Lazy List Comprehension in Logic Programming

BIRGIT ELBL, Fakultät für Informatik, UniBw München, 85577 Neubiberg, Germany.
E-mail: birgit@informatik.unibw-muenchen.de

Abstract

The pure prolog evaluation of a goal yields a list of answers, but the tools provided for manipulating these structures are very poor. We discuss augmenting pure prolog with a list comprehension construct that offers the possibility of referring to the finite or infinite list of answers produced. Thus meta-predicates can be defined. A substructural calculus is used to give an axiomatic semantics to the extended language. Soundness and completeness of the intended evaluation with respect to this semantics is proved.

Keywords: Logic programming, lazy list comprehension, axiomatic semantics.

1 Introduction

A well-known candidate for the use of a fragment of predicate logic as programming language is the logic of Horn clauses, usually regarded as the theoretical foundation of the programming language prolog (see e.g. [10]). A set of Horn clauses can be viewed as a set of recursive definitions for predicates where the defining expressions are built using standard logical connectives. With respect to these predicates, goals are evaluated which may contain variables for which we seek answer terms. The same predicates can be used to check membership in a relation or to generate elements of this relation. In a prolog style evaluation this generation is an enumeration according to a fixed strategy. In terms of SLD resolution this can be described as follows: consider the SLD tree for the leftmost selection function where the branches are ordered according to the order of clauses in the program; on traversing this tree using a depth-first strategy and visiting left branches before their right neighbours, output the answers associated with success branches. In this manner a finite or infinite list of answers may be produced. Furthermore there are two kinds of finite lists: either the whole SLD tree is finite in which case we can search it completely and, after producing all answers, receive a signal that we are done; or the search leads into an infinite branch without reaching further answers. In the latter case we are not told that there is no further answer. As this list of answers is generated lazily — either driven by the user who asks for more answers by backtracking or initiated by another goal whose failure for certain answers invokes backtracking and a search for alternatives — we can nevertheless obtain the answers that are found before entering the infinite branch. Hence, what is actually computed by a pure prolog program is a stream or lazy list of answers.

In the pure Horn/prolog setting, i.e. if we consider the set of programs that can be read as definite Horn clause sets, but assume prologs evaluation, we cannot refer to the stream of answers inside a program, e.g. by picking the nth answer or testing for emptiness (without a version of negation-as-failure). In the full programming language prolog the situation is different: there are several means of control as restricting the search to the first answer or
testing for failure. Finite collections of answers can be gathered in a list, and we could define a predicate picking the \( n \)th answer for a goal. But these features are outside the scope of the logical semantics.

The course followed here is to add the powerful construct of lazy list comprehension to pure prolog. This is named after and reminds of the axioms of set comprehension or the programming construct list comprehension as present in Miranda [18]. The language is extended by this construct building terms from formulas and by facilities to manipulate them. This introduces lazy lists as objects to compute with as in the functional setting. Lazy list algorithms can already be described in pure prolog by implementing reduction to canonical form for new terms which are meant to stand for lazy lists and applying this reduction appropriately (see [15, 13] Section 7). In contrast to this way to model lazy structures we are concerned with the access to those already present: the comprehension construct as defined below relates lazy lists and goals, referring to the list of answers produced by a logic program evaluation.

This programming language enjoys a logical semantics. The pure prolog evaluation procedure deviates from the standard declarative semantics in more than one way: on the one hand, it is incomplete — there may be correct answers which are not found. On the other hand, the standard semantics identifies programs where the sets of answers coincide, ignoring the order in which they are produced. In the sequel we will be interested in these differences, hence we assume a more refined semantics.

As there is a close relation between lazy lists and goals, we can in general define list functionals and use them as control operators. As an example we show how to define control similar to some prolog constructs. But we can also define a version of negation-as-failure, extending the pure prolog kernel in a quite different way.

The paper is organized as follows: first we introduce the syntax of logic programs with lazy lists and describe a goal-directed evaluation. Then we discuss some examples before we turn to the axiomatic semantics and prove soundness and completeness of the evaluation. Finally, we have a look at conservative extensions and inversion properties as these are easily obtained as well as useful in verification.

2 Logic programs with lazy lists

2.1 Syntax

Logic programs are formulated in a formal language depending on a given signature. We presuppose a set VAR of variables, a set CONS of constructor symbols, and a set PRED of predicate symbols. Every constructor symbol has a fixed arity. The notation CONS\(_n\) is used for the set of constructor symbols of arity \( n \). We consider a typed language. Types are built from the simple types \( i \) and \( \langle i \rangle \) using \( \to \). The type \( \langle i \rangle \) is the type of streams or lazy lists, \( \to \) is used to build a function type. We presuppose a fixed assignment of types to variables and type tuples to predicate symbols, where the length of the tuple is the arity of the predicate symbol. The notation VAR\(_\alpha\), or PRED\(_\alpha\) are used for the set of variables or predicates to which we assigned \( \alpha \) or \( \bar{\alpha} \). Based on these symbols we define the set of terms of type \( \alpha \) and the set of formulas or goals inductively. As usual, \( t : \alpha \) is a shorthand for \( 't \) is a term of type \( \alpha \), similarly \( \bar{t} : \bar{\alpha} \) for lists of terms and types.

**Definition 2.1**

The sets of terms and goals are simultaneously inductively defined by:

- **terms:** \( x : \alpha \) for every \( x \in \text{VAR}_\alpha \).
goals: \(1, F, 0\) are goals,
\(p(\overline{t})\) is a goal if \(p \in \text{PRED}_\alpha\) and \(\overline{t} : \alpha\).
\(A \otimes B\) and \(A \ast B\) are goals if \(A, B\) are goals, and
\(\exists x. A\) is a goal if \(A\) is a goal and \(x \in \text{VAR}_\alpha\).

Here \(\otimes\) is used for conjunction corresponding to prologs ‘\&\&’ and \(\ast\) is the disjunction corresponding to ‘\|’; and \(\exists\) is the existential quantifier usually omitted in prolog programs. The constants \(F, 1\) are neutral elements w.r.t. disjunction and conjunction respectively. The reading as logical conjunction, disjunction and quantification abstracts from the operational behaviour but also ignores the effect this operational interpretation has. Pure prolog is known to be incomplete with respect to that interpretation. A more exact, informal reading of the connectives could be phrased as follows: assume that goals stand for some procedures which take substitutions as their input and produce substitutions as output. Then \(\ast\) corresponds to an execution in sequence without interaction. Executing \(A \otimes B\) for \(\theta\) means executing \(B\) for all \(\theta'\) where \(\sigma\) is produced by \(A\) for \(\theta\); on output the results are composed. The quantification \(\exists x\) introduces a new local variable, for which no input is taken and no output produced. The term \(\langle y \mid G \rangle\) should be read as the stream of terms obtained from the output of \(G\) restricted to \(y\). Here, \(G\) may take substitutions for the variables different from \(y\) as input, and \(y\) is renamed to avoid a name clash if it occurs free in one of the substituted terms. The operation \(\text{fst}\) takes the first element of a stream. \(\text{NO}\) is a special constant for the result produced by applying \(\text{fst}\) to the empty stream. For a first informal understanding of \(\epsilon\) we state that \(\epsilon u. t\) has the same value as \(\text{fst}(\langle y \mid \text{U}(y = t)\rangle)\) for terms \(t\) that are built from constructors and the variable \(u\) only if \(u, y\) are distinct. Further explanation is provided later. The remaining constructs are interpreted as usual. As a shorthand we introduce \(t(n), n \geq 1\), for \(\text{fst}(\text{rest}(\ldots \text{rest}(t) \ldots))\), where \(\text{rest}\) is applied \(n - 1\) times and \(t : \langle i \rangle\). Furthermore \(\mu_n y. G\) stands for \((y \mid G(n))\). In the sequel we assume for every \(n \geq 1\) an \(n\text{-ary constructor symbol}\) \(\text{tuple}_n\) and write \(\overline{t}\) for \(\text{tuple}_n(\overline{t})\).

As the meaning of the connectives deviates from the classical meaning, we use a different notation. The symbols \(\otimes, 1, 0\) are taken from linear logic [7], as the proof rules in \(K_\otimes\) (see Section 3) and the semantics of the calculus suggest a correspondence (see [6] for further discussion). As the relation \(\otimes/\ast\) is essentially different from the relation \(\otimes/\text{\&\&}\) in linear logic, we do not adopt this notation for disjunction.

Using \(\epsilon, \lambda, \exists\), or a list comprehension bind the variable which is explicitly mentioned. Terms that are equal up to renaming of bound variables and permuting \(\epsilon\) bindings are called congruent. In the sequel we will identify congruent terms. The notation \(\epsilon(t)\) is used for the \(\epsilon\) closure of a term \(t\) of type \(\iota\). The notation \(\text{FV}(t)\) is used for the set of free variables of an expression \(t\). We assume an arbitrary but fixed ordering on the set of \(\iota\) variables and use \(\text{fV}(t)\) for the term \((\overline{\iota})\) where \(\overline{\iota}\) is the ordered list of variables of type \(\iota\) in \(t\).
Programs consist of definitions for predicates. Some predicates are considered ‘built-in’. We assume that our language has at least the following predicate symbols: =, =,, and ≈, which have type (t, t) and are written infix, furthermore for every n > 0 an n-ary predicate symbol \( \nabla_n \) having type (t, \ldots, t). The subscript is usually omitted. The procedures associated with the equality predicates can be described as unification, comparison up to renaming, evaluation–unification, and will be explained later in detail. The procedure associated with \( \nabla(\tilde{t}) \) is as follows: input \( \theta \); if \( \tilde{t}\theta \) are distinct variables then output the identity substitution and stop else loop without returning an answer. Although \( \nabla(\tilde{t}) \) can be executed, it should be considered rather a tool for specification than for programming. It helps to restrict a judgement to the case where certain variables are uninstantiated, thus proving valuable in the axiomatization presented below. This will become clear when we turn to the semantics. We are now ready to define programs.

**Definition 2.2**

A program \( P \) is a finite set of clauses \( p(\tilde{x}) : - \text{Def}_p^P(\tilde{x}) \) satisfying the following conditions:

1. \( \text{Def}_p^P(\tilde{x}) \) is a goal where all free variables are in \( \{ \tilde{x} \} \) and \( \tilde{x} \) is a list of distinct variables meeting the typing of the predicate symbol \( p \).

2. No predicate symbol on the left is in =, =,, ≈, \( \nabla \), no predicate symbol \( p \) occurs more than once on the left side, and every predicate symbol occurring on the right is either =, =,, ≈, \( \nabla \) or occurs also on some left side.

The goal \( \text{Def}_p^P(\tilde{x}) \) is called \( p \)'s definition in \( P \). An executable goal for \( P \) is a goal where all free variables have type \( t \) and all predicate symbols \( p \) not in =, =,, ≈, \( \nabla \) have a definition in \( P \).

In the sequel we use \( \text{Def}_p^P(\tilde{t}) \) for the goal obtained from \( \text{Def}_p^P(\tilde{x}) \) by substituting \( \tilde{t} \) for \( \tilde{x} \) where bound variables are renamed to avoid name clashes with variables free in \( \tilde{t} \) if necessary. Pure prolog is embedded in this language by the transformation that is usually a preparatory step for building the completion of a program [1], using now \( \odot \), \( * \) as connectives: for every \( n \)-ary predicate symbol \( p \) proceed as follows: first choose new variables \( \tilde{x} = x_1, \ldots, x_n \), where \( n \) is the arity of \( p \); then replace every clause \( p(\tilde{t}) : - G \) by \( p(\tilde{x}) : - x_1 = t_1 \odot \ldots \odot x_n = t_n \odot G \); next, add existential quantifiers on the right side, binding all variables except \( \tilde{x} \); finally replace the sequence \( p(\tilde{x}) : - G_1, \ldots, p(\tilde{x}) : - G_m \) obtained so far by a single definition \( p(\tilde{x}) : - G_1 \ast \ldots \ast G_m \). If \( p \) is a predicate symbol in the language of the program for which there is no clause, add the definition \( p(\tilde{x}) : - F \). For a detailed treatment of this fragment see [5, 6].

**Example 2.3**

Consider the prolog program:

\[
\begin{align*}
\text{add}(X, 0, X). \\
\text{add}(X, s(Y), s(Z)) : - \text{add}(X, Y, Z). \\
\text{rep}(X, Y, Y). \\
\text{rep}(X, Y, Z) : - \text{add}(X, Y, U), \text{rep}(Y, U, Z). \\
\text{fib}(X) : - \text{rep}(0, s(0), X).
\end{align*}
\]

The corresponding predicate definitions are:
add\( (x, y, z) :- \) \( \text{Eu}(x = u \otimes y = 0 \otimes z = u) \ast \\
\quad \text{EuEu(w)(x = u \otimes y = s(v) \otimes z = s(w) \otimes \text{add}(u, v, w))} \)

\( \text{rep}(x, y, z) :- \) \( \text{Eu}(x = u \otimes y = v \otimes z = v) \ast \\
\quad \text{EuEu(w)(x = u \otimes y = v \otimes z = w \otimes} \\
\quad \text{add}(u, v, w') \otimes \text{rep}(v, w', w)) \)

\( \text{fibs}(x) :- \text{Eu}(x = u \otimes \text{rep}(0, s(0), u)) \)

Next we extend this program by predicates for extracting elements out of lists:

\( \text{answer}(L, x, y) :- x = s(0) \otimes y \approx \text{fst}(L) \ast \\
\quad \exists z (x = s(z) \otimes \text{answer}(\text{rest}(L), z, y)) \)

\( \text{call}(L, y) :- y \approx \text{fst}(L) \ast \text{call}(\text{rest}(L), y) \)

The goals \( \text{fibs}(y), \text{call}(y \mid \text{fibs}(y)), y \), and \( \text{answer}(y \mid \text{fibs}(y)), s^n(0), y \) \((n \in \mathbb{N})\) are executable.

The notation \( \bar{t} = \bar{s} \) where \( \bar{t} = t_1, \ldots, t_n \) and \( \bar{s} = s_1, \ldots, s_n \) is used as a shorthand for \( t_1 = s_1 \otimes \ldots \otimes t_n = s_n \). The output of the evaluation of a goal \( G(\bar{x}) \) is a stream of substitutions \( \{\bar{t}/\bar{s}\} \) where substitutions that are equal up to renaming the substituted terms are considered equivalent and \( \bar{t} \) are built using constructor symbols only. We represent these answer substitutions by goals \( \text{Ea}(\bar{x} = \bar{t}) \). According to our reading of * we use

\[
\text{Ea}^{(1)}(\bar{x}^{(1)} = \bar{t}^{(1)}) \ast \ldots \ast \text{Ea}^{(m)}(\bar{x}^{(m)} = \bar{t}^{(m)})(s0)
\]

to represent finite total (or partial) lazy lists of answers. To put it differently, the output on evaluating these goals can be read off immediately.

**Definition 2.4**

The set \( \text{ct} \) is the set of types of \( t \) that are built from variables using constructor symbols only. The set \( \text{ct} \) is the set \( \{e \bar{u}. t \mid t \in \text{ct}\} \), and \( \text{ect} := \text{ct} \cup \{\text{NO}\} \).

**Definition 2.5**

Let \( m \geq 0 \). The goals \( S_1 \ast \ldots \ast S_m \) and \( S_1 \ast \ldots \ast S_m \ast \text{NO} \) are called solved if each \( S_i \) has the form \( \text{Ea}(\bar{x} = \bar{t}) \) for some pairwise distinct variables \( \bar{x}, \bar{u} = x_1, \ldots, x_n, u_1, \ldots, u_k \) and ct terms \( \bar{t} = t_1, \ldots, t_n \) where no \( x_i \) occurs in any \( t_j \).

Note that \( m = 0 \) is included, thus \( \text{F} \) and \( \text{NO} \) are solved. Also \( n \) may equal 0, thus \( 1, 1 \ast 1 \), etc. are solved.

Now we can ascribe a meaning to all goals by defining which solved goals ‘approximate’ them. First, this is done for the built-ins \( =, \ast, \), and \( \nabla \).

### 2.2 Normalization and unification

To explain our built-ins, an extension of normalization and unification is fixed.

**Definition 2.6**

The relation \( \mapsto \) is defined as follows: \((\lambda x.t)s \mapsto t[s/x] \) \((\beta \text{ reduction})\), and \( e(\bar{t}, e\bar{u}.s, \bar{r}) \mapsto \) \( eu.e(\bar{t}, \bar{s}, \bar{r}) \) if \( u \) not free in \( \bar{t}, \bar{r} \), furthermore \( e(\bar{t}, \text{NO}, \bar{r}) \mapsto \text{NO} \), and \( e\bar{u}.\text{NO} \mapsto \text{NO} \). The relation \( \triangleright \) is the corresponding closure under context, \( \triangleright \ast \) denotes the reflexive transitive closure.

The notions ‘normal’ and ‘normal form’ are used with respect to \( \triangleright \).
Next we extend unification [16] to terms for some calculus is strongly normalizing, and this carries over immediately to the terms and goals defined above as long as no further reductions are considered. So every term is \( \beta \)-normalizing. Let \( \nu(t) \) denote the length of the longest \( \beta \)-reduction sequence starting at \( t \). Let us call the remaining reductions \( \tilde{\beta} \). The \( \tilde{\beta} \)-reductions do not influence \( \beta \) redexes except perhaps removing some: if \( t \tilde{\beta} t' \tilde{\beta} t'' \), then there is a term \( s \) so that \( t \tilde{\beta} s \tilde{\beta}^* t'' \). Hence \( t \tilde{\beta} t' \) implies \( \nu(t') \leq \nu(t) \). On the other hand the number of \( \tilde{\beta} \) steps that can be performed in sequence is obviously bounded. Combining this we can conclude that \( \tilde{\beta} \) is strongly normalizing.

Checking all types of pairs of reductions, we find that \( \tilde{\beta} \) is locally confluent. (Remember that we identify congruent terms and that euev.\( t \equiv_e \text{euev.} t \).) As its normalizing, this implies Church–Rosser and the uniqueness of the normal form. By induction on \( t \), it is easy to prove the following:

Let \( t \) be a normal term where all free variables have type \( \iota \). If \( t : \alpha \Rightarrow \beta \) then \( t \equiv \lambda x.t' \) for some \( t' \). If \( t : \langle \iota \rangle \) then \( t \equiv \text{rest}^n(y \mid G) \) for some goal \( G, n \in \mathbb{N} \). If \( t : \iota \) then \( t \equiv NO \) or \( t \equiv e\bar{u}.t_0 \{ \bar{s}/\bar{x} \} \) for a ct term \( t_0 \), distinct variables \( \bar{x} = x_1, \ldots, x_n \) and distinct terms \( \bar{s} = s_1, \ldots, s_n \) where \( s_i \equiv \langle y_i \mid G_i \rangle(n_i) \) for some goals \( G_i \) and \( n_i \geq 1 \).

Next we extend unification [16] to terms \( t, s \) of type \( \iota \) where all free variable have type \( \iota \). We do this step by step:

1. If \( t, s \) are ct terms, unify using Robinson’s algorithm. If the result is a substitution \( \{ \bar{r}/\bar{x} \} \) return \( \bar{x} = \bar{r} \).
2. If \( t, s \) are normal and do not contain \( e \) or NO; replace all subterms \( \langle y \mid G \rangle(n) \) by new constants. Equal terms are replaced by identical constants. Then unify as usual using Robinson’s algorithm. If the result is some substitution \( \{ \bar{r}/\bar{x} \} \) where \( \bar{r} \) do not contain any new constant, then return \( \bar{x} = \bar{r} \) else return \( F \).
3. If \( t \equiv e\bar{u}.t_0, s \equiv e\bar{v}.s_0 \), where \( t_0, s_0 \) satisfy the condition in 2. and \( \bar{u} \) do not occur in \( s \), \( \bar{v} \) do not occur in \( t \), then unify \( t_0, s_0 \) according to 2. If the result is \( \bar{x} = \bar{r} \) then return the result of removing the equalities \( x_i = r_i \) where \( x_i \) occurs in \( \bar{u}, \bar{v} \) and all quantifications \( Ex_i \) or \( Ev_i \) where the bound variable does not occur in \( \bar{r} \) from \( E\bar{u}E\bar{v}(\bar{x} = \bar{r}) \). Otherwise return \( F \).
4. For unification, normalize \( t, s \) first. If \( \text{nf}(t) \equiv \text{nf}(s) \) return 1. If exactly one of \( \text{nf}(t), \text{nf}(s) \) is NO then return NO. Otherwise unify \( t, s \) as in 3.

The result of this procedure is denoted by \( \text{unify}(t, s) \). For the atomic formulas with a built-in predicate different from \( \approx \) we define ‘solve’.

**Definition 2.8**

Let \( \bar{t} = t_1, \ldots, t_n \) be terms of type \( \iota \). Then solve(\( t_1 = t_2 := \text{unify}(t_1, t_2) \). If \( \text{nf}(t) \{ \bar{u}/\bar{x} \} \equiv \text{nf}(s) \) for a renaming \( \{ \bar{u}/\bar{x} \} \) where \( \bar{v}(t) = (\bar{x}) \), then solve(\( t := s \) \( ) := \bar{x} = \bar{u} \). Otherwise solve(\( t := s \) \( ) := F \). If \( \text{nf}(t_1), \ldots, \text{nf}(t_n) \) are distinct variables, then solve(\( \nabla(\bar{t}) \) \( ) := 1 \). Otherwise solve(\( A \) is not defined.
EXAMPLE 2.9
Consider the unification of \( t \equiv c(eu.s(u), u, 0) \) and \( s \equiv ev.c(s(0), s(v), w) \): the normal form of the first term is \( eu'.c(s(u'), u, 0) \), the second is normal. Unifying \( c(s(u'), u, 0) \) and \( c(s(0), s(v), w) \) in the usual way yields \( \{ 0/u', s(v)/u, 0/w \} \), hence the result of ‘unify’ for these terms is \( u' = 0 \otimes u = s(v) \otimes w = 0 \) (up to reordering the equations). To compute unify\((t, s)\) we consider \( Eu'Ev(u' = 0 \otimes u = s(v) \otimes w = 0) \): the equation \( u' = 0 \) is removed as \( u' \) is quantified, the quantifier \( Ev \) is kept because \( v \) does indeed occur in the right sides. Thus, the result ist \( Ev(u = v) \otimes w = 0 \).

Now consider \( t \equiv NO \) and \( s \equiv ev.c(s(0), s(v), w) \): both terms are normal, exactly one is \( NO \), hence unify\((t, s) = F \). Finally let \( t \equiv NO \) and \( s \equiv ev.c(NO, s(v), w) \): normalizing \( s \) yields \( NO \). Hence unify\((t, s) = 1 \).

2.3 Evaluation
The first step in describing how to search for solutions and answers is to define a reduction relation on goals not containing \( \infty \). The sets of goals and terms are denoted by \( \text{goal} \) or \( \text{term} \) respectively. Here we consider executable goals only, i.e. goals where all free variables have type \( t \) and all non-built-in predicate names have a definition in \( P \).

DEFINITION 2.10
Let \( P \) be a program.

1. Let \( \triangleright_0 \) denote the following reduction relation on goals:

\[
1 \otimes G \triangleright_0 G, \ 0 \otimes G \triangleright_0 0, \text{ and } F \otimes G \triangleright_0 F,
\]

\[
(G_1 \otimes G_2) \otimes G_3 \triangleright_0 G_1 \otimes (G_2 \otimes G_3)
\]

\[
(G_1 \otimes G_2) \otimes H \triangleright_0 G_1 \otimes H \otimes G_2 \otimes H
\]

\[
\text{Ex}G_1 \otimes G_2 \triangleright_0 \text{Ex}(G_1 \otimes G_2) \text{ if } x \text{ not free in } G_2
\]

\[
p(\bar{f})(\otimes G) \triangleright_0 \text{Det}_p(\bar{f})(\otimes G)
\]

\[
b(\bar{f})(\otimes G) \triangleright_0 \text{solve}(b(\bar{f}))(\otimes G) \text{ if } b \text{ is a built-in predicate, all free variables in } \bar{f} \text{ have type } t \text{, solve}(b(\bar{f})) \text{ defined and different from } b(\bar{f}).
\]

2. Let \( \triangleright_0 \) also denote the following relation on \( (\text{goal} \times \text{ct}) \times (\text{goal} \times \text{ct})^\star \) (the empty list is denoted by \( - \)):

\[
(G, t) \triangleright_0 (G', t) \text{ if } G \triangleright_0 G'
\]

\[
(F, t) \triangleright_0 -
\]

\[
(x = s \otimes G, t) \triangleright_0 (G[s/x], t[s/x]) \text{ if } s \in \text{ct}, x \text{ not in } s
\]

\[
(x = s, t) \triangleright_0 (1, t[s/x]) \text{ if } s \in \text{ct}, x \text{ not in } s
\]

\[
(\text{Ex}G, t) \triangleright_0 (1, t[s/x]) \text{ if } s \in \text{ct}, x \text{ not in } t
\]

\[
(G_1 \otimes G_2, t) \triangleright_0 (G_1, t), (G_2, t)
\]

3. Let \( \triangleright_0 \) also denote the following relation on \( (\text{goal} \times \text{ct})^\star \times (\text{goal} \times \text{ct})^\star \):

\[
\mathcal{A}, (G, t), B \triangleright_0 \mathcal{A}, C, B \text{ if } (G, t) \triangleright_0 C \text{ and } \mathcal{A} = (1, s_1), \ldots, (1, s_n) \text{ for some } s_1, \ldots, s_n.
\]

Now we extend this relation to include the evaluation of terms with list expressions:

DEFINITION 2.11
Let \( P \) be a program. Then \( \triangleright \subset (\text{goal} \times \text{ct}) \times (\text{goal} \times \text{ct})^\star \) and \( \triangleright \subset \text{term} \times \text{term} \) are the smallest relations satisfying:

1. \( \mathcal{A} \triangleright B \text{ if } \mathcal{A} \triangleright_0 B \)

\[
\mathcal{A}, (t \approx s(\otimes G), r), B \triangleright \mathcal{A}, (t' = s'(\otimes G), r), B \text{ if } t \triangleright t' \text{ and } s \triangleright s' \text{ and } \mathcal{A} = (1, s_1), \ldots, (1, s_n) \text{ for some } s_1, \ldots, s_n
\]
Consider the program
\[ (y \mid G)(n) \downarrow \epsilon(r_n) \text{ if } (G, y) \triangleright^* (1, r_1), \ldots, (1, r_n), \mathcal{A} \text{ for some } \mathcal{A} \]
\[ (y \mid G)(n) \downarrow \text{ NO if } (G, y) \triangleright^* (1, r_1), \ldots, (1, r_k) \text{ for some } k < n \]
\[ t \downarrow \text{ NO if } nf(t) = \text{ NO} \]
\[ t \downarrow \text{ nf}(e\epsilon, t_0[\bar{r} / \bar{s}]) \text{ if } nf(t) \equiv e\epsilon, t_0[\bar{r} / \bar{s}] \text{ for variables } \bar{r} \text{ occurring in } t_0 \text{ and terms } \bar{s} = s_1, \ldots, s_m \text{ so that, for all } 1 \leq i \leq m, s_i \equiv \langle y_t \mid G_t(n_i) \rangle \text{ for some } y_t, G_t, n_i, \text{ and } s_i \downarrow r_i. \]

Here we use \( \triangleright^* \) for the reflexive, transitive closure. If necessary we add a subscript \( P \) to \( \triangleright \) and \( \downarrow \). In the sequel we refer frequently to the number of \textit{computation} steps (in contrast to \textit{reduction} steps). This is meant to include all steps necessary to show that the single \( \triangleright \) steps are indeed reductions. In the definition above there are conditions of the kind ‘if \( x \) is not free in ...’ in the clauses for the existential quantifier. As we identified congruent terms, this should be read as an instruction to rename the bound occurrence of \( x \) before performing a reduction. Using different variants here yields same results up to renaming.

\textbf{Lemma 2.12}

Let \( P \) be a program, \( G \) an executable goal. If \( (G_1, t_1), \ldots, (G_n, t_n) \triangleright^* (H_1, s_1), \ldots, (H_m, s_m) \) and \( \delta_1, \ldots, \delta_n \) are renamings for \( (G_1, t_1), \ldots, (G_n, t_n) \) respectively, then there are renamings \( \delta'_1, \ldots, \delta'_m \) for \( (H_1, s_1), \ldots, (H_m, s_m) \) respectively so that \( (G_1, t_1) \triangleright^* (H_1, s_1) \delta'_1, \ldots, (G_n, t_n) \triangleright^* (H_n, s_n) \delta'_m. \)

If \( (G, t) \triangleright^* (1, s_1), \ldots, (1, s_n), \mathcal{A} \) and \( (G, t) \triangleright^* (1, r_1), \ldots, (1, r_m), \mathcal{B} \) then \( s_i \) equals \( r_i \) up to renaming for all \( 1 \leq i \leq \min\{n, m\} \). If \( (G, t) \triangleright^* (1, s_1), \ldots, (1, s_n) \) and \( (G, t) \triangleright^* (1, r_1), \ldots, (1, r_m), \mathcal{B} \) then \( m \leq n. \)

\textbf{Proof.} Induction on the number of computation steps.

\textbf{Example 2.13}

Consider the program
\[ p(F, y) : - y = c(0) \ast y \approx c(eu.(Fc(u))) \]
and the goal \( p(f_2, z) \) where \( f_2 : \equiv \lambda x. \text{fst}((y \mid p(f_1, x) \odot y = x)) \) and \( f_1 : \equiv \lambda z. z. \)

\[ (p(f_2, z), z) \triangleright (z = c(0) \ast z \approx c(eu.(f_2c(u))), z) \]
\[ \triangleright (z = c(0), z) \approx c(eu.(f_2c(u))), z) \]
\[ \triangleright (1, c(0)), (z \approx c(eu.(f_2c(u))), z). \]

The normal form of \( c(eu.(f_2c(u))) \) is \( c(u).c(\text{fst}((y \mid p(f_1, c(u)) \odot y = c(u)))). \) Hence we consider the goal \( p(f_1, c(u)) \odot y = c(u): \)
\[ (p(f_1, c(u)) \odot y = c(u), y) \triangleright^* \]
\[ (c(u) = c(0) \odot y = c(u), y), (c(u) \approx c(eu.(f_1c(v)))) \odot y = c(u), y) \triangleright^* \]
\[ (1, c(0)), (c(u) \approx c(eu.(f_1c(v)))) \odot y = c(u), y). \]

So \( c(eu.(f_2c(u))) \downarrow \approx eu.c(c(0)), \) hence
\[ (p(f_2, z), z) \triangleright^* (1, c(0)), (z = eu.c(c(0)), z) \triangleright^* (1, c(0)), (1, c(c(0))). \]

Thus \( \langle z \mid p(f_2, z) \rangle(1) \downarrow c(0), \langle z \mid p(f_2, z) \rangle(2) \downarrow c(0), \langle z \mid p(f_2, z) \rangle(3) \downarrow \text{ NO}. \)

The definition of \( \downarrow \) determines when and to what extent list expressions are evaluated. The goal \( G \) inside \( (y \mid G) \) is called only in evaluating an executable atom \( t \approx s \) and only if the
list expression occurs in the normal forms of $t, s$. List expressions which are removed by normalization are not evaluated. Due to the normal form (see Lemma 2.7), a list expression $\langle y \mid G \rangle$ in a normal term $t : i$ with only $i$ variables free occurs as $\langle y \mid G \rangle(n)$ and the number $n$ tells how many answers should be produced. Note that atoms $\langle y \mid G \rangle \approx (z \mid H) \not\approx$ do not occur, as $\not\approx$ has type $(t, t)$. If the call for the $n$th answer does not terminate for $G$ or $H$, then $\langle y \mid G \rangle(n) \approx (y \mid H)(n)$ cannot be reduced, even if $G, H$ are identical, in contrast to the goal $\langle y \mid G \rangle(n) = \langle y \mid G \rangle(n)$.

In the sequel we use the semantic domain of streams over a set $M$:

**Definition 2.14**

For every set $M$ let:

$$\text{stream}(M) = \bigcup_{n \in \mathbb{N}} M^n \cup \bigcup_{n \in \mathbb{N}} M^n \times \{\bot\} \cup \bigcup M^N$$

($\bot \not\in M$). We use the notation $[m_1, \ldots, m_n]$ for the elements in $M^n$ (finite total streams) and $[m_1, \ldots, m_n, \bot]$ for the elements in $M^n \times \{\bot\}$ (finite partial streams). Streams are ordered by the following relation: let $\sqsubseteq$ be the smallest reflexive relation satisfying $[m_1, \ldots, m_n, \bot] \sqsubseteq (m_{n+1})_{\mathbb{N}}$ for all $n \in \mathbb{N}$, $[m_1, \ldots, m_n, \bot] \subseteq [m_1, \ldots, m_n, \bot]$ if $n \leq n'$, and $[m_1, \ldots, m_n, \bot] \subseteq [m_1, \ldots, m_n]$ if $n \leq n'$ for all $(m_{n+1})_{\mathbb{N}} \in M^N$. The $i$th element of a stream $S$ is denoted by $S(i)$. For streams of ect terms we extend this by $S(i) := \text{NO}$ if $S = [m_1, \ldots, m_n]$ and $i > n$.

**Definition 2.15**

Let $P$ be a program, $G$ an executable goal for $P$, $\text{fv}(G) = (\bar{x})$, and $t : i$ with only $i$ variables free. Then for every ct term $s$:

$$[G]_s \equiv \text{lub}\{\{e(t_1), \ldots, e(t_n), \bot\} \mid \exists A : (G, s) \triangleright^* (1, t_1), \ldots, (1, t_n), A\}$$

$$\cup\{\{e(t_1), \ldots, e(t_n)\} \mid (G, s) \triangleright^* (1, t_1), \ldots, (1, t_n)\}.$$}

Furthermore $[G] := [G]_x$ and $[\bar{t}] := t_0$ if $t \downarrow t_0$.

As a consequence of Lemma 2.12, the set considered in this definition is a chain, hence the least upper bound exists.

**Example 2.16**

Let $S := \exists \bar{x}[\bar{x} = \bar{t}(1)] \ast \ldots \ast \exists \bar{x}[\bar{x} = \bar{t}(n)](\ast 0)$ be solved, $\text{FV}(S) \subseteq \{\bar{x}\}$, $\text{FV}(\bar{t}(i)) = \{\bar{t}(i)\}$. Then $[S]_x := \{e(\bar{t}(1)), \ldots, e((\bar{t}(n)))(\bot, \bot)\}$.

By definition, $[[y \mid G](n)] \simeq [G]_y(n)$. Restriction to $y$ and picking the $n$th component can be exchanged as stated in the next lemma. Furthermore we consider the effect of renaming and reordering variables.

**Lemma 2.17**

Let $P$ be a program, $G$ an executable goal for $P$, $\text{fv}(G) = (\bar{x})$, and $y$ a variable. Assume that $S$ is a solved goal satisfying $\text{FV}(S) \subseteq \{\bar{x}\}$, and $\delta$ renaming for $\{\bar{x}\}$ so that $\text{fv}(G \delta) = (\bar{u})$.

(a) Then $[[t \delta]] \simeq [t]_\delta$, $[[G \delta]]_{(x)} = [G]$, $[[y \mid G](n) \delta]] \simeq [[y \mid G](n)]$, furthermore $[[S]_{(x)}] \subseteq [[G]] \iff [[S]]_{(x)} \subseteq [G]$. If $\pi$ is a permutation then $[G]_{\pi(x)}$ can be obtained from $[G]_{(x)}$ by applying $\pi$ componentwise.

(b) If $y$ is not free in $G$ then $[[y \mid G](n)] \simeq \varepsilon y, y$ iff $[[G](n)] \in \text{ect}$. If $\bar{x} = x_1, \ldots, x_m$, $1 \leq i \leq m$, and $\bar{t} \in \text{ct}$ then $[[x_i \mid G](n)] \simeq \varepsilon (t)$ iff there are terms $t_j$ so that $[[G](n)] \simeq \varepsilon (t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_m)$. Furthermore $[[y \mid G](n)] \simeq \text{NO}$ iff $[[G](n)] \simeq \text{NO}$. 

Lazy List Comprehension in Logic Programming 755
Variables. Consider a lazy list expression \( \varphi_i \). An aspect relevant to lazy list expressions as well as all solutions predicates is how to treat \( \varphi_i \) for \( 1 \leq i \leq n \). Then by induction on the number of the computation steps we can prove the following:

1. If \((G_1, t_1), \ldots, (G_n, t_n) \triangleright^* (H_1, s_1), \ldots, (H_m, s_m)\) then there are terms \( s^i_j \), \( 1 \leq i \leq m \), \( 1 \leq j \leq k \) so that

\[
(G_1, (t_1, \varphi_1)), \ldots, (G_n, (t_n, \varphi_n)) \triangleright^* (H_1, (s_1 \delta_1, \varphi)), \ldots, (H_m, (s_m \delta_m, \varphi))
\]

for some renamings \( \delta_1, \ldots, \delta_m \).

2. Let \( \pi \) be a permutation of \( k \)-tuples. If \((G_1, (\varphi_1)), \ldots, (G_n, (\varphi_n)) \triangleright^* (H_1, s_1), \ldots, (H_m, s_m)\) then there are lists \( \bar{s}_1, \ldots, \bar{s}_m \) of length \( k \) so that \( s_i \equiv (\bar{s}_i) \) for \( 1 \leq i \leq m \) and

\[
(G_1, \pi(\varphi_1)), \ldots, (G_n, \pi(\varphi_n)) \triangleright^* (H_1, \pi(\bar{s}_1)), \ldots, (H_m, \pi(\bar{s}_m)).
\]

3. If \((G_1, (t_1, \varphi_1)), \ldots, (G_n, (t_n, \varphi_n)) \triangleright^* (H_1, (s_1 \delta_1, \varphi)), \ldots, (H_m, (s_m \delta_m, \varphi))\) then

\[
(G_1, t_1), \ldots, (G_n, t_n) \triangleright^* (H_1, s_1), \ldots, (H_m, s_m).
\]

(1)–(3) yield (b). For (a) use (2) and Lemma 2.12.

**Definition 2.18**

Let \( P \) be a program, \( G \) an executable goal for \( P \), and \( \text{fv}_i(G) = \{ \bar{x} \} \). If \( \langle y \mid G \rangle(n) \downarrow t \) then \( t \) is called the \( n \)-th value for \( y \) in \( G \) computed by \( P \). A solved goal \( S \), \( \text{FV}(S) \subseteq \{ \bar{x} \} \), is called a computed solution for \( G \) if \( [S]_{\{x\}} \subseteq [G]_{\{x\}} \).

As a consequence of Lemma 2.17, these notions do not depend on the order of variables. As \( [\langle y \mid G \rangle(n)] \simeq [G]_{y}(n) \), the term \( \langle y \mid G \rangle \) can be used to refer to \([G]_{y} \). In contrast to the all solutions predicates common in non-pure Prolog this includes the infinite and finite partial case, as does the lazy ‘collect’ predicate suggested in [9] (see [4], [17] for the usual findall, setof, bagof, [14] for a discussion of several variants). The fact that \( \langle y \mid G \rangle \) is a term of type \( \langle \bar{i} \rangle \), not a goal creating an ordinary list, implies of course differences in use and semantics.

An aspect relevant to lazy list expressions as well as all solutions predicates is how to treat variables. Consider a lazy list expression \( \langle y \mid G \rangle \) containing free variables. As \( \langle y \mid G \rangle : \langle \bar{i} \rangle \), its meaning should be a stream of terms without any side effect of variable bindings. This implies a decision for a deterministic variant like ‘findall’. Now let us turn to variables in answers and consider

\[
? - \text{findall}(Y, Y = f(X), S), \text{member}(f(1), S), \text{member}(f(2), S)
\]

(this is an adaptation of an example in [14]) with standard `findall`. The goal fails, as the evaluation of the first atom returns \( S = [f(X)] \), the second instantiates \( X = 1 \), and \( \text{member}(f(2), [f(1)]) \) fails. In contrast to this,

\[
? - 1\text{member}(f(1), \langle y \mid y = f(x) \rangle), 1\text{member}(f(2), \langle y \mid y = f(x) \rangle)
\]

succeeds. (Here we assume an appropriate membership predicate for lazy lists, failing, of course, only if the first argument is no member of the second and the second is finite and total.) To see this, note that \( \langle y \mid y = f(x) \rangle(1) \downarrow \text{ex} f(x) \), and \( f(1) = \text{ex} f(x) \) as well.

As \( f(2) = \text{ex} f(x) \) succeed without instantiation.
Lazy list comprehension is a means to refer to the list of answers produced when evaluating a goal. Hence it can be used to describe failure or restriction to the first answer which are usually considered means of control.

**Example 2.19**

Let \( p, q_1, q_2 \in \text{PRED}_r \) be \( n \)-ary predicate symbols and:

\[
\begin{align*}
\text{fail}_{\text{if}}(p)(\vec{x}) & :\equiv \text{NO} \approx \text{fst}(\langle z \mid p(\vec{x}) \rangle) \\
\text{once}(p)(\vec{x}) & :\equiv (\vec{x}) \approx \text{fst}(\langle z \mid \forall p(\vec{x}) \odot z = (\vec{x}) \rangle) \\
\text{if \_then\_else}(p, q_1, q_2)(\vec{x}) & :\equiv \text{once}(p)(\vec{x}) \odot q_1(\vec{x}) \ast \text{fail}_{\text{if}}(p)(\vec{x}) \odot q_2(\vec{x}).
\end{align*}
\]

Let \( \vec{t} \) be ct terms, \( \text{FV}(\vec{t}) = \{ \vec{g} \} \). Then \( \text{fail}_{\text{if}}(p)(\vec{t}) \) has the solution \( 1 \) if \( p(\vec{t}) \) fails. It fails if \( p(\vec{t}) \) has a solution \( \text{E}u(\vec{g} = \vec{s}) \ast \text{NO} \). The goal \( \text{once}(p)(\vec{t}) \) fails if \( p(\vec{t}) \) does. It has the solution \( \text{E}u(\vec{g} = \vec{s}) \) if \( p(\vec{t}) \) has the solution \( \text{E}u(\vec{g} = \vec{s}) \ast \text{NO} \). If furthermore \( q_1(\vec{t})(\vec{g} / \vec{s}) \) has a solution \( \text{E}\tilde{u}^{(i)}(\tilde{u} = \tilde{r}^{(i)}) \ast \cdots \ast \text{E}\tilde{u}^{(n)}(\tilde{u} = \tilde{r}^{(n)}) \ast \text{NO} \) then \( \text{if \_then\_else}(p, q_1, q_2)(\vec{t}) \) has the solution \( \text{E}\tilde{u}^{(i)}(\tilde{u} = \tilde{s}^{(i)} / \tilde{u}) \ast \cdots \ast \text{E}\tilde{u}^{(n)}(\tilde{u} = \tilde{s}^{(n)} / \tilde{u}) \ast \text{NO} \). If \( p(\vec{t}) \) fails then the solutions for \( q_1(\vec{t}) \) are also solutions for \( \text{if \_then\_else}(p, q_1, q_2)(\vec{t}) \). If \( p(\vec{t}) \) has only the solution \( \text{NO} \), then the same holds for \( \text{fail}_{\text{if}}(p)(\vec{t}) \), \( \text{once}(p)(\vec{t}) \) and \( \text{if \_then\_else}(p, q_1, q_2)(\vec{t}) \).

We associate a list function with every predicate symbol \( p \):

**Definition 2.20**

Let \( p \) be a predicate symbol in \( \text{PRED}_{\alpha, r} \) where \( \alpha_j \) are different from \( \iota \). Then \( \mathcal{L}(p) :\equiv \lambda \vec{X} \lambda \vec{x}. \langle y \mid p(\vec{x}, \vec{y}) \odot y = (\vec{x}) \rangle \).

Now predicates can be used to refer to the result of the evaluation of \( p(\vec{T}, \vec{t}) \) by applying them to \( \mathcal{L}(p)(\vec{T}, \vec{t}) \). In the examples below we represent natural numbers \( n \) by \( \text{\'r} \text{n}^{\gamma} :\equiv \text{s}^{\text{n}}(0) \).

**Example 2.21**

Consider the following predicate:

\[
\begin{align*}
\text{repeat2}(F, X, Y, z) :&\equiv \text{E}u \text{E}w \text{E}w((u, v, w) \approx F(\text{fst}(X))(\text{fst}(Y))) \odot \\
&((\text{\'r}0^{\gamma}, \text{\'r}1^{\gamma}) = (v, w) \odot (z = u \ast \text{repeat2}(F, X, \text{rest}(Y))) \ast ) \\
&((\text{\'r}1^{\gamma}, \text{\'r}0^{\gamma}) = (v, w) \odot (z = u \ast \text{repeat2}(F, \text{rest}(X), Y)) \ast ) \\
&((\text{\'r}1^{\gamma}, \text{\'r}1^{\gamma}) = (v, w) \odot (z = u \ast \text{repeat2}(F, \text{rest}(X), \text{rest}(Y)))) \ast ).
\end{align*}
\]

It can be used to apply a function repeatedly to the elements of lists: it is assumed that the application of the function produces a triple. The first component is used as output. The remaining components tell whether the first elements should be reconsidered on repetition.

It is not possible to reconsider both first elements. Now specialize with an extended minimum function: let \( f \) be a term of type \( \iota \rightarrow \iota \rightarrow \iota \) so that \( f t_1 t_2 \Downarrow (\text{\'r} \text{min}(n_1, n_2)^{\gamma}, c_{\downarrow}(n_1, \text{min}(n_1, n_2)), c_{\downarrow}(n_2, \text{min}(n_1, n_2))) \) if \( t_1 \Downarrow \text{\'r} n_1^{\gamma} \) and \( t_2 \Downarrow \text{\'r} n_2^{\gamma} \) for all \( n_1, n_2 \in \mathbb{N} \) where \( c_{\downarrow} \) stands for the characteristic function for \( \neq \) which yields 0 if the arguments are not equal, 1 otherwise. Consider the predicate

\[
\text{merge2}(X, Y, z) :\equiv \text{repeat2}(f, X, Y, z).
\]

If the terms \( L_1, L_2 \) represent ascending infinite lists of numbers, then \( \text{merge2}(L_1, L_2, z) \) merges them, i.e. produces the union of elements, sorted and without repetitions. If \( p(x) \), \( q(x) \) generate ascending infinite lists of numerals, then \( \text{merge2}(\mathcal{L}(p)(x), \mathcal{L}(q)(x), z) \) generates the merged list.
Let repeat3, merge3 be the ternary analogue to repeat2, merge2 above, \( g_i : t \rightarrow t \) so that \( g_i(n) \downarrow g_i \cdot n^3 \) for \( n \in \mathbb{N} \), let the predicate map be defined by \( \text{map}(f, L, z) : \neg z \approx f(\text{fst}(L)) \cdot \text{map}(f, \text{rest}(L), z) \), and \( l_i, (\text{ham}) \) denote the expression \( \langle z | \text{map}(g_i, \langle z | \text{ham}(z)), z) \rangle \) for \( i = 2, 3, 5 \). Then the predicate

\[
\text{ham}(z) : \neg z \approx 1 \cdot \text{merge3}(l_2(\text{ham}), l_3(\text{ham}), l_5(\text{ham}), z)
\]

e numerates the Hamming numbers, i.e. the ascending stream of numbers \( 2^n \cdot 3^m \cdot 5^k \).

Predicates which have list arguments can be used to intertwine the results of goals. This improves the definability of control which is expressible in terms of results. An approach leading beyond this is presented in [8] where a means to refer to a list of alternatives in the reduction process is provided, making it possible to determine in what manner these alternatives should be processed. The power to define control in our setting is limited by the fact that we do not have access to the intermediate steps in the computation. But this is also its advantage: the value of a lazy list comprehension expression depends only on the outcome of the evaluation of the corresponding goal but not on the computation sequence itself. Therefore, for semantic description as well as application, it is not necessary to study or even fix operational details.

The next example for list comprehensions is a way to define negation which is sound w.r.t. a negation-as-failure interpretation [1] in contrast to fail if. The easiest way to achieve this is a restriction of fail if to ground arguments. An alternative negation is presented below which, similar to IC-Prologs [3] not, is based on a version of negation-as-failure failing non-ground negative literals if the corresponding positive literal returns a renaming. The definition of not below is a sound but incomplete depth-first variant of this with fixed selection. The goal \( \text{not}(p)(\bar{t}) \) has a correct solution \( 1 \) if \( p(\bar{t}) \) fails. A negated goal fails if the unnegated atom has a solution \( \bigvee_{i=1}^{n} S_i \cdot 0 \) where some disjunct is \( 1 \). A predicate is defined for searching for this answer.

**Example 2.23**

Let \( P \) be a program that contains the predicate definitions:

\[
\begin{align*}
\text{cmp1}(L, x) & : \Rightarrow \exists z \approx \text{fst}(L) \cdot x \equiv z \cdot \text{cmp1}(\text{rest}(L), x) \\
\text{cmp}(L, x, y) & : \Rightarrow \text{no} \approx \text{fst}(L) \cdot y \equiv 0 \cdot \text{cmp1}(L, x) \cdot y \equiv 1 \cdot 0
\end{align*}
\]

and let \( \text{not}(p)(x) \equiv 0 \cdot \text{fst}(y | \text{cmp}(L(p); x, x, y)) \).

### 3 Axiomatic semantics

#### 3.1 Calculus \( K_0 \)

The kernel of the declarative semantics is a substructural calculus. As in linear logic [7], we do not have weakening and contraction rules. All connectives used here are multiplicative and non-commutative. In contrast to linear logic the multiplicatives are not dual to each other. Furthermore, the quantifier is not standard. Last not least, we augment the pure logic by special axioms. Thus, the relation \( \otimes/\ast \) is not the same as for the multiplicatives in linear logic. The calculus presented below is a fragment of the calculus used in [5, 6] to define a declarative semantics for pure prolog with negation. We take the essence necessary for definite programs and augment the language by further built-ins and lazy list comprehension. There
is no universal quantification but it is understood that whenever we formulate an axiom we
want to accept every substitution instance of it as an axiom. There is no implication but we
consider a sequent calculus thus having some kind of 'top-level' implication. If \( \Gamma \) and \( \Delta \) are
finite lists of formulas then \( \Gamma \Rightarrow \Delta \) is a sequent. The notation \( A \iff B \) is used to refer to
the sequents \( A \Rightarrow B \) and \( B \Rightarrow A \). An inference rule with premises \( S_1, \ldots, S_n \) \( (n \geq 0) \) and
conclusion \( S \) is written \( S_1, \ldots, S_n \vdash S \).

**Definition 3.1**

The axioms and rules of the calculus \( K_0 \) are as follows:

1. propositional connectives:
   \[
   \vdash G \Rightarrow G \text{(ax)}, \text{and } (\Gamma \Rightarrow G), (G \Rightarrow \Delta) \vdash \Gamma \Rightarrow \Delta \text{(cut)}
   \]
   \[
   \vdash 0, \Gamma \Rightarrow \Delta
   \]
   \[
   \Gamma, G_1, G_2, \Pi \Rightarrow \Delta \vdash \Gamma, G_1 \land G_2, \Pi \Rightarrow \Delta
   \]
   \[
   \Gamma \Rightarrow G_2, (\Pi \Rightarrow G_2) \vdash \Gamma, \Pi \Rightarrow G_1 \land G_2
   \]
   \[
   \Gamma \Rightarrow \Delta, (G_2 \Rightarrow \Sigma) \vdash G_1 \lor G_2 \Rightarrow \Delta, \Sigma
   \]

2. quantification:
   \[
   \Gamma, G_1, \Pi \Rightarrow \Delta \vdash \Gamma, \text{Ex}G_1, \Pi \Rightarrow \Delta !x(x)
   \]
   \[
   \Gamma, G_1, \Pi \Rightarrow G_2 \vdash \Gamma, \text{Ex}G_1, \Pi \Rightarrow \text{Ex}G_2 !x(x)
   \]
   \[
   \Gamma, \Pi \Rightarrow G_2 \vdash \Gamma, \Pi \Rightarrow \text{Ex}G_2 !x(x)
   \]

3. mixed:
   \[
   \vdash \text{Ex}G_1 \lor \text{Ex}G_2 \Rightarrow \text{Ex}(G_1 \lor G_2)
   \]
   \[
   \vdash \text{Ex}(G_1 \land G_2) \Rightarrow \text{Ex}G_1 \lor \text{Ex}G_2 !x(x)
   \]
   \[
   \vdash \text{Ex}(G_1 \land G_2) \Rightarrow G_1 \land \text{Ex}G_2 !x(x)
   \]

Here \( G_1, G_2, H \) are arbitrary formulas, \( \Gamma, \Pi, \Delta, \Sigma \) finite lists of formulas, and \(!x(x) \) stands for
the eigenvariable condition: \( x \) must not occur free in the conclusion.

Consider as an example a formula of the form \( D \equiv ((A \times B) \times F \times F) \times C \) where \( F \) stands for falsum. Interpreting \( \times, \lor \) as classical conjunction and disjunction, \( D \equiv F \) but this involves
laws which are not available in \( K_0 \). Without further restrictions for \( A, B, C \) we cannot even derive \( F \Rightarrow D \) or \( \forall x(x) \Rightarrow F \Rightarrow D \). We are interested in the latter, as this sequent formalizes
“\( \neg D \) fails.” If \( \text{fv}(D) = (x) \). We present a derivation of \( \forall x(x), F \Rightarrow ((\neg x \land B) \land F \land F) \land C \) using non-logical axioms \( (\forall x(x) \land F \Rightarrow A \land F), (\forall x(x) \land F \Rightarrow B \land F), \) and \( (\forall x(x), F \Rightarrow \forall \langle x \rangle \land F \land F \rangle \) in Figure 1.

All axioms and rules are correct w.r.t. a classical interpretation. Here are some examples
for classical laws which are not derivable if \( \land, \lor, \exists \) are replaced by \( \times, \lor, E \) respectively:
\[
A \land B \Rightarrow B, B \Rightarrow A \lor B, F \Rightarrow A \land F, A \lor B \Rightarrow B \lor A, (A \land B) \lor (A \land C) \Rightarrow A \land (B \lor C),
\]
\[
A \{t/x\} \Rightarrow \exists x A.
\]

The following relation of \( \triangleright_0 \) and derivability in \( K_0 \) is immediate: consider a program \( P \)
and \( T := \{ \text{Def}_{p}^{P}(i) \Rightarrow p(i) \mid p \in \text{Pred}, i \text{ terms} \} \). If we use \( T \vdash \Gamma \Rightarrow \Delta \) for derivability in \( K_0 \) accepting elements of \( T \) as further axioms and assume that \( G, H \) contain no built-ins, then obviously \( G \triangleright_0 H \) implies \( T \vdash H \Rightarrow G \).
3.2 Additional axioms

Now further axioms and rules are introduced. When we accept $\nabla(\bar{x})$ for $\bar{x}$ of arbitrary length, this is meant to include the absence of the $\nabla$ expression. The notation $G \iff H$ is used for the two axioms $G \Rightarrow H$ and $H \Rightarrow G$. Furthermore $G \iff \nabla(\bar{x})$ $H$ stands for $\nabla(\bar{x}), G \Rightarrow H$ and $\nabla(\bar{x}), H \Rightarrow G$.

First we consider the built-ins and add further axioms for equality. Then we present axioms for lazy lists and the lazy list comprehension rule (llcr). Finally axioms and rules for manipulating the $\nabla$ expressions are given.

**Definition 3.2**

The axioms and rules of the calculus $K_1$ are those of $K_0$ plus:

- **(epsilon)** $\vdash t = eu.s \iff Eu(t = s)$ if $u$ not free in $t$
- **(unification)** $\vdash$ unify($t$, $s$) $\iff \nabla(\bar{x})$ $t = s$ if FV($t$, $s$) $\subset \{\bar{x}\}$
- **(normalization)** $\Gamma(t) \Rightarrow \Delta(t) \vdash \Gamma(s) \Rightarrow \Delta(s)$ if n(f($t$)) $\equiv$ n(f($s$))
- **(=)** $\vdash \nabla(\bar{x})$, $\text{solve}(t = s) \Rightarrow t = s$, if FV($t$, $s$) $\subset \{\bar{x}\}$

- **(≈)** For etcn terms $\bar{l}, t, t'$ and arbitrary $\bar{s}, s, \bar{f}, r$:
  $\vdash r \approx \bar{s} \approx s \approx r$
  $\vdash \nabla(\bar{f}), t, t' \approx s, \Pi \Rightarrow \Delta \vdash \nabla(\bar{f}), \Gamma, t \Rightarrow t', \Pi \Rightarrow \Delta$ if FV($t$, $t'$) $\subset$ FV($\bar{f}$)
  $\nabla(\bar{f}, u), z \approx t \Rightarrow z \approx s \vdash \nabla(\bar{f}), r \approx \alpha u \Rightarrow r \approx \alpha u$ if FV($t$) $\subset$ FV($\bar{f}, u$) $\not\subset z$
  $\nabla(\bar{f}, z \approx t_i \Rightarrow z \approx s_i)_{1 \leq i \leq n} \vdash \nabla(\bar{f}), r \approx c(\bar{f}) \Rightarrow r \approx c(\bar{s})$ if FV($\bar{f}$) $\subset$ FV($\bar{f}$), and FV($\bar{s}, \bar{f}$) $\not\subset z$.

- **(≡)** For ct terms $\bar{l}, t, \bar{s}, s$, FV($\bar{l}, t, \bar{s}, s$) $\subset$ $\{\bar{x}\}$, and arbitrary $r, r'$:
  $\vdash 1 \iff r = r$
  $\vdash r = r' \Rightarrow r' = r$
  $\vdash \bar{l} = \bar{s} \iff \nabla(\bar{x}) c(\bar{l}) = c(\bar{s})$
Let \( \vdash \nabla(x), t = s, G\{t/x\} \Rightarrow t = s \circ G\{s/x\} \)
\( \vdash \tau = r^* \circ (G \ast H) \Leftrightarrow (r = r^* \circ G) \ast (r = r^* \circ H) \)

(lazy list comprehension axioms): For closed ect terms s:
\( \vdash z \approx \no \Rightarrow z \approx (y \mid F)(n) \)
\( \vdash z \approx s \Rightarrow z \approx (y \mid y = s \ast G)(1) \)
\( \vdash z \approx (y \mid G)(n) \Rightarrow z \approx (y \mid y = s \ast G)(n + 1) \)
\( \vdash \nabla(x), t \Rightarrow (y \mid E\bar{x}G(n)) \Rightarrow t \Rightarrow (y \mid G)(n) \)

\( \nabla(y), G \Rightarrow H \vdash t \Rightarrow \langle y \mid G(n) \Rightarrow t \Rightarrow \langle y \mid H(n) \rangle \)

(\( \nabla \) weakening)
\( \Gamma, \Pi \Rightarrow \Sigma \vdash \Gamma, \nabla(t), \Pi \Rightarrow \Sigma \)
\( \Gamma, \nabla(t), \Pi \Rightarrow \Sigma \vdash \Gamma, \nabla(t, \bar{s}), \Pi \Rightarrow \Sigma \)

(\( \nabla \) contraction)
\( \Gamma, \nabla(t), t, \Pi \Rightarrow \Sigma \vdash \Gamma, \nabla(t, \bar{s}), \Pi \Rightarrow \Sigma \)

(\( \nabla \) permutation)
\( \vdash \nabla(t) \Rightarrow \nabla(\pi \bar{t}) \) for every permutation \( \pi \)

(\( \nabla \) $\otimes$)
\( \vdash \nabla(x, y), y = t \Rightarrow y = t \circ \nabla(x) \)

(\( \nabla \) $\otimes$)
\( \vdash \nabla(x), E\bar{g}G \Rightarrow E\bar{g}(\nabla(x, \bar{g}) \circ G) \) (if \( \{\bar{g}\} \cap \{\bar{x}\} = \emptyset \)

(\( \nabla \) distributivity)
\( \vdash \nabla(\bar{x}) \circ (G \ast H) \Rightarrow (\nabla(\bar{x}) \circ G) \ast (\nabla(\bar{x}) \circ H) \).

Let us first have a look at the axioms for \( \approx \). Consider an ect term \( t \equiv \epsilon u.t_o\{\bar{s}/\bar{y}\} \) where \( t_o \) is in ct, \( \bar{s} = s_1, \ldots, s_n \) and the terms \( s_i \) have the form \( (y_i \mid G_i)(n_i) \). If \( s_i \downarrow r_i \) for ect terms \( r_i \) then \( t \downarrow E\bar{n}, t_o\{\bar{r}/\bar{g}\} \). If \( r_i \) are the \( n_i \)th correct values for \( y_i \) in \( G_i \) as given by Definition 3.6 then, using the third and fourth \( \approx \) rule, we can derive \( \nabla(x), z \approx \epsilon u.t_o\{\bar{r}/\bar{g}\} \Rightarrow z \approx \epsilon \bar{u}, t_o\{\bar{s}/\bar{y}\} \) where \( z \) is fresh and \( \bar{s} \) are the free variables in \( t \). Note also that we do not have the usual equality axioms for \( \approx \): it could not be justified to add reflexivity as \( t \approx t \) may loop while \( \Gamma \) succeeds.

The expression \( \nabla(\bar{x}) \) serves to restrict a judgement to the case where \( \bar{x} \) are distinct variables. The main interest is in sequents of the form \( \nabla(\bar{x}), \Gamma \Rightarrow \Delta \). The calculus \( K_t \) offers axioms and rules to manipulate the \( \nabla \) part. As an abbreviation for \( \nabla(\bar{x}), \Gamma \Rightarrow \Delta \) we introduced \( \Gamma \Rightarrow \nabla(\bar{x}), \Delta \) and this is quite convenient but should be used with caution: for example \( \Gamma \Rightarrow \nabla(x), A \), \( \Pi \Rightarrow \nabla(x), B \vdash (\Gamma, \Pi \Rightarrow \nabla(x), A \otimes B) \) is not an acceptable inference. To see this, consider the sequents \( \nabla(x), x = d \Rightarrow x = d \) and \( \nabla(x), c \approx \langle y \mid Ex(x = c)(1) \Rightarrow c \approx \langle y \mid x = c\rangle(1) \). Both are obviously derivable but \( \nabla(x) \otimes x = d \otimes c \approx \langle y \mid Ex(x = c)\rangle(1) \) succeeds and \( x = d \otimes c \approx \langle y \mid x = c\rangle(1) \) fails.

The connective \( \otimes \) is in general not commutative but, using the fact that we have postulated to have a binary constructor, we can derive:

\( \nabla(x), t = s \circ t' = s' \Rightarrow t' = s' \circ t = s \)

for \( t, t', s, s' \in \text{ct}, FV(t', t, s, s') \subset \{\bar{x}\} \). If \( t \) is in ect and \( x \) not free in \( t \) then \( \vdash Ex(x = t) \Leftrightarrow ex(x = t) \) by (e). As unify(\( ex(x, t) \), \( 1 \)) we can deduce \( \nabla(\bar{g}) \Rightarrow Ex(x = t) \) and \( \nabla(\bar{g}), Ex(x = t) \Rightarrow 1 \) if \( \{\bar{g}\} \subset FV(t) \).

The sequent \( \Gamma \Rightarrow \Gamma \otimes \Delta \) is an axiom but \( \nabla(\bar{x}), \Gamma \Rightarrow \Delta \) is not in general derivable. It is derivable, however, for \( \Delta \equiv \bar{\bar{t}} = \bar{s} \) where \( \bar{\bar{t}}, \bar{s} \) are ct terms, \( FV(\bar{\bar{t}}, \bar{s}) \subset \{\bar{x}\} \).

Example 3.3

Let \( t \in \{s(s(0)), s(s(s(0)))\} \). The sequents \( (\{\} \Rightarrow F) \) and \( \nabla(x), \Gamma \Rightarrow (x, ()) = (t, ()) \) are instances of the (unification) axioms, \( \nabla(x), (x, ()) \Rightarrow t = x \otimes () = ((()) \) is an (equality) axiom. Combining these and applying (\( \nabla \) contraction) we obtain a derivation of \( \nabla(x), F \Rightarrow x = t \otimes F \). Now, using (\( \nabla \) distributivity), we can proceed as in
Let \( \nabla(x), p(x) \quad \text{and} \quad q(x, y, z) \).

The sequent \( \nabla(x), x = c \quad \text{and} \quad \nabla(x), p(c) \quad \text{and} \quad q(x, x, x) \) is an axiom but \( \nabla(x), p(x) \quad \text{and} \quad q(x, x, x) \) is not derivable. As a justification for the latter, consider the program
\[
(p(x) : x = d \circ 0) \quad \text{where} \quad c \neq d.
\]
Here the goal \( p(c) \quad \text{and} \quad q(x, x, x) \) fails while \( p(x) \quad \text{and} \quad q(x, x, x) \) loops due to the fixed literal selection.

In Corollary 3.4 we list some useful derivable rules. Let \( \mathbf{n} \) stand for \( 1 \ldots n \) with \( n \) copies of \( \mathbf{1} \).

**Corollary 3.4**

Let \( G \) be a goal, \( \text{FV}(G) \subset \{ \bar{x}, y \}, r \quad \text{and} \quad S \equiv *_{i=1}^{n} S_i \) be a solved goal where
\[
S_n \equiv E\bar{u}(\bar{x} = \bar{t} \quad \text{and} \quad y = s) \quad \text{and} \quad \text{FV}(S) \subset \{ \bar{x}, y \}.
\]
The following rules are derivable:

1. (llcr') \( \nabla(\bar{x}, y), G \quad \vdash \quad H \quad \vdash \quad \nabla(\bar{x}), t \equiv \mu_n y \quad E\bar{u}G \quad \vdash \quad t \equiv \mu_n y \quad H \).
2. \( \nabla(\bar{x}, y), S(*0) \quad \vdash \quad G \quad \vdash \quad n(*0) \quad \Rightarrow \quad E\bar{u}EyG \).
3. \( \nabla(\bar{x}, y), S(*0) \quad \vdash \quad G \quad \vdash \quad \nabla(\bar{x}), r \equiv \epsilon(s) \quad \Rightarrow \quad r \equiv \mu_n y \quad G \)
\[
\nabla(\bar{x}), S \quad \vdash \quad G \quad \vdash \quad \nabla(\bar{x}), r \equiv NO \quad \Rightarrow \quad r \equiv \mu_n y \quad G \quad \text{for} \quad m > n \quad \text{and}
\]
\[
\nabla(\bar{x}), F \quad \vdash \quad G \quad \vdash \quad \nabla(\bar{x}), r \equiv NO \quad \Rightarrow \quad r \equiv \mu_m y \quad G \quad \text{for} \quad m \geq 1.
\]
4. For \( G \equiv t_1 \equiv t_2 \) and ect terms \( s_1, s_2 \):
\[
(\nabla(\bar{x}, y), z \equiv s_1 \quad \Rightarrow \quad z \equiv s_1 \quad t_{i=1,2}, (\nabla(\bar{x}, y), S \quad \Rightarrow \quad s_1 = s_2) \quad \vdash \quad \nabla(\bar{x}, y), S \quad \Rightarrow \quad t_1 \equiv t_2.
\]

**Definition 3.5**

If \( \Gamma \vdash \Delta \) is derivable in \( K_1 \), we write \( \vdash \Gamma \Rightarrow \Delta \). Let \( P \) be a program. We write \( P \vdash \Gamma \Rightarrow \Delta \) if \( \Gamma \Rightarrow \Delta \) is derivable in \( K_1 \) plus axioms
\[
\text{Def}^p_\Gamma(\bar{t}) \Rightarrow p(\bar{t}) \quad \text{where} \quad (p(\bar{x}) : \neg \text{Def}^p_\Gamma(\bar{x}) ) \in P.
\]

**Definition 3.6**

Let \( P \) be a program, \( G \) an executable goal for \( P \), \( \text{FV}(G) \subset \{ \bar{x} \} \), \( \bar{x} \), a solved goal \( S \in \{ \bar{x} \} \), is a correct solution for \( P, G \) if \( P \vdash \nabla(\bar{x}), S \Rightarrow G \). Let \( t \) have the form \( \text{NO} \) or \( \epsilon(t_0) \) for \( t_0 \in \text{ct} \). Then \( t \) is called \( n \)th correct value for \( y \) in \( G \) w.r.t. \( P \). If \( P \vdash \nabla(\bar{x}), s \equiv t \Rightarrow s \equiv (y \mid G)'(n) \) for all terms \( s \) where \( \bar{x}' \) is obtained from \( \bar{x} \) by deleting \( y \).

**Example 3.7**

Let \( P \) be a program containing a definition for \( p \) and \( \text{fail}_{\text{iff}}(p(\bar{t})) := \text{NO} \Rightarrow \mu_1 z \cdot p(\bar{t}) \) for ct terms \( \bar{t}, \text{FV}(\bar{t}) = \{ \bar{u} \} \). Using Corollary 3.4 we obtain:
\[
\text{if} \quad p(\bar{t}) \quad \text{has a correct solution} \quad E\bar{u}(\bar{u} = \bar{t}) \quad \text{then} \quad P \vdash \nabla(\bar{u}), \text{NO} \Rightarrow Eu.w \Rightarrow \text{NO} \Rightarrow \mu_1 z \cdot p(\bar{t}), \text{hence} \quad P \vdash \nabla(\bar{u}), F \Rightarrow \text{fail}_{\text{iff}}(p(\bar{t})).
\]
If \( p(\bar{t}) \) has the correct solution \( F \) then \( P \vdash \nabla(\bar{u}), \text{NO} \Rightarrow \text{NO} \Rightarrow \text{NO} \Rightarrow \mu_1 z \cdot p(\bar{t}), \text{hence} \quad P \vdash \nabla(\bar{u}) \Rightarrow \text{fail}_{\text{iff}}(p(\bar{t})).
\]

Obviously \( t \) is the \( n \)th correct value for \( y \) in \( G \) if \( P \vdash \nabla(\bar{x}), z \equiv t \Rightarrow z \equiv (y \mid G)'(n) \) for fresh \( z \). Furthermore \( P \vdash \nabla(\bar{x}) \Rightarrow t \equiv (y \mid G)'(n) \) if \( t \) is the \( n \)th value for \( y \) in \( G \). The converse of this is in general not true: \( P \vdash \nabla(\bar{x}) \Rightarrow \exists x. x \equiv (y \mid G)'(n) \) whenever there is an ect term which is an \( n \)th value.

### 3.3 Soundness and completeness

The first aim is the soundness of the evaluation as stated in the next lemma.

**Lemma 3.8**

Let \( P \) be a program, \( G \) an executable goal for \( P \), \( \text{FV}(G) = \{ \bar{x} \} \). If a solved goal \( S, \text{FV}(S) \subset \{ \bar{x} \} \), is a computed solution for \( P,G \) then it is a correct solution for \( P,G \). If \( t \downarrow t' \) and
FV(t, t′) ⊂ {x} then P ⊢ Γ(x), z ≈ t′ ⇒ z ≈ t. If a term t is the nth computed value for y in G w.r.t. P then it is also an nth correct value for y in G w.r.t. P.

The indefinite article is used, as the uniqueness of the nth correct value is not immediate. The uniqueness is, however, a consequence of Theorem 3.15.

**Proof.** By induction on the number of computation steps we prove:

1. Let 〈x〉 ⊃ FV(G, H₁, . . . , Hₙ, ̃t, ̃x₁, . . . , ̃xₙ) and ̃y be fresh variables.
   If (G, ̃t) ⊢* (H₁, ̃x₁), . . . , (Hₙ, ̃xₙ),
   then P ⊢ Γ(̃y), ̃x₁, . . . , ̃xₙ = ̃x₁ ⊗ H₁ ⇒ Γ(̃y) = ̃t ⊗ G.

2. Let FV(G, ̃x) ⊂ {x, y}.
   If (G, y) ⊢* (1, s₁), . . . , (1, sₙ), A
   then P ⊢ Γ(̃x), z ≈ e(s₁) ⇒ z ≈ (γ | G)(n).
   If (G, y) ⊢* (1, s₁), . . . , (1, sₙ) for some m < n
   then P ⊢ Γ(̃x), z ≈ NO ⇒ z ≈ (γ | G)(n).

3. If t ⊢ t′ and FV(t, t′) ⊂ {x} then P ⊢ Γ(x), z ≈ t′ ⇒ z ≈ t.

1. It is sufficient to consider a single step. The case ⊢* is easy. Otherwise it is sufficient to show P ⊢ Γ(x), t ≈ s′ ⇒ t ≈ s if t ⊢ t′ and s ⊢ s′. By IH3 we have P ⊢ Γ(x), t ≈ s′ ⇒ t ≈ s and P ⊢ Γ(x), s ≈ t′ ⇒ s′ ≈ t. Using the symmetry of ≈ and contraction of Γ(x) we obtain P ⊢ Γ(x), t ≈ s′ ⇒ t ≈ s. As t′, s′ are ecm terms, we can deduce P ⊢ Γ(x), t′ ≈ t ≈ s.

2. Using 1. and the axiom for epsilon, we have P ⊢ Γ(y), e(s₁) ≈ . . . e(sₙ) = e(sₙ)(sₙ) ⇒ ΓE(G). Applying (ilcr) and axioms for lazy lists we can infer the desired result.

3. Assume t ⊢ t′ and FV(t, t′) ⊂ {x}. Using the normalization rule we can assume that t is normal. If t ∈ ecm there is nothing left to show. Otherwise, using IH2, there is a ct term t₀ and terms ̃s = s₁, . . . , sₙ of the form s_i ≡ (γ_i | G_i)(n_i) and ecm terms ̃t = r₁, . . . , r_m so that t ≡ eₜₕₖ(̃t/̃y), ̃t ∩ {x} = φ, and t′ = nf(eₜₕₖ(̃t/̃y)), and P ⊢ Γ(x, ̃t), z ≈ r₁ ⇒ z ≈ sᵢ for all 1 ≤ i ≤ m. Using the axioms (≈), this implies P ⊢ Γ(x), z ≈ eₜₕₖ(̃t/̃y) ⇒ z ≈ t. An application of the normalization rule completes the proof.

**Example 3.9**
Let G₁ = ((x = 3 * x = 2) ⊗ F ∗ F) ⊗ q(x, x, x) as in Example 3.3. The derivation of Γ(x), F ⇒ G₁ that we obtain from the soundness proof is closer to the evaluation of G₁.

Build derivations of

(1) \[ \nabla(y), F \Rightarrow E(x) = x \oplus F \ominus q(x, x, x) \]
(2) \[ \nabla(y), F \Rightarrow y = 3 \ominus F \ominus q(3 \ominus 2, 3, 3) \]
(3) \[ \nabla(y), F \Rightarrow y = 2 \ominus F \ominus q(2 \ominus 2, 2, 2) \]

Using (2), (3) and equality axioms, construct derivations of

(4) \[ \nabla(y), F \Rightarrow E(x) = x \oplus x = 3 \ominus F \ominus q(x, x, x) \]
(5) \[ \nabla(y), F \Rightarrow E(x) = x \oplus x = 2 \ominus F \ominus q(x, x, x) \]

Combine first (4), (5) to a derivation of

\[ \nabla(y), F \Rightarrow E(x) = x \oplus (x = 2 \ominus 3 = 2) \ominus F \ominus q(x, x, x) \]

then this latter sequent and (1) to a derivation of \( \nabla(x), F \Rightarrow G₁ \), using distributivity.
Now consider \( G_2 \equiv y \approx f(x) \). The reduction sequence \((y \approx f(x), y) \triangleright (y = f(x), y) \triangleright (1, f(x))\) can be transformed into a derivation of \( \nabla(z), \text{Ex}\{z = f(x)\} \Rightarrow \text{Ex}\{z = y \approx f(x)\} \), hence \( \vdash \nabla(z), z = ex.\{f(x)\} \Rightarrow \text{Ex}\{z \approx f(x)\} \). Following the construction in 2 we obtain first

\[
y \approx (z \mid z = ex.\{f(x)\})(1) \Rightarrow y \approx (z \mid \text{Ex}\{z \approx f(x)\})(1)
\]

by (Icr). Now we use the lazy list axioms

1. \( y \approx ex.\{f(x)\} \Rightarrow y \approx (z \mid z = ex.\{f(x)\})(1) \)
2. \( \nabla(x, y) \approx (z \mid \text{Ex}\{z \approx f(x)\})(1) \Rightarrow y \approx (z \mid z \approx f(x))(1) \)

To obtain a derivation of \( \nabla(x), y \approx ex.\{f(x)\} \Rightarrow y \approx (z \mid z \approx f(x))(1) \). Furthermore we can follow 3 to obtain first

\[
\nabla(x, u), y \approx g(ex.\{f(x)\}, u) \Rightarrow y \approx g((z \mid z \approx f(x))(1), u),
\]

and then

\[
\nabla(x), y \approx eu.\{g(ex.\{f(x)\}, u) \Rightarrow y \approx eu.\{g((z \mid z \approx f(x))(1), u) \}
\]

by (\(\approx\)). Using normalization twice, we obtain a derivation of:

\[
\nabla(x), y \approx eu.\{ex.\{f(x)\}, u) \Rightarrow y \approx eu.\{g((z \mid z \approx f(x))(1), eu.\{u) \}
\]

The next aim is to show completeness of the evaluation w.r.t. the axiomatic semantics, i.e. correct values and solutions are also computed. To this end we define a preorder \( \sqsubseteq_p \) on executable goals based on the operational semantics and prove

\[
P \vdash \Gamma \Rightarrow \Delta \text{ implies } \otimes \Gamma \sqsubseteq_p \star \Delta \quad (\ast)
\]

This can be read as proving the axioms and rules correct with respect to an interpretation of ‘\(\Rightarrow\)’ as ‘\(\sqsubseteq_p\)’.

**Definition 3.10**

Let \( P \) be a program. The preorder \( \sqsubseteq_p \) on executable goals is defined by \( G_1 \sqsubseteq_p G_2 \) iff \( \llbracket G_1 \theta \rrbracket_e \sqsubseteq \llbracket G_2 \theta \rrbracket_e \) for all \( \iota \in \text{ct} \) and substitutions \( \theta \) which replace \( \iota \) variables by ct terms.

The subscript \( P \) is usually left out. As a consequence of soundness, see Lemma 3.8, the reverse implication of (\(\ast\)) holds for \( \Gamma = \nabla(\bar{x}), S \) where \( \bar{x} \) contains the free variables in \( \Delta \) and \( S \) is solved, \( FV(S) \subset \{\bar{x}\} \). It is not true for arbitrary \( \Gamma, \Delta \), and equivalence could not be achieved for any extension for which derivability is semi-decidable.

To prove \( (\ast) \) we consider all axioms and rules. For some cases we need some properties of the operational semantics which are stated in Lemmas 3.11, 3.12, and 3.13.

**Lemma 3.11**

Let \( P \) be a program, \( G, G_1, G_2, \Gamma, \Pi \) executable goals and \( t \in \text{ct} \). Then \( \llbracket \otimes (\Gamma, G_1 \otimes G_2, \Pi) \rrbracket_t \llbracket \otimes (\Gamma, G_1, G_2, \Pi) \rrbracket_t = \llbracket \otimes (\Gamma, G, \Pi) \rrbracket_t \) if \( x \) not free in \( \Gamma, \Pi, t \), and \( \llbracket \otimes (\Gamma, 1, \Pi) \rrbracket_t = \llbracket \otimes (\Gamma, \Pi) \rrbracket_t \).

**Proof.** Consider the first equation, part \( \sqsubseteq \). By induction on the number of computation steps we prove the following: if \( \otimes (\Gamma, G_1 \otimes G_2, \Pi), t \triangleright A \) for some \( A \equiv (1, s_1), \ldots, (1, s_n), B \) then there is \( A' \) so that \( \otimes (\Gamma, G_1, G_2, \Pi), t \triangleright A' \) and there are renamings \( \delta_i \) and some \( B' \)
so that $A' = (1, s_1 \delta_1), \ldots, (1, s_n \delta_n), B'$ and $B'$ is empty if $B$ is. To see this, consider the first
reduction step. If $\Gamma$ is empty and also $\Pi$ is empty then there is nothing to show. If $\Gamma$ is empty
and $\Pi$ is not, then the first reduction step is $((G_1 \otimes G_2) \otimes \otimes (\Pi), t) \triangleright (\otimes (G_1, G_2, \Pi), t)$
implies the desired result. If $\Gamma \equiv F, \Gamma'$ then the first step is $((\otimes (\Gamma, G_1 \otimes G_2, \Pi), t) \triangleright (F, t)$ and also
$((\otimes (\Gamma, G_1, G_2, \Pi), t) \triangleright (\otimes (H_1, G_1 \otimes G_2, \Pi), t), (\otimes (H_2, \Gamma', G_1 \otimes G_2, \Pi), t)$
and $(\otimes (\Gamma, G_1, G_2, \Pi), t)$ reduces in two steps to $(\otimes (H_1, G_1, G_2, \Pi), t), (\otimes (H_2, G_1, G_2, \Pi), t).$
Otherwise there are $\Gamma', G_1', G_2', \Pi', t'$ so that $(\otimes (\Gamma, G_1 \otimes G_2, \Pi), t) \triangleright^+ (\otimes (\Gamma', G_1' \otimes G_2',
\Pi'), t').$ Thus an application of the induction hypothesis completes the proof. The direction $\supseteq$ and the remaining equalities
are proved similarly.

**Lemma 3.12**
If $\nf(t) = \nf(s)$ then $G\{t/x\} \subseteq G\{s/x\}.$

**Proof.** As $\nf(\theta t) = \nf(\nf(t) \theta t)$ it is sufficient to show $[G\{t/x\}]_r \subseteq [G\{s/x\}]_r$ for all ct
terms $r$ and goals $G$ if $\nf(s) = \nf(t).$ Assume $\nf(s) = \nf(t).$ By induction on the number of
computation steps involved it can be shown: If $(G\{t/x\}, r) \triangleright^* (1, r_1), \ldots, (1, r_m), A$ then $(G\{s/x\}, r) \triangleright^* (1, r_1), \ldots, (1, r_m), A'$ up to renaming for some $A'$ which is empty if $A$ is.

The next lemma can be rephrased as monotonicity of $\otimes$ w.r.t. $\subseteq$. Note that $[G_1]_x \subseteq [G_2]_x, [G_3]_x \subseteq [G_4]_x$ where $\bar{x}$ is a listing of the free variables in $G_1, G_2, G_3, G_4$
does not imply $G_1 \otimes G_2 \subseteq (x = d), G_3 \equiv (x = c \circ 0), G_4 \equiv 0.$

**Lemma 3.13**
If $G_1 \subseteq G_2$ and $G_3 \subseteq G_4$ then $G_1 \otimes G_3 \subseteq G_2 \otimes G_4.$

**Proof.** Let us consider reduction sequences. The initial steps serve to extract the leftmost
atom and are determined by the left component of a conjunction. More formally:

(I) If $G$ is a goal and $A$ the leftmost atom in $G,$ then there are lists of goals $\Gamma_0, \ldots, \Gamma_n$ so that

$$(\otimes (G, \Pi), t) \triangleright^* (\otimes (A, \Gamma_0, \Pi), t), (\otimes (\Gamma_1, \Pi), t), \ldots, (\otimes (\Gamma_n, \Pi), t)$$

up to renaming for all $\Pi, t$ where no variable bound in $G$ is free in $\Pi, t.$ (Otherwise the parts $A, \Gamma_i$ may have to be renamed.)

Results for the conjuncts can be composed to yield results for the conjunction in the following
way:

(II) If $\text{FV}(t) \supseteq \text{FV}(G, H)$ and $(G, t) \triangleright^* (1, t \theta_1), \ldots, (1, t \theta_k), A$ for some substitutions
$\theta_i$ for the variables in $t,$ and $(H \theta_k, t \theta_i) \triangleright^* (1, s_i), \ldots, (1, s_i \theta_i)$ for all $1 \leq i < k,$
and $(H \theta_k, t \theta_1) \triangleright^* (1, s_k), \ldots, (1, s_{k+1}), B,$ then $(G \otimes H, t) \triangleright^* (1, s_1 \delta_1), \ldots, (1, s_k \delta_k, \delta_k), C$ for some
renamings $\delta_{rk}$ and some $C$ which is empty if $A$ and $B$ are.

We prove this by induction on the number of computation steps involved. Consider $A,$
$\Gamma_0, \ldots, \Gamma_n$ as given by (I) where the bound variables in $G$ are chosen so that they do not
occur in $H, t,$ and distinguish cases according to the leftmost atom $A$: $A \equiv (x = s)$ and $x$ does not occur in the ct term $s:$

Then $(\otimes \Gamma_0 \{s/x\}, t \{s/x\}, (\otimes \Gamma_1, t), \ldots, (\otimes \Gamma_n, t) \triangleright^* (1, t \theta_1), \ldots, (1, t \theta_k), A$ in less steps.
We use the induction hypothesis for all parts that contribute to obtaining \((1, t\theta_1), \ldots, (1, t\theta_k)\). (We consider it also a contribution to fail and make the neighbour available.) There is \(k' (k' \geq 0)\) and \(A'\) so that \((\circ \Gamma_0 \{s/x\}, t\{s/x\}) \triangleright^* (1, t\theta_1), \ldots, (1, t\theta_k), A'\) and \(A'\) is empty if some \(\Gamma_i, i > 0\), contributes to the solution or \(A\) empty. For \(1 \leq i \leq k'\) we choose \(\sigma_i\) so that \(\theta_i \equiv \{s/x\} \sigma_i\) and \(H\theta_i \equiv H\{s/x\} \sigma_i\). The term \(t\{s/x\}\) can be extended to subsume all variables in \(\circ \Gamma_0 \{s/x\}\), hence by induction hypothesis \((\circ \Gamma_0 \{s/x\} \circ H\{s/x\}, t\{s/x\})\) reduces to \((1, s_{i1}), \ldots, (1, s_{im}), (A')\) up to renaming for some \(A'\) which is empty if \(\circ \Gamma_i\) contributes to the solution for some \(i > 0\) or \(n = 0\) and \(A, B\) are empty. Adding an initial step, we find a reduction sequence with the same result for \((x = s \circ \circ \Gamma_0 \circ H), t)\), using Lemma 3.11 also for \((\circ (x = s, \Gamma_0, H), t)\). Similar reasoning applies to the components \(\Gamma, i, i > 0\), hence \((\circ (x = s, \Gamma_0, H), t), (\circ (\Gamma_1, H), t), \ldots, (\circ (\Gamma_n, H), t)\) reduces to the sequence \((1, s_{11}), \ldots, (1, s_{km}), C\) (up to renaming) for some \(C\) which is empty if \(A\) and \(B\) are. Furthermore the \(\Gamma_i\) are chosen so that \((G \circ H, t) \triangleright^* (\circ (x = s, \Gamma_0, H), t), (\circ (\Gamma_1, H), t), \ldots, (\circ (\Gamma_n, H), t)\) and this completes the proof of (II) in this case.

The remaining cases are treated similarly. Now we turn to decomposing solutions for conjunctions:

(III) If \((\circ (\Gamma, \Pi), t) \triangleright^* (1, s_1), \ldots, (1, s_m), A\) then there are \(\theta_1, \ldots, \theta_k\) so that \((\circ (\Gamma, t) \triangleright^* (1, t\theta_1), \ldots, (1, t\theta_k), B\) and \((\circ \Pi \theta_1, t\theta_1), \ldots, (\circ \Pi \theta_k, t\theta_k) \triangleright^* (1, s_1), \ldots, (1, s_m), C\) for some \(B, C\) which are empty if \(A\) is.

This is again proved by induction on the number of computation steps. If \(\Gamma\) or \(\Pi\) is empty, then it is immediate. Now let \(\Gamma = G, \Gamma'\), and \(A\) be the leftmost atom in \(G\) and choose \(\Gamma_0, \Gamma_1, \ldots, \Gamma_n\) for \(G\) according to (I). We distinguish cases depending on \(A\). As an example consider \(A \equiv (x = s)\) for a ct term \(s\) not containing \(x\). Then \((\circ (\Gamma_0, \Gamma', \Pi) \{s/x\}, t\{s/x\})\), \((\circ (\Gamma_1, \Gamma', \Pi), t)\), \ldots, \((\circ (\Gamma_n, \Gamma', \Pi), t)\) reduces to \((1, s_1), \ldots, (1, s_m), A\) in less steps. We apply the induction hypothesis to all components that contribute to \((1, s_1), \ldots, (1, s_m)\). Let the sequence \((1, s_1), \ldots, (1, s_m), A'\) be the contribution of \((\circ (\Gamma_0, \Gamma', \Pi) \{s/x\}, t\{s/x\})\), where \(A'\) is empty if \(m' < m\) or \(A\) empty. By IH there are substitutions \(\theta_1, \ldots, \theta_k\) so that

\[
(\circ (\Gamma_0, \Gamma') \{s/x\}, t\{s/x\}) \triangleright^* (1, t\{s/x\} \theta'_1, \ldots, (1, t\{s/x\} \theta'_k), B',
\]

\[
(\circ \Pi \{s/x\} \theta'_1, t\{s/x\} \theta'_1), \ldots, (\circ \Pi \{s/x\} \theta'_k, t\{s/x\} \theta'_k) \triangleright^* (1, s_1), \ldots, (1, s_m')C'
\]

for some \(B', C'\) which are empty if \(A'\) is. So choose \(\theta_i := \{s/x\} \circ \theta'_i\) for \(1 \leq i \leq k'\). Application of the induction hypothesis to all remaining \((\circ (\Gamma_j, \Gamma', \Pi), t)\) that contribute to \((1, s_1), \ldots, (1, s_m)\) yields \(\theta_{i+1}, \ldots, \theta_k\).

Now assume \(G_1 \sqsubseteq G_2\) and \(G_3 \sqsubseteq G_1\), let \(\theta\) be a substitution replacing \(t\) variables by ct terms, and \(t\) a ct term containing all variables in \(G_1 t, G_2 t, G_3 t, G \theta t\). Let \((\circ (G_1 \circ G_3 \circ \theta, t) \triangleright^* (1, s_1), \ldots, (1, s_m), A\) for ct terms \(s_1, \ldots, s_m\) and some \(A\). Using (III) we can infer that there are substitutions \(\theta_1, \ldots, \theta_k\) so that

\[
(G_1 t, t) \triangleright^* (1, t\theta_1), \ldots, (1, t\theta_k), B,
\]

\[
(G_2 t, t\theta_1), \ldots, (G_3 \circ \theta \circ t\theta_k, t\theta_k) \triangleright^* (1, s_1), \ldots, (1, s_m), C.
\]

As \(G_1 \sqsubseteq G_2\), there is some \(B'\) which is empty if \(B\) is so that

\[
(G_2 \circ \theta, t) \triangleright^* (1, \theta_1), \ldots, (1, t\theta_k), B'
\]

up to renaming. W.l.o.g. we assume the renaming to be empty. As \(G_3 \sqsubseteq G_4\), there is some \(C'\) so that

\[
(G_4 \circ \theta_1, t\theta_k), \ldots, (G_4 \circ \theta_k, t\theta_k) \triangleright^* (1, s_1), \ldots, (1, s_m), C'
\]
up to renaming and \( C' \) is empty if \( C \) is. Now (II) applies and yields
\[
(\mathcal{G}_2 \theta \odot \mathcal{G}_4 \theta, t) \vdash^* (1, s_1), \ldots, (1, s_m), \mathcal{D}
\]
up to renaming for some \( \mathcal{D} \) which is empty if \( B', C' \) are, hence if \( \mathcal{A} \) was.

Now we have finished all preparations for the proof of (*) which is restated below.

**Lemma 3.14**
Let \( P \) be a program and \( \Gamma, \Delta \) consist of goals executable for \( P \). If \( P \vdash \Gamma \Rightarrow \Delta \) then \( \bullet \Gamma \subseteq P \ast \Delta \).

**Proof.** If \( P \vdash \Gamma \Rightarrow \Delta \) for executable goals \( \Gamma, \Delta \), we can obtain a derivation where all goals are executable for \( P \) by replacing every non-\( \iota \) variable by a closed term of appropriate type and every atomic formula with an undefined predicate symbol by 0. Now we prove the lemma by induction on the height of such a derivation. This amounts to checking all axioms and rules. We considered the more involved cases in Lemmas 3.11, 3.12, 3.13. The case of the lazy list comprehension rule is presented as an example: let \( \nabla(y), G \Rightarrow H \vdash t \approx (y \mid G)(n) \Rightarrow t \approx (y \mid H)(n) \) by induction hypothesis. To show \( t \approx (y \mid G)(n) \subseteq t \approx (y \mid H)(n) \), let \( \theta \) be a substitution replacing \( \iota \) variables by ct terms and \( r \) a ct term. We assume \( (y \mid G)(n) \theta \not\vdash s \) and show \( (y \mid H)(n) \theta \not\vdash s \): let \( \theta' \) be obtained from \( \theta \) by replacing a pair \( r'/y \) — if there is one — by a renaming \( z/y \), \( z \) fresh. Let \( \bar{u} \) be a listing of the variables in the range of \( \theta' \) distinct from \( z \). If \( s \equiv \text{NO} \) then by Lemma 2.17 \( \llbracket G \theta' \rrbracket_{\bar{u}} \) is total and has less than \( n \) elements. This carries over to \( \llbracket H \theta' \rrbracket_{\bar{u}} \), using \( \nabla(y) \odot G \subseteq H \) also to \( \llbracket H \theta' \rrbracket_{\bar{u}} \), hence to \( \llbracket H \theta' \rrbracket_{\bar{u}} \), implying \( \llbracket (y \mid H)(n) \theta \rrbracket = \text{NO} \). Now assume that \( s \) is an ect term. By similar reasoning we can deduce that \( \llbracket G \theta' \rrbracket_{\bar{u}} \) has at least \( n \) elements and \( \llbracket G \theta' \rrbracket_{\bar{u}}(n) \equiv e(r_1, \ldots, r_m) \) for some \( \bar{r} \) so that \( e(r_1) = s \). Using \( \nabla(y) \odot G \subseteq H \) again, this carries over to \( \llbracket H \theta' \rrbracket_{\bar{u}}(n) \), implying \( \llbracket (y \mid H)(n) \theta \rrbracket = s \).

Combining this with Lemma 3.8 we obtain the main theorem.

**Theorem 3.15**
Let \( P \) be a program, \( G \) an executable goal for \( P \), \( \text{fv}(G) = \{ \bar{x} \} \), \( S \) a solved goal, \( \text{FV}(S) \subseteq \{ \bar{x} \} \), and \( t \equiv \text{NO} \) or \( t \equiv e(t_0) \) for a ct term \( t_0 \). Then \( \llbracket S \rrbracket_{\bar{x}} \subseteq \llbracket G \rrbracket_{\bar{x}} \) iff \( S \) is a correct solution for \( P \). The term \( t \) is the \( n \)th computed value for \( y \) in \( G \) w.r.t. \( P \) if it is the \( n \)th correct value for \( y \) in \( G \) w.r.t. \( P \).

**Proof.** The direction ‘\( \vdash \Rightarrow \)’ is proved in Lemma 3.8. If \( S \) is a correct solution for \( P \), \( G \) then \( P \vdash \nabla(\bar{x}), S \Rightarrow G \). Using Lemma 3.14 we obtain \( \nabla(\bar{x}) \odot S \subseteq G \), hence \( \llbracket S \rrbracket_{\bar{x}} \subseteq \llbracket G \rrbracket_{\bar{x}} \). Now let \( t \) be the \( n \)th correct value for \( y \) in \( G \). Then \( P \vdash \nabla(\bar{x}), s \Rightarrow t \Rightarrow s \approx (y \mid G)(n) \) for all terms \( s \). By Lemma 3.14 this implies \( \llbracket s \approx t \rrbracket_{\bar{u}} \subseteq \llbracket s \approx (y \mid G)(n) \rrbracket_{\bar{u}} \), hence \( \llbracket s = t \rrbracket_{\bar{u}} = \llbracket s \approx (y \mid G)(n) \rrbracket_{\bar{u}} \), for all ect terms \( s \) and variables \( \bar{u} \). Choosing \( s \equiv t \) we obtain that there is an ect term \( r \) so that \( (y \mid G)(n) \not\vdash r \) and \( r \equiv \text{NO} \) if \( t \equiv \text{NO} \). Now assume \( t \equiv e(t_0), r \equiv e(r_0) \) for ct terms \( t_0, r_0 \). Then \( \llbracket s_0 = t_0 \rrbracket_{\bar{u}} = \llbracket s_0 = r_0 \rrbracket_{\bar{u}} \) for all ct terms \( s_0 \) and variables \( \bar{u} \). Choosing first \( s_0 \equiv t_0 \) then \( s_0 \equiv r_0 \) we obtain that \( t_0, r_0 \) are unifiable and restricting the mgu to the variables of \( t_0 \) or \( r_0 \) respectively yields a renaming.

As a consequence, the \( n \)th correct value is uniquely determined. Furthermore we obtain the following alternative characterizations.

**Lemma 3.16**
Let \( P \) be a program, \( G \) an executable goal for \( P \), and \( \bar{x}, \bar{y} \) distinct variables so that \( \text{FV}(G) = \{ \bar{x}, \bar{y} \} \).
For $t \equiv e(t_0)$ where $t_0$ is a ct term, $FV(t_0) \cap \{x\} = \emptyset$, the following are equivalent:

1. $P \vdash \nabla(y), y = s_1 \ldots \ast y = s_{n-1} \ast y = t \ast 0 \Rightarrow \exists G$ for some closed ct terms $s_1, \ldots, s_{n-1}$.
2. $P \vdash \nabla(x), z \approx t \Rightarrow z \approx (y \mid G)(n)$ for fresh $z$.
3. $P \vdash \nabla(x), \vec{u} \Rightarrow t_0 \approx \langle y \mid G \rangle(n)$ where $FV(t_0) = (\vec{u})$ and all ct terms satisfying $P \vdash \nabla(x), \vec{u} \Rightarrow s_0 \approx \langle y \mid G \rangle(n)$ where $FV(s_0) = (\vec{v})$, $\vec{v} \cap \{x\} = \emptyset$, are instances of $t_0$.

If (1)–(3) hold then for ct terms $s_0$, $P \vdash \nabla(x) \Rightarrow e(s_0) \approx \langle y \mid G \rangle(n)$ iff $s_0$ is unifiable with $t_0$, and $P \not\vdash \nabla(x) \Rightarrow NO \approx \langle y \mid G \rangle(n)$.

(b) The following are equivalent:

1. $P \vdash \nabla(y), y = s_1 \ast \ldots \ast y = s_k \Rightarrow \exists G$ for some $k < n$ and closed ct terms $s_1, \ldots, s_k$.
2. $P \vdash \nabla(x), z \approx NO \Rightarrow z \approx \langle y \mid G \rangle(n)$ for fresh $z$.
3. $P \vdash \nabla(x) \Rightarrow NO \approx \langle y \mid G \rangle(n)$

If (1)–(3) hold then $P \not\vdash \nabla(x) \Rightarrow s \approx \langle y \mid G \rangle(n)$ for closed ct terms $s$.

**Proof.** (1) implies (2) by a combination of (llcr) and lazy list axioms. Now assume (2), use Theorem 3.15, and show (1),(3). If $\langle y \mid G \rangle(n) \Downarrow t$, $t \in$ ect, then there are terms $s_1, \ldots, s_n$ so that $(G, y) \triangleright^* (1, s_1), \ldots, (1, s_n), A$ for some $A$ and $t \equiv e(s_n)$, hence $(E\exists G, y) \triangleright^* (1, s_1), \ldots, (1, s_{n-1}), (1, t), A$. Using Lemma 3.8 and (epsilon) axioms, this implies (1). Substituting by $t_0$, (2) implies $P \vdash \nabla(x), \vec{u} \Rightarrow t_0 \approx \langle y \mid G \rangle(n)$, as $P \vdash \nabla(\vec{u}) \Rightarrow t_0 \approx e\vec{u}.t_0$. If furthermore $P \vdash \nabla(x), \vec{v} \Rightarrow s_0 \approx \langle y \mid G \rangle(n)$ for a ct term $s_0$, $FV(s_0) = (\vec{v})$, then $[e\vec{v}.(\vec{v})] = [s_0 \approx \langle y \mid G \rangle(n)](\sigma) = [s_0 = e(t_0)](\sigma)$ by Lemma 3.14, hence $s_0$ is an instance of $t_0$. The case $t \equiv NO$ is treated similar. As to (3) $\Rightarrow$ (2), we use Theorem 3.15 to deduce from (3) that $\langle y \mid G \rangle \Downarrow r$ for some ectn term $r$ and that $r \equiv NO$ iff $t \equiv NO$. Now we use Lemma 3.8 to deduce that $r$ satisfies condition (2). For $t \equiv NO$ we know already $t \equiv r$. If $t$ is an ect term then so is $r$. We have just shown that (2) $\Rightarrow$ (3), and (3) uniquely determines $e(t_0)$. So $t \equiv r$ also in the case $t \in$ ect. Finally assume (1)–(3). By Theorem 3.15 this implies $\langle y \mid G \rangle(n) \Downarrow t$. Combining this with Theorem 3.15 yields for closed ectn terms $s$: $P \vdash \nabla(x) \Rightarrow s \approx \langle y \mid G \rangle(n)$ iff $[s = t](\sigma) = [()]$ which concludes the proof.

### 3.4 Conservative extensions and inversion properties

It has been shown that the axioms and rules presented above are sufficient to justify the computed results. For verification purposes however, it may be desirable to have more properties of the evaluation at hand. More knowledge concerning results can be integrated by extending the calculus. Furthermore we can use properties of the derivability relation or evaluation process. Considering extensions $\vdash^c$ we want to stick to our convention that derivability can not be lost by substitution, i.e. $P \vdash^c \Gamma \Rightarrow \Delta$ implies $P \vdash^c \Gamma \Theta \Rightarrow \Delta \Theta$.

**Definition 3.17**

Let $\vdash^c$ be an extension of $\vdash$, and Th a function mapping programs to axiom sets. (Th,$\vdash^c$) is called an extension iff $P \vdash^c \Gamma \Rightarrow \Delta$ implies Th($P$) $\vdash^c \Gamma \Rightarrow \Delta$ for all programs $P$ and goals $\Gamma, \Delta$. An extension is called **conservative w.r.t. results** iff for all programs $P$ and executable goals $G$ for $P$ the notions of correct solutions and values are not changed by that extension, i.e. for executable goals $G$, $FV(G) = \{\vec{x}\}$, and solved goals $S$, $FV(S) \subset \{\vec{x}\}$: Th($P$) $\vdash^c \nabla(\vec{x}), S \Rightarrow G$ iff $P \vdash^c \nabla(\vec{x}), S \Rightarrow G$, and Th($P$) $\vdash^c \nabla(\vec{x}), z \approx t \Rightarrow z \approx \langle y \mid G \rangle(n)$ iff...
The extension obtained by replacing the useful in proving properties without referring to the operational semantics directly. As these inversion properties can be phrased in terms of solutions and values, they immediately carry over to extensions conservative w.r.t. results. Inversions and extensions can be useful in proving properties without referring to the operational semantics directly.

**Lemma 3.18**

An extension \((\text{Th}, \vdash \varepsilon)\) is conservative w.r.t. results iff for all programs \(P\) and all finite sequences \(\Gamma, \Delta\) of executable goals \(\text{Th}(P) \vdash \varepsilon \Gamma \Leftarrow \Delta\) implies \(\otimes \Gamma \subseteq_P * \Delta\).

**Proof.** For ‘\(\Leftarrow\)’, the conservativeness follows using Lemma 3.8; consider ‘\(\Rightarrow\)’: Let \(P\) be a program, \(\Gamma, \Delta\) executable goals for \(P\), \(\text{Th}(P) \vdash \varepsilon \Gamma \Rightarrow \Delta\). Then \(\theta\) is a substitution replacing \(i\) variables by ct terms and \(\bar{x}\) the free variables in \(\Gamma\theta, \Delta\theta\). We have to show \(\llbracket \otimes \Gamma\theta \rrbracket \llbracket x \rrbracket \subseteq \llbracket \ast \Delta\theta \rrbracket \llbracket x \rrbracket\). It is sufficient to show that \(\llbracket S \rrbracket \llbracket x \rrbracket \subseteq \llbracket \otimes \Gamma\theta \rrbracket \llbracket x \rrbracket\) implies \(\llbracket S \rrbracket \llbracket x \rrbracket \subseteq \llbracket \ast \Delta\theta \rrbracket \llbracket x \rrbracket\) for all solved goals \(S\) satisfying \(FV(S) \subseteq \{\bar{x}\}\). If \(\llbracket S \rrbracket \llbracket x \rrbracket \subseteq \llbracket \otimes \Gamma\theta \rrbracket \llbracket x \rrbracket\) for such an \(S\), then \(\text{Th}(P) \vdash \varepsilon \nabla(\bar{x}), S \Rightarrow \otimes \Gamma\theta\), as \((\text{Th}, \vdash \varepsilon)\) is an extension and Lemma 3.8 holds, hence \(\text{Th}(P) \vdash \varepsilon \nabla(\bar{x}), S \Rightarrow * \Delta\theta\). As the extension is conservative w.r.t. results, this implies \(P \vdash \nabla(\bar{x}), S \Rightarrow * \Delta\theta\), thus by Theorem 3.15 \(\llbracket S \rrbracket \llbracket x \rrbracket \subseteq \llbracket \ast \Delta\theta \rrbracket \llbracket x \rrbracket\). 

**Example 3.19**

The extension obtained by replacing the \(\Rightarrow\) in the program dependent axioms \(\text{Def}_{P}(\bar{t}) \Rightarrow p(\bar{t}), \nabla(\bar{E})\), and in the axioms for lazy lists by \(\Leftrightarrow\) or \(\Leftrightarrow_{\nabla}(\bar{x})\) respectively, and strengthening \(\nabla(\text{contraction})\) to \(\nabla(\bar{t}) \nabla(\bar{t}) \Leftrightarrow \nabla(\bar{t})\) is conservative w.r.t. results.

The rules 2–4 in Corollary 3.4 can be inverted:

**Lemma 3.20**

Let \(P\) be a program and \(G\) an executable goal for \(P\), \(FV(G) \subseteq \{\bar{x}, y\}\), \(\bar{x} = x_1, \ldots, x_k\).

(a) If \(P \vdash \mu\bar{x}(\emptyset) \Rightarrow E \exists \bar{y} G(\bar{x})\), then there are ct tuples \((\bar{r}(i), s_i), 1 \leq i \leq n\), of length \(k+1\), so that \(P \vdash \nabla(\bar{x}, y), s_n^0 = (\bar{r}(1), \mu s_1, G) \Rightarrow G\).

(b) If \(P \vdash \nabla(\bar{x}), z \approx \varepsilon(s) \Rightarrow z \approx \mu \bar{y}, G\), for \(s\) in ct, then there are ct terms \(\bar{t}\) and a solved goal \(S \equiv \mu s^0 = S_i\) where \(S_n \equiv E \exists \bar{x}(\bar{r}(i), \mu y = s)\) so that \(P \vdash \nabla(\bar{x}, y), S \Rightarrow G\). If \(P \vdash \nabla(\bar{x}), z \approx \emptyset \Rightarrow z \approx \mu \bar{y}, G\), then there is \(m < n\) and a solved goal \(S \equiv \mu s^0 = S_i, S_m \neq \emptyset\), so that \(P \vdash \nabla(\bar{x}, y), S \Rightarrow G\).

(c) If \(G \equiv t_1 \approx t_2\) has a correct solution \(S \neq \emptyset\), \(FV(G) \subseteq \{\bar{x}\}\), then there are ct terms \(s_1, s_2\) so that \(P \vdash \nabla(\bar{x}), z \approx s_i \Rightarrow z \approx t_i\) for \(i = 1, 2\) and \(P \vdash \nabla(\bar{x}), S \Rightarrow s_1 = s_2\). If \(t_i = \varepsilon(t_i'), t_i' \in\) ct, or \(t_i \Leftarrow \emptyset\) then we can choose \(s_i \equiv t_i\).

**Proof.** Assume \(P \vdash \mu\bar{x}(\emptyset) \Rightarrow E \exists \bar{y} G(\bar{x})\). Then by Theorem 3.15 \(\llbracket (\ldots, (\ldots, (\emptyset, y), \bot) \ldots) \rrbracket \subseteq \llbracket G(\bar{x}) \rrbracket\) with \(n\) copies of \((\ldots, (\emptyset, y), \bot) \ldots\). Using Lemma 2.17, we obtain that there are \(\bar{r}(i), s_i\) so that \((G, (\bar{x}, y)) \Rightarrow^* (1, (\bar{r}(i), s_i)), \ldots, (1, (\bar{r}(n), s_n)))\), and \(\varepsilon(s_n) = \llbracket \mu \bar{y}, G \rrbracket\). Applying Theorem 3.15 completes the proof of (a). Similar reasoning yields (b). Now consider (c). If \(P \vdash \nabla(\bar{x}), S \Rightarrow t_1 \approx t_2\), then by Theorem 3.15 \(\llbracket S \rrbracket \llbracket x \rrbracket \subseteq \llbracket t_1 \approx t_2 \rrbracket \llbracket x \rrbracket\). As \(S \neq \emptyset\) is solved either \((S, (\bar{x})) \Rightarrow^* (1, r, s, A) \Rightarrow^* (1, r, \delta, A')\) or \((S, (\bar{x})) \Rightarrow^* (1, r, \delta, A) \Rightarrow^* (1, r, \delta, A')\) respectively for some renaming \(\delta\), and some \(A'\). So there are terms \(s_1, s_2\) so that \(t_1 \downarrow s_1\) and \(t_2 \downarrow s_2\) and \(\llbracket S \rrbracket \llbracket x \rrbracket \subseteq \llbracket t_1 \approx t_2 \rrbracket \llbracket x \rrbracket = \llbracket s_1 \approx s_2 \rrbracket \llbracket x \rrbracket\). An application of Lemma 3.8 completes the proof.
Example 3.21
Let \( P \) be a program containing a predicate definition for \( p \) and none for \( q \). Extend \( P \) by:

\[
q(x) : \neg \text{NO} \approx \text{fst}(\langle y \mid p(x) \rangle)
\]

In Example 3.7 we showed for ct terms \( t \): If \( p(t) \) fails then \( q(t) \) has the correct solution \( 1 \), and if \( p(t) \) has a correct solution \( \forall \bar{y} \bar{x} = \bar{s} \) then \( q(t) \) fails. To show the converse, we consider the extension suggested in Example 3.19. If \( \text{Th} (P) \vdash \forall \bar{y}, S \Rightarrow q(t) \) for some solved \( S \neq \emptyset \), then \( \text{Th} (P) \vdash \forall \bar{y}, S \Rightarrow \text{NO} \approx \text{fst}(\langle y \mid p(x) \rangle) \). By Lemma 3.20 this implies the termination of \( p(t) \).

4 Conclusion

We presented an axiomatic extension of pure prolog by adding the concept of list comprehension and exemplified its usefulness. The operational semantics defined above describes a goal-directed evaluation.

The language contains functional terms but no recursion on the level of function definitions. It contains higher order terms but no (restricted) version of higher order unification or any means to compute higher order terms as present in higher order logic programming [11]. An integration of these features is not yet considered.

Lazy list comprehension can be used as a tool for defining control which can be expressed in terms of testing for failure and picking or ignoring answers at certain positions. In contrast to more general approaches to defining search as [8], lazy list comprehension does not refer to the computation itself but only to the result. Hence it is independent of operational details and fits well into a declarative setting. The language is based on pure prolog which implies determinism and fixed selection but lazy list comprehension as described here can also be combined with an extension. For retaining the axiomatic semantics, the critical point is describing the extension itself. The axiomatic approach is suitable for multiple definitions which can be nondeterministically chosen and restrictions for applications, e.g. to ground arguments. Although it is not straightforward how to integrate powerful computation rule control as presented in [2, 12], one could, to describe a ‘delaying negation until ground’ mechanism, modify the calculus to relax the fixed literal selection.

So far, lazy lists can be accessed only by the \( \text{fst}/\text{rest} \) functions presented above. In the same manner we can introduce a binary ‘fsts’ which produces (nondeterministically) one of the first elements if possible, accompanied by the information which argument was chosen. This yields parallel access to the first elements of several lists. Interleaving of lazy lists in a not necessarily fair manner is then definable. Parallel composition, however, would still have to be added explicitly. For the axiomatic description it should be accompanied by further operations on lists or parallel composition of goals, changing in any case the underlying logical language.

The logical semantics is based on a calculus. For the substructural kernel, biquantale structures have already been introduced in [5, 6] as a class of models. There we also defined a standard model for pure prolog programs. It makes use of a domain of stream functions which is adequate for a compositional semantics of those programs. This suggests a way to define a denotational semantics for programs with lazy lists. Also a model theoretic semantics could be obtained by adapting the notions that proved valuable in the pure prolog case. A thorough treatment of this class of models, however, is left to future work.
Lazy List Comprehension in Logic Programming

References


Received 18 May 1998