
AN APPROACH TO DECLARATIVE PROGRAMMING BASED ON A REWRITING LOGIC

J.C.GONZÁLEZ-MORENO, M.T.HORTALÁ-GONZÁLEZ,
F.J.LÓPEZ-FRAGUAS, AND M.RODRÍGUEZ-ARTALEJO

- ▷ We propose an approach to declarative programming which integrates the functional and relational paradigms by taking possibly non-deterministic lazy functions as the fundamental notion. Classical equational logic does not supply a suitable semantics in a natural way. Therefore, we suggest to view programs as theories in a constructor-based conditional rewriting logic. We present proof calculi and a model theory for this logic, and we prove the existence of free term models which provide an adequate intended semantics for programs. We develop a sound and strongly complete lazy narrowing calculus, which is able to support sharing without the technical overhead of graph rewriting and to identify safe cases for eager variable elimination. Moreover, we give some illustrative programming examples, and we discuss the implementability of our approach.

KEYWORDS: Declarative programming, non-deterministic functions, constructor-based rewriting logic, lazy narrowing.

◁

This paper is a substantially extended and revised version of [21]. The authors have been partially supported by the spanish CICYT (project TIC 95-0433-C03-01 "CPD") and by the EU (ESPRIT BR Working Group EP 22457 "CCLII").

Address correspondence to Dpto. de Sist. Inf. y Prog. (DSIP). Fac. Matemáticas, Univ. Complutense de Madrid (UCM), Av. Complutense s/n, Madrid (SPAIN), E-28040.

E-mail: {jcmoreno,alp94teja,fraguas,mario}@sip.ucm.es

THE JOURNAL OF LOGIC PROGRAMMING

© Elsevier Science Inc., 1994
655 Avenue of the Americas, New York, NY 10010

0743-1066/94/\$7.00

1. INTRODUCTION

The interest in combining different declarative programming paradigms, especially functional and logic programming, has grown over the last decade; see [23] for a recent survey. The operational semantics of many functional logic languages is based on so-called *narrowing*, which combines the basic execution mechanisms of functional and logic languages, namely *rewriting* and *unification*. Several modern functional languages, as e.g. Haskell [44] allow *non-strict* functions, that may return a defined result even if some argument is not defined. The operational semantics of non-strict functional languages relies on a *lazy* reduction strategy, roughly equivalent to outermost rewriting (see [9]), which delays the evaluation of function arguments as much as possible. This feature has been adopted by some functional logic languages, such as K-LEAF [15] and BABEL [42]. These languages use so-called *constructor-based* term rewriting systems to model the behaviour of non-strict functions. A constructor-based term rewriting system classifies operation symbols in two categories: *defined functions*, whose behaviour is given by rewrite rules, and *constructors*, which are used to represent computed values as constructor terms. Another special feature of lazy functional logic languages is the distinction between ordinary equality $e = e'$ and *strict equality* $e == e'$. Strict equality means that e and e' can be reduced to the same constructor term (a finite, totally defined value), while ordinary equality does not exclude the possibility that the common value of e and e' may be infinite and/or partially defined. Typically, strict equations are used for building goals and conditions of conditional rewrite rules.

On the other hand, the usefulness of non-deterministic operations for algebraic specification and programming has been advocated by Hussmann [27, 28], who provides several examples (including the specification of semantics for communicating sequential processes) and leaves as an interesting open question ‘*the integration of non-strict operations (at least non-strict constructors)*’ (see [28], Section 8.2). In this paper, we propose a quite general approach to declarative programming, where possibly non-deterministic functions are taken as the fundamental notion. The main idea is to keep constructors deterministic, and to allow defined functions to return more than one constructor term as a computed result, for fixed constructor terms given as arguments. For instance, the following constructor-based rewrite rules define a function `merge` that merges two given lists non-deterministically in all possible ways. We use Prolog’s syntax for the list constructors.

$$\begin{array}{ll} \text{merge}([], Ys) & \rightarrow Ys \\ \text{merge}([X | Xs], []) & \rightarrow [X | Xs] \\ \text{merge}([X | Xs], [Y | Ys]) & \rightarrow [X | \text{merge}(Xs, [Y | Ys])] \\ \text{merge}([X | Xs], [Y | Ys]) & \rightarrow [Y | \text{merge}([X | Xs], Ys)] \end{array}$$

Given this definition, the function call `merge([1], [2,3])` is expected to return three possible results, namely `[1,2,3]`, `[2,1,3]` and `[2,3,1]`. More concretely, we expect a backtracking mechanism to search for the results and deliver them one after the other, as in typical Prolog systems. Note that search is possible in our setting even for ground goals. But, as we will see soon, goals including logic variables are also allowed.

Our approach gives a positive answer to Hussmann’s question, since both our con-

structors and our defined functions have a non-strict semantics. Deterministic functions are of course possible as a particular case. Therefore, our framework can exploit known advantages of determinism, such as dynamic cut [33] or simplification [24, 25]. Relations can be also modelled as boolean functions. However, the main reason for our choice of functions as the fundamental notion is that functions have often a better operational behaviour, thus helping to avoid divergent computations or to detect failure earlier. In the case of deterministic functions, these benefits can be obtained thanks to deterministic simplification by rewriting, as shown in [22, 23]. In the case of non-deterministic functions, similar benefits are still possible thanks to lazy evaluation. This idea has been already advocated in [1]; we will show two concrete examples in Section 2.

Technically, non-deterministic functions can be modelled by means of non-confluent constructor-based term rewriting systems, where a given term may be rewritten to constructor terms in more than one way. To express goals and conditions of conditional rewrite rules, strict equality is replaced by the more general notion of *joinability*: two terms \mathbf{a} , \mathbf{b} are regarded as joinable (in symbols, $\mathbf{a} \bowtie \mathbf{b}$) iff they can be rewritten to a common constructor term. For instance, a possible goal for the `merge` function is `merge(Xs,Ys) \bowtie [A,B]` for which the following four answers can be computed:

$$\begin{array}{ll} Xs = [], & Ys = [A,B] \\ Xs = [A,B], & Ys = [] \\ Xs = [A], & Ys = [B] \\ Xs = [B], & Ys = [A] \end{array}$$

Note that we are able to compute general solutions for this goal, while most previous approaches to strict equality [15, 42, 4, 37] attempt an (often infinite) enumeration of ground solutions. We have designed a *lazy narrowing calculus* that is sound and complete for solving arbitrary goals. In contrast to other lazy narrowing calculi, as e.g. those in [15, 42, 4, 24, 37], completeness of our calculus holds without any confluence or non-ambiguity hypothesis, even for conditional rewrite systems with extra variables in the conditions, which may cause incompleteness of narrowing w.r.t. the semantics of equational logic [40]. Previous works [15, 42, 19] have already shown how to overcome such incompleteness problems by adopting the restriction to left-linear, constructor-based rewrite rules (which are expressive enough for programming) and replacing algebraic equality in goals and conditions by strict equality. As novel points w.r.t. these papers, we allow non-determinism and we elaborate a logical framework, so-called *constructor-based rewriting logic*, which provides a declarative and model-theoretic semantics for programs. In particular, the intended semantics of an arbitrary program is given by a free term model.

The presence of non-determinism in programming languages raises semantic problems that are not always well understood. Many different semantics are possible according to the decisions taken for several independent issues, including:

- strict/non-strict functions;
- call-time choice /run-time choice for parameter passing;
- angelic/demonic/erratic view of non-deterministic choices

A nice discussion of these questions can be found in [49]. The semantic option in this paper corresponds to *angelic non-determinism* with *call-time choice* for *non-strict* functions. The ‘angelic’ view means, intuitively, that the results of all possible computations (due to different non-deterministic choices) are collected by the semantics, even if the possibility of infinite computations is not excluded. Technically, angelic non-determinism corresponds to the choice of one of the three main powerdomain constructions, so-called *Hoare’s powerdomain* [52]. The ‘call-time choice’ view, also adopted by Hussmann in [27, 28], has the following intuitive meaning: given a function call $f(\mathbf{e}_1, \dots, \mathbf{e}_n)$, one chooses some fixed (possibly *partial*) value for each of the actual parameters \mathbf{e}_i before applying the rewrite rules that define f ’s behaviour. This option seems to be the most natural way to combine non-deterministic choice with parameter passing and it must not be confused with strictness, since the complete evaluation of the terms \mathbf{e}_i to constructor terms in normal form is not required. The usual notion of rewriting, however, is not sound w.r.t. call-time choice semantics¹. The following example, inspired by [27], will clarify this point. Let us consider the following rewrite rules:

$$\begin{array}{ll} \text{coin} & \rightarrow 0 \\ \text{coin} & \rightarrow 1 \\ \text{double}(X) & \rightarrow X+X \end{array}$$

together with suitable rules for the addition operation $+$. According to call-time choice semantics, the term $\text{double}(\text{coin})$ should have 0 and 2, but not 1, as possible values. Unfortunately, an outermost rewriting derivation is able to compute the incorrect result 1:

$$\text{double}(\text{coin}) \rightarrow \text{coin} + \text{coin} \rightarrow 0 + \text{coin} \rightarrow 0+1 \rightarrow 1$$

Innermost rewriting is sound w.r.t. call-time choice, but it is not complete w.r.t. the semantics of non-strict functions. Therefore, we adopt another solution suggested in [28]: all those variables that have more than one occurrence in the right-hand side of some rewrite rule (such as X in the rewrite rule for double) must be shared. In [28, 47] sharing is realized by performing rewriting and narrowing over *term graphs* [8], which leads to a significant technical overhead. In our setting, the effect of sharing is built-in within our rewriting and narrowing calculi, in such a way that term graphs can be avoided.

For the sake of simplicity, we restrict our presentation to the unsorted case, but all our results can be extended to many-sorted signatures, or even to typed languages with parametric polymorphism; see [6, 7]. The paper is organized as follows: In the next section we give further examples attempting to motivate the interest of our approach. In Section 3 we recall some technical preliminaries. In Section 4 we introduce a *Constructor-based conditional ReWriting Logic* (CRWL) that formalizes the non-classical version of rewriting needed in our setting. Section 5 is concerned with the model-theoretic semantics for CRWL programs; this includes the existence of free term models, which are closely related to CRWL-provability. In Section 6 we present a *Constructor-based Lazy Narrowing Calculus* CLNC. In

¹This makes an important difference between our setting and a classical paper by Boudol [10], which is based on classical rewriting. Moreover, Boudol’s work does not deal with narrowing.

Section 7 we establish the soundness and strong completeness of CLNC w.r.t. the model-theoretic semantics of CRWL. In Section 8 we discuss possible refinements of CLNC, based on demand-driven strategies, that can be used to build efficient implementations. Finally, in Section 9 we summarize our conclusions.

2. MOTIVATING EXAMPLES

Non-determinism is one of the main novelties of our proposal. Therefore, we include here two small CRWL-programs attempting to show how (lazy) non-deterministic functions contribute to a more clear, concise and thus productive declarative programming style. For a discussion of the general advantages of functional logic programming w.r.t. relational logic programming, the reader is referred to [23].

In the first example we program a small parser for a grammar of simple arithmetic expressions. In the second one we show how a typically inefficient ‘generate and test’ logic program can be converted into a more efficient functional logic program by exploiting the combination of non-deterministic functions and lazy evaluation. Both examples use a hopefully self-explanatory syntax for constructor-based, conditional rewrite rules, which is formally explained in Section 4.

A parser

BNF-descriptions of grammars have most of the times some degree of non-determinism, because of the presence of different alternatives for the same non-terminal symbol. For this reason, the task of writing a parser is simplified if a language supporting some kind of non-deterministic computations is used. This happens, for instance, with logic programming, for which the writing of a parser is one of the nicest examples of declarative programming. The standard translation of a BNF-rule for a non-terminal symbol s into a clause for a predicate $s(In, Out)$ (see, e.g., [50]) gives as result a logic program which is quite similar to the original BNF-description. The similarity is even increased with the hiding of the arguments In, Out by means of the formalism of DCG’s (definite clause grammars), but in this case a preprocessing is needed for obtaining an executable program².

In our setting, the use of non-deterministic functions allows a formulation of BNF-rules even more natural than in the case of logic programs: no extra arguments nor preprocessing is needed. As a concrete situation we consider a grammar for simple arithmetic expressions (using only 0 and 1, for further simplicity), given by the following BNF-rules (terminals are enclosed by ‘ ’ and | indicates a non-deterministic alternative):

```

expression ::= term
expression ::= term ('+' | '-') expression

term      ::= factor
term      ::= factor ('*' | '/') term
factor    ::= '0' | '1'

```

²The preprocessing is performed automatically by many Prolog systems.

factor ::= '(expression)'

For writing a CRWL-parser for this grammar, we assume that the input is given in the form of a list of *tokens* of the form 0,1, +,-,*,/, (,) (which are all constants, i.e., constructors of arity 0). We model each non-terminal *s* by a non-deterministic function (of arity 0) returning a piece of input (a list of tokens). The set of possible values to which *s* reduces is the set of lists of tokens recognized by *s*. For expressing alternatives we introduce the non-deterministic function 'par excellence' //, used in infix notation³ and defined by the rules:

$$\begin{aligned} X // Y &\rightarrow X \\ X // Y &\rightarrow Y \end{aligned}$$

Sequencing of symbols in right-hand sides of BNF-rules is expressed by means of list concatenation, which is defined as a function ++ (used in infix notation, and associating to the right) with rules:

$$\begin{aligned} [] ++ Ys &\rightarrow Ys \\ [X | Xs] ++ Ys &\rightarrow [X | Xs ++ Ys] \end{aligned}$$

According to this, terminals must appear as an explicit list of tokens (the same happens in DCG's). This results in the following CRWL-rules for the grammar:

$$\begin{aligned} \text{expression} &\rightarrow \text{term} \\ \text{expression} &\rightarrow \text{term} ++ [+ // -] ++ \text{expression} \\ \text{term} &\rightarrow \text{factor} \\ \text{term} &\rightarrow \text{factor} ++ [* // /] ++ \text{term} \\ \text{factor} &\rightarrow [0 // 1] \\ \text{factor} &\rightarrow [(// expression //)] \end{aligned}$$

Given this program, `expression` yields all the expressions accepted by the grammar (as the results of alternative computations). Therefore, we can solve a variety of goals, as e.g.

- `tokenList` \bowtie `expression`, where `tokenList` is any concrete list of tokens. This goal will succeed if `tokenList` represents a correct arithmetic expression, accepted by the grammar.
- `[T1, T2, T3, T4, T5, T6, T7]` \bowtie `expression`, where `Ti` are logic variables. This goal will have several solutions, representing all the well-formed arithmetic expressions that can be built with seven tokens.

Permutation sort

A quite usual example of a very concise, descriptive, but inefficient logic program is *permutation sort*: the list `L'` is the result of sorting `L` if `L'` is a permutation of `L` and `L'` is sorted. The most immediate formulation of this idea leads to the following naive 'generate and test' logic program:

```
permutation_sort(L,L') :- permute(L,L'), sorted(L').
```

³We do not use | for avoiding confusion with the bar of lists.

where `permute` and `sorted` are defined in the usual way.

Of course this is not a good method for sorting lists, but the interesting point to discuss here is the reason of its *extreme* inefficiency when the program is executed under Prolog's standard computation rule: every candidate solution `L'` is completely generated before it is tested. An extension of Prolog's computation model by a coroutining mechanism [43] has been proposed to solve this problem. The coroutining approach requires the explicit introduction of 'wait' declarations for some predicates.

On the other hand, in order to solve generate-and-test problems in a lazy functional language (where there is no built-in search for solutions), one would typically follow the 'list of successes' approach [51]: generate the list of all candidate solutions (all permutations, in this case) and filter it by means of the tester. Although lazy evaluation ensures that the list of candidates is generated only to the extent required by the tester (which can reject a partially generated solution), in any case it can be a very large structure. Moreover, some special language constructions, such as *list comprehension* [9], are usually needed to program the generation of the candidates' list in a declaratively neat way.

In our setting, we can use a non-deterministic function to describe the generation of candidate solutions (i.e., permutations) in a concise and declarative way. Since candidate solutions are generated one by one, we can avoid the computation of a bigger structure (namely, the list of all candidates) without loss of completeness. At the same time, we avoid the inefficiency of the naive logic program, because lazy evaluation will ensure that the generation of each particular permutation will be interrupted as soon as `sorted` recognizes that it cannot lead to an ordered list. This combination of a lazy, non-deterministic generator and a lazy tester can be described as the 'lazy generate and test' approach.

More precisely, our generator `permute` is defined as follows:

```
permute([])      → []
permute([X | Xs]) → insert(X, permute(Xs))
insert(X,Ys)    → [X | Ys]
insert(X,[Y | Ys]) → [Y | insert(X,Ys)]
```

Note that `permute` is a non-deterministic function, due to the auxiliary function `insert`. To each of the permutations produced by `permute(L)`, say `L'`, we want to apply the test `sorted(L')` and return `L'` as final result in case of success. If the test fails (remember: this may happen even if `permute(L)` has been only partially evaluated) this particular computation fails, and another possible value of `permute(L)` must be tried. Using some kind of CRWL-pseudocode with a 'where-construction' typical of functional programming, we could then write:

$$\text{sort}(L) \rightarrow L' \Leftarrow \text{sorted}(L') \bowtie \text{true} \quad \text{where } L' = \text{permute}(L)$$

In the following true CRWL-program we 'lift' the 'where-construction' by means of an auxiliary function `check`:

$$\begin{aligned} \text{sort}(L) &\rightarrow \text{check}(\text{permute}(L)) \\ \text{check}(L') &\rightarrow L' \Leftarrow \text{sorted}(L') \bowtie \text{true} \end{aligned}$$

The definition of the tester `sorted` is the natural one: if we assume that lists consist of natural numbers represented by means of the constructors `zero,suc`, then we can define:

$$\begin{aligned} \text{sorted}([]) &\rightarrow \text{true} \\ \text{sorted}([X]) &\rightarrow \text{true} \\ \text{sorted}([X,X' | Xs]) &\rightarrow \text{true} \Leftarrow \text{leq}(X,X') \bowtie \text{true}, \text{sorted}([X' | Xs]) \bowtie \text{true} \\ \text{leq}(\text{zero},Y) &\rightarrow \text{true} \\ \text{leq}(\text{suc}(X),\text{zero}) &\rightarrow \text{false} \\ \text{leq}(\text{suc}(X),\text{suc}(Y)) &\rightarrow \text{leq}(X,Y) \end{aligned}$$

This completes the program. Note that call-time choice is essential for its correct behaviour. More precisely, since `sort(L)` calls `check(permute(L))`, call-time choice is needed to ensure that both occurrences of `L'` in the right-hand side of the rewrite rule for `check` refer to the same permutation of `L`.

To sort any given concrete list *list*, it is now sufficient to solve `sort(list) \bowtie SortedList`, which will succeed and bind the logic variable `SortedList` to the desired result.

Note that the technique described in this subsection can be applied to any generate and test problem. In particular, the parsing goal `tokenList \bowtie expression` shown in the preceding subsection can be also viewed as a generate and test problem, where `expression` is the generator and the joinability comparison with the given `tokenList` acts as a tester. Exactly as in the permutation sort example, many candidate token lists produced by `expression`, but different from `tokenList`, will be discarded without generating them completely.

3. TECHNICAL PRELIMINARIES

The reader is assumed to have some familiarity with the basics of logic programming [5, 31] and term rewriting [12, 30]. We will also need some notions related to semantic domains [48]. This section intends to make the paper more selfcontained. We fix basic notions, terminology and notations, to be used in the rest of the paper.

Posets and CPOs

A *partially ordered set* (in short, *poset*) with *bottom* \perp is a set S equipped with a partial order \sqsubseteq and a least element \perp (w.r.t. \sqsubseteq). We say that an element $x \in S$ is *totally defined* iff 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 $\perp \in A$ and A is *downclosed*, 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 $\mathcal{C}(S), \mathcal{I}(S)$ for the sets of cones and ideals of S respectively. The set $\bar{S} =_{\text{def}} \mathcal{I}(S)$, equipped with the set-inclusion \subseteq as ordering, is a poset with bottom, called the *ideal completion* of S . Note that \bar{S} 's bottom is precisely $\{\perp\}$, where \perp is S 's bottom. 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 , $\langle x \rangle =_{def} \{y \in S : y \sqsubseteq x\} \in \bar{S}$.

A poset with bottom C is a *complete partial order* (in short, *cpo*) iff D has a least upper bound $\sqcup D$ (also called *limit*) for every directed set $D \subseteq C$. In particular, $\sqcup \emptyset = \perp$. An element $u \in C$ is called a *finite* element iff whenever $u \sqsubseteq \sqcup D$ for a non-empty directed D , there exists $x \in D$ with $u \sqsubseteq x$. Moreover, u is called *total* iff u is maximal w.r.t. \sqsubseteq , and *partial* otherwise. A cpo C is called *algebraic* iff any element of C is the limit of a directed set of finite elements. For any poset with bottom S , its ideal completion \bar{S} turns out to be the least cpo including S . Furthermore, \bar{S} is an algebraic cpo whose finite elements are precisely the principal ideals $\langle x \rangle$, $x \in S$; see e.g. [38]. Note that elements $x \in \text{Def}(S)$ correspond to finite and total elements $\langle x \rangle$ in the ideal completion.

Algebraic cpos are commonly used as semantic domains for the denotational semantics of programming languages [48]. The partial order is interpreted as an approximation ordering between partially defined values; i.e. $x \sqsubseteq y$ is understood as the statement that y is more defined than x . Infinite values such as infinite lists or functions (in the case of higher-order languages) can be captured as limits of directed sets of finite partial values. In this paper, we will work with posets instead of algebraic cpos, because this simplifies technicalities. Any given poset S must be imagined as an incomplete semantic domain, which provides only finite semantic values. The ideal completion \bar{S} supplies the missing infinite values. As we will see in Section 5, finite values are enough for describing the semantics of our programs.

Signatures, terms and C-terms

A *signature with constructors* is a countable set $\Sigma = \text{DC}_\Sigma \cup \text{FS}_\Sigma$, where $\text{DC}_\Sigma = \bigcup_{n \in \mathbb{N}} \text{DC}_\Sigma^n$ and $\text{FS}_\Sigma = \bigcup_{n \in \mathbb{N}} \text{FS}_\Sigma^n$ are disjoint sets of *constructor* and *defined function symbols* respectively, each of them with associated *arity*. We assume a countable set \mathcal{V} of *variables*, and we omit explicit mention of Σ in the subsequent notations. We write Term for the set of terms built up with aid of Σ and \mathcal{V} , and we distinguish the subset CTerm of those terms (called *constructor terms*, shortly *C-terms*) which only make use of DC and \mathcal{V} . We will need sometimes to enhance Σ with a new constant (0-arity constructor) \perp , obtaining a new signature Σ_\perp ⁴. We will write Term_\perp and CTerm_\perp for the corresponding sets of terms in this extended signature, the so-called *partial terms*. As frequent notational conventions we will also use $c, d \in \text{DC}$; $f, g \in \text{FS}$; $s, t \in \text{CTerm}_\perp$; $a, b, e \in \text{Term}_\perp$. Moreover, $\text{var}(e)$ will be used for the set of variables occurring in the term e .

A natural *approximation ordering* \sqsubseteq for partial terms can be defined as the least partial ordering over Term_\perp satisfying the following properties:

- $\perp \sqsubseteq e$, for all $e \in \text{Term}_\perp$
- $e_1 \sqsubseteq e'_1, \dots, e_n \sqsubseteq e'_n \Rightarrow h(e_1, \dots, e_n) \sqsubseteq h(e'_1, \dots, e'_n)$,
for all $h \in \text{DC}^n \cup \text{FS}^n$, $e_i, e'_i \in \text{Term}_\perp$

⁴Semantically, the symbol \perp is intended to denote the bottom element, also written as \bot by an abuse of notation.

We are particularly interested in the restriction of \sqsubseteq to the set \mathbf{CTerm}_\perp of partial C-Terms. The ideal completion of \mathbf{CTerm}_\perp is isomorphic to a cpo whose elements are possibly infinite trees with nodes labelled by symbols from $\mathbf{DC} \cup \{\perp\}$ in such a way that the arity of each label corresponds to the number of sons of the node; see [16]. For instance, if we assume a signature that includes the constructors `zero`, `suc` for natural numbers, as well as list constructors (written in Prolog syntax), we can obtain the infinite list of all natural numbers as limit of the following chain of partial C-terms:

$$\perp \sqsubseteq [\text{zero} \mid \perp] \sqsubseteq [\text{zero}, \text{suc}(\text{zero}) \mid \perp] \sqsubseteq [\text{zero}, \text{suc}(\text{zero}), \text{suc}(\text{suc}(\text{zero})) \mid \perp] \sqsubseteq \dots$$

Substitutions

C-substitutions are mappings $\theta : \mathcal{V} \rightarrow \mathbf{CTerm}$ which have $\hat{\theta} : \mathbf{Term} \rightarrow \mathbf{Term}$ as unique natural extension, also noted as θ . The set of all C-substitutions is noted as \mathbf{CSubst} . The bigger set \mathbf{CSubst}_\perp of all partial C-substitutions $\theta : \mathcal{V} \rightarrow \mathbf{CTerm}_\perp$ is defined analogously. We note as $t\theta$ the result of applying the substitution θ to the term t , and we define the composition $\sigma\theta$ such that $t(\sigma\theta) \equiv (t\sigma)\theta$. As usual, the *domain* of a substitution θ is defined as $\text{dom}(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$, and $\theta = \{X_1 / t_1, \dots, X_n / t_n\}$ stands for the substitution that satisfies $X_i\theta \equiv t_i$ ($1 \leq i \leq n$) and $Y\theta \equiv Y$ for all $Y \in \mathcal{V} \setminus \{X_1, \dots, X_n\}$. A substitution θ such that $\theta\theta = \theta$ is called *idempotent*. The approximation ordering over \mathbf{CTerm}_\perp induces a natural approximation ordering over \mathbf{CSubst}_\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 \mathbf{CSubst}_\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}).

4. A CONSTRUCTOR-BASED CONDITIONAL REWRITING LOGIC

As we have seen in Section 1, our intended semantics embodies *angelic non-determinism* with *call-time choice* for *non-strict* functions, and the usual notion of rewriting is unsound w.r.t. such a semantics. Therefore, we will use a special proof system called *Constructor-based conditional ReWriting Logic* - CRWL, for short - to formalize the non-classical version of rewriting needed for our purposes. In the rest of the paper, we will see that CRWL can be equipped with a natural model theory, and used as a logical basis for declarative programming.

Assume any fixed signature with constructors $\Sigma = \mathbf{DC} \cup \mathbf{FS}$. CRWL-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:

$$\underbrace{f(\bar{t})}_{\text{left hand side (l)}} \rightarrow \underbrace{r}_{\text{right hand side}} \Leftarrow \underbrace{C}_{\text{Condition}}$$

where $f \in \mathbf{FS}^n$, \bar{t} must be a linear n-tuple of C-terms $t_i \in \mathbf{CTerm}$ and the condition C must consist of finitely many (possibly zero) joinability statements $a \bowtie b$ with $a, b \in \mathbf{Term}$. As usual (see e.g. [12]) “ \bar{t} linear” means that each variable occurring in \bar{t} must have a single occurrence. In the sequel we use the following notation for possibly partial C-instances of rewrite rules:

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

From a given CRWL-program \mathcal{R} we wish to be able to derive statements of the following two kinds:

- *Reduction statements:* $a \rightarrow b$, with $a, b \in \text{Term}_{\perp}$.
- *Joinability statements:* $a \bowtie b$, with $a, b \in \text{Term}_{\perp}$.

The intended meaning of a reduction statement $a \rightarrow b$ is that a can be reduced to b , where ‘reduction’ includes the possibility of applying rewrite rules from \mathcal{R} or replacing some subterms of a by \perp . We are particularly interested in reduction statements of the form $a \rightarrow t$ with $t \in \text{CTerm}_{\perp}$, which we will call *approximation statements*. One such statement is intended to mean that t represents one of the values that must be collected in order to obtain the non-deterministic semantics of a . Note that t can be a partial C-term; this means that we are aiming at a non-strict semantics.

On the other hand, the intended meaning of a joinability statement $a \bowtie b$ is that a and b can be both reduced to some common totally defined value; in other words, $a \bowtie b$ will hold if we can prove $a \rightarrow t$ and $b \rightarrow t$ for some common *total* C-term $t \in \text{CTerm}$. Note that, according to this idea, \bowtie behaves indeed as a generalization of strict equality.

A formal specification of CRWL-derivability from a program \mathcal{R} is given by the following rewriting calculus:

Definition 4.1. (Basic Rewriting Calculus -BRC-)

$$\begin{array}{ll}
\mathbf{B} & \text{Bottom: } e \rightarrow \perp \\
\mathbf{MN} & \text{Monotonicity: } \frac{e_1 \rightarrow e'_1 \dots e_n \rightarrow e'_n}{h(e_1, \dots, e_n) \rightarrow h(e'_1, \dots, e'_n)} \quad \text{for } h \in \text{DC}^n \cup \text{FS}^n. \\
\mathbf{RF} & \text{Reflexivity: } e \rightarrow e \\
\mathbf{R} & \text{Reduction: } \frac{C}{l \rightarrow r} \quad \text{for any instance } (l \rightarrow r \Leftarrow C) \in [\mathcal{R}]_{\perp}. \\
\mathbf{TR} & \text{Transitivity: } \frac{e \rightarrow e' \quad e' \rightarrow e''}{e \rightarrow e''} \\
\mathbf{J} & \text{Join: } \frac{a \rightarrow t \quad b \rightarrow t}{a \bowtie b} \quad \text{if } t \in \text{CTerm}.
\end{array}$$

□

Some comments about this calculus are in order. Rules **MN**, **RF**, **R** and **TR** reflect the usual behaviour of rewriting, except that **R** allows to apply only (partial) C-instances of rewrite rules, while traditional rewriting would allow arbitrary instances. At this point, BRC reflects our option for non-strict functions and call-time choice (an explicit sharing mechanism is not needed, because C-instances are built from C-terms, that represent computed - possibly partial - values). Rule **B** enables to derive approximation statements $a \rightarrow t$ with partial t ; the need to collect such statements is also due to non-strictness. Finally, rule **J** obviously reflects the

intended meaning of joinability, as explained above. Note that **J** requires t to be total. Otherwise, we would obtain a useless notion, since $a \bowtie b$ would follow from **B**, for arbitrary terms a, b . As a concrete example of BRC-derivability, assume that \mathcal{R} includes the rewrite rule:

$$\text{from}(N) \rightarrow [N \mid \text{from}(\text{succ}(N))]$$

Then, the following approximation statements can be derived from \mathcal{R} in BRC:

$$\begin{aligned} \text{from}(\text{zero}) &\rightarrow \perp \\ \text{from}(\text{zero}) &\rightarrow [\text{zero} \mid \perp] \\ \text{from}(\text{zero}) &\rightarrow [\text{zero}, \text{succ}(\text{zero}) \mid \perp] \\ &\dots \end{aligned}$$

Note that $\{ t \in \text{CTerm}_\perp \mid \mathcal{R} \vdash_{\text{BRC}} \text{from}(\text{zero}) \rightarrow t \}$ is a directed set, whose limit (in the ideal completion of CTerm_\perp) represents the infinite list of all natural numbers.

At this point we can compare CRWL with another well known approach to rewriting as logical deduction, namely Meseguer's *Rewriting Logic* [35] (shortly RWL, in what follows), which has been also used as a basis for computational systems and languages such as ELAN [29] and Maude [36]. In spite of some obvious analogies, there are several clear differences regarding both the intended applications and the semantics. CRWL intends to model the evaluation of terms in a constructor-based language including non-strict and possibly non-deterministic functions, so that it can serve as a logical basis for declarative programming languages involving lazy evaluation. On the other hand, RWL was originally proposed with broader aims, as a logical framework in which other logics could be represented, as well as a semantic framework for the specification of languages and (possibly concurrent) systems. Accordingly, RWL is not constructor-based and lacks the analogon to our rule **B**. Moreover, RWL relies on a more general notion of rewriting, namely rewriting modulo equational axioms (typically, associativity and/or commutativity of some operators), intended to provide a structural equivalence between terms. Finally, RWL adopts run-time choice rather than call-time choice. This option corresponds to the classical behaviour of rewriting, using arbitrary instances of the rewrite rules, in contrast to our rule **R**. We believe that call-time choice is a more convenient option for programming purposes; the discussion of the permutation sort example in Section 2 has provided some evidence to this point. Moreover, call-time choice leads to an implementation based on sharing, which is the most efficient choice for a lazy language. See the discussion in Section 1.

In order to use CRWL as a logical basis for declarative programming, it is convenient to introduce a second rewriting calculus GORC which allows to build *goal-oriented* proofs of approximation and joinability statements. Goal-oriented proofs have the property that the outermost syntactic structure of the statement to be proved determines the inference rule which must be applied at the last proof step; in this way, the structure of the proof is determined by the structure of the goal. This will provide a very helpful technical support for proving the completeness of a goal solving procedure based on lazy narrowing; see Section 7.

We will prove that BRC and GORC have the same deduction power for deriving

approximation and joinability statements. The fact that BRC-provable statements always have GORC-proofs bears some analogy to the existence of so-called *uniform proofs* in abstract logic programming languages based on sequent calculi [41]. The formal presentation of GORC is as follows:

Definition 4.2. (Goal-Oriented Rewriting Calculus –GORC–)

- B** *Bottom:* $e \rightarrow \perp$.
- RR** *Restricted Reflexivity:* $X \rightarrow X$, for $X \in \mathcal{V}$.
- DC** *Decomposition:*
$$\frac{e_1 \rightarrow t_1 \dots e_n \rightarrow t_n}{c(e_1, \dots, e_n) \rightarrow c(t_1, \dots, t_n)}$$
,
for $c \in \text{DC}^n$, $t_i \in \text{CTerm}_\perp$.
- OR** *Outer Reduction:*
$$\frac{e_1 \rightarrow t_1 \dots e_n \rightarrow t_n \quad C \quad r \rightarrow t}{f(e_1, \dots, e_n) \rightarrow t}$$
,
if $t \neq \perp$, $(f(t_1, \dots, t_n) \rightarrow r \Leftarrow C) \in [\mathcal{R}]_\perp$.
- J** *Join:* $\frac{a \rightarrow t \quad b \rightarrow t}{a \bowtie b}$, if $t \in \text{CTerm}$. □

Note that GORC can derive only such reduction statements that are in fact approximation statements. The following proposition ensures the desired equivalence between BRC and GORC:

Proposition 4.1. (Calculi Equivalence)

For any program \mathcal{R} , the calculi BRC and GORC derive the same approximation and joinability statements.

PROOF.

Let \mathcal{R} (program) and φ (approximation or joinability statement) be given. We will show that $\mathcal{R} \vdash_{\text{BRC}} \varphi$ iff $\mathcal{R} \vdash_{\text{GORC}} \varphi$.

The “if” part holds because any step within a given GORC-proof can be easily replaced by one or several BRC-steps. This is obvious for **B-**, **RR-**, **DC-** and **J-**steps, while any **OR-**step can be replaced by four BRC-steps according to the following scheme⁵:

$$\text{TR} \frac{\frac{\text{MN} \frac{f(e_1, \dots, e_n) \rightarrow f(t_1, \dots, t_n)}{e_1 \rightarrow t_1, \dots, e_n \rightarrow t_n} \quad \text{TR} \frac{f(t_1, \dots, t_n) \rightarrow t}{f(t_1, \dots, t_n) \rightarrow r} \quad \text{R} \frac{f(t_1, \dots, t_n) \rightarrow r}{C} \quad r \rightarrow t}{f(e_1, \dots, e_n) \rightarrow t}}$$

The “only if” part can be proved by induction on the structure of BRC-derivations. The induction relies on the following key observation: Any given BRC-proof for:

$$\mathcal{R} \vdash_{\text{BRC}} f(e_1, \dots, e_n) \rightarrow t$$

(where t is not \perp) determines a rewriting⁶ sequence of the form

⁵Here and in the sequel, we draw BRC and GORC proofs as trees growing downwards, where each node corresponds to the conclusion of some inference rule whose premises correspond to the node’s children.

⁶Here, ‘rewriting sequence’ refers to the usual notion of term rewriting; see e.g. [12].

$$f(\mathbf{e}_1, \dots, \mathbf{e}_n) \rightarrow^* f(\mathbf{t}_1, \dots, \mathbf{t}_n) \rightarrow r \rightarrow^* \mathbf{t}$$

where each rewrite step applies some rewriting rule from $[\mathcal{R}]_{\perp}$, or a rewrite rule of the form $\mathbf{e} \rightarrow \perp$. In particular the rewrite step $f(\mathbf{t}_1, \dots, \mathbf{t}_n) \rightarrow r$ will correspond to some instance $f(\mathbf{t}_1, \dots, \mathbf{t}_n) \rightarrow r \Leftarrow C \in [\mathcal{R}]_{\perp}$. By induction hypothesis we can assume that:

$$\begin{aligned} \mathcal{R} \vdash_{\text{GORC}} \mathbf{e}_i \rightarrow \mathbf{t}_i & \quad (1 \leq i \leq n) \\ \mathcal{R} \vdash_{\text{GORC}} \mathbf{a} \bowtie \mathbf{b}, \text{ for each } \mathbf{a} \bowtie \mathbf{b} \text{ in } C & \\ \mathcal{R} \vdash_{\text{GORC}} r \rightarrow \mathbf{t} & \end{aligned}$$

because each of the statements has a shorter BRC-proof than $f(\mathbf{e}_1, \dots, \mathbf{e}_n) \rightarrow \mathbf{t}$. Then we can conclude that $\mathcal{R} \vdash_{\text{GORC}} f(\mathbf{e}_1, \dots, \mathbf{e}_n) \rightarrow \mathbf{t}$ by applying rule **OR**. \square

In the rest of the paper, the notation $\mathcal{R} \vdash_{\text{CRWL}} \varphi$ will mean provability of φ (an approximation or joinability statement) in any of the calculi BRC or GORC.

We close this section with a technical result which will be used in Sections 5 and 7 below.

Lemma 4.1. (Monotonicity Lemma)

Let \mathcal{R} be a program, $\mathbf{e} \in \text{Term}_{\perp}$, $\theta, \theta' \in C\text{Subst}_{\perp}$, and $\mathbf{t} \in C\text{Term}_{\perp}$. If $\theta \sqsubseteq \theta'$ and Π is a GORC-proof of $\mathcal{R} \vdash_{\text{CRWL}} \mathbf{e}\theta \rightarrow \mathbf{t}$, there exists a GORC-proof Π' of $\mathcal{R} \vdash_{\text{CRWL}} \mathbf{e}\theta' \rightarrow \mathbf{t}$ with the same length and structure as Π .

PROOF.

First, we note that $\theta \sqsubseteq \theta'$ entails $\mathbf{e}\theta \sqsubseteq \mathbf{e}\theta'$. Therefore, we can assume $\mathbf{e}\theta \equiv \mathbf{a} \sqsubseteq \mathbf{a}' \equiv \mathbf{e}\theta'$. We reason by induction on the size of Π , measured as the number of GORC-inference steps.

Base case ($n=0$): Π must have one of the two forms:

$$\mathbf{B} \quad \mathbf{a} \rightarrow \perp \quad \text{or} \quad \mathbf{RR} \quad \mathbf{a} \rightarrow \mathbf{a}$$

In the first case we can take $\mathbf{B} \quad \mathbf{a}' \rightarrow \perp$ as Π' . In the second case \mathbf{a} must be a variable. Then $\mathbf{a} \sqsubseteq \mathbf{a}'$ entails $\mathbf{a} \equiv \mathbf{a}'$ and Π itself can be taken as Π' .

Inductive case ($n>0$): We distinguish two subcases according to the GORC-rule used for the last inference step in Π .

- Rule **DC**. Then Π must be of the form:

$$\mathbf{DC} \quad \frac{\Pi_1 \dots \Pi_n}{\mathbf{a} \equiv c(\mathbf{a}_1, \dots, \mathbf{a}_n) \rightarrow c(\mathbf{t}_1, \dots, \mathbf{t}_n)}$$

where Π_i is a GORC-proof for $\mathcal{R} \vdash_{\text{CRWL}} \mathbf{a}_i \rightarrow \mathbf{t}_i$ ($1 \leq i \leq n$). Then we can build Π' as

$$\mathbf{DC} \quad \frac{\Pi'_1 \dots \Pi'_n}{\mathbf{a}' \equiv c(\mathbf{a}'_1, \dots, \mathbf{a}'_n) \rightarrow c(\mathbf{t}_1, \dots, \mathbf{t}_n)}$$

where the GORC-proofs Π'_i for $\mathcal{R} \vdash_{\text{CRWL}} \mathbf{a}'_i \rightarrow \mathbf{t}_i$ do exist by induction hypothesis.

- Rule **OR**. In this case the structure of Π is:

$$\text{OR} \quad \frac{\Pi_1 \dots \Pi_n \quad \Delta \quad \Gamma}{\mathbf{a} \equiv f(\mathbf{a}_1, \dots, \mathbf{a}_n) \rightarrow \mathbf{t}}$$

where

- Π_i is a GORC-proof for $\mathcal{R} \vdash_{\text{CRWL}} \mathbf{a}_i \rightarrow \mathbf{t}_i$ ($1 \leq i \leq n$)
- Δ is a GORC-proof for $\mathcal{R} \vdash_{\text{CRWL}} \mathbf{C}$
- Γ is a GORC-proof for $\mathcal{R} \vdash_{\text{CRWL}} r \rightarrow \mathbf{t}$
- $f(\mathbf{t}_1, \dots, \mathbf{t}_n) \rightarrow r \Leftarrow \mathbf{C} \in [\mathcal{R}]_{\perp}$

Thus, by induction hypothesis we can assume GORC-proofs Π'_i for $\mathcal{R} \vdash_{\text{CRWL}} \mathbf{a}'_i \rightarrow \mathbf{t}_i$ ($1 \leq i \leq n$) and build Π' as

$$\text{OR} \quad \frac{\Pi'_1 \dots \Pi'_n \quad \Delta \quad \Gamma}{\mathbf{a}' \equiv f(\mathbf{a}'_1, \dots, \mathbf{a}'_n) \rightarrow \mathbf{t}}$$

□

5. MODEL-THEORETIC SEMANTICS FOR CRWL-PROGRAMS

In this section we define models for CRWL and we establish soundness and completeness of CRWL-provability w.r.t. semantic validity in models. Moreover, we prove that every program has a free term model, which can be seen as a generalization of \mathcal{C} -semantics [14] for Horn clause programs.

CRWL-Algebras

In Section 3 we have explained that the elements of a poset S can be viewed as finite approximations of values of a semantic domain D , that can be obtained from S by adding limit elements via an ideal completion construction. Therefore, we will use models with posets as carriers. In any such model, we will interpret function symbols as monotonic mappings taking elements as arguments and returning as result a cone of elements rather than a single element, because of the possibility of call-time choice non-determinism. A technical justification for the use of cones is the construction of Hoare's powerdomain [48, 52], where cones of finite elements of a given domain D , partially ordered by set inclusion, become the elements of Hoare's powerdomain $\mathcal{P}(D)$. In fact, when applying ideal completion to our models, monotonic mappings from elements to cones become continuous mappings from domains to Hoare's powerdomains. The poset-based presentation turns out to be convenient for all our purposes. Deterministic functions, in this setting, can be viewed as those mappings from elements to cones that return directed cones (i.e., ideals) as results. This is because ideals correspond to (possibly infinite) elements in the ideal completion.

Technically, we need the following definition:

Definition 5.1. (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:
 $[D \rightarrow_n E] =_{def} \{ f: D \rightarrow \mathcal{C}(E) \mid \forall u, u' \in D: (u \sqsubseteq u' \Rightarrow f(u) \sqsubseteq f(u')) \}$.
- The set of all *deterministic* functions from D to E as:
 $[D \rightarrow_d E] =_{def} \{ f \in [D \rightarrow_n E] \mid \forall u \in D: f(u) \in \mathcal{I}(E) \}$.

□

Note also that any non-deterministic function $f \in [D \rightarrow_n E]$ can be extended to a monotonic mapping $\hat{f}: \mathcal{C}(D) \rightarrow \mathcal{C}(E)$ defined by $\hat{f}(C) =_{def} \bigcup_{u \in C} f(u)$. The behaviour of \hat{f} according to this definition reflects call-time choice. By an abuse of notation, we will note \hat{f} also as f in the sequel.

Now we can define the class of algebras which will be used as models for CRWL:

Definition 5.2. (CRWL-Algebras)

For any given signature, *CRWL-algebras* are algebraic structures of the form:

$$\mathcal{A} = (D_{\mathcal{A}}, \{ c^{\mathcal{A}} \}_{c \in DC}, \{ f^{\mathcal{A}} \}_{f \in FS})$$

where $D_{\mathcal{A}}$ is a poset, $c^{\mathcal{A}} \in [D_{\mathcal{A}}^n \rightarrow_d D_{\mathcal{A}}]$ for $c \in DC^n$, and $f^{\mathcal{A}} \in [D_{\mathcal{A}}^n \rightarrow_n D_{\mathcal{A}}]$ for $f \in FS^n$. For $c^{\mathcal{A}}$ we still require the following additional condition:

For all $u_1, \dots, u_n \in D_{\mathcal{A}}$ there is $v \in D_{\mathcal{A}}$ such that $c^{\mathcal{A}}(u_1, \dots, u_n) = \langle v \rangle$. Moreover, $v \in \text{Def}(D_{\mathcal{A}})$ in case that all $u_i \in \text{Def}(D_{\mathcal{A}})$.

□

The additional condition required for the interpretation of constructors means that constructors must be interpreted as deterministic mappings that map finite (and total) elements into finite (and total) elements. To understand this, recall the characterization of finite and total elements in the ideal completion of a poset, given in Section 3.

The next definition shows how to evaluate terms in CRWL-algebras:

Definition 5.3. (Term evaluation)

Let \mathcal{A} be a CRWL-algebra of signature Σ . A *valuation* over \mathcal{A} is any mapping $\eta: \mathcal{V} \rightarrow D_{\mathcal{A}}$, and we say that η is *totally defined* iff $\eta(X) \in \text{Def}(D_{\mathcal{A}})$ for all $X \in \mathcal{V}$. We denote by $\text{Val}(\mathcal{A})$ the set of all valuations, and by $\text{DefVal}(\mathcal{A})$ the set of all totally defined valuations. The *evaluation* of a partial term $e \in \text{Term}_{\perp}$ in \mathcal{A} under η yields $\llbracket e \rrbracket^{\mathcal{A}} \eta \in \mathcal{C}(D_{\mathcal{A}})$ which is defined recursively as follows:

- $\llbracket \perp \rrbracket^{\mathcal{A}} \eta =_{def} \langle \perp_{\mathcal{A}} \rangle$.
- $\llbracket X \rrbracket^{\mathcal{A}} \eta =_{def} \langle \eta(X) \rangle$, for $X \in \mathcal{V}$.
- $\llbracket h(e_1, \dots, e_n) \rrbracket^{\mathcal{A}} \eta =_{def} h^{\mathcal{A}}(\llbracket e_1 \rrbracket^{\mathcal{A}} \eta, \dots, \llbracket e_n \rrbracket^{\mathcal{A}} \eta)$, for all $h \in DC^n \cup FS^n$.

□

Due to non-determinism, the evaluation of a term yields a cone rather than an element. However, this cone can still represent an element (in the ideal completion)

in the case that it is an ideal.

In the next proposition and lemma we prove some properties related to term evaluation:

Proposition 5.1.

Given a CRWL-algebra \mathcal{A} , for any $\mathbf{e} \in \text{Term}_\perp$ and any $\eta \in \text{Val}(\mathcal{A})$:

- a) $\llbracket \mathbf{e} \rrbracket^{\mathcal{A}} \eta \in \mathcal{C}(\mathcal{D}_{\mathcal{A}})$.
- b) $\llbracket \mathbf{e} \rrbracket^{\mathcal{A}} \eta \in \mathcal{I}(\mathcal{D}_{\mathcal{A}})$ if $\mathfrak{f}^{\mathcal{A}}$ is deterministic for every defined function symbol f occurring in \mathbf{e} .
- c) $\llbracket \mathbf{e} \rrbracket^{\mathcal{A}} \eta = \langle \mathbf{v} \rangle$ for some $\mathbf{v} \in \mathcal{D}_{\mathcal{A}}$, if $\mathbf{e} \in \text{CTerm}_\perp$. Moreover, $\mathbf{v} \in \text{Def}(\mathcal{D}_{\mathcal{A}})$ if $\mathbf{e} \in \text{CTerm}$ and $\eta \in \text{DefVal}(\mathcal{A})$.

PROOF.

- a) We reason by structural induction.
 - For $\mathbf{e} \equiv X \in \mathcal{V}$, $\llbracket \mathbf{e} \rrbracket^{\mathcal{A}} \eta = \langle \eta(X) \rangle$ is a cone.
 - For $\mathbf{e} \equiv \mathfrak{h}(\mathbf{e}_1, \dots, \mathbf{e}_n)$, we can assume that $A_i = \llbracket \mathbf{e}_i \rrbracket^{\mathcal{A}} \eta$ ($1 \leq i \leq n$) are cones by I.H. Then, $\llbracket \mathbf{e} \rrbracket^{\mathcal{A}} \eta = \mathfrak{h}^{\mathcal{A}}(A_1, \dots, A_n) = \bigcup \{ \mathfrak{h}^{\mathcal{A}}(\mathbf{u}_1, \dots, \mathbf{u}_n) / \mathbf{u}_i \in A_i \}$, a non-empty union, because $\perp \in A_i$ ($1 \leq i \leq n$). Each $\mathfrak{h}^{\mathcal{A}}(\mathbf{u}_1, \dots, \mathbf{u}_n)$ is a downward closed set including \perp , since it is a cone. Thus $\llbracket \mathbf{e} \rrbracket^{\mathcal{A}} \eta$ is also a cone.
- b) Similar reasoning as in a). Now, in the inductive step we can assume that A_1, \dots, A_n are ideals and $\mathfrak{h}^{\mathcal{A}}$ is deterministic. We check that $\llbracket \mathbf{e} \rrbracket^{\mathcal{A}} \eta$ is a directed set. For given $\mathbf{v}, \mathbf{v}' \in \llbracket \mathbf{e} \rrbracket^{\mathcal{A}} \eta$ we can find $\mathbf{u}_i, \mathbf{u}'_i \in A_i$ ($1 \leq i \leq n$) such that $\mathbf{v} \in \mathfrak{h}^{\mathcal{A}}(\mathbf{u}_1, \dots, \mathbf{u}_n)$, $\mathbf{v}' \in \mathfrak{h}^{\mathcal{A}}(\mathbf{u}'_1, \dots, \mathbf{u}'_n)$. Since each A_i is directed there are some $\mathbf{u}''_i \in A_i$ such that $\mathbf{u}_i, \mathbf{u}'_i \sqsubseteq \mathbf{u}''_i$ ($1 \leq i \leq n$). Monotonicity of $\mathfrak{h}^{\mathcal{A}}$ ensures that $\mathbf{v}, \mathbf{v}' \in \mathfrak{h}^{\mathcal{A}}(\mathbf{u}''_1, \dots, \mathbf{u}''_n)$. Since $\mathfrak{h}^{\mathcal{A}}$ is deterministic, $\mathfrak{h}^{\mathcal{A}}(\mathbf{u}''_1, \dots, \mathbf{u}''_n)$ is an ideal, and it must include some \mathbf{v}'' such that $\mathbf{v}, \mathbf{v}' \sqsubseteq \mathbf{v}''$.
- c) We reason by induction as in a), b). Now in the inductive step we can assume $A_i = \langle \mathbf{u}_i \rangle$ ($1 \leq i \leq n$), and even that \mathbf{u}_i are totally defined if η is. By monotonicity, $\mathfrak{c}^{\mathcal{A}}(A_1, \dots, A_n) = \mathfrak{c}^{\mathcal{A}}(\mathbf{u}_1, \dots, \mathbf{u}_n)$. Because of the requirements in Definition 5.2, this is of the form $\langle \mathbf{v} \rangle$, where \mathbf{v} can be chosen totally defined if $\mathbf{u}_1, \dots, \mathbf{u}_n$ are totally defined. □

Lemma 5.1. (Substitution Lemma)

Let η be a valuation over a CRWL-algebra \mathcal{A} . For any $\mathbf{e} \in \text{Term}_\perp$ and any $\theta \in \text{Csubst}_\perp$ we have: $\llbracket \mathbf{e}\theta \rrbracket^{\mathcal{A}} \eta = \llbracket \mathbf{e} \rrbracket^{\mathcal{A}} \eta_\theta$, where η_θ is the uniquely determined valuation that satisfies:

$$\langle \eta_\theta(X) \rangle = \llbracket X\theta \rrbracket^{\mathcal{A}} \eta \text{ for all } X \in \mathcal{V}$$

PROOF.

Note that η_θ is indeed uniquely determined since for all $X \in \mathcal{V}$ there is a unique $\mathbf{v} \in \mathcal{D}_{\mathcal{A}}$ such that $\llbracket X\theta \rrbracket^{\mathcal{A}} \eta = \langle \mathbf{v} \rangle$, due to proposition 5.1(c). The lemma is proved by structural induction over \mathbf{e} :

- For $e \equiv X \in \mathcal{V}$, we get $\llbracket e\theta \rrbracket^{\mathcal{A}\eta} = \llbracket X\theta \rrbracket^{\mathcal{A}\eta} = \langle \eta_\theta(X) \rangle = \llbracket e \rrbracket^{\mathcal{A}\eta_\theta}$
- For $e \equiv h(e_1, \dots, e_n)$, we have

$$\begin{aligned} \llbracket e\theta \rrbracket^{\mathcal{A}\eta} &= h^{\mathcal{A}}(\llbracket e_1\theta \rrbracket^{\mathcal{A}\eta}, \dots, \llbracket e_n\theta \rrbracket^{\mathcal{A}\eta}) \\ &= h^{\mathcal{A}}(\llbracket e_1 \rrbracket^{\mathcal{A}\eta_\theta}, \dots, \llbracket e_n \rrbracket^{\mathcal{A}\eta_\theta}) \quad \% \text{ By Induction Hypothesis} \\ &= \llbracket e \rrbracket^{\mathcal{A}\eta_\theta} \end{aligned}$$

□

Models

We are now prepared to introduce models. The main ideas are to interpret reduction statements in \mathcal{A} as inclusions between cones (i.e. approximations in Hoare's powerdomain $\mathcal{P}(\mathcal{D}_{\mathcal{A}})$) and to interpret joinability statements as asserting the existence of some common, totally defined approximation. Remember that totally defined elements correspond to finite and total elements in the ideal completion.

Definition 5.4. (Models)

Assume a program \mathcal{R} and a CRWL-algebra \mathcal{A} . We define:

- \mathcal{A} satisfies a reduction statement $a \rightarrow b$ under a valuation η (in symbols, $(\mathcal{A}, \eta) \models a \rightarrow b$) iff $\llbracket a \rrbracket^{\mathcal{A}\eta} \supseteq \llbracket b \rrbracket^{\mathcal{A}\eta}$.
- \mathcal{A} satisfies a joinability statement $a \bowtie b$ under a valuation η (in symbols, $(\mathcal{A}, \eta) \models a \bowtie b$) iff $\llbracket a \rrbracket^{\mathcal{A}\eta} \cap \llbracket b \rrbracket^{\mathcal{A}\eta} \cap \text{Def}(\mathcal{D}_{\mathcal{A}}) \neq \emptyset$.
- \mathcal{A} satisfies a rule $l \rightarrow r \Leftarrow C$ iff every valuation η such that $(\mathcal{A}, \eta) \models C$ verifies: $(\mathcal{A}, \eta) \models l \rightarrow r$.
- \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} .

□

CRWL-provability is sound w.r.t. models in the following sense:

Theorem 5.1. (Soundness)

For any program \mathcal{R} and any reduction or joinability statement φ :

$$\mathcal{R} \vdash_{\text{CRWL}} \varphi \Rightarrow (\mathcal{A}, \eta) \models \varphi, \text{ for all } \mathcal{A} \models \mathcal{R} \text{ and all } \eta \in \text{DefVal}(\mathcal{A}).$$

PROOF.

Let $\mathcal{A} \models \mathcal{R}$ and $\eta \in \text{DefVal}(\mathcal{A})$ be arbitrarily fixed. We prove that satisfaction in \mathcal{A} under η is preserved by all the inference rules of the BRC-calculus. The theorem follows then by a trivial induction on the length of BRC-proof. Rules **B**, **MN**, **RF** and **TR** are obviously sound.

For rule **R**, let us consider $(l \rightarrow r \Leftarrow C)\theta \in [\mathcal{R}]_{\perp}$, where $(l \rightarrow r \Leftarrow C) \in \mathcal{R}$ and $\theta \in \text{CSubst}_{\perp}$. Assume that $(\mathcal{A}, \eta) \models C\theta$. Then $(\mathcal{A}, \eta_\theta) \models C$, by Lemma 5.1. Since $\mathcal{A} \models \mathcal{R}$, we conclude that $(\mathcal{A}, \eta_\theta) \models l \rightarrow r$, and using again Lemma 5.1, we come to $(\mathcal{A}, \eta) \models l\theta \rightarrow r\theta$. Finally, for rule **J** let us assume $(\mathcal{A}, \eta) \models a \rightarrow t$, $(\mathcal{A}, \eta) \models b \rightarrow t$ for some $t \in \text{CTerm}$. By Definition 5.4, this means

$$\llbracket a \rrbracket^{\mathcal{A}\eta} \supseteq \llbracket t \rrbracket^{\mathcal{A}\eta} \subseteq \llbracket b \rrbracket^{\mathcal{A}\eta}$$

By Proposition 5.1 c), we know that $\llbracket t \rrbracket^{\mathcal{A}} \eta = \langle v \rangle$ for some $v \in \text{Def}(D_{\mathcal{A}})$. Again by Definition 5.4, we can conclude that $(\mathcal{A}, \eta) \models a \bowtie b$. \square

Canonic Term Models

Completeness of \vdash_{CRWL} can be proved with the help of canonic term models, that are closely related to CRWL-provability.

Definition 5.5. (Canonic Term Model, Term Algebras)

Given a program \mathcal{R} , the *canonic term model* $\mathcal{M}_{\mathcal{R}}$ is defined as follows:

- a) $D_{\mathcal{M}_{\mathcal{R}}}$ is the poset CTerm_{\perp} with approximation ordering \sqsubseteq , as defined in Section 3.
- b) $c^{\mathcal{M}_{\mathcal{R}}}(\mathbf{t}_1, \dots, \mathbf{t}_n) =_{\text{def}} \langle c(\mathbf{t}_1, \dots, \mathbf{t}_n) \rangle$ (principal ideal), for all $\mathbf{t}_i \in \text{CTerm}_{\perp}$.
- c) $f^{\mathcal{M}_{\mathcal{R}}}(\mathbf{t}_1, \dots, \mathbf{t}_n) =_{\text{def}} \{ t \in \text{CTerm}_{\perp} \mid \mathcal{R} \vdash_{\text{CRWL}} f(\mathbf{t}_1, \dots, \mathbf{t}_n) \rightarrow t \}$, for all $\mathbf{t}_i \in \text{CTerm}_{\perp}$.

Moreover, any CRWL-algebra \mathcal{A} with $D_{\mathcal{A}} = \text{CTerm}_{\perp}$ that satisfies item b) above, is called a *term algebra*. \square

$\mathcal{M}_{\mathcal{R}}$ is a well-defined CRWL-algebra. In fact, $f^{\mathcal{M}_{\mathcal{R}}}$ is monotonic (as required by Definitions 5.1, 5.2) as a consequence the Monotonicity Lemma 4.1. More precisely, by applying the Monotonicity Lemma with $\mathbf{e} \equiv f(X_1, \dots, X_n)$, $\theta = \{ X_1/t_1 \dots X_n/t_n \}$ and $\theta' = \{ X_1/t'_1 \dots X_n/t'_n \}$ we obtain:

$$f^{\mathcal{M}_{\mathcal{R}}}(\mathbf{t}_1, \dots, \mathbf{t}_n) \subseteq f^{\mathcal{M}_{\mathcal{R}}}(\mathbf{t}'_1, \dots, \mathbf{t}'_n)$$

under the assumption $\mathbf{t}_i \sqsubseteq \mathbf{t}'_i$ ($1 \leq i \leq n$). Thus, $f^{\mathcal{M}_{\mathcal{R}}} \in [D_{\mathcal{M}_{\mathcal{R}}}^n \rightarrow_n D_{\mathcal{M}_{\mathcal{R}}}]$, as requested.

For any $t \in \text{CTerm}_{\perp}$, $\theta \in \text{CSubst}_{\perp}$ it is easily checked that $\langle t\theta \rangle = \llbracket t \rrbracket^{\mathcal{M}_{\mathcal{R}}} \theta$. This fact will be used several times in the rest of the paper.

Next we show that satisfaction in $\mathcal{M}_{\mathcal{R}}$ can be characterized in terms of CRWL provability.

Lemma 5.2. (Characterization Lemma)

Let id be the identity valuation over $\mathcal{M}_{\mathcal{R}}$, defined by $\text{id}(X) = X$ for all $X \in \mathcal{V}$.

For any approximation or joinability statement φ , we have $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi \Leftrightarrow \mathcal{R} \vdash_{\text{CRWL}} \varphi$.

PROOF. -

Using the equivalence between the rewriting calculi BRC and GORC (cf. Proposition 4.1), we split the proof in two parts:

- a) $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi \Rightarrow \mathcal{R} \vdash_{\text{BRC}} \varphi$
- b) $\mathcal{R} \vdash_{\text{GORC}} \varphi \Rightarrow (\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi$

To prove a) we reason by induction on $\|\varphi\| \in \mathbb{N}$ defined as follows:

- $\|\mathbf{e} \rightarrow \perp\| =_{def} \mathbf{size}(\mathbf{e})$.
- $\|\mathbf{a} \bowtie \mathbf{b}\| =_{def} \mathbf{size}(\mathbf{a}) + \mathbf{size}(\mathbf{b})$

where $\mathbf{size}(\mathbf{e})$ is meant as the number of symbols (function, constructor and variable symbols) occurring in \mathbf{e} . Under the assumption that $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi$, there are five cases to consider:

- a.1 $\varphi \equiv \mathbf{e} \rightarrow \perp$, this is a base case. $\mathcal{R} \vdash_{\text{BRC}} \varphi$ holds because of rule **B**.
- a.2 $\varphi \equiv \mathbf{e} \rightarrow \mathbf{e}$, with $\mathbf{e} \in \mathcal{V}$. This is also a base case. $\mathcal{R} \vdash_{\text{BRC}} \varphi$ holds because of rule **RF**.
- a.3 $\varphi \equiv \mathbf{c}(\mathbf{e}_1, \dots, \mathbf{e}_n) \rightarrow \mathbf{c}(\mathbf{t}_1, \dots, \mathbf{t}_n)$. By construction of $\mathcal{M}_{\mathcal{R}}$, we have that $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi$ entails $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \mathbf{e}_i \rightarrow \mathbf{t}_i$ ($1 \leq i \leq n$). Then $\mathcal{R} \vdash_{\text{BRC}} \varphi$ follows by I.H. and rule **MN**.
- a.4 $\varphi \equiv \mathbf{f}(\mathbf{e}_1, \dots, \mathbf{e}_n) \rightarrow \mathbf{t}$, with $\mathbf{t} \neq \perp$. Since $\llbracket \mathbf{t} \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} = \langle \mathbf{t} \rangle$, $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi$ entails:

$$\mathbf{t} \in \mathbf{f}^{\mathcal{M}_{\mathcal{R}}}(\llbracket \mathbf{e}_1 \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}}, \dots, \llbracket \mathbf{e}_n \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}})$$

Hence, there are some $\mathbf{t}_i \in \llbracket \mathbf{e}_i \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}}$ ($1 \leq i \leq n$) such that $\mathbf{t} \in \mathbf{f}^{\mathcal{M}_{\mathcal{R}}}(\mathbf{t}_1 \dots \mathbf{t}_n)$. Therefore, we have:

$$(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \mathbf{e}_i \rightarrow \mathbf{t}_i \quad (1 \leq i \leq n) \quad \text{and} \quad \mathcal{R} \vdash_{\text{BRC}} \mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n) \rightarrow \mathbf{t},$$

by construction of $\mathcal{M}_{\mathcal{R}}$. By I.H. we can assume $\mathcal{R} \vdash_{\text{BRC}} \mathbf{e}_i \rightarrow \mathbf{t}_i$ ($1 \leq i \leq n$). Then, $\mathcal{R} \vdash_{\text{BRC}} \varphi$ follows by application of rules **MN** and **TR**.

- a.5 $\varphi \equiv \mathbf{a} \bowtie \mathbf{b}$. Since totally defined elements in $\mathcal{M}_{\mathcal{R}}$ are total C-terms, we get that $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi$ entails the existence of some $\mathbf{t} \in \text{CTerm}$ such that $\mathbf{t} \in \llbracket \mathbf{a} \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} \cap \llbracket \mathbf{b} \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}}$. Due to the fact that $\llbracket \mathbf{t} \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} = \langle \mathbf{t} \rangle$, we can conclude $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \mathbf{a} \rightarrow \mathbf{t}$, $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \mathbf{b} \rightarrow \mathbf{t}$. Then, $\mathcal{R} \vdash_{\text{BRC}} \varphi$ follows by I.H. and rule **J**.

To prove b) we reason by induction on the size of GORC-proofs, measured as the number of inference steps. There are five cases to consider corresponding to the five possibilities of the last step in a GORC-proof.

- b.1 $\mathcal{R} \vdash_{\text{GORC}} \varphi$ is proved by a single application of rule **B**. Then $\varphi \equiv \mathbf{e} \rightarrow \perp$ and $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi$ holds trivially.
- b.2 $\mathcal{R} \vdash_{\text{GORC}} \varphi$ is proved by a single application of rule **RR**. Then $\varphi \equiv \mathbf{X} \rightarrow \mathbf{X}$, with $\mathbf{X} \in \mathcal{V}$, and $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi$ is also trivial.
- b.3 $\mathcal{R} \vdash_{\text{GORC}} \varphi$ by a GORC-proof ending with a **DC**-step. In this case we know that $\varphi \equiv \mathbf{c}(\mathbf{e}_1, \dots, \mathbf{e}_n) \rightarrow \mathbf{c}(\mathbf{t}_1, \dots, \mathbf{t}_n)$. The case $n = 0$ is trivial. In the case $n > 0$, we can assume $\mathcal{R} \vdash_{\text{GORC}} \mathbf{e}_i \rightarrow \mathbf{t}_i$ ($1 \leq i \leq n$) with shorter GORC-proofs. By I.H. we get $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \mathbf{e}_i \rightarrow \mathbf{t}_i$ ($1 \leq i \leq n$). Then $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi$ follows by construction of $\mathcal{M}_{\mathcal{R}}$.

b.4 $\mathcal{R} \vdash_{\text{GORC}} \varphi$ by a GORC-proof ending with an **OR**-step. Then $\varphi \equiv f(\mathbf{e}_1, \dots, \mathbf{e}_n) \rightarrow t$ and we have shorter GORC-proofs for $\mathcal{R} \vdash_{\text{GORC}} \mathbf{e}_i \rightarrow t_i$ ($1 \leq i \leq n$), $\mathcal{R} \vdash_{\text{GORC}} C$ and $\mathcal{R} \vdash_{\text{GORC}} r \rightarrow t$, for some t_i, C, r such that $f(t_1, \dots, t_n) \rightarrow r \Leftarrow C \in [\mathcal{R}]_{\perp}$. From this we can conclude that $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \mathbf{e}_i \rightarrow t_i$ ($1 \leq i \leq n$), using I.H. Moreover, $\mathcal{R} \vdash_{\text{BRC}} f(t_1, \dots, t_n) \rightarrow t$ also follows easily, using Proposition 4.1 and the BRC-rules **R** and **TR**. By construction of $\mathcal{M}_{\mathcal{R}}$, we conclude that $t \in f^{\mathcal{M}_{\mathcal{R}}}(t_1, \dots, t_n)$. This together with $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \mathbf{e}_i \rightarrow t_i$ ($1 \leq i \leq n$) entails $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models f(\mathbf{e}_1, \dots, \mathbf{e}_n) \rightarrow t$ again by construction of $\mathcal{M}_{\mathcal{R}}$.

b.5 $\mathcal{R} \vdash_{\text{GORC}} \varphi$ by a GORC-proof ending with a **J**-step. Then $\varphi \equiv \mathbf{a} \bowtie \mathbf{b}$ and we have shorter GORC-proofs for $\mathcal{R} \vdash_{\text{GORC}} \mathbf{a} \rightarrow t$, and $\mathcal{R} \vdash_{\text{GORC}} \mathbf{b} \rightarrow t$, for some $t \in \text{CTerm}$. Using I.H. and $\llbracket t \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} = \langle t \rangle$, we conclude that $t \in \llbracket \mathbf{a} \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} \cap \llbracket \mathbf{b} \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}}$. Since t is totally defined in $\mathcal{M}_{\mathcal{R}}$ this entails $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi$. □

As a consequence of the Characterization Lemma, we get also:

Corollary 5.1.

For any partial C-substitution θ (which is also a valuation over $\mathcal{M}_{\mathcal{R}}$), any approximation or joinability statement φ , and any term \mathbf{e} , we have:

- a) $(\mathcal{M}_{\mathcal{R}}, \theta) \models \varphi \Leftrightarrow \mathcal{R} \vdash_{\text{CRWL}} \varphi \theta$.
- b) $\llbracket \mathbf{e} \rrbracket^{\mathcal{M}_{\mathcal{R}} \theta} = \{ t \in \text{CTerm}_{\perp} \mid \mathcal{R} \vdash_{\text{CRWL}} \mathbf{e} \theta \rightarrow t \}$

PROOF.

- a) By applying lemmas 5.1 and 5.2 we get

$$(\mathcal{M}_{\mathcal{R}}, \text{id}_{\theta}) \models \varphi \Leftrightarrow (\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi \theta \Leftrightarrow \mathcal{R} \vdash_{\text{CRWL}} \varphi \theta$$

where id_{θ} is determined as the unique valuation such that:

$$\langle \text{id}_{\theta}(X) \rangle = \llbracket X \theta \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} \text{ for all } X \in \mathcal{V}$$

Since $X \theta$ is a partial C-term, we have $\llbracket X \theta \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} = \langle X \theta \rangle$. Therefore, $\text{id}_{\theta} = \theta$, and the result is proved.

- b) We can reason as follows:

$$\begin{aligned} \llbracket \mathbf{e} \rrbracket^{\mathcal{M}_{\mathcal{R}} \theta} &= \llbracket \mathbf{e} \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}_{\theta}} && \% \theta = \text{id}_{\theta} \\ &= \llbracket \mathbf{e} \theta \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} && \% \text{ Lemma 5.1} \\ &= \{ t \in \text{CTerm}_{\perp} \mid t \in \llbracket \mathbf{e} \theta \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} \} \\ &= \{ t \in \text{CTerm}_{\perp} \mid \langle t \rangle = \llbracket t \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} \subseteq \llbracket \mathbf{e} \theta \rrbracket^{\mathcal{M}_{\mathcal{R}} \text{id}} \} \\ &= \{ t \in \text{CTerm}_{\perp} \mid (\mathcal{M}_{\mathcal{R}}, \text{id}) \models \mathbf{e} \theta \rightarrow t \} \\ &= \{ t \in \text{CTerm}_{\perp} \mid \mathcal{R} \vdash_{\text{CRWL}} \mathbf{e} \theta \rightarrow t \} && \% \text{ Lemma 5.2} \end{aligned}$$

□

Properties of Canonic Term Models

Now we can discuss why canonic term models are important. First, we prove a main result relating provability and models in CRWL:

Theorem 5.2. (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) $\mathcal{R} \vdash_{\text{CRWL}} \varphi$.
- b) $(\mathcal{A}, \eta) \models \varphi$ for every $\mathcal{A} \models \mathcal{R}$, and every $\eta \in \text{DefVal}(\mathcal{A})$.
- c) $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models \varphi$, where id is the identity valuation.

PROOF.

In order to prove $\mathcal{M}_{\mathcal{R}} \models \mathcal{R}$, we consider a rule $l \rightarrow r \Leftarrow C \in \mathcal{R}$ and a valuation over $\mathcal{M}_{\mathcal{R}}$, which is the same as a substitution $\theta \in \text{Csubst}_{\perp}$. Assume that $(\mathcal{M}_{\mathcal{R}}, \theta) \models C$. By Corollary 5.1.a), this amounts to $\mathcal{R} \vdash_{\text{CRWL}} C\theta$. It follows that $\mathcal{R} \vdash_{\text{CRWL}} (l \rightarrow r)\theta$. We conclude that $\{ t \in \text{CTerm}_{\perp} \mid \mathcal{R} \vdash_{\text{CRWL}} r\theta \rightarrow t \} \subseteq \{ t \in \text{CTerm}_{\perp} \mid \mathcal{R} \vdash_{\text{CRWL}} l\theta \rightarrow t \}$ using the BRC-rule **TR**. By Corollary 5.1.b), this means $\llbracket r \rrbracket^{\mathcal{M}_{\mathcal{R}}\theta} \subseteq \llbracket l \rrbracket^{\mathcal{M}_{\mathcal{R}}\theta}$.

We come now to the equivalences:

- a) entails b) by the Soundness Theorem 5.1. b) entails c) because $\mathcal{M}_{\mathcal{R}}$ is a model of \mathcal{R} and id is a totally defined valuation. Finally, c) entails a) by the Characterization Lemma 5.2.

□

Note that the completeness of \vdash_{CRWL} also follows from Theorem 5.2. According to this result, $\mathcal{M}_{\mathcal{R}}$ can be regarded as the intended (canonical) model of program \mathcal{R} . In particular, a given $f \in \text{FS}^n$ will denote a deterministic function iff $f^{\mathcal{M}_{\mathcal{R}}}(t_1, \dots, t_n)$ is an ideal for all $t_i \in \text{CTerm}_{\perp}$. This property is undecidable in general, but some decidable sufficient conditions are known which work quite well in practice; see e.g. the sufficient non-ambiguity conditions in [18].

There is a clear analogy between $\mathcal{M}_{\mathcal{R}}$ and so-called \mathcal{C} -semantics [14] for Horn clause programs. In fact, \mathcal{C} -semantics corresponds to the least Herbrand model over the Herbrand Universe of *open C-terms* (i.e., terms built from constructors and variables). Moreover, Horn clause logic programs correspond to CRWL-programs whose defining rules are of the form

$$p(\bar{t}) \rightarrow \text{true} \Leftarrow C$$

where each joinability condition in C is of the form $q(\bar{s}) \bowtie \text{true}$. For such programs, it is easily checked that $\mathcal{M}_{\mathcal{R}}$ indeed corresponds to the \mathcal{C} -semantics. By a construction similar to that of $\mathcal{M}_{\mathcal{R}}$, using the poset of *ground* partial C -terms as carrier, we could obtain also an analogon of the least Herbrand model semantics for Horn clause logic programs. However, $\mathcal{M}_{\mathcal{R}}$ bears more interesting information due to Theorem 5.2.

In relation to functional programming, there is an intuitive analogy between $\mathcal{M}_{\mathcal{R}}$ and the denotational semantics of \mathcal{R} . We have not developed any formal compari-

son. It is known, however, that the analogy breaks down for the case of higher-order functional programs. It is also known that a natural generalization of the term model $\mathcal{M}_{\mathcal{R}}$ to a higher-order setting can provide nice soundness and completeness results for rewriting and narrowing calculi; see [18, 17] for the case of deterministic higher-order functions, and [20] for a generalization including non-determinism.

As the last result in this section, we present a categorical characterization of $\mathcal{M}_{\mathcal{R}}$ as the free model of \mathcal{R} , generated by the set of variables \mathcal{V} . We will use only very elementary notions from category theory; see e.g. [45]. First, we need a suitable notion of homomorphism. There exist several known possibilities for defining homomorphisms between algebras involving non-deterministic operations; see e.g. Hussmann's discussion in [28], Chapter 3. Our definition follows the idea of *loose element-valued homomorphisms*, in Hussmann's terminology.

Definition 5.6. (Homomorphisms)

Let \mathcal{A}, \mathcal{B} two given CRWL-algebras. A CRWL-homomorphism $h: \mathcal{A} \rightarrow \mathcal{B}$ is any deterministic function $h \in [D_{\mathcal{A}} \rightarrow_d D_{\mathcal{B}}]$ which satisfies the following conditions:

- H1** h is element-valued: for all $u \in D_{\mathcal{A}}$ there is $v \in D_{\mathcal{B}}$ such that $h(u) = \langle v \rangle$.
- H2** h is strict: $h(\perp_{\mathcal{A}}) = \langle \perp_{\mathcal{B}} \rangle$.
- H3** h preserves constructors: for all $c \in DC^n$, $u_i \in D_{\mathcal{A}}$:
 $h(c^{\mathcal{A}}(u_1, \dots, u_n)) = c^{\mathcal{B}}(h(u_1), \dots, h(u_n))$.
- H4** h loosely preserves defined functions: for all $f \in FS^n$, $u_i \in D_{\mathcal{A}}$:
 $h(f^{\mathcal{A}}(u_1, \dots, u_n)) \subseteq f^{\mathcal{B}}(h(u_1), \dots, h(u_n))$.

□

CRWL-algebras as objects with CRWL-homomorphism as arrows form a category CRWL-ALG. Moreover, for every CRWL-program \mathcal{R} , we can consider the full subcategory CRWL-ALG $_{\mathcal{R}}$ whose objects are all the possible models of \mathcal{R} . Now, we can prove:

Theorem 5.3. (Freeness of $\mathcal{M}_{\mathcal{R}}$)

For any program \mathcal{R} , the canonic term model $\mathcal{M}_{\mathcal{R}}$ is freely generated by \mathcal{V} in the category CRWL-ALG $_{\mathcal{R}}$; that is, given any $\mathcal{A} \models \mathcal{R}$ and any valuation $\eta \in \text{DefVal}(\mathcal{A})$, there is a unique homomorphism $h: \mathcal{M}_{\mathcal{R}} \rightarrow \mathcal{A}$ extending η in the sense that $h(X) = \langle \eta(X) \rangle$ for all $X \in \mathcal{V}$.

PROOF.

$$\begin{array}{ccc}
 \mathcal{M}_{\mathcal{R}} & \xrightarrow{h} & \mathcal{A} \\
 \uparrow & \nearrow \eta & \\
 \mathcal{V} & &
 \end{array}$$

Existence: Let us define $h(t) =_{def} \llbracket t \rrbracket^{\mathcal{A}} \eta$ for all $t \in \text{CTerm}_{\perp}$. This mapping belongs to $[\text{CTerm}_{\perp} \rightarrow_n D_{\mathcal{A}}]$, since the required monotonicity property

$$t \sqsubseteq t' \Rightarrow \llbracket t \rrbracket^{\mathcal{A}} \eta \subseteq \llbracket t' \rrbracket^{\mathcal{A}} \eta$$

follows from $\mathcal{A} \models \mathcal{R}$ and the Soundness Theorem 5.1, taking into account that $t \sqsubseteq t'$ entails $\mathcal{R} \vdash_{\text{CRWL}} t \rightarrow t'$. The determinism of h follows from condition **H1** in Definition 5.6, which is itself a straightforward consequence of Proposition 5.1(c). Condition **H2** is trivial. Since $c^{\mathcal{M}_{\mathcal{R}}}(t_1, \dots, t_n) = \langle c(t_1, \dots, t_n) \rangle$, and h is monotonic, condition **H3** reduces to $h(c(t_1, \dots, t_n)) = c^{\mathcal{A}}(h(t_1), \dots, h(t_n))$ which is clear by definition of h . Lastly, by taking $f^{\mathcal{M}_{\mathcal{R}}}$'s definition into account (see Definition 5.5), condition **H4** reduces to prove the inclusion:

$$\bigcup \{ \llbracket t \rrbracket^{\mathcal{A}} \eta \mid t \in \text{CTerm}_{\perp}, \mathcal{R} \vdash_{\text{CRWL}} f(t_1, \dots, t_n) \rightarrow t \} \subseteq f^{\mathcal{A}}(\llbracket t_1 \rrbracket^{\mathcal{A}} \eta, \dots, \llbracket t_n \rrbracket^{\mathcal{A}} \eta)$$

Indeed, this holds because $\mathcal{A} \models \mathcal{R}$ and the Soundness Theorem 5.1 entail that $(\mathcal{A}, \eta) \models f(t_1, \dots, t_n) \rightarrow t$ for all $t_1, \dots, t_n, t \in \text{CTerm}_{\perp}$ such that $\mathcal{R} \vdash_{\text{CRWL}} f(t_1, \dots, t_n) \rightarrow t$.

Uniqueness: Assume any homomorphism $h: \mathcal{M}_{\mathcal{R}} \rightarrow \mathcal{A}$ such that $h(X) = \langle \eta(X) \rangle$ for all $X \in \mathcal{V}$. We have to prove that $h(t) = \llbracket t \rrbracket^{\mathcal{A}} \eta$ for all $t \in \text{CTerm}_{\perp}$. This follows trivially by structural induction on t , using homomorphism conditions **H2**, **H3**. □

The intuitive meaning of Theorem 5.3 is that $\mathcal{M}_{\mathcal{R}}$ behaves as the least term algebra that is a model of \mathcal{R} . An alternative characterization of $\mathcal{M}_{\mathcal{R}}$ as the least fixpoint of a continuous transformation which maps term algebras to term algebras is also possible; see [39].

6. A LAZY NARROWING CALCULUS

In this section we set the basis for using CRWL as a declarative programming language. To this purpose, we introduce admissible goals and solutions for programs, and we present a *Constructor-based Lazy Narrowing Calculus* (shortly, CLNC) for goal solving.

Let \mathcal{R} be any program. Goals for \mathcal{R} are certain finite conjunctions of CRWL-statements, and solutions are C-substitutions such that the goal affected by the substitution becomes CRWL-provable. The precise definition of *admissible goal* includes a number of technical conditions which are needed to achieve the effect of lazy evaluation with sharing during goal solving. In particular, the effect of sharing will be emulated by means of approximation statements of the form $e \rightarrow X$ within goals. A variable X will be allowed to occur at most one time at the right-hand side of such a statement, and it will be used to share e 's value with the rest of the goal, but avoiding both the eager replacement of X by e and the eager evaluation of e to a C-term. In the sequel, the symbol \square used in the syntax of goals must be interpreted as conjunction.

Definition 6.1. (Admissible goals)

An admissible goal for a given program \mathcal{R} must have the form $G \equiv \exists \overline{U}. S \square P \square E$, where:

- $\text{evar}(G) \equiv \overline{U}$ is the set of so-called existential variables of the goal G .

- $S \equiv X_1 = s_1, \dots, X_n = s_n$ is a set of equations, called solved part. Each s_i must be a total C-term, and each X_i must occur exactly once in the whole goal. (Intuition: Each s_i is a computed answer for X_i .)
- $P \equiv e_1 \rightarrow t_1, \dots, e_k \rightarrow t_k$ is a multiset of approximation statements, with $t_i \in \text{CTerm}$. $\text{pvar}(P) \stackrel{\text{def}}{=} \text{var}(t_1) \cup \dots \cup \text{var}(t_k)$ 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 $X \in \text{var}(e_i)$ and $Y \in \text{var}(t_i)$. (Intuition: $e_i \rightarrow t_i$ demands narrowing e_i to match t_i . This may produce bindings for variables in e_i .)
- $E \equiv a_1 \bowtie b_1, \dots, a_m \bowtie b_m$ is a multiset of joinability statements. $\text{dvar}(E) \stackrel{\text{def}}{=} \{ X \in \mathcal{V} / X \equiv a_i \text{ or } X \equiv b_i, \text{ for some } 1 \leq i \leq m \}$ is called the set of demanded variables of the goal G . (Intuition: Due to the semantics of joinability, goal solving must compute totally defined values for demanded variables.)

Additionally, any admissible goal must fulfil the following conditions:

- LIN** The tuple (t_1, \dots, t_k) must be linear. (Intuition: Each produced variable is produced only once.)
- EX** 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.)
- CYC** 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.)
- SOL** The solved part contains no produced variables. (Intuition: The solved part includes no reference to intermediate results.)

□

Properties similar to **LIN**, **EX** and **CYC** have been used previously in the so-called outermost strategy for the functional logic language K-LEAF [15] (based on flattening plus SLD-resolution), in the constrained lazy narrowing calculus from [34] and in a call-by-need strategy for higher-order lazy narrowing [46]. In comparison to the present approach, [15, 34] allow for less general programs⁷, while the higher-order language in [46] lacks a model-theoretic semantics and uses more restricted conditions of the form $l \rightarrow r$, where r is a ground normal form.

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 . To accept any admissible goal as initial goal seems not very natural, but it would cause no major problem, except minor technical changes in some of the results below. The notion of solution for any admissible goal is defined as follows:

Definition 6.2. (Solutions)

Let $G \equiv \exists \bar{U}. S \square P \square E$ be an admissible goal, and θ a partial C-substitution.

- θ is allowable for G iff $X\theta$ is a total C-term for every $X \notin \text{pvar}(P)$.

⁷More precisely, this is true only for the sublanguage of [34] which omits the use of *disequality constraints*.

- θ is a solution for G iff θ is allowable for G , $X_i\theta \equiv s_i$ for all $X_i = s_i \in S$, and $(P \square E)\theta$ has a ‘witness’ \mathcal{M} . A witness is defined as a multiset containing a GORC-proof (see Definition 4.2) for each condition $e\theta \rightarrow t\theta \in P\theta$ and $a\theta \bowtie b\theta \in E\theta$.
- We write $Sol(G)$ for the set of all solutions for G .

□

Our definition of solution considers *partial* C-substitutions, because produced variables (which are not present in initial goals, are existential and can eventually disappear during the computation) may need to be given only partial values, since they serve to express approximations. Note, however, that solutions of both initial and final goals (where only the solved part S will be present) are always *total* C-substitutions.

Due to the Adequateness Theorem 5.2, it is immediate to give a model-theoretic characterization of solutions, equivalent to the proof-theoretic definition. It is enough for our purposes to do this for initial goals.

Lemma 6.1.

Let \mathcal{R} be a program, G an initial goal, θ a C-substitution. The following statements are equivalent:

- $\theta \in Sol(G)$
- $\mathcal{R} \vdash_{\text{CRWL}} G\theta$
- $(\mathcal{M}_{\mathcal{R}}, \text{id}) \models G\theta$
- $(\mathcal{A}, \eta) \models G\theta$, for all $\mathcal{A} \models \mathcal{R}, \eta \in \text{DefVal}(\mathcal{A})$

□

We present now a *Constructor-based Lazy Narrowing Calculus* (shortly, CLNC) for solving initial goals, obtaining *solutions* in the sense of Definition 6.2. The calculus CLNC consists of a set of *transformation rules* for goals. Each transformation rule takes the form $G \# G'$, specifying one of the possible ways of performing one step of goal solving. Derivations are sequences of $\#$ -steps. For writing failure rules we use **FAIL**, that represents an irreducible inconsistent goal. We recall that in a goal $\exists \bar{U}. S \square P \square E$, 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 S, P or E . In addition, to the purpose of applying the rules, we see conditions $a \bowtie b$ as symmetric. The notation $\text{svar}(\mathbf{e})$, used in some transformation rules, stands for the set of all variables X occurring in \mathbf{e} at some position whose ancestor positions are all occupied by constructors.

The CLNC-calculus

Rules for \bowtie

- DC1 Decomposition:**
 $\exists \bar{U}. S \square P \square c(\bar{a}) \bowtie c(\bar{b}), E \vdash \exists \bar{U}. S \square P \square \dots, a_i \bowtie b_i, \dots, E.$
- ID Identity:**
 $\exists \bar{U}. S \square P \square X \bowtie X, E \vdash \exists \bar{U}. S \square P \square E$
 if $X \notin \text{pvar}(P).$
- BD Binding:**
 $\exists \bar{U}. S \square P \square X \bowtie s, E \vdash \exists \bar{U}. X = s, (S \square P \square E)\sigma$
 if $s \in \text{CTerm}, \text{var}(s) \cap \text{pvar}(P) = \emptyset, X \notin \text{var}(s), X \notin \text{pvar}(P),$ where $\sigma = \{X/s\}.$
- IM Imitation:**
 $\exists \bar{U}. S \square P \square X \bowtie c(\bar{e}), E \vdash \exists \bar{X}, \bar{U}. X = c(\bar{X}), (S \square P \square \dots, X_i \bowtie e_i, \dots, E)\sigma$
 if $c(\bar{e}) \notin \text{CTerm}$ or $\text{var}(c(\bar{e})) \cap \text{pvar}(P) \neq \emptyset,$ and $X \notin \text{pvar}(P), X \notin \text{svar}(c(\bar{e})),$
 where $\sigma = \{X/c(\bar{X})\}, \bar{X}$ are new variables.
- NR1 Narrowing:**
 $\exists \bar{U}. S \square P \square f(\bar{e}) \bowtie a, E \vdash \exists \bar{X}, \bar{U}. S \square \dots, e_i \rightarrow t_i, \dots, P \square C, r \bowtie a, E$
 where $R : f(\bar{t}) \rightarrow r \Leftarrow C$ is a variant of a rule in $\mathcal{R},$ with $\bar{X} = \text{var}(R)$ new variables.

Rules for \rightarrow

- DC2 Decomposition:**
 $\exists \bar{U}. S \square c(\bar{e}) \rightarrow c(\bar{t}), P \square E \vdash \exists \bar{U}. S \square \dots, e_i \rightarrow t_i, \dots, P \square E.$
- OB Output Binding:**
- OB1** $\exists \bar{U}. S \square X \rightarrow t, P \square E \vdash \exists \bar{U}. X = t, (S \square P \square E)\sigma$
 if $t \notin \mathcal{V}, X \notin \text{pvar}(P),$ where $\sigma = \{X/t\}.$
- OB2** $\exists X, \bar{U}. S \square X \rightarrow t, P \square E \vdash \exists \bar{U}. S \square (P \square E)\sigma$
 if $t \notin \mathcal{V}, X \in \text{pvar}(P),$ where $\sigma = \{X/t\}.$
- IB Input Binding:**
 $\exists X, \bar{U}. S \square t \rightarrow X, P \square E \vdash \exists \bar{U}. S \square (P \square E)\sigma$
 if $t \in \text{CTerm},$ where $\sigma = \{X/t\}.$
- IIM Input Imitation:**
 $\exists X, \bar{U}. S \square c(\bar{e}) \rightarrow X, P \square E \vdash \exists \bar{X}, \bar{U}. S \square (\dots, e_i \rightarrow X_i, \dots, P \square E)\sigma$
 if $c(\bar{e}) \notin \text{CTerm}, X \in \text{dvar}(E),$ where $\sigma = \{X/c(\bar{X})\}, \bar{X}$ new variables.
- EL Elimination:**
 $\exists X, \bar{U}. S \square e \rightarrow X, P \square E \vdash \exists \bar{U}. S \square P \square E$
 if $X \notin \text{var}(P \square E).$
- NR2 Narrowing:**
 $\exists \bar{U}. S \square f(\bar{e}) \rightarrow t, P \square E \vdash \exists \bar{X}, \bar{U}. S \square \dots, e_i \rightarrow t_i, \dots, r \rightarrow t, P \square C, E$
 if $t \notin \mathcal{V}$ or $t \in \text{dvar}(E),$ where $R : f(\bar{t}) \rightarrow r \Leftarrow C$ is a variant of a rule in $\mathcal{R},$
 with $\bar{X} = \text{var}(R)$ new variables.

Failure rules

- CF1 Conflict:** $\exists \bar{U}. S \square P \square c(\bar{a}) \bowtie d(\bar{b}), E \vdash \text{FAIL}$ if $c \not\equiv d.$
- CY Cycle:** $\exists \bar{U}. S \square P \square X \bowtie a, E \vdash \text{FAIL}$ if $X \not\equiv a, X \in \text{svar}(a).$
- CF2 Conflict:** $\exists \bar{U}. S \square c(\bar{a}) \rightarrow d(\bar{t}), P \square E \vdash \text{FAIL}$ if $c \not\equiv d.$

The following remarks attempt to clarify some relevant aspects of the CLNC-calculus.

- In all transformation rules involving a substitution σ (namely **BD**, **IM**, **OB**, **IB**, **IIM**), σ replaces a variable by a C-term. This means, in particular, that for an approximation statement $f(\bar{e}) \rightarrow X$, in no case the substitution $\{X/f(\bar{e})\}$ is applied. Actually, to perform the *eager* replacement $\{X/f(\bar{e})\}$ would be unsound because of our option of call-time choice semantics for non-deterministic functions. As explained in Section 1, one possible solution to this problem is to use *term graph narrowing* [8] to achieve *sharing*. In our CLNC-calculus, the effect of sharing is achieved in a different manner, avoiding the technical overhead of term graph narrowing. More precisely, in presence of $f(\bar{e}) \rightarrow X$, the following possibilities are considered:
 - a) Transformation **EL** deletes the approximation condition if 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 $f(\bar{e})$ is not needed and is indeed not performed. Hence, the rule **EL** is crucial for respecting the non-strict semantics of functions.
 - b) Transformation **NR2** uses a program rule for reducing $f(\bar{e})$, but only if X is detected as a *demand* variable, which in particular implies that X 's value in a solution cannot be \perp , and therefore requires the evaluation of $f(\bar{e})$. After one or more applications of **NR2** it will be the case (if the computation is going to succeed) that **IB** or **IIM** are applicable, thus propagating (partially, in the case of **IIM**) the obtained value to all the occurrences of X . As a result, *sharing* is achieved, and computations are lazy.
 - c) If neither **EL** nor **NR2** are applicable, nothing can be done with the approximation $f(\bar{e}) \rightarrow X$ but waiting until one of them becomes applicable. This will eventually happen, as our completeness results show.
- The absence of cycles of produced variables (property **CYC** of admissible goals) implies that no occur check is needed in **OB**, **IB**, **IIM**.
- *Eager variable elimination* can greatly help to eliminate redundant narrowing derivations, but unfortunately this transformation is proved to be complete only in some cases; see e.g. [37, 46]. In our setting, eager variable elimination can be unsound, as discussed above. However, the rules **BD**, **OB**, **IB** correspond to safe cases for eager variable elimination (via binding): these transformations are sound, and they do not compromise the completeness of CLNC. Note that special care is taken with produced variables. For instance, the goal

$$\exists N. \Box X \rightarrow s(N) \Box X \bowtie N$$

is admissible, but if **BD** would be applied (which is not allowed in CLNC, since N is a produced variable) we would obtain $\exists N. X = N \Box N \rightarrow s(N) \Box$, 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** include a *don't know* choice of the rule **R** of the program \mathcal{R} to be used. All the other transformation rules are completely deterministic (modulo the symmetry of \bowtie) and, what is more important, if several transformation rules are applicable to a given goal, a *don't care* choice among them can be done, as a consequence of the Progress Lemma 7.3 below. This kind of *strong completeness* does not hold in general for other lazy narrowing calculi, as shown in [37].

As an additional consequence of Lemma 7.3, a goal is \vdash -irreducible iff it is *FAIL* or takes the form $\exists \overline{U}. S \square \square$ (we call these goals *solved forms*). It is easy to see that solved forms are satisfiable. Each solved form $\exists \overline{U}. S \square \square$, with $S \equiv X_1 = t_1, \dots, X_n = t_n$, defines an associated *answer substitution* $\sigma_S = \{X_1/t_1, \dots, X_n/t_n\}$, which is idempotent. Notice that $\sigma_S \in \text{Sol}(\exists \overline{U}. S \square \square)$.

We close this section with an example of goal solving in the CLNC-calculus. The following CLNC-derivation computes the solution $\{X_s/[B], Y_s/[A]\}$ for the initial goal: $G_0 \equiv \square \square \text{merge}(X_s, Y_s) \bowtie [A, B]$ with respect to the program for the non-deterministic function `merge`, shown in Section 1. The notation \vdash^i will indicate i consecutive \vdash -steps.

$$\begin{aligned}
& \square \square \text{merge}(X_s, Y_s) \bowtie [A, B] \vdash \text{NR1} \\
& \exists X', Xs', Y', Ys'. \\
& \quad \square Xs \rightarrow [X' \mid Xs'], \quad Ys \rightarrow [Y' \mid Ys'] \quad \square [Y' \mid \text{merge}([X' \mid Xs'], Ys')] \bowtie [A, B] \vdash \text{OB1}^2 \\
& \exists X', Xs', Y', Ys'. Xs = [X' \mid Xs'], \quad Ys = [Y' \mid Ys'] \\
& \quad \square \square [Y' \mid \text{merge}([X' \mid Xs'], Ys')] \bowtie [A, B] \vdash \text{DC1} \\
& \exists X', Xs', Y', Ys'. Xs = [X' \mid Xs'], \quad Ys = [Y' \mid Ys'] \\
& \quad \square \square [Y' \bowtie A, \text{merge}([X' \mid Xs'], Ys')] \bowtie [B] \vdash \text{BD} \\
& \exists X', Xs', Y', Ys'. Y' = A, \quad Xs = [X' \mid Xs'], \quad Ys = [A \mid Ys'] \\
& \quad \square \square \text{merge}([X' \mid Xs'], Ys') \bowtie [B] \vdash \text{NR1} \\
& \exists X'', Xs'', X', Xs', Y', Ys'. Y' = A, \quad Xs = [X' \mid Xs'], \quad Ys = [A \mid Ys'] \\
& \quad \square [X' \mid Xs'] \rightarrow [X'' \mid Xs''], Ys' \rightarrow [] \quad \square [X'' \mid Xs''] \bowtie [B] \vdash \text{DC2, IB, OB1} \\
& \exists X', Xs', Y', Ys'. Ys' = [], \quad Y' = A, \quad Xs = [X' \mid Xs'], \quad Ys = [A] \\
& \quad \square \square [X' \mid Xs'] \bowtie [B] \vdash \text{DC1} \\
& \exists X', Xs', Y', Ys'. Ys' = [], \quad Y' = A, \quad Xs = [X' \mid Xs'], \quad Ys = [A] \\
& \quad \square \square [X' \bowtie B, Xs' \bowtie []] \vdash \text{BD}^2 \\
& \exists X', Xs', Y', Ys'. X' = B, \quad Xs' = [], \quad Ys' = [], \quad Y' = A, \quad Xs = [B], \quad Ys = [A] \quad \square \quad \square
\end{aligned}$$

7. SOUNDNESS AND COMPLETENESS

In this section we establish the soundness and completeness of CLNC w.r.t. the declarative semantics of CRWL. We first collect in the following lemma some sim-

ple facts about GORC-provable statements involving C-terms, which will be used several times, possibly without mentioning them explicitly. The proof is straightforward by induction over the structure of C-terms (for the *if* parts) or over the structure of GORC-proofs (for the *only if* parts).

Lemma 7.1.

For any partial $\mathbf{t}, \mathbf{s} \in \mathbf{CTerm}_\perp$, we have:

- a) $\mathbf{t} \rightarrow \mathbf{s}$ is GORC-provable iff $\mathbf{t} \sqsupseteq \mathbf{s}$ (see Section 3 for the definition of the approximation ordering \sqsupseteq over \mathbf{CTerm}_\perp). Furthermore, if $\mathbf{s} \in \mathbf{CTerm}$ then $\mathbf{t} \sqsupseteq \mathbf{s}$ can be replaced by $\mathbf{t} \equiv \mathbf{s}$.
- b) $\mathbf{t} \bowtie \mathbf{s}$ is GORC-provable iff $\mathbf{t}, \mathbf{s} \in \mathbf{CTerm}$ and $\mathbf{t} \equiv \mathbf{s}$.

□

The next result proves correctness of a single CLNC-step. It says that \vdash -steps preserve admissibility of goals, fail only in case of unsatisfiable goals and do not introduce new solutions. In the latter case, some care must be taken with existential variables.

Lemma 7.2. (Correctness lemma)

Invariance.- If $G \vdash G'$ and G is admissible, then G' is admissible.

Correctness₁.- If $G \vdash \text{FAIL}$ then $\text{Sol}(G) = \emptyset$.

Correctness₂.- If $G \vdash G'$ and $\theta' \in \text{Sol}(G')$ then there exists $\theta \in \text{Sol}(G)$ with $\theta = \theta'[\mathcal{V} \setminus (\text{evar}(G) \cup \text{evar}(G'))]$.

PROOF. Within this proof we assume, for each CLNC-rule, that G and G' are exactly as they appear in the presentation of the CLNC-calculus.

Invariance.-For each CLNC-rule we give succinct explanations justifying the preservation of the admissibility conditions given in Definition 6.1.

DC1, ID Trivial.

BD

LIN: Since X is not produced, σ does not modify the right-hand sides of the approximation statements in P .

EX: For the same reason, $\text{pvar}(P)$ is not modified, neither is \overline{U} .

CYC: Since $\text{pvar}(P)$ is not modified, and $\text{var}(\mathbf{s})$ does not intersect $\text{pvar}(P)$, no produced variables are introduced in the new left-hand sides of conditions in P . Hence, no cycle of produced variables is created.

SOL: Neither X nor \mathbf{s} contain produced variables.

IM

LIN: Since X is not produced, σ does not modify the right-hand sides of the approximation statements in P .

EX: For the same reason, $\text{pvar}(P)$ is not modified, while \overline{U} is enlarged to $\overline{U}, \overline{X}$.

CYC: Since $\text{pvar}(P)$ is not modified, and \overline{X} are new, no produced variables are introduced in the new left-hand sides of conditions in P . Hence, no cycle of produced variables is created.

SOL: Neither X nor $c(\bar{X})$ contain produced variables.

NR1

LIN: \bar{t} is a linear tuple of C-terms with new variables.

EX: All the new variables \bar{X} are existentially quantified.

CYC: Variables in each t_i are new (hence not appearing in any left-hand side of approximation conditions), so no cycle of produced variables can be created.

SOL: S is not changed and variables in t_i are new.

DC2

LIN: Clear, $c(\bar{t})$ and \bar{t} share the same linearity properties.

EX, SOL: Trivial.

CYC: Due to the decomposition, the new production relation is a subset of the old one.

OB1

LIN: Since X is not produced, σ does not modify the right-hand sides of the approximation statements in P .

EX: For the same reason $\text{pvar}(P)$ decreases (since variables in t are not produced anymore, due to **LIN**), while \bar{U} is not modified.

CYC: Since X is not produced, only approximation statements $l \rightarrow s$ with $X \in \text{var}(l)$ are affected by σ . In these cases, for each $Y \in \text{var}(t)$, $Z \in \text{var}(s)$, the relation $Y \gg_P Z$ is created. But $Y \gg_P Z$ cannot take part of a cycle of variables, because linearity ensures that Y does not appear in any right-hand side of an approximation statement in the new goal.

SOL: $\text{pvar}(P)$ decreases, X is not produced, variables in t are not produced anymore, and σ can only introduce the variables in t as new variables in S .

OB2

LIN: Let $l \rightarrow r$ be the unique approximation statement in P such that $X \in \text{var}(r)$ (X is produced, G verifies **LIN**). Note that σ replaces X by t in r (and no other right-hand side is changed), but $X \rightarrow t$ is deleted, so **LIN** is preserved.

EX: $\text{pvar}(P') = \text{pvar}(P) \setminus \{X\} \subseteq \text{evar}(G') = \text{evar}(G) \setminus \{X\}$.

CYC: The production relation is modified in the following ways:

- (i) If $l \rightarrow r$ is the unique approximation statement in P such that $X \in \text{var}(r)$, then in G' we have $Y \gg_P Z$ for each $Y \in \text{var}(l)$, $Z \in \text{var}(t)$. But in G we had $Y \gg_P X$, $X \gg_P Z$, and therefore \gg_P^* is not enlarged.
- (ii) If $l' \rightarrow r'$ is another approximation statement in P such that $X \in \text{var}(l')$, then in G' we have $Z \gg_P V$ for each $Z \in \text{var}(t)$, $V \in \text{var}(r')$. Any cycle in G' going through $Z \gg_P V$ must be of the form $\dots, Y \gg_P Z$, $Z \gg_P V$, \dots , where $Y \gg_P Z$ comes from (i). But in G we would have $\dots, Y \gg_P X$, $X \gg_P V$, \dots , contradicting **CYC** of G .

SOL: σ does not modify S and $\text{pvar}(P') \subseteq \text{pvar}(P)$.

IB

LIN: Due to linearity of G , σ does not modify the right-hand sides of P .

EX: $\text{pvar}(P') = \text{pvar}(P) \setminus \{X\} \subseteq \text{evar}(G') = \text{evar}(G) \setminus \{X\}$.

CYC: Since X is produced, only approximation statements $l \rightarrow s$ with $X \in \text{var}(l)$ are affected by σ . In these cases, for each $Y \in \text{var}(t), Z \in \text{var}(s)$, the relation $Y \gg_P Z$ is created, where previously we had $Y \gg_P X, X \gg_P Z$. Therefore \gg_P^* is not enlarged.

SOL: σ does not modify S and $\text{pvar}(P') \subseteq \text{pvar}(P)$.

IIM

If G is admissible, so is $G'' \equiv \exists X, \bar{X}, \bar{U}. S \sqcap c(\bar{e}) \rightarrow X, X \rightarrow c(\bar{X}), P \sqcap E$. To apply **IIM** to G is equivalent to apply **OB2**, **DC2** to G'' .

EL

LIN, **CYC**, **SOL:** Trivial.

EX: $\text{pvar}(P') = \text{pvar}(P) \setminus \{X\} \subseteq \text{evar}(G') = \text{evar}(G) \setminus \{X\}$

NR2

LIN: \bar{t} is a linear tuple of C-terms with new variables.

EX: All the new variables \bar{X} are existentially quantified.

CYC: Variables in each t_i are new, hence not appearing in any left-hand side of approximation statements, with the exception of $r \rightarrow t$. So, if a cycle goes through a variable $X \in \text{var}(t_i)$, it must take the form $\dots, Y \gg_P X, X \gg_P Z, \dots$, where $Y \in \text{var}(e_i), Z \in \text{var}(t)$. But in G we would have $\dots, Y \gg_P Z, \dots$, contradicting **CYC** of G .

SOL: S is not changed and variables in t_i are new.

Correctness₁-

We proceed by considering CLNC failure rules one by one.

CF1

It is clear, since for no θ the statement $(c(\bar{a}) \bowtie d(\bar{b}))\theta \equiv c(\bar{a}\theta) \bowtie d(\bar{b}\theta)$ can be GORC-provable.

CY

Assume that θ is a solution of G . Then $X\theta \bowtie a\theta$ must be GORC-provable, which means that there exists $t \in \text{CTerm}$ such that $X\theta \rightarrow t$ and $a\theta \rightarrow t$ are both GORC-provable. From the facts that $X \not\equiv a$ and $X \in \text{svar}(a)$, it is not difficult to see that $X\theta$ is a strict subterm of t . Therefore, $X\theta$ and t are distinct total C-terms, and hence it cannot be true that $X\theta \rightarrow t$ is GORC-provable.

CF2

Similar to the case of **CF1**.

Correctness₂.

We again proceed rule by rule.

- DC1** It is clear that θ' is also a solution of G (GORC-proofs of $\mathbf{a}_i\theta \bowtie \mathbf{b}_i\theta$ can be extended to a GORC-proof of $\mathbf{c}(\bar{\mathbf{a}})\theta \bowtie \mathbf{c}(\bar{\mathbf{b}})\theta$).
- ID** Since X is not produced, $X\theta'$ must be a total C-term, and then $X\theta' \bowtie X\theta'$ is GORC-provable. Therefore θ' is also solution of G .
- BD** We prove that θ' is also a solution of G : θ' solution of G' implies that $X\theta'$ is a total C-term and $X\theta' \equiv \mathbf{s}\theta'$. Therefore $X\theta' \bowtie \mathbf{s}\theta'$ is GORC-provable. With respect to the rest of conditions in G (S, P, E), simply observe that $X\theta' = \mathbf{s}\theta'$ and $\sigma = \{X/\mathbf{s}\}$ imply $\sigma\theta' = \theta'$, and therefore $S\theta' = S\sigma\theta'$, and similarly for P and E .
- IM** We prove that θ' is also a solution of G : θ' solution of G' implies that $X\theta'$ is a total C-term, that $X\theta' = \mathbf{c}(\bar{X})\theta'$, and that $(X_i \bowtie \mathbf{e}_i)\sigma\theta'$ are GORC-provable. Now, as in the case of **BD**, $\sigma\theta' = \theta'$. Therefore $(X_i \bowtie \mathbf{e}_i)\theta'$ are GORC-provable, hence $\mathbf{c}(\bar{X})\theta' \bowtie \mathbf{c}(\bar{\mathbf{e}})\theta'$ is also provable. But $\mathbf{c}(\bar{X})\theta' \equiv X\theta'$. For the rest of conditions in G we argue as with **BD**.
- NR1** We prove that θ is a solution of G , where θ is identical to θ' except for the variables in \bar{X} , for which θ is the identity. Note that θ and θ' coincide over all the variables occurring in G . We limit ourselves to prove that $\mathbf{f}(\bar{\mathbf{e}})\theta' \bowtie \mathbf{a}\theta'$ is GORC-provable. For this, notice that θ' solution of G' implies that $\mathbf{e}_i\theta' \rightarrow \mathbf{t}_i\theta'$, $C\theta'$ and $\mathbf{r}\theta' \bowtie \mathbf{a}\theta'$ are all GORC-provable. In the latter case, this means that there exists a C-term \mathbf{t} such that $\mathbf{r}\theta' \rightarrow \mathbf{t}$ and $\mathbf{a}\theta' \rightarrow \mathbf{t}$ are GORC-provable. Now, $\mathbf{e}_i\theta' \rightarrow \mathbf{t}_i\theta'$, $C\theta'$ and $\mathbf{r}\theta' \rightarrow \mathbf{t}$ GORC-provable implies that $\mathbf{f}(\bar{\mathbf{e}})\theta' \rightarrow \mathbf{t}$, and therefore also $\mathbf{f}(\bar{\mathbf{e}})\theta' \bowtie \mathbf{a}\theta'$, are GORC-provable.
- DC2** We prove that θ' is also a solution of G : θ' solution of G' implies that $\mathbf{e}_i\theta' \rightarrow \mathbf{t}_i\theta'$ are GORC-provable, and therefore $\mathbf{c}(\bar{\mathbf{e}})\theta' \rightarrow \mathbf{c}(\bar{\mathbf{t}})$ is also GORC-provable.
- OB1** We prove that θ' is also a solution of G : θ' solution of G' implies that $X\theta' \equiv \mathbf{t}\theta'$, and then $X\theta' \rightarrow \mathbf{t}\theta'$ is GORC-provable. $X\theta' \equiv \mathbf{t}\theta'$ implies also that $\sigma\theta' = \theta'$. Therefore $S\theta' \equiv S\sigma\theta'$, and similarly for P, E .
- OB2** We prove that θ is a solution of G , where θ is defined as $X\theta \equiv \mathbf{t}\theta'$ and $Y\theta \equiv Y\theta'$ for $Y \neq X$. Since X does not occur in \mathbf{t} , we deduce $\mathbf{t}\theta \equiv \mathbf{t}\theta'$, and therefore $X\theta \rightarrow \mathbf{t}\theta$ is GORC-provable. Furthermore, $\theta = \sigma\theta'$, hence $S\theta \equiv S\sigma\theta'$, and similarly for P, E .
- IB** Identical to the case of **OB2**.
- IIM** We prove that θ is a solution of G , where θ is defined as $X\theta \equiv \mathbf{c}(\bar{X})\theta'$, $X_i\theta \equiv X_i$ for $X_i \in \bar{X}$ and $Y\theta \equiv Y\theta'$ for $Y \notin X \cup \bar{X}$. Since θ' is a solution of G' , $\mathbf{e}_i\sigma\theta' \rightarrow X_i\sigma\theta'$ are GORC-provable. Now, since $\mathbf{e}_i\sigma\theta' = \mathbf{e}_i\theta$ and $X_i\sigma\theta' = X_i\theta'$, we obtain that $\mathbf{e}_i\theta \rightarrow X_i\theta'$ are GORC-provable, and therefore $\mathbf{c}(\bar{\mathbf{e}})\theta \rightarrow \mathbf{c}(\bar{X})\theta' (= X\theta)$ is GORC-provable. For the rest of the conditions in G observe that, as in the case of **OB2**, $\theta = \sigma\theta'$.
- EL** It is clear that θ is a solution of G , where θ is defined as $X\theta \equiv \perp$ and $Y\theta \equiv Y\theta'$ for $Y \neq X$.
- NR2** We prove that θ is a solution of G , where θ is identical to θ' except for the variables in \bar{X} , for which θ is the identity. Note that θ and θ' coincide over all the variables occurring in G . We limit ourselves to prove that $\mathbf{f}(\bar{\mathbf{e}})\theta' \rightarrow \mathbf{t}\theta'$ is GORC-provable. For this, notice that θ' solution of G'

implies that $e_i\theta' \rightarrow t_i\theta'$, $C\theta'$ and $r\theta' \rightarrow t\theta'$ are all GORC-provable. But then $f(\bar{e})\theta' \rightarrow t\theta'$ is GORC-provable. \square

It is easy now to obtain the following result, stating that computed answers for a goal G are indeed solutions of G . We recall that, according to Lemma 6.1, we can give both proof-theoretic and model-theoretic readings to this result. The same remark holds for the Completeness Theorem 7.2 below.

Theorem 7.1. (Soundness of CLNC)

If G_0 is an initial goal and $G_0 \vdash G_1 \vdash \dots \vdash G_n$, where $G_n \equiv \exists \bar{U}. S \square \square$, then $\sigma_S \in \text{Sol}(G_0)$.

PROOF.

If we repeatedly backwards apply (Correctness₂) of Lemma 7.2, we obtain $\theta \in \text{Sol}(G_0)$ such that $\theta = \sigma_S[\mathcal{V} - \bigcup_{i=0}^n \text{evar}(G_i)]$. By noting that $\text{evar}(G_0) = \emptyset$ and $\text{var}(G_0) \cap \bigcup_{i=0}^n \text{evar}(G_i) = \emptyset$, we conclude $\theta = \sigma_S[\text{var}(G_0)]$. But then, since σ_S is a total C-substitution, $\sigma_S \in \text{Sol}(G_0)$. \square

We address now the question of completeness of CLNC. Given a solution θ of a goal G , we need to ensure the existence of some terminating sequence of CLNC-transformations, leading to a solved form whose associated answer substitution is more general than θ . In Definition 6.2 we have introduced ‘witnesses’ for solutions, which are multisets of GORC-proofs. Now we define a well-founded ordering over such witnesses, which is intended to ‘measure the distance’ of a goal from a solved form.

Definition 7.1. (Multiset ordering for witnesses)

Let \mathcal{R} be a program. If $\mathcal{M} \equiv \{\{\Pi_1, \dots, \Pi_n\}\}$ and $\mathcal{M}' \equiv \{\{\Pi'_1, \dots, \Pi'_m\}\}$ are multisets of GORC-proofs of approximation and joinability statements, we define

$$\mathcal{M} \triangleleft \mathcal{M}' \Leftrightarrow \{\{|\Pi_1|, \dots, |\Pi_n|\}\} \prec \{\{|\Pi'_1|, \dots, |\Pi'_m|\}\}$$

where $|\Pi|$ is the size (i.e., the number of inference steps) of Π , and \prec is the multiset extension [13] of the usual ordering over \mathbb{N} . \square

The overall idea for proving completeness is now the following: given a solution θ for a goal G which is not in solved form, there is some CLNC-transformation that is applicable to G . Moreover, any applicable CLNC-transformation can be used for performing a \vdash -step in such a way that θ is kept as solution for the new goal with a smaller witness. More formally, we can prove:

Lemma 7.3. (Progress Lemma)

Progress₁.- *If $G \not\equiv \text{FAIL}$ is not a solved form, then there exists some CLNC-transformation applicable to G .*

Progress₂.- *If \mathcal{M} is a witness of $\theta \in \text{Sol}(G)$, and T is any CLNC-transformation applicable to G , then there exist G', θ' and \mathcal{M}' such that:*

- (i) $G \vdash G'$ by means of the CLNC-transformation T
- (ii) \mathcal{M}' is a witness of $\theta' \in \text{Sol}(G')$

- (iii) $\mathcal{M}' \triangleleft \mathcal{M}$
- (iv) $\theta = \theta'[\mathcal{V} \setminus (\text{evar}(G) \cup \text{evar}(G'))]$.

PROOF.

Progress₁.-If G is not a solved form, then P or E are not empty. We will proceed by assuming gradually that no rule, except one (namely **EL**), is applicable to G , and then we will conclude that this remaining rule **EL** must be applicable.

Assume that failure rules are not applicable. Assume also that **DC1** and **NR1** are not applicable. Then, all the joinability statements in E must be of one of the following two forms:

$$X \bowtie Y \quad \text{or} \quad X \bowtie c(\bar{a})$$

Now assume that **ID**, **BD** and **IM** are not applicable. Then it must be the case that all X in the previous joinability statements must be produced variables.

Now assume that **DC2** and **OB** are not applicable. Then all the approximation statements in P must be of one of the forms

$$Y \rightarrow X \quad \text{or} \quad c(\bar{e}) \rightarrow X \quad \text{or} \quad f(\bar{e}) \rightarrow t$$

Now, if **IB** is not applicable, then the possible forms for approximation statements reduce to

$$c(\bar{e}) \rightarrow X \quad \text{or} \quad f(\bar{e}) \rightarrow t$$

where $c(\bar{e})$ is not a C-term. Now, if **IIM** and **NR2** are not applicable, then the previous forms reduce to

$$c(\bar{e}) \rightarrow X \quad \text{or} \quad f(\bar{e}) \rightarrow X$$

where X is not a demanded variable. Moreover, at this point E must be empty. Otherwise, E would include some statement $X \bowtie a$ with produced X , and we could apply either **IIM**, or **NR2** to some $e \rightarrow X$ occurring in P .

Finally, let X be minimal in the \gg_P relation (such minimal elements do exist, due to the finite number of variables occurring in G and the property **CYC** of admissible goals). Such X cannot appear in any other approximation statement in P , and therefore **EL** can be applied to the condition $e \rightarrow X$ where X appears.

Progress₂.-In each of the cases below, G' is the goal obtained by application of the corresponding CLNC-transformation. Unless otherwise stated, it is assumed that $\theta' = \theta$. We will use the following notations within this proof:

- $\Pi \rightsquigarrow \varphi$ indicates that Π is a GORC-proof of φ .
- $(\Pi_1 \ \& \ \dots \ \& \ \Pi_n) + \mathbf{R}$ denotes the GORC-proof which consists of Π_1 followed by \dots followed by Π_n followed by an application of the GORC-rule **R**.

DC1 \mathcal{M} must contain a proof Π_0 of $(c(\bar{a}) \bowtie c(\bar{b}))\theta$, which must take the form

$$\Pi_0 \equiv (\Pi \rightsquigarrow c(\bar{a})\theta \rightarrow t \ \& \ \Pi' \rightsquigarrow c(\bar{b})\theta \rightarrow t) + \mathbf{J}$$

where t is a C-term of the form $c(\bar{t})$. Π and Π' must be of the forms

$$\Pi \equiv (\dots \ \& \ \Pi_i \rightsquigarrow (a_i\theta \rightarrow t_i) \ \& \ \dots) + \mathbf{DC}$$

$$\Pi' \equiv (\dots \& \Pi'_i \rightsquigarrow (b_i\theta \rightarrow t_i) \& \dots) + \mathbf{DC}$$

Now, for each i , $\Pi''_i \equiv (\Pi_i \& \Pi'_i) + \mathbf{J}$ is a proof of $(a_i \bowtie b_i)\theta$. Since $|\Pi_0| > |\Pi''_i|$ for each i , we have $\mathcal{M}' \triangleleft \mathcal{M}$, where \mathcal{M}' is the result of replacing $\{\{\Pi_0\}\}$ by $\{\{\dots, \Pi''_i, \dots\}\}$ in \mathcal{M} .

- ID** We can take \mathcal{M}' as the result of deleting in \mathcal{M} the proof of $(X \bowtie X)\theta$.
- BD** Taking into account that $\theta = \sigma\theta$, we can take \mathcal{M}' as the result of deleting in \mathcal{M} the proof of $(X \bowtie s)\theta$.
- IM** \mathcal{M} must contain a proof Π_0 of $(X \bowtie c(\bar{e}))\theta$, which must take the form

$$\Pi_0 \equiv (\Pi \rightsquigarrow X\theta \rightarrow t \& \Pi' \rightsquigarrow c(\bar{e})\theta \rightarrow t) + \mathbf{J}$$

where t is a C-term of the form $c(\bar{t})$. It follows that $X\theta \equiv c(\bar{t})$. We take $X_i\theta' \equiv t_i\theta$ for X_i in \bar{X} , and $Y\theta' \equiv Y\theta$ for all other variables Y . It holds that $(S \square P \square E)\theta \equiv (S \square P \square E)\sigma\theta'$ and that $X\theta' \equiv c(\bar{X})\theta'$. Now, Π and Π' must be of the forms

$$\Pi \equiv (\dots \& \Pi_i \rightsquigarrow (t_i \rightarrow t_i) \& \dots) + \mathbf{DC}$$

$$\Pi' \equiv (\dots \& \Pi'_i \rightsquigarrow (e_i\theta \rightarrow t_i) \& \dots) + \mathbf{DC}$$

For each i , $\Pi''_i \equiv (\Pi_i \& \Pi'_i) + \mathbf{J}$ is a proof of $(X_i \bowtie e_i)\theta'$. Since $|\Pi_0| > |\Pi''_i|$ for each i , we have $\mathcal{M}' \triangleleft \mathcal{M}$, where \mathcal{M}' is the result of replacing $\{\{\Pi_0\}\}$ by $\{\{\dots, \Pi''_i, \dots\}\}$ in \mathcal{M} .

- NR1** \mathcal{M} must contain a proof Π_0 of $(f(\bar{e}) \bowtie a)\theta$, which must take the form

$$\Pi_0 \equiv (\Pi \rightsquigarrow f(\bar{e})\theta \rightarrow t \& \Pi' \rightsquigarrow a\theta \rightarrow t) + \mathbf{J}$$

where t is a total C-term, which implies that t is not \perp . Therefore Π must take the form $\Pi \equiv (\dots \& \Pi_i \rightsquigarrow (e_i\theta \rightarrow t'_i) \& \dots \& \mathcal{M}_C \rightsquigarrow C' \& \Pi'' \rightsquigarrow r' \rightarrow t) + \mathbf{OR}$ where $f(\bar{t}') \rightarrow r' \Leftarrow C'$ is a rule R' of $[\mathcal{R}]_{\perp}$ (\mathcal{M}_C indicates a multiset of proofs for the conditions in C'). By definition of $[\mathcal{R}]_{\perp}$, there exists a rule in \mathcal{R} with a variant $R \equiv f(\bar{t}) \rightarrow r \Leftarrow C$ with variables $\text{var}(R) = \{X_1, \dots, X_n\}$ disjoint of $\text{var}(G)$, such that $R' = R\mu$, where $\mu \in \text{CSubst}_{\perp}$ and $\text{dom}(\mu) \subseteq \{X_1, \dots, X_n\}$. Let G' be the goal obtained by applying **NR1** to G using the rule R . We take $\theta'(X_i) \equiv \mu(X_i)$ for X_i in \bar{X} , and $\theta'(Y) \equiv \theta(Y)$ for the all other variables Y . Now, we have $(e_i \rightarrow t_i)\theta' \equiv e_i\theta \rightarrow t_i\mu \equiv e_i\theta \rightarrow t'_i$, $C\theta' \equiv C\mu \equiv C'$, and $(r \bowtie a)\theta' \equiv r\mu \bowtie a\theta \equiv r' \bowtie a\theta$. Therefore Π_i serves, for each i , as proof of $(e_i \rightarrow t_i)\theta'$, \mathcal{M}_C serves as multiset of proofs for the conditions in $C\theta'$ and $(\Pi' \& \Pi'') + \mathbf{J}$ serves as a proof of $(r \bowtie a)\theta'$. Hence the witness \mathcal{M}' of G' obtained by replacing in \mathcal{M} the proof Π_0 by all these (shorter) proofs verifies $\mathcal{M}' \triangleleft \mathcal{M}$.

- DC2** \mathcal{M} must contain a proof Π of $(c(\bar{e}) \rightarrow c(\bar{t}))\theta$, which must take the form

$$\Pi \equiv (\dots \& \Pi_i \rightsquigarrow (e_i\theta \rightarrow t_i\theta) \& \dots) + \mathbf{DC}$$

Since $|\Pi| > |\Pi_i|$ for each i , we have $\mathcal{M}' \triangleleft \mathcal{M}$, where \mathcal{M}' is the result of replacing $\{\{\Pi\}\}$ by $\{\{\dots, \Pi_i, \dots\}\}$ in \mathcal{M} .

- OB1** \mathcal{M} must contain a proof Π_0 of $X\theta \rightarrow t\theta$. Due to the linearity of t it is possible to lift θ over $\text{var}(t)$ ($\subseteq \text{evar}(G)$, hence condition *(iv)* in the statement of our lemma is respected) to obtain $\theta' \sqsupseteq \theta$ such that $\theta' = \theta[\mathcal{V} \setminus \text{var}(t)]$ and $X\theta \equiv X\theta' \equiv t\theta'$. As an additional consequence we have $\sigma\theta' \sqsupseteq \theta$. What remains to prove is that θ' is solution of G' with witness $\mathcal{M}' \triangleleft \mathcal{M}$. Since S does not contain variables in $\text{var}(t)$, it is not difficult

to see that $S\sigma\theta'$ consists only of identities. For joinability statements $\mathbf{a} \bowtie \mathbf{b} \in E$, as \mathcal{M} is a witness of θ , it must contain a GORC-proof Π of $\mathbf{a}\theta \bowtie \mathbf{b}\theta$. Π must take the form

$$\Pi \equiv (\Pi_1 \rightsquigarrow (\mathbf{a}\theta \rightarrow \mathbf{u}) \ \& \ \Pi_2 \rightsquigarrow (\mathbf{b}\theta \rightarrow \mathbf{u})) + \mathbf{J}$$

for some $\mathbf{u} \in \mathbf{CTerm}$. Since $\sigma\theta' \sqsupseteq \theta$, the Monotonicity Lemma 4.1 ensures the existence of Π'_1, Π'_2 , GORC-proofs of $\mathbf{a}\sigma\theta' \rightarrow \mathbf{u}$ and $\mathbf{b}\sigma\theta' \rightarrow \mathbf{u}$, such that $|\Pi'_1| = |\Pi_1|$, $|\Pi'_2| = |\Pi_2|$. Then $\Pi' \equiv (\Pi'_1 \ \& \ \Pi'_2) + \mathbf{J}$ is a proof of $\mathbf{a}\sigma\theta' \bowtie \mathbf{b}\sigma\theta'$ of the same length as Π . A similar reasoning can be done for approximation statements $\mathbf{e} \rightarrow \mathbf{s} \in P$ other than $\mathbf{X} \rightarrow \mathbf{t}$ (in this case it is important to use the linearity condition of P , which ensures that $\text{var}(\mathbf{s}) \cap \text{var}(\mathbf{t}) = \emptyset$, and therefore $\mathbf{s}\theta' \equiv \mathbf{s}\theta$; otherwise we could not apply the Monotonicity Lemma 4.1). Collecting all these new proofs results in a witness \mathcal{M}' which verifies $\mathcal{M}' \triangleleft \mathcal{M}$, because one proof in \mathcal{M} has been deleted (that of $\mathbf{X}\theta \rightarrow \mathbf{t}\theta$, since the condition $\mathbf{X} \rightarrow \mathbf{t}$ does not appear in G'), and the rest have been replaced by proofs of the same length.

OB2 Similar to the case of **OB1**.

IB \mathcal{M} must include a proof Π of $(\mathbf{t} \rightarrow \mathbf{X})\theta$, which implies that $\mathbf{t}\theta \sqsupseteq \mathbf{X}\theta$. If we consider $\theta' = \sigma\theta$, it holds that $\theta' = \theta[\mathcal{V} \setminus \{\mathbf{X}\}]$, $\theta' \sqsupseteq \theta$ and $\sigma\theta' = \theta'$. Now we can reason similarly to the case of **OB1** (in both cases the effect of θ' is to increase the value of the righthand side of an approximation condition; the absence of produced variables in S , linearity of P and the Monotonicity Lemma 4.1 make the rest).

IIM \mathcal{M} must contain a proof Π of $(\mathbf{c}(\bar{\mathbf{e}}) \rightarrow \mathbf{X})\theta$. Since \mathbf{X} is a demanded variable, $\mathbf{X}\theta$ is not \perp ⁸, and therefore $\mathbf{X}\theta \equiv \mathbf{c}(\bar{\mathbf{t}})$, for some $\mathbf{t}_i \in \mathbf{CTerm}_\perp$. Π must then take the form

$$\Pi \equiv (\dots \ \& \ \Pi_i \rightsquigarrow (\mathbf{e}_i\theta \rightarrow \mathbf{t}_i) \ \& \ \dots) + \mathbf{DC}$$

We define $\theta'(\mathbf{X}_i) \equiv \mathbf{t}_i$ and $\theta' = \theta[\mathcal{V} \setminus \bar{\mathbf{X}}]$. It holds that $\sigma\theta' = \theta[\mathcal{V} \setminus \bar{\mathbf{X}}]$, which implies that $(S \square P \square E)\theta \equiv (S \square P \square E)\sigma\theta'$. It implies also that $(\mathbf{e}_i \rightarrow \mathbf{X}_i)\sigma\theta' \equiv (\mathbf{e}_i\theta \rightarrow \mathbf{t}_i)$, and then each proof Π_i is also a proof of the statement $(\mathbf{e}_i \rightarrow \mathbf{X}_i)\sigma\theta'$. Hence, we can take \mathcal{M}' as the result of replacing in \mathcal{M} the proof Π by the shorter proofs \dots, Π_i, \dots .

EL We can take \mathcal{M}' as the result of deleting in \mathcal{M} the proof of $(\mathbf{e} \rightarrow \mathbf{X})\theta$.

NR2 \mathcal{M} must contain a proof Π of $(\mathbf{f}(\bar{\mathbf{e}}) \rightarrow \mathbf{t})\theta$. The condition imposed over \mathbf{t} in **NR2** implies that $\mathbf{t}\theta$ is not \perp . Therefore, Π must take the form

$$\Pi \equiv (\dots \ \& \ \Pi_i \rightsquigarrow (\mathbf{e}_i\theta \rightarrow \mathbf{t}'_i) \ \& \ \dots \ \mathcal{M}_C \rightsquigarrow C' \ \& \ \Pi' \rightsquigarrow r' \rightarrow \mathbf{t} \ \& \) + \mathbf{OR}$$

and the rest of the reasoning is similar (simpler, indeed) to the case of **NR1**.

□

We can now prove that any solution for a goal is subsumed by a computed answer, i.e., our goal solving calculus is complete.

Theorem 7.2. (Completeness of CLNC)

Let \mathcal{R} be a program, G an initial goal and $\theta \in \text{Sol}(G)$. Then there exists a solved form $\exists \bar{\mathbf{U}}. S \square \square$ such that $G \#^ \exists \bar{\mathbf{U}}. S \square \square$ and $\sigma_S \leq \theta[\text{var}(G)]$.*

⁸This is in fact all what is needed. The condition $\mathbf{X} \in \text{dvar}(G)$ could be relaxed to any other condition implying $\mathbf{X}\theta \neq \perp$.

PROOF.

Thanks to Lemma 7.3 it is possible to construct a derivation

$$G \equiv G_0 \# G_1 \# G_2 \# \dots$$

for which there exist $\theta_0 = \theta$, $\theta_1, \theta_2, \dots$ and $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \dots$ such that $\theta_i \in \text{Sol}(G_i)$, $\theta_i = \theta_{i-1}[\mathcal{V} \setminus (\text{evar}(G_{i-1}) \cup \text{evar}(G_i))]$, \mathcal{M}_i is a witness of $\theta_i \in \text{Sol}(G_i)$ and $\mathcal{M}_i \triangleleft \mathcal{M}_{i-1}$. Since \triangleleft is well founded, such a derivation must be finite, ending with a solved form $G_n \equiv \exists \overline{U}. S \square \square$. Since $\text{evar}(G_0) = \emptyset$ and $\text{var}(G_0) \cap \text{evar}(G_i) = \emptyset$ for all $i = 1, \dots, n$ it is easy to see that $\theta_n = \theta[\text{var}(G)]$. Now, if $X \in \text{var}(G)$ and there is an equation $X = s$ in S , we can use the facts $\theta = \theta_n[\text{var}(G)]$ and $\theta_n \in \text{Sol}(S)$ to obtain $X\theta \equiv X\theta_n \equiv s\theta_n \equiv X\sigma_S\theta_n$. It follows that $\theta = \sigma_S\theta_n[\text{var}(G)]$, and thus $\sigma_S \leq \theta[\text{var}(G)]$. \square

This theorem can be considered as the main result of this section. However, it is important to notice that Lemma 7.3 contains relevant information which is lost in Theorem 7.2, namely the *don't care* nature of the choice of the CLNC-rule to be applied (among all the applicable rules). This property has been sometimes called *strong completeness* [37] and leaves much room to experiment with different selection strategies.

8. PRACTICABILITY OF THE APPROACH

Up to this point, we have shown a quite general and expressive framework for declarative programming, based on non-deterministic lazy functions. Nevertheless, there is still a big gap between our current presentation of lazy narrowing and an implemented system. In this section, we argue that this gap can be filled with the help of a suitable narrowing strategy, and we report on an actual experience with an implemented system.

In fact, our narrowing calculus CLNC is not intended as an operational model, but rather as an abstract description of goal solving that provides a very convenient basis for soundness and completeness proofs, while ignoring irrelevant control issues and implementation details. Since CLNC-derivations proceed by outermost narrowing and sharing, it is fair to say that the behaviour of lazy evaluation is properly reflected by the structure of *successful* CLNC-derivations. Nevertheless, the lack of an efficient mechanism to guide (and possibly avoid) don't know choices renders CLNC inadequate as a concrete specification of computational behaviour. Consider for example a CRWL-program \mathcal{R} consisting of the following rewrite rules:

$$\begin{array}{ll} \text{none}(\text{zero}) & \rightarrow \text{zero} \\ \text{none}(\text{suc}(\text{N})) & \rightarrow \text{none}(\text{N}) \\ \text{one}(\text{zero}) & \rightarrow \text{suc}(\text{zero}) \\ \text{one}(\text{suc}(\text{N})) & \rightarrow \text{one}(\text{N}) \\ \text{leq}(\text{zero}, \text{Y}) & \rightarrow \text{true} \\ \text{leq}(\text{suc}(\text{X}), \text{zero}) & \rightarrow \text{false} \\ \text{leq}(\text{suc}(\text{X}), \text{suc}(\text{Y})) & \rightarrow \text{leq}(\text{X}, \text{Y}) \end{array}$$

Assume also a C-term num , built from the constructors $zero$ and suc , that represents some very big natural number. Given the initial goal

$$G_0 \equiv \square \quad \square \quad \text{leq}(\text{one}(num), \text{none}(num)) \bowtie R$$

the CLNC-calculus has a don't know choice between three different applications of the **NR1** transformation, according to the three rewrite rules given for leq in the program. The three corresponding alternatives for the next goal are as follows:

$$\begin{array}{lll} G_1 \equiv \exists Y. & \square \quad \text{one}(num) \rightarrow zero, \text{none}(num) \rightarrow Y & \square \quad \text{true} \bowtie R \\ G_2 \equiv \exists X. & \square \quad \text{one}(num) \rightarrow \text{suc}(X), \text{none}(num) \rightarrow zero & \square \quad \text{false} \bowtie R \\ G_3 \equiv \exists X.Y. & \square \quad \text{one}(num) \rightarrow \text{suc}(X), \text{none}(num) \rightarrow \text{suc}(Y) & \square \quad \text{leq}(X, Y) \bowtie R \end{array}$$

Assume that these alternatives are tried sequentially in the given order. Then, G_1 will fail after an expensive computation of $\text{one}(num)$, yielding value $\text{suc}(zero)$. Next, G_2 will succeed after performing a second evaluation of $\text{one}(num)$, as well as another expensive evaluation of $\text{none}(num)$, with value $zero$. If the user asks for alternative solutions, G_3 will be attempted, and it will fail after repeating again the evaluations of both $\text{one}(num)$ and $\text{none}(num)$. In contrast to this unfortunate behaviour, a purely functional language would be able to compute false as the value of $\text{leq}(\text{one}(num), \text{none}(num))$ by means of a single deterministic computation, involving no backtracking.

In order to avoid such pitfalls as those shown by the previous example, CLNC must be refined by means of some efficient strategy. As shown by the example, trying the different rewrite rules independently, in some sequential order, leads easily to the repeated evaluation of terms given as actual arguments for some function call (such as $\text{one}(num)$ above). A better strategy for dealing with a function call $e \equiv f(e_1, \dots, e_n)$ can be informally described as follows:

- Step 1** Discard all the rewrite rules for f which fail due to constructor clash between the left-hand side of the rule and e . If all rules are discarded then fail. Otherwise, go to **Step 2**.
- Step 2** If, among the remaining rewrite rules for f , there is a single one whose left-hand side matches e , then apply this rule and discard all the others. If there are several rewrite rules whose left-hand sides match e , then don't know choice is unavoidable. Otherwise, go to **Step 3**.
- Step 3** If possible, choose an outermost subterm a of some argument term e_i , such that a 's evaluation is demanded by the patterns in all the left-hand sides of applicable rewrite rules for f , and go to **Step 4**. If this is not possible, then don't know choice among the rewrite rules is unavoidable.
- Step 4** Recursively using the same strategy, compute a *head normal form* (shortly, *hnf*) h for a ; the resulting h will be either a variable or a term headed by a constructor. Consider the new function call e' which has replaced e after the computation of h , and go to **Step 1** again.

For the example shown above, this strategy leads to a fully deterministic evaluation of the term $\text{leq}(\text{one}(num), \text{none}(num))$:

- The left-hand sides of all the rewrite rules for leq demand the evaluation of the first argument to *hnf*. Therefore, the first argument term $\text{one}(num)$ is

chosen and its *hnf* $\text{succ}(\text{zero})$ is computed.

- Now, the first rewrite rule for leq is discarded, and the left-hand sides of the two remaining rules demand the evaluation of the second argument term to *hnf*. Thus, the second argument term $\text{none}(\text{num})$ is chosen and its *hnf* zero is computed.
- At this point, we are dealing with the term $\text{leq}(\text{succ}(\text{zero}), \text{zero})$. The second rewrite rule for leq is the only one whose left-hand side matches this term. Hence, we can commit to this rule, that leads to the result false .

A more formal elaboration of these ideas, using so-called *definitional trees* [2] to guide unification with the left-hand sides of rewrite rules, led to the *demand driven strategy* developed in [32]. An independent formulation of essentially the same strategy was presented under the name *needed narrowing* in [4] for so called *inductively sequential rewrite systems*, a proper subclass of CRWL-programs which defines only deterministic functions by means of unconditional rewrite rules. In [4] it was proved that needed narrowing for inductively sequential rewrite systems enjoys very nice optimality properties. In particular, when restricted to term evaluation, needed narrowing achieves deterministic needed reductions of minimal length (under the assumption of sharing). More recently, needed narrowing has been extended to a computational model for functional logic programming that accommodates a *residuation* mechanism [26], and some of its optimality properties have been established for a broader class of rewrite systems that allows for non-deterministic functions [3]. The rewrite systems in [3] are more general than those in [4], but still less general than CRWL-programs. In particular, they are unconditional.

In summary, there are known optimality results for demand driven (also called needed) narrowing strategies that apply to inductively sequential (and thus unconditional) CRWL-programs. On the other hand, the formulation of a demand driven strategy given in [32] relies essentially on the left-hand sides of rewrite rules, and is applicable to any CRWL-program⁹. The missing piece to complete the picture is a theoretical result that would guarantee soundness and completeness of a demand driven strategy w.r.t. CRWL-semantics (as given by the free term models in Section 5 or, equivalently, the rewriting calculi in Section 4). We strongly conjecture that such a result can be established.

Taking the previous conjecture for granted, it follows that any implementation of the demand driven strategy (as presented in [32]) can be safely used for the execution of CRWL-programs, provided that sharing is supported. Such an implementation is provided by the \mathcal{TCY} system [11]. Our experiments with \mathcal{TCY} have shown indeed that CRWL-programs (written with minor syntactic modifications in order to conform \mathcal{TCY} 's concrete syntax), can be correctly executed. Sharing avoids computing those solutions that would be unsound w.r.t. call-time choice. Among other CRWL-programs, we have tested the examples presented in Section 2 in the \mathcal{TCY} system. Our expectation that the ‘permutation sort’ program using the ‘lazy generate and test’ approach should be much more efficient than the naive,

⁹Originally, [32] was not intended to cover the case of non-deterministic functions.

Prolog-like ‘generate and test’ version has been confirmed by our experiments.

In addition to the demand-driven strategy, \mathcal{TOY} provides additional features (polymorphic types, higher-order functional and logic computations, syntactic disequality constraints, arithmetic constraints over the real numbers) which merge also without problems with non-deterministic functions, the whole result being an attractive practical framework for a productive declarative programming. Some special optimizations for deterministic functions, such as dynamic cut [33] or simplification [24, 25] are currently not supported by the \mathcal{TOY} system. However, they can be implemented in principle for all those defined functions that are known to be deterministic, either on the basis of user given declarations, or because of known decidable sufficient conditions, as e.g. those proposed in [18].

9. CONCLUSIONS

We have achieved a logical presentation of a quite general approach to declarative programming, based on the notion of non-deterministic lazy function. Besides proof calculi and a model theory for a constructor-based conditional rewriting logic CRWL, we have presented a sound and strongly complete lazy narrowing calculus CLNC, which is able to support sharing and to identify safe cases for eager replacement of variables. All this shows the potential of our approach as a firm foundation for the development of functional logic languages.

Admittedly, CLNC is an abstract description of goal solving rather than a concrete operational model. Nevertheless, we have argued that it can be refined by adopting demand-driven narrowing strategies, so that a convenient and efficiently implementable operational model is obtained. This claim has been further supported by the successful use of an implemented functional logic programming system [11] for executing CRWL-programs.

Planned future work will include further theoretical investigation of completeness results for demand-driven narrowing strategies w.r.t. CRWL’s semantics, as well as suitable extensions of CRWL to obtain a logical foundation for other features of the \mathcal{TOY} system, such as higher-order functions, types, and constraints.

Acknowledgements: We would like to thank Ana Gil-Luezas, Puri Arenas-Sánchez, Rafael Caballero-Roldán and Jaime Sánchez-Hernández for useful comments and contributions to implementation work. The constructive criticisms of several anonymous referees have helped to improve an older version of the paper.

REFERENCES

1. S. Antoy, Non-determinism and Lazy Evaluation in Logic Programming, in *Proc. LOPSTR’91*, 1991, pp. 318-331.
2. S. Antoy, Definitional Trees, in *Proc. ALP’92*, Springer LNCS 632, 1992, pp. 143-157.
3. S. Antoy, Optimal Non-Deterministic Functional Logic Computations, in *Proc. ALP’97*, Springer LNCS 1298, 1997, pp. 16-30.

4. S. Antoy, R. Echahed, and M. Hanus, A Needed Narrowing Strategy, in *Proc. 21st ACM Symp. on Principles of Prog. Lang.*, Portland, 1994, pp. 268-279.
5. K.R. Apt, Logic Programming, In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, vol. B, Chapter 10, pp. 493-574, Elsevier and The MIT Press, 1990.
6. P.Arenas-Sánchez and M. Rodríguez-Artalejo, A Semantic Framework for Functional-Logic Programming with Algebraic Polymorphic Types, in *Proc. TAP-SOFT'97*, Springer LNCS 1214, 1997, pp.453-464.
7. P.Arenas-Sánchez and M. Rodríguez-Artalejo, A Lazy Narrowing Calculus for Functional Logic Programming with Algebraic Polymorphic Types, in *Proc. ILPS'97*, The MIT Press, 1997, pp.53-69.
8. H.P. Barendregt, M.C.J.D. van Eeckelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep, Term Graph Rewriting, in: *Proc. PARLE'87*, Springer LNCS 259, 1987, pp. 141-158.
9. R. Bird and P. Wadler, *Introduction to Functional Programming*, Prentice Hall, 1988.
10. G. Boudol, Computational semantics of term rewriting systems, in: M. Nivat and J.C. Reynolds (eds.), *Algebraic methods in semantics*, Chapter 5, pp. 169-236, Cambridge University Press, 1985.
11. R. Caballero-Roldán , F.J. López-Fraguas and J. Sánchez-Hernández, User's Manual for TOY, Tech. Rep. DIA 97/57, UCM Madrid, 1997. System available at <http://mozart.sip.ucm.es/toy>
12. N. Dershowitz and J.P. Jouannaud, Rewrite Systems, in J.van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. B, Chapter 6, pp. 243-320, Elsevier and The MIT Press, 1990.
13. N. Dershowitz and Z. Manna, Proving Termination with Multiset Orderings, *Comm. of the ACM* 22(8):465-476 (1979).
14. 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):86-113 (1993).
15. E. Giovannetti, G. Levi, C. Moiso and C. Palamidessi, Kernel-LEAF: A Logic plus Functional Language, *Journal of Computer and System Science* 42(2):139-185 (1991).
16. J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, Initial Algebra Semantics and Continuous Algebras, *Journal of the ACM* 24(1):68-95 (1977).
17. J.C. González-Moreno, Programación Lógica de Orden Superior con Combinadores, Ph.D. Thesis, Univ. Complutense Madrid, 1994. (In Spanish)
18. J.C. González-Moreno, M.T. Hortalá-González and M. Rodríguez-Artalejo, Denotational versus Declarative Semantics for Functional Programming, in *Proc. CSL'91*, Springer LNCS 626, 1992, pp. 134-148.
19. 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, in *Proc. CSL'92*, Springer LNCS 702, 1993, pp. 216-230.
20. J.C. González-Moreno, M.T. Hortalá-González and M. Rodríguez-Artalejo, A Higher Order Rewriting Logic for Functional Logic Programming, in *Proc. ICLP'97*, The MIT Press, 1997, pp. 153-167.

21. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas and M. Rodríguez-Artalejo, A Rewriting Logic for Declarative Programming, in *Proc. ESOP'96*, Springer LNCS 1058, 1996, pp. 156-172.
22. M. Hanus, Improving Control of Logic Programs by Using Functional Logic Languages, in *Proc. PLILP'92*, Springer LNCS 631, 1992, pp. 1-23.
23. M. Hanus, The Integration of Functions into Logic Programming: A Survey, *Journal of Logic Programming, Special issue 'Ten Years of Logic Programming'* (19&20):583-628 (1994).
24. M. Hanus, Lazy Unification with Simplification, in *Proc. ESOP'94*, Springer LNCS 778, 1994, pp. 272-286.
25. M. Hanus, Combining Lazy Narrowing and Simplification, in *Proc. PLILP'94*, Springer LNCS 844, 1994, pp. 370-384.
26. M. Hanus, A Unified Computation Model for Functional and Logic Programming, in *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, Paris, 1997, pp. 80-93.
27. H. Hussmann, Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting, *Journal of Logic Programming* 12:237-255 (1992).
28. H. Hussmann, *Non-determinism in Algebraic Specifications and Algebraic Programs*, Birkhäuser Verlag, 1993.
29. C. Kirchner, H. Kirchner and M. Vittek, Designing Constraint Logic Programming Languages Using Computational Systems. in V. Saraswat and P. van Hentenryck (eds.), *Principles and Practice of Constraint Programming*, Chapter 8, pp. 133-160, The MIT Press, 1995.
30. J.W. Klop, Term rewriting systems, In S. Abramsky, D.M. Gabbay and T.S.E. Maibaum (eds.), *Handbook of Logic in Computer Science*, vol.2, pp. 2-116, Oxford University Press, 1992.
31. J.W. Lloyd, *Foundations of Logic Programming*, Springer Verlag, 2nd. ed., 1987.
32. R. Loogen, F.J. López-Fraguas and M. Rodríguez-Artalejo, A Demand Driven Computation Strategy for Lazy Narrowing, in *Proc. PLILP' 93*, Springer LNCS 714, 1993, pp. 184-200.
33. R. Loogen and S. Winkler, Dynamic detection of determinism in functional logic languages, *Theoretical Computer Science* 142:59-87 (1995).
34. F.J. López-Fraguas, Programación Funcional y Lógica con Restricciones, Ph.D. Thesis, Univ. Complutense Madrid, 1994. (In Spanish)
35. J. Meseguer, Conditional Rewriting Logic as a Unified Model of Concurrency, *Theoretical Computer Science* 96:73-155 (1992).
36. J. Meseguer, A Logical Theory of Concurrent Objects and Its Realization in the Maude Language, in G. Agha, P. Wegner and A. Yonezawa (eds.), *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, pp. 314-390, 1993.
37. A. Middeldorp, S. Okui and T. Ida, Lazy Narrowing: Strong Completeness and Eager Variable Elimination, *Theoretical Computer Science* 167:95-130 (1996).
38. B. Möller, On the Algebraic Specification of Infinite Objects - Ordered and Continuous Models of Algebraic Types, *Acta Informatica* 22:537-578 (1985).
39. J.M. Molina-Bravo and E. Pimentel, Modularity in Functional-Logic Programming, in *Proc. ICLP'97*, The MIT Press, 1997, pp. 183-197.
40. A. Middeldorp and E. Hamoen, Completeness Results for Basic Narrowing, *Applicable Algebra in Engineering, Comm. and Comp.* 5:213-253 (1994).

-
41. D. Miller, G. Nadathur, F. Pfenning and A. Scedrov, Uniform Proofs as a Foundation of Logic Programming, *Annals of Pure and Applied Logic* 51:125-157 (1991).
 42. J.J. Moreno-Navarro and M. Rodríguez-Artalejo, Logic Programming with Functions and Predicates: The Language BABEL, *Journal of Logic Programming* 12:191-223 (1992).
 43. L. Naish, *Negation and Control in Prolog*, Springer LNCS 238, 1987.
 44. J. Peterson, K. Hammond (eds.) Report on the Programming Language Haskell. A Non-strict, Purely Functional Language, Version 1.4, April 7, 1997.
 45. B.J. Pierce, *Basic Category Theory for Computer Scientists*, The MIT Press, Foundations of Computer Science Series, 1991.
 46. C. Prehofer, A Call-by-Need Strategy for Higher-Order Functional Logic Programming, in *Proc. ILPS'95*, The MIT Press, 1995, pp. 147-161.
 47. A. Sarmiento-Escalona, Una aproximación a la Programación Lógica con Funciones Indeterministas, Ph.D. Thesis, Univ. La Coruña, 1992. (In Spanish)
 48. D.S. Scott, Domains for Denotational Semantics, in *Proc. ICALP'82*. Springer LNCS 140, 1982, pp. 577-613.
 49. H. Søndergaard and P. Sestoft, Non-determinism in Functional Languages, *The Computer Journal* 35(5):514-523 (1992).
 50. L. Sterling and E. Shapiro, *The Art of Prolog*, The MIT Press, 1994.
 51. P. Wadler, How to Replace Failure by a List of Successes, in *Proc. IFIP Int. Conf. on Funct. Prog. Lang. and Computer Architectures*, Springer LNCS 201, 1985, pp. 113-128.
 52. G. Winskel, On Powerdomains and Modality, *Theoretical Computer Science* 36:127-137 (1985).