Abductive Logic Programming with CIFF: System Description

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`

1 Introduction

Abduction has long been recognised as a powerful mechanism for hypothetical reasoning in the presence of incomplete knowledge. Here, we discuss the implementation of a novel abductive proof procedure, which we call CIFF, as it extends the IFF proof procedure [7] by dealing with Constraints, as in constraint logic programming. The procedure also relaxes the strong allowedness restrictions on abductive logic programs imposed by IFF. The procedure is described in detail in [6]. It is currently employed to realise a number of reasoning tasks of KGP agents [8] within the platform for developing agents and agent applications PROSOCS [12]. These tasks include (partial) planning in a dynamic environment [9, 5], reactivity to changes in the environment [5], temporal reasoning in the absence of complete information about the environment [2, 1], communication and negotiation [11], and trust-mediated interaction [10]. Some details on an earlier version of the system and the planning application can be found in [5]. Although the implementation of CIFF that we describe here has been tested successfully within PROSOCS in a number of settings, this is an initial prototype and more research into proof strategies and heuristics as well as fine-tuning are required to achieve satisfactory runtimes for larger examples.

2 Abductive Logic Programming with CIFF

An *abductive logic program* is a triple $\langle P, I, A\rangle$, where P is a *normal logic program (with constraints à-la CLP)*, I is a finite set of sentences in the language of P (called *integrity constraints*), and A is a set of *abducible* atoms in the language of P. A *query* Q is a conjunction of literals, possibly containing (implicitly existentially quantified) variables. An *abductive answer* to a query Q for a program $\langle P, I, A\rangle$, containing constraint predicates defined over a structure $\Re$, is a pair $\langle \Delta, \sigma\rangle$, where Δ is a set of ground abducible atoms and σ is a substitution for the variables in Q such that $P \cup \Delta\sigma \models_{\Re} I \wedge Q\sigma$. In our case, $\models_{\Re}$ represents entailment with respect to the *completion semantics* [4], extended *à-la* CLP to take the constraint structure into account.

Like IFF [7], CIFF uses an alternative representation of an abductive logic program $\langle P, I, A\rangle$, namely a pair $\langle Th, I\rangle$, where Th is a set of iff-definitions,

$p(X_1, \dots, X_k) \Leftrightarrow D_1 \vee \dots \vee D_n$, obtained by *selectively completing* P [4, 7] with respect to all predicates except *special* predicates (*true*, *false*, constraint and abducible predicates). CIFF deals with integrity constraints I which are implications: $L_1 \wedge \dots \wedge L_m \Rightarrow A_1 \vee \dots \vee A_n$, with L_i literals and A_j atoms. Any variables are implicitly universally quantified with scope the entire implication.

In CIFF, the search for abductive answers for queries Q amounts to constructing a proof tree, starting with an initial tree with root consisting of $Q \wedge I$. The procedure then repeatedly manipulates a currently selected node by applying equivalence-preserving *proof rules* to it. The nodes are sets of formulas (so-called *goals*) which may be atoms, implications, or disjunctions of literals. The implications are either integrity constraints, their residues obtained by propagation, or obtained by rewriting negative literals *not* p as $p \Rightarrow$ *false*.

IFF requires abductive logic programs and queries to meet a number of allowedness conditions (avoiding certain problematic patterns of quantification). CIFF relaxes these conditions checking them *dynamically*, i.e. at runtime, by means of a *dynamic allowedness rule* included amongst its proof rules. This rule labels nodes with a problematic quantification pattern as *undefined*. Undefined nodes are not selected again. In addition to the dynamic allowedness rule, CIFF includes, e.g., the following proof rules (for full details and other rules see [6]):

- *Unfolding:* Replace any atomic goal $p(\vec{t})$, for which there is an iff-definition $p(\vec{X}) \Leftrightarrow D_1 \vee \dots \vee D_n$ in *Th*, by $(D_1 \vee \dots \vee D_n)[\vec{X}/\vec{t}]$.
- *Propagation:* Given goals $[p(\vec{t}) \wedge A \Rightarrow B]$ and $p(\vec{s})$, add $[(\vec{t} = \vec{s}) \wedge A \Rightarrow B]$.
- *Constraint solving:* Replace a node with unsatisfiable constraints by *false*.

In a proof tree for a query, a node containing *false* is called a *failure node*. If all leaf nodes in a tree are failure nodes, then the search is said to *fail*. A node to which no more proof rules can be applied is called a *final node*. A non-failure final node not labelled as *undefined* is called a *success node*. CIFF has been proved *sound* [6]: it is possible to extract an abductive answer from any success node and if the search fails then there exists no such answer.

3 Implementation of CIFF

We have implemented the CIFF procedure in Sicstus Prolog [3] relying upon its built-in constraint logic programming solver over finite domains (CLPFD) [3]. Our implementation includes a simple module that translates abductive logic programs into completed logic programs, which are then fed as input to CIFF. The main predicate of our implementation is `ciff/4`:

```
    ciff( +Defs, +ICs, +Query, -Answer).
```

The first argument is a list of iff-definitions, the second is a list of integrity constraints, and the third is the list of literals in the given query. Alternatively, the first two arguments may be replaced with the name of a file containing an abductive logic program. The `Answer` consists of three parts: a list of abducible atoms, a list of restrictions on variables, and a list of (arithmetic) constraints (the latter two can be used to construct the σ component in an abductive answer). Iff-definitions are terms of the form A `iff` B, where A is an atom and B is a list of

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

lists of literals (representing a disjunction of conjunctions). Integrity constraints are expressions of the form A `implies` B, where A is a list of literals (representing a conjunction) and B is a list of atoms (representing a disjunction). The syntax chosen to represent atoms is that of Prolog. Negative literals are represented as Prolog terms of the form `not(`P`)` Atoms can be (arithmetic) constraints, such as `T1 #< T2 + 5`. The available constraint predicates are `#=`, `#\=`, `#<`, `#=<`, `#>`, and `#>=`, each of which takes two arguments that may be any arithmetic expressions over variables and integers (using any arithmetic operation that CLPFD can handle [3]). Note that, for equalities over terms that are not arithmetic terms, the usual equality predicate `=` should be used (e.g. `X = bob`). The Prolog predicate implementing the proof rules is:

```
sat( +Node, +EV, +CL, +LM, +Defs, +FreeVars, -Answer).
```

`Node` is a list of goals, representing a conjunction. `EV` is used to keep track of existentially quantified variables in the node (to assess the applicability of some of the proof rules). `CL` (for constraint list) is used to store the constraints accumulated so far. The next argument, `LM` (for loop management), is a list of expressions of the form A`:`B recording pairs of formulas that have already been used during a computation allowing us to avoid loops that would result if rules, for instance the *propagation* rule, were applied over and over to the same arguments. `Defs` is the list of iff-definitions in the theory. `FreeVars` is used to store the list of variables appearing in the original query. Finally, running `sat/7` will result in the variable `Answer` to be instantiated with a representation of the abductive answer found by the procedure. Each proof rule corresponds to a Prolog clause in `sat/7`. E.g., the *unfolding rule* (for atoms) is implemented as follows:

```
sat( Node, EV, CL, LM, Defs, FreeVars, Answer) :-
  member( A, Node), is_atom( A), get_def( A, Defs, Ds),
  delete( Node, A, Node1), NewNode = [Ds|Node1], !,
  sat( NewNode, EV, CL, LM, Defs, FreeVars, Answer).
```

`is_atom(A)` succeeds if `A` is an atomic goal. `get_def(A,Defs,Ds)` will instantiate `Ds` with the list of lists according to the iff-definition for `A` in `Defs` whenever there *is* such a definition (i.e. the predicate will fail for abducibles). Once `get_def(A,Defs,Ds)` succeeds we know that the unfolding rule is applicable: there exists an atomic conjunct `A` in the current `Node` and it is not abducible. The cut in the penultimate line ensures that we will not backtrack over the *order* in which rules are being applied. We generate the successor `NewNode` deleting the atom `A` from `Node` and replacing it with the disjunction `Ds`. The predicate `sat/7` then recursively calls itself with the new node.

The proof rules are repeatedly applied to the current node. Whenever a disjunction is encountered, it is split into a set of successor nodes (one for each disjunct). The procedure then picks one of these successor nodes to continue the search and backtracking over this choicepoint results in all possible successor nodes being explored. In theory, the choice of which successor node to explore next is taken nondeterministically; in practice we simply move through nodes from left to right. The procedure terminates when no more proof rules apply (to the current node) and finishes by extracting an answer from this node. Enforced backtracking will result in the next branch (if any) of the proof tree being explored, i.e. in any remaining abductive answers being enumerated.

The Prolog clauses in the implementation of `sat/7` may be reordered almost arbitrarily (only the clause used to implement answer extraction has to be listed last). Each order of clauses corresponds to a different proof strategy. This feature of our implementation allows for an experimental study of which strategies yield the fastest derivations. With respect to the implementation in [5], the current implementation integrates rules (that humans would combine automatically) into more complex rules avoiding to waste time checking for their applicability. E.g. rewriting a disjunction consisting of one disjunct as that very disjunct has been integrated into the *unfolding* rule as we have noted that rewriting unary disjunctions often occurs after an unfolding step. Another improvement concerns the *propagation* rule: we now only allow for propagation with respect to the leftmost atom in the antecedent of an implication. This refinement does not affect soundness and can reduce the number of implications in nodes. We plan to explore further optimisation techniques in the future.

Acknowledgments. This work was partially funded by the IST programme of the EC, FET under the IST-2001-32530 SOCS project, within the GC proactive initiative. The last author was also supported by the Italian MIUR programme "Rientro dei cervelli". We thank M. Gavanelli and M. Milano for suggestions on the constraint solver.

References

1. A. Bracciali, N. Demetriou, U. Endriss, A. Kakas, W. Lu, P. Mancarella, F. Sadri, K. Stathis, G. Terreni, and F. Toni. The KGP model of agency for GC: Computational model and prototype implementation. In *Proc. Global Computing 2004 Workshop*, LNCS. Springer Verlag.
2. A. Bracciali and A. C. Kakas. Frame consistency: Computing with causal explanations. In *Proc. NMR2004*.
3. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. PLILP97*.
4. K. L. Clark. Negation as failure. In *Logic and Data Bases*. Plenum Press, 1978.
5. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Abductive logic programming with CIFF: Implementation and applications. In *Proc. CILC04*.
6. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The CIFF proof procedure for abductive logic programming with constraints. In *Proc. JELIA04*.
7. T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.
8. A. C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In *Proc. ECAI2004*.
9. P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Planning partially for situated agents. Technical report, SOCS, 2004.
10. F. Sadri and F. Toni. A logic-based approach to reasoning with beliefs about trust. In *Proc. ARSPA04, Workshop affiliated to IJCAR04*, 2004.
11. F. Sadri, F. Toni, and P. Torroni. An abductive logic programming architecture for negotiating agents. In *Proc. JELIA2002*, volume 2424 of *LNCS*. Springer-Verlag.
12. K. Stathis, A. C. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali. PROSOCS: A platform for programming software agents in computational logic. In *Proc. AT2AI-2004*.