Foundational Issues in Implementing Constraint Logic Programming Systems

James H. Andrews
Dept. of Computing Science
Simon Fraser University
Burnaby, BC, Canada V5A 1S6
jamie@cs.sfu.ca

Abstract: Implementations of Constraint Logic Programming (CLP) systems are often incomplete with respect to the theories they are intended to implement. This paper studies two issues that arise in dealing with these incomplete implementations. First, the notion of incomplete "satisfiability function" (the analogue of unification) is formally defined, and the question of which such functions are reasonable is studied. Second, techniques are given for formally (proof-theoretically) specifying an intended CLP theory or a characterizing an existing CLP system, for the purpose of proving soundness and completeness results. Notions from linear logic and the notion of Henkinness of the theory are shown to be important here.

1 Introduction

The semantics of Constraint Logic Programming (CLP) languages is now well-understood. Implementations of CLP languages, however, are often not complete with respect to their intended semantics. In this paper, I study the theory of such incomplete implementations, by giving a recursion-theoretic, rather than model-theoretic, basis for CLP operational semantics. Based on this work, I then study techniques for specifying the *intended* theory of a CLP language, or giving characterizations of the *actual* theory implemented by a CLP language.

A simple example shows that some CLP implementations are necessarily incomplete. Consider a first order language, structure and theory in which the terms encode lambda-expressions, and in which equality $=$ holds between two terms iff they are $\beta\eta$-equivalent. The theory can be made satisfaction-complete, and the structure is solution-compact. The CLP scheme [JL87] defines a theoretical operational semantics for this theory such that a ground query $\mathbf{s} = \mathbf{t}$ fails iff $\mathbf{s}$ and $\mathbf{t}$ are not $\beta\eta$-equivalent. However, in practice we have no complete algorithm for testing whether two lambda-terms are $\beta\eta$-equivalent; so the operational semantics cannot be implemented completely, not even by using the standard breadth-first technique of complete Herbrand-domain logic programming interpreters.

Furthermore, many implementations of CLP languages are incomplete for efficiency reasons; for instance, CLP(R) [JMSY92] implements an efficient but incomplete linear equation solver rather than Tarski's complex algorithm for deciding real arithmetic.

For these kinds of situations, we need to develop a theoretical framework, incorporating notions of computability, in which we can study issues of how

complete a CLP implementation is. This framework can then be used as a basis for comparing a CLP implementation with its intended theory, or specifying the smaller theory that an incomplete implementation actually implements. This paper is intended to make steps in the direction of such a framework.

In section 2, I present some basic definitions. In section 3, I construct CLP operational semantics based on a class of recursive functions called "satisfiability functions", which formalize the basic step in CLP languages (the analogue of unification). I also define what it means for a structure to "realize" a satisfiability function, and give a necessary and sufficient, non-logical condition for such a function to be realizable. In section 4, I study various techniques for characterizing CLP theories with proof systems, and show how they could be used to prove the soundness and completeness of implementations. In section 5, I present some conclusions and discuss related work.

2 Basic Definitions

Definition 2.1 A *first order language* $\mathcal{L}$ is a tuple $\langle F, P, V \rangle$, where F is a recursive set of function symbols, each with an associated arity, P is a recursive set of predicate names, each with an associated arity, and V is a recursive set of variable names.

Let C be a set of predicate names of $\mathcal{L}$, $C \subseteq P(\mathcal{L})$. Constrs$(\mathcal{L}, C)$ is the set of all predicate application formulae from $\mathcal{L}$ formed using a predicate name from C. We also call these predicate application formulae "constraints".

We define the terms and formulae of $\mathcal{L}$ in the standard logical way; likewise the notions of *structure, valuation, satisfiability w.r.t. a structure, model,* and *theory.*

3 Satisfiability Functions

The basic step in constraint logic programming interpreters (even incomplete ones) is the step which decides whether a new constraint is consistent with the previously-processed constraints. Every CLP interpreter has an algorithm for doing this for its intended theory; if the algorithm returns "true", the interpreter goes further down the same branch in the search tree, and if it returns "false", the interpreter backtracks[1]. This section studies the theory of such "satisfiability functions", which will be defined as *partial* recursive functions from finite sets of constraints to results *including* "true" and "false".

In the first subsection, I define the notion of satisfiability function, and show how an operational semantics can be built on the basis of that notion (rather than the notion of constraint theory) in order to ensure computability. In the

[1] This does not necessarily describe completely the operation of all languages referred to as constraint logic programming languages. (For instance, it does not take into account disjunctive constraints.) However, it describes one common framework for CLP operational semantics, and is also the one defined in the standard literature on CLP, e.g. [Mah93].

second subsection, I point out that not all satisfiability functions correspond to actual constraint theories which "realize" them, and thus that the operational semantics arising from them do not define rational logic programming systems. I give, however, a condition on satisfiability functions which is necessary and sufficient for realizability. In the final subsection, I point out that for a given constraint theory, there are either either 0 or 1 maximal satisfiability functions which are realized by it.

3.1 Satisfiability Functions and Operational Semantics

Definition 3.1 A *satisfiability function* (in a language $\mathcal{L}$ with constraint predicates C) is a general recursive function from finite sets of constraints in Constrs($\mathcal{L}, C$) to a set $\mathcal{T} \supseteq \{true, false\}$.

This definition of satisfiability function is general enough to capture the behaviour of a wide variety of complete and incomplete implementations. We always interpret a result of *false* as "unsatisfiable" and *true* as "satisfiable"; but other results, or no result, is also possible. To capture the very well-behaved satisfiability functions, we make the following definition:

Definition 3.2 A *strict* satisfiability function is one which is never undefined and always returns *true* or *false*.

The satisfiability function associated with basic Prolog, for instance, is strict. Here is a non-strict example.

Example. The basic behaviour of the CLP(R) system [JMSY92] on real-number constraints can be characterized with the following (non-strict) satisfiability function *sat*.

- If S contains a subset T consisting of unsatisfiable, linear constraints, $sat(S)$ returns *false*.
- Otherwise, if S contains no non-linear constraints, $sat(S)$ returns *true*.
- Otherwise (i.e. S contains non-linear constraints but its linear constraints are satisfiable), $sat(S)$ returns *unsure*.

□

In basing an operational semantics on a satisfiability function *sat*, we may want to give a definition that takes into account the various truth values which can act as results of *sat*. We must make the following minimum requirements, following Maher [Mah93].

Definition 3.3 Given a satisfiability function *sat* on $(\mathcal{L}, C)$ whose range is the truth values in $\mathcal{T}$, a binary relation $\rightarrow$ is an *operational transition relation* for *sat* with program P if:

1. It is a relation between *states*, which are either truth values from $\mathcal{T}$ or pairs $\langle G, C\rangle$, where G is a multiset of non-constraint atoms and C is a multiset of constraints;

2. The relation includes the transition

$$\langle G \cup H\theta, C\rangle \rightarrow \langle G \cup B\theta, C \cup C'\theta\rangle$$

if the program P contains (some renaming of) the clause $(H \leftarrow C', B)$, and $sat(C \cup C'\theta) \simeq true$; and

3. The relation includes the transition

$$\langle G \cup H, C\rangle \rightarrow false$$

if for every (renamed) clause in P of the form $(H' \leftarrow C', B)$ and substitution θ such that $H'\theta$ is H, we have that $sat(C \cup C'\theta) \simeq false$.

We say that a goal G *succeeds* if $\langle G, \emptyset\rangle \rightarrow^* \langle \emptyset, C\rangle$ and $sat(C) \simeq true$; we define *fair* derivations in the usual way and say that G *fails* if every fair derivation ends in the state $false$. Note that if *sat* is strict, then the above definition will yield a unique transition relation, similar to that defined by Maher [Mah93].

Example. Based on the definition of *sat* for CLP(R) above, we can characterize an operational semantics for (an idealized version of a part of) CLP(R) as follows. The transition relation is the unique transition relation for (sat, P) having the additional property that:

- We have the transition

$$\langle G \cup H\theta, C\rangle \rightarrow \langle G \cup B\theta, C \cup C'\theta\rangle$$

if the program P contains (some renaming of) the clause $H \leftarrow C', B$, and $sat(C \cup C'\theta) \simeq unsure$.

□

In this operational semantics, even if we are unsure of the satisfiability of the resulting set of constraints (if the system of equations is not linear), we go on as if it were satisfiable. We may wish to say that a goal G is *indeterminate* if it does not fail, but every fair derivation ends in either $false$ or $unsure$.

3.2 Realizable Satisfiability Functions

We would like our operational semantics to define sensible logic programming systems, and to achieve that we have to put a condition on the satisfiability functions we use. The condition is "realizability", and is best defined model-theoretically. In this subsection, we study realizability and give an equivalent condition, "reliability", which is stated in terms independent of model theory.

Definition 3.4 An $\mathcal{L}$-structure $\Re$ *realizes* a satisfiability function *sat* if:

1. whenever $sat(S) \simeq true$, S is $\Re$-satisfiable; and
2. whenever $sat(S) \simeq false$, S is not $\Re$-satisfiable.

If $\Re$ realizes *sat*, we also say that *sat implements* $\Re$.

Not every satisfiability function is realizable, not even the strict ones. For instance, if we have a satisfiability function which maps $\{p(x)\}$ onto *false* but $\{p(x), q(x)\}$ onto *true*, then this will not have any realizing structure, because any valuation satisfying $\{p(x), q(x)\}$ will surely satisfy $\{p(x)\}$. An operational semantics based on such a satisfiability function would give unexpected results.

The realizability of *sat* can be given an equivalent characterization in terms of the theory associated with *sat*.

Definition 3.5 Θ_{sat}, the *theory associated with sat*, is defined as

$$\{\exists[S] \mid sat(S) \simeq true\} \cup \{\neg\exists[S] \mid sat(S) \simeq false\}$$

where $\exists[S]$ is the existential closure of the conjunction of the constraints in S.

We have the following simple proposition:

Proposition 3.6 *sat* is realizable iff Θ_{sat} is consistent.

However, for the purposes of checking a given implementation of a satisfiability function, it would be better to have a more direct method of testing whether it is realizable. The condition called "reliability" allows us to do this. First, some technical definitions.

Definition 3.7 Let S be a set of constraints and let $c \in S$. The *variable sharing class* $S|_c$ is the smallest subset of S such that:

- $c \in S|_c$;
- if $b \in S|_c$, and $a \in S$ shares a free variable with b, then $a \in S|_c$.

Definition 3.8 Let *sat* be a satisfiability function. A set of constraints S is *sat-covered* if for all $c \in S$, there is a set $T \supseteq S|_c$ such that $sat(T) \simeq true$. A set of constraints S is *sat-consistent* if there is some substitution θ such that $S\theta$ is *sat*-covered.

Basically, a set S is *sat*-consistent if, for some θ, every element of this partition of $S\theta$ is satisfiable in any structure realizing *sat*.

Definition 3.9 A satisfiability function *sat* is *reliable* if whenever $sat(S) \simeq false$, S is not *sat*-consistent.

For non-reliable satisfiability functions, some sets S are considered unsatisfiable despite the fact that some instance of S can be partitioned into sets which are generalizations of sets considered satisfiable. This is a situation which does not meet with our intuitions, and indeed the next theorem proves that reliability is a necessary condition for realizability.

Theorem 3.10 If a satisfiability function *sat* is realizable, then it is reliable.

Proof (sketch). Assume (toward a contradiction) that *sat* is realized by some $\Re$ but not reliable. There must be an S and θ such that $sat(S) \simeq false$ but $S\theta$ is *sat*-covered. Each variable sharing class T_i in $S\theta$ results in a satisfying valuation v_i; but the union of these v_i's is a valuation satisfying $S\theta$, contradicting the assumption that $\Re$ realizes *sat*.
□

So reliability is necessary for realizability. It is also sufficient, as we will see next.

Theorem 3.11 If a satisfiability function *sat* is reliable, then it is realizable.

Proof (sketch). By Proposition 3.6, it is sufficient to prove that if *sat* is reliable, Θ_{sat} is consistent. By compactness, it is therefore sufficient to prove that every finite subset of Θ_{sat} is consistent.

Let $\mathbf{S}$ be a finite subset of Θ_{sat}, where

$$\mathbf{S} = \{\exists[S_1], \ldots, \exists[S_j]\} \cup \{\neg\exists[T_1], \ldots, \neg\exists[T_k]\}$$

Let $\Re$ be the minimal structure which contains a unique element for every free variable in all the S_i's, and in which each S_i is satisfiable by a valuation mapping variables to these elements. Clearly this structure is a model of the positive formulae in $\mathbf{S}$; we have only to prove that it is a model of the negated formulae too.

Assume, toward a contradiction, that T_i is satisfied by some valuation v in $\Re$. There is some θ and v' such that $v = \theta v'$, v' maps free variables directly to domain elements, and $T_i\theta$ is satisfied by v'. But by the construction and minimality of $\Re$, this means that $T_i\theta$ can be partitioned into sets which are free-variable variants of subsets of the S_i's. T_i is therefore *sat*-consistent; but we know that $sat(T_i) \simeq false$, thus contradicting the assumption that *sat* was reliable.
□

The upshot of this is that if we base operational semantics on satisfiability functions, they are by definition computable; but we are still able to give assurances that they define sensible systems, by proving that the satisfiability function is reliable.

3.3 Maximality Results

Finally, some words about maximality. Given a particular constraint theory, what is the "biggest" satisfiability function which implements it? It turns out that either a theory has a complete, strict satisfiability function, or else there is no maximal satisfiability function which implements it.

We can define maximality by comparing the completeness of two satisfiability functions with respect to a structure.

Definition 3.12 Let sat_1 and sat_2 both implement $\Re$. We say $sat_1 \sqsubseteq_{\Re} sat_2$, in words "$sat_1$ is a better approximation to $\Re$ than sat_2", if:
1. Whenever $sat_1(S) \simeq true$ we have that $sat_2(S) \simeq true$; and
2. Whenever $sat_1(S) \simeq false$ we have that $sat_2(S) \simeq false$.

A satisfiability function *sat* is a *maximal approximation* to $\Re$ if there is no satisfiability function *sat'* such that $sat' \neq sat$ and $sat \sqsubseteq_{\Re} sat'$.

Theorem 3.13 A structure $\Re$ has either 0 or 1 maximal approximations.

Proof. If the set S' of satisfiable finite sets of constraints of $\Re$ is recursive, then clearly there is a unique, strict, maximal approximation to $\Re$. Otherwise, for any *sat* which implements $\Re$, we can build a better approximation *sat'* by allowing more sets to return *true* or *false*.
□

Thus for the example structure given in the Introduction ($\beta\eta$-equivalence of lambda expressions), there is no maximal implementation. Whenever we have a satisfiability function (and thus a constraint logic programming system) which implements this structure, we can always do better.

4 Specification and Characterization with Proof Systems

We have seen that incomplete implementations of CLP languages are sometimes desirable or even necessary. We have also seen that such incomplete implementations can be given a coherent theoretical basis. But in order to make practical use of incomplete implementations, we need to be able to compare them directly with descriptions of theories.

To do so, we really need *formal* (syntactic) descriptions of the theories to be compared to; the kinds of informal descriptions found in the literature are sometimes too imprecise to be used in formal proofs, and this imprecision is multiplied when we have several groups of interacting constraints (Herbrand, rational tree, integer, real, etc.). There are at least two other practical reasons for developing formal descriptions of CLP languages:

- With increasing prominence of constraint systems, it will become necessary to develop some standard formalism for describing constraint theories, much as BNF was developed to describe programming language syntax.
- Syntactic, or more specifically logical, characterizations of constraint theories will be absolutely necessary to any program-logic system which intends to prove properties of constraint logic programs.

One possible framework for such formal descriptions is proof theory, which has been used to good advantage in the past to describe basic logic programming [HS84, HSH90, MNPS91]. The use of proof theory as a general framework for characterizing CLP, which has not to my knowledge been studied before, is the topic of this section.

There are two ways in which proof-theoretic characterizations could be used:
(a) to *specify* an intended theory; and
(b) to *characterize* an existing implementation.
Examples of (a) would include giving an axiomatization of Horn clauses with Presburger arithmetic, to which we could then compare individual CLP implementations. Examples of (b) would include giving a proof-theoretic characterization of the CLP(R) implementation [JMSY92], to see how it looks compared to an axiomatization of real arithmetic.

In this section, I will first discuss some techniques we could use to give proof-theoretic descriptions of CLP theories. One of the major issues that will emerge is the question of whether or not the theory is "Henkin" (has a closed "witness" term for every existential truth). I will then give examples of how such descriptions could be used in proving soundness and completeness of implementations, and in characterizing existing CLP implementations. It will turn out that ideas from linear logic [Gir87] will be relevant to this last point.

4.1 Description Techniques

Here I give two distinct techniques that we might use to describe constraint theories. The first technique is fairly simple but applicable only to Henkin theories; the second is more complex but applicable to both Henkin and non-Henkin theories.

4.1.1 Closed Constraint Technique

In this technique, we characterize the entire theory by characterizing the truth or falsehood of *closed* (ground) constraints. We then rely on proof rules for existential quantification to handle free variables. I will give two examples, then discuss the technique in general.

Example: Herbrand universe logic programming. Proof-theoretic characterizations of basic logic programming have been studied for years. Consider the following example, similar to the system of Miller et al. [MNPS91]. The syntax of goals (G) and definitions (D) is as follows.

$$G ::= t_1 = t_2 \mid p(t_1, \ldots, t_n) \mid G \& G \mid G \vee G \mid \exists x G$$

$$D ::= \forall x D \mid p(x_1, \ldots, x_n) \leftarrow G$$

Sequents are of the form

$$D_1, \ldots, D_m \vdash G$$

and are intended to express "the goal G follows from the definitions $D_1, \ldots, D_m$." Sequents can be given formal derivations by using the proof rules in Figure 1.

The rules of the proof system both act as a formal and unambiguous description of truth, and give us some intuitive sense of the meaning of the connectives ($G_1 \& G_2$ follows from Γ if both G_1 and G_2 follow from Γ, and so on). We

$$\frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1 \& G_2} \qquad \frac{\Gamma \vdash G_1}{\Gamma \vdash G_1 \vee G_2} \qquad \frac{\Gamma \vdash G_2}{\Gamma \vdash G_1 \vee G_2}$$

$$\frac{\Gamma \vdash G[x := t]}{\Gamma \vdash \exists x(G)} \qquad \frac{}{\Gamma \vdash t = t}$$

$$\frac{D, D, \Gamma \vdash G}{D, \Gamma \vdash G} \qquad \frac{D[x := t], \Gamma \vdash G}{\forall x D, \Gamma \vdash G} \qquad \frac{\Gamma \vdash G}{(p(t_1, \ldots, t_n) \leftarrow G), \Gamma \vdash p(t_1, \ldots, t_n)}$$

Figure 1: A proof-theoretic characterization of a basic logic programming language. Γ is any sequence of definitions.

$$\frac{\Gamma \vdash a = b}{\Gamma \vdash b = a} \qquad \frac{\Gamma \vdash a = b \quad \Gamma \vdash b = c}{\Gamma \vdash a = c} \qquad \frac{\Gamma \vdash a = b}{\Gamma \vdash s(a) = s(b)}$$

$$\frac{}{\Gamma \vdash a + 0 = a} \qquad \frac{}{\Gamma \vdash a + s(b) = s(a + b)}$$

Figure 2: Additional rules for a proof-theoretic characterization of a Presburger arithmetic constraint logic programming language.

can prove the soundness and completeness of an SLD-resolution operational semantics with respect to this proof system by two relatively simple structural inductions (e.g. [And89]).
□

Note that we had only to characterize the behaviour of equality formulae, and that we did so by saying that two terms are equal if they are identical. (We must make the assumption that there is at least one closed term in the language, but this is reasonable.)

Example: Presburger arithmetic. Consider the standard language and theory of Presburger arithmetic (i.e., integers constructed from 0 and the successor function $s(\_)$, with the arithmetic operation $+$ and the relation $=$). A characterizing proof system can be constructed taking the one for Herbrand logic programming, and adding some more rules to make it a complete proof system for Presburger arithmetic sequents of the form $D_1, \ldots, D_m \vdash G$ (see for example [Kle52]). Such additional rules are shown in Figure 2.
□

This specifying proof system does not tell us *how* to solve such constraints or how to construct our implementation; it merely gives a formal and intuitive description of what kinds of constraints we want to solve. It thus acts as a specification of a CLP language for solving Presburger arithmetic constraints. Any implementation, however, would presumably use Presburger's algorithm [Mon76] or some variant of it.

$$\frac{A, \Gamma \vdash G}{\Gamma \vdash G} \qquad \frac{B[x := y], \Gamma \vdash G}{\exists x B, \Gamma \vdash G} \qquad \frac{B, C, \Gamma \vdash G}{B \& C, \Gamma \vdash G} \qquad \frac{x = t, \Gamma \vdash G[z := t]}{x = t, \Gamma \vdash G[z := x]}$$

Figure 3: Additional rules for a proof-theoretic characterization of a rational-trees CLP language. A is a WRTA, and y is a new variable.

Again, we had only to give additional rules for the behaviour of closed arithmetic formulae; essentially, we have expanded the meaning of $=$ from identity between closed terms (the $t = t$ axiom) to arithmetic equality of closed terms (the $t = t$ axiom plus the new rules and axioms).

For theories like the ones given, the task of coming up with a proof-theoretic specification can be reduced to coming up with a proof-theoretic specification of the valid closed constraints. The essential property here is that the intended theory is *Henkin* [Sho67]; that is, whenever $\exists x B$ is true, there is a closed term t such that $B[x := t]$ is true[2].

Unfortunately, however, not all interesting constraint theories are Henkin. A simple example is the theory of rational trees: we have that $\exists x(x = f(x))$ is true, but there is no closed t such that $t = f(t)$. Another important example is the theory of real arithmetic, in which we have no closed term t such that $t \times t = 2$. For these theories, we have to use other methods.

4.1.2 Axiomatic Technique

Another technique for specification is to provide axioms which can be added to the left-hand side of a sequent in order to obtain proofs of solvable queries. Unlike the closed-constraint approach, this technique is universal.

Example: Rational trees. Van Emden and Lloyd [vEL84] and Maher [Mah88] have given Hilbert-style logical descriptions of the theory of rational trees, with varying soundness and completeness properties. The logical consequences of these theories include all formulae of the form

$$\exists x_1 \ldots \exists x_n((x_1 = t_1) \& \ldots \& (x_n = t_n))$$

where the x_i's are distinct and the t_i's are terms whose variables are among the x_i's. Let us call these the *weak rational tree axioms (WRTAs)*.

Using WRTAs, we can build a sequent-calculus characterization of a rational-trees CLP language. It consists of the rules in Figure 1 with the addition of the rules in Figure 3. An example derivation of $\exists x(x = f(f(x))$ can be found in Figure 4.

□

[2] A more precise and traditional definition of Henkin theories is those theories T for which, for every sentence $\exists x A$ of the language, there is a constant e in the language such that $T \models (\exists x A) \supset (A[x := e])$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y = f(y) \vdash f(f(y)) = f(f(y))}}{y = f(y) \vdash f(y) = f(f(y))}}{y = f(y) \vdash y = f(f(y))}}{y = f(y) \vdash \exists x(x = f(f(x)))}}{\dfrac{\exists x(x = f(x)) \vdash \exists x(x = f(f(x)))}{\vdash \exists x(x = f(f(x)))}}$$

Figure 4: Example derivation of $\exists x(x = f(f(x))$ in the theory of rational trees.

Given any satisfiability function *sat*, we can develop a proof-theoretic characterization $\vdash$ such that $\Gamma \vdash G$ if G succeeds with respect to Γ; we can do this by giving rules which allow us to add any element of Θ_{sat} to the left-hand side. This $\vdash$ does not necessarily have the converse property – that if $\Gamma \vdash G$ then G succeeds with respect to Γ – but as we will see later, we can achieve such results with the use of techniques from linear logic.

4.2 Proving Properties of Implementations

We are now in a position to tie together the theory of satisfiability functions with the techniques for description of CLP theories. Say we are given a satisfiability function *sat* on $(\mathcal{L}, C)$ and a transition relation $\rightarrow$ for *sat*. We are also given a proof system intended to describe the intended theory in a logical form. The kinds of results we would want to prove are:

- *Soundness of success:* if $\langle G, \emptyset\rangle \rightarrow^* \langle \emptyset, C\rangle$ w.r.t. program P, where $sat(C) \simeq true$, then $P \vdash G$.
- *Soundness of failure:* if $\langle G, \emptyset\rangle \rightarrow^*$ w.r.t. program P, then $P \nvdash G$.
- *Completeness of success:* if $P \vdash G$, then $\langle G, \emptyset\rangle \rightarrow^* \langle \emptyset, C\rangle$ w.r.t. P, where $sat(C) \simeq true$.
- *Completeness of failure:* if $P \nvdash G$, then $\langle G, \emptyset\rangle \rightarrow^* false$.

Clearly, if *sat* is strict and we can prove soundness and completeness of success, or soundness of success and failure, then we can prove the rest of the results.

I will just note here that the soundness and completeness of Prolog II with respect to the proof system given in the last subsection can be proven by techniques similar to those given in [Col83, Mah88, And89]. I will go into more detail about the CLP(R) example from Section 3.1.

Example: a characterization of CLP(R). Consider the satisfiability function *sat* and transition relation $\rightarrow$ we defined for CLP(R) in Section 3.1. We can define a characterizing proof system for this operational semantics using the axiomatic technique and some ideas from linear logic [Gir87].

$$\frac{\Gamma_1, !\Gamma \vdash G_1 \quad \Gamma_2, !\Gamma \vdash G_2}{\Gamma_1, \Gamma_2, !\Gamma \vdash G_1 \& G_2} \qquad \frac{\Gamma \vdash G_1}{\Gamma \vdash G_1 \vee G_2} \qquad \frac{\Gamma \vdash G_2}{\Gamma \vdash G_1 \vee G_2}$$

$$\frac{\Gamma \vdash G[x := t]}{\Gamma \vdash \exists x(G)} \qquad \frac{}{c, !\Gamma \vdash c}$$

$$\frac{!D, D, \Gamma \vdash G}{!D, \Gamma \vdash G} \qquad \frac{D[x := t], \Gamma \vdash G}{\forall x D, \Gamma \vdash G} \qquad \frac{\Gamma \vdash G}{(p(t_1, \ldots, t_n) \leftarrow G), \Gamma \vdash p(t_1, \ldots, t_n)}$$

$$\frac{\exists [c_1 \& \cdots \& c_k], \Gamma \vdash G}{\Gamma \vdash G} \qquad \frac{B[x := y], \Gamma \vdash G}{\exists x B, \Gamma \vdash G} \qquad \frac{B, C, \Gamma \vdash G}{B \& C, \Gamma \vdash G}$$

Figure 5: A proof-theoretic characterization of CLP(R). Γ is any sequence of formulae, $!\Gamma$ is a sequence of formulae preceded by !, y is a new variable, and $c_1, \ldots, c_k$ are constraints such that $sat(\{c_1, \ldots, c_k\}) \simeq true$.

The proof system is given in Figure 5. The important ideas in this proof system are as follows.

- As in the axiomatic characterization of rational trees, we allow axioms about satisfiable systems of equations to be introduced and manipulated in the antecedent (left-hand side) of a sequent.
- In the antecedent, we precede each rule from the program with the linear logic ! operator. This is an operator which allows for duplication of assumptions; for formulae in the antecedent not preceded by !, we do not allow duplication. (The antecedent is viewed as a multiset.)
- We require that at the top of the proof be axiomatic sequents of the form $c, !\Gamma \vdash c$, where $!\Gamma$ is a sequence of formulae *all* of which are preceded by !.
- We require that the sequence of formulae not preceded by ! be *split* across the two premisses of the rule introducing & on the right.

All this has the effect of forcing each assumption arising from the *sat* rule to be used once and only once in the course of a proof.
□

An example proof is given in Figure 6. Note that the proof would not have gone through if the query had been simply $\exists x, y(x \cdot y = 4)$, because we would have had the extra formula $y' = 2$ to contend with. Conversely, if we had allowed axioms to be simply of the form $(c, \Gamma \vdash c)$, we would have been able to prove $\exists x, y(x \cdot y = 4)$, even though that is not a linear equation and thus not solvable in the operational semantics. As it is, with respect to this proof system, we should be able to prove that the operational semantics has the properties of soundness and completeness of success and soundness of failure. Because *sat* is not strict in this case, however, we cannot prove completeness of failure.

This characterizing proof system is somewhat unwieldy due to the fact that the real-number axioms introduced in the antecedent may be huge. It may be

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y'=2,!\forall z(p(z)\leftarrow z=2)\vdash y'=2)}}{y'=2,!\forall z(p(z)\leftarrow z=2),(p(y')\leftarrow y'=2)\vdash p(y')}}{y'=2,!\forall z(p(z)\leftarrow z=2),\forall z(p(z)\leftarrow z=2)\vdash p(y')}}{y'=2,!\forall z(p(z)\leftarrow z=2)\vdash p(y')}}{x'\cdot y'=4,y'=2,!\forall z(p(z)\leftarrow z=2)\vdash x'\cdot y'=4\&p(y')}}{x'\cdot y'=4,y'=2,!\forall z(p(z)\leftarrow z=2)\vdash \exists y(x'\cdot y=4\&p(y))}}{x'\cdot y'=4,y'=2,!\forall z(p(z)\leftarrow z=2)\vdash \exists x,y(x\cdot y=4\&p(y))}}{(x'\cdot y'=4\&y'=2),!\forall z(p(z)\leftarrow z=2)\vdash \exists x,y(x\cdot y=4\&p(y))}}{\exists y'(x'\cdot y'=4\&y'=2),!\forall z(p(z)\leftarrow z=2)\vdash \exists x,y(x\cdot y=4\&p(y))}}{\exists x',y'(x'\cdot y'=4\&y'=2),!\forall z(p(z)\leftarrow z=2)\vdash \exists x,y(x\cdot y=4\&p(y))}}{!\forall z(p(z)\leftarrow z=2)\vdash \exists x,y(x\cdot y=4\&p(y))}$$

Figure 6: An example proof in the characterizing proof system for CLP(R).

possible to simplify the axioms, but even as it stands the proof system can be used for proving properties of CLP(R) programs.

The linear logic technique may be useful for characterizing other incomplete CLP-like languages. Some implementations of finite-domain constraint solvers are incomplete for efficiency reasons [Mac85]; and many implementations of Prolog use unification without occurs check, which is sound with respect to the rational-tree axioms but which sometimes goes into an infinite loop.

5 Conclusions and Related Work

I have shown that the class of satisfiability functions adequately characterizes the behaviour of a wide variety of implementations of CLP languages, and that there is a simple, non-model-theoretic condition ("reliability") for testing whether a satisfiability function is reasonable.

I have also discussed techniques for specifying and characterizing CLP systems with sequent calculi. I have pointed out that the question of whether the theory is Henkin is important, and that the notation and proof theory of linear logic (or other such "substructural" logics) can help in characterization.

The definition of a reliable satisfiability function is closely related to Scott's definition of an *information system* [Sco82]. However, neither the space of satisfiability functions, nor the space of information systems (under a reasonable mapping from one notion to the other), are proper subsets of the other.

Höhfeld and Smolka [HS88] and Frühwirth [Frü92] have both explored the idea of formally describing constraint theories. Höhfeld and Smolka describe an alternative framework to Jaffar and Lassez's for constraint systems; like Jaffar and Lassez, however, they do not consider explicitly any computability restrictions on constraint satisfaction algorithms. Frühwirth gives a Horn-clause-based language for defining constraint simplification rules, or SiRs, for any given domain. However, while SiRs have a logical form, they do not necessarily take the form of a simple and intuitive axiomatization or proof system.

There are several directions for future work in this area:

- Case studies. I would very much like to see these ideas applied for the purpose of fully and precisely characterizing existing, practical systems.
- Negation. I have avoided talking about negation in this paper because it poses general problems for logic programming theory which have not been adequately answered yet. A framework which characterizes the failure of constraint queries as well as their success would be desirable.
- Moving toward a standard description language. It would be premature at this point to propose some standard for describing constraint systems, but this would bring many benefits if done, much as BNF brought a standard manner of describing programming language syntax.

6 Acknowledgements

I appreciate the helpful comments and suggestions I have received from Veronica Dahl, Alistair Lachlan, Sanjeev Mahajan, Fred Popowich, Stephan Wehner (all of SFU), Thom Frühwirth, Nevin Heintze, Tim Hickey, and Gert Smolka, as well as Torkel Franzen and the anonymous referees. This research has been supported by the Natural Sciences and Engineering Research Council of Canada, the SFU Centre for Systems Science, and the SFU President's Research Grant Committee, via Infrastructure Grants NSERC 06-4231, CSS 02-7960 and PRG 02-4028, Equipment Grants NSERC 06-4232, CSS 02-7961 and PRG 02-4029, Operating Grants OGP0002436 (Dahl) and OGP0041910 (Popowich), and the author's NSERC Postdoctoral Fellowship.

References

[And89] James H. Andrews. Proof-theoretic characterisations of logic programming. In *Mathematical Foundations of Computer Science*, volume 379 of *Lecture Notes in Computer Science*, pages 145–154, Porąbka-Kozubnik, Poland, 1989. Springer.

[Col83] Alain Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, 1983.

[Frü92] Thom Frühwirth. Constraint simplification rules. Technical Report 92-18, ECRC, Munich, Germany, July 1992.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[HS84] Masami Hagiya and Takafumi Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2:59–77, 1984.

[HS88] Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. Technical Report 53, LILOG, IBM Deutschland, Stuttgart, Germany, October 1988. To appear in Journal of Logic Programming.

[HSH90] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming I: Clauses as rules. *Journal of Logic and Computation*, 1(2), 1990.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Conference on Principles of Programming Languages*, Munich, 1987.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*, volume 1 of *Bibliotheca Mathematica*. North-Holland, Amsterdam, 1952.

[Mac85] Alan Mackworth. Constraint satisfaction. Technical Report 85-15, Department of Computer Science, University of British Columbia, September 1985.

[Mah88] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the Third Annual Symposium on Logic In Computer Science*, pages 348–357, Edinburgh, July 1988. Computer Society Press.

[Mah93] Michael J. Maher. A logic programming view of CLP. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 737–753, Budapest, July 1993. MIT Press.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Mon76] James Donald Monk. *Mathematical Logic*, volume 37 of *Graduate texts in mathematics*. Springer-Verlag, New York, 1976.

[Sco82] Dana Scott. Domains for denotational semantics. In *International Colloquium on Automata, Languages, and Programming*, 1982.

[Sho67] Joseph Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Mass., 1967.

[vEL84] Maarten H. van Emden and John W. Lloyd. A logical reconstruction of Prolog II. *Journal of Logic Programming*, 2:143–149, 1984.