Refinements of the CIFF Procedure

Ulle Endriss[1], Markos Hatzitaskos[1], Paolo Mancarella[2], Fariba Sadri[1], Giacomo Terreni[2], and Francesca Toni[1]

[1] Department of Computing, Imperial College London
Email: {ue,mh803,fs,ft}@doc.ic.ac.uk

[2] Dipartimento di Informatica, Università di Pisa
Email: {paolo,terreni}@di.unipi.it

1 Introduction

Abductive Logic Programming (ALP) combines abductive reasoning with logic programming. In recent work [2], we have introduced a new proof procedure for ALP which we call CIFF. Our procedure extends the IFF procedure of Fung and Kowalski [3] by integrating ALP with constraint solving and by relaxing the restrictions on allowed inputs for which the procedure can operate correctly. CIFF has been applied successfully in the context of multiagent systems. An implementation of the procedure is available at `http://www.doc.ic.ac.uk/~ue/ciff/`.

In this extended abstract, we summarise several recent refinements of the CIFF procedure. After giving a brief introduction to ALP and CIFF in the next section, we firstly indicate how to extend CIFF with *negation as failure* and then discuss a number of improvements to the implementation of the procedure, pertaining to both optimisation and usability issues.

2 ALP and CIFF

An *abductive logic program* consists of a (constraint) logic program (called the *theory*) and a finite set of *integrity constraints*. In our case, the integrity constraints are implications of the form $L_1 \wedge \cdots \wedge L_m \rightarrow A_1 \vee \cdots \vee A_n$ (not to be confused with constraint predicates, such as `T#<10`). We assume that the theory is presented as a set of "iff-definitions", which may be regarded as the (selective) *completion* of a normal logic program [1].

A theory provides definitions for certain predicates and the integrity constraints restrict the range of possible interpretations. Predicates that are neither defined nor special (such as constraint predicates) are called *abducibles*. A *query* (a conjunction of literals) 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. Under a different interpretation, a query may be seen as a *goal*, in which case an answer would provide a *plan* for achieving that goal.

The CIFF procedure can be used to compute such answers. There are three possible outputs: (1) the procedure succeeds and returns an answer to the query; (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*. Allowedness relates to input formulas with certain quantification patterns for which the concept of a (finite) answer cannot be defined. Whereas the original IFF procedure [3] requires allowedness conditions to be checked in advance (with the result of being overly restrictive), CIFF can handle this issue dynamically. Our procedure has been shown to be sound (both in the case of success and in the case of failure) with respect to the completion semantics for ALP [2].

The CIFF procedure operates on so-called *nodes*. A node is a set (representing a conjunction) of formulas which are called *goals*. A proof is initialised with a node containing the integrity constraints and the literals of the query. The theory is kept in the background and is only used to *unfold* defined predicates as they are being encountered. The proof procedure repeatedly manipulates the current node by rewriting goals in the node, adding new goals to it, or deleting superfluous goals from it. If a disjunction is encountered, then the *splitting* rule can be applied, giving rise to different branches in the proof search tree. Besides *unfolding* and *splitting*, CIFF uses rewrite rules such as the following:

- *Propagation:* Given goals of the form $p(\vec{t}) \wedge A \rightarrow B$ and $p(\vec{s})$, add the goal $(\vec{t} = \vec{s}) \wedge A \rightarrow B$.
- *Negation elimination:* Replace any implication of the form $\neg A_1 \wedge \cdots \wedge \neg A_n \rightarrow B$ by $A_1 \vee \cdots \vee A_n \vee B$.
- *Case analysis for constraints:* Replace any goal of the form $Con \wedge A \rightarrow B$, where Con is a constraint not containing any universally quantified variables, by $[Con \wedge (A \rightarrow B)] \vee \overline{Con}$.

A node containing an apparent contradiction is called a failure node. If all branches in a derivation terminate with failure nodes, then the procedure is said to fail (*i.e.* there exists no answer to the query). A non-failure node to which no more proof rules apply (and that is allowed) can be used to extract an answer by collecting all the atomic abducibles (and constraints) in that node.

3 Negation as failure

We have found that ALP and CIFF are useful formalisms for modelling intelligent agents. Abducibles can be used to represent the actions available to an agent and integrity constraints can be used to specify an agent's behaviour by means of reactive rules. However, in some cases the classical treatment of negation in CIFF can lead to non-intuitive answers being computed. To exemplify the problem, consider the following integrity constraint:

$$request \wedge \neg have \rightarrow reject$$

Intuitively, this (simplified) rule expresses that, if a request has been made to an agent for an object and the agent does not have that object, then the agent should reject the request. But given the observation *request*, CIFF will first propagate that observation with the integrity constraint to get $(\neg have \rightarrow reject)$ and then rewrite

the latter as $(\neg\neg have \vee reject)$ using negation elimination. Thus, CIFF will compute the intuitively correct answer $\{request, reject\}$, as well as the non-intuitive answer $\{request, have\}$. The second answer says that an agent using CIFF might react to a request for an object by suddenly "having" that object (which does, of course, not make sense as "having" should not be treated as an action available to the agent).

We have therefore investigated a different way of treating negation within (the residues of) integrity constraints, namely *negation as failure* [1]. Following the approach proposed in [5], we have integrated *negation as failure* into CIFF. In the modified version of the procedure, all implications in a node are either *marked* (using the symbol $*$) or *unmarked*. Initially, all integrity constraints are marked (but negative literals in the query, which we represent as $A \rightarrow \bot$, are not). Every proof rule can be applied to marked implications, except for negation elimination. We define a new rule to handle negation:

- *Negation rewriting:* Replace any marked implication of the form $*(\neg A_1 \wedge \cdots \wedge \neg A_n \rightarrow B)$ by the disjunction $\mathtt{pr}(A_1) \vee \cdots \vee \mathtt{pr}(A_n) \vee [*(A_1 \rightarrow \bot) \wedge \cdots \wedge *(A_n \rightarrow \bot) \wedge B]$.

Here $\mathtt{pr}(A)$ stands for "A is provably true". Given a negative element A in a marked implication, negation rewriting can be thought of as either proving that A is true or, otherwise, taking A to be false and considering the rest of the implication. With respect to our earlier example, we would still propagate the observation *request* with the integrity constraint, but instead of the second proof step, it would now rewrite the negation as follows:

$$\mathtt{pr}(have) \vee [*(have \rightarrow \bot) \wedge reject]$$

Intuitively, this means that either the agent (can prove that it) has got the object in question (and thus could accept the request to give it away), or the agent does not have the object and has to reject the request. After splitting the disjunction, the new CIFF would then try to prove $\mathtt{pr}(have)$ and fail (meaning that the corresponding node cannot be used to extract an answer). Hence, only the answer $\{request, reject\}$, corresponding to the second disjunct, would get generated.

We are currently implementing *negation as failure* in CIFF and a first version of the system is described in [4].

4 Implementation refinements

CIFF has been implemented in Sicstus Prolog and constraint solving is delegated to its built-in finite domain solver. Since the first release of CIFF we have implemented a number of improvements of the system, most of which concern (high-level) optimisation issues; e.g.:

- *Guided propagation.* The propagation rule of CIFF (and IFF) allows for an atomic formula to be resolved with *any* matching atom in the antecedent of an implication. This rule can be refined by allowing propagation only with respect to the leftmost atom, which can reduce the number of residues to be considered.
- *Ordering of proof rules.* We have experimented with different ways of ordering proof rules to identify the configuration most suitable for typical queries. Concerning the main rules, our new heuristics give highest priority to the constraint rules and lowest priority to unfolding within implications. Minor proof rules, such as rewriting according to logical equivalences, are also given lower priority than before.
- *Simple rules.* Our implementation includes certain proof rules that humans would apply implicitly, such as rewriting a disjunction consisting of a single disjunct as that very disjunct. Our original heuristic has been to give high preference to such rules, while our new approach is not to waste computing time on constantly testing for their applicability, but rather to integrate them into the more complex rules.

Recently, we have also made a number of improvements regarding the usability of the system, in particular with respect to the representation of constraint predicates in extracted answers:

- *Constraint simplification.* While the original system (being a faithful implementation of the abstract proof procedure) only uses the Sicstus constraint solver to *check* whether a given set of constraints is satisfiable, we now also simplify such satisfiable sets of constraints before reporting them to the user.
- *Projection.* CIFF often generates auxiliary variables during execution, which should be removed from the set of constraints returned as part of an answer. We are therefore currently working on a refinement of CIFF that would allow us to project the answer constraints onto the relevant set of variables. While, in the general case, no efficient algorithm for projection onto (integer) variables is known, in the context of CIFF we can take advantage of the fact that (a) the overall number of variables tends to be relatively small, and (b) we only need to consider a limited range of constraint patterns.

Acknowledgements. This research has been supported by the European Commission as part of the SOCS project (IST-2001-32530).

References

[1] K. L. Clark. Negation as failure. In *Logic and Data Bases*. Plenum Press, 1978.

[2] U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The CIFF proof procedure for abductive logic programming with constraints. In *Proc. 9th European Conference on Logics in Artificial Intelligence*. Springer-Verlag, 2004.

[3] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.

[4] M. Hatzitaskos. Extending the CIFF proof procedure with negation as failure and implementing part of a multi-stage negotiation architecture for sharing resources amongst logic-based agents. Master's thesis, Dept. of Computing, Imperial College London, 2004.

[5] F. Sadri and F. Toni. Abduction with negation as failure for active and reactive rules. In *Proc. 6th Congress of the Italian Association for Artificial Intelligence*. Springer-Verlag, 1999.