Negation and Constraint Logic Programming

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

Running head: Negation and CLP

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

Abstract

Almost all constraint logic programming systems include negation, yet nowhere has a sound operational model for negation in CLP been discussed. The SLDNF approach of only allowing ground negative subgoals to execute is very restrictive in constraint logic programming where most variables appearing in a derivation never become ground. By describing a scheme for constructive negation in constraint logic programming we give a sound and complete operational model for negation in these languages.

Constructive negation was first formulated for logic programming in the Herbrand Universe and involves introducing disequality constraints. Constraint logic programming thus provides a much more natural framework for describing constructive negation. In this paper we describe a framework for constructive negation for constraint logic programming over arbitrary structures which is sound and complete with respect to the three-valued consequences of the completion of a program. Through this description, we arrive at a simpler, more efficient form of constructive negation for the Herbrand Universe. Finally we characterize what makes a structure particularly suited to the use of constructive negation and show this suitability condition for a number of structures and classes of structures.

The constraint viewpoint of constraint logic programming is well matched with constructive negation. Not only is constructive negation easier to understand from this point of view, but more importantly it gives the first clean approach to negation in constraint logic programming.

List of Symbols

$\neg$	negation	$\wedge$	conjunction	$\vee$	disjunction
$\leftarrow$	leftarrow	$\rightarrow$	rightarrow	$\leftrightarrow$	leftrightarrow
$\exists$	exists	$\forall$	forall		
$\bigvee$	big disjunction	$\bigwedge$	big conjunction	$\Leftrightarrow$	double leftrightarrow
$=$	equals	$\neq$	not equals	$>$	greater than
$<$	less than	$\geq$	greater than or equal to	$\leq$	less than or equal to
$+$	plus	$*$	times	$-$	minus
$\mathbf{t}$	bold t	$\mathbf{u}$	bold u	$\mathbf{f}$	bold f
$\mathbf{\vert}$	vertical separator	$\uparrow$	uparrow	$\downarrow$	downarrow
$\mathcal{A}$	calligraphic A	$\mathcal{L}$	calligraphic L	$\mathcal{R}$	calligraphic R
$\mathcal{HU}$	calligraphic HU	$\ll$	double left angle brackets	$\gg$	double right angle brackets
ω	omega	Φ	capital Phi	$\Re$	reals
α	alpha	β	beta	γ	gamma

1 Introduction

The traditional approach to negation in logic programming is the negation as failure rule, that is, a goal is false if we can prove that it cannot be proved. [Clark(1978)] showed how negation as failure can be seen as ordinary first order negation for the completed version of a program, in the case where negation does not occur in the body of clauses. Negation as failure is firmly entrenched as negation in logic programming because it can be efficiently implemented, using subderivations. Unfortunately, unrestricted implementations of negation as failure have severe drawbacks, they are incomplete and worse unsound. Consider the usual PROLOG implementation of `not`.

```
not(G) :- call(G), !, fail.
not(G).
```

Consider the program and goal

```
eq(X, X).
p :- not (eq(X, 1)), eq(X, 2).
?- p.
?- not(p).
```

Because of the cut in the definition of `not` the answer to the first query depends on the order in which sub-goals are selected. With the usual PROLOG left-right approach the goal fails where it should (logically) succeed. The second query demonstrates unsoundness as a consequence of the incompleteness of the first query. This problem with negation as failure occurs when the subderivation binds variables occurring in the negated goal, as in this example. The derivation of $eq(X, 1)$ proves that $eq(X, 1) \leftrightarrow X = 1$. Negation as failure incorrectly assumes $\forall X\ eq(X, 1)$ and thus concludes $\exists X\ \neg(eq(X, 1))$ is false.

Because of the above behaviour use of negation as failure is usually restricted to ensure soundness. The most common restriction is that a negative sub-goal cannot be selected unless the sub-goal is ground (SLDNF [Lloyd(1987)]). This prevents the problem above from occurring, but introduces another problem – *floundering*. Even when delaying methods [Naish(1986)] are used a goal may be unable to execute further when it consists only of non-ground negative goals.

The other "problem" with negation as failure is that a negative goal is not like a positive goal. Asking a positive goal we expect to have returned more than just a true/false answer, we expect computed answers. This facility is an important distinction between logic programming and other kinds of theorem proving. But negative goals can only return true/false. They are unable to construct answers even if they do exist. A simple example of this is the following double negation program

```
r(a).
q(X) :- not(r(X)).
p(X) :- not(q(X)).
```

Clearly `p(a)` is a consequence of the program, and as a goal it succeeds, but we cannot construct it as an answer to the goal `p(X)` which, under negation as failure, either flounders or succeeds with the empty substitution.

The aim of "constructive negation" is to extend the negation as failure rule to avoid floundering and to allow negative goals to construct answers. The first work related to constructive negation is that of [KUNEN(1987)]. He notes that answers to a queries in positive logic programs are a sequence of terms representing the sets of all there ground instances. If we call a set defined by all the ground instances of a term a *basic set*. Then the negative of a basic set or *cobasic set* is all ground terms that are not an instance of a particular term. He demonstrates that finite answers to logic programming queries can be represented by Boolean combinations of basic sets.

The term "constructive negation" was first introduced in [CHAN (1988)]. He gave an operational scheme for non-ground negative subgoals (called SLD-CNF). In essence SLD-CNF involves taking a negative goal, running the positive version and negating the answers. It is the negating of answers in such a way that we obtain usable information that is the key to his scheme. This constructive negation scheme inherits many of the advantages of negation as failure that make it so attractive from a programming point of view. It uses sub-derivations to determine negative goals, thus exploiting the efficiency of the underlying logic programming system, and handling "special" features of the language, e.g. cut.

More recently, [SATO AND MOTOYOSHI(1991)] defined a top-down interpreter for logic programs with negation, based on unfolding. They provide a completeness result with respect to Kunun's three-valued semantics [KUNEN(1987)] similar to that in this paper. The interpreter they describe revolves around a method of generalized quantifier elimination specific to the Herbrand Universe and does not appear readily extensible to other domains. In addition since it based on unfolding its breadth-first nature makes it appear even more difficult to implement efficiently than the approach described herein.

Negating answers to queries naturally introduces quantified disequations $\forall \mathbf{Z}\ s \neq t$. Consider the following program and goal:

```
p(X, f(X, Z))
?- ¬ p(X, Y)
```

The goal $p(X, Y)$ leads to the answer substitution $\{Y \leftarrow f(X, Z)\}$ which is equivalent to the equation $\{\exists Z\ Y = f(X, Z)\}$, restricting to the original variables X and Y. After negating this we obtain a disequation $\{\forall Z\ Y \neq f(X, Z)\}$.

Other schemes for constructive negation that do not introduce quantified negations apply only to specific kinds of programs. For instance [FOO *et al*(1988)] give a scheme for constructive negation for function-free programs (DATALOG). By deducing a finite *relevant type* for each predicate, they can ensure that a negative subgoal is always ground by simply grounding the variables appearing in the subgoal using predicates representing the deduced type. [WALLACE(1987)] also gives a scheme for function-free programs including (unquantified) disequations, where the number of constants is infinite.

Another approach [MAŁUSZYŃSKI AND NÄSLUND(1989)], perhaps the most "constructive", is aimed directly at allowing negative goals to return computed answers. A positive derivation for G returns γ such that $comp(P) \models \tilde{\forall} G\gamma$, this approach attempts to let a negative derivation for G return θ such that $comp(P) \models \tilde{\forall} \neg G\theta$. θ is called a fail substitution. Unfortunately [LASSEZ AND MARRIOTT(1987)] have shown that in general answers to negative queries cannot be represented by finite positive information alone, thus this approach must sometimes return an infinite number of fail substitutions. [DRABENT(1993)] proposes a similar approach to [MAŁUSZYŃSKI AND NÄSLUND(1989)] but allows quantified disequations as answers, combining the ideas of fail answers with constructive negation. His scheme has a form of completeness with respect to Kunen's semantics.

The schemes for constructive negation in logic programming can basically be divided into two classes, those assuming that the Herbrand Universe has an infinite number of functors ([Chan (1988), Wallace(1987), Sato and Motoyoshi(1991)]) and those assuming a finite number of functors ([Małuszyński and Näslund(1989), Foo *et al*(1988)]). The first group uses the infinite number of function symbols to simplify handling disequations. In this case each disequation is independent in the sense that we can determine whether a conjunction of equations and disequations are satisfiable by examining whether each disequation is individually satisfiable with respect to the equations. For this group Clark's Equality Theory (CET) [Clark(1978)] is a complete axiomatization [Maher(1989)]. The second group uses domain closure to derive positive information from "purely" negative information but are not as generally applicable as the first group. In this case CET must be extended with a domain closure axiom (see [Maher(1989)]) to arrive at a complete axiomatization. The work of [Drabent(1993)] encompasses both groups by using the appropriate complete axiomatization.

Since quantified disequations must appear in any scheme for constructive negation that is applicable to all logic programs there is no impediment to extending the syntax of bodies and goals to include disequations, and this is exactly what Chan describes. With this modification logic programs begin to look like constraint logic programs over the Herbrand Universe with constraints of the form $s = t$ and $\forall \mathbf{Z}\ s \neq t$. Similarly answers are conjunctions of these constraints. Thus we propose that the natural formulation of constructive negation is in terms of constraint logic programming.

Chan's approach to constructive negation is quite entrenched in the Herbrand Universe. Both SLD-CNF resolution and (of course) the constraint simplification procedure are specific to the Herbrand Universe. By characterizing constructive negation in the CLP framework we extend its applicability to many more languages, and in particular for the domain that Chan describes construct a more efficient procedure by treating both the equations and disequations similarly. More importantly our scheme provides a completeness result which Chan does not provide.

Almost all constraint logic programming systems include negation in the body, yet nowhere has a sound operational model for negation in CLP been discussed (except in [Maher(1987)] briefly). The SLDNF approach of only allowing ground negative sub-goals to execute is very restrictive in constraint logic programming where most variables appearing in a derivation never become ground. Indeed the structure of computation may not even include constants (so there are no "ground" objects). By describing a scheme for constructive negation in constraint logic programming we give a sound and complete operational model for negation in these languages. We obtain for constraint logic programs with negation the central theoretical results of logic programming, that is the correspondence of an operational, iterative and logical semantics.

The paper is organized as follows: in the next section we give a brief review of constraint logic programming. In Section 3 we define a our scheme for constructive negation in constraint logic programming, and in Section 4 we compare this approach with that of Chan [Chan(1989)] and Drabent [Drabent(1993)] in more detail. In Section 5 we discuss derivation trees for various computation rules, and relates these to Fittings three-valued consequence operator in Section 6. In Sections 7 and 8 we give the soundness and completeness results. Finally in Section 9 we discuss which structures are particularly suited for constructive negation.

2 A Brief Review of Constraint Logic Programming

A constraint logic programming [Jaffar and Lassez(1987)] language, CLP($\mathcal{A}$), exists in the context of a particular structure $\mathcal{A}$ which determines the meaning of the function and (constraint) relation symbols of some language $\mathcal{L}$. Constraints in the structure are relations upon terms of the structure. A *primitive constraint* takes the form $r(t_1, ..., t_n)$ where r is an n-ary relation symbol of $\mathcal{L}$, and t_i are terms of $\mathcal{L}$. For example, the following are primitive constraints in the domain of real numbers: $X + Y > 7$, $Z^2 - 3Z + 8 = 0$, $X \times Y \times Z = 3.14$. A *constraint* (or constraint formula) is any formula involving only primitive constraints, propositional connectives and quantifiers, in other words a first-order $\mathcal{L}$-formula. In Section 9 we discuss limiting constraints to simpler forms than arbitrary $\mathcal{L}$-formulae. We assume that $\mathcal{A}$ is a two-valued structure, e.g. every $\mathcal{L}$-sentence is either true or false in $\mathcal{A}$.

An *atom* takes the form $p(t_1, ..., t_n)$ where p is an n-ary (programmed) predicate symbol and t_i are terms. Constraint logic programs differ from logic programs by allowing constraints in bodies of rules and goals. A *constraint logic program* is thus a finite set of rules of the form

$$A \leftarrow c, B_1, \ \ldots, \ B_n$$

where A and B$_i$, $1 \leq i \leq n$, are atoms and c is a constraint.

A *(primitive) goal* takes the form

$$(c \mid B_1, ..., B_m)$$

where c is a constraint and $B_1, ..., B_m$ are atoms. For the purposes of derivations one can understand the goal $(c \mid B_1, \ldots, B_m)$ to represent the formula $c \wedge B_1 \wedge \cdots \wedge B_m$. In an abuse of terminology we shall sometimes refer to ground atoms and goals in $\mathcal{A}$ which are the result of assigning an element in $\mathcal{A}$ to each free variable in the atom or goal. Clearly in general ground atoms and goals are not expressible in the language $\mathcal{L}$.

A *computation rule* R selects a atom (or more generally a subgoal) for each goal G. Given a computation rule R that selects atom D_j from goal G of the form

$$(c \mid D_1, \ \ldots, D_j, \ldots \ , \ D_n)$$

then the children of G in the $(P, \mathcal{A}, R)$-derivation tree for G are given by the following:

- If D_j is an atom, there is a child,

  $$(c'' \mid D_1, \ \ldots, \ D_{j-1}, \ B_1, \ldots, B_m, \ D_{j+1}, \ldots, D_n).$$

  for each (standardized apart) clause

  $$A \leftarrow c', B_1, \ldots, B_m$$

  from P such that $c'' = c \wedge \{D_j = A\} \wedge c'$ is satisfiable in $\mathcal{A}$, where $\{D_j = A\}$ is a conjunction of constraints equating the arguments of atoms D_j and A.

A *$(P, \mathcal{A}, R)$-derivation of* G is a path on the $(P, \mathcal{A}, R)$-derivation tree of G. A $(P, \mathcal{A}, R)$-derivation is *successful* if it is finite and its last goal contains no atoms, i.e. the goal can be written in the form

$(c \,|\, \emptyset)$ where $\emptyset$ denotes the empty sequence of atoms. A $(P, \mathcal{A}, R)$-derivation is *finitely failed* if it is finite and not successful. All other $(P, \mathcal{A}, R)$-derivations are *infinite*. The *answer constraint* of a successful $(P, \mathcal{A}, R)$-derivation is the collected constraint c appearing in its last goal.

In what follows we use bold face letters, e.g. $\mathbf{X}$, to denote sets of variables (possibly empty), and tildes $\tilde{}$ to represent finite tuples of objects (whose type should be apparent from the context). We let $\exists \mathbf{W} F$ denote the formula $\exists v_1 \exists v_2 \cdots \exists v_n F$ where variable set $\mathbf{W} = \{v_1, \ldots, v_n\}$. We use the notation $\overline{\exists}_S$ to denote restriction to the variables in S. That is $\overline{\exists}_S F$ is $\exists \mathbf{W} F$ where $\mathbf{W} = vars(F) \setminus vars(S)$ and function $vars$ takes a syntactic object and returns the set of free variables occurring in it. Finally $\tilde{\exists}$ and $\tilde{\forall}$ are used to denote existential and universal closure respectively.

If P is a program with the following collection of clauses with (programmed) predicate p (of arity m) in the head

$$\begin{gathered} p(\tilde{t}_1) \leftarrow c_1 \wedge \tilde{B}_1 \\ \vdots \\ p(\tilde{t}_n) \leftarrow c_n \wedge \tilde{B}_n \end{gathered}$$

then the *completion* of P [Clark(1978)], $P*$ is the conjunction of formulae of the form

$$\forall \tilde{x}\; p(\tilde{x}) \quad \leftrightarrow \quad \begin{array}{l} \exists \mathbf{Y_1}\ (\tilde{x} = \tilde{t}_1 \wedge c_1 \wedge \tilde{B}_1) \quad \vee \\ \qquad\qquad \vdots \\ \exists \mathbf{Y_n}\ (\tilde{x} = \tilde{t}_n \wedge c_n \wedge \tilde{B}_n) \end{array}$$

where $\tilde{x}$ is a sequence of m new variables and $\mathbf{Y}_i$ are the variables in the i^{th} clause, and $\tilde{x} = \tilde{t}_i$ represents the conjunction of equations $x_1 = t_{i1} \wedge \cdots \wedge x_{m_i} = t_{im_i}$. If there is a predicate symbol, p, in P that does not appear in any head, then $P*$ contains the formula

$$\forall \tilde{x}\ \neg p(\tilde{x})$$

Note that $P*$ unlike $comp(P)$ does not contain CET.

Throughout the paper we assume the usual strong three-valued interpretation of the symbols $\wedge, \vee, \neg, \exists, \rightarrow$, but following [Kunen(1987)] use Łukaiewicz's truth table for the connective $\leftrightarrow$, that is $x \leftrightarrow y$ is $\mathbf{t}$ iff x and y are assigned the same truth value $\{\mathbf{t}, \mathbf{u}, \mathbf{f}\}$, and $\mathbf{f}$ otherwise. We denote three-valued consequence $\models_3$.

Crucial to our handling of negation will be the concept of a frontier of a derivation tree. A *$(P, \mathcal{A}, R)$-frontier* of a $(P, \mathcal{A}, R)$-derivation tree of G (or just a frontier of G) is a finite set of nodes in the tree, excluding the root, such that every derivation of G is either finitely failed or passes through exactly one *frontier node*. In figure 1, where $\Box$ indicates a successful goal and $\Diamond$ indicate a failed goal, $\{G_1, G_2, G_3\}, \{G_4, G_5, G_2, G_7\}$ and $\{G_4, G_6, G_3\}$ are all frontiers of the derivation tree shown for G_0, while $\{G_5, G_2, G_3\}$ is not.

In what follows we shall assume the computation rule R is extended to choose a (computable) frontier of a derivation-tree when desired. Clearly each frontier of G can be obtained by starting from the the root node and repeatedly applying the following step: choose a non-success node and replace it with its (possibly empty set of) children.

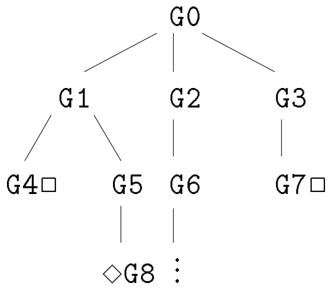

Figure 1: Derivation tree for G

Lemma 2.1 *If P is a definite program and $\{G_1, \ldots, G_n\}$ is a $(P, \mathcal{A}, R)$-frontier for G then*

$$\mathcal{A} \wedge P* \models_3 G \leftrightarrow (\exists \mathbf{Y_1}\ G_1) \vee \cdots \vee (\exists \mathbf{Y_n}\ G_n)$$

where $\mathbf{Y}_i$ is the set of variables in G_i not in G. $\square$

Effectively this means we can replace a goal G by a disjunction of goals appearing as the leaves of a partial derivation tree of G.

3 A Scheme for Constructive Negation

Once we allow negation in the body of rules we can simulate programs with bodies that are arbitrary first order formulae [Lloyd and Topor(1984)]. [Chan(1989)] extends the syntax of logic programs to allow bodies of this form. We do the same for constraint logic programs.

A *complex subgoal* is defined recursively as either

- an atom,
- or $\neg \exists \mathbf{Z}\ \ c, B_1, \ldots, B_n$ where $n \geq 1$, each B_i is a complex subgoal, and c is a constraint.

Complex subgoals include the cases where $\mathbf{Z}$ is empty and/or c is the constraint *true*, e.g. $\neg p(X, Y)$ is a complex subgoal. A *complex body* is a conjunction of complex subgoals. A *complex goal* is of the form $(c \mid B)$ where c is a constraint and B is a complex body. For the remainder of the paper we shall consider *normal programs*, which have complex bodies, and complex goals.

We need to define an operational mechanism for handling complex goals. We extend our definition of $(P, \mathcal{A}, R)$-derivation trees for complex goals informally as follows: If the current goal is $(c \mid \neg G)$, we first select some part c' of the constraint c that will be passed to the negative sub-derivation for G. The goal $(c' \mid G)$ is executed until it reaches some finite frontier, chosen by the computation rule R. The frontier nodes of the negative sub-derivation are gathered as a disjunction, and this formula is negated and simplified and the result converted to a disjunction of goals. Note that the negative sub-derivation itself may involve subsidiary negative sub-derivations.

More formally, derivations for complex goals are defined as follows. In a $(P, \mathcal{A}, R)$-derivation tree for goal G of the form

$$(c \mathbin{\|} D_1, \ \ldots, D_j, \ldots \ , \ D_n)$$

where D_j is the selected complex subgoal (by computation rule R), the children of G are given by:

- If D_j is an atom then the children of G are given by the standard definition.
- If D_j is $\neg\exists \mathbf{Z} \ \ c_q, \tilde{Q}$, where c_q is a constraint and $\tilde{Q}$ is a conjunction of complex subgoals, then let c' be a constraint such that $\mathcal{A} \models c \rightarrow c'$. Let the goal of the negative sub-derivation be $(c' \wedge c_q \mathbin{\|} \tilde{Q})$ and let $F \equiv \{(c' \wedge c_1 \mathbin{\|} B_1), \ldots, (c' \wedge c_m \mathbin{\|} B_m)\}$

  be a frontier (chosen by computation rule R) for $(c' \wedge c_q \mathbin{\|} \tilde{Q})$. (Because the constraint formula strictly grows by conjunction each node in the $(P, \mathcal{A}, R)$-derivation tree of $(c' \wedge c_q \mathbin{\|} \tilde{Q})$ includes the constraints c'). If F is empty then the (unique) child is the goal

  $$(c \mathbin{\|} D_1, \ \ldots, \ D_{j-1}, \ D_{j+1}, \ldots, D_n).$$

  Otherwise let c'_i, N_i be constraints and conjunctions of complex subgoals respectively such that

  $$\begin{array}{c}(c'_1 \wedge N_1) \vee \cdots \vee (c'_p \wedge N_p) \quad \leftrightarrow \\ \neg(\exists \mathbf{Z}, \mathbf{Y_1} \ c_1 \wedge B_1) \wedge \cdots \wedge \neg(\exists \mathbf{Z}, \mathbf{Y_m} \ c_m \wedge B_m)\end{array}$$

  where $\mathbf{Y}_k, 1 \leq i \leq m$ denote the variables in $c_k \wedge B_k$ which do not appear in $c \wedge c_q \wedge \tilde{Q}$, and each of the $c'_i \wedge N_i$ is a complex subgoal. There are children

  $$(c \wedge c'_i \mathbin{\|} D_1, \ \ldots, \ D_{j-1}, \ N_i, \ D_{j+1}, \ldots, D_n).$$

  for each $1 \leq i \leq p$ such that $c \wedge c'_i$ is satisfiable. If $p = 0$ there are no children.

There are a number of freedoms in the above scheme. The computation rule specifies the constraint c' passed to the negative sub-derivation and the frontier chosen. The most usual choice for c' is the complete formula c. This choice ensures the maximum amount of information is passed to the negative sub-derivation. This decreases the size of the negative sub-derivation, in general, since more derivations will fail. However the completeness results do not rely on any particular choice of constraints and frontiers.

How we obtain the $c'_i \wedge N_i$ from the formula

$$\neg(\exists \mathbf{Z}, \mathbf{Y_1} \ c_1 \wedge B_1) \wedge \cdots \wedge \neg(\exists \mathbf{Z}, \mathbf{Y_m} \ c_m \wedge B_m)$$

is a far more important question. To ensure completeness we must extract as much constraint information from the frontier of the negative sub-derivation as possible. In order to do this we separate out the constraints in the following way.

$$\neg\exists \mathbf{Y} \ c \wedge B \ \leftrightarrow \ (\neg\exists \mathbf{Y} \ c) \vee (\neg\exists \mathbf{Y} \ c \wedge B)$$

The first disjunct is a constraint formula, which holds all the constraint information of the negated subgoal. In practice we shall simplify this formula to less complex form (see Section 9). The second disjunct is a complex subgoal which we shall not attempt to simplify further at all. The disjunction $\bigvee c'_i \wedge N_i$ is obtained by taking the disjunctive normal form of the conjunction of the above formulae after simplification.

Example 3.1 Consider the following program and goal in the domain of linear arithmetic over reals:

```
p(X, Y) :- X > Z, Y <= 1 + Z, Z >= 0, q(Z).
p(X, Y) :- X < Z, Y <= 1 - Z, Z <= -2, r(Y, Z).
?- X <= 0, ¬∃ Y p(X, Y).
```

The negative sub-derivation has initial goal $(X \leq 0 \,\mathbf{|}\, p(X,Y))$ (assuming the entire constraint formula $X \leq 0$ is chosen to be passed in by the computation rule), and the frontier at depth one (the chosen frontier) of this sub-derivation consists of one node (since the other derivation fails):

$$(X \leq 0 \wedge X < Z \wedge Y \leq 1 - Z \wedge Z \leq -2 \,\mathbf{|}\, r(Y,Z))$$

Removing the input constraint `X` $\leq$ `0`, leaves us with a formula which is separated into two parts

$$\begin{array}{rl} & \neg\exists Y, Z\ X < Z \wedge Y \leq 1 - Z \wedge Z \leq -2 \\ \vee & \neg\exists Y, Z\ X < Z \wedge Y \leq 1 - Z \wedge Z \leq -2 \wedge r(Y,Z) \end{array}$$

The first part can be simplified by eliminating Y and Z to give $X \geq -2$. Thus the frontier at depth one for the original goal consists of the two nodes:

$$\begin{array}{c} (X \leq 0 \wedge X \geq -2) \\ (X \leq 0 \,\mathbf{|}\, \neg\exists Y, Z\ X < Z \wedge Y \leq 1 - Z \wedge Z \leq -2 \wedge r(Y,Z)) \end{array}$$

The next derivation step for the second node (if again we choose c' to be $X \leq 0$) involves the sub-derivation with initial goal:

$$(X \leq 0 \wedge X < Z \wedge Y \leq 1 - Z \wedge Z \leq -2 \,\mathbf{|}\, r(Y,Z))$$

In effect we restart the negative sub-derivation interrupted before (although this will not always be the case in general). □

Clearly negating a frontier

$$\neg(\exists \mathbf{Z}, \mathbf{Y_1}\ c_1 \wedge B_1) \wedge \cdots \wedge \neg(\exists \mathbf{Z}, \mathbf{Y_m}\ c_m \wedge B_m)$$

using the rule

$$\neg\exists \mathbf{Y}\ c \wedge B \ \leftrightarrow\ (\neg\exists \mathbf{Y}\ c) \vee (\neg\exists \mathbf{Y}\ c \wedge B)$$

to extract constraint information can easily lead to an exponential explosion of subgoals. In practice this can often be avoided by applying the Andorra principle in selection of negative subgoals. The Andorra principle proposed by D.H.D. Warren (see e.g. [Haridi and Brand(1988)]) suggests only performing non-deterministic computation steps when no deterministic steps are available. A (partial) derivation is deterministic if every goal in the derivation is the unique child of its parent.

To apply this to constructive negation we can select as frontier, the final goal in the longest deterministic derivation of a negative subgoal. (In some cases there is no deterministic computation for a negative subgoal and it should not be selected). In this case frontiers always consist of a single node and there is no explosion in size. Of course applying this rule rigidly leads to a form of floundering, where no subgoal can be selected. Hence we must relax the rule and allow non-singular frontiers to be chosen when they are the only possibility. Chan's suggestion of using delay declarations [CHAN(1989)] to control frontier selection can be used to partially implement the above methodology.

4 Comparison with other approaches

Chan's scheme [CHAN(1989)], since it is solely for the Herbrand Universe, concentrates more on the positive (or equational) information than the negative information. The constraint c' is chosen to be all the equations that appear in the collected constraint of the goal (which is always in the form of a conjunction of equations and disequations). This accords with a logic programming viewpoint that sees equations (or substitutions) as the heart of the computation, while disequations are supplementary. Consider the following program and goal

```
p(a,b,c).
p(b,a,c).
p(c,a,b).

?- X ≠ a, Z = c, ¬p(X, Y, Z).
```

Chan's scheme only passes the equational information $Z = c$ inside the sub-derivation for $p(X, Y, Z)$ (using the substitution $\{Z/c\}$). The constraint approach may pass both constraints, thus the sub-derivation goal is $(X \neq a, Z = c \mid p(X, Y, Z))$ which has one success $X \neq a, Z = c, X = b, Y = a$, rather than the two of Chan's approach. This reduces the number of disjuncts after negation. The advantage of our approach is that specializing the negative subgoal in general reduces the size of the frontier, and thus the number of children of the negative subgoal. In fact although Chan's scheme is described in this way, in the actual SEPIA implementation all constraints including disequalities are passed into sub-derivations.

The method for obtaining constraint information from the frontier of the negative sub-derivation is particular to the Herbrand Universe. It relies on the following property of the Herbrand Universe

$$\neg\exists \mathbf{Y}, \mathbf{Z}(x = s \wedge Q) \leftrightarrow \forall \mathbf{Y}(x \neq s) \vee \exists \mathbf{Y}(x = s \wedge \neg\exists \mathbf{Z}\ Q)$$

where x is a free variable in $\neg\exists \mathbf{Y}, \mathbf{Z}(x = s \wedge Q)$ and $\mathbf{Y}$ are the non-free variables in s. The simplification also relies on the assumption that the number of function symbols is infinite.

Using this property we can simplify the formula $\neg\exists \mathbf{Y}\ c \wedge B$ by moving existentially quantified variables that appear in equations in c outside the negation. In this approach variables in $\mathbf{Y}$ that appear in the complex subgoal B cannot be eliminated, and information from constraints involving these variables cannot be transported outside the negation symbol. For more details see [CHAN(1989)]. Because the processing of frontiers is done differently the form of answers given by Chan's scheme and that defined herein will be different, but they are comparable. When a goal has a finite number of answers under Chan's scheme then our approach will return a finite set of answers which are logically equivalent treating the sets as disjunctions.

Chan's approach provides more simplification in the sense that bodies B may be simplified during the process, as well as just constraints. In particular, where possible double negations can be eliminated from complex subgoals. The penalty in this approach is that when we are precluded from eliminating a variable because it appears in the body of a negated goal we lose some constraint information as well. In structures other than the Herbrand Universe the consequences of this are incompleteness. For example, consider the following program and goal in the domain of real numbers

```
p(X) :- X = T * T, q(T).
q(T) :- q(T).
r :- ¬ p(X).
?- r.
```

The goals of the negative sub-derivation are all of the form $(X = T * T \,\|\, q(T))$. This is because in Chan's approach extended to this domain we are unable to simplify the formula $\neg \exists T \; X = T * T \wedge q(T)$, since we are unable to eliminate the variable T. Thus there can be no successful derivation. But $\neg p(-1)$ and thus $\exists X \; \neg p(X)$ and r are consequences of the completion of the above program. Our scheme correctly finds the success. In the structure of the Herbrand Universe we have been unable to find any such examples using Chan's scheme which suggests that it may be complete (in the sense of Section 8).

Drabent's scheme [DRABENT(1993)] defines SLDFA resolution for Herbrand Universe domains. It can be viewed as a generalization of Chan's first approach [CHAN (1988)] in which only successful answers to a negative sub-derivation are used in constructing answers to the negative goal. Drabent extends Chan's approach in the following way: while [CHAN (1988)] is only applicable if the negative sub-derivation tree is finite, and in this case the frontier selected is always the success nodes themselves, [DRABENT(1993)] considers only selecting some finite subset of the success nodes.

The essential step in Drabent's method is as follows: the answers to a negative subgoal $(c \,\|\, \neg G)$ are over approximated by taking the success nodes $c_1, \ldots, c_n$ of some frontier to the goal $(true \,\|\, G)$ and constructing "answers" $d_1, \ldots d_k$ such that $c \rightarrow c_1 \vee \cdots \vee c_n \vee d_1 \vee \cdots \vee d_k$. The over approximations can be safely used in determining falsity of formulae. The goal $(c \,\|\, (\neg G), H)$ is false if each goal $(d_i \,\|\, H)$ is false.

Drabent shows soundness and completeness results for his approach with respect to Kunen's semantics. Because of the approximation steps the notion of a SLDFA tree is considerably complicated in this scheme, and the completeness results rely on considering all possible SLDFA trees for a particular goal. In contrast, for the scheme defined herein a single derivation tree captures all the information about a goal. In particular to obtain a sound and complete implementation of his scheme Drabent must include some form of iterative deepening or re-execution of negative subgoals to ensure that eventually an accurate enough approximation is obtained to ensure failure for all false goals.

5 Derivation Trees

In this section we define an important class of computation rules, the DEPTHFRONT computation rules, and from this a breadth-first computation rule that defines a canonical derivation tree for each goal $(c \mid G)$. First we define an important concept of derivation trees and discuss some restrictions on computation rules. For notational convenience during the remainder of the paper we shall use the term $\overline{\exists}$ to denote $\overline{\exists}_{(c \,\mid\, G)}$.

5.1 Total success

We define the notion of *total success*, the dual notion to finite failure. A goal $(c \mid G)$ has a *totally successful* $(P, \mathcal{A}, R)$-derivation tree if it includes successful derivations leading to a finite collection of nodes $(c \wedge c_1 \mid), \ldots, (c \wedge c_n \mid)$ such that

$$\mathcal{A} \models c \rightarrow \overline{\exists} c_1 \vee \cdots \vee \overline{\exists} c_n$$

In other words all "instances" of the goal also succeed. Total success is the natural notion of success for definite constraint logic programs on domains which do not satisfy the independence of negative constraints condition [MAHER(1993)]. Total success is more important when we consider negation since we negate entire frontiers, success must be defined over a frontier rather than a derivation.

Example 5.1 Consider the program

```
p(X) :- X >= 0, X < 3.
p(X) :- X >= 2.
p(X) :- X < 0, q(X).
```

Then $(X > 1 \mid p(X))$ is totally successful, since the two successful derivations "cover" the initial constraints $X > 1$. But note that $(X > 1 \mid p(X))$ has no (single) successful derivation leading to constraints c such that $X > 1 \rightarrow c$. $(X > 1 \mid \neg p(X))$ is finitely failed just because $(X > 1 \mid p(X))$ is totally successful. □

5.2 Fairness

We need to clarify the notion of *fairness* of a computation rule for the derivations we have defined. Consider the following program and goal

```
p(X) :- p(X).
q(b) :- p(b).
q(b) :- p(a).
?- X = a, ¬ (p(X), q(X)).
```

If the selected atom in the negative sub-derivation is $p(X)$ then the next goal in the top-level derivation is again $X = a \wedge \neg(p(X) \wedge q(X))$. Hence a computation rule, fair in the sense that it never ignores a top-most subgoal forever, never finds the success branch.

The correct definition of fairness for computation rules is that no subgoal (whether appearing within another subgoal or at the top-level) is ignored forever. More formally, in a derivation step at depth n in the derivation tree, we mark any newly introduced subgoals with the mark n (including those resulting from negative sub-derivations). A computation rule R is fair if no marked subgoal remains forever.

For the above example a fair derivation, where the underlined goals are selected in the negative sub-derivation, might be

$$\begin{array}{l}
(X = a \,\|\, \neg(\underline{p(X)^1} \wedge q(X)^1)^1) \\
(X = a \,\|\, \neg(p(X)^2 \wedge \underline{q(X)^1})^2) \\
(X = a \wedge X \neq b \,\|\,)
\end{array}$$

5.3 Consistency

Complex subgoals may be introduced into a derivation in two different ways: through their appearance in the body of a rule corresponding to a selected atom, or as a result of the negative sub-derivation for $(c \,\|\, \neg G)$ returning complex subgoals. The first manner is the essential *introduction* of complex subgoals.

Definition 5.2 A computation rule R is *consistent* if for each complex subgoal $\neg\exists\mathbf{Y}\ G$ introduced by a derivation step where an atom was selected, the same constraints c' are passed into the sub-derivation trees for G and all complex subgoals resulting from negative sub-derivations of G, and the subgoal selection for sub-derivation trees for G and all complex subgoals resulting from negative sub-derivations of G is exactly that of the derivation tree for $(c' \,\|\, G)$.

Consistency is really a restriction made for technical convenience, and does not appear truly necessary for the completeness results. The concept is more easily understood through an example

Example 5.3

```
p(X) :- X = a.
p(X) :- r(X).
r(b).
?- X ≠ c, ¬ (p(X), q(X)).
```

Given the R-derivation for $X \neq c, \neg(p(X), q(X))$ selects $X \neq c$ as the constraints to be passed in to the negative sub-derivation, then if the R-derivation for $X \neq c, p(X), q(X)$ selects $p(X)$ then the negative sub-R-derivation for $X \neq c, p(X), q(X)$ must also select $p(X)$. The resulting goals are

$$\begin{array}{c}
X \neq c, X \neq a, \neg(r(X), q(X)) \\
X \neq c, \neg(X = a, q(X)), \neg(r(X), q(X))
\end{array}$$

If the selected atom in the subgoal $X \neq c, r(X), q(X)$ in the R-derivation for $X \neq c, p(X), q(X)$ is $r(X)$, then $r(X)$ must be the selected atom in both negative sub-derivations arising when the subgoals $X \neq c, \neg(r(X), q(X))$ are selected. See Figure 2. $\Box$

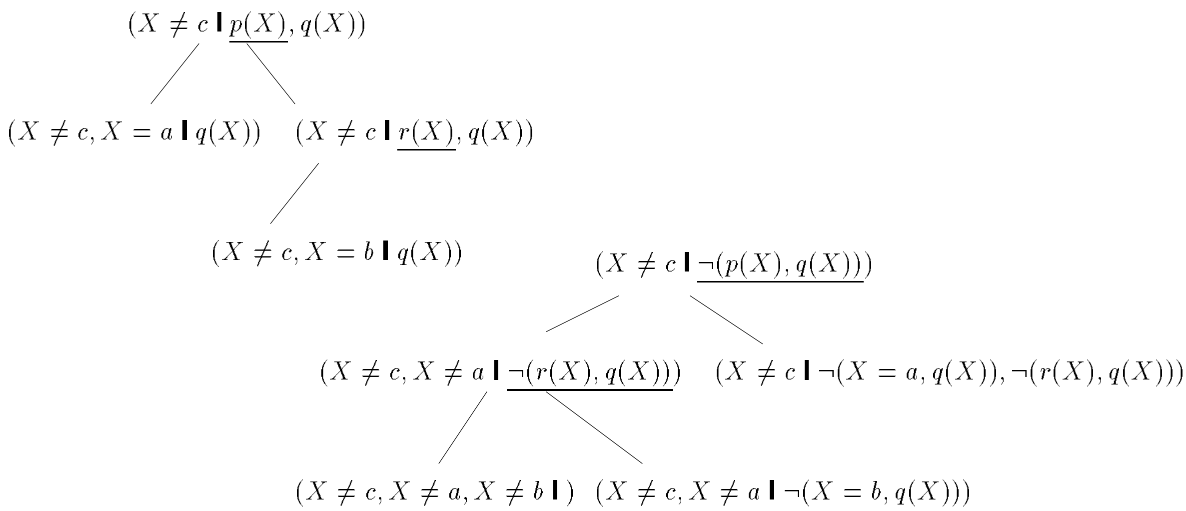

Figure 2: Consistent computation rules

Note that consistency is quite a strong restriction, in effect it means we cannot pass any more information into a negative sub-derivation beyond that passed in the first time it is selected. Consistent computation rules give us a strong correspondence between the derivation trees for $(c \,\|\, \neg G)$ and $(c' \,\|\, G)$, where c' are the constraints passed in to the negative sub-derivation. Every goal in the $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, \neg G)$ corresponds to a frontier of the $(P, \mathcal{A}, R)$-derivation tree for $(c' \,\|\, G)$ since it contains negated subgoals from the derivation tree for $(c' \,\|\, G)$ and constraints that arise from these subgoals. The correspondence is most clearly seen in a diagram. In Figure 3 the immediate children of the node $(c \,\|\, \neg G)$ all correspond to the frontier $\{(c' \wedge c_1 \,\|\, G_1), (c' \wedge c_2 \,\|\, G_2)\}$ and each includes either the constraints (e.g. $\neg\overline{\exists} c_1$) or negated subgoals (e.g. $\neg\overline{\exists}(c_1 \wedge G_1)$) for each node. Similarly the depth 2 nodes shown all correspond to the frontier $\{(c' \wedge c_1 \,\|\, G_1), (c' \wedge c_3 \,\|\, G_3), (c' \wedge c_4 \,\|\, G_4)\}$

This correspondence together with the fairness of the computation rule gives rise to the following result.

Lemma 5.4 (Correspondence Lemma) *Let R be a fair consistent computation rule. Let c' be the constraints passed in to the negative sub-derivation for $(c \,\|\, \neg G)$. Then, for each frontier F of the $(P, \mathcal{A}, R)$-derivation tree for $(c' \,\|\, G)$, in each $(P, \mathcal{A}, R)$-derivation for $(c \,\|\, \neg G)$ either the derivation is finitely failed or successful or at some stage in the derivation each negated subgoal appearing in the derivation is at or below the frontier F.*

Proof: We consider an infinite $(P, \mathcal{A}, R)$-derivation $(d_0 \,\|\, H_0), (d_1, H_1), \ldots$ for $(d_0 \,\|\, H_0) = (c \,\|\, \neg G)$ and show that for each frontier F of $(c' \,\|\, G)$ eventually a goal $(d_k \,\|\, H_k)$ is reached where each negated subgoal appearing in H_k is at or below F.

In the first derivation step a frontier F_1 of $(c' \,\|\, G)$ is chosen and each child of $(c \,\|\, \neg G)$ including (d_1, H_1) only contains negated subgoals corresponding to nodes appearing in F_1.

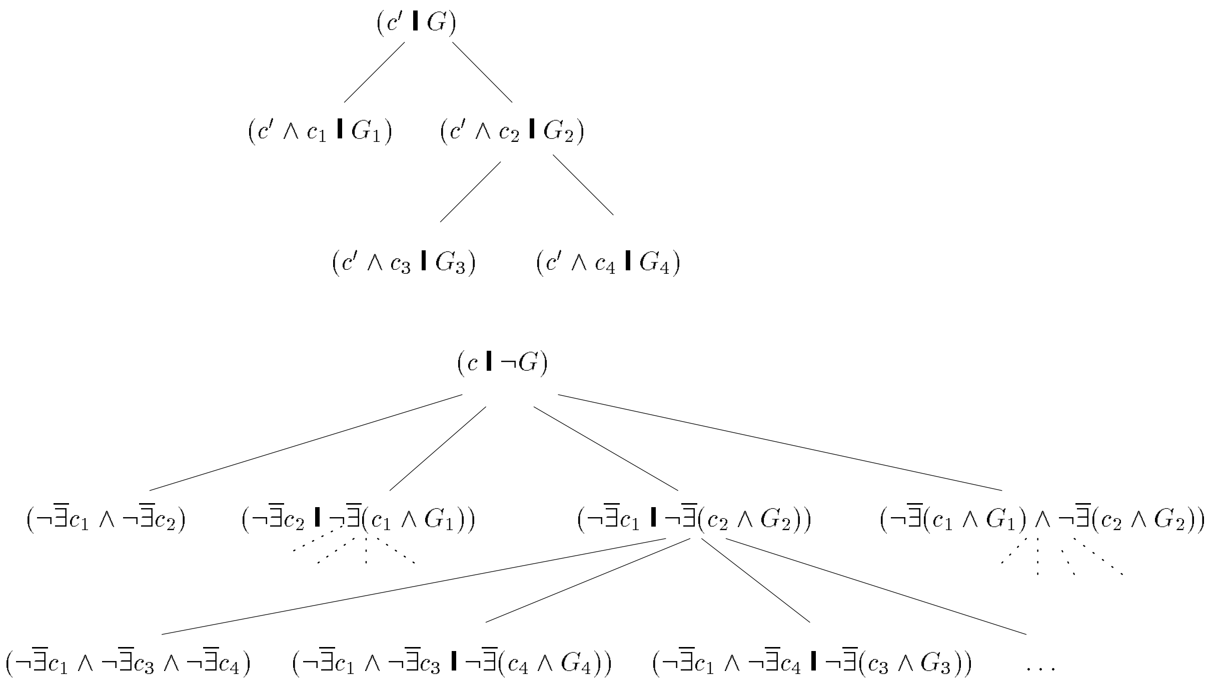

Figure 3: Correspondence between positive and negative derivation trees

In the i^{th} derivation step a negated subgoal $\neg\overline{\exists}(c_j \wedge G_j)$ of $(d_{i-1} \mid H_{i-1})$ corresponding to a node in the frontier F_{i-1} of $(c' \mid G)$ is selected. A frontier E_i of the goal $(c' \wedge c_j \mid G_j)$ is chosen by the computation rule, which is by the consistency restriction a sub-frontier of $(c' \mid G)$. The negated subgoal $\neg\overline{\exists}(c_j \wedge G_j)$ in $(d_{i-1} \mid H_{i-1})$ is replaced by constraints and negated subgoals corresponding to nodes in E_i to obtain $(d_i \mid H_i)$. Clearly the negated subgoals appearing in $(d_i \mid H_i)$ all correspond to nodes in the frontier of $(c' \mid G)$ of the form $F_i = F_{i-1} - \{(c' \wedge c_j \mid G_j)\} \cup E_i$.

Clearly the frontiers F_i are monotonically deepening, that is each frontier F_{i+1} only contains nodes at or below those in F_i. Consider an arbitrary frontier F of $(c' \mid G)$. If $(d_k \mid H_k)$ contains a negated subgoal corresponding to a node (in F_k) strictly above nodes in F then by the fairness of the computation rule eventually after some finite number of steps this negated subgoal is selected. The resulting goal $(d_{k'} \mid H_{k'})$ has corresponding frontier $F_{k'}$ which is strictly closer to F.

To completely formalize the notion of closer to we can define the distance metric $d(F, k)$ from goals $(d_k \mid H_k)$ to frontiers F as follows: the lexicographically ordered pair (maximum distance from a node in F to node above it in F_k corresponding to a negated subgoal in $(d_k \mid H_k)$, number of nodes in F at this maximum distance). Then applying the argument of the previous paragraph to a negated subgoal at maximum distance from F means $d(F, k') < d(F, k)$. □

5.4 DepthFront computation rules

A DepthFront (or DF) computation rule R always chooses the empty constraint $true$ to be passed into a negative sub-derivation (e.g. if the current goal is $(c \mid \neg G)$ the negative sub-derivation is for $(true \mid G)$), and always chooses the depth 1 frontier, that is the immediate children of the subgoal $(true \mid G)$. DF computation rules are important because they are simple. The Lemmas 5.6, 5.7 and 5.8 continue to hold for more general computation rules but are very difficult to prove directly.

Example 5.5 Consider the following program and goal:

```
p(X) :- X > 1, ¬∃ Y   q(X, Y).
q(X, Y) :- X = 2, Y = 4.
q(X, Y) :- X = 4, t(X, Y).
t(X, Y) :- Y ≠ 4, s(X).
?- X > 3, p(X).
```

The (partial) derivation trees shown in Figure 4 for the above program are DF-derivation trees. The negative goal is $(X > 3 \wedge X > 1 \mid \neg\exists Y\ q(X, Y)$, leads to the sub-derivation for subgoal $(true \mid q(X, Y))$. The immediate children of this goal $(X = 2 \wedge Y = 4 \mid), (X = 4 \mid t(X, Y))$ are negated to give the goals

$$(X \neq 2 \wedge X \neq 4 \mid),$$
$$(X \neq 2 \mid \neg\exists Y(X = 4 \wedge t(X, Y))$$

and these are used to form the children of the original negative goal. Similarly the immediate child of $(X = 4 \mid t(X, Y))$ forms the children of the goal $\neg\exists Y(X = 4 \wedge t(X, Y))$ when selected. □

$$(true \mid \underline{q(X,Y)})$$

$$(X = 2 \wedge Y = 4 \mid \,) \qquad (X = 4 \mid \underline{t(X,Y)})$$

$$(X = 4 \wedge Y \neq 4 \wedge X = Y \mid \underline{s(X)})$$

$$(X > 3 \mid \underline{p(X)})$$

$$(X > 3 \wedge X > 1 \mid \underline{\neg \exists Y \; q(X,Y)})$$

$$(X > 3 \wedge X > 1 \wedge X \neq 2 \wedge X \neq 4 \mid \,) \qquad (X > 3 \wedge X > 1 \wedge X \neq 2 \mid \underline{\neg \exists Y (X = 4 \wedge t(X,Y))})$$

$$(X > 3 \wedge X > 1 \wedge X \neq 2 \mid \,) \qquad (X > 3 \wedge X > 1 \wedge X \neq 2 \mid \neg \exists Y (X = 4 \wedge Y \neq 4 \wedge X = Y \wedge s(X)))$$

Figure 4: DF-derivation trees

The DF-derivation trees for goals $(c \,\|\, \neg G)$ and $(true \,\|\, \neg G)$ are closely related since they both obtain their goals from the negative sub-derivation of $(true \,\|\, G)$. We can obtain the DF-derivation tree for $(c \,\|\, \neg G)$ by adding constraints c to every node in the DF-derivation tree for $(true \,\|\, \neg G)$ and removing failed nodes. This leads to the following "lifting" property.

Lemma 5.6 (Lifting Lemma) *Let R be a fair consistent DF computation rule. (a) The $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, G)$ is totally successful iff there are successful derivations in the $(P, \mathcal{A}, R)$-derivation tree for $(true \,\|\, G)$ ending in constraints $c_1, \ldots, c_n$ such that $\mathcal{A} \models c \rightarrow \overline{\exists} c_1 \vee \cdots \vee \overline{\exists} c_n$. (b) The $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, G)$ is finitely failed iff there exists n such that every derivation in the $(P, \mathcal{A}, R)$-derivation tree for $(true \,\|\, G)$ reaches a node, $(c' \,\|\, G')$ within n steps such that $\mathcal{A} \models c \rightarrow \neg\overline{\exists} c'$.*

Proof: Note that the $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, G)$ and $(true \,\|\, G)$ differ only in the constraints at each node and the absence of branches in $(c \,\|\, G)$ which conflict with the constraints c. This is because the immediate children resulting from a derivation step where an atom is selected only differ in that some children (that have constraints incompatible with c) are removed. Similarly for a derivation step where a complex subgoal $\neg G$ is selected: the negative sub-derivation in both cases is for the goal $(true \,\|\, G)$ and the children differ only in removing the children incompatible with c.

Part (a) follows since the branches of the derivation tree leading to $(c_1 \,\|\,), \ldots, (c_n \,\|\,)$ are never removed by adding the constraints c since they are compatible with c. Part (b) follows similarly since the only extra branches in the derivation for $(true \,\|\, G)$ are those that are whose constraints c' are incompatible with c, that is $c \rightarrow \neg\overline{\exists} c'$. □

Furthermore, the derivation step for a complex subgoal in a DF-derivation tree is the same "size" as a derivation step for an atom. This gives us a strong correspondence between the derivation trees for $(c \,\|\, G)$ and $(c \,\|\, \neg G)$. This correspondence together with the above "lifting" property leads to the proof of the following results.

Lemma 5.7 *Let R be a fair consistent DF computation rule. The $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, G)$ is finitely failed iff the $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, \neg G)$ is totally successful.*

Proof: $(\Rightarrow)$ Since $(c \,\|\, G)$ is finitely failed then by Lemma 5.6 there exists a frontier $(c_1 \,\|\, G_1), \ldots, (c_n \,\|\, G_n)$ of the $(P, \mathcal{A}, R)$-derivation tree for $(true \,\|\, G)$ such that $c \rightarrow \neg\overline{\exists} c_i$ for all $1 \leq i \leq n$. There exists a derivation of $(true \,\|\, \neg G)$ which corresponds to a frontier at or below $(c_1 \,\|\, G_1), \ldots, (c_n \,\|\, G_n)$, being the one that obtains constraints only from success nodes. The only success nodes $(d_1 \,\|\,), \ldots (d_m \,\|\,)$ are at or below the above frontier hence since $c \rightarrow \neg\overline{\exists} d_j$ the derivation does not fail. Now examine the derivation extending this one that always obtains constraints. The constraints are always of the form $\neg\overline{\exists} e_k$ where $e_i \rightarrow c_i$. Hence $c \rightarrow \neg\overline{\exists} c_i \rightarrow \neg\overline{\exists} e_i$. Thus the derivation does not fail. Eventually by the fairness of R it is successful. Now $c \rightarrow \neg\overline{\exists} e_1 \wedge \cdots \wedge \neg\overline{\exists} e_l$ hence by Lemma 5.6 the derivation tree for $(c \,\|\, \neg G)$ is totally successful.

$(\Leftarrow)$ If $(c \,\|\, \neg G)$ is totally successful there exists successful derivations $(c \wedge \neg\overline{\exists} c_1^1 \wedge \cdots \wedge \neg\overline{\exists} c_1^{n_1} \,\|\,), \ldots (c \wedge \neg\overline{\exists} c_m^1 \wedge \cdots \wedge \neg\overline{\exists} c_m^{n_m} \,\|\,)$ such that $c \rightarrow \neg\overline{\exists} c_1^1 \wedge \cdots \wedge \neg\overline{\exists} c_1^{n_1} \vee \cdots \vee \neg\overline{\exists} c_m^1 \wedge \cdots \wedge \neg\overline{\exists} c_m^{n_m}$. Each successful derivation corresponds to a frontier $(c_i^1 \,\|\, G_i^1), \ldots, (c_i^{n_i} \,\|\, G_i^{n_i})$ of $(true \,\|\, G)$. If we take a frontier $\{(c_1' \,\|\, G_1), \ldots, (c_k' \,\|\, G_k)\}$ of $(true \,\|\, G)$ that is below all of the m frontiers above then then $c \rightarrow \neg\overline{\exists} c'_1 \wedge \cdots \wedge \neg\overline{\exists} c'_k$ and so $c \rightarrow \neg\overline{\exists} c'_j$ for all $1 \leq j \leq k$. Thus by Lemma 5.6 $(c \,\|\, G)$ is finitely failed. □

Lemma 5.8 *Let R be a fair consistent DF computation rule. The $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, G)$ is totally successful iff the $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, \neg G)$ is finitely failed.*

Proof: ($\Rightarrow$) Since $(c \,\|\, G)$ is totally successful by Lemma 5.6 there are successful derivations in $(true \,\|\, G)$ leading to $(c_1 \,\|\,), \ldots, (c_n \,\|\,)$ such that $c \rightarrow \overline{\exists} c_1 \vee \cdots \vee \overline{\exists} c_n$. Consider any derivation of $(c \,\|\, \neg G)$ either it is (a) finitely failed or (b) successful or (c) it reaches a node where each negated goal in the node is below a frontier including all the successful derivations above (by Lemma 5.4). In case (b) the derivation ends in a goal of the form $(c \wedge \neg\overline{\exists} d_1 \wedge \cdots \wedge \neg\overline{\exists} d_k \,\|\,)$ where $d_1, \ldots d_k$ represent a complete frontier of $(true \,\|\, G)$. Now for each $1 \leq i \leq n$ there exists d_j such that $\neg\overline{\exists} d_j \rightarrow \neg\overline{\exists} c_i$ and thus $c \wedge \neg\overline{\exists} d_1 \wedge \cdots \wedge \neg\overline{\exists} d_k$ is unsatisfiable, contradiction. For case (c) the derivation reaches a node $(c \wedge \neg\overline{\exists} d_1 \wedge \cdots \wedge \neg\overline{\exists} d_k \,\|\, G')$ such that the same conditions as in (b) hold. Thus all derivations of $(c \,\|\, \neg G)$ are finitely failed.

($\Leftarrow$) Given that $(c \,\|\, \neg G)$ is finitely failed consider the derivations which only gets constraints when there is no body. Constraints appearing in these derivations correspond to successful nodes $(c_1 \,\|\,), \ldots, (c_n \,\|\,)$ in the $(P, \mathcal{A}, R)$-derivation tree for $(true \,\|\, G)$. Now each set of constraints $c \wedge \neg\overline{\exists} c_1 \wedge \cdots \wedge \neg\overline{\exists} c_n$ is unsatisfiable, hence $(true \,\|\, G)$ has successful derivations leading to $(c_1 \,\|\,), \ldots, (c_n \,\|\,)$ such that $c \rightarrow \overline{\exists} c_1 \vee \cdots \vee \overline{\exists} c_n$ holds. Hence by Lemma 5.6 $(c \,\|\, G)$ is totally successful. □

Example 5.9 Reviewing Example 5.1, since $(X > 1 \,\|\, p(X))$ is totally successful we expect $(X > 1 \,\|\, \neg p(X))$ to be finitely failed. Negating the frontier

$$(X \geq 0, X < 3 \,\|\,), (X \geq 2 \,\|\,)$$

(ignoring the third goal for simplicity) we obtain

$$(X < 0 \wedge X < 2) \vee (X \geq 3 \wedge X < 2)$$

Clearly when we conjoin the constraint $X > 1$ with this formula, the result is unsatisfiable. Hence $(X > 1 \,\|\, \neg p(X))$ is finitely failed □

It is not difficult to extend the earlier definition for (SLD-) derivation trees to breadth-first (or even GLD [Wolfram *et al*(1984)]) derivation trees. Then we can define a *unique* breadth-first (*BF*) derivation tree for every goal $(c \,\|\, G)$. The BF computation rule is the DepthFront computation rule that selects every subgoal at each step. Lemmas 5.6, 5.7 and 5.8 continue to hold for BF computation rules.

BF derivation trees have even a stronger correspondence between the trees for $(c \,\|\, G)$ and $(c \,\|\, \neg G)$ since every complex subgoal appearing at depth k of the BF-derivation tree for $(c \,\|\, \neg G)$ is a goal appearing at depth k in the BF-derivation for $(true \,\|\, G)$. We define a BF-k-derivation tree for $(c \,\|\, G)$ by taking the BF-derivation tree for $(c \,\|\, G)$ to depth k (where 0 is the depth of the root). The trees in Figure 4 (as well as being DF-derivation trees) are the BF-2-derivation tree for $(true \,\|\, q(X, Y))$ and the BF-3-derivation tree for $(X > 3 \,\|\, p(X))$.

Before proceeding to show the correspondence of total success and finite failure between BF and DF derivation trees we present two technical Lemmas for one of the cases of the proof.

Lemma 5.10 *Let R be a fair consistent DF-computation rule. If the $(P,\mathcal{A},R)$-derivation tree for $(c \mid G_1 \wedge G_2)$ is finitely failed then either $(c \mid G_1)$ or $(c \mid G_2)$ has a finitely failed $(P,\mathcal{A},R)$-derivation tree or there exist d_1, d_2 such each that $(d_1 \mid G_1)$ and $(d_2 \mid G_2)$ have finitely failed $(P,\mathcal{A},R)$-derivation trees and $c \rightarrow d_1 \vee d_2$.*

Proof: The $(P,\mathcal{A},R)$-derivation tree for $(c \mid G_1 \wedge G_2)$ is finitely failed hence by Lemma 5.6 the leaves of some frontier the $(P,\mathcal{A},R)$-derivation tree for $(true \mid G_1 \wedge G_2)$ are all of the form $(c^1 \wedge c^2 \mid G)$ such that $c \rightarrow \neg\overline{\exists}_{(c \mid G_1 \wedge G_2)}(c^1 \wedge c^2)$, where c^1 are the constraints from G_1 and similarly for G_2. Now examining the $(P,\mathcal{A},R)$-derivation trees for $(true \mid G_1)$ and $(true \mid G_2)$ we see that the constraints appearing in leaves of these trees are $c^1_1, \ldots, c^1_{n_1}$ and $c^2_1, \ldots, c^2_{n_2}$ respectively, and each leaf in the $(P,\mathcal{A},R)$-derivation tree for $(true \mid G_1 \wedge G_2)$ is of the form $c^1_i \wedge c^2_j$, and each such pair exists or $c^1_i \wedge c^2_j$ is unsatisfiable. Now either there exists

$$d_1 \leftrightarrow \neg\overline{\exists}_{(c \mid G_1 \wedge G_2)} c^1_1 \wedge \cdots \wedge \neg\overline{\exists}_{(c \mid G_1 \wedge G_2)} c^1_{n_1}$$

or the right hand side is equivalent to false. Similarly for G_2. Now $c \rightarrow \neg\overline{\exists}_{(c \mid G_1 \wedge G_2)}(c^1_i \wedge c^2_j)$ for all $1 \leq i \leq n_1, 1 \leq j \leq n_2$. Hence since c^1_i and c^2_j only share variables in G_1 and G_2

$$c \rightarrow (\neg\overline{\exists}_{(c \mid G_1 \wedge G_2)} c^1_1 \wedge \cdots \wedge \neg\overline{\exists}_{(c \mid G_1 \wedge G_2)} c^1_{n_1}) \vee (\neg\overline{\exists}_{(c \mid G_1 \wedge G_2)} c^2_1 \wedge \cdots \wedge \neg\overline{\exists}_{(c \mid G_1 \wedge G_2)} c^2_{n_2})$$

If both of these conjunctions are false then we have $c \rightarrow false$ and a contradiction. If one of these conjunctions is equivalent to false (w.l.o.g say the second) then

$$c \rightarrow \neg\overline{\exists}_{(c \mid G_1 \wedge G_2)} c^1_1 \wedge \cdots \wedge \neg\overline{\exists}_{(c \mid G_1 \wedge G_2)} c^1_{n_1}$$

and by Lemma 5.6 $(c \mid G_1)$ is finitely failed. Otherwise

$$c \rightarrow d_1 \vee d_2$$

By construction $(d_1 \mid G_1)$ and $(d_2 \mid G_2)$ have finitely failed $(P,\mathcal{A},R)$-derivation trees using Lemma 5.6. □

The following lemma can be proved almost identically.

Lemma 5.11 *If the BF-k-derivation tree for $(c \mid G_1 \wedge G_2)$ is finitely failed then either $(c \mid G_1)$ or $(c \mid G_2)$ has a finitely failed BF-k-derivation tree or there exist d_1, d_2 such each that $(d_1 \mid G_1)$ and $(d_2 \mid G_2)$ have finitely failed BF-k-derivation trees and $c \rightarrow d_1 \vee d_2$.* □

We are now ready to show the relationship between BF derivation trees and DF derivation trees.

Lemma 5.12 *Let R be a fair consistent DF computation rule. $(c \mid G)$ has a totally successful (resp. finitely failed) BF-k-derivation tree for some finite k iff it has a totally successful (resp. finitely failed) $(P,\mathcal{A},R)$-derivation tree,*

Proof: The proof proceeds by induction on the $\omega.k + m$ where k is the least k such that there is a totally successful or finitely failed BF-k-derivation tree for $(c \mid G)$, and m is the number of connectives in G.

Base Case
If $k = 0$ and $m = 0$ then the only totally successful and finitely-failed zero length derivations are for constraints. Clearly the induction hypothesis holds.

Connectives Case
If $m > 0$ then the topmost symbol of G is $\neg, \exists, \wedge$. We treat each connective in turn.

Not
T: Let $(c \mid \neg G)$ be the goal in question. Then $(c \mid \neg G)$ has a totally successful BF-k-derivation tree iff $(c \mid G)$ has a finitely-failed BF-k-derivation tree by Lemma 5.7 iff, the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid G)$ is finitely failed using the induction hypothesis, iff the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid \neg G)$ is totally successful by Lemma 5.7.

F: Let $(c \mid \neg G)$ be the goal in question. Then $(c \mid \neg G)$ has a finitely-failed BF-k-derivation-tree iff $(c \mid G)$ has a totally successful BF-k-derivation tree by Lemma 5.8, iff the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid G)$ is totally successful by the induction hypothesis, iff the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid \neg G)$ is finitely failed by Lemma 5.8.

Exists
T: Recall that the derivation trees for $(c \mid \exists y G)$ and $(c \mid G)$ are identical in form. Let $(c \mid \exists y G)$ be the goal in question. The BF-k-derivation tree for $(c \mid \exists y G)$ is totally successful iff the $(true \mid \exists y G)$ leads to successful goals $(c_1 \mid), \ldots, (c_n \mid)$ such that $c \rightarrow \exists y \overline{\exists} c_1 \vee \cdots \vee \exists y \overline{\exists} c_n$ by Lemma 5.6, iff the BF-k-derivation tree for each $(c \wedge \overline{\exists} c_i \mid G)$ is totally successful by Lemma 5.6, iff the $(P, \mathcal{A}, R)$-derivation tree for each goal $(c \wedge \overline{\exists} c_i \mid G)$ is totally successful by the induction hypothesis, iff the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid \exists y G)$ by two more applications of Lemma 5.6.

F: Recall that the derivation trees for $(c \mid \exists y G)$ and $(c \mid G)$ are identical in form. The BF-k-derivation tree for $(c \mid \exists y G)$ is finitely failed iff the BF-k-derivation tree for $(c \mid G)$ is finitely failed, iff the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid G)$ is finitely failed, by the induction hypothesis, iff the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid \exists y G)$ is finitely failed

And
T: Let $(c \mid G_1 \wedge G_2)$ be the goal in question. Then the BF-k-derivation trees for $(c \mid G_1)$ and $(c \mid G_2)$ are totally successful. By induction the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid G_1)$ is totally successful. That is there are successful derivations with final constraints $c \wedge c_1^1, \ldots, c \wedge c_n^1$, where $c \rightarrow \overline{\exists}_{(c \mid G_1)} c_1^1 \vee \cdots \vee \overline{\exists}_{(c \mid G_1)} c_n^1 \leftrightarrow \overline{\exists}_{(c \mid G_1 \wedge G_2)} c_1^1 \vee \cdots \vee \overline{\exists}_{(c \mid G_1 \wedge G_2)} c_n^1$ since c_i^1 only contains new variables and variables from $(c \mid G_1)$. Similarly for $(c \mid G_2)$. Thus the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid G_1 \wedge G_2)$ is totally successful since the constraints occurring on derivations corresponding to the totally successful derivations of $(c \mid G_1)$ and $(c \mid G_2)$ are exactly $(\overline{\exists}_{(c \mid G_1 \wedge G_2)} c_1^1 \vee \cdots \overline{\exists}_{(c \mid G_1 \wedge G_2)} c_n^1) \wedge (\overline{\exists}_{(c \mid G_1 \wedge G_2)} c_1^2 \vee \cdots \vee \overline{\exists}_{(c \mid G_1 \wedge G_2)} c_m^2)$ (since c_i^1 and

C_j^2 only share variables in $(c \,\|\, G_1 \wedge G_2))$ which is clearly implied by c. The reverse direction follows similarly.

F: By Lemma 5.10 if $(c \,\|\, G_1 \wedge G_2)$ has a finitely failed BF-k-derivation tree then either $(c \,\|\, G_1)$ has a finitely failed BF-k-derivation tree or $(c \,\|\, G_2)$ has a finitely failed BF-k-derivation tree or there exist c_1 and c_2 such that $(c_1 \,\|\, G_1)$ and $(c_2 \,\|\, G_2)$ have finitely failed BF-k-derivation trees and $c \rightarrow c_1 \vee c_2$.

For the first two cases $(c \,\|\, G_1 \wedge G_2)$ has a finitely failed $(P, \mathcal{A}, R)$-derivation tree from the induction hypothesis and the fact that R is fair. For the remaining case, by induction the $(P, \mathcal{A}, R)$-derivation tree for $(c_1 \,\|\, G_1)$ is finitely failed. By Lemma 5.6 every derivation in the $(P, \mathcal{A}, R)$-derivation tree for $(true \,\|\, G_1)$ leads to a node $(c_1' \,\|\, G_1')$ such that $c_1 \rightarrow \neg\overline{\exists}_{(true \,\|\, G_1)} c_1' \leftrightarrow \neg\overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_1'$ because c_1' only includes variables from G_1 and new variables. Similarly for $(c_2 \,\|\, G_2)$. Consider the $(P, \mathcal{A}, R)$-derivation tree for $(true \,\|\, G_1 \wedge G_2)$ on each derivation by the fairness of R eventually we have constraints $c_1' \wedge c_2'$ such that $c \rightarrow c_1 \vee c_2 \rightarrow \neg\overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_1' \vee \neg\overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_2' \rightarrow \neg\overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} (c_1' \wedge c_2')$. Hence by Lemma 5.6 the $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, G_1 \wedge G_2)$ is finitely failed. The reverse direction is similar.

Limit Ordinal

$m = 0$ hence the goal is of the form $(c \,\|\, c')$ or $(c \,\|\, p(\tilde{x}))$. For constraints see the base case.

T: Suppose $(c \,\|\, p(\tilde{x}))$ has a totally successful BF-$k+1$-derivation tree. Then there exist renamed clauses of P of the form

$$p(\tilde{x}) \leftarrow B_i$$

and c_i such that $(c_i \,\|\, B_i)$ has a totally successful BF-k-derivation tree and $c \rightarrow c_1 \vee \cdots \vee c_n$. By induction the $(P, \mathcal{A}, R)$-derivation trees for $(c_i \,\|\, B_i)$ are totally successful and hence the $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, p(\tilde{x}))$ is totally successful. The reverse direction is similar.

F: Suppose $(c \,\|\, p(\tilde{x}))$ has a finitely failed BF-$k+1$-derivation tree. Then for each renamed clause in P of the form

$$p(\tilde{x}) \leftarrow B_i$$

the BF-k-derivation tree for $(c \,\|\, B_i)$ is finitely failed and by induction so is the $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, B_i)$. Combining these we have that the $(P, \mathcal{A}, R)$-derivation tree for $(c \,\|\, G)$ is finitely failed. The reverse direction is similar. □

Intuitively extending the results to more general computation rules follows from two observations. First, passing of constraints c' into sub-derivations only serves to simplify the derivation trees, hence any DF-derivation tree that is totally successful or finitely failed will maintain this property if we change the computation rule to pass in constraints. In the reverse direction the additional derivations that result from removing constraints c' can all be seen to be subsumed by other derivations. Second, any frontier selection can be imitated by a number of DF derivation steps. The formal steps to justify this intuition are somewhat laborious and technical and are left to the Appendix. The result of interest is

Corollary A.5 Let R be a DF-computation rule, let R' be any fair consistent computation rule. $(c \,\|\, G)$ has a totally successful (resp. finitely failed) $(P, \mathcal{A}, R)$-derivation tree iff it has a totally successful (resp. finitely failed) $(P, \mathcal{A}, R')$-derivation tree. □

6 Three-valued consequence

The main technical results of this paper are soundness and completeness of our scheme with respect to (a) $\Phi_P \uparrow \omega$, where Φ_P is the semantic operator of Fitting described in [Fitting(1985)] and extended to constraint logic programs in [Fitting and Ben-Jacob(1988)] (see also [Kunen(1989)]), and (b) the three-valued consequences of the program completion. In this section we reproduce the definition of the operator Φ_P of Fitting and define a "non-ground" version of Φ_P, denoted $\Phi_P^{\mathcal{A}}$ (somewhat similar to $\hat{I}_n$ defined in [Kunen(1987)]). We conclude with the crucial connection between derivation trees and $\Phi_P^{\mathcal{A}}$.

6.1 Φ_P

Definition 6.1 A *partial interpretation* I is a mapping from ground atoms to $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$, where we read $\mathbf{t}$ as truth, $\mathbf{f}$ as falsehood, and $\mathbf{u}$ as undefined.

We extend a partial interpretation to ground constraints in the natural manner $I(c) = \mathbf{t}$ if $\mathcal{A} \models c$ and $I(c) = \mathbf{f}$ if $\mathcal{A} \models \neg c$. We extend a partial interpretation to arbitrary ground goals using the usual strong three-valued interpretation of the symbols $\neg, \wedge, \exists$. For example $I(\neg G) = \mathbf{f}$ if $I(G) = \mathbf{t}$, $I(\neg G) = \mathbf{t}$ if $I(G) = \mathbf{f}$ and $I(\neg G) = \mathbf{u}$ otherwise.

Definition 6.2 Define the *immediate consequence* function Φ_P from partial interpretations to partial interpretations as follows:

- $\Phi_P(I)(A) = \mathbf{t}$ if there exists clause in P
$$p(\tilde{s}) \leftarrow B$$
and valuation α such that $A = p(\tilde{s})\alpha$ and $I(B\alpha) = \mathbf{t}$.
- $\Phi_P(I)(A) = \mathbf{f}$ if for each clause in P of the form
$$p(\tilde{s}_i) \leftarrow B_i$$
and valuation α such that $A = p(\tilde{s}_i)\alpha$, then $I(B_i\alpha) = \mathbf{f}$.

Definition 6.3 Let I_0 be the partial interpretation that maps everything except goals involving only constraints and connectives to undefined.

$$\begin{array}{rcl} \Phi_P \uparrow 0 & = & I_0 \\ \Phi_P \uparrow (k+1) & = & \Phi_P(\Phi_P \uparrow k) \\ \Phi_P \uparrow \alpha & = & sup\{\Phi_P \uparrow \lambda | \lambda < \alpha\} \\ & & \text{when } \alpha \text{ is a limit ordinal} \end{array}$$

The *Fitting semantics* of a normal program P is defined as the least fixed point of Φ_P, $lfp(\Phi_P)$. Unfortunately the operator Φ_P is not continuous (since in some ways its corresponds to computing both $T_P \uparrow \alpha$ and $T_P \downarrow \alpha$) and hence its least fixpoint may occur at any recursive ordinal. However, the natural cut off point for computability is after ω steps, thus we claim (just as [Fitting and Ben-Jacob(1988)]) that $\Phi_P \uparrow \omega$ is a natural definition of the true and false things we can hope to compute from program P.

6.2 A constraint based operator

In this subsection we define a constraint-based version of the Φ_P operator, denoted $\Phi_P^{\mathcal{A}}$. We show that this operator agrees with $\Phi_P \uparrow k$ for finite k. The major reason for introducing this operator is to allow us to talk directly about goals $(c \,\rule{0pt}{0pt}\mathbf{|}\, G)$ in the language $\mathcal{L}$ rather than their instances in the structure $\mathcal{A}$. This will be essential for the completeness results with respect to Kunen's semantics (Theorem 8.4). We conclude with the key lemma that relates derivations to $\Phi_P^{\mathcal{A}}$.

For the remainder of the paper we assume programs P are in *normal form*, where each atom appearing in the program is of the form $p(\tilde{x})$ where $\tilde{x}$ are distinct variables. It is not difficult to rewrite any clause to an equivalent normal form clause by introducing new variables and equality constraints.

Consider how we can relate a partial interpretation I to a goal $(c \,\mathbf{|}\, G)$. To do so we must consider all ground instances of the goal. To that effect we introduce the following notation.

Definition 6.4 Let $\ll (c \,\mathbf{|}\, G) \gg$ represent all the ground instances of G satisfying c, that is

$$\ll (c \,\mathbf{|}\, G) \gg \;\equiv\; \{G\alpha \mid \mathcal{A} \models c\alpha\}$$

where α ranges over valuations in $\mathcal{A}$.

If we wish to define a partial interpretation in terms of goals rather than ground atoms, we can use a set of atomic goals to represent their ground instances. This description has the advantage of remaining inside the language $\mathcal{L}$ of the structure $\mathcal{A}$.

Definition 6.5 Define a *constraint interpretation* I as a mapping from goals of the form $(c \,\mathbf{|}\, p(\tilde{x}))$ to $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ where $\tilde{x}$ are distinct variables and c is a constraint formula with free variables $\tilde{x}$ such that $\mathcal{A} \models \tilde{\exists} c$.

The meaning of a constraint interpretation I can be understood by relating it to a partial interpretation denoted $\ll I \gg$ where

$$\ll I \gg (p(\tilde{s})) = \mathbf{t} \text{ iff } p(\tilde{s}) \in \ll (c \,\mathbf{|}\, p(\tilde{x})) \gg \;\wedge\; I(c \,\mathbf{|}\, p(\tilde{x})) = \mathbf{t}$$

$$\ll I \gg (p(\tilde{s})) = \mathbf{f} \text{ iff } p(\tilde{s}) \in \ll (c \,\mathbf{|}\, p(\tilde{x})) \gg \;\wedge\; I(c \,\mathbf{|}\, p(\tilde{x})) = \mathbf{f}$$

Intuitively $I(c(\tilde{x}) \,\mathbf{|}\, p(\tilde{x})) = \mathbf{t}$ means that $\forall \tilde{x}\; c(\tilde{x}) \rightarrow p(\tilde{x})$ and $I(c(\tilde{x}) \,\mathbf{|}\, p(\tilde{x})) = \mathbf{f}$ means that $\forall \tilde{x}\; c(\tilde{x}) \rightarrow \neg p(\tilde{x})$. We can extend a constraint interpretation to map arbitrary goals $(c \,\mathbf{|}\, G)$, by extending the above intuition, i.e. $I(c(\tilde{x}) \,\mathbf{|}\, G(\tilde{x})) = \mathbf{t}$ means that $\forall \tilde{x}\; c(\tilde{x}) \rightarrow G(\tilde{x})$ and $I(c(\tilde{x}) \,\mathbf{|}\, p(\tilde{x})) = \mathbf{f}$ means that

$\forall \tilde{x}\ c(\tilde{x}) \rightarrow \neg G(\tilde{x})$. The rules correspond to the strong three-valued interpretation of the connectives.

$$
\begin{array}{lcl}
I(c \mid A \wedge B) = \mathbf{t} & \Leftrightarrow & \exists c_1, c_2\ \ I(c_1 \mid A) = \mathbf{t} \wedge I(c_2 \mid B) = \mathbf{t} \wedge \mathcal{A} \models c \rightarrow c_1 \wedge c_2 \\
I(c \mid A \wedge B) = \mathbf{f} & \Leftrightarrow & \exists c_1\ \ I(c_1 \mid A) = \mathbf{f}\ \wedge\ \mathcal{A} \models c \rightarrow c_1 \\
 & \vee & \exists c_2\ \ I(c_2 \mid B) = \mathbf{f}\ \wedge\ \mathcal{A} \models c \rightarrow c_2 \\
 & \vee & \exists c_1, c_2\ \ I(c_1 \mid A) = \mathbf{f}\ \wedge\ I(c_2 \mid B) = \mathbf{f}\ \wedge\ \mathcal{A} \models c \rightarrow c_1 \vee c_2 \\
 & & \\
I(c \mid \neg G) = \mathbf{t} & \Leftrightarrow & I(c \mid G) = \mathbf{f} \\
I(c \mid \neg G) = \mathbf{f} & \Leftrightarrow & I(c \mid G) = \mathbf{t} \\
 & & \\
I(c \mid \exists y G) = \mathbf{t} & \Leftrightarrow & \exists c'\ \ I(c' \mid G) = \mathbf{t}\ \wedge\ \mathcal{A} \models c \rightarrow \exists y\ c' \\
I(c \mid \exists y G) = \mathbf{f} & \Leftrightarrow & \exists \text{ class } c_\alpha\ \ I(c_\alpha \mid G) = \mathbf{f}\ \wedge\ \mathcal{A} \models c \rightarrow \forall y \bigvee_\alpha c_\alpha \\
 & & \\
I(c \mid c') = \mathbf{t} & \Leftrightarrow & \mathcal{A} \models c \rightarrow c' \\
I(c \mid c') = \mathbf{f} & \Leftrightarrow & \mathcal{A} \models c \rightarrow \neg c'
\end{array}
$$

Now it is simple to define an immediate consequence operator $\Phi_P^{\mathcal{A}}$ that operates on constraint interpretations.

Definition 6.6 Define an immediate consequence function $\Phi_P^{\mathcal{A}}$ from constraint interpretations to constraint interpretations as follows:

- $\Phi_P^{\mathcal{A}}(I)(c \mid p(\tilde{x})) = \mathbf{t}$ if there exist renamed versions of clauses in P

  $$p(\tilde{x}) \leftarrow B_i$$

  and constraints c_i such that $I(c_i \mid B_i) = \mathbf{t}$ and $\mathcal{A} \models c \rightarrow c_1 \vee \cdots \vee c_n$.

- $\Phi_P^{\mathcal{A}}(I)(c \mid p(\tilde{x})) = \mathbf{f}$ if for each clause in P renamed to be of the form

  $$p(\tilde{x}) \leftarrow B_i$$

  there exists c_i such that $I(c_i \mid B_i) = \mathbf{f}$ and $\mathcal{A} \models c \rightarrow c_i$

- $\Phi_P^{\mathcal{A}}(I)(c \mid p(\tilde{x})) = \mathbf{u}$ otherwise.

Definition 6.7 Let $I_0^{\mathcal{A}}$ be the constraint interpretation that maps all atomic goals to undefined, i.e. $I_0^{\mathcal{A}}(c \mid p(\tilde{x})) = \mathbf{u}$ for all $(c \mid p(\tilde{x}))$. Note that $I_0^{\mathcal{A}}$ does not map all *goals* to undefined, for example $(X > 2 \mid X < 1)$ and $(X > 2 \mid X < 1 \wedge q(X, Y))$ are mapped to $\mathbf{f}$.

$$
\begin{array}{rcl}
\Phi_P^{\mathcal{A}} \uparrow 0 & = & I_0^{\mathcal{A}} \\
\Phi_P^{\mathcal{A}} \uparrow k+1 & = & \Phi_P^{\mathcal{A}}(\Phi_P^{\mathcal{A}} \uparrow k)
\end{array}
$$

By examining the definition of $\Phi_P^{\mathcal{A}}$ we can see that it is not really dependent on the structure $\mathcal{A}$ itself but rather the theory of $\mathcal{A}$, all the $\mathcal{L}$-sentences true in $\mathcal{A}$, which we shall denote $th(\mathcal{A})$. This is because the appearance of $\mathcal{A}$ in the definition is only to verify the truth of some $\mathcal{L}$-sentence. This makes it apparent that although we think about a CLP($\mathcal{A}$) program as existing in the context of some structure $\mathcal{A}$, since the program (and goals) involves only first-order $\mathcal{L}$-formulae, the meaning of the program really only depends on $th(\mathcal{A})$. Thus it could just as well be thought of as being a CLP($\mathcal{A}'$) program for some structure $\mathcal{A}'$ that is elementarily equivalent to $\mathcal{A}$ (sharing the same language $\mathcal{L}$).

We now show that $\Phi_P^{\mathcal{A}} \uparrow k$ represents the same information as $\Phi_P \uparrow k$ for finite k. First we show that each $\Phi_P^{\mathcal{A}} \uparrow k$ has a finite representation, so we can handle goals with existential quantifiers.

Lemma 6.8 *For all finite k and goals G, there exists finite sets $c_1, \ldots, c_n$, $d_1, \ldots, d_m$ such that $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G) = \mathbf{t}$ implies there exists $1 \leq i \leq n$ such that $c \rightarrow c_i$, and $\Phi_P^{\mathcal{A}} \uparrow k(d \,\|\, G) = \mathbf{f}$ implies there exists $1 \leq j \leq m$ such that $d \rightarrow d_j$.*

Proof: Let m be the number of connectives in G. By induction on $\omega.k + m$. If $m = k = 0$ then $I = \Phi_P^{\mathcal{A}} \uparrow 0$ and G is either a constraint or atom. Clearly if G is an atom $I(c \,\|\, p(\tilde{x})) = \mathbf{u}$ by definition. Otherwise if G is a constraint c then $c_1 = c$ and $d_1 = \neg c$ satisfy the conditions of the Lemma.

If $m > 0$ then G contains a connective. Given I applied to G is finitely definable clearly so is $\neg G$. Similarly for $G_1 \wedge G_2$ from finitely definable G_1 and G_2. The same is the case for $\exists y G$ from G, since we can use quantifier elimination to get c such that $\mathcal{A} \models c \leftrightarrow \exists y c'$, and similarly for the negative case. Hence for all bodies B in P, I applied to B is finitely definable.

Finally $m = 0$ and we have the limit case. Given $I = \Phi_P^{\mathcal{A}} \uparrow k$ is finitely definable for all G, we show that $I' = \Phi_P^{\mathcal{A}} \uparrow k + 1$ is finitely definable for atoms and constraints. The case for constraints is the same as the base case.

Now $I'(c \,\|\, p(\tilde{x})) = \mathbf{t}$ is finitely definable since each body B_i of a clause $p(\tilde{x}) \leftarrow B_i$ is finitely definable, and hence so is any satisfiable disjunction $c \equiv c_1 \vee \cdots \vee c_n$. For each c' such that $I(c' \,\|\, B) = \mathbf{t}$ for some body B then $I'(c' \,\|\, p(\tilde{x}))$ is a finite definition of I' applied to $p(\tilde{x})$. Similarly for $I'(c \,\|\, p(\tilde{x})) = \mathbf{f}$ we take each satisfiable conjunction $c \equiv c_1 \wedge \cdots \wedge c_n$ where $I(c_i \,\|\, B_i) = \mathbf{f}$ and B_i are the bodies of clauses for $p(\tilde{x})$ and this gives us a finite definition of $I'(c \,\|\, p(\tilde{x})) = \mathbf{f}$. $\Box$

Lemma 6.9 *$\ll \Phi_P^{\mathcal{A}} \uparrow k \gg = \Phi_P \uparrow k$ for all finite k.*

Proof: We prove $\Phi_P^{\mathcal{A}} \uparrow n(c \,\|\, G) = \Phi_P \uparrow n(G\alpha)$, where $\mathcal{A} \models \alpha \rightarrow c$. Let k be the first $\ll \Phi_P^{\mathcal{A}} \uparrow k \gg$ or $\Phi_P \uparrow k$ where G is not mapped to $\mathbf{u}$. Let m be the number of connectives in G. Induction on $\omega.k + m$

Clearly for the base case $k = 0, m = 0$ the only goals involved are constraints c and the proposition holds.

Connectives Case

Suppose $m > 0$, then the topmost symbol of G is one of $\wedge, \neg$ or $\exists$. The induction hypothesis holds because of the correspondence between the strong three valued interpretation of the connectives and the definition of constraint interpretations.

exists

Let $I_g = \Phi_P \uparrow k$ and $I_a = \Phi_P^{\mathcal{A}} \uparrow k$. $I_a(c \,\|\, \exists y G) = \mathbf{t}$ implies there exists c' such that $\mathcal{A} \models c \rightarrow \exists y c'$ and $I_a(c' \,\|\, G) = \mathbf{t}$. By the induction hypothesis $I_g(G\alpha') = \mathbf{t}$ for each α' such that $\mathcal{A} \models \alpha' \rightarrow c'$. Hence $I_g((\exists y G)\alpha) = \mathbf{t}$ for each α such that $\mathcal{A} \models \alpha \rightarrow c$ since they are restrictions of α's.

$I_g((\exists y G)\alpha) = \mathbf{t}$ implies there exists extension of α, α' such that $I_g(G\alpha') = \mathbf{t}$. By the induction hypothesis there exists c' such that $I_a(c' \mid G) = \mathbf{t}$ and $\mathcal{A} \models \alpha' \rightarrow c'$. Take $c \leftrightarrow \exists y c'$ then clearly $\mathcal{A} \models \alpha \rightarrow c$ and $I_a(c \mid \exists y G) = \mathbf{t}$.

$I_a(c \mid \exists y G) = \mathbf{f}$ implies there exists a collection c_β such that $A \models c \rightarrow \forall y \vee_\beta c_\beta$ and $I_a(c_\beta \mid G) = \mathbf{f}$. Consider any α such that $\mathcal{A} \models \alpha \rightarrow c$, then we can extend α to a substitution α' such that $\mathcal{A} \models \alpha' \rightarrow c'_\beta$ say. But by the induction hypothesis $I_g(G\alpha') = \mathbf{f}$ and hence $I_g((\exists y G)\alpha) = \mathbf{f}$.

$I_g((\exists y G)\alpha) = \mathbf{f}$ implies for each extension α' of α $I_g(G\alpha') = \mathbf{f}$. Hence by the induction hypothesis there exists c'_α such that $I_a(c \mid G) = \mathbf{f}$ and $\mathcal{A} \models \alpha' \rightarrow c'$. Since this is the case for all such extensions and because of the finite presentation theorem above, there exist a finite class c_β that include all such α'. Take $c \leftrightarrow \forall y \vee_\beta c_\beta$ then clearly $I_a(c \mid \exists y G) = \mathbf{f}$ and $\mathcal{A} \models \alpha \rightarrow c$.

not, and

Let $I_g = \Phi_P \uparrow k$ and $I_a = \Phi_P^{\mathcal{A}} \uparrow k$. Clearly the result holds for $\neg$ since $I_a(c \mid \neg G) = \mathbf{t} \Leftrightarrow I_a(c \mid G) = \mathbf{f}$ and $I_g(G\alpha) = \mathbf{t} \Leftrightarrow I_g(G\alpha) = \mathbf{f}$, and similarly for $\mathbf{f}$. Similarly for $G_1 \wedge G_2$.

Limit Case

Otherwise $m = 0$ and we have the limit case. Since there are no connectives G must be a constraint or an atom. For constraints see the base case, otherwise let $I_g^+ = \Phi_P \uparrow k+1$ and $I_a^+ = \Phi_P^{\mathcal{A}} \uparrow k+1$ and $I_g = \Phi_P \uparrow k$ and $I_a = \Phi_P^{\mathcal{A}} \uparrow k$. Suppose $I_g^+(A) = \mathbf{t}$. Then there exists renamed clauses in P

$$p(\tilde{x}) \leftarrow B$$

and valuation α such that $A = p(\tilde{x})\alpha$ and $I_g(B\alpha) = \mathbf{t}$. By induction there exists c' such that $\mathcal{A} \models \alpha \rightarrow c$ and $I_a(c' \mid B) = \mathbf{t}$. Hence $I_a^+(c' \mid p(\tilde{x})) = \mathbf{t}$ and $A \in \ll (c' \mid p(\tilde{x})) \gg$.

Suppose $I_a^+(c \mid p(\tilde{x})) = \mathbf{t}$. Then there exists renamed clauses in P

$$p(\tilde{x}) \leftarrow B_i$$

and c_i such that $c \rightarrow c_1 \vee \cdots \vee c_n$ and $I_a(c_i \mid B_i) = \mathbf{t}$. By induction for each valuation α such that $\alpha \rightarrow c_i$, for some i $I_g(B_i\alpha) = \mathbf{t}$. Hence using the clause above $I_g^+(p(\tilde{x})\alpha\gamma) = \mathbf{t}$ for any grounding substitution γ. Now $\ll (c \mid p(\tilde{x})) \gg \subseteq \{p(\tilde{x})\alpha\gamma\}$.

Suppose $I_g^+(A) = \mathbf{f}$. Then for each renamed clause in P of the form

$$p(\tilde{x}) \leftarrow B_i$$

if $p(\tilde{x})\alpha = A$ then $I_g(B_i\alpha) = \mathbf{f}$. By induction there exists c_i^A such that $I_a(c_i^A \mid B_i) = \mathbf{f}$ and $B_i\alpha \in \ll (c_i^A \mid B_i) \gg$. Let

$$c^A \rightarrow \bigwedge_i c_i^A$$

Now $\alpha \rightarrow c^A$ hence $\mathcal{A} \models c^A$ and $I_a^+(c^A \mid p(\tilde{x})) = \mathbf{f}$ by definition and $A \in \ll (c^A \mid p(\tilde{x})) \gg$.

Suppose $I_a^+(c \mid p(\tilde{x})) = \mathbf{f}$, then for each renamed clause in P of the form

$$p(\tilde{x}) \leftarrow B_i$$

there exists c_i such that $I_a(c_i \mid B_i) = \mathbf{f}$ and $c \rightarrow c_i$. For each $A = p(\tilde{x})\alpha \in \ll (c \mid p(\tilde{x}))$ since $\alpha \rightarrow c$, then $\alpha \rightarrow c_i$ and by induction $I_g(B_i\gamma) = \mathbf{f}$ for $B_i\gamma \in \ll (c_i \mid B_i) \gg$, hence $I_g^+(A) = \mathbf{f}$.
$\square$

We are now able to state the key lemma in connecting derivation trees to the bottom-up semantics $\Phi_P^{\mathcal{A}} \uparrow k$.

Lemma 6.10 (Key Lemma)
$\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G) = \mathbf{t}$ iff $(c \,\|\, G)$ has a totally successful BF-k-derivation tree. $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G) = \mathbf{f}$ iff $(c \,\|\, G)$ has a finitely failed BF-k-derivation tree.

Proof: By induction on $\omega.k + l$ where k is the least k such that $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G) = \mathbf{f}$ or $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G) = \mathbf{t}$ and l is the number of connectives in G.

Base case
T: Only goals such that $\Phi_P^{\mathcal{A}} \uparrow 0(c \,\|\, G) = \mathbf{t}$ are $(c \,\|\, c')$ where $c \rightarrow c'$ and $\mathcal{A} \models c$. These are the only goals (where $\omega.k + n = 0$) with totally successful BF-0-derivation trees.

F: Only goals such that $\Phi_P^{\mathcal{A}} \uparrow 0(c \,\|\, G) = \mathbf{f}$ are $(c \,\|\, c')$ where $c \rightarrow \neg c'$ and $\mathcal{A} \models c$. These are the only goals (where $\omega.k + n = 0$) with finitely failed BF-0-derivation-trees.

Successor Ordinal
$n > 0$ thus their is a topmost connective either $\exists y$, $\neg$ or $\wedge$. We examine each case in turn.

And
T: $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G_1 \wedge G_2) = \mathbf{t}$ means there exist $(c_1 \,\|\, G_1), (c_2 \,\|\, G_2)$ such that $\Phi_P^{\mathcal{A}} \uparrow k(c_1 \,\|\, G_1) = \mathbf{t}$ and $\Phi_P^{\mathcal{A}} \uparrow k(c_2 \,\|\, G_2) = \mathbf{t}$ and $c \rightarrow c_1 \wedge c_2$. By induction $(c_1 \,\|\, G_1)$ and $(c_2 \,\|\, G_1)$ have totally successful BF-k-derivations trees. By Lemma 5.6 the BF-k-derivation trees for $(true \,\|\, G_1)$ and $(true \,\|\, G_2)$ have successful derivations leading to constraints $c_1^1, \ldots, c_n^1$ and $c_1^2, \ldots, c_m^2$ respectively such that $c_1 \rightarrow \overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_1^1 \vee \cdots \vee \overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_n^1$ and $c_2 \rightarrow \overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_1^2 \vee \cdots \vee \overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_m^2$. Cross-multiplying the trees for $(true \,\|\, G_1)$ and $(true \,\|\, G_2)$ leads to a BF-k-derivation tree for $(true \,\|\, G_1 \wedge G_2)$ which has a successful derivations with constraints $c_i^1 \wedge c_j^2$ (or $c_i^1 \wedge c_j^2$ is unsatisfiable). Hence since $c \rightarrow (c_1 \wedge c_2) \rightarrow (\overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_1^1 \vee \cdots \vee \overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_n^1) \wedge (\overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_1^2 \vee \cdots \vee \overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_m^2)$ by Lemma 5.6 we have a totally successful BF-k-derivation tree for $(c \,\|\, G_1 \wedge G_2)$.

In the reverse direction by splitting the constraints arising in successful derivations of the BF-k-derivation tree for $(true \,\|\, G_1 \wedge G_2)$ into those derived from G_1 and G_2 we can derive constraints c_1 and c_2 such that $c \leftrightarrow c_1 \wedge c_2$ and $(c_1 \,\|\, G_1)$ and $(c_2 \,\|\, G_2)$ are totally successful. By induction $\Phi_P^{\mathcal{A}} \uparrow k(c_1 \,\|\, G_1) = \mathbf{t}$ and $\Phi_P^{\mathcal{A}} \uparrow k(c_2 \,\|\, G_2) = \mathbf{t}$ and hence $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G_1 \wedge G_2) = \mathbf{t}$

F: We tackle the most general case, the others are similar. $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G_1 \wedge G_2) = \mathbf{f}$ and suppose $\Phi_P^{\mathcal{A}} \uparrow k(c_1 \,\|\, G_1) = \mathbf{f}$, $\Phi_P^{\mathcal{A}} \uparrow k(c_2 \,\|\, G_2) = \mathbf{f}$ and $c \rightarrow c_1 \vee c_2$. Consider a BF-$k$-derivation tree for $(true \,\|\, G_1 \wedge G_2)$. Now on each path from the root in this tree, by some depth $\leq k$ either the path is finitely failed or the constraints at some node c_1' collected from the BF-k-derivation tree of $(true \,\|\, G_1)$ are such that $c_1 \rightarrow \neg\overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_1'$ by Lemma 5.6. Similarly for c_2'. Clearly $c \rightarrow c_1 \vee c_2 \rightarrow (\neg\overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_1') \vee (\neg\overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} c_2') \rightarrow \neg\overline{\exists}_{(c \,\|\, G_1 \wedge G_2)} (c_1' \wedge c_2')$. Hence the constraints collected at this node $c_1' \wedge c_2'$ are incompatible with c. Thus by Lemma 5.6 the BF-k-derivation tree for $(c \,\|\, G_1 \wedge G_2)$ is finitely failed.

In the reverse direction (again tackling the most general case) using Lemma 5.11 we have that there exists c_1, c_2 such that $c \rightarrow c_1 \vee c_2$ and the BF-k-derivation trees for $(c_1 \,\|\, G_1)$ and $(c_2 \,\|\, G_2)$ are finitely failed. By the induction hypothesis $\Phi_P^{\mathcal{A}} \uparrow k(c_1 \,\|\, G_1) = \mathbf{f}$, $\Phi_P^{\mathcal{A}} \uparrow k(c_2 \,\|\, G_2) = \mathbf{f}$ and hence $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G_1 \wedge G_2) = \mathbf{f}$

Not

T: $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, \neg G) = \mathbf{t}$ iff $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G) = \mathbf{f}$ iff $(c \,\|\, G)$ has a finitely-failed BF-k-derivation tree by induction, iff the BF-k-derivation tree for $(c \,\|\, \neg G)$ is totally successful by Lemma 5.7.

F: $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, \neg G) = \mathbf{f}$ iff $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G) = \mathbf{t}$ iff $(c \,\|\, G)$ has a totally successful BF-k-derivation tree by induction, iff the BF-k-derivation tree for $(c \,\|\, \neg G)$ is finitely failed by Lemma 5.8.

Exists

T: $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, \exists y G) = \mathbf{t}$ hence there exists c' such that $\Phi_P^{\mathcal{A}} \uparrow k(c' \,\|\, G) = \mathbf{t}$ and $c \rightarrow \exists y c'$. By induction we have a totally successful BF-k-derivation tree for $(c' \,\|\, G)$, and using Lemma 5.6 twice we obtain a totally successful BF-k-derivation tree for $(c \,\|\, \exists y G)$.

In the reverse direction, if the BF-k-derivation tree for $(c \,\|\, \exists y G)$ is totally successful then using Lemma 5.6 the BF-k-derivation tree for $(true \,\|\, \exists y G)$ has successful derivations ending in constraints $c_1, \ldots c_n$ such that $c \rightarrow \exists y \overline{\exists} c_1 \vee \cdots \vee \exists y \overline{\exists} c_n$. Let $c' \leftrightarrow \overline{\exists} c_1 \vee \cdots \vee \overline{\exists} c_n$. Now clearly BF-$k$-derivation tree for $(c' \,\|\, G)$ is totally successful and by induction $\Phi_P^{\mathcal{A}} \uparrow k(c' \,\|\, G) = \mathbf{t}$. Now $c \rightarrow \exists y c'$ hence $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, \exists y G) = \mathbf{t}$.

F: $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, \exists y G) = \mathbf{f}$ hence there exists class c_α such that $\Phi_P^{\mathcal{A}} \uparrow k(c_\alpha \,\|\, G) = \mathbf{f}$ and $c \rightarrow \forall y \vee_\alpha c_\alpha$. Take the BF-$k$-derivation tree for $(true \,\|\, G)$. On each branch by depth $\leq k$ for each c_α the constraints c' at some node are such that $c_\alpha \rightarrow \neg \overline{\exists} c'$. Now $c \rightarrow \forall y \vee_\alpha c_\alpha \rightarrow \forall y \neg \overline{\exists} c' \rightarrow \neg \exists y \overline{\exists} c'$. Hence each branch in the BF-k-derivation tree for $(c \,\|\, \exists y G)$ is finitely failed by Lemma 5.6.

In the reverse direction, if the BF-k-derivation tree for $(c \,\|\, \exists y G)$ is finitely failed then the BF-k-derivation tree for $(c \,\|\, G)$ is also finitely failed and by induction $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, G) = \mathbf{f}$, and thus $\Phi_P^{\mathcal{A}} \uparrow k(c \,\|\, \exists y G) = \mathbf{f}$.

Limit Ordinal

$l = 0$ hence the number of connectives is zero hence the goal is either of the form $(c \,\|\, p(\tilde{x}))$ or $(c \,\|\, c')$. For constraints see the base case.

T: $\Phi_P^{\mathcal{A}} \uparrow k + 1(c \,\|\, p(\tilde{x})) = \mathbf{t}$ hence there exist renamed versions of clauses of P

$$p(\tilde{x}) \leftarrow B_i$$

and constraints $c_1, \ldots, c_n$ such that $\Phi_P^{\mathcal{A}} \uparrow k(c_i \,\|\, B) = \mathbf{t}$ and $c \rightarrow c_1 \vee \cdots \vee c_n$. By induction we have a totally successful BF-k-derivation tree for each $(c_i \,\|\, B_i)$ and clearly using Lemma 5.6 we have a totally successful BF-$k+1$-derivation tree for for $(c \,\|\, p(\tilde{x}))$.

In the reverse direction, using Lemma 5.6 the BF-$k+1$-derivation tree $(true \,\|\, p(\tilde{x}))$ has successful derivations ending in constraints $c_1, \ldots c_n$ such that $c \rightarrow \overline{\exists}_{(c \,\|\, p(\tilde{x}))} d_1 \vee \cdots \vee d_n$. Let $d_1^i, \ldots, d_{n_i}^i$ be the constraints which occur in a derivation using the clause

$$p(\tilde{x}) \leftarrow B_i$$

of P at the first step. Let $c_i \leftrightarrow \exists_{(c \,\mathbf{|}\, p(\tilde{x}))} d_1^i \vee \cdots \vee d_{n_i}^i$. Clearly using Lemma 5.6 $(c_i \,\mathbf{|}\, B_i)$ is totally successful, and hence by induction $\Phi_P^{\mathcal{A}} \uparrow k(c_i \,\mathbf{|}\, B) = \mathbf{t}$. Thus $\Phi_P^{\mathcal{A}} \uparrow k+1(c \,\mathbf{|}\, p(\tilde{x})) = \mathbf{t}$.

F: $\Phi_P^{\mathcal{A}} \uparrow k+1(c \,\mathbf{|}\, p(\tilde{x})) = \mathbf{f}$ hence for each renamed clause of P of the form

$$p(\tilde{x}) \leftarrow B_i$$

there exists c_i such that $\Phi_P^{\mathcal{A}} \uparrow k(c_i \,\mathbf{|}\, B_i) = \mathbf{f}$ and $c \rightarrow c_i$. By induction we have a finitely failed BF-k-derivation tree for each $(c_i \,\mathbf{|}\, B_i)$. Clearly the BF-k-derivation trees for $(c \,\mathbf{|}\, B_i)$ are all finitely failed. Hence the BF-$k+1$-derivation tree for $(c \,\mathbf{|}\, p(\tilde{x}))$ is finitely failed.

If the BF-$k+1$-derivation tree for $(c \,\mathbf{|}\, p(\tilde{x}))$ is finitely failed then for each renamed clause of P of the form

$$p(\tilde{x}) \leftarrow B_i$$

the BF-k-derivation trees for $(c \,\mathbf{|}\, B_i)$ are all finitely failed. Hence, by induction, $\Phi_P^{\mathcal{A}} \uparrow k(c \,\mathbf{|}\, B_i) = \mathbf{f}$ and thus $\Phi_P^{\mathcal{A}} \uparrow k+1(c \,\mathbf{|}\, p(\tilde{x})) = \mathbf{f}$. □

7 Soundness

In this section we give the soundness results of our scheme with respect to both $\Phi_P \uparrow \omega$ and three-valued consequences of the completed program.

Theorem 7.1 *Let $\mathcal{A}$ be a structure, P be a normal program over $\mathcal{A}$, R a fair consistent computation rule and $(c \,\mathbf{|}\, G)$ a goal. Then if $(c \,\mathbf{|}\, G)$ has a totally successful (resp. finitely failed) $(P, \mathcal{A}, R)$-derivation tree, $\Phi_P^{\mathcal{A}} \uparrow k(c \,\mathbf{|}\, G) = \mathbf{t}$ (resp. $\mathbf{f}$) for some finite k.*

Proof: The result follows directly from Lemmas 5.12, 6.10 and Corollary A.5. □

Corollary 7.2 (Soundness I) *Let $\mathcal{A}$ be a structure, P be a normal program over $\mathcal{A}$, R a fair consistent computation rule and G a literal and c a constraint. Then if $(c \,\mathbf{|}\, G)$ has a totally successful (resp. finitely failed) $(P, \mathcal{A}, R)$-derivation tree, $\Phi_P \uparrow \omega(G') = \mathbf{t}$ (resp. $\mathbf{f}$) for each $G' \in \ll (c \,\mathbf{|}\, G) \gg$.*

Proof: The result follows from the definition of $\Phi_P \uparrow \omega$ together with 6.9 and the above theorem. □

The soundness result for the completion is relatively straightforward. The following lemma (the extension of Lemma 2.1 to normal programs) provides the key.

Lemma 7.3 *If P is a normal program and $\{G_1, \ldots, G_n\}$ is a $(P, \mathcal{A}, R)$-frontier for G then*

$$\mathcal{A} \wedge P* \models_3 G \leftrightarrow (\exists \mathbf{Y_1}\ G_1) \vee \cdots \vee (\exists \mathbf{Y_n}\ G_n)$$

where each $\mathbf{Y}_i$ is the set of variables in G_i not in G.

Proof: We proceed by induction on the depth of nesting of negative sub-derivations

The base case is exactly Lemma 2.1. For the induction step we show each node in the $(P, \mathcal{A}, R)$-derivation tree for G is equivalent to the existentially closed disjunction of its children. The only interesting case in the induction is if the selected subgoal D_j is a negative subgoal $\neg\exists \mathbf{Z}\ Q$, then by the induction hypothesis the chosen frontier $c' \wedge Q_1, \ldots c' \wedge Q_n$ for the derivation of $(c' \mid Q)$ is such that

$$\mathcal{A} \wedge P* \models_3 c' \wedge Q \leftrightarrow \exists \mathbf{Y_1}\ c' \wedge Q_1 \vee \cdots \vee \exists \mathbf{Y_n}\ c' \wedge Q_n$$

Because c' does not involves any of the variables $\mathbf{Z}, \mathbf{Y}_1, \ldots, \mathbf{Y}_n$, and because $\mathcal{A} \models c \rightarrow c'$ we can follow the chain of equivalence (wrt $\mathcal{A} \wedge P*$)

$$\begin{array}{ll} & c \wedge \neg\exists \mathbf{Z}\ Q \\ \leftrightarrow & c \wedge \neg\exists \mathbf{Z}\ (c' \wedge Q) \\ \leftrightarrow & c \wedge \neg\exists \mathbf{Z}\ (\exists \mathbf{Y_1}\ c' \wedge Q_1 \vee \cdots \vee \exists \mathbf{Y_n}\ c' \wedge Q_n) \\ \leftrightarrow & c \wedge \neg\exists \mathbf{Z}\ (c' \wedge (\exists \mathbf{Y_1}\ Q_1 \vee \exists \mathbf{Y_n}\ Q_n)) \\ \leftrightarrow & c \wedge \neg\exists \mathbf{Z}\ (\exists \mathbf{Y_1}\ Q_1 \vee \exists \mathbf{Y_n}\ Q_n) \end{array}$$

Now if $R_1, \ldots R_p$ are the goals returned after negating and simplifying the frontier we have

$$A \models_3 \neg\exists \mathbf{Z}\ (\exists \mathbf{Y_1}\ Q_1 \vee \cdots \vee \exists \mathbf{Y_n}\ Q_n) \leftrightarrow R_1 \vee \cdots \vee R_p$$

Thus

$$\mathcal{A} \wedge P* \models_3 c \wedge \neg\exists \mathbf{Z}\ Q \leftrightarrow c \wedge (R_1 \vee \cdots \vee R_p)$$

Clearly then every goal G is equivalent to the disjunction of its immediate children. Thus we can easily show that any frontier $\{G_1, \ldots, G_n\}$ of G, that

$$\mathcal{A} \wedge P* \models_3 G \leftrightarrow (\exists \mathbf{Y_1}\ G_1) \vee \cdots \vee (\exists \mathbf{Y_n}\ G_n)$$

by building the frontier iteratively and removing finitely failed leaves. □

Considering the discussion in the previous section it is easy to see that the above result does not depend strictly on $\mathcal{A}$ but just on $th(\mathcal{A})$. The second soundness result follows straightforwardly from the above lemma, and these considerations.

Theorem 7.4 (Soundness II) *Let P be a normal program over structure $\mathcal{A}$, and R a computation rule. If $(c \mid G)$ has a totally successful $(P, \mathcal{A}, R)$-derivation tree then*

$$th(\mathcal{A}) \wedge P* \ \models_3\ \tilde{\forall} c \rightarrow G$$

If $(c \mid G)$ has a finitely failed $(P, \mathcal{A}, R)$-derivation tree then

$$th(\mathcal{A}) \wedge P* \ \models_3\ \neg\tilde{\exists}(c \wedge G)\ \square$$

8 Completeness

In this section we give the completeness of our scheme with respect to $\Phi_P \uparrow \omega$, and with respect to Kunen's three valued semantics [Kunen(1987)]. In contrast, Chan does not give any completeness result for his scheme, although [Przymusinski(1989)] provides a result for a *very* restricted class of programs. [Sato and Motoyoshi(1991)] and [Drabent(1993)] obtains similar completeness results for the particular case of the Herbrand Universe.

Theorem 8.1 *Let $\mathcal{A}$ be a structure, P be a normal program over $\mathcal{A}$, R a fair consistent computation rule and $(c \mid G)$ a goal. If $\Phi_P^{\mathcal{A}} \uparrow k(c \mid G) = \mathbf{t}$ (resp. $\mathbf{f}$) for some finite k then the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid G)$ is totally successful (resp. finitely failed).*

> **Proof:** The result follows directly from Lemmas 5.12, 6.10 and Corollary A.5. □

Corollary 8.2 (Completeness I) *Let $\mathcal{A}$ be a structure, P be a normal program over $\mathcal{A}$, R a fair consistent computation rule and G a ground literal. If $\Phi_P \uparrow \omega(G) = \mathbf{t}$ (resp. $\mathbf{f}$) then there exists goal $(c \mid G')$ such that $G \in \ll (c \mid G') \gg$ and the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid G')$ is totally successful (resp. finitely failed).*

> **Proof:** The result follows from the definition of $\Phi_P \uparrow \omega$ together with Lemma 6.9 and the above theorem. □

For the structure of the Herbrand Universe, with a countably infinite number of function symbols $\mathcal{HU}$, [Kunen(1987)] gives a 3-valued model-theoretic justification for cutting off the iteration of Φ_P at the ω step. Let CET^2 be Clark's equality theory for the language of $\mathcal{HU}$, with the following axiom to restrict the models of equality to be two valued: $\forall x \forall y \; x = y \vee x \neq y$.

Theorem 8.3 *[Kunen(1987)] Let P be a normal program over the structure $\mathcal{HU}$ and let $(c \mid G)$ be any goal. Then the following are equivalent:*

- $\Phi_P^{\mathcal{HU}} \uparrow k(c \mid G) = \mathbf{t}$ *for some finite* k
- $CET^2 \wedge P* \models_3 \tilde{\forall} c \rightarrow G$ □

The proof is virtually independent of the structure $\mathcal{HU}$ and readily extends to other structures $\mathcal{A}$. This together with Theorem 8.1 leads to the final completeness result.

Theorem 8.4 (Completeness II) *Let P be a normal program over a structure $\mathcal{A}$, R a fair consistent computation rule and $(c \mid G)$ a goal. If*

$$th(\mathcal{A}) \wedge P* \models_3 \tilde{\forall} c \rightarrow G$$

then the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid G)$ is totally successful. If

$$th(\mathcal{A}) \wedge P* \models_3 \neg \tilde{\exists}(c \wedge G)$$

then the $(P, \mathcal{A}, R)$-derivation tree for $(c \mid G)$ is finitely failed. □

9 Structures for Constructive Negation

In this section we examine structures that are commonly used in constraint logic programming and show how they may be practically used within our constructive negation scheme. An essential part of each derivation step is checking the satisfaction of the constraints in the derivation. In practice we desire an efficient decision procedure for this question, and hence we wish to limit the constraints formulae to some simple form if possible. To this end we introduce the concept of admissible constraints of a structure $\mathcal{A}$.

The *admissible constraints* of $\mathcal{A}$ are a particular subset of the formulae over the primitive constraints of $\mathcal{A}$. Usually the admissible constraints are the primitive constraints themselves, but for some structures this is not the case: for example Chan's structure for the Herbrand Universe has admissible constraints $s = t$ and $\forall \mathbf{Z}\ s \neq t$.

Admissible constraints intuitively represent the constraints which the constraint solver can deal with, and thus for a decidable structure $\mathcal{A}$ we could choose all formulae as admissible constraints. To ensure that satisfiability of conjunctions of admissible constraints is efficient, we usually choose admissible constraints to be as close to the primitive constraints as possible. Now to be able to restrict computation to admissible constraints we must be able to simplify a formula of the form $\neg \exists \mathbf{Z}\ c$, where c is a conjunction of admissible constraints, into a formula involving only disjunctions of conjunctions of admissible constraints.

Definition 9.1 A structure $\mathcal{A}$ is *admissible closed* if for each conjunction of admissible constraints $F(\mathbf{X}, \mathbf{Y})$, involving variables $\mathbf{X}$ and $\mathbf{Y}$, there is a disjunction $G(\mathbf{X}, \mathbf{Z})$ of conjunctions of admissible constraints involving variables $\mathbf{X}$ and $\mathbf{Z}$ only such that

$$\mathcal{A} \models \forall \mathbf{X}\ (\neg \exists \mathbf{Y} F(\mathbf{X}, \mathbf{Y})) \leftrightarrow \exists \mathbf{T}\ G(\mathbf{X}, \mathbf{T}))$$

Why is admissible closure different from quantifier elimination? The difference is that we need to remove negations from the formula $G(\mathbf{X}, \mathbf{T})$ since the satisfaction procedure for a structure $\mathcal{A}$ is only necessarily applicable to conjunctions of admissible constraints and $\mathcal{A}$ may allow primitive constraints (e.g. $\geq$) without their negations ($<$). Also, in general, some quantifiers remain in $G(\mathbf{X}, \mathbf{T})$. This occurs in structures such as the Herbrand Universe where admissible constraints sometimes contain quantifiers e.g. $\forall \mathbf{Z}\ s \neq t$.

Example 9.2 The admissible constraints for $\mathcal{HU}$ are equations and universally quantified disequations. Using the procedure in [Chan (1988)] a conjunction of admissible constraints can be negated to obtain a disjunction of conjunctions of admissible constraints. For example

$$\begin{aligned} & \neg \exists u \exists v\ x_1 = f(x_2, u, v) \wedge \forall w (x_2 \neq g(w, v))) \\ \leftrightarrow \quad & (\forall u \forall v\ x_1 \neq f(x_2, u, v)) \vee (\exists u \exists v \exists w\ x_1 = f(x_2, u, v) \wedge x_2 = g(w, v)) \end{aligned}$$

□

Clearly if the structure $\mathcal{A}$ has a quantifier elimination procedure then any constraint formula can be reduced to a disjunction of conjunction of primitive constraints and their negations. Hence the structure is admissible closed if the negations of primitive constraints are themselves admissible constraints. Let structure $\mathcal{A}$ be *closed under negation* if the negation of each primitive constraint is also a primitive constraint.

Proposition 9.3 *If $\mathcal{A}$ admits quantifier elimination, and is closed under negation then $\mathcal{A}$ is admissible closed.* $\Box$

It is not difficult to show that many of the structures used in constraint logic programming are admissible closed. The structures of linear rational arithmetic, free Boolean algebra, real arithmetic, finite integer domains, Presburger arithmetic and function-free Herbrand equations and disequations, all admit quantifier elimination and are closed under negation (for a natural choice of primitive constraints) and hence are admissible closed. $\mathcal{HU}$ can be shown to be admissible closed using the simplification technique of Chan. The structure of rational trees is admissible closed when we choose admissible constraints to be positive and negative (negated) *rational basic formula*, where a rational basic formula is a formula (described in [Maher(1989)]) of the form

$$\exists \mathbf{U}\ \mathbf{X} = t(\mathbf{X}, \mathbf{U}) \wedge \mathbf{U} = r(\mathbf{X}, \mathbf{U})$$

where $\mathbf{X} = t(\mathbf{X}, \mathbf{U})$ is a conjunction of constraints of the form $x = t$ where each $x \in \mathbf{X}$ appears exactly once on the left hand side and t is a term containing variables from $\mathbf{X} \cup \mathbf{U}$, and similarly for $\mathbf{U} = r(\mathbf{X}, \mathbf{U})$.

An important class of structures for constraint logic programming are those of the form finite trees over another structure $\mathcal{A}$. For example, the finite trees over the real numbers are the structure underlying the language CLP($\Re$) [Jaffar *et al*(1992)]. Most constraint logic programming systems incorporate the uninterpreted functors in their structure. This enables them to retain PROLOG programming techniques, and provides data-structures. The structure of such systems, denoted $f(\mathcal{A})$, is made up of an underlying structure $\mathcal{A}$ augmented with uninterpreted functors (with the usual axioms). Terms in such structures are made up of finite trees whose leaves may be constants, variables, or expressions over the underlying domain. For example: $f(a, X+Y), T-4, g(a)$ are terms in the structure $f(\Re)$. Admissible constraints are made up of the admissible constraints from the underlying structure $\mathcal{A}$ and equality and quantified disequality over the terms.

We can show that structures $f(\mathcal{A})$, where $\mathcal{A}$ is admissible closed, are also admissible closed, by a procedure based on splitting the constraints over the domain $f(\mathcal{A})$ into more or less independent parts. The principal difficulties arise from handling quantified disequation constraints. Details and proof of the algorithm can be found in [Stuckey(1992)].

Theorem 9.4 *If $\mathcal{A}$ is admissible closed, then the structure $f(\mathcal{A})$ is admissible closed.* $\Box$

We have shown all of the commonly used structures in constraint logic programming satisfy the admissible closure requirement, in particular the structures of the languages:

- DATALOG: function-free uninterpreted equations and disequations,
- CLP($\mathcal{R}$) [Jaffar *et al*(1992)]: $f(\Re)$,
- (the core part of) CHIP [Dincbas *et al*(1988)]: finite integer domains,
- PROLOG-II [Colmerauer(1982)]: rational tree equations and disequations,
- and CAL [Sakai and Aiba(1987]: free Boolean algebra.

Hence constructive negation is suited for use with these languages.

10 Conclusion

In conclusion, constructive negation provides the first operational mechanism for truly dealing with negation in constraint logic programs. The alternative, safe SLDNF, is subject to floundering and useless in constraint programming where often variables are never grounded. Constructive negation gives us an operational semantics which matches the iterative semantics ($\Phi_P \uparrow \omega$) and logical semantics. In addition, the constraint logic programming viewpoint is the natural view for describing constructive negation even for the Herbrand Universe, since it essentially involves constraints other than equality.

Constructive negation is necessarily expensive in the general case (since it extends programs to incorporate bodies which are arbitrary formulae). The promise of constructive negation is that in usual programs the cost of constructive negation is not too far beyond that of negation as failure, and this cost is directly proportional to its usefulness. Experiments in [Chan and Wallace(1989)] found that using constructive negation could improve the efficiency of procedural programs since they could be rewritten as simpler programs using pure negation.

Acknowledgments

I would like to thank James Harland, Nevin Heintze, Joxan Jaffar, John Jeavons, Ken Kunen and Michael Maher for valuable discussions of this material. This work is partially supported by the Centre for Intelligent Decision Systems.

References

[Chan (1988)] Chan, D. (1988), Constructive Negation Based on the Completed DataBase, *in* "Proceedings, 5th Int. Conf. and Symp. on Logic Programming", pp. 111–125.

[Chan(1989)] Chan, D.(1989), An Extension of Constructive Negation and its Application in Coroutining, *in* "Proceedings, North American Conf. on Logic Programming 89", pp. 477–493.

[Chan and Wallace(1989)] Chan, D. and Wallace, M.(1989), An Experiment with Programming Using Pure Negation, *in* "Proceedings. AISB Conference", Sussex, England, 1989.

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

[Colmerauer(1982)] Colmerauer, A.(1982), PROLOG II – Reference Manual and Theoretical Model, Internal Report, Groupe Intelligence Artificielle, Université Aix-Marseille II.

[Dincbas *et al*(1988)] Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F.(1988), The Constraint Logic Programming Language CHIP, *in* "Proceedings, Conference on Fifth Generation Computer Systems".

[Drabent(1993)] Drabent, W.(1993) What is Failure? Or: Constructive Negation by Fail Answers, Linköping University, Technical Report.

[FITTING(1985)] FITTING, M.(1985), A Kripke/Kleene Semantics for Logic Programs, *Journal of Logic Programming*, **2**, pp. 295–312.

[FITTING AND BEN-JACOB(1988)] FITTING, M. AND BEN-JACOB, M.(1988), Stratified and Three-valued Logic Programming Semantics, *in* "Proceedings. 5th Int. Conf. and Symp. on Logic Programming", pp. 1054–1069.

[FOO *et al*(1988)] FOO, N., RAO, A., TAYLOR, A., AND WALKER, A.(1988), Deduced Relevant Types and Constructive Negation, *in* "Proceedings, 5th Int. Conf. and Symp. on Logic Programming", pp. 126–139.

[HARIDI AND BRAND(1988)] HARIDI, S. AND BRAND, P.(1988), Andorra Prolog, an integration of Prolog and committed choice languages, *in* "Proceedings, Fifth Generation Computer Systems 88".

[JAFFAR *et al*(1992)] JAFFAR, J., MICHAYLOV, S., STUCKEY, P., AND YAP, R.(1992), The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, **14**, 339–395. "The CLP($\Re$) Language and System" IBM Technical Report RC 16292 (#72336), November 1990.

[JAFFAR AND LASSEZ(1987)] JAFFAR, J. AND LASSEZ, J–L.(1987), Constraint Logic Programming, *in* "Proceedings, ACM Conference on Principles of Programming Languages", pp. 111-119.

[KUNEN(1987)] KUNEN, K.(1987), Negation in Logic Programming, *Journal of Logic Programming*, **4**, 289–308.

[KUNEN(1989)] KUNEN, K.(1989), Signed Data Dependencies in Logic Programs, *Journal of Logic Programming*, **7**, 231–245.

[LASSEZ AND MARRIOTT(1987)] LASSEZ, J–L. AND MARRIOTT, K.(1987), Explicit Representation of Terms Defined by Counter Examples, *Journal of Automated Reasoning*, **3**.

[LOVELAND AND SHOSTAK(1985)] LOVELAND, D. AND SHOSTAK, R.(1985), Simplifying Interpreted Formulas, *in* "Proceedings, Conf. on Automated Deduction", L.N.C.S 87, pp. 97–109.

[LLOYD AND TOPOR(1984)] LLOYD, J. AND TOPOR, R.(1984), Making PROLOG More Expressive, *Journal of Logic Programming*, **1**, 225–240.

[LLOYD(1987)] LLOYD, J. W.(1987), "Foundations of Logic Programming", Springer-Verlag.

[MAHER(1987)] MAHER, M.(1987), Logic Semantics for a Class of Committed-choice Programs, *in* "Proceedings, 4th Int. Conf. on Logic Programming", pp. 858–876.

[MAHER(1989)] MAHER, M.(1988), Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees, *in* "Proceedings, 3rd Symposium on Logic in Computer Science", pp. 348–357.

[MAHER(1993)] MAHER, M.(1988), A Logic Programming View of CLP, *in* "Proceedings, 10th Int. Conf. on Logic Programming" pp. 737–756.

[MAŁUSZYŃSKI AND NÄSLUND(1989)] MAŁUSZYŃSKI, J. AND NÄSLUND, T.(1989), Fail Substitutions for Negation as Failure, *in* "Proceedings, North American Conf. on Logic Programming", pp. 461–476.

[NAISH(1986)] NAISH, L.(1986), "Negation and Control in Prolog", L.N.C.S. 238, Springer-Verlag.

[PRZYMUSINSKI(1989)] PRZYMUSINSKI, T.C.(1989), On Constructive Negation in Logic Programming, *in* "Proceedings, North American Conf. on Logic Programming", preprint.

[Sakai and Aiba(1987] Sakai, K. and Aiba, A.(1987), Introduction to CAL, ICOT Technical Report, December.

[Sato and Motoyoshi(1991)] Sato T. and Motoyoshi, F.(1991), A Complete Top-down Interpreter for First Order Programs, *in* "Proceedings, International Logic Programming Symposium", pp. 35–53.

[Stuckey(1992)] Stuckey, P.(1992), Quantifier Elimination for Structures with Uninterpreted Functors, University of Melbourne, Department of Computer Science, Technical report, 92/23.

[Wallace(1987)] Wallace, M.(1987), Negation by Constraints: A Sound and Efficient Implementation of Negation in Deductive Databases, *in* "Proceedings, IEEE Symposium on Logic Programming", pp. 253–263.

[Wolfram *et al*(1984)] Wolfram, D., Maher, M., and Lassez, J-L.(1984), A Unified Treatment of Resolution Strategies for Logic Programs, *in* "Proceedings, 2nd Int. Conf. on Logic Programming", pp. 263–276.

A Other Computation Rules

Extending the results of Section 5 to more general computation rules follows from two observations. First, passing of constraints c' into sub-derivations only serves to simplify the derivation trees, hence any DF-derivation tree that is totally successful or finitely failed will maintain this property if we change the computation rule to pass in constraints. In the reverse direction the additional derivations that result from removing constraints c' can all be seen to be subsumed by other derivations. Second, any frontier selection can be imitated by a number of DF derivation steps. To extend the results to computation rules that pass constraints into a derivation (as opposed to DF computation rules), we introduce notions of frontier success and failure.

Definition A.1 If $(c_1 \mid G_1), \ldots, (c_n \mid G_n)$ is a frontier of $(c \mid G)$ then it is c_0*-successful* if there exist success nodes $(c_{i_1} \mid), \ldots, (c_{i_m} \mid)$ in the frontier such that $c_0 \rightarrow \overline{\exists} c_{i_1} \vee \cdots \vee \overline{\exists} c_{i_m}$. (Note that $c_0 \rightarrow c$ must hold). If $(c_1 \mid G_1), \ldots, (c_n \mid G_n)$ is a frontier of $(c \mid G)$ then it is c_0*-failed* if for each node $(c_i \mid G_i)$ in the frontier $c_0 \rightarrow \neg \overline{\exists} c_i$.

Note any frontier below a c_0-successful (resp c_0-failed) frontier is also c_0-successful (resp c_0-failed). Given an arbitrary frontier $(c_1 \mid G_1), \ldots, (c_n \mid G_n)$ for goal $(c \mid G)$ there exists a most general constraint c_0 such that it is c_0-successful (possibly $false$), and a most general constraint c_0 such that it is c_0-failed.

We are now in a position to give two lemmas generalizing Lemmas 5.7 and 5.8

Lemma A.2 *Let R be a fair consistent computation rule. $(c \mid \neg G)$ has a c_0-successful frontier ($c_0 \rightarrow c$) iff $(c' \mid G)$ has a c_0-failed frontier, where c' are the constraints chosen by R for the negative sub-derivation.*

Proof: ($\Leftarrow$) Since $(c' \mid G)$ is c_0-failed then there exists a frontier $(c_1 \mid G_1), \ldots, (c_n \mid G_n)$ of the $(P, \mathcal{A}, R)$-derivation tree for $(c' \mid G)$ such that $c_0 \rightarrow \neg \overline{\exists} c_i$ for all $1 \leq i \leq n$. There exists a derivation of $(c \,\|\, \neg G)$ which corresponds to a frontier at or below $(c_1 \,\|\, G_1), \ldots, (c_n \,\|\, G_n)$, being the one that obtains constraints only from success nodes. The only success nodes $(d_1 \,\|\,), \ldots (d_m \,\|\,)$ are at or below the above frontier hence since $c_0 \rightarrow \neg\overline{\exists} d_j$ the derivation does not fail. Now examine the derivation extending this one that always obtains constraints. The constraints are always of the form $\neg\overline{\exists} e_k$ where $e_i \rightarrow c_i$. Hence $c_0 \rightarrow \neg\overline{\exists} c_i \rightarrow \neg\overline{\exists} e_i$. Thus the derivation does not fail. Eventually by the fairness of R it is successful. Now $c_0 \rightarrow \neg\overline{\exists} e_1 \wedge \cdots \wedge \neg\overline{\exists} e_l$ hence the derivation tree for $(c \,\|\, \neg G)$ has a c_0-successful frontier.

($\Rightarrow$) If $(c \,\|\, \neg G)$ is c_0-successful there exists successful derivations $(c \wedge \neg\overline{\exists} c_1^1 \wedge \cdots \wedge \neg\overline{\exists} c_1^{n_1} \,\|\,), \ldots (c \wedge \neg\overline{\exists} c_m^1 \wedge \cdots \wedge \neg\overline{\exists} c_m^{n_m} \,\|\,)$ such that $c_0 \rightarrow \neg\overline{\exists} c_1^1 \wedge \cdots \wedge \neg\overline{\exists} c_1^{n_1} \vee \cdots \vee \neg\overline{\exists} c_m^1 \wedge \cdots \wedge \neg\overline{\exists} c_m^{n_m}$. Each successful derivation corresponds to a frontier $(c_i^1 \,\|\, G_i^1), \ldots, (c_i^{n_i} \,\|\, G_i^{n_i})$ of $(c' \,\|\, G)$. If we take a frontier $\{(c'_1 \,\|\, G_1), \ldots, (c'_k \,\|\, G_k)\}$ of $(true \,\|\, G)$ that is below all of the m frontiers above then then $c_0 \rightarrow \neg\overline{\exists} c'_1 \wedge \cdots \wedge \neg\overline{\exists} c'_k$ and so $c_0 \rightarrow \neg c'_j$ for all $1 \leq j \leq k$. Thus $(c' \,\|\, G)$ has a c_0-failed frontier. □

Lemma A.3 *Let R be a fair consistent computation rule. $(c \,\|\, \neg G)$ has a c_0-failed frontier ($c_0 \rightarrow c$) iff $(c' \,\|\, G)$ has a c_0-successful frontier, where c' are the constraints chosen by R for the negative sub-derivation.*

Proof: ($\Leftarrow$) Since $(c' \,\|\, G)$ is c_0-successful there are successful derivations in $(c' \,\|\, G)$ leading to $(c_1 \,\|\,), \ldots, (c_n \,\|\,)$ such that $c_0 \rightarrow \overline{\exists} c_1 \vee \cdots \vee \overline{\exists} c_n$. Consider any derivation of $(c \,\|\, \neg G)$ either it is (a) finitely failed (and hence c_0-failed) or (b) successful or (c) it reaches a node where each negated goal in the node is below a frontier including all the successful derivations above (by Lemma 5.4). In case (b) the derivation ends in a goal of the form $(c \wedge \neg\overline{\exists} d_1 \wedge \cdots \wedge \neg\overline{\exists} d_k \,\|\,)$ where $d_1, \ldots d_k$ represent a complete frontier of $(c' \,\|\, G)$. Now for each $1 \leq i \leq n$ there exists d_j such that $\neg\overline{\exists} d_j \rightarrow \neg\overline{\exists} c_i$ and thus $c_0 \wedge \neg\overline{\exists} d_1 \wedge \cdots \wedge \neg\overline{\exists} d_k$ is unsatisfiable, contradiction. For case (c) the derivation reaches a node $(c \wedge \neg\overline{\exists} d_1 \wedge \cdots \wedge \neg\overline{\exists} d_k \,\|\,)$ such that the same conditions as in (b) hold. Thus all derivations of $(c \,\|\, \neg G)$ are c_0-failed.

($\Rightarrow$) Given that $(c \,\|\, \neg G)$ is c_0-failed consider the derivations which only gets constraints when there is no body. Constraints appearing in these derivations correspond to successful nodes $(c_1 \,\|\,), \ldots, (c_n \,\|\,)$ in the $(P, \mathcal{A}, R)$-derivation tree for $(c' \,\|\, G)$. Now each set of constraints $c_0 \wedge \neg\overline{\exists} c_1 \wedge \cdots \wedge \neg\overline{\exists} c_n$ is unsatisfiable, hence $(c' \,\|\, G)$ has successful derivations leading to $(c_1 \,\|\,), \ldots, (c_n \,\|\,)$ such that $c_0 \rightarrow \overline{\exists} c_1 \vee \cdots \vee \overline{\exists} c_n$ holds. Hence $(c \,\|\, G)$ has a c_0-successful frontier. □

We are now in a position to prove the correspondence of arbitrary fair consistent computation rules to DF computation rules.

Theorem A.4 *Let R be a DF-computation rule, let R' be any fair consistent computation rule. The $(P, \mathcal{A}, R)$-derivation tree for $(true \,\|\, G)$ is c_0-successful (resp c_0-failed) iff the $(P, \mathcal{A}, R')$-derivation tree for $(c \,\|\, G)$ is $c \wedge c_0$-successful (resp $c \wedge c_0$-failed).*

Proof: ($\Rightarrow$) By induction on the depth of negative sub-derivation. In the base case there are no negative goals selected in the $(P, \mathcal{A}, R)$-derivation tree for $(true \,\|\, G)$ hence any successful derivations do not depend on them. Using a similar argument to that of [Wolfram *et al*(1984)] we can show any successful derivations in the $(P, \mathcal{A}, R)$-derivation tree for $(true \mid G)$ also appear in the $(P, \mathcal{A}, R')$-derivation tree for $(c \mid G)$ unless the constraints contradict c. Thus the $(P, \mathcal{A}, R')$-derivation tree for $(c \mid G)$ is $c \wedge c_0$-successful.

Similarly each $(P, \mathcal{A}, R')$-derivation of $(c \mid G)$ is either failed or eventually selects all the atoms appearing in a $(P, \mathcal{A}, R)$-derivation of $(true \mid G)$. Hence each such derivation must be c_0-failed or c-failed, and hence $c \wedge c_0$-failed.

For the induction step notice that if $\neg G'$ is a selected negative subgoal in the derivation of $(true \mid G)$ then its contribution to the constraint-success or failure frontiers of $(true \mid G)$ is either constraints c_0 for which it has a c_0-successful frontier, or constraints c_0 for which it has a c_0-failed frontier. Using the above Lemmas, if $(true \mid \neg G')$ has a c_0-successful (resp. c_0-failed) frontier then $(true \mid G')$ has a c_0-failed (resp. c_0-successful) frontier. By induction $(c' \mid G')$ has a $c' \wedge c_0$-failed (resp $c' \wedge c_0$-successful) frontier for any computation rule R'. And again by the above Lemmas $(c \mid \neg G')$ has a $c' \wedge c_0$-successful (resp. $c' \wedge c_0$-failed) frontier. Since $c \rightarrow c'$, $(c \mid \neg G')$ has a $c \wedge c_0$-successful (resp. $c \wedge c_0$-failed) frontier.

Given this relationship we can using similar reasoning to the base case to show that the successful $(P, \mathcal{A}, R)$-derivation of $(true \mid G)$ leading to $(c_i \mid\,)$ has a corresponding successful $(P, \mathcal{A}, R')$-derivation tree for $(c \mid G)$ unless $c_i \wedge c$ is unsatisfiable. This occurs since we can determine a corresponding derivation that select the same atoms and initial negative subgoals and when selecting the initial negative subgoals (those that are not the result of a selection of a negative subgoal) regardless of the constraints passed into the negative sub-derivation we eventually achieve a $c \wedge c_i$-successful derivation.

For c_0-failure, by fairness each $(P, \mathcal{A}, R')$-derivation of $(c \mid G)$ is either failed or eventually selects all the subgoals selected in a $(P, \mathcal{A}, R)$-derivation of $(true \mid G)$. Again the negative subgoals selected eventually reach a frontier which is $c \wedge c_0$-failed.

($\Leftarrow$) Again by induction on the depth of negation. For the base case by similar reasoning to the above base case if $(c \mid G)$ is $c \wedge c_0$-successful using computation rule R' then $(true \mid G)$ is also $c \wedge c_0$-successful. Similarly if $(c \mid G)$ is $c \wedge c_0$-failed then $(true \mid G)$ is also $c \wedge c_0$-failed by the fairness of DF computation rules.

For the induction step notice that if $\neg G'$ is a selected negative subgoal in the derivation of $(c \mid G)$ then its contribution to the constraint-success or failure frontiers of $(c \mid G)$ is either constraints c_0 for which it has a $c \wedge c_0$-successful frontier, or constraints c_0 for which it has a c_0-failed frontier. Using the above Lemmas, if $(c_1 \mid \neg G')$ has a $c \wedge c_0$-successful (resp. $c \wedge c_0$-failed) frontier then $(c' \mid G')$ has a $c \wedge c_0$-failed (resp. $c \wedge c_0$-successful) frontier. By induction $(true \mid G')$ has a $c' \wedge c_0$-failed (resp $c' \wedge c_0$-successful) frontier for DF computation rule R. And again by the above Lemmas $(true \mid \neg G')$ has a $c' \wedge c_0$-successful (resp. $c' \wedge c_0$-failed) frontier. Since $c \rightarrow c'$, $(true \mid \neg G')$ has a $c \wedge c_0$-successful (resp. $c \wedge c_0$-failed) frontier. $\square$

Corollary A.5 *Let R be a DF-computation rule, let R' be any fair consistent computation rule. $(c \mid G)$ has a totally successful (resp. finitely failed) $(P, \mathcal{A}, R)$-derivation tree iff it has a totally successful (resp. finitely failed) $(P, \mathcal{A}, R')$-derivation tree.* $\square$