ALGEBRAIC AND LOGICAL SEMANTICS FOR CLP LANGUAGES WITH DYNAMIC SCHEDULING

KIM MARRIOTT

The first logic programming languages, such as Prolog, used a fixed left-to-right atom scheduling rule. Unfortunately, this meant that programs written in a clean, declarative style were often very inefficient, only terminated when certain inputs were fully instantiated or "ground," and (if negation was used) produced wrong results. For this reason, nearly all recent logic programming languages provide more flexible scheduling in which computation generally proceeds left-to-right but in which some calls are dynamically "delayed" until their arguments are sufficiently instantiated to allow the call to run efficiently. Despite the increasing practical importance of logic programming languages with dynamic scheduling, relatively little attention has been paid to their semantics. We lift the standard algebraic and logical soundness and completeness results for success and finite failure for constraint logic programs to constraint logic programming languages with dynamic scheduling. The proofs are quite simple and essentially rely on treating the delayed calls as part of the answer constraint. © Elsevier Science Inc., 1997

1. INTRODUCTION

The first logic programming languages, such as DEC-10 Prolog, used a fixed scheduling rule in which all atoms in the goal were processed left-to-right. Unfortunately, this meant that programs written in a clean, declarative style were often very inefficient, only terminated when certain inputs were fully instantiated or "ground," and (if negation was used) produced wrong results. For this reason,
nearly all recent logic programming languages provide more flexible scheduling in which computation generally proceeds left-to-right but in which some calls are dynamically “delayed” until their arguments are sufficiently instantiated to allow the call to run efficiently.

Dynamic scheduling overcomes the problems associated with traditional Prologs and their fixed scheduling. First, it allows the same program to have many different and efficient operational semantics as the operational behaviour depends on which arguments are supplied in the query. Thus, programs really behave efficiently as relations, rather than as functions. Second, the treatment of negation is sound, as negative calls are delayed until all arguments are ground. Third, it allows intelligent search in combinatorial constraint problems. Another benefit of dynamic scheduling is that it allows a new style of programming in which Prolog procedures are viewed as processes which communicate asynchronously through shared variables. Yet another reason for interest in languages with dynamic scheduling is that they are being used for the implementation of concurrent constraint programming languages [1] and for implementing extendible “open” constraint solvers in which the solver can be extended using guarded rules [3].

Despite the increasing practical importance of logic programming languages with dynamic scheduling, relatively little attention has been paid to their semantics. We extend the standard algebraic and logical soundness and completeness results for success and finite failure for constraint logic programs to constraint logic programming languages with dynamic scheduling of literals. The proofs are quite simple, and essentially rely on the observation that the delayed atoms behave like constraints.

The operational semantics of logic programming languages with delay has been discussed by Naish [7, 8]. Yellick and Zachary [9] and Naish [8] show that, under certain restrictions, the operational semantics of logic programming languages with delay is confluent in the sense that different atom schedulings give rise to the same possible outcomes. Marriott et al. [6] give a simple denotational semantics for logic languages with delay based on sets of closure operators. In a sense, the semantics of Marriott et al. generalizes the bottom-up $S$-semantics of logic programs [2].

Strangely, there has not been a systematic investigation of the algebraic and logical semantics for languages with delay. The usual folk-wisdom appears to be that derivations which “flounder,” that is, cannot proceed because all atoms are delayed, are a runtime error. In the absence of floundering the usual results for algebraic and logical semantics for constraint logic programs routinely carry over. However, floundering is an essential characteristic of languages with delay. Here we show that with the view that the delayed atoms are just constraints, the logical and algebraic semantics carry over even to goals which lead to floundering. This view of delayed atoms as constraints is implicit in the behavior of one of the earliest languages with delay, MU-Prolog [7].

The plan of the paper is as follows: In the next section we give two example programs which illustrate the usefulness of dynamic scheduling. They will be used as running examples throughout the text. In Section 3 we define the operational semantics of constraint logic programming with dynamic scheduling. In Section 4 we give soundness and completeness results for success and in Section 5 we give soundness and completeness results for finite failure. Finally, in Section 6 we summarize our results.
2. EXAMPLES

The following program adapted from Naish [7] is a well-known example to illustrate the power of dynamic scheduling. The program `permute` is a definition of the relation "to be a permutation of." It makes use of the procedure `delete(X, Y, Z)`, which holds if Z is the list obtained by removing X from the list Y (uppercase letters denote variables and ":" denotes list concatenation):

\[
\begin{align*}
\text{permute}(\text{nil}, \text{nil}) & \leftarrow \text{true} \\
\text{permute}(U:X, Y) & \leftarrow \text{delete}(U, Y, Z) \land \text{permute}(X, Z). \\
\text{delete}(X, X:Z, Z) & \leftarrow \text{true}. \\
\text{delete}(X, U:Y, U:Z) & \leftarrow \text{delete}(X, Y, Z).
\end{align*}
\]

Clearly the relation declaratively given by `permute` is symmetric. Unfortunately, the behaviour of the program with traditional Prolog is not: Given the goal Q1,

\[? - \text{permute}(X, a:b:\text{nil}).\]

Prolog will correctly backtrack through the answers \(X = a:b:\text{nil}\) and \(X = b:a:\text{nil}\). However for the goal Q2,

\[? - \text{permute}(a:b:\text{nil}, Y),\]

Prolog will first return the answer \(Y = a:b:\text{nil}\) and on subsequent backtracking will go into an infinite derivation without returning any more answers.

For languages with delay the program `permute` does behave symmetrically. For instance, if the above program is given to the NU-Prolog compiler, a preprocessor will generate the following when declarations:

\[
\begin{align*}
? - \text{permute}(X,Y) \text{ when } X \text{ or } Y. \\
? - \text{delete}(X,Y:Z,U) \text{ when } Z \text{ or } U.
\end{align*}
\]

These may be read as: the call `permute(X,Y)` should delay until \(X\) or \(Y\) is not a variable and that the call `delete(X,Y:Z,U)` should delay until \(Z\) or \(U\) is not a variable. Of course programmers can also annotate their programs with `when` declarations.

Given these declarations, the above goals will behave in a symmetric fashion, backtrack through the possible permutations, and then fail. What happens is that, with Q1, execution proceeds as in standard Prolog because no atoms are delayed. With Q2, however, calls to `delete` are delayed and only awaken after the recursive calls to `permute`.

It is also interesting to consider what will happen with the goal Q3,

\[? - \text{permute}(X,Y).\]

In this case the atom will delay and the constraint `true` will be returned. This behavior is very different from traditional Prolog executed with any literal selection rule and illustrates the difference that dynamic scheduling brings.
Another example of the use of dynamic scheduling is the program to compute the factorial function

\[ \text{fac}(0, 1) \leftarrow \text{true}. \]

\[ \text{fac}(N, N \ast F) \leftarrow N \geq 1 \land \text{fac}(N - 1, F). \]

This could be annotated with the guard

\[ ? - \text{fac}(N, F) \text{ when exists a } N \leq a. \]

which can be read as delay \( \text{fac}(N, F) \) until the current constraints imply that \( N \) is bounded above by some fixed \( a \).

3. OPERATIONAL SEMANTICS

In this section we recall some basic notions and we define an operational semantics for constraint logic programs with dynamic scheduling. The operational semantics is based on that given in [8], but generalised to arbitrary constraints.

A constraint logic program (CLP) or program is a finite set of rules. A rule is of the form \( H \leftarrow B \), where \( H \), the head, is an atom and \( B \), the body, is a finite, nonempty sequence of literals. We let \( \text{nil} \) denote the empty sequence. A literal is either an atom or a primitive constraint. An atom has the form \( p(t_1, \ldots, t_n) \), where \( p \) is a user-defined predicate symbol and the \( t_i \) are terms which may be constructed by using predefined functions such as real addition, predefined constants, and variables. A primitive constraint is similar to an atom except that the predicate symbol has a predefined interpretation, such as term equations or inequalities over the reals. We assume that equality (=) is a primitive constraint.

A constraint is a conjunction of primitive constraints. Constraints are usually treated modulo logical equivalence. We let \( \exists_W c \) denote the constraint \( \exists V_1 \exists V_2, \ldots, \exists V_n c \), where variable set \( W = \{V_1, \ldots, V_n\} \), and we let \( \exists_W c \) denote the restriction of the constraint \( c \) to the variables in \( W \). That is, \( \exists_W c = \exists_{\text{vars}(c) \setminus W} c \) where the function \( \text{vars} \) takes a syntactic object and returns the set of free variables occurring in it. We let \( \exists c \) denote the existential closure of \( c \).

The constraints have an intended interpretation \( D \) and a first-order logical theory \( T \). \( D \) is required to be a model of \( T \). The theory and intended interpretation are required to treat equality correctly. That is, the theory must contain the equality axioms and in the interpretation equality should be identity.

A renaming is a bijective mapping between variables. We naturally extend renamings to mappings between atoms, rules, and constraints. Syntactic objects and \( s' \) are said to be variants if there is a renaming \( \rho \) such that \( \rho(s) = s' \). The definition of an atom \( p(t_1, \ldots, t_n) \) in program \( P \), \( \text{defn}_p(p(t_1, \ldots, t_n)) \), is the set of variants of rules in \( P \) such that the head of each rule has form \( p(s_1, \ldots, s_n) \). To sidestep renaming issues, we assume that each time \( \text{defn}_P \) is called it return variants with distinct new variables.

We first review the operational semantics of CLP programs in which atom cannot dynamically delay. The operational semantics is given in terms of the "derivations" from goals. Derivations are sequences of reductions between "states, where a state is a tuple \( \langle G, c \rangle \) which contains the current literal sequence \( c \) "goal" \( G \) and the current constraint \( c \). At each reduction step, a literal in the go
is selected according to some fixed selection rule, which is often left-to-right. If the literal is a primitive constraint and it is consistent with the current constraint, then it is added to it. If it is inconsistent, then the derivation "fails." If the literal is an atom, it is reduced using one of the rules in its definition.

The operational semantics is defined in terms of a constraint solver, \( solv(c) \), which takes a constraint \( c \) and returns one of \{true, false, unknown\}. True and false indicate that the constraint theory (and hence the intended constraint interpretation) imply that \( c \) is satisfiable or unsatisfiable, respectively. If the solver returns unknown this means the solver cannot determine satisfiability; it does not mean that the constraint theory does not imply satisfiability or unsatisfiability of the constraint. Thus the solver is allowed to be incomplete. If the solver only ever returns true or false it is said to be complete. We require that the solver does not take variable names into account, that is, for all renamings \( \rho \), \( solv(c) = solv(\rho(c)) \).

A state \( \langle G, c \rangle \) can be reduced as follows: Select a literal \( L \) from \( G \) and let \( G' \) be the remaining literals in \( G \). Then:

1. If \( L \) is a primitive constraint and \( solv(c \land L) \neq false \), it is reduced to \( \langle G', (c \land L) \rangle \).
2. If \( L \) is a primitive constraint and \( solv(c \land L) = false \), it is reduced to \( \langle \emptyset, false \rangle \).
3. If \( L \) is an atom, then it is reduced to \( \langle s_1 = t_1 \land \ldots \land s_n = t_n \land A \land G', c \rangle \) for some \( (A \leftarrow B) \in defn_\rho(L) \), where \( L \) is of form \( p(s_1, \ldots, s_n) \) and \( A \) is of form \( p(t_1, \ldots, t_n) \).
4. If \( L \) is an atom and \( defn_\rho(L) = \emptyset \), it is reduced to \( \langle \emptyset, false \rangle \).

Here we have used the symbol \( \emptyset \) to denote the empty goal.

A derivation from a goal \( G \) in a program \( P \) using selection strategy \( S \) is a sequence of states \( S_0 \rightarrow S_1 \rightarrow \cdots \rightarrow S_n \), where \( S_0 = \langle G, true \rangle \) and there is a reduction from each \( S_{i-1} \) to \( S_i \), using rules in \( P \) and at each stage the selection strategy \( S \) is used to choose the selected literal. We require that the selection strategy be complete in the sense that it will always select a literal in a state unless the goal in the state is empty. A derivation from \( G \) is finished if the last state in the derivation cannot be reduced. In the CLP semantics without dynamic scheduling, the last state in a finished derivation from \( G \) must have the form \( \langle \emptyset, c \rangle \). If \( c \) is false, the derivation is said to be failed. Otherwise the derivation is successful, with answer \( \exists_{vars(G)} c \).

The operational semantics for CLP languages with dynamic scheduling is very similar. It makes use of the predicate \( delay(L, c) \), which holds if a call to the literal \( L \) delays with the constraint \( c \). We require that \( delay \) does not take variable names into account, that is, for all renamings \( \rho \),

\[
\text{delay}(L, c) \leftrightarrow delay(\rho(L), \rho(c)).
\]

The only difference in the operational semantics is that the selection strategy \( S \) is not allowed to select a literal \( L \) from state \( \langle G, c \rangle \) if \( delay(L, c) \) holds. Note that the selection strategy is still required to be as complete as possible, so that the only time it will not select a literal from a state is if all literals are delayed. We call such a selection strategy safe.

With a safe selection strategy, a successful derivation from \( G \) may end in a state of the form \( \langle G, c \rangle \), where for every literal \( L \) in \( G \), \( delay(L, c) \) holds. In this case,
we call $\exists_{\text{var}(G)}(c \land G')$ a (qualified) answer to $G$. A successful derivation with last state $\langle G, c \rangle$ is said to have floundered if $G$ is not the empty goal.

The derivations from a goal $G$ for a program $P$ using a selection strategy $\mathcal{S}$ can be combined into a derivation tree in which each path in the tree is a maximal derivation and different branches occur because of different choices of which rule an atom is reduced with. Because both delay and solv do not take variable names into account, modulo variable renaming, the derivation tree for a particular goal, program, and selection rule is unique. A goal $G$ finitely fails for a program $P$ using a selection strategy $\mathcal{S}$ if its derivation tree is finite and contains only failed derivations.

As an example, consider the initial state $\langle \text{permute}(a:nil,Y), \text{true} \rangle$ and the permute program from Section 2. These have the successful derivation shown in Figure 1, where

- $c_1$ is $a:nil = U' : X'$,
- $c_2$ is $Y = Y' \land c_1$,
- $c_3$ is $X' = \text{nil} \land c_2$,
- $c_4$ is $Z' = \text{nil} \land c_3$,
- $c_5$ is $U' = X'' \land c_4$,
- $c_6$ is $Y' = X' : Z'' \land c_5$,
- $c_7$ is $Z' = Z'' \land c_6$.

Note that at each step in the figure the selected literal has been underlined.

In the above discussion, a constraint logic programming language is parameterised by the underlying constraint domain $\mathcal{C}$ and solver. The constraint domain determines the primitive constraint symbols and the set of function and constant symbols from which terms in the program may be constructed. The solver deter-

\[
\begin{align*}
(\text{permute}(a : \text{nil}, Y), \text{true}) & \downarrow \\
(a : \text{nil} = U' : X' \land Y = Y' \land \text{delete}(U', Y', Z') \land \text{permute}(X', Z'), \text{true}) & \downarrow \\
(Y = Y' \land \text{delete}(U', Y', Z') \land \text{permute}(X', Z'), c_1) & \downarrow \\
(\text{delete}(U', Y', Z') \land \text{permute}(X', Z'), c_2) & \downarrow \\
(\text{delete}(U', Y', Z') \land X' = \text{nil} \land Z' = \text{nil} \land \text{true}, c_2) & \downarrow \\
(\text{delete}(U', Y', Z') \land Z' = \text{nil} \land \text{true}, c_3) & \downarrow \\
(\text{delete}(U', Y', Z') \land \text{true}, c_4) & \downarrow \\
(\text{delete}(U', Y', Z'), c_4) & \downarrow \\
(U' = X'' \land Y' = X'' : Z'' \land Z' = Z'' \land \text{true}, c_4) & \downarrow \\
(Y' = X'' : Z'' \land Z' = Z'' \land \text{true}, c_5) & \downarrow \\
(Z' = Z'' \land \text{true}, c_6) & \downarrow \\
(\text{true}, c_7) & \downarrow \\
(\Omega, c_7)
\end{align*}
\]

**FIGURE 1.** Example derivation.
CLP LANGUAGES WITH DYNAMIC SCHEDULING

mines when (or if) to prune a branch in the derivation tree. Different choices of constraint domain and solver give rise to different programming languages. For a particular constraint domain $\mathcal{D}$, we let $CLP(\mathcal{D})$ be the constraint programming language based on $\mathcal{D}$, $D_\mathcal{D}$ be the intended interpretation of the constraints, $T_\mathcal{D}$ be the first-order logical theory, and $sol_{\mathcal{D}}$ be the constraint solver. The following two constraint domains and consequent languages are used in the examples given in this paper.

**Example 3.1.** The language $CLP(\text{Real})$ is based on the constraint domain $\text{Real}$, which has $\leq$, $\geq$, $<$, $>$, and $=$ as the primitive constraints, function symbols $+$, $-$, $\ast$, and $/$, and sequences of digits with an optional decimal point as constant symbols. The program $fac$ is an example of a program written in $CLP(\text{Real})$.

The intended interpretation of $\text{Real}$ has as its domain the set of real numbers $\mathbb{R}$. The primitive constraints $\leq$, $\geq$, $<$, $>$, and $=$ are interpreted as the obvious arithmetic relations over $\mathbb{R}$, and the function symbols $+$, $-$, $\ast$, and $/$, are the obvious arithmetic functions over $\mathbb{R}$. Constant symbols are interpreted as the decimal representation of elements of $\mathbb{R}$. The theory of the real closed fields is a theory for $\text{Real}$.

**Example 3.2.** The language $CLP(\text{Term})$ is based on the constraint domain $\text{Term}$ which has $=$ as the primitive constraint and strings of alphanumeric characters as function symbols or as constant symbols. The program $permute$ is an example of a program written in $CLP(\text{Term})$. $CLP(\text{Term})$ is, of course, the core of Prolog.

The intended interpretation of $\text{Term}$ is the set of finite trees, $\text{Tree}$. The interpretation of a constant $a$ is a tree with a single node labelled with $a$. The interpretation of the $n$-ary function symbol $f$ is the function $f_{\text{Tree}}:\text{Tree}^n \rightarrow \text{Tree}$, which maps the trees $T_1, \ldots, T_n$ to a new tree with root node labelled by $f$ with $T_1, \ldots, T_n$ as children. The interpretation of $=$ is the identity relation over $\text{Tree}$. The natural theory $\text{Term}_T$ for the $\text{Term}$ constraint domain is the free-equality theory in which $=$ is required to be identity.

We have seen that the primitive constraints have three different semantics: an operational semantics given by the solver, an algebraic semantics given by the intended interpretation, and a logical semantics given by the theory. One of the nicest properties of the CLP languages without dynamic scheduling is that it is possible to give analogous semantics to the user-defined constraints, that is, programs. In the next two sections we shall show how to give an algebraic and logical semantics for CLP languages with dynamic scheduling which accords with their operational semantics.

4. SEMANTICS FOR SUCCESS

We now give an algebraic and a logical semantics for successful derivations and their answers. We first review some definitions from the algebraic and logical semantics of CLP languages without dynamic scheduling [5]. In essence, we prove that the soundness and completeness results for successful derivations, constructed with a complete literal selection rule, continue to hold for a safe (and possibly incomplete) selection rule, as long as an answer is now understood to be a qualified answer.
In the semantics of success a program is understood to represent the conjunction of the universal closure of its rules. The algebraic semantics of success for CLP languages is based on the “least model” of the program $P$. Arguably, the least model is the intended interpretation of the program. Clearly the intended interpretation should not change the interpretation of the primitive constraints or function symbols. All it can do is extend the intended interpretation so as to provide an interpretation for each user-defined predicate symbol in $P$.

**Definition 4.1.** A $\mathcal{E}$-interpretation for a CLP($\mathcal{E}$) program $P$ is an interpretation which agrees with $D_{\mathcal{E}}$ on the interpretation of the symbols in $\mathcal{E}$.

Since the meaning of primitive constraints is fixed by $\mathcal{E}$, we may represent each $\mathcal{E}$-interpretation $I$ simply by a subset of the $\mathcal{E}$-base of $P$, written $\mathcal{E}$-base$_P$, which is the set

$$\{p(d_1, \ldots, d_n) | p \text{ is a user-defined predicate in } P \text{ of arity } n$$

and each $d_i$ is a domain element of $D_{\mathcal{E}}\}.$

Note that the set of all possible $\mathcal{E}$-interpretations for $P$ is $\mathcal{P}(\mathcal{E}$-base$_P)$. Also note that $\mathcal{E}$-base$_P$ itself is the $\mathcal{E}$-interpretation in which each user-defined predicate is mapped to the set of all tuples, that is, in which everything is considered true.

**Definition 4.2.** A $\mathcal{E}$-model of a CLP($\mathcal{E}$) program $P$ is a $\mathcal{E}$-interpretation which is a model of $P$.

Every program has at least $\mathcal{E}$-model which is usually regarded as the intended interpretation of the program since it is the most conservative $\mathcal{E}$-model.

**Definition 4.3.** We denote the least $\mathcal{E}$-model of a CLP($\mathcal{E}$) program $P$ by $lm(P, \mathcal{E})$.

**Example 4.1.** The factorial program from Section 2 has an infinite number of Real-models, including

$$\{\text{fac}(n, n!) | n \in \{0, 1, 2, \ldots\} \} \cup \{\text{fac}(n, 0) | n \in \{0, 1, 2, \ldots\}\}$$

and

$$\{\text{fac}(r, r') | r, r' \in \mathbb{R}\}.$$

As one might hope, the least Real-model is

$$\{\text{fac}(n, n!) | n \in \{0, 1, 2, \ldots\}\}.$$

The next theorem shows that the operational semantics is sound for the least model. The proof is essentially identical to the case of CLP languages without delay and relies on inductive use of the following lemma.

**Lemma 4.1.** Let $P$ be a CLP($\mathcal{E}$) program. If $\langle G, c \rangle$ is reduced to $\langle G', c' \rangle$, $P, T_{\mathcal{E}} \models (G' \land c') \rightarrow (G \land c)$.
PROOF. Follows from a straightforward analysis of the different cases in the definition of reduction.

**Theorem 4.1 (Algebraic soundness of success).** Let $P$ be a CLP($\mathcal{E}$) program. If goal $G$ has answer $A$ obtained with a safe scheduling strategy, then

$$\text{lm}(P, \mathcal{E}) \vdash A \rightarrow G.$$ 

**PROOF.** Let $A$ be the answer. Then there must be a successful derivation

$$\langle G_0, c_0 \rangle \Rightarrow \cdots \Rightarrow \langle G_n, c_n \rangle,$$

where $G_0$ and $G, c_0$ is true, and $A$ is $\exists_{\text{vars}(G)}(c_n \land G_n)$. By repeated use of Lemma 4.1, we have that

$$\text{lm}(P, \mathcal{E}) \vdash (G_n \land c_n) \rightarrow (G_0 \land c_0).$$

Thus, $\text{lm}(P, \mathcal{E}) \vdash \exists_{\text{vars}(G_0 \land c_0)}(c_n \land G_n) \rightarrow (c_0 \land G_0)$, and so $\text{lm}(P, \mathcal{E}) \vdash A \rightarrow G$. □

Soundness tells us that the operational semantics only returns solutions which are solutions to the goal. However, we would also like completeness, that is, that the operational semantics will return all solutions to the goal. The following theorem shows that the operational semantics is also complete for the least model.

**Theorem 4.2 (Algebraic completeness of success).** Let $P$ be a CLP($\mathcal{E}$) program, $G$ be a goal, and $\mathcal{S}$ be a safe scheduling strategy. If $\text{lm}(P, \mathcal{E}) \vdash G$, for some $D_\mathcal{E}$ valuation $\sigma$, then $G$ has an answer $A$ using $\mathcal{S}$ such that $\text{lm}(P, \mathcal{E}) \vdash \sigma A$.

**PROOF.** Let $\mathcal{S}'$ be a complete (possibly unsafe) scheduling strategy which extends $\mathcal{S}$. That is, if $\mathcal{S}$ can select a literal from a goal then $\mathcal{S}'$ will select the same literal; otherwise, $\mathcal{S}'$ will arbitrarily select one of the delayed literals in the goal. From the algebraic completeness of success for CLP programs without dynamic scheduling (Theorem 6.1 of [5]) we have that for $\mathcal{S}'$ there is a successful (finished) derivation

1. $\langle G_0, c_0 \rangle \Rightarrow \cdots \Rightarrow \langle G_n, c_n \rangle$, where $G_0$ is $G$, $c_0$ is true, $G_n$ is $\boxempty$, and

2. $D_\mathcal{S} \vdash \exists_{\text{vars}(G)} c_n$.

From the definition of $\mathcal{S}'$ and (1) there must be a successful derivation from $G$ using $\mathcal{S}$ of the form

$$\langle G_0, c_0 \rangle \Rightarrow \cdots \Rightarrow \langle G_i, c_i \rangle,$$

where $i \leq n$. From Lemma 4.1, we have that

$$\text{lm}(P, \mathcal{E}) \vdash (G_n \land c_n) \rightarrow (G_i \land c_i).$$

Thus, from (2),

$$\text{lm}(P, \mathcal{E}) \vdash \sigma \exists_{\text{vars}(G)}(G_i \land c_i).$$
Hence, \( G \) has an answer, namely, \( \exists_{c \in \text{vars}(G)} (G_i \land c_i) \), using \( \mathcal{S} \) which has the desired property. \( \square \)

We now look at a logical semantics for a \( \text{CLP}(\mathcal{C}) \) program. In this case, the meaning of a program is expressed in terms of all of the program's models. The logical semantics of a \( \text{CLP}(\mathcal{C}) \) program \( P \) is the theory obtained by adding the axioms of \( P \) to a theory of the constraints. The next two theorems show that the operational semantics for CLP languages with dynamic scheduling is sound and complete for the logical semantics.

**Theorem 4.3 (Logical soundness of success).** Let \( P \) be a \( \text{CLP}(\mathcal{C}) \) program. If goal \( G \) has answer \( A \), then
\[
P, T_{\mathcal{C}} \models A \rightarrow G.\]

**Proof.** The proof is by repeated use of Lemma 4.1. \( \square \)

**Theorem 4.4 (Logical completeness of success).** Let \( P \) be a \( \text{CLP}(\mathcal{C}) \) program, \( G \) be a goal, and \( \mathcal{S} \) be a safe selection strategy. If
\[
P, T_{\mathcal{C}} \models c \rightarrow G\]
for constraint \( c \), then \( G \) has answers \( A_1, \ldots, A_n \) using \( \mathcal{S} \) such that
\[
P, T_{\mathcal{C}} \models c \rightarrow (A_1 \lor \cdots \lor A_n).\]

**Proof.** Let \( \mathcal{S}' \) be a complete (possibly unsafe) selection strategy which extends \( \mathcal{S} \). From the logical completeness of success for CLP programs without dynamic scheduling (Theorem 6.1 of [5]) we have that for \( \mathcal{S}' \) there are successful derivations with answers \( A_1', \ldots, A_n' \) s.t.
\[
T_{\mathcal{C}} \models c \rightarrow (A_1' \lor \cdots \lor A_n').\]
Using a similar argument to that in the proof of Theorem 4.2, for each \( A_i' \) there is an answer \( A_i \) to \( G \) using \( \mathcal{S} \) such that
\[
P, T_{\mathcal{C}} \models A_i' \rightarrow A_i.\]
Thus, there are (possibly nondistinct) answers to \( G \) using \( \mathcal{S} \) such that
\[
P, T_{\mathcal{C}} \models c \rightarrow (A_1 \lor \cdots \lor A_n).\]
\( \square \)

It is worth pointing out, as in the case of CLP languages without dynamic scheduling, that, in general, \( n \) can be greater than 1.

**Example 4.2.** Consider the \( \text{CLP}(\text{Real}) \) program \( P \):
\[
p(X) \leftarrow X \geq 2,\]
\[
p(X) \leftarrow X \leq 2.\]
Then
\[
P, \text{Real}_T \models \text{true} \rightarrow p(X)\]
and the answers to \( p(X) \) are \( X \geq 2 \) and \( X \leq 2 \). Both answers are needed to cover \( \text{true} \):
\[
\text{Real}_T \models \text{true} \rightarrow (X \geq 2 \lor X \leq 2).\]
However, for some constraint domains, the number of answers which need to be considered is just one. The following definition captures such cases.

**Definition 4.4.** A theory $T$ for a constraint domain has independence of constraints if for all constraints $c, c_1, \ldots, c_n$,

$$T \models c \iff \left( \exists_{\text{vars}(c)} c_1 \lor \cdots \lor \exists_{\text{vars}(c)} c_n \right)$$

implies that for some $i$, $T \models c \iff c_i$.

The following is a corollary of Theorem 4.4.

**Corollary 4.1.** Let $P$ be a CLP($\mathcal{E}$) program, $G$ be a goal, and $\mathcal{S}$ be a safe selection strategy. Let $T_{\mathcal{S}}$ have independence of constraints. If

$$P, T_{\mathcal{S}} \models c \rightarrow G$$

for constraint $c$, then $G$ has an answer $A$ using $\mathcal{S}$ such that

$$P, T_{\mathcal{S}} \models c \rightarrow A.$$

The constraint theory $\text{Real}_r$ does not have independence of constraints, witness Example 4.2. The constraint theory $\text{Term}_r$ does have independence of constraints as long as there are an infinite number of function symbols. Thus in the case of Prolog, any logical answer will be covered by a single qualified answer.

**SEMANTICS FOR FINITE FAILURE**

We now give an algebraic and a logical semantics for finite failure for CLP languages with dynamic scheduling. We first review some (more) definitions from the algebraic and logical semantics of CLP languages without delay [5]. Soundness of finite failure continues to hold, but only a weaker form of completeness holds.

When dealing with finite failure, a constraint logic program must be understood as representing its “Clark completion.” The Clark completion captures the reasonable assumption that the programmer really wants the rules defining a predicate to be an “if and only if” definition—the rules should cover all of the cases which make the predicate true.

**Definition 5.1.** The definition of $n$-ary predicate symbol $p$ in the program $P$, is the formula

$$\forall X_1 \ldots \forall X_n p(X_1, \ldots, X_n) \iff B_1 \lor \cdots \lor B_m,$$

where each $B_i$ corresponds to a rule in $P$ of the form

$$p(t_1, \ldots, t_n) :- L_1, \ldots, L_k$$

and $B_i$ is

$$\exists Y_1 \cdots \exists Y_j (X_1 = t_1 \land \cdots \land X_n = t_n \land L_1 \cdots \land L_k),$$
where \( Y_1, \ldots, Y_j \) are the variables in the original rule. Note that if there is no rule with head \( p \), then the definition of \( p \) is simply

\[
\forall X_1 \cdots \forall X_n p (X_1, \ldots, X_n) \leftrightarrow \text{false}
\]

as \( \lor \emptyset \) is naturally considered to be \text{false}.

The (Clark) completion \( P^* \) of a constraint logic program \( P \) is the conjunction of the definitions of the user-defined predicates in \( P \).

**Example 5.1.** The completion of the factorial program is

\[
\text{fac}(X_1, X_2) \leftrightarrow (X_1 = 0 \land X_2 = 1) \\
\lor \exists N \exists F (X_1 = N \land X_2 = N \ast F \land N \geq 1 \land \text{fac}(N - 1, F)).
\]

We note that the results for success given in the last section continue to hold if a program \( P \) is replaced by its completion \( P^* \).

The algebraic semantics of success for CLP languages is based on the least \( \models \)-model of the program \( P \). Dually, when considering the algebraic semantics of failure we are interested in the greatest \( \models \)-model of the program completion. In terms of failure, this is the most conservative \( \models \)-model of a program.

**Definition 5.2.** The greatest \( \models \)-model of the completion of a \( \text{CLP}(\mathcal{C}) \) program \( P \) is denoted by \( \text{gm}(P^*, \mathcal{C}) \).

It follows immediately from soundness of finite failure for languages without delay, that for languages with delay finite failure is sound. This is because if a goal finitely fails, it cannot have floundered derivations.

**Theorem 5.1 (Algebraic soundness of finite failure).** Let \( P \) be a \( \text{CLP}(\mathcal{C}) \) program. If goal \( G \) finitely fails with a safe scheduling strategy, then

\[
\text{gm}(P^*, \mathcal{C}) \models \neg \exists G.
\]

Completeness with respect to the algebraic semantics is more problematic. First, it is clear that we must require the solver to be complete; otherwise, derivations whose constraints are unsatisfiable will never be determined to have failed. The second restriction concerns fairness of the literal selection rule.

**Example 5.2.** Consider the goal \( r \) and the \( \text{CLP}(\text{Real}) \) program \( P \),

\[
r \leftarrow p(x) \land x = 2 \land q \\
p(x) \leftarrow x = 1 \\
q \leftarrow q
\]

in which \( p(x) \) always delays and \( r \) and \( q \) never delay. Clearly

\[
\text{gm}(P^*, \mathcal{C}) \models \neg \exists r.
\]

However, with any safe scheduling strategy, the goal \( r \) has an infinite derivation

\[
\langle r, \text{true} \rangle \Rightarrow \langle p(x) \land x = 2 \land q, \text{true} \rangle \Rightarrow \langle p(x) \land q, x = 2 \rangle \Rightarrow \langle p(x) \land q, x = 2 \rangle \\
\Rightarrow \ldots
\]
and so will not finitely fail. The problem is that the infinite derivation is not fair, in
the sense that the delayed atom $p(x)$ is never selected.

The example shows that for completeness we require a scheduling strategy
which is fair in the sense that in every infinite derivation, every literal is selected.

This definition of fairness is quite strong since it means that in a fair infinite
derivation there cannot be an atom which is delayed forever. Thus, there is a
conflict between fairness and safeness, and, as in Example 5.2, it may be impossible
to find a scheduling which is both fair and safe. Of course fairness does not imply
that no literals delay; in particular, a safe scheduling strategy is fair if it has a finite
derivation tree regardless of whether the tree contains floundered derivations.

In practice, most safe selection rules for programs are fair since one of the main
reasons for delaying literals is to not explore infinite derivations. Hence, such safe
selection rules lead to finite derivation trees, and so the selection rules are
inherently fair.

The third restriction is rather more technical. As in the case without delay, we
require that the program is canonical [4]. That is, the greatest fix point of the
immediate consequence operator, $T^\phi$, for the $CLP(\phi)$ program $P$, is $T^\phi \downarrow \omega$. This
technical condition is more fully described in [5] and is satisfied by almost all real
programs.

Given these restrictions we can prove algebraic completeness of finite failure.
The result is weaker than that for programs without dynamic scheduling, as goals
which are not satisfiable in the greatest $\phi$-model may still have successful deriva-
tions. In a sense, explicit failure can be masked by a goal which is floundered.

**Theorem 5.2** (Algebraic completeness of finite failure). Let $P$ be a canonical
$CLP(\phi)$ program and $G$ be a ground goal such that

$$\text{gm}(P^*, \phi) \models \neg \exists G.$$

If $G$ is evaluated with a complete solver and a safe and fair selection strategy $\mathcal{S}$,
then it will have a finite derivation tree. Furthermore, for each answer $A$ in the tree,

$$\text{gm}(P^*, \phi) \models \neg \exists A.$$

**Proof.** Let $\mathcal{S}'$ be a complete and fair (possibly unsafe) selection strategy which
extends $\mathcal{S}$. From the algebraic completeness of failure for CLP programs without
dynamic scheduling (Theorem 5 of [4]) we have that the derivation tree $T'$,
constructed for $G$ with $\mathcal{S}'$, is finitely failed. Using a similar argument to that in
the proof of Theorem 4.2, the derivation tree $T$, constructed for $G$ using $\mathcal{S}$, must
be a subtree of $T'$. Thus $T$ must be finite.

It follows from the logical soundness of success (Theorem 4.3) that if $A$ is an
answer to $G$, then $\text{gm}(P^*, \phi) \models A \rightarrow G$. Hence, $\text{gm}(P^*, \phi) \models \neg \exists A$. \qed

Soundness and completeness results for the logical semantics are very similar to
those for the algebraic semantics. The only difference is that the requirement for
canonicity can be dropped and the goal does not need to be ground. The proofs are
also similar to those for the algebraic semantics and rely on leveraging from the
results for CLP without dynamic scheduling [5].
Note that completeness of the solver implies the theory is satisfaction complete. That is, for all constraints \( c \) either \( T \models \exists c \) or \( T \models \neg \exists c \).

**Theorem 5.3** (Logical soundness of finite failure). Let \( P \) be a CLP(\( \mathcal{C} \)) program. If goal \( G \) finitely fails with a safe scheduling strategy, then

\[
P^*, T_{\mathcal{C}} \models \neg \exists G.
\]

**Theorem 5.4** (Logical completeness of finite failure). Let \( P \) be a CLP(\( \mathcal{C} \)) program and \( G \) be a goal such that

\[
P^*, T_{\mathcal{C}} \models \exists G.
\]

If \( G \) is evaluated with a complete solver and a safe and fair selection strategy \( \mathcal{S} \), then it will have a finite derivation tree. Furthermore, for each answer \( A \) in the tree,

\[
P^*, T_{\mathcal{C}} \models \exists A.
\]

6. CONCLUSION

We have given an algebraic and logical semantics for CLP languages with dynamic scheduling which accords with their operational semantics. The algebraic and logical semantics accord with those for CLP languages without dynamic scheduling, although the completeness result for finite failure is necessarily weaker. The results hold even for programs which lead to floundering derivations and rely on treating the delayed literals as part of the answer constraint.

REFERENCES