The CIFF Proof Procedure for Abductive Logic Programming with Constraints

U. Endriss[1], P. Mancarella[2], F. Sadri[1], G. Terreni[2], and F. Toni[1,2]

[1] Department of Computing, Imperial College London
Email: {ue,fs,ft}@doc.ic.ac.uk
[2] Dipartimento di Informatica, Università di Pisa
Email: {paolo,terreni,toni}@di.unipi.it

Abstract. We introduce a new proof procedure for abductive logic programming and present two soundness results. Our procedure extends that of Fung and Kowalski by integrating abductive reasoning with constraint solving and by relaxing the restrictions on allowed inputs for which the procedure can operate correctly. An implementation of our proof procedure is available and has been applied successfully in the context of multiagent systems.

1 Introduction

Abduction has found broad application as a tool for hypothetical reasoning with incomplete knowledge, which can be handled by labelling some pieces of information as *abducibles*, i.e. as possible hypotheses that can be assumed to hold, provided that they are consistent with the given knowledge base. Abductive Logic Programming (ALP) combines abduction with logic programming enriched by *integrity constraints* to further restrict the range of possible hypotheses. Important applications of ALP include planning [10], requirements specification analysis [8], and agent communication [9]. In recent years, a variety of proof procedures for ALP have been proposed, including the IFF procedure of Fung and Kowalski [4]. Here, we extend this procedure in two ways, namely (1) by integrating abductive reasoning with constraint solving (in the sense of CLP, not to be confused with integrity constraints), and (2) by relaxing the *allowedness* conditions given in [4] to be able to handle a wider class of problems.

Our interest in extending IFF in this manner stems from applications developed in the SOCS project, which investigates the use of computational logic-based techniques in the context of multiagent systems for global computing. In particular, we use ALP extended with constraint solving to give computational models for an agent's *planning*, *reactivity* and *temporal reasoning* capabilities [5]. We found that our requirements for these applications go beyond available state-of-the-art ALP proof procedures. While ACLP [6], for instance, permits the use of constraint predicates (unlike IFF), its syntax for integrity constraints is too restrictive to express the planning knowledge bases (using a variant of the abductive event calculus [10]) used in SOCS. In addition, many procedures put strong,

sometimes unnecessary, restrictions on the use of variables. The procedure proposed in this paper, which we call CIFF, manages to overcome these restrictions to a degree that has allowed us to apply it successfully to a wide range of problems. We have implemented CIFF in Prolog;[3] the system forms an integral part of the PROSOCS platform for programming agents in computational logic [11].

In the next section we are going to set out the ALP framework used in this paper and discuss the notion of allowedness. Section 3 then specifies the CIFF proof procedure which we propose as a suitable reasoning engine for this framework. Two soundness results for CIFF are presented in Section 4 and Section 5 concludes. An extended version of this paper that, in particular, contains detailed proofs of our results is available as a technical report [3].

2 Abductive Logic Programming with Constraints

We use classical first-order logic, enriched with a number of special predicate symbols with a fixed semantics, namely the equality symbol $=$, which is used to represent the unifiability of terms (i.e. as in standard logic programming), and a number of constraint predicates. We assume the availability of a sound and complete constraint solver for this constraint language. In principle, the exact specification of the constraint language is independent from the definition of the CIFF procedure, because we are going to use the constraint solver as a *black box* component.[4] However, the constraint language has to include a relation symbol for equality (we are going to write $t_1 =_c t_2$) and it must be closed under complements. In general, the complement of a constraint Con will be denoted as $\overline{Con}$ (but we are going to write $t_1 \neq_c t_2$ for the complement of $t_1 =_c t_2$). The range of admissible arguments to constraint predicates again depends on the specifics of the chosen constraint solver. A typical choice for a constraint system would be an arithmetic constraint solver over integers providing predicates such as $\leq$ and $>$ and allowing for terms constructed from variables, integers and function symbols representing operations such as addition and multiplication.

Abductive logic programs. An *abductive logic program* is a pair $\langle Th, IC \rangle$ consisting of a *theory* Th and a finite set of *integrity constraints* IC. We present theories as sets of so-called iff-definitions:

$$p(X_1, \ldots, X_k) \leftrightarrow D_1 \vee \cdots \vee D_n$$

The predicate symbol p must not be a *special* predicate (constraints, $=$, $\top$ and $\bot$) and there can be at most one iff-definition for every predicate symbol. Each of the disjuncts D_i is a conjunction of literals. Negative literals are written as implications (e.g. $q(X, Y) \rightarrow \bot$). The variables $X_1, \ldots, X_k$ are implicitly universally quantified with the scope being the entire definition. Any other variable is implicitly existentially quantified, with the scope being the disjunct in which it

[3] The CIFF system is available at `http://www.doc.ic.ac.uk/~ue/ciff/`.

[4] Our implementation uses the built-in finite domain solver of Sicstus Prolog [1], but the modularity of the system would also support the integration of a different solver.

occurs. A theory may be regarded as the (selective) completion of a normal logic program (i.e. of a logic program allowing for negative subgoals in a rule) [2]. Any predicate that is neither defined nor special is called an *abducible*.

In this paper, the integrity constraints in the set IC (not to be confused with constraint predicates) are implications of the following form:

$$L_1 \wedge \cdots \wedge L_m \ \rightarrow \ A_1 \vee \cdots \vee A_n$$

Each of the L_i must be a literal (with negative literals again being written in implication form); each of the A_i must be an atom. Any variables are implicitly universally quantified with the scope being the entire implication.

A *query* Q is a conjunction of literals. Any variables in Q are implicitly existentially quantified. They are also called the *free* variables. In the context of the CIFF procedure, we are going to refer to a triple $\langle Th, IC, Q\rangle$ as an *input*.

Semantics. A theory provides definitions for certain predicates, while integrity constraints restrict the range of possible interpretations. A query may be regarded as an *observation* against the background of the world knowledge encoded in a given abductive logic program. An *answer* to such a query would then provide an *explanation* for this observation: it would specify which instances of the abducible predicates have to be assumed to hold for the observation to hold as well. In addition, such an explanation should also validate the integrity constraints. This is formalised in the following definition:

Definition 1 (Correct answer). *A correct answer to a query Q with respect to an abductive logic program $\langle Th, IC\rangle$ is a pair $\langle \Delta, \sigma\rangle$, where Δ is a finite set of ground abducible atoms and σ is a substitution for the free variables occurring in Q, such that $Th \cup Comp(\Delta) \models IC \wedge Q\sigma$.*

Here $\models$ is the usual consequence relation of first-oder logic with the restriction that constraint predicates have to be interpreted according to the semantics of the chosen constraint system and equalities evaluate to *true* whenever their two arguments are unifiable. $Comp(\Delta)$ stands for the *completion* of the set of abducibles in Δ, i.e. any ground atom not occurring in Δ is assumed to be *false*. If we have $Th \cup IC \models \neg Q$ (i.e. if Q is false for all instantiations of the free variables), then we say that there exists no correct answer to the query Q given the abductive logic program $\langle Th, IC\rangle$.

Example 1. Consider the following abductive logic program:

$$\begin{array}{ll} Th: & p(T) \ \leftrightarrow \ q(X,T') \ \wedge \ T'<T \ \wedge \ T<8 \\ & q(X,T) \ \leftrightarrow \ X=a \ \wedge \ s(T) \\ IC: & r(T) \ \rightarrow \ p(T) \end{array}$$

The set of abducible predicates is $\{r, s\}$. The query $r(6)$, for instance, should succeed; a possible *correct answer* would be the set $\{r(6), s(5)\}$, with an empty substitution. Intuitively, given the query $r(6)$, the integrity constraint in IC would fire and force the atom $p(6)$ to hold, which in turn requires $s(T')$ for some $T' < 6$ to be true (as can be seen by *unfolding* first $p(6)$ and then $q(X, T')$). $\square$

Allowedness. Fung and Kowalski [4] require inputs $\langle Th, IC, Q \rangle$ to meet a number of so-called *allowedness conditions* to be able to guarantee the correct operation of their proof procedure. These conditions are designed to avoid constellations with particular (problematic) patterns of quantification. Unfortunately, it is difficult to formulate appropriate allowedness conditions that guarantee a correct execution of the proof procedure without imposing too many unnecessary restrictions. This is a well-known problem, which is further aggravated for languages that include constraint predicates. Our proposal is to tackle the issue of allowedness *dynamically*, i.e. at runtime, rather than adopting a static and overly strict set of conditions. In this paper, we are only going to impose the following *minimal allowedness conditions:*[5]

- An integrity constraint $A \rightarrow B$ is allowed iff every variable in it also occurs in a positive literal within its antecedent A.
- An iff-definition $p(X_1, \ldots, X_k) \leftrightarrow D_1 \vee \cdots \vee D_n$ is allowed iff every variable other than $X_1, \ldots, X_k$ occurring in a disjunct D_i also occurs inside a positive literal within the same D_i.

The crucial allowedness condition is that for integrity constraints: it ensures that, also after an application of the *negation rewriting* rule (which moves negative literals in the antecedent of an implication to its consequent), every variable occurring in the consequent of an implication is also present in its antecedent. The allowedness condition for iff-definitions merely allows us to maintain this property of implications when the *unfolding* rule (which, essentially, replaces a defined predicate with its definition) is applied to atoms in the antecedent of an implication. We do not need to impose any allowedness conditions on queries.

3 The CIFF Proof Procedure

We are now going to formally introduce the CIFF proof procedure. The *input* $\langle Th, IC, Q \rangle$ to the procedure consists of a theory Th, a set of integrity constraints IC, and a query Q. There are three possible *outputs:* (1) the procedure succeeds and indicates an answer to the query Q; (2) the procedure fails, thereby indicating that there is no answer; and (3) the procedure reports that computing an answer is not possible, because a critical part of the input is not allowed.

The CIFF procedure manipulates, essentially, a set of formulas that are either atoms or implications. The theory Th is kept in the background and is only used to *unfold* defined predicates as they are being encountered. In addition to atoms and implications the aforementioned set of formulas may contain disjunctions of atoms and implications to which the *splitting* rule may be applied, i.e. which give rise to different branches in the proof search tree. The sets of formulas manipulated by the procedure are called *nodes*. A node is a set (representing a

[5] Note that the CIFF procedure could easily be adapted to work also on inputs not conforming even to these minimal conditions, but then it would not be possible anymore to represent quantification implicitly.

conjunction)[6] of formulas (atoms, implications, or disjunctions thereof) which are called *goals*. A proof is initialised with the node containing the integrity constraints IC and the literals of the query Q. The proof procedure then repeatedly manipulates the current node of goals by rewriting goals in the node, adding new goals to it, or deleting superfluous goals from it. Most of this section is concerned with specifying these *proof rules* in detail.

The structure of our proof rules guarantee that the following *quantification invariants* hold for every node in a derivation:

- No implication contains a universally quantified variable that is not also contained in one of the positive literals in its antecedent.
- No atom contains a universally quantified variable.
- No atom inside a disjunction contains a universally quantified variable.

In particular, these invariants subsume the *minimal allowedness conditions* discussed in the previous section. The invariants also allow us to keep quantification implicit throughout a CIFF derivation by determining the quantification status of any given variable. Most importantly, any variable occurring in either the original query or an atomic conjunct in a node must be existentially quantified.

Notation. In the sequel, we are frequently going to write $\vec{t}$ for a "vector" of terms such as $t_1, \ldots, t_k$. For instance, we are going to write $p(\vec{t})$ rather than $p(t_1, \ldots, t_k)$. To simplify presentation, we assume that there are no two predicates that have the same name but different arities. We are also going to write $\vec{t} = \vec{s}$ as a shorthand for $t_1 = s_1 \wedge \cdots \wedge t_k = s_k$ (with the implicit assumption that the two vectors have the same length), and $[\vec{X}/\vec{t}]$ for the substitution $[X_1/t_1, \ldots, X_k/t_k]$. Note that X and Y always represent variables. Furthermore, in our presentation of proof rules, we are going to abstract from the order of conjuncts in the antecedent of an implication: the critical subformula is always represented as the first conjunct. That is, by using a pattern such as $X = t \wedge A \rightarrow B$ we are referring to any implication with an antecedent that has a conjunct of the form $X = t$. A represents the remaining conjunction, which may also be "empty", that is, the formula $X = t \rightarrow B$ is a special case of the general pattern $X = t \wedge A \rightarrow B$. In this case, the residue $A \rightarrow B$ represents the formula B.

Proof rules. For each of the proof rules in our system, we specify the type of formula(s) which may trigger the rule (*"given"*), a number of side *conditions* that need to be met, and the required *action* (such as replacing the given formula by a different one). Executing this action yields one or more successor nodes and the current node can be discarded. The first rule replaces a defined predicate occurring as an atom in the node by its defining disjunction:

Unfolding atoms
Given: $p(\vec{t})$
Cond.: $[p(\vec{X}) \leftrightarrow D_1 \vee \cdots \vee D_n] \in Th$
Action: replace by $(D_1 \vee \cdots \vee D_n)[\vec{X}/\vec{t}]$

[6] If a proof rule introduces a conjunction into a node, this conjunction is understood to be broken up into its subformulas right away.

Note that any variables in $D_1 \vee \cdots \vee D_n$ other than those in $\vec{X}$ are existentially quantified with respect to the definition, i.e. they must be new to the node and they will be existentially quantified in the successor node.

Unfolding predicates in the antecedent of an implication yields one new implication for every disjunct in the defining disjunction:

Unfolding within implications
Given: $p(\vec{t}) \wedge A \rightarrow B$
Cond.: $[p(\vec{X}) \leftrightarrow D_1 \vee \cdots \vee D_n] \in Th$
Action: replace by $D_1[\vec{X}/\vec{t}] \wedge A \rightarrow B$, ..., $D_n[\vec{X}/\vec{t}] \wedge A \rightarrow B$

Observe that variables in any of the D_i that have been existentially quantified in the definition of $p(\vec{t})$ are going to be universally quantified in the corresponding new implication (because they appear within the antecedent).

The next rule is the *propagation* rule, which allows us to resolve an atom in the antecedent of an implication with a matching atom in the node. Unlike most rules, this rule does not *replace* a given formula, but it merely *adds* a new one. This is why we require explicitly that propagation cannot be applied again to the same pair of formulas. Otherwise the procedure would be bound to loop.

Propagation
Given: $p(\vec{t}) \wedge A \rightarrow B$ and $p(\vec{s})$
Cond.: the rule has not yet been applied to this pair of formulas
Action: add $\vec{t} = \vec{s} \wedge A \rightarrow B$

The *splitting* rule gives rise to (not just a single but) a whole set of successor nodes, one for each of the disjuncts in $A_1 \vee \cdots \vee A_n$, each of which gives rise to a different branch in the derivation:

Splitting
Given: $A_1 \vee \cdots \vee A_n$
Cond.: none
Action: replace by one of $A_1, \ldots, A_n$

The next rule is a logical simplification that moves negative literals in the antecedent to the consequent of an implication:

Negation rewriting
Given: $(A \rightarrow \bot) \wedge B \rightarrow C$
Cond.: none
Action: replace by $B \rightarrow A \vee C$

There are two further logical simplification rules:

Logical simplification (trivial condition)
Given: $\top \wedge A \rightarrow B$
Cond.: none
Action: replace by $A \rightarrow B$

Logical simplification (redundant formulas)
Given: either $\bot \rightarrow A$ or $\top$
Cond.: none
Action: delete formula

The following *factoring* rule can be used to separate cases in which particular abducible atoms unify from those in which they do not:

Factoring

Given: $p(\vec{t})$ and $p(\vec{s})$
Cond.: p abducible; the rule has not yet been applied to $p(\vec{t})$ and $p(\vec{s})$
Action: replace by $[p(\vec{t}) \wedge p(\vec{s}) \wedge (\vec{t} = \vec{s} \rightarrow \bot)] \vee [p(\vec{t}) \wedge \vec{t} = \vec{s}]$

The next few rules deal with equalities. The first two of these involve simplifying equalities according to the following rewrite rules:

(1) Replace $f(t_1, \ldots, t_k) = f(s_1, \ldots, s_k)$ by $t_1 = s_1 \wedge \cdots \wedge t_k = s_k$.
(2) Replace $f(t_1, \ldots, t_k) = g(s_1, \ldots, s_l)$ by $\bot$ if f and g are distinct or $k \neq l$.
(3) Replace $t = t$ by $\top$.
(4) Replace $X = t$ by $\bot$ if t contains X.
(5) Replace $t = X$ by $X = t$ if X is a variable and t is not.
(6) Replace $Y = X$ by $X = Y$ if X is a univ. quant. variable and Y is not.
(7) Replace $Y = X$ by $X = Y$ if X and Y are exist. quant. variables and X occurs in a constraint predicate, but Y does not.

Rules (1)–(4) essentially implement the term reduction part of the unification algorithm of Martelli and Montanari [7]. Rules (5)–(7) ensure that completely rewritten equalities are always presented in a normal form, thereby simplifying the formulation of our proof rules.

Equality rewriting for atoms

Given: $t_1 = t_2$
Cond.: the rule has not yet been applied to this equality
Action: replace by the result of rewriting $t_1 = t_2$

Equality rewriting for implications

Given: $t_1 = t_2 \wedge A \rightarrow B$
Cond.: the rule has not yet been applied to this equality
Action: replace by $C \wedge A \rightarrow B$ where C is the result of rewriting $t_1 = t_2$

The following two *substitution rules* also handle equalities:

Substitution rule for atoms

Given: $X = t$
Cond.: $X \notin t$; the rule has not yet been applied to this equality
Action: apply substitution $[X/t]$ to entire node except $X = t$ itself

Substitution rule for implications

Given: $X = t \wedge A \rightarrow B$
Cond.: X univ. quant.; $X \notin t$; t contains no univ. quant. variables *or* $X \notin B$
Action: replace by $(A \rightarrow B)[X/t]$

The purpose of the third side condition (of t not containing any universally quantified variables or X not occurring within B) is to maintain the quantification invariant that any universally quantified variable in the consequent of an implication is also present in the antecedent of the same implication.

If neither *equality rewriting* nor a *substitution rule* are applicable, then an equality may give rise to a *case analysis:*

Case analysis for equalities

Given: $X = t \wedge A \rightarrow B$ (exception: do not apply to $X = t \rightarrow \bot$)
Cond.: X exist. quant.; $X \notin t$; t is not a univ. quant. variable
Action: replace by $X = t$ and $A \rightarrow B$, or replace by $X = t \rightarrow \bot$

Case analysis should not be applied to formulas of the form $X = t \rightarrow \bot$ (despite this being an instance of the pattern $X = t \wedge A \rightarrow B$), because this would lead to a loop (with respect to the second successor node). Also note that, if the third of the above side conditions was not fulfilled and if t was a universally quantified variable, then *equality rewriting* could be applied to obtain $t = X \wedge A \rightarrow B$, to which we could then apply the *substitution rule for implications*.

Observe that the above rule gives rise to two successor nodes (rather than a disjunction). This is necessary, because the term t may contain variables that would be quantified differently on the two branches, i.e. a new formula with a disjunction in the matrix would not (necessarily) be logically equivalent to the disjunction of the two (quantified) subformulas. In particular, in the first successor node all variables in t will become existentially quantified. To see this, consider the example of the implication $X = f(Y) \wedge A \rightarrow B$ and assume X is existentially quantified, while Y is universally quantified. We can distinguish two cases: (1) either X represents a term whose main functor is f, or (2) this is not the case. In case (1), there *exists* a value for Y such that $X = f(Y)$, and furthermore $A \rightarrow B$ must hold. Otherwise, i.e. in case (2), $X = f(Y)$ will be false for *all* values of Y.

Case analysis for constraints

Given: $Con \wedge A \rightarrow B$
Cond.: Con is a constraint predicate without univ. quant. variables
Action: replace by $[Con \wedge (A \rightarrow B)] \vee \overline{Con}$

Observe that the conditions on quantification are a little stricter for *case analysis for constraints* than they were for *case analysis for equalities*. Now all variables involved need to be existentially quantified. This simplifies the presentation of the rule a little, because no variables change quantification. In particular, we can replace the implication in question by a disjunction (to which the splitting rule may be applied in a subsequent step).

While case analysis is used to separate constraints from other predicates, the next rule provides the actual *constraint solving* step itself. It may be applied to any set of constraints in a node, but to guarantee soundness, eventually, it has to be applied to the set of *all* constraint atoms.

Constraint solving

Given: constraint predicates $Con_1, \ldots, Con_n$
Cond.: $\{Con_1, \ldots, Con_n\}$ is not satisfiable
Action: replace by $\bot$

If $\{Con_1, \ldots, Con_n\}$ is found to be satisfiable it may also be replaced with an equivalent but simplified set (in case the constraint solver used offers this feature). To simplify presentation, we assume that the constraint solver will fail (rather than come back with an undefined answer) whenever it is presented with

an ill-defined constraint such as, say, $bob \leq 5$ (in the case of an arithmetic solver). For inputs that are "well-typed", however, such a situation will never arise.

Our next two rules ensure that (dis)equalities that affect the satisfiability of the constraints in a node are correctly rewritten using the appropriate constraint predicates. Here we refer to a variable as a *constraint variable* (with respect to a particular node) iff that variable occurs inside a constraint atom in that node. For the purpose of stating the next two rules in a concise manner, we call a term *c-atomic* iff it is either a variable or a ground element of the constraint domain (e.g. an integer in the case of an arithmetic domain).

Equality-constraint rewriting
Given: $X = t$
Cond.: X is a constraint variable
Action: replace by $X =_c t$ if t is c-atomic; replace by $\bot$ otherwise

Disequality-constraint rewriting
Given: $X = t \rightarrow \bot$
Cond.: X is a constraint variable
Action: replace by $X \neq_c t$ if t is c-atomic; delete formula otherwise

For example, if we are working with an arithmetic constraint domain, then the formula $X = bob \rightarrow \bot$ would be deleted from the node as it holds vacuously whenever X also occurs within a constraint predicate.

We call a formula of the form $t_1 = t_2 \rightarrow \bot$ a *disequality* provided no universally quantified variables occur in either t_1 or t_2. The next rule is used to identify nodes containing formulas with problematic quantification, which could cause difficulties in extracting an abductive answer:

Dynamic allowedness rule (DAR)
Given: $A \rightarrow B$ (exception: do not apply to disequalities)
Cond.: A consists of equalities and constraints alone; no other rule applies
Action: label node as *undefined*

In view of the second side condition, recall that the only rules applicable to an implication with only equalities and constraints in the antecedent are the equality rewriting and substitution rules for implications and the two case analysis rules.

Answer extraction. A node containing $\bot$ is called a *failure node*. If all branches in a derivation terminate with failure nodes, then the derivation is said to fail (the intuition being that there exists no answer to the query). A node to which no more rules can be applied is called a *final node*. A final node that is not a failure node and that has not been labelled as *undefined* is called a *success node*.

Definition 2 (Extracted answer). *An extracted answer for a final success node N is a triple $\langle \Delta, \Phi, \Gamma \rangle$, where Δ is the set of abducible atoms, Φ is the set of equalities and disequalities, and Γ is the set of constraint atoms in N.*

An *extracted* answer in itself is not yet a *correct* answer in the sense of Definition 1, but —as we shall see— it does *induce* such a correct answer. The basic idea

is to first define a substitution σ that is consistent with both the (dis)equalities in Φ and the constraints in Γ, and then to ground the set of abducibles Δ by applying σ to it. The resulting set of ground abducible atoms together with the substitution σ then constitutes a correct answer to the query (i.e., an extracted answer will typically give rise to a whole range of correct answers). To argue that this is indeed possible, i.e. to show that the described procedure of deriving answers to a query is a *sound* operation, will be the subject of the next section.

Example 2. We show the derivation for the query $r(6)$ given the abductive logic program of Example 1. Recall that CIFF is initiated with the node N_0 composed of the query and the integrity constraints in IC.

N_0: $r(6) \wedge [r(T) \rightarrow p(T)]$ [initial node]
N_1: $r(6) \wedge [T = 6 \rightarrow p(T)] \wedge [r(T) \rightarrow p(T)]$ [by propagation]
N_2: $r(6) \wedge p(6) \wedge [r(T) \rightarrow p(T)]$ [by substitution]
N_3: $r(6) \wedge q(X, T') \wedge T' < 6 \wedge 6 < 8 \wedge [r(T) \rightarrow p(T)]$ [by unfolding]
N_4: $r(6) \wedge q(X, T') \wedge T' < 6 \wedge [r(T) \rightarrow p(T)]$ [by constraint solving]
N_5: $r(6) \wedge X = a \wedge s(T') \wedge T' < 6 \wedge [r(T) \rightarrow p(T)]$ [by unfolding]

No more rules can be applied to the node N_5 and it neither contains $\bot$ nor has it been labelled as *undefined*. Hence, it is a success node and we get an *extracted answer* with $\Delta = \{r(6), s(T')\}$, $\Phi = \{X = a\}$ and $\Gamma = \{T' < 6\}$, of which the *correct answer* given in Example 1 is an instance. □

4 Soundness Results

In this section we are going to present the soundness of the CIFF procedure with respect to the semantics of a correct answer to a given query. Due to space restrictions, we have to restrict ourselves to short sketches of the main ideas involved. Full proofs may be found in [3]. Our results extend those of Fung and Kowalski for the original IFF procedure in two respects: (1) they apply to abductive logic programs with constraints, and (2) they do not rely on a static (and overly strict) definition of allowedness.

For an abductive proof procedure, we can distinguish two types of soundness results: *soundness of success* and *soundness of failure*. The first one establishes the correctness of derivations that are successful (soundness of success): whenever the CIFF procedure terminates successfully then the *extracted* answer (consisting of a set of abducible atoms, a set of equalities and disequalities, and a set of constraints) gives rise to a *true* answer according to the semantics of ALP (i.e. a ground set of abducible atoms and a substitution). Note that for this result to apply, it suffices that a single final success node can be derived. This node will give rise to a correct answer, even if there are other branches in the derivation that do not terminate or for which the DAR has been triggered. The second soundness result applies to derivations that fail (soundness of failure): it states that whenever the CIFF procedure fails then there is indeed no answer according to the semantics. This result applies only when *all* branches in a derivation have

failed; if there are branches that do not terminate or for which the DAR has been triggered, then we cannot draw any conclusions regarding the existence of an answer to the query (assuming there are no success nodes).

The proofs of both these results heavily rely on the fact that our proof rules are *equivalence preserving:*

Lemma 1 (Equivalence preservation). *If N is a node in a derivation with respect to the theory Th, and $\mathcal{N}$ is the disjunction of the immediate successor nodes of N in that derivation, then $Th \models N \leftrightarrow \mathcal{N}$.*

Note that the disjunction $\mathcal{N}$ will have only a single disjunct whenever the rule applied to N is neither splitting nor case analysis for equalities. Equivalence preservation is easily verified for most of our proof rules. Considering that $IC \wedge Q$ is the initial node of any derivation, the next lemma then follows by induction over the number of proof steps leading to a final success node:

Lemma 2 (Final nodes entail initial node). *If N is a final success node for the input $\langle Th, IC, Q \rangle$, then $Th \models N \rightarrow (IC \wedge Q)$.*

Our third lemma provides the central argument in showing that it is possible to extract a correct abductive answer from a final success node:

Lemma 3 (Answer extraction). *If N is a final success node and Δ is the set of abducible atoms in N, then there exists a substitution σ such that $Comp(\Delta\sigma) \models N\sigma$.*

The first step in proving this lemma is to show that any formulas in N that are not directly represented in the extracted answer must be implications where the antecedent includes an abducible atom and no negative literals. We can then show that implications of this type are logical consequences of $Comp(\Delta\sigma)$ by distinguishing two cases: either *propagation* has been applied to the implication in question, or it has not. In the latter case, the claim holds vacuously (because the antecedent is not *true*); in the former case we use an inductive argument over the number of abducible atoms in the antecedent.

The full proof of Lemma 3 makes reference to all proof rules except *factoring.* Indeed, factoring is not required to ensure soundness. However, as can easily be verified, factoring is equivalence preserving in the sense of Lemma 1; that is, our soundness results apply both to the system with and to the system without the factoring rule. We are now ready to state these soundness results:

Theorem 1 (Soundness of success). *If there exists a successful derivation for the input $\langle Th, IC, Q \rangle$, then there exists a correct answer for that input.*

Theorem 2 (Soundness of failure). *If there exists a derivation for the input $\langle Th, IC, Q \rangle$ that terminates and where all final nodes are failure nodes, then there exists no correct answer for that input.*

Theorem 1 follows from Lemmas 2 and 3, while Theorem 2 can be proved by induction over the number of proof steps in a derivation, using Lemma 1 in the

induction step. We should stress that these soundness results only apply in cases where the DAR has not been triggered and the CIFF procedure has terminated with a defined outcome, namely either *success* or *failure*. Hence, such results are only interesting if we can give some assurance that the DAR is "appropriate": In a similar but ill-defined system where an (inappropriate) allowedness rule would simply label all nodes as *undefined*, it would still be possible to prove the same soundness theorems, but they would obviously be of no practical relevance.

The reason why our rule is indeed appropriate is that extracting an answer from a node labelled as *undefined* by the DAR would either require us to extend the definition of a correct answer to allow for *infinite* sets of abducible atoms or at least involve the enumeration of *all* the solutions to a set of constraints. We shall demonstrate this by means of two simple examples. First, consider the following implication:

$$X = f(Y) \ \rightarrow \ p(X)$$

If both X and Y are universally quantified, then this formula will trigger the DAR. Its meaning is that the predicate p is true whenever its argument is of the form $f(\_)$. Hence, an "answer" induced by a node containing this implication would have to include the infinite set $\{p(f(t_1)), p(f(t_2)), \ldots\}$, where $t_1, t_2, \ldots$ stand for the terms in the Herbrand universe. This can also be seen by considering that, if we were to ignore the side conditions on quantification of the *substitution rule for implications*, the above implication could be rewritten as $p(f(Y))$, with Y still being universally quantified.

For the next example, assume that our constraint language includes the predicate $<$ with the usual interpretation over integers:

$$3 < X \ \wedge \ X < 100 \ \rightarrow \ p(X)$$

Again, if the variable X is universally quantified, this formula will trigger the DAR. While it would be possible to extract a finite answer from a node including this formula, this would require us to enumerate all solutions to the constraint $3 < X \wedge X < 100$; that is, a correct answer would have to include the set of atoms $\{p(4), p(5), \ldots, p(99)\}$. In cases where the set of constraints concerned has an infinite number of solutions, even in theory, it is not possible to extract a correct answer (as it would be required to be both ground and finite).

5 Conclusion

We have introduced a new proof procedure for ALP that extends the IFF procedure in a non-trivial way by integrating abductive reasoning with constraint solving. Our procedure shares the advantages of the IFF procedure [4], but covers a larger class of inputs: (1) predicates belonging to a suitable constraint language may be used, and (2) the allowedness conditions have been reduced to a minimum. Both these extension are important requirements for our applications of ALP to modelling and implementing autonomous agents [5, 11]. In cases where no answer is possible due to allowedness problems, the CIFF procedure will report this dynamically. However, if an answer is possible despite such problems, CIFF will report a defined answer. For instance, one node may give rise

to a positive answer while another has a non-allowed structure, or a derivation may fail correctly for reasons that are independent of a particular non-allowed integrity constraint. For inputs conforming to any appropriate *static* allowedness definition, the DAR will never be triggered.

We have proved two soundness results for CIFF: *soundness of success* and *soundness of failure*. Together these two results also capture some aspect of completeness: For any class of inputs that are known to be allowed (in the sense of never triggering the DAR) and for which termination can be guaranteed (for instance, by imposing suitable acyclicity conditions [12]) the CIFF procedure will terminate successfully whenever there exists a correct answer according to the semantics. We hope to investigate the issues of termination and completeness further in our future work. Another interesting issue for future work on CIFF would be to investigate different strategies for proof search and other optimisation techniques. Such research could then inform an improvement of our current implementation and help to make it applicable to more complex problems.

Acknowledgements. This work was partially funded by the IST-FET programme of the European Commission under the IST-2001-32530 SOCS project, within the Global Computing proactive initiative. The last author was also supported by the Italian MIUR programme "Rientro dei cervelli".

References

[1] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. PLILP-1997*, 1997.
[2] K. L. Clark. Negation as failure. In *Logic and Data Bases*. Plenum Press, 1978.
[3] U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The CIFF proof procedure: Definition and soundness results. Technical Report 2004/2, Department of Computing, Imperial College London, May 2004.
[4] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.
[5] A. C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In *Proc. ECAI-2004*, 2004. To appear.
[6] A. C. Kakas, A. Michael, and C. Mourlas. ACLP: Abductive constraint logic programming. *Journal of Logic Programming*, 44:129–177, 2000.
[7] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
[8] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An abductive approach for analysing event-based requirements specifications. In *Proc. ICLP-2002*. Springer-Verlag, 2002.
[9] F. Sadri, F. Toni, and P. Torroni. An abductive logic programming architecture for negotiating agents. In *Proc. JELIA-2002*. Springer-Verlag, 2002.
[10] M. Shanahan. An abductive event calculus planner. *Journal of Logic Programming*, 44:207–239, 2000.
[11] K. Stathis, A. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali. PROSOCS: A platform for programming software agents in computational logic. In *Proc. AT2AI-2004*, 2004.
[12] I. Xanthakos. *Semantic Integration of Information by Abduction*. PhD thesis, Department of Computing, Imperial College London, 2003.