Programming Applications in CIFF

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

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

Abstract. We show how to deploy the CIFF System 4.0 for abductive logic programming with constraints in a number of applications, ranging from combinatorial applications to web management. We also compare the CIFF System 4.0 with a number of logic programming tools, namely the A-System, the DLV system and the SMODELS system.

1 Introduction

Abduction has broad applications as a tool for hypothetical reasoning with incomplete knowledge. It allows the labeling of some pieces of information as *abducibles*, i.e. as 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. ALP has also been integrated with *Constraint Logic Programming* (CLP), for having an arithmetic tool for *constraint solving*. Important applications of ALP with constraints (ALPC) include agent programming [1, 2] and web management tools [3].
Many proof procedures for ALPC have been proposed, including ACLP [4], the A-System [5] and CIFF [6], which extends the IFF procedure for ALP [7] by integrating a CLP solver and by relaxing some *allowedness* conditions given in [7]. In this paper we describe the CIFF System 4.0[3] implementation and we compare it empirically with other systems showing that (1) they have comparable performances and (2) the CIFF System 4.0 has some unique features, in particular its handling of variables taking values on unbound domains.

2 Background

Here we summarize some background concepts. For more details see [6–8].
An *abductive logic program with constraints* consists of three components: (i) A constraint logic program, referred to as the *theory*, namely a set of clauses of the form $A \leftarrow L_1 \wedge \ldots \wedge L_m$, where the L_is are literals (ordinary atoms, their negation, or constraint atoms) and A is an ordinary atom, whose variables are all implicitly universally quantified from the outside. (ii) A finite set of *abducible*

[3] The CIFF System 4.0 is available at www.di.unipi.it/∼terreni/research.php.

predicates, that do not occur in any conclusion A of any clauses in the theory. (iii) A finite set of *integrity constraints* (ICs), namely implications of the form $L_1 \wedge \cdots \wedge L_m \rightarrow A_1 \vee \cdots \vee A_n$ where the L_is are literals and the A_js are (ordinary or constraint) atoms or the special atom $false$. All the variables in an IC are implicitly universally quantified from the outside, except for variables occurring only in the A_js which are existentially quantified with scope $A_1 \vee \cdots \vee A_n$. The theory provides definitions for non-abducible, non-constraint, ordinary predicates; it can be extended by means of sets of atoms in the abducible predicates, subject to satisfying the integrity constraints. Constraint atoms are evaluated within an underlying structure, as in conventional CLP.

A *query* is a conjunction of literals (whose variables are free). An *answer* to a query specifies which instances of the abducible predicates have to be assumed to hold so that both (some instance of) the query is entailed by the constraint logic program extended with the abducibles and the ICs are satisfied, wrt a chosen semantics for (constraint) logic programming and a notion of satisfaction of ICs.

The CIFF procedure computes such answers, with respect to the notion of entailment given by the 3-valued completion. It operates with a presentation of the theory as a set of *iff-definitions*, which are obtained by the (selective) *completion* of all predicates defined in the theory except for the abducible and the constraint predicates. CIFF returns three possible outputs: (1) an abductive answer to the query, (2) a failure, indicating that there is no answer, and (3) an *undefined* answer, indicating that a critical part of the input is not *allowed* (i.e. does not satisfy certain restrictions on variable occurrences). (1) is in the form of a set of (non-ground) abducible atoms and a set of constraints on the variables of the query and of the abducible atoms.

The CIFF procedure operates on so-called *nodes* which are conjunctions of formulas called *goals*. Intuitively, sequences of nodes represent branches in the proof search tree. A proof (and the search tree) is initialised with a node containing the ICs and the original query. The iff-definitions are used to *unfold* defined predicates as they are encountered in goals. The proof procedure repeatedly replaces one node with another by applying the procedure's *rewrite rules* to the goals. If a disjunction of goals is encountered, then the *splitting* rule can be applied, giving rise to alternative branches in the search tree. Besides *unfolding* and *splitting*, CIFF uses other rewrite rules (see [6]) such as *propagation* with the ICs.

A node containing a goal $false$ is called a *failure node*. If all branches in a derivation terminate with failure nodes, then the procedure is said to fail (no answer to the query). A non-failure (allowed) node to which no more rewrite rules apply can be used to extract an (abductive) answer.

ICs can be used to specify *reactive rules* in many applications, e.g modelling agents. In some cases the classical treatment of negation in ICs in CIFF can lead to non-intuitive answers being computed. For example, ICs $A \wedge \neg B \rightarrow C$ and $A \rightarrow C \vee B$ are treated equivalently in CIFF. Hence, one way to satisfy these ICs is to ensure that C holds whenever A holds. However, the only "reactive meaning" of the original IC is to ensure that C holds when *both* A and $\neg B$ have been proven. We have therefore investigated a different way of treating

negation within ICs, namely *negation as failure* (NAF) [8], and have integrated this treatment into CIFF [9], by allowing ICs to be either marked or unmarked depending upon the required treatment of negation in them.

3 The CIFF System 4.0

CIFF 4.0 is a Sicstus Prolog implementation of CIFF. It maintains the computational basis of version 3.0 [10, 9], but the underlying engine has been almost completely rewritten in order to improve efficiency. The main predicate is `run_ciff( +ALP, +Query, -Answer)` where the first argument is a list of `.alp` files representing an abductive logic program with constraints, the `Query` is a query, represented as a list of literals, and `Answer` will be instantiated either to a triple with a list of abducible atoms and two lists of variable restrictions (i.e. disequalities and constraints on the variables in the `Answer` and/or in the `Query`) or to the special atom `undefined` if an allowedness condition is not met. In (any file in) `ALP`, abducible predicates `Pred`, e.g. with arity 2, are declared via `abducible(Pred(_,_))`, equality/disequality atoms are defined via `=, \==` and constraint atoms are defined via `#=, #\=, #<, #=<` and so on. Finally, negative literals are of the form `not(A)` where `A` is an ordinary atom. Clauses and ICs are represented (resp.) as

```
A :- L_1, ..., L_n.      [L_1, ..., L_m] implies [A_1, ..., A_n].
```

CIFF rewrite rules are implemented as Prolog clauses defining `sat(+State, -Answer)`, where `State` represents the current CIFF node and it is initialised, within the prolog clause defining `run_ciff( +ALP, +Query, -Answer)`, to `Query` plus all the ICs in the `ALP`. `State` is defined as:

```
state(Diseqs,CLPStore,ICs,Atoms,Abds,Disjs).
```

The predicate `sat/2` calls itself recursively until no more rules can be applied to the current `State`.Then it instantiates the `Answer`.

Below we sketch the most important techniques used to render CIFF 4.0 efficient.

Managing variables. Variables play a fundamental role in CIFF nodes: they can be either universally quantified or existentially quantified or free. A universal variable can appear only in an IC (which defines its scope). An existential/free variable can appear anywhere in the node with scope the entire node. To distinguish at run-time free/existential and universal variables we associate with the former an `existential` attribute.

Determining variable quantification efficiently is very important as CIFF proof rules for variable operations such as *equality rewriting* and *substitution* are heavily used in a CIFF computation. In CIFF 4.0 these rules are not treated as separate rewrite rules, but have been incorporated within the main rewrite rules (*propagation*, *unfolding* etc) resulting in improvements of performances.

Constraint solving. Interfacing efficiently CIFF 4.0 with the underlying CLPFD solver in Sicstus Prolog is fundamental for performance purposes. However, the CLPFD solver binds variables to numbers when checking satisfiability of constraints in the `CLPstore`, while we want to be able to return non-ground answers. The solution adopted in CIFF 4.0 is an algorithm which allows, when

needed, to check the satisfiability of the `CLPstore` as usual and then to restore the non-ground values via a forced backtracking.

Grounded integrity constraints. CIFF 4.0 adopts some specialised techniques for managing some classes of ICs, referred to as *grounded ICs*. Roughly speaking, grounded ICs are ICs whose variables will eventually be grounded during a computation, after unfolding and propagation. For example, if `p(1)` is the only clause for `p`, then the IC `[p(X)] implies [a(X)]` is grounded. If `p(1)` is replaced by `p(Y)` then the IC is not grounded anymore.

Grounded ICs are managed at a system level by exploiting both dynamic assertions/retractions of ground terms and the coroutining mechanisms of the underlying Prolog. This algorithm allows both to reduce the node size and to perform the operations on the grounded ICs efficiently because they are not physically in the node but they are dynamically maintained in the Prolog global state.

To declare an IC as grounded, the operator `implies_g` is used instead of `implies`.

4 Experimentation and Comparison

Experiments have been made on a Linux machine equipped with a 2.4 Ghz PENTIUM 4 - 1Gb DDR Ram, using SICStus Prolog version 3.11.2. Execution times are in seconds ("—" means "above 5 minutes"). Comparisons are with the A-system (`AS`) [5], the DLV system [11], and SMODELS (`SM`) [12]. In all examples, unless otherwise specified, the CIFF initial query is *true*. The adopted problem representations for the other systems are omitted due to lack of space but they can be found on the CIFF web site.

Problem 1: N-Queens. The CIFF formalisation of this problem is very simple:

```
abducible(q_pos(_,_)).                %%%ABDS

queen(X) :- q_domain(X).              %%%CLAUSES
q_domain(X) :- X in 1..n.             %%% in real code n is an integer!
exists_q(R) :- q_pos(R,C), q_domain(C).
safe(R1,C1,R2,C2) :- C1#\=C2, R1+C1#\=R2+C2, C1-R1#\=C2-R2.

[queen(X)] implies [exists_q(X)].     %%%ICS
[q_pos(R1,C1),q_pos(R2,C2),R1#\=R2] implies [safe(R1,C1,R2,C2)].
```

All systems return all the correct solutions. In the comparison below, we also include CIFF 3.0 to underline the performance improvements of CIFF 4.0.

	Queens	**CIFF 3**	**CIFF 4**	**AS**	**SM**	**DLV**
N-Queens results (first solution)	n = 8	24.75	0.03	0.03	0.01	0.01
	n = 28	—	0.29	0.27	55.32	35.17
	n = 32	—	0.37	0.32	—	—
	n = 64	—	1.62	1.52	—	—
	n = 100	—	4.55	4.24	—	—

Problem 2: Hamiltonian cycles. The CIFF 4.0 encoding makes use of the NAF module for ICs in order to avoid loops and to collect all possible answers.

```
abducible(ham_edge(_,_,_)).   abducible(checked(_,_)).  %%%ABDS

ham_cycle(X) :- ham_cycle(X,X,0).                       %%%CLAUSES
ham_cycle(X,Y,N) :- ham_edge(X,Y,N),edge(X,Y),checked(X,N).
ham_cycle(X,Y,N) :- ham_edge(X,Z,N),edge(X,Z),checked(X,N),
                    ham_cycle(Z,Y,M),M#=N+1,Z\==Y.
is_checked(V2) :- checked(V2,M).

[checked(X,N),checked(X,M),M#\=N] implies [false].      %%%ICS
[vertex(V2),not(is_checked(V2))] implies [false].
```

The predicates `edge/2` and `vertex/1` represent any given graph and are given as (domain-dependent) additional clauses, and `is_checked(V2)` is introduced to guarantee allowedness. The query is `[ham_cycle(V)]` where `V` is any vertex of the graph. In the comparison below, CIFF_G stands for CIFF but replacing the first IC by a grounded IC. We omit here a comparison with the A-system as we were unable to specify the problem avoiding looping.

	Nodes	CIFF	CIFF_G	SM	DLV
Hamiltonian cycles results (all solutions)	4	0.04	0.03	0.03	0.02
	20	0.45	0.15	0.16	0.02
	40	1.93	0.41	1.53	0.03
	80	10.95	1.20	11.41	0.04
	120	27.62	2.39	43.43	0.07

Problem 3: Web Sites repairing. The last example shows how abduction can be used for checking and repairing links in a web site, given the specification of the site via an abductive logic program with constraints. Here, a *node* represents a web page. [3]. As an example, consider a web site where a node is either a *book*, a *review* or a *library*, a *link* is a relation between two nodes and every book must have at least a link to both a review and a library. The CIFF System 4.0 formalisation of this problem (together with a simple web site instance) is:

```
abducible(add_node(_,_)).    abducible(add_link(_,_)).   %%% ABDS

is_node(N,T) :- node(N,T),node_type(T).            %%%CLAUSES
is_node(N,T) :- add_node(N,T),node_type(T).
node_type(lib).     node_type(book).    node_type(review).

is_link(N1,N2) :- link(N1,N2),link_check(N1,N2).
is_link(N1,N2) :- add_link(N1,N2),link_check(N1,N2).
link_check(N1,N2) :- is_node(N1,_), is_node(N2,_), N1 \== N2.
book_links(B) :- is_node(B,book), is_node(R,review),is_link(B,R),
                 is_node(L,lib),is_link(B,L).

[is_node(B,book)] implies [book_links(B)].     %%% ICS
[add_node(N,T),node(N,T)] implies [false].
[add_link(N1,N2),link(N1,N2)] implies [false].
[is_node(N,T1),is_node(N,T2),T1 \== T2] implies [false].
```

```
node(n1,book).  node(n3,review).  link(n1,n3). %%%WEB SITE INSTANCE
```

CIFF 4.0 returns the following answer representing correctly the need of a new link between the *book* `n1` and a new *library node* `L`:

```
Abds:   [add_link(n1,L), add_node(L,lib)].
Diseqs: [L\==n3,L\==n1]          CLP store: []
```

Notice that the variable `L` in the answer can neither be bounded to a finite domain nor grounded. This is the reason why the other systems seem unable to deal with these cases and thus no comparison is provided.

5 Conclusions

The experiments performed (including some for planning and graph-coloring omitted here for lack of space) show that CIFF 4.0 performances are comparable with other existing systems on classical problems, though allowing the exploitation of abduction on problems where non ground solutions are required. We plan to improve the treatment of (grounded) ICs, and to build a GUI for better usability. Finally, we are porting the system onto a free Prolog platform.

References

1. Kakas, A.C., Mancarella, P., Sadri, F., Stathis, K., Toni, F.: The KGP model of agency. In: Proc. ECAI-2004. (2004) 340–367
2. Sadri, F., Toni, F., Torroni, P.: An abductive logic programming architecture for negotiating agents. In: Proc. JELIA02. (2002) 419–431
3. Toni, F.: Automated information management via abductive logic agents. Journal of Telematics and Informatics (2001) 89–104
4. Kakas, A.C., Michael, A., Mourlas, C.: ACLP: Abductive constraint logic programming. Journal of Logic Programming **44** (2000) 129–177
5. Denecker, M., C.Kakas, A., Nuffelen, B.V.: A-system: Declarative problem solving through abduction. In: Proc. IJCAI 2001. (2001) 591–597
6. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: The CIFF proof procedure for abductive logic programming with constraints. In: Proc. JELIA04. (2004) 31–43
7. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming **33**(2) (1997) 151–165
8. Sadri, F., Toni, F.: Abduction with negation as failure for active databases and agents. In: Proc. AI*IA 99. (1999) 49–60
9. Endriss, U., Hatzitaskos, M., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: Refinements of the CIFF procedure. In: Proc. ARW05. (2005)
10. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: Abductive logic programming with CIFF: system description. In: Proc. JELIA04. (2004) 680–684
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (2006) 499–562
12. Niemela, I., Simons, P.: SMODELS - an implementation of the stable model and well-founded semantics for normal logic programs. In: Proc. LPNMR97. (1997) 420–429