Semantics of Constraint Logic Programs with Optimization

Kim Marriott
Department of Computer Science
Monash University
Clayton 3168, Australia
marriott@cs.monash.edu.au

Peter J. Stuckey
Department of Computer Science
University of Melbourne
Parkville 3052, Australia
pjs@cs.mu.oz.au

Abstract

Many applications of constraint logic programming (CLP) languages require not only testing if a set of constraints is satisfiable, but also finding the optimal solution which satisfies them. Unfortunately, the standard declarative semantics for CLP languages does not consider optimization but only constraint satisfaction. Here we give a model theoretic semantics for optimization which is a simple extension of the standard semantics and a corresponding operational semantics which may be efficiently implemented.

Categories and Subject Descriptors: D.1.6 Logic Programming; F.4.1 Logic Programming; G.1.6 [Constrained Optimization.
General Terms: Languages, Theory
Additional Key Words and Phrases: constraint logic program, semantics.

1 Introduction

One of the most promising innovations in recent programming language design is the amalgamation of constraint programming and logic programming [8]. Constraints provide a powerful and natural programming paradigm, in which the objects of computation are not explicitly constructed but rather they are implicitly defined using constraints. Applications for constraint logic programming languages have been in many diverse areas. They include electrical circuit analysis [18], synthesis and diagnosis [6], options trading and financial planning [13]. Other applications are in traditional operations research problems, such as cutting stock and scheduling. Pure constraint logic programming languages only provide for testing constraint satisfaction, however, many applications desire an optimal solution. For this reason, although the standard semantics of CLP languages [8] does not include optimization operators, some existing CLP languages provide ad hoc non-logical optimization [3] and in other languages, optimization may be obtained using meta-level facilities [7]. We address the problem of giving a simple declarative semantics for optimization which has a corresponding operational semantics that allows efficient implementation.

Our main contribution is a model theoretic and operational semantics for optimization in CLP languages which is a simple extension of the usual semantics. Features of the operational semantics which allow it be given a corresponding declarative semantics are that the current best optimum is used to *prune* the search space, allowing the operational semantics to find optimums for subgoals which ostensibly have infinite derivations and optimization subgoals are not selected until the global variables are ground. A novel feature of the operational semantics is that optimization subgoals return *constraints* when the optimal value occurs over a range of points. We give soundness and completeness results for this operational semantics in terms of the 3-valued completion of the program by translating optimization to negation.

2 Example

A major application of CLP languages is for constraint satisfaction problems (CSPs). This requires constructing a set of constraints and determining whether the set is satisfiable, that is whether it has an answer. However, CSPs, contrary to their name, often include an optimization component. Not only must the answer satisfy the required constraints, it should also be the best such answer.

Consider the following CLP($\mathcal{R}$) program for determining the values of stock options (adapted from [13]). A "call" is a contract to allow the holder the option to buy a stock at some exercise price (X) at some later date if desired. Obviously if later the stock price (S) is greater than the exercise price the call has value (V) given by $S - X$. If the stock price is lower than the exercise price the call is worthless. A "put" is a contract to allow the holder to sell at the exercise price. The total value of a "call" or "put" must consider the cost to purchase the contract, and the interest on this cost between purchase and exercising the option.

```
call_value(S, X, V) :- S >= X, V = S-X.
call_value(S, X, V) :- S < X, V = 0.
put_value(S, X, V) :- S >= X, V = 0.
put_value(S, X, V) :- S < X, V = X-S.

value(call, Buy_or_Sell, Stockprice,Cost,InterestRate,ExercisePrice,Value) :-
    call_value(StockPrice, ExercisePrice, Option_Value),
    Value = Buy_or_Sell * (Option_Value - Cost * (1 + InterestRate)).
value(put, Buy_or_Sell,Stockprice,Cost,InterestRate,ExercisePrice,Value) :-
    put_value(StockPrice, ExercisePrice, Option_Value),
    Value = Buy_or_Sell * (Option_Value - Cost * (1 + InterestRate)).
```

The program can be used for answering many and varied questions relating values of stock options. For instance, the following query asks for the value of a combination of selling a call and a put option.

```
query1(Stockprice, Wealth) :-
    I = 0, X = 99, P = 10, C = 10, Sell = -1,
    value(put, Sell, Stockprice, P, I, X, Wealth1),
    value(call, Sell, Stockprice, C, I, X, Wealth2),
    Wealth = Wealth1 + Wealth2.
```

The answers are

```
Wealth = Stockprice - 79, 0 <= Stockprice, Stockprice < 99.
Wealth = -Stockprice + 119, 99 <= Stockprice
```

Constraint logic programs only provide for the capability of asking existential questions, however, many applications desire an optimal solution. An obvious query to the above program, given the nature of the problem, is to ask for the conditions that maximize wealth, that is minimize loss. We would like to be able to write a query of the form

```
?- min( query1(S, W), -W, [S, W], [MaxStock, MaxWealth]).
```

which is read as: find the values `MaxStock` and `MaxWealth` for the variables `S` and `W` that minimize the expression `-W` for the answers to the query `query1(S, W)`. This returns the answer constraint

```
MaxWealth = 20, MaxStock = 99.
```

For this reason, some existing CLP languages e.g. CHIP, provide ad hoc non-logical optimization and in other languages, optimization may be obtained using meta-level facilities [7]. In both cases the advantages of the simple declarative semantics of the CLP language are lost. We wish to provide language facilities for asking for optimal solutions which admit an efficient implementation, but also have a simple declarative semantics.

In the above case the maximum wealth is achieved at exactly one stock price, but for other option combinations the maximum may occur over a range. Rather than just returning one such position, or returning an infinite number of answers representing all such positions, we would like, in keeping with the spirit of constraint logic programming, to return constraints determining when the maximum occurs. The following goal, which asks for the maximum profit from a combination of two put and two call options,

```
?- min( query2(S, W), -W, [S, W], [MaxStock, MaxWealth]).
query2(Stockprice, Wealth) :-
    Buy = 1, Sell = -1, I = 0.1
    value(put, Buy, Stockprice, 10, I, 20, Wealth1),
    value(put, Sell, Stockprice, 18, I, 40, Wealth2),
    value(call, Sell, Stockprice, 15, I, 60, Wealth3),
    value(call, Buy, Stockprice, 10, I, 80, Wealth4),
    Wealth = Wealth1 + Wealth2 + Wealth3 + Wealth4.
```

returns the answer constraints

```
MaxWealth = 14.3, 40 <= MaxStock, MaxStock < 60.
MaxWealth = 14.3, MaxStock = 60.
```

representing all the stock prices where the maximum wealth is achieved. Note the two answers could be represented by one but they are computed in this manner because they arise from two distinct answer constraints for `query2`.

Existing constraint logic programming languages are ideally suited to answering satisfaction questions, this being the core of their operational behavior. However solving optimization problems with a CLP language is more problematic. Because of the importance of optimization some languages like CHIP [3] provide an ad hoc optimization facility which destroys the underlying declarative semantics. In the remainder of this paper, we show how to introduce a refinement of the optimization operator used above into the CLP language without compromising the usual model theoretic semantics.

3 A naive operational semantics for optimization

In this section we first give the usual operational semantics for CLP programs, and extend this in a simple way to include minimization. We show this simple extension has a number of problems.

3.1 Standard operational semantics of CLP

Predicates in a CLP program are divided into two classes: the *primitive constraints*, *Prim*, and the programmer-defined *atoms*, *Atom*. Primitive constraints are predefined in the sense that they have an intended meaning or interpretation which, for efficiency, is built into the solver for the language. For example the primitive constraints in CLP($\mathcal{R}$) are linear arithmetic equalities and inequalities over the real numbers and syntactic equalities over terms. We let $\mathcal{A}$ be the intended model for the primitive constraints.

For simplicity we require that atoms have the form $p(x_1, .., x_n)$ where the x_i are distinct variables. Primitive constraints, however, can have terms constructed from (pre-defined) function symbols as arguments. A *literal* is an atom or a primitive constraint. A *constraint* is a conjunction of primitive constraints. We do not differentiate two constraints that are logically equivalent.

We let $\overline{\exists}_S \pi$ be constraint π restricted to the variables in S. That is $\overline{\exists}_S \pi$ is $\exists\, V_1 \exists\, V_2 ... \exists\, V_n \pi$ where $\{V_1, V_2, ..., V_n\} = vars(\pi) \setminus vars(S)$ and the function $vars$ takes a syntactic object and returns the set of (free) variables occurring in it. We let $\tilde{\exists}$ and $\tilde{\forall}$ stand for existential and universal closure respectively. Hence $\tilde{\exists}\pi$ denotes whether π is satisfiable. The set of constraints is assumed to be closed under existential quantification.

A *(constraint logic) program* is a finite set of clauses of the form $H \leftarrow B$, where the *head*, H, is an atom and the *body*, B, is a sequence of literals. A *goal* is a (possibly empty) sequence of literals.

A *renaming* is a bijective mapping between variables. We naturally extend renamings to mappings between atoms, clauses, and constraints. Syntactic objects s and s' are said to be *renamings* if there is a renaming ρ such that $\rho(s) = s'$. The *definition of an atom A in program P with respect to variables W*, $defn_P(A, W)$, is the set of renamings of clauses in P such that each renaming has A as a head and has variables disjoint from $(W - vars(A))$.

The operational semantics of a program is in terms of "answers" to its "derivations" which are reduction sequences of "states" where a state is a tuple consisting of the current constraint, and the current literal sequence, or "goal".

A state $\langle \pi, G \rangle$ is *reduced* as follows. A fixed *selection rule* is used to choose a literal L in G.

1. If $L \in Prim$ and $\tilde{\exists}(L \wedge \pi)$, the state is reduced to $\langle L \wedge \pi, G \setminus L \rangle$;

2. If $L \in Atom$, the state is reduced to $\langle \pi, B :: G \setminus L \rangle$ where $(L \leftarrow B) \in defn_P(L, vars(G) \cup vars(\pi))$

Note that :: denotes concatenation of sequences and $\setminus$ deletes an element from a sequence.
A *derivation* of state s for program P is a sequence of states $s_0 \rightarrow \cdots \rightarrow s_n$ where $s = s_0$ and there is a reduction from s_i to s_{i+1}. A derivation is *successful* when the last state has an empty sequence of atoms. Let π be the constraint in the last state of a successful derivation from a state s. The constraint π restricted to the variables in s is said to be an *answer* to state s. We will denote the set of answers to state s by $answer(s)$. Informally we will talk about the answers to a goal G meaning the answers to the state $\langle true, G \rangle$. A state is *finitely evaluable* if it has no infinite derivation.

3.2 A naive operational semantics for optimization

For optimization to make sense there must be a partial order $\leq$ on the intended model $\mathcal{A}$ so we can compare elements. In the case of the reals it is just the usual arithmetic inequality. For technical convenience, we will assume that this ordering is total, thus if a minimum exists, it is unique, and that there are ground terms in the language representing the possible minimum values. We also extend the ordering to include two new elements $-\infty$ and $+\infty$ respectively smaller than and larger than every element of $\mathcal{A}$, thus ensuring that lubs and glbs always exist.

The point of optimization is to find "solutions" of a constraint set which are optimal with respect to the underlying ordering. More exactly, a mapping, α, from variables to the domain of $\mathcal{A}$ is a *solution* of constraint π if $\mathcal{A} \models \alpha(\pi)$. We let $soln(\pi)$ denote the set of solutions of π. The *greatest lower bound* of expression M in the context of constraint π, $glb(M, \pi) = \bigsqcap\{\alpha(M) \mid \alpha \in soln(\pi)\}$. The *minimum* of M wrt π is defined by $m = glb(M, \pi)$ if $\tilde{\exists}(M = m \wedge \pi)$, otherwise there is no minimum. Dually we can define the least upper bounds and maximums of M. However we will ignore maximal solutions because in the usual constraint domains we can rewrite maximization into minimization, and regardless the treatment is dual to that for minimization.

The syntax of CLP programs and goals is extended so as to allow optimization subgoals as literals in the body or goal. An *optimization subgoal* has the form

$$min(G, M, E, E')$$

where G is a goal and the other arguments are expressions. The intuitive reading of this subgoal is: find the value E' for expression E which gives a minimal value of M in the context of the answers to G.

The easiest way to incorporate optimization into the CLP operational semantics is to evaluate optimization subgoals whenever they are selected. The intuitive idea is to first find all of the answers to the state $\langle \pi, G \rangle$ where π is the current constraint. Then compute the minimum value m of M for the answers, considered as a disjunction of constraints and find the value e that E takes at this minimum. Finally, the constraint $E' = e$, reflecting the result of the minimization subgoal, is added to the current constraint.

Unfortunately, there is a problem with this intuitive idea. The problem is that for a particular minimum m there may be (possibly infinitely) many different values for e. A solution to this problem, in keeping with the design philosophy of CLP languages, is to return a disjunction of constraints representing these possible different values. As long as there are only finitely many answers from the original query compatible with the minimum value, a finite disjunction suffices to represent the different values for e. These considerations give rise to the following definitions:

The (partial) function $minimize(\pi, L)$ where L is the minimization subgoal $min(G, M, E, E')$, where $\langle \pi, G \rangle$ is finitely evaluable, is defined by

$$minimize(\pi, L) = \{\bar{\exists}_{E'} \pi' \wedge M = m \wedge E' = E \mid \pi' \in \Pi\}/\{false\}$$

where $\Pi = answers(\langle \pi, G \rangle)$ and $m = glb(M, \bigvee \Pi)$, unless $m = -\infty$ or $m = +\infty$ in which case $minimize(\pi, L) = \emptyset$. Note that if m is not a minimum wrt Π, then the above definition makes $minimize(\pi, L) = \emptyset$ because $\pi' \wedge M = m$ is unsatisfiable for all $\pi' \in \Pi$. The case $m = -\infty$ corresponds to $\langle \pi, G \rangle$ having an answer that does not constrain M below, and $m = +\infty$ to where $\langle \pi, G \rangle$ has no answers.

Example 3.1 Consider the following program

```
g(X, Y, Z) :- X < 1, Y >= 0, Z = Y + 1.
g(X, Y, Z) :- Y ≥ 2, Z ≥ X + Y.
g(X, Y, Z) :- Y ≥ X + 2, Z ≥ 1.
```

If $L = min(g(X, Y, Z), Y, f(X, Y, Z), V)$ and $\pi = \{X = 2\}$ then

$$\Pi = \{\{X = 2, Y \geq 2, Z \geq X + Y\}, \{X = 2, Y \geq X + 2, Z \geq 1\}\}$$

$m = glb(Y, \Pi) = 2$, and $minimize(\pi, L) = \{\{V = f(2, 2, U), U \geq 4\}\}$. □

The operational semantics is modified by adding the following reduction step:
When the selected literal L from G is a minimization subgoal, the state $\langle \pi, G \rangle$ can be reduced to $\langle \pi' \wedge \pi, G \setminus L \rangle$ where $\pi' \in minimize(\pi, L)$.
Note that if the goal G' in $L = min(G', M, E, E')$ has no answers then $\langle \pi, G \rangle$ cannot be reduced. Conceptually a sub-process is spawned to compute all answers to G'. If this sub-process does not terminate as G' has an infinite derivation then the original derivation will not terminate and is considered to have an infinite derivation.

Unfortunately, this simple operational semantics has a number of undesirable properties. First, the answers are now dependent on the order of literal evaluation.

Example 3.2 Consider the program P,

```
p(Y) ← 1 = Z, min(q(X,Z),X,X,Y).
q(X,Z) ← 0 ≤ X, 2*Z ≤ X.
```

With a left-to-right selection rule, the goal $p(Y)$ has the single answer $Y = 2$. However with a right-to-left selection rule the same goal has the single answer $Y = 0$. □

Second, it does not give answers to programs and goals which have an intuitively obvious meaning.

Example 3.3 Evaluation of the goal $p(Y)$ with the program

```
p(Y) ← min(q(X),X,X,Y),
q(X) ← 0 ≤ X.
q(X) ← 1 ≤ X, q(X).
```

will not terminate. However, it is intuitively clear that the correct answer should be $Y = 0$. Similarly if the constraint in the second clause is changed to `0 < X` then there is clearly no minimum, so the goal should finitely fail. □

It is therefore very difficult to find a model theoretic semantic basis for optimization with this simple operational semantics. For this reason we now develop a slightly more complex operational semantics for which there is a model theoretic basis.

4 A safe operational semantics for optimization

The problem with Example 3.2 is that variables affecting the optimization result are *further constrained* after the optimization as this means that if the optimization is performed later in the execution it gives a different result. The problem with Example 3.3 is that derivations which cannot possibly lead to a smaller minimum should be pruned – especially if they are infinite!

These motivate the following three changes. First, we add an argument to an optimization subgoal which contains the set of variables which are "local" to the minimization problem. These variables are implicitly universally quantified. For convenience we assume that they are disjoint from other variables in the clause. Thus an optimization subgoal now has the form

$$min(W, G, M, E, E')$$

where W is the set of local variables, G is a goal and the other arguments are expressions. The intuitive reading of this subgoal is: find the value E' for expression E which gives a minimal value for M in the context of the answers to G. The minimum is taken over all values of the variables in W.

This allows us to clearly distinguish between the local variables which cannot be constrained after the optimization and the other, "global", variables in the optimization which can be constrained outside of the optimization.

Second, we modify the operational semantics for optimization so that it uses a "safe" selection rule. The selection rule is safe in the sense that an optimization subgoal is only selected when we can be sure that subsequent execution cannot change the result of the optimization. This is true whenever the global variables have a unique value.

More precisely, let L be the minimization subgoal $min(W, G, M, E, E')$. Define the Boolean function *delay* by

$$delay(\pi, L) \Leftrightarrow (\exists\, V \in (vars(G) \cup vars(M) \cup vars(E)) \setminus W)\ \neg\, ground(\pi, V)$$

where $ground(\pi, V)$ holds when constraint π forces V to take a unique value. Thus $delay(\pi, L)$ holds if the result of evaluating L in the context of π could be changed by adding constraints over the global variables. A selection rule is *safe* if it never selects a minimization subgoal L in state $\langle \pi, G\rangle$ when $delay(\pi, L)$ holds.

The third modification is to change the evaluation of the optimization subgoal so that derivations are ignored if they cannot lead to a smaller minimum value. The following function captures this intuition.

The (partial) function $minimize^*(\pi, L)$ where L is the minimization subgoal $min(W, G, M, E, E')$ is defined by

$$minimize^*(\pi, L) = \{\overline{\exists}_{E'} \pi' \wedge M = m \wedge E' = E \ \mid\ \pi' \in \Pi\}/\{false\}$$

where $\Pi = answers(\langle \pi, G\rangle)$, and $m = glb(M, \bigvee \Pi)$, $m \neq -\infty$ and $m \neq +\infty$, and $\langle \pi \wedge M < m, G\rangle$ is finitely evaluable. If $m = -\infty$ then $minimize^*(\pi, L) = \emptyset$. If $m = +\infty$ then $minimize^*(\pi, L) = \emptyset$ if $\langle \pi, G\rangle$ is finitely evaluable. The restriction that $\langle \pi \wedge M < m, G\rangle$ is finitely evaluable is the weakest restriction that ensures that once we have found the minimum m the proof of its minimality is finite. In Section 6 we show how to efficiently implement $minimize^*$.

Putting these modifications together, we define the *safe operational semantics* for optimization as follows. This extends the usual operational semantics by allowing optimization subgoals. When the selected literal L from G is a minimization subgoal, the state $\langle \pi, G\rangle$ can be reduced to $\langle \pi' \wedge \pi, G \setminus L\rangle$ where $\pi' \in minimize^*(\pi, L)$. A safe selection rule must be used.

A derivation *flounders* when the last state has a non-empty sequence of delayed optimization subgoals or a derivation for some optimization subgoal flounders. A goal or state *flounders* if it has a derivation which flounders.

We now show how the safe operational semantics overcomes the difficulties associated with the naive semantics.

Example 4.1 Consider the programs P from Example 3.2. The variable X is a local variable in the minimization subgoal, therefore P is rewritten to,

```
p(Y) ← 1 = Z, min({X},q(X,Z),X,X,Y).
q(X,Z) ← 0 ≤ X, 2*Z ≤ X.
```

Using the safe operational semantics, the goal $p(Y)$ has the single answer $Y = 2$. This is because the minimization subgoal cannot be selected until the global variable Z has been constrained to the value 1. □

Example 4.2 Now consider the program from Example 3.3. It is rewritten into

```
p(Y) ← min({X},q(X),X,X,Y),
q(X) ← 0 ≤ X.
q(X) ← 1 ≤ X, q(X).
```

Evaluation of the goal $p(Y)$ with this program will now terminate with the desired answer $X = 0$ because once we have found the possible minimum using the first clause, derivations using the second clause will be pruned. Similarly if the constraint in the second clause becomes `0 < X` then it will terminate with failure. □

5 Declarative semantics

In this section we give a declarative semantics for minimization based on translating minimization subgoals into a subgoal with a negation. The usual semantics for programs with negation can then be used to give a declarative semantics for the resultant program. We first study the obvious translation which is based on the fact that M is the minimum value of X in the goal $p(X)$ iff $p(M)$ holds and there is no Z smaller than M for which $p(Z)$ holds. We then give a refinement of this translation which allows a stronger completeness result.

Definition 5.1 Let F be a formula possibly containing minimization subgoals. We define the *minimization translation* of F, written $mt(F)$, to be the formula obtained by replacing each minimization subgoal of the form $min(W, G, M, E, E')$ in F by the sub-formula

$$\exists M' (\exists W \ G \wedge M' = M \wedge E' = E) \wedge \neg(\exists W \ G \wedge M < M').$$

This simple translation captures that M' is the minimum of M for goal G if $M' = M$ and $E' = E$ are compatible with an answer to G and for each answer to G there is none compatible with a value less than M'. A similar translation is used in [5]. □

Example 5.2 Consider the following program

```
p(Y) :- min({X}, g(X), X, f(X), Y).
g(X) :- X >= 0.
```

The goal $p(Y)$ has the unique answer $Y = f(0)$. The minimization translation of the first clause is

```
p(Y) ← ∃ M (∃ X g(X), M = X, Y = f(X)), not (∃ X g(X), X < M).
```

Substituting $X \geq 0$ for $g(X)$ we obtain $Y = f(M) \wedge M \geq 0 \wedge \neg(0 < M) \leftrightarrow M = 0 \wedge Y = f(0)$ as expected. □

The usual declarative semantics of a logic program with negation is given by the three valued models of the program's Clark completion [12]. The Clark completion [2] captures the reasonable assumption that the predicate is fully defined by the clauses in the program and so an "if-and-only-if" definition of the predicate can be obtained by combining these. The reason why three valued logic is used rather than the more usual two valued logic is because of non-termination. A goal which succeeds has the truth value *true*, a goal which fails is has the truth value *false*, and a goal which does not terminate has the truth value *undefined*. Following [19] we extend these notions to constraint logic programs.

Definition 5.3 The *completion*, $comp(P, \mathcal{A})$, of a program P over domain $\mathcal{A}$ is defined as follows. If

$$\begin{array}{lcl} p(\tilde{x}) & \leftarrow & B_1 \\ & \vdots & \\ p(\tilde{x}) & \leftarrow & B_n \end{array}$$

are the rules in P defining p, where $B_1, \ldots B_n$ are assumed to share no variables except $\tilde{x}$, then P^* contains

$$\forall\, \tilde{x}\;\; p(\tilde{x}) \leftrightarrow B_1 \vee \cdots \vee B_n$$

If there are no rules defining p then P^* contains $\forall\, \tilde{x} \neg p(\tilde{x})$. Let $th(\mathcal{A})$ be the theory of $\mathcal{A}$, that is all first order sentences true in $\mathcal{A}$. Then $comp(P, \mathcal{A})$ is $P^* \wedge th(\mathcal{A})$. □

Note we assume Kleene's strong three valued interpretation [11] of the connectives, except for $\leftrightarrow$ which is given Łukasiewicz's interpretation, that is $x \leftrightarrow y$ is true iff x and y have the same truth value (including both undefined), and false otherwise. Combining the minimization translation with the completion gives rise to a simple declarative semantics for programs with minimization.

Definition 5.4 The *declarative semantics* of a program P containing minimization subgoals with constraints over $\mathcal{A}$ is the three-valued consequences of the theory $comp(mt(P), \mathcal{A})$. Thus we write

$$comp(mt(P), \mathcal{A}) \models F$$

if F holds in every three valued model of $comp(mt(P), \mathcal{A})$.
□

We wish to relate the operational semantics to the declarative semantics given by the completion. The following theorem and corollaries relates the two for the queries of most interest to a programmer, namely those that terminate.

Theorem 5.5 (Soundness and Completeness of Minimization) *Let P be a program and L be a minimization subgoal. If state $\langle \pi, L \rangle$ is finitely evaluable and non-floundering then*

$$comp(mt(P), \mathcal{A}) \models \pi \wedge mt(L) \leftrightarrow \bigvee \{\pi' \mid \pi' \in minimize^*(\pi, L)\}$$

Proof: (Sketch) By induction on the depth of minimization subgoals and simultaneously proving the following intermediate result: If $\langle \pi, G \rangle$ is a state and $\{\langle \pi_1, G_1 \rangle, \ldots, \langle \pi_n, G_n \rangle\}$ is the (possibly empty) set of states (modulo variable renaming) that $\langle \pi, G \rangle$ can be reduced to in one step then

$$comp(mt(P), \mathcal{A}) \models \pi \wedge G \leftrightarrow \bigvee_{i=1}^{n} (\pi_i \wedge G_i)$$

□

Corollary 5.6 (Soundness and Completeness for Finitely Evaluable Goals) *Let $\langle \pi, G \rangle$ be a non-floundering finitely evaluable state with answers $\pi_1, ..., \pi_n$. Then,*

$$comp(mt(P), \mathcal{A}) \models \pi \wedge mt(G) \leftrightarrow \pi_1 \vee ... \vee \pi_n. \square$$

As another corollary of the above proof we have a strong soundness result. Because any evaluated minimization subgoals in a successful derivation are obviously non-floundering and finitely evaluable, any successful derivation returns a correct answer. Similarly for goals G where every derivation is finitely failed. Formally

Corollary 5.7 (Soundness of Success and Finite Failure) *If G has a successful (non-floundering) derivation with answer constraint π then $comp(mt(P), \mathcal{A}) \models \tilde{\forall} \pi \rightarrow mt(G)$. If every derivation for G is finitely failed then $comp(mt(P), \mathcal{A}) \models \neg \tilde{\exists} mt(G)$.* □

We would also like a completeness result for goals which are not finitely evaluable. If we use constructive negation to implement optimization then completeness with respect to the three-valued consequences of $comp(mt(P), \mathcal{A})$ is immediate [19]. However, even in the case of goals which do not flounder, no such completeness result holds for the more efficient operational semantics considered here. There are two problems.

The first problem arises because completeness of finite failure requires the use of a "fair" selection rule. A selection rule is *fair* if, in any infinite derivation, no subgoal remains unselected forever. However, safeness may force the selection rule to be unfair. This is most easily illustrated using an example involving negation. The usual operational semantics for negation, *safe SLDNF*, is safe in the sense that only ground negative literals may be selected. Thus it can also suffer from interaction between fairness and safeness.

Example 5.8 Consider the following program and goal $t(X), \neg r(X)$.

```
t(1).
t(X) :- X ≥ 1, t(X - 1).
r(X) :- X ≥ 0.
```

Clearly the goal has no answers since $t(X) \rightarrow X \geq 1$ and $\neg r(X) \rightarrow X < 0$. But even though the safe SLDNF derivation tree for the goal is flounder-free it is not finitely failed. This is because the negative goal $\neg r(X)$ can only be selected after an answer for $t(X)$ is found. □

A similar problem can occur with minimization subgoals. Consider the execution of the goal

```
?- Z < 0, min({X,Y}, (X = 1, t(Y)), X, Y, Z).
```

Using the definition of $minimize^*$ there are an infinite number of minimal answers to the goal ($Z = 1, Z = 2, \ldots$), each of which is incompatible with other constraints in the goal. However the safe operational semantics does not finitely fail. To overcome this lack of fairness we must make a small change to the operational semantics.

The *complete operational semantics* is the same as the safe operationals semantics except that it uses the following rule for reducing minimization subgoals. When the selected literal $L = min(W, G', M, E, E')$ from G is a minimization subgoal, the state $\langle \pi, G \rangle$ can be reduced to $\langle M = m \wedge E = E' \wedge \pi, G' :: G \setminus L \rangle$ where $\Pi = answers(\langle \pi, G' \rangle)$, and $m = glb(M, \bigvee \Pi)$, $m \neq -\infty$, $m \neq +\infty$, and $\langle \pi \wedge M < m, G \rangle$ is finitely evaluable.

The complete operational semantics essentially behaves the same as the safe operational semantics except when the minimization subgoal has an infinite number of answers. However it is less efficient because of repeated computation in the calculation of m and the execution of $\langle M = m \wedge \pi, G' \rangle$.

The second problem arises because in the operational semantics constraints that the environment π places on E' are not applied until the minimization subgoal returns. If it does not terminate then these constraints are not considered even though they may be incompatible with the derivations causing non-termination.

Example 5.9 Let P be the program:

```
p :- Y ≥ 2, min({X}, g(X), X, X, Y).
g(X) :- X ≤ 1, g(X).
```

Consider the derivation for goal p, clearly the minimization subgoal runs forever. The completion of $mt(P)$ is just

```
p ↔ Y ≥ 2, ∃ M (∃ X g(X), M = X, Y = X), ¬ (∃ X g(X), X < M)
g(X) ↔ X ≤ 1, g(X).
```

Clearly $comp(mt(P), \mathcal{R}) \models p \leftrightarrow false$ since $Y \geq 2, X = Y, g(X)$ is false. Hence our operational semantics is incomplete. □

What the second example illustrates is that our simple translation of minimization into negation does not agree with the operational semantics. To obtain a completeness result for non-terminating computations, we require a more complex translation which makes the translation of a minimization subgoal *undefined* whenever the subgoal is not finitely evaluable. The following translation achieves this.

Definition 5.10 Let F be a formula possibly containing minimization subgoals. We define the *complex minimization translation* of F, written $cmt(F)$, to be the formula obtained by replacing each minimization subgoal of the form $min(W, G, M, E, E')$ in F by the sub-formula

$$[\exists E''\ mt(min(W, G, M, E, E''))] \quad \wedge \quad \langle[\exists E''\ mt(min(W, G, M, E, E''))] \rightarrow mt(min(W, G, M, E, E'))\rangle.$$

□

The reason that this translation gives the desired behavior is that if $[\exists E''\ mt(min(W, G, M, E, E''))]$ has truth value *undefined* then so does the whole sub-formula, otherwise the sub-formula is equivalent to $mt(min(W, G, M, E, E'))$, our original translation. Thus the preceding theorems of soundness and completeness for finitely evaluable sub-goals still hold for this new translation because in such cases the translations are equivalent.

Examining the program P from the previous example, because

$$\exists E''\, \exists M(\exists X\ g(X), X = M, X = E''), \neg(\exists X\ g(X), X < M)$$

is undefined, $comp(mtc(P), \mathcal{A})$ makes p undefined rather than false.

Given the new translation for $min(W, G, M, E, E')$ and assuming evaluation with the complete operational semantics we can now give the desired completeness result.

Theorem 5.11 (Completeness of Success and Finite Failure) *If P is a program over constraint domain $\mathcal{A}$ and there exists a safe and fair selection rule for which all derivations of $\pi \wedge G$ for P are flounder-free, then*
(a) if $comp(cmt(P), \mathcal{A}) \models \tilde{\forall} cmt(\pi \wedge G)$ then $\pi \wedge G$ has successful derivations with answers $\pi_1, \ldots \pi_n$ such that $\mathcal{A} \models \pi \leftrightarrow \pi_1 \vee \cdots \vee \pi_n$, and
(b) if $comp(cmt(P), \mathcal{A}) \models \neg\tilde{\exists} cmt(\pi \wedge G)$ then every (fair) derivation for G is finitely failed. □

Proof: (Sketch) The proof relies on results from constructive negation [19] which show $comp(cmt(P), \mathcal{A}) \models \tilde{\forall} cmt(\pi \wedge G)$ iff using constructive negation the goal $cmt(\pi \wedge G)$ is totally successful, i.e. has answer constraints $\pi_1, \ldots, \pi_n$ such that $\mathcal{A} \models \pi \leftrightarrow \pi_1 \vee \cdots \vee \pi_n$ and $comp(cmt(P), \mathcal{A}) \models \neg\tilde{\exists} cmt(\pi \wedge G)$ iff using constructive negation the goal $cmt(\pi \wedge G)$ is finitely failed. The proof is by induction on the least depth of the breadth first derivation tree that makes $cmt(\pi \wedge G)$ totally successful or finitely failed. Essentially we prove that $cmt(\pi \wedge G)$ is totally successful (finitely failed) wrt $cmt(P)$ iff $\pi \wedge G$ is totally successful (resp. finitely failed) wrt P. □

6 Implementation

In this section we sketch how we have extended an existing CLP compiler so as to provide optimization. The compiler is for the language CLP($\mathcal{R}$) [9, 10] in which constraints are linear arithmetic equations and inequalities. Adding optimization is useful as it means that linear programming problems from operations research can be naturally expressed as simple programs.

There are three features to note about this implementation. The first feature is that by a simple modification to the CLP($\mathcal{R}$) solver for testing satisfiability of linear inequalities, we obtain a low level minimization function $min_{exp}(E, \pi)$ which computes the greatest lower bound of the arithmetic expression E for the current constraint store π or $-\infty$ if no such bound exists. The Simplex algorithm already employed in the solver for testing satisfaction can also be used for minimization.

The second feature of the implementation to note, is how we efficiently prune off useless derivations when computing the minimal value of an expression with respect to a goal. This is done by the function $min_{goal}(G, E, \pi)$ which uses a type of branch-and-bound algorithm, in which the current best value for the minimum is used to prune the search. The definition of min_{goal} makes use of the function $get\_first\_answer(G, \pi)$ which returns a tuple $\langle \pi', B \rangle$ such that π' is the constraint store after finding the first answer to the state $\langle \pi, G \rangle$ and B is the backtrack information. Subsequent answers to the goal G are found using $get\_next\_answer(B, \pi')$ which backtracks and finds a new answer to G and returns a tuple containing the new constraint store and new backtrack information. The procedure $add\_constraint(\pi', \pi)$ adds the constraint π' to the current store π in such a way that subsequent backtracking inside min_{goal} will not remove it.

$min_{goal}(G, E, \pi) =$
 $cmin$ `:=` $+\infty$;
 $\langle \pi, B \rangle$ `:=` $get\_first\_answer(G, \pi)$;
 `while (` $\pi \neq false$ `) do`
  `if` $min_{exp}(E, \pi) < cmin$ `then`
   $cmin$ `:=` $min_{exp}(E, \pi)$;
   `if` $cmin$ `=` $-\infty$ `then return(`$cmin$`);`
   $add\_constraint(E < cmin, \pi)$;
  $\langle \pi, B \rangle$ `:=` $get\_next\_answer(B, \pi)$;
 `return (`$cmin$`)`

The final feature of the implementation to note, is that, by using meta-level constructs in CLP($\mathcal{R}$) for projection and access to the current constraints in the store, it is possible to write the function $minimize^*(L, \pi)$ using the function $min_{goal}(G, E, \pi)$. In fact, in the current implementation, a program with optimization subgoals is translated into an equivalent program which contains meta-level calls and calls to the function $min_{exp}(E, \pi)$.

The current implementation compromises completeness in two ways. First it uses a left-to-right selection rule. This is unfair and if the leftmost literal is an optimization subgoal whose global variables are not ground there is a runtime error. Second it uses a depth-first traversal of the derivation tree. This means that it will not terminate when it encounters an infinite branch which cannot be pruned by the current estimate of $cmin$. These restrictions improve efficiency, and in practice are not severe, resembling those used in most Prolog implementations. Of course, the implementation is still sound.

7 Related Work

The declarative semantics is based on completion semantics developed for negation by Kunen [12] for logic programs, and extended to constraint logic programs by Stuckey [19]. Our operational semantics is related to that proposed by Naish [17] for negation and aggregation, in that optimization subgoals must delay until their global variables have a fixed value. Using the current best optimum to prune the search space is related to the operational semantics of optimization in 2LP [16] and CHIP's `minimize(G,M)` predicate [3].

Our proofs are based on expressing optimization in terms of negation. We emphasize that although negation is used to formalize optimization, the replacement of minimization with negation does not lead to an operationally feasible approach to minimization using safe SLDNF. The translation of the clause in Example 5.2 to a normal program gives

```
p(Y) :- g(X), X = M, f(X) = Y, not np(M).
np(M) :- g(X), X < M.
```

Safe SLDNF execution of the program flounders on the goal $p(Y)$ while our operational semantics succeeds. If we use constructive negation then the translation does lead to executable programs. Furthermore there is no need for a safe selection rule. Unfortunately few implementations of constructive negation exist and in general they have severe efficiency problems. For this reason the operational semantics we give is quite different to (and more efficient than) the usual operational semantics for constructive negation.

Deductive database researchers have studied aggregation, in particular minimization [5]. For example their goal `groupby(p(X,Y), [X], T = min(Y))` corresponds in intent to the minimization subgoal $min(Y, p(X,Y), Y, Y, T)$. Although apparently related, we cannot make use of their results in our setting because of the bottom-up operational paradigm used in deductive databases. Other research has considered building an interactive querying facility above the CLP program, which allows optimization queries [15]. However, in this setting optimization is not part of the language but rather sits on top of it.

We have recently been made aware of work by Fages [4]. He has independently suggested a number of related operational semantics for optimization in CLP languages. He also provides a declarative semantics based on a translation of minimization to negation, similar to *mt*. He does not give explicit soundness or completeness results for these operational semantics, but he seems to suggest that soundness and completeness follows immediately from results in constructive negation [19]. However our need for the more complex translation, *cmt*, suggests that completeness is more difficult to prove.

References

[1] Chan, D. An extension of constructive negation and its application in coroutining. *Procs. North American Conf. on Logic Programming 89* (Cleveland, October 1989) 477–493.

[2] Clark, K. Negation as failure. *Logic and Databases.* Gallaire H. and Minker J. (Eds.) Plenum Press, New York, 1978, 293–322.

[3] Dincbas, M., van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. The constraint logic programming language CHIP. *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88* (Tokyo, Japan, December 1988) 693–702.

[4] Fages, F. On the semantics of optimization predicates in CLP languages. Presented as a position paper at *First Workshop on Principles and Practice of Constraint Programming.* Newport, Rhode Island, April 1993.

[5] Ganguly, S., Greco, S., and Zaniolo, C. Minimum and maximum predicates in logic programming. *Proceedings of the Tenth Principles of Databases Symposium* (Denver, June 1991) 154–163.

[6] Gorlick, M.M., Kesselman, C.F., Marotta, D.A., and Parker, D.S. Mockingbird: a logical methodology for testing. *Journal of Logic Programming* 8 (1990) 95–119.

[7] Heintze, N.C., Michaylov, S., Stuckey, P.J. and Yap, R. On meta-programming in CLP($\mathcal{R}$). *Proc. North American Conference on Logic Programming* (Cleveland, October 1989) 52–68.

[8] Jaffar, J. and Lassez, J.-L. Constraint logic programming. *Proc. Fourteenth Ann. ACM Symp. Principles of Programming Languages* (San Francisco, California, 1987) 111-119.

[9] Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems* 14 (1992) 339–395.

[10] Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. An abstract machine for CLP($\mathcal{R}$). *Proc ACM SIGPLAN Conf. on Programming Language Design and Implementation* (San Francisco, June 1992) 128–139.

[11] Kleene, S.C. *Introduction to Metamathematics.* North-Holland, 1952.

[12] Kunen, K. Negation in logic programming. *Journal of Logic Programming* 4 (1987) 289–308.

[13] Lassez, C., McAloon, K., and Yap, R. Constraint logic programming and options trading. *IEEE Expert* 2 (1987) 42–50.

[14] Lloyd, J.W. *Foundations of Logic Programming* (2nd Ed.) Springer-Verlag, Berlin, 1987.

[15] Maher, M.J. and Stuckey, P.J. Expanding query power in constraint logic programming. *Procs. North American Conf. on Logic Programming 89* (Cleveland, October 1989) 20–36.

[16] McAloon, K. and Tretkoff, C. 2LP: A logic programming and linear programming system. Brooklyn College of CUNY, CIS department, Technical Report No. 1989-21, 1989.

[17] Naish, L. *Negation and Control in Prolog.* Lecture Notes in Computer Science 238. Springer Verlag, New York, 1986.

[18] Simonis, H. and Dincbas, M. Using an extended Prolog for digital circuit design. *IEEE International Workshop on AI Applications to CAD Systems for Electronics* (Munich, October 1987) 165–188.

[19] Stuckey, P.J. Constructive negation for constraint logic programming. *Procs. Fifth IEEE Logic in Computer Science Symposium* (Amsterdam, 1991) 328–341.