1 \bowtie b_1, \ldots, a_r \bowtie b_r \) is a multiset of joinability conditions. \( \text{dvar}(E) \equiv \text{def} \{X \in V / (X \in e_1 \ldots e_m) \equiv a, \text{or} (X \in e_1 \ldots e_m) \equiv b, \text{for some} 1 \leq i \leq l, \text{and} m \geq 0\} \) 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 \( \perp \).)

Additionally, any admissible goal must fulfill the following conditions:

- The tuple \( (t_1, \ldots, t_k) \) must be linear.
  (Intuition: Each produced variable is produced only once.)
- All the produced variables must be existential, i.e. \( \text{pvar}(P) \subseteq \text{evar}(G) \).
  (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 \( \theta \) that renders \( G\theta \) provable in the rewriting logic \( HOCRL \). The next definition allows \( \theta \) 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 \( \theta \) a partial substitution.

- \( \theta \) is allowable for \( G \) iff \( X \theta \) is a total pattern for every \( X \not\in \text{pvar}(P) \).
- \( \theta \) is a solution for \( G \) iff \( \theta \) is allowable for \( G \), and for all \( (X_i = s_i) \in S, \) and \( (P \Box E) \theta \) has a “witness” \( M \).
- A witness is defined as a multiset containing a HOGFIC-proof (see Section 3) for each condition \( e_i \Rightarrow t_i \in P \) and \( a \bowtie b \theta \in E \).
- We write \( \text{Sd}(G) \) 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 \( (HOCI\text{NC}) \) for solving initial goals, obtaining solutions in the sense of Definition
4.2. The calculus \textit{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 \textit{FAIL}, representing an irreducible inconsistent goal. We recall that in a goal \( \exists \vec{U}, S \ \square P \ \square E \), \( S \) 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: \( a \Rightarrow b \) stands for \( a \Rightarrow b \) or \( b \Rightarrow a \); \( \overline{ar} \) stands for \( a_1 \ldots a_m ; h\overline{ar} \downarrow h\overline{ar} \) stands for \( a_1 \cdots a_m \Rightarrow b_1 \cdots b_m \) if \( h\overline{ar} \). \( h\overline{ar} \) are rigid and passive expressions; and \textit{sva}(\( e \)) 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 \( e \in \text{Exp} \) we define its set of safe variables \textit{sva}(\( e \)) as follows: for a flexible expression \( e \equiv Xe_1 \ldots e_m \), \textit{sva}(\( e \)) = \{ \( X \) \} if \( m = 0 \); for a rigid and passive expression \( e \equiv he_1 \ldots e_m \), \textit{sva}(\( e \)) = \( \bigcup_{i=1}^{m} \textit{sva}(\( e_i \)) \); in any other case, \textit{sva}(\( e \)) = \{ \}. 

The \textit{HOCLNC} calculus

Rules for \( \infty \)

- **Identity:** (ID) \( \exists \vec{U}, S \ \square P \ \square X \Rightarrow X, E + \exists \vec{U}, S \ \square P \ \square E \) if \( X \notin \textit{sva}(\( P \)) \).

- **Decomposition:** (DC1) \( \exists \vec{U}, S \ \square P \ \square \ h\overline{ar}_m \Rightarrow h\overline{ar}_m \), \( E \vdash \) if \( h\overline{ar}_m \) are rigid and passive expressions.

- **Bining & Decomposition:** (BD) \( \{ k \geq 0 \} \exists \vec{U}, S \ \square P \ \square X \overline{at}_k \Rightarrow \overline{at}_k, E \vdash \exists \vec{U}, X = s, (S \ \square P \ \square \ldots, a_i \Rightarrow b_i, \ldots, E)\sigma \)

if \( (\overline{at}_k \) is a rigid and passive expression or \( (k = 0, s \in V) \), \( X \notin \textit{sva}(\( P \)) \), \( X \notin \textit{sva}(\( s \)) \), \( s \in \text{Pat} \), \( \textit{var}(s) \cap \textit{sva}(\( P \)) \neq \emptyset \). Where \( \sigma = \{ X/s \} \).

- **Imitation & Decomposition:** (IM) \( \{ k \geq 0 \} \exists \vec{U}, S \ \square P \ \square X \overline{at}_m \Rightarrow h\overline{at}_m \overline{at}_k, E + \exists \vec{V}_m, \vec{U}, X = h\overline{V}_m, (S \ \square P \ \square \ldots, V_i \Rightarrow e_i, \ldots, a_j \Rightarrow b_j, \ldots, E)\sigma \)

if \( h\overline{at}_m \overline{at}_k \) is a rigid and passive expression, \( X \notin \textit{sva}(\( P \)) \), \( X \notin \textit{sva}(h\overline{at}_m \overline{at}_k) \), \( h\overline{at}_m \notin \text{Pat} \) or \( \textit{var}(h\overline{at}_m) \cap \textit{sva}(\( P \)) \neq \emptyset \). Where \( \sigma = \{ X/h\overline{V}_m \} \), and \( \overline{V}_m \) are fresh variables.

- **Outer Narrowing:** (ONR) \( \{ k \geq 0 \} \exists \vec{U}, S \ \square P \ \square \overline{at}_n \overline{at}_k \Rightarrow b, E + \exists \vec{V}, \vec{U}, S \ \square P \ \square \epsilon_i \Rightarrow t_i, \ldots, P \ \square C, r\overline{at}_k \Rightarrow b, E \)

if \( \overline{at}_n \overline{at}_k \) is a rigid and active expression. Where \( R : \overline{f}_n \rightarrow r \Leftarrow C \) is a variant of a rule in \( \mathcal{R} \), with \( \overline{V} = \textit{var}(\( R \)) \) fresh variables.

- **Guess & Outer Narrowing:** (GN) \( \{ k \geq 0 \} \exists \vec{U}, S \ \square P \ \square \overline{at}_n \overline{at}_k \Rightarrow b, E + \exists \vec{V}, \vec{U}, X = \overline{f}_n, (S \ \square \ldots, e_i \Rightarrow s_i, \ldots, P \ \square C, r\overline{at}_k \Rightarrow b, E)\sigma \)

if \( q > 0 \), \( X \notin \textit{sva}(\( P \)) \). Where \( R : \overline{f}_n \rightarrow r \Leftarrow C \) is a variant of a rule in \( \mathcal{R} \), with \( \overline{V} = \textit{var}(\( R \)) \) fresh variables, and \( \sigma = \{ X/\overline{f}_n \} \).

- **Guess & Decomposition:** (GD) \( \exists \vec{U}, S \ \square P \ \square \overline{at}_m \overline{at}_n \overline{at}_k \Rightarrow b, E \vdash \exists \vec{V}, \vec{U}, X = \overline{f}_n, Y = h\overline{V}_q, (S \ \square P \ \square h\overline{V}_q \ \square \overline{at}_m \overline{at}_n \downarrow h\overline{V}_q \overline{at}_k, b_m, E)\sigma \)

if \( (m \perp P) \), \( (m \perp q) \), \( (m \perp q) \), \( X \notin \textit{sva}(\( P \)) \), \( h\overline{V}_q \overline{at}_m \overline{at}_n \overline{at}_k \overline{at}_p \overline{at}_q \overline{at}_b \) are rigid and passive expressions. Where \( \sigma = \{ X/h\overline{V}_q \}, \{ Y/h\overline{V}_q \}, \overline{V}_q, \overline{W}_q \} \) fresh variables.
Rules for $\rightarrow$

Decomposition: (DC2) $\exists \overline{U}, S \otimes \overline{x}_m \rightarrow \overline{h}_m, P \otimes E \vdash \exists \overline{U}, S \otimes \ldots \otimes \overline{x}_i \rightarrow \overline{t}_i, \ldots, P \otimes E$. if $h_m, h_m$ are rigid and passive expressions.

Output Binding & Decomposition: (OB) ($k \geq 0$)

(1) $\exists \overline{U}, S \otimes \overline{x}_k \rightarrow \overline{h}_m \overline{x}_k, P \otimes E \vdash $

(2) $\exists \overline{U}, \overline{x}_k \rightarrow \overline{h}_m \overline{x}_k, P \otimes E \vdash$

Input Binding: (IB) $\exists X, \overline{U}, S \otimes \overline{t} \rightarrow X, P \otimes E \vdash \exists \overline{U}, S \otimes (P \otimes E)\sigma$

Input Iteration: (II) $\exists Y, \overline{U}, S \otimes h \overline{t}_m \rightarrow Y, P \otimes E \vdash$

Elimination: (EL) $\exists X, \overline{U}, S \otimes \overline{t} \rightarrow X, P \otimes E \vdash \exists \overline{U}, S \otimes P \otimes E$ if $X \notin \text{var}(P \otimes E)$.

Outer Narrowing: (NR2) ($k \geq 0$) $\exists \overline{U}, S \otimes \overline{a}_k \rightarrow \overline{t}, P \otimes E \vdash$

if $\overline{a}_k \notin \text{dvar}(E)$, where $R : \overline{t} \rightarrow r \rightarrow C$ is a variant of a rule in $R$, with $\overline{V} = \text{var}(R)$ fresh variables.

Output Guess & Outer Narrowing: (OGN) ($k \geq 0$) $\exists \overline{U}, S \otimes \overline{a}_k \rightarrow \overline{t}, P \otimes E \vdash$

if $\overline{a}_k \notin \text{dvar}(E)$, where $R : \overline{t} \rightarrow r \rightarrow C$ is a variant of a rule in $R$, with $\overline{V} = \text{var}(R)$ fresh variables, and $\overline{x} = \text{var}(R)$. fresh variables.

Output Guess & Decomposition: (OGD)

(1) $\exists \overline{U}, \overline{V}_p, S \otimes \overline{x}_q \rightarrow \overline{y}, P \otimes E \vdash$

if $\overline{x}_k \notin \text{dvar}(E)$, where $R : \overline{t} \rightarrow r \rightarrow C$ is a variant of a rule in $R$, with $\overline{V} = \text{var}(R)$ fresh variables, and $\overline{x} = \text{var}(R)$. fresh variables.

Conflict: (CF1) $\exists \overline{U}, S \otimes P \otimes h \overline{t}_a \rightarrow \overline{t}_d, E \vdash \text{FAIL}$ if $h \neq h'$ or $p \neq a$, and $\overline{t}_d, h \overline{t}_a$ are rigid and passive expressions.

Cycle: (CY) $\exists \overline{U}, S \otimes P \otimes X \rightarrow \alpha, E \vdash \text{FAIL}$ if $X \neq a$, $X \in \text{var}(a)$.

Junk: (JK1) $\exists \overline{U}, S \otimes P \otimes \overline{a} \rightarrow \overline{t}_d, E \vdash \text{FAIL}$ if $a$ or $t$ are junk expressions.

Conflict: (CF2) $\exists \overline{U}, S \otimes h \overline{t}_a \rightarrow \overline{t}_d, P \otimes E \vdash \text{FAIL}$ if $h \neq h'$ or $p \neq a$, and $h \overline{t}_a, h \overline{t}_d$ are rigid and passive expressions.

Junk: (JK2) $\exists \overline{U}, S \otimes a \rightarrow \overline{t}, P \otimes E \vdash \text{FAIL}$ if $a$ is a junk expression, and $(t \notin \overline{V}$ or $t \notin \text{dvar}(E)$).

In spite of the complex appearance of this narrowing calculus, there is a quite natural way to discover its rules, taking as a guide the fact that any solution $\theta$ of a
Successful goal transformations. Such goal narrowing rules are needed in the absence of cycles of produced variables, implying that no occurrence of the produced variable is active, the following possibilities are considered:

(i) The rule (EL) deletes an approximation statement whose produced variable $X$ does not appear elsewhere, because in this case any value (even $\bot$) is valid for the (existential) variable $X$ to satisfy the goal. As a consequence, the evaluation of $\bar{\tau}_m$ or $\bar{\varphi}_m$ 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 $\bar{\tau}_m$ and $\bar{\varphi}_m$ respectively, but only if $X$ is detected as a demanded variable, which implies that $X$’s value in a solution cannot be $\bot$, and therefore requires the evaluation of $\bar{\tau}_m$ or $\bar{\varphi}_m$. 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 (ODG) are applicable, thus propagating (partially, in the case of (IIM) and (ODG)) 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), (ODG) is applicable, nothing can be done with $\bar{\tau}_m \rightarrow X$ resp. $\bar{\varphi}_m \rightarrow X$, 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 $\exists N. \bigtriangleup X \rightarrow (s N) \bigtriangleup X \bowtie N$ is admissible, but if (BD) would be applied (which is not allowed in HOCLNC, since $N$ is a produced variable) we would obtain $\exists N. X = N \bigtriangleup N \rightarrow (s N) \bigtriangleup N$, 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 $N \bowtie F N$, 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 $\bar{\iota}$ 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 (ODG), is also needed for completeness.

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

We can prove soundness and completeness of $\text{HOCLNC}$ w.r.t. $\text{HOCRW}$ 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 $G \vdash F A I I$, then $Sol(G) = \emptyset$. Moreover, if $G \vdash G'$ and $\theta' \in Sol(G')$ then there exists $\theta \in Sol(G')$ with $\theta = \theta'[\text{ev}ar(G)]$.

Theorem 4.1 (Soundness of HOCLNC).- Assume $G \vdash^* \exists U, S \Box \Box$, where $G$ is an initial goal. Then $\sigma_S \in Sol(G)$.

Proof.- Repeatedly backwards apply Lemma 4.1, taking into account that $\sigma_S \in Sol(\exists U, S)$ and that none of the possible existential variables intervening in the computation belongs to $\text{var}(G)$.

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

$$M < M' \iff \{|\Pi_1|, \ldots, |\Pi_n|\} < \{|\Pi'_1|, \ldots, |\Pi'_m|\}$$

where $|\Pi|$ is the size (i.e., the number of inference steps) of $\Pi$, and $<$ 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 $G \not\vdash F A I I$ be any goal for a given program $R$, and assume $\theta \in Sol(G)$ with witness $M$. Then, we can choose some HOCLNC-transformation rule $T$ applicable to $G$, so that for some $G'$ and $\theta'$ we have: $G \vdash G'$ by means of $T$; $\theta' \in Sol(G')$ with witness $M' \triangleleft M$; and $\theta = \theta'[\text{ev}ar(G) \cup n\text{var}(G')]$, where $n\text{var}(G') = \text{var}(G') \setminus \text{var}(G)$ ($\subseteq \text{ev}ar(G)$).

Theorem 4.2 (Completeness of HOCLNC) .- Let $R$ be a program, $G$ an initial goal, and $\theta \in Sol(G)$. Then there exists a solved form $\exists U, S \Box \Box$ such that $G \vdash^* \exists U, S \Box \Box$ and $\sigma_S < \theta[\text{ev}ar(G)]$.

Proof.- Repeatedly apply the progress lemma 4.2 until a solved form is reached. This will eventually happen, since $<$ 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 illus-
tration 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 \( f \, t_1..t_n = r \, <= \, a_1 == b_1, \ldots, a_k == b_k \)):

\[
\begin{align*}
\text{satisfies \{\}} & \quad C = \text{true} \\
\text{satisfies \{(In,Out) | Rest\} C} & \quad = \text{satisfies Rest C} <= C \quad \text{In == Out}
\end{align*}
\]

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:

- \( \text{input I Xs = [Xs !! I]} \)
  
  \% \( Xs !! I \) returns the \( I \)-th element of list \( Xs \).
- \( \text{andGate I J Xs = [(Xs !! I) /\ (Xs !! J)]} \)
  
  \% analogously for other gates.
- \( (C_2 \cdot C_1) \, Xs = C_2 \, (C_1 \, Xs) \)
- \( (C_1 \# C_2) \, Xs = C_1 \, Xs \, ++ \, C_2 \, Xs \)
  
  \% \( Ys ++ Zs \) is the concatenation of lists \( Ys \) and \( Zs \).

For instance, the pattern

\[ \text{andGate 1 2 . } (\text{notGate 1 . andGate 1 2}) \# \text{orGate 1 2} \]

represents a circuit that realizes the behaviour of an \( \text{XOR} \)-gate, using \( \text{NAND} \), \( \text{AND} \),
and \( \text{OR} \) 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:

\[
\begin{align*}
\text{findSolution Generate Test X} & \quad = \text{check (Generate X) Test} \\
\text{check Y Test} & \quad = Y <= Test \, Y
\end{align*}
\]

Typically, \( \text{Generate} \) will be a nondeterministic function and \( (\text{Generate X}) \)
will return several candidate solutions, even if \( X \) is bound to a ground pattern and
no narrowing is involved. Due to lazy evaluation, \( \text{check} \) can reject a partially eval-
uated candidate solution \( Y \). Thus, the scheme achieves incremental generation in-
terleaved with testing in a very concise declarative formulation. Experiments based
on the permutation sort algorithm ([13]) have shown significant time speedups of
\( \text{findSolution} \) 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 back-
tracking to compute the candidate solutions one by one. In both cases, the com-
putation 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 na"ive 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://rczart.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 advantages of higher-order programming 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


