SOCS

A COMPUTATIONAL LOGIC MODEL FOR THE DESCRIPTION, ANALYSIS AND VERIFICATION OF GLOBAL AND OPEN SOCIETIES OF HETEROGENEOUS COMPUTEES

IST-2001-32530

Implementing the CIFF Procedure

Project number:	IST-2001-32530
Project acronym:	SOCS
Document type:	IN (information note)
Document distribution:	PU (open distribution without limitation)
CEC Document number:	IST32530/ICSTM/???/IN/PU/a1
File name:	1???-a1[ciff-implementation].pdf
Editor:	Ulle Endriss
Contributing partners:	ICSTM
Contributing workpackages:	WP4
Estimated person months:	—
Date of completion:	5 December 2003
Date of delivery to the EC:	—
Number of pages:	15

ABSTRACT
This is a preliminary and incomplete draft of a paper on our implementation of the CIFF proof procedure for abductive logic programming with constraints.

Implementing the CIFF Procedure

Ulle Endriss

Department of Computing, Imperial College London
Email: ue@doc.ic.ac.uk

ABSTRACT
This is a preliminary and incomplete draft of a paper on our implementation of the CIFF proof procedure for abductive logic programming with constraints.

Contents

1 Introduction

This paper describes our implementation of the CIFF proof procedure for abductive logic programming with constraints.[1] This proof procedure is both an extension and a refinement of the IFF proof procedure proposed by Fung and Kowalski [2]. The implementation has been carried out in the context of the SOCS project funded by the European Commission and forms the basis of the implementation of the planning, reactivity, and temporal reasoning capabilities of the logic-based autonomous agents developed in this project.

Our description of the CIFF procedure is structured as follows. We first give an overview of the CIFF module as a whole. Then we are going to define the syntax of abductive logic programs used in the implementation and show how to use the module, that is, how to run the procedure for a given ALP. We also include here the description of an auxiliary tool to transform a logic program into a list of iff-definitions, i.e. to compute the completion of a given logic program. This is used to preprocess theories provided in the form of facts and rules before passing them on to the CIFF proof procedure. We are then going to turn to various details of the implementation, in particular how the theoretical description of a set of proof rules without a specific proof strategy has been turned into an actual procedure. Finally, we are going to discuss the integration of a constraint solver into the overall system.

2 Structure of the CIFF Module

The CIFF proof procedure has been implemented in Sicstus Prolog [3]. Most of the code could very easily be ported to any other Prolog system conforming to standard Edinburgh syntax.[2] The CIFF module consists of four Prolog files:

- `ciff-main.pl` provides the main predicates (`ciff/3` and `ciff/4`) as well as tools to read ALPs and to configure the debugging facility. Consulting this file will also cause the compilation of the other program files.
- `ciff-proc.pl` implements the actual proof procedure. The central predicates are `sat/7` defining the proof rules and `simplify_equalities/3` implementing the rewrite rules for equalities.
- `ciff-constraints.pl` provides the wrapper around the constraint solver. The most important predicates are `post_constraint/1` to post a constraint to the constraint store and to perform a fast (but incomplete) consistency check, and `exhaustive_check/1` to perform a complete consistency check (by enumeration). Additional predicates such as `is_constraint/1` allow for a complete encapsulation of the actual constraint solver.
- `ciff-aux.pl` implements various auxiliary predicates.

From a user's perspective, only `ciff-main.pl` is of interest and in the first part of this description (that is, in Sections 3 to 7), where we are going to describe the syntax of ALPs (i.e. the

[1] To obtain a copy of the system either visit `http://www.doc.ic.ac.uk/~ue/ciff/` or contact the author.

[2] A small exception is the module concerned with constraint solving as it relies on Sicstus' built-in constraint logic programming solver over finite domains (CLPFD) [1]. However, the modularity of our implementation would also make it relatively easy to integrate a different constraint solver into the system. The only changes required would be an appropriate re-implementation of a handful of simple predicates providing a wrapper around the constraint solver chosen for the current implementation.

```
grass_is_wet iff [[rain_last_night], [sprinkler_was_on]].
[rain_last_night] implies [cloudy_last_night].
[cloudy_last_night] implies [false].
```

Table 1: The ALP for the grass-is-wet example

input to the CIFF procedure) and explain how to run the procedure, we are only going to refer to predicates provided by that program file. From a technical point of view, `ciff-proc.pl` is the core of the CIFF module. In the second part (starting with Sections 8), where we are going to comment on details of the implementation, we are mostly going to refer to predicates implemented in that file. Finally (in Section 13), we are going to explain `ciff-constraints.pl` and discuss how to integrate an alternative constraint solver into our system.

3 Syntax of Abductive Logic Programs

An abductive logic program with constraints is a list of definitions and integrity constraints. If provided in a file, a program may also be given as a collection of clauses (rather than as a Prolog list). Table 1 shows a very simple ALP for propositional logic taken from [2]. The predicate `read_alp/3` may be used to read such a file into two lists (one for the definitions and one for the integrity constraints). Example:

```
?- read_alp( 'grass.alp', Defs, ICs).
ICs = [[rain_last_night]implies[cloudy_last_night],
       [cloudy_last_night]implies[false]],
Defs = [grass_is_wet iff[[rain_last_night],[sprinkler_was_on]]] ?
yes
```

We are now going to define the syntax of definitions and integrity constraints using a variant of BNF where `[`*Term*`+]` denotes a non-empty Prolog list of elements of type *Term*. Definitions have the following structure:

Definition `::=` *Pred* `iff [[`*Literal*`+] +]`

The list of lists on the righthand side represents a disjunction of conjunctions of literals. *Pred* is a Prolog term representing an atomic formula (not an equality or a constraint) all of whose arguments (if any) are variables. A typical example would be `p(X,Y)`. Integrity constraints are represented as follows:

IC `::=` `[`*Literal*`+] implies [`*Atom*`+]`

Here the list on the lefthand side represents a conjunction of literals, while that on the righthand side represents a disjunction of atoms.

A query, which is itself not part of an abductive logic program, is a list of literals representing a conjunction:

Query `::=` `[`*Literal*`+]`

A literal is either an atom or a negated atom:

Literal `::=` *Atom* `|` `not(`*Atom*`)`

Internally, negative literals will be represented as implications and this syntax is also admissible as an input to the proof procedure, i.e. you may write `[`*Atom*`]` `implies` `[false]` instead of `not(`*Atom*`)`. Atoms themselves may be either equalities, constraints, or "normal" atomic predicates consisting of a functor and any number of arguments:

Atom `::=` *Pred* `|` *Equality* `|` *Constraint* `|` `true` `|` `false`

Here, *Pred* stands for such a "normal" atomic predicate. Technically, this could be any Prolog term that is neither a variable nor a term representing an equality or a constraint. A typical example would be `p(X,a)`. The predicates `true` for $\top$ and `false` for $\bot$ are listed separately as they have a fixed semantics.

The next type of atoms, namely equalities, are represented as follows:

Equality `::=` *Term* `=` *Term*

Here, each occurrence of *Term* may be instantiated with any Prolog term. Typical examples are `X`, `a`, and `f(Y,b)`.

Finally, the exact syntax of constraints depends on the constraint solver used together with the program. We give here the syntax for the solver currently implemented, which is a wrapper around the built-in CLP finite domain solver of Sicstus Prolog [1, 3] covering a number of simple arithmetic constraints:

Constraint `::=` *ArithTerm* `#=` *ArithTerm* `|` *ArithTerm* `#\=` *ArithTerm* `|`
ArithTerm `#<` *ArithTerm* `|` *ArithTerm* `#=<` *ArithTerm* `|`
ArithTerm `#>` *ArithTerm* `|` *ArithTerm* `#>=` *ArithTerm*

Arguments to constraints may be simple arithmetic expressions over variables and integers:

ArithTerm `::=` *Var* `|` *Num* `|` *ArithTerm* `+` *ArithTerm* `|`
ArithTerm `-` *ArithTerm* `|` *ArithTerm* `*` *ArithTerm*

Here *Var* stands for Prolog variables while *Num* may be instantiated with any (suitably small) integer. Other arithmetic operators (such as division) may also be used as long as they are provided by the Sicstus CLPFD module. Using any of the additional arithmetic *relations* provided by that module, on the other hand, would also require minor modifications to the wrapper module connecting the constraint solver with the main program.

4 Allowedness

We should stress here that not every list of definitions and integrity constraints following the syntax definitions given in the previous section constitutes an *allowed* abductive logic program. Additional restrictions are required to be able to avoid the explicit representation of quantifiers:

- A definition is allowed iff any variable occurring either in a constraint in a disjunct on the righthand side or anywhere in a disjunct on the righthand side but not on the lefthand side also occurs in a non-constraint/non-equality atom in that disjunct.
- An integrity constraint is allowed iff any variable occurring anywhere in the integrity constraint also occurs in a non-constraint atom in the antecedent.

- A query is allowed iff any variable occurring anywhere in the query also occurs in a non-constraint atom in the query.

For inputs that are not allowed in this sense, the program may return incorrect answers. Still, in certain cases the allowedness restrictions may be relaxed a little, but this requires in-depth knowledge of the proof procedure.

5 Completing Logic Programs

The CIFF procedure is defined over *completed* logic programs, i.e. the logic program in the input is required to be a list of predicate definitions in iff-form rather than a list of rules (in if-form) and facts. As these definitions can become rather long and difficult to read, our implementation includes a simple module that translates logic programs into completed logic programs which may be used as input to the CIFF procedure. Being able to complete logic programs on the fly also allows us to spread the definition of a particular predicate over different knowledge bases.

The main predicate of this module (program file `closure.pl`) is `close_program/4`. The third argument is a list of clauses (Prolog rules and facts) and the fourth argument will be instantiated with a list of iff-definitions. The first argument is a list of predicate names for which we require an iff-definition whether or not that predicate is actually mentioned in the program listing provided at the third argument position. This feature is useful to declare predicates not covered by a logic program as *not* being abducible. The list of iff-definitions returned will then include a definition that declares the respective predicate as being equivalent to the "empty disjunction", that is, as being equivalent to $\bot$. Finally, the third argument position may be used to provide a list of predicates for which no iff-definition should be returned, whether or not that predicate actually occurs in the program listing. This latter feature is useful, for instance, when we parse files that include both a logic program and a list of integrity constraints. In that case, we are not interested in the "iff-definition" for the predicate `implies/2` (the operator used to represent integrity constraints). Here is a simple example for the use of `close_program/4`:

```
?- close_program( [p/1], [], [(q(X,Y):-p(X),r(X,Y)), q(47,11)], Defs).
Defs = [p(_A)iff[],q(_B,_C)iff[[_B=X,_C=Y,p(X),r(X,Y)],[_B=47,_C=11]]] ?
yes
```

The predicate `q/2` is defined in terms of one rule and one fact. The resulting iff-definition is a list of list, representing a disjunction of conjunctions: `q(_B,_C)`, where `_B` and `_C` are new variable names chosen by Sicstus Prolog, will be true if (and only if) either the rule or the fact apply. The predicate `p/2` supplied in the first argument position has been defined as `[]`, i.e. as a disjunction with no arguments. The predicate `r/2` appearing in the body of the definition of `q/2` would be interpreted as an abducible predicate by CIFF, because there is no iff-definition for it in the completed program.

6 Running CIFF and Sample Sessions

We are now going to see a small number of sample runs of the proof procedure. After having consulted the main Prolog file in the CIFF module (`ciff-main.pl`), we may use either `ciff/3` or `ciff/4` to invoke the procedure on a given ALP and a given query. For `ciff/3`, the first argument is the name of a file containing a collection of iff-definitions and integrity constraints.

```
lamp(X) iff [[X=a]].
battery(X,Y) iff [[X=b, Y=c]].
faulty_lamp(X) iff [[lamp(X), broken(X)], [power_failure(X), not(backup(X))]].
backup(X) iff [[battery(X,Y), not(empty(Y))]].
```

Table 2: The ALP for the faulty-lamp example

For `ciff/4`, the first argument is a list of definitions and the second is a list of integrity constraints.[3] In either case, the penultimate argument is used to supply a query, i.e. a list of literals, and the variable given in the final argument will be instantiated with an answer to that query.

If backtracking into either one of these two predicates is forced (by the user or some other part of the system using CIFF) all answers will be enumerated in turn. Note, however, as it is possible to define cyclic theories and given that the problem of finding an abductive answer is not generally decidable, termination cannot always be guaranteed.

The answers returned by the `ciff` predicates consist of three lists each. The first list contains the abducible atoms of the abductive answer. The second list reports on the substitution for the variables occurring in the query as well as any other variables occurring in the first part of the answer. Substitutions may be either "positive" (e.g. `X/a47` means that variable `X` should be instantiated with the value `a47`) or "negative" (e.g. `not(Y/a11)` means that `Y` may be instantiated with any value *but* `a11`). Note that only the latter kind of substitution will ever be required for variables not occurring in the query. The third list contains the constraints associated with an answer. As for the list of substitutions, these constraints may involve variables from the query as well as other variables created during computation. Any particular variable may only occur either in the list of substitutions *or* in the list of constraints, but not in both of them. By default, any of the variables in the query will appear in the list of substitutions *unless* its use in the query or the input ALP indicates that it must be a constraint variable.[4]

Our first example is taken from [2] and does not include any constraints. The abductive logic program is given in Table 2 (note that the example does not contain any integrity constraints, only definitions).[5] Assuming that this program is stored in a file called `lamp.alp`, we may run the following query:

```
?- ciff( 'lamp.alp', [faulty_lamp(X)], Answer).
Answer = [broken(a)]:[X/a]:[] ? ;
Answer = [empty(c),power_failure(b)]:[X/b]:[] ? ;
Answer = [power_failure(X)]:[not(X/b)]:[] ? ;
no
```

[3] In fact, `ciff/3` is implemented in terms of `ciff/4`: It first reads the content of the file supplied and splits it into a list of definitions and a list of integrity constraints and then calls `ciff/4`.

[4] A variable must be a constraint variable not only if it is itself used within a constraint, but also, for instance, if it is used in the same argument position of a predicate as some other variable which in turn *is* used within a constraint.

[5] Note that this example has been tweaked to get "interesting" answers: faulty lamps are defined as either devices that are lamps and broken or devices that experience a power failure and do not have any backup (whether or not these devices have also been defined as lamps themselves is not relevant). This explains the second and the third answer obtained.

Here the user has enforced backtracking, so all three answer are being reported by the system. Note that the third (empty) list in each of the answers would be used to store the associated constraints.

In the first answer, the observation `faulty_lamp(X)` ("there is a lamp `X` that is faulty") has been explained by the assumption that the lamp in question is `a` and that `a` is broken. In the second answer, the explanation given is that `X` refers to `b` which is a device that experiences a power failure and whose battery `c` is empty. The final answer is an example for a "negative" substitution: the query can be explained as long as `X` is not being instantiated with `b` and we assume that `X` is a device experiencing a power failure.

Note that an answer substitution will not cover any free variables that could be instantiated arbitrarily. As an example, consider the following trivial query where `X` could have any value:[6]

```
?- ciff( [], [], [X=X], Answer).
Answer = []:[]:[] ?
yes
```

7 Debugging and Reviewing Proofs

The current implementation includes a rudimentary debugging facility which enables the user to review the proofs generated by the system. The predicate `switch_debugging/1` lets you switch between different debugging modes. If the argument supplied is `on`, Prolog will print out the name of the rule applied, the formula(s) the rule has been applied to, and a representation of the new proof node for each and every proof step. If the argument given is `step`, Prolog will also wait for the user to press the Return key after each step. The default argument is `off`, in which case no output occurs.

Table 3 shows an example. The abductive logic program loaded is the grass-is-wet example of Table 1 (assumed to be stored in a file called `grass.alp`). The observation "the grass is wet" is explained by the assumption that the sprinkler has been on.

8 Data Structure and Proof Rules

We are now going to turn our attention to the actual implementation of the CIFF proof procedure and shall explain some of the design decisions made during its development. The procedure manipulates, essentially, a set of formulas that are either atoms or implications (the latter coming from the integrity constraints).[7] The set of definitions of the abductive logic program 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.

In the terminology of CIFF, the sets of formulas manipulated by the procedure are called *nodes*. A node is a set (representing a conjunction) of formulas (atoms, implications, or disjunctions thereof) which are called *goals*. Furthermore, in [2], the term *frontier* has been used to refer to a set of nodes (representing a disjunction, i.e. all the branches in the proof tree). In

[6] The ALP, represented by the first two lists passed as arguments to `ciff/4`, is empty for this example.

[7] Note that ALPs and queries are preprocessed and all negative literals of the form `not(`*Atom*`)` are rewritten as implications of the form `[`*Atom*`] implies [false]`.

```
?- switch_debugging( on).
yes
?- ciff( 'grass.alp', [grass_is_wet], Answer).
initialise:
node: [grass_is_wet,[rain_last_night]implies[cloudy_last_night],
[cloudy_last_night]implies[false]]
unfold atom: grass_is_wet iff[[rain_last_night],[sprinkler_was_on]]
node: [[[rain_last_night],[sprinkler_was_on]],
[rain_last_night]implies[cloudy_last_night],[cloudy_last_night]implies[false]]
splitting: [[rain_last_night],[sprinkler_was_on]]
node: [rain_last_night,[rain_last_night]implies[cloudy_last_night],
[cloudy_last_night]implies[false]]
propagation: [rain_last_night]implies[cloudy_last_night]:rain_last_night
node: [[]implies[cloudy_last_night],rain_last_night,
[rain_last_night]implies[cloudy_last_night],[cloudy_last_night]implies[false]]
simplification ([] -> A): []implies[cloudy_last_night]
node: [[[cloudy_last_night]],rain_last_night,
[rain_last_night]implies[cloudy_last_night],[cloudy_last_night]implies[false]]
rewriting unary disjunction: [[cloudy_last_night]]
node: [cloudy_last_night,rain_last_night,[rain_last_night]implies[cloudy_last_night],
[cloudy_last_night]implies[false]]
propagation: [cloudy_last_night]implies[false]:cloudy_last_night
node: [[]implies[false],cloudy_last_night,rain_last_night,
[rain_last_night]implies[cloudy_last_night],[cloudy_last_night]implies[false]]
simplification ([] -> A): []implies[false]
node: [[[false]],cloudy_last_night,rain_last_night,
[rain_last_night]implies[cloudy_last_night],[cloudy_last_night]implies[false]]
rewriting unary disjunction: [[false]]
node: [false,cloudy_last_night,rain_last_night,
[rain_last_night]implies[cloudy_last_night],[cloudy_last_night]implies[false]]
removed failure node: false
node: [false]
splitting: [[rain_last_night],[sprinkler_was_on]]
node: [sprinkler_was_on,[rain_last_night]implies[cloudy_last_night],
[cloudy_last_night]implies[false]]
final constraint checking (success): []
node: [sprinkler_was_on,[rain_last_night]implies[cloudy_last_night],
[cloudy_last_night]implies[false]]
Answer = [sprinkler_was_on]:[]:[] ?
yes
```

Table 3: Using the debugging facility to review proofs

our implementation, however, we do not represent frontiers explicitely, but use backtracking to visit the different branches in a proof tree in turn.

A proof is initialised with the node containing the integrity constraints in the program and the literals of the query. 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. These *proof rules* used to manipulate the current node and to derive its successor node are described in detail in [2].[8] Whenever a disjunction is encountered, the current node is split into a set of successor nodes (one for each disjunct). The procedure then picks one of these successor nodes to continue the proof 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 the final node (see Section 12). Enforced backtracking will result in the next branch of the proof tree being explored, i.e. in the remaining abductive answers being enumerated.

The Prolog predicate implementing proof rules in `ciff-proc.pl` is called `sat/7` (as it is used to *saturate* a set of formulas):

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

`Node` is a list of goals, representing a conjunction. Given our earlier discussion, from a syntactic point of view, we can distinguish three kinds of goals: (1) Prolog terms representing atoms; (2) Prolog terms of the form `[`*Atom*`+] implies [`*Literal*`+]` representing implications (residues of integrity constraints) whose antecedents are conjunctions of atoms and whose consequents are disjunctions of literals; and (3) lists of lists of either of the above, representing disjunctions of conjunctions of formulas.

`EV`, the second argument of `sat/7`, is used to keep track of existentially quantified variables in the node. This set is relevant to assess the applicability of some of the proof rules. The details of this aspect of the proof procedure will be discussed in Section 10.

`CL` (for constraint list) is the list of CLP constraints that have so far been encountered within the respective node. In fact, `CL` is a pair of lists `CL1:CL2`. `CL1` is the list of constraints and `CL2` is a copy of that list. The constraints actually posted to the constraint solver are taken from `CL2`, while `CL1` is used to report answers. This avoids variables in the answer being instantiated, i.e. we can report substitutions and constraints on the meta-level.

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 with particular proof rules, thereby allowing us to avoid loops by applying these rules over and over to the same arguments (this is necessary for both the *propagation* and the *factoring* rule). Details on loop management are discussed in Section 11.

`Defs` is a list of iff-definitions, i.e. terms of the category *definition*. They represent the completed logic program with respect to which we are evaluating the query.

The penultimate argument, `FreeVars`, is used to store the list of free variables, i.e. the list of variables appearing in the original query. These are only needed during answer extraction at the very end.

Finally, running `sat/7` will result in the variable `Answer` given in the final argument position to be instantiated with a representation of the abductive answer found by the procedure. The

[8] To be able to deal with constraints, two additional rules are required: a *case analysis* rule for constraints similar to the case analysis rule for equalities and a rule that checks the set of constraints in a node for consistency.

format chosen for reporting answers has already been presented in Section 6; how an answer is actually being extracted from a saturated node will be explained in Section 12.

Let us now look at an example of how proof rules have been implemented. Each rules corresponds to a Prolog clause in the implementation of `sat/7`. The *unfolding rule for atoms* is used to replace an atom in a node with its definition according to the ALP in question. this rule has been 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],
  inform( 'unfold atom', A iff Ds, NewNode),
  sat( NewNode, EV, CL, LM, Defs, FreeVars, Answer).
```

The auxiliary predicate `is_atom/1` will succeed whenever the argument represents an atom (i.e., in particular, whenever it is not an implication). Furthermore, `get_def( A, Defs, Ds)`, with the first two arguments being instantiated at the time of calling the predicate, will instantiate `Ds` with the list of lists representing the disjunction that defines the atom `A` according to the iff-definitions given in `Defs` whenever there *is* such a definition (i.e. the predicate will fail for abducible predicates). The appropriate substitutions in `Ds` are also made within `get_def/3`. Once `get_def( A, Defs, Ds)` succeeds we definitely know that the unfolding rule is applicable to the current: There exists and atom `A` in the current `Node` and it is not abducible. Therefore, this is the right point to insert a cut into the clause. We do not want to allow any backtracking over the order in which rules are being applied. After we are certain that this rule should be applied we manipulate the current `Node` and generate its successor `NewNode`. We first delete the atom `A` and then replace it with the disjunction `Ds`. The predicate `sat/7` then recursively calls itself with the new node.

The predicate `inform/3` is used by the debugger to display the name of the rule that has been applied, the formulas involved, and the new node. Whether or not this information will indeed be printed out depends on the chosen debugging mode (see Section 7).

We are going to see further examples for the implementation of particular proof rules in the following sections where we are going to look at a number of specific aspects of the implementation of `sat/7`.

It should be noted that the Prolog clauses in the implementation of `sat/7` may be reordered almost arbitrarily (the only requirement is that the clause used to implement answer extraction is listed last). Each order of clauses corresponds to a different proof strategy, as it implicitly assigns different priorities to the different proof rules. This feature of our implementation would, in principle, allow for an experimental study of which strategies yield the fastest derivations. The order in which proof rules are applied in the current implementation follows some simple heuristics. For instance, logical simplification rules as well as rules to rewrite equality atoms are always applied first. Splitting, on the other hand, is one of the last rules to be applied.

9 Rewriting Equalities

Several of the proof rules of the CIFF proof procedure are concerned with rewriting equalities. These rules are implemented in `ciff-proc.pl` via the predicate `simplify_equalities/3`:

```
simplify_equalities( +Eqs, +EV, -NewEqs).
```

Given a list of equalities `Eqs`, this predicate will return the list `NewEqs` using the result of exhaustively applying the rewrite rules for equalities given in [2].[9] The predicate manipulates lists of equalities (rather than single equalities), because of the rules (that for decomposing compound terms) rewrites a single equality as a list of equalities. Reference to the list `EV` of existentially quantified variables is required for one of the rules in order to be able to determine whether a variable is universally quantified or not.

A simple example for a clause in the implementation of `simplify_equalities/3` would be the following which expresses that an equality should be reordered in case its righthand argument is a variable and its lefthand argument is not:

```
simplify_equalities( [T=X|Rest], EV, List) :-
  var( X), \+ var( T), !,
  simplify_equalities( [X=T|Rest], EV, List).
```

The predicate `simplify_equalities/3` is called from within `sat/7` whenever the latter encounters either an equality atom in the current node or an equality atom inside an implication in the current node.

10 Keeping Track of Existentially Quantified Variables

Quantification of variables in the formulas manipulated by the CIFF procedure is implicit. Variables are either free, existentially quantified, or universally quantified. Free variables are those that appear in the original query. Amongst the variables that are not free, existentially quantified variables are those that appear in an atom inside the current node. All remaining variables are universally quantified. For all practical purposes, free variables are treated in the same way as existentially quantified variables (except for the fact that free variables play a special role in the context of answer extraction).

For the applicability of some of the proof rules, the type of quantification of the variables involved may matter. The rules affected are the *case analysis* rule for both equalities and constraints, the *equality rewriting* rule used to reorder equalities of the form $X = Y$ where both X and Y are variables, and the *substitution rule* rule for equalities occurring in the antecedent of an implication:

- *Case analysis for equalities:* An implication of the form $(X = t \wedge A) \Rightarrow B$ (where X is a variable and the term t does not contain X) may be rewritten as a disjunction $[X = t \wedge (A \Rightarrow B)] \vee [X = t \Rightarrow \bot]$ *provided* X is either free or existentially quantified and t is not a universally quantified variable.

- *Case analysis for constraints:* An implication of the form $(C \wedge A) \Rightarrow B$ (where C is a constraint atom) may be rewritten as the disjunction $[C \wedge (A \Rightarrow B)] \vee \overline{C}$[10] *provided* all variables occurring in C are either free or existentially quantified.

- *Rewriting equalities of the form $X = Y$:* An equality of the form $Y = X$ (where both X and Y are variables) that occurs either as an atom in the current node or as a conjunct in

[9] Note that only rules 1–5 in the paper by Fung and Kowalski [2] are in fact true rewrite rules. The remaining two rules (6*a* and 6*b*) are substitution rules that regulate how to "apply" fully simplified equalities to other formulas.

[10] Here, $\overline{C}$ denotes the *complement* of the constraint C. For example, the complement of the constraint $X \leq 42$ would be $X > 42$.

the antecedent of an implication in that node, may be rewritten as $X = Y$ *provided* that X is universally quantified and Y is not.

- *Substitution within implications:* An implication of the form $(X = t \wedge A) \Rightarrow B$ (where X is a variable and the term t does not occur in X) may be rewritten as the implication $(A \Rightarrow B)[X/t]$, i.e. as the formula we obtain by replacing all occurrences of X within $A \Rightarrow B$ by the term t, *provided* X is universally quantified.

Observe that we only need to be able to distinguish whether a variable is universally or existentially quantified in case it appears in either an atom or an implication that is itself a conjunct of a node, that is, for a variable occurring only within a disjunction inside a node the decision whether that variable is universally or existentially quantified can be postponed. In such a case, the proof rule in question would only become applicable after splitting. This observation allows us to maintain a list of existentially quantified variables (including, for practical reasons, the free variables) in a very simple manner rather than trying to infer the implicit quantification from the structure of the node whenever the applicability of one of the above rules needs to be verified.

This list is `EV`, the second argument in the predicate `sat/7` discussed earlier. It is initialised with the list of free variables. Then, whenever the splitting rule is applied (which is the only rule that may result in new atoms being added to a node), `EV` gets updated by appending the variables occurring in the new atom.[11]

11 Loop Management

The CIFF proof procedure may loop for a number of different reasons. For example, if we run the procedure on an ALP that includes a cyclic definition such as $P \Leftrightarrow P \vee Q$, the unfolding rule will be applicable over and over again. We are *not* going to deal with problems of looping of this kind here. One point of view would be that it is the responsibility of the user to provide non-cyclic theories as the input.

The problem we are going to address here are loops caused by the fact that a rule may apply again and again to the same arguments. This is, for instance, *not* the case for the unfolding rule. After having replaced an atom P by the disjunction defining it according to our ALP, the original atom P get deleted from the node, i.e. this instance of the rule application could not possibly be repeated.[12] [to be completed]

12 Answer Extraction

[to do]

[11] Note that our implication also includes a "special" splitting rule to rewrite unary disjunctions as simple atoms. For this rule, `EV` needs to be updated as well. The reason for making this a separate rule is that (unlike for splitting in general), it seems reasonable to rewrite unary disjunctions with a very high priority as they do not lead to a branching point in the search tree.

[12] The pathological case discussed before where P gets replaced by a disjunction one of whose disjuncts is itself P is different, as the unfolding rule would not be applied to the same but a new copy of that formula.

13 Integration of the Constraint Solver

[to do] [Planned: a brief explanation of what has been done and a discussion of how to integrate a different constraint solver, emphasising how easy this would be due to modularity]

14 Conclusion

[to do]

References

[1] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.

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

[3] Sicstus Prolog User Manual (Release 3.11.0). Technical report, Swedish Institute of Computer Science, October 2003.