Interpreting Abduction in CLP *

M. Gavanelli[1], E. Lamma[1], P. Mello[2], M. Milano[2], and P. Torroni[2]

[1] Dip. di Ingegneria, Università di Ferrara
Via Saragat 1, 44100 Ferrara, Italy
{mgavanelli, elamma}@ing.unife.it
[2] DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
{pmello, mmilano, ptorroni}@deis.unibo.it

Abstract. Constraint Logic Programming (CLP) and Abductive Logic Programming (ALP) share the important concept of *conditional answer*. We exploit their deep similarities to implement an efficient abductive solver where abducibles are treated as constraints. We propose two possible implementations, in which integrity constraints are exploited either (*i*) as the *definition* of a CLP solver on an abductive domain, or (*ii*) as constraints à la CLP. Both the solvers are implemented on top of CLP(Bool), that typically have impressively efficient propagation engines.

1 Abduction in CLP

Abduction and Constraint Logic Programming have been successfully integrated in various works [1] [2].

Abductive reasoning is aimed at inferring hypotheses about unknowns: typically, some predicates are labelled as *abducibles* and treated in a special way. It is an extremely powerful reasoning mechanism; through abduction one can deal with uncertainty, non-monotonicity and, of course, hypothetical reasoning. On the other hand, abductive proof procedures are often based on meta-interpretation, which could lessen the efficiency.

Constraint Logic Programming (CLP), instead, is aimed at solving efficiently combinatorial problems, and employs various algorithms from the areas of artificial intelligence and operation research to deal efficiently with difficult problems.

This paper starts from the observation due to Kowalski et al. [1, 3] that CLP and abduction share an important concept: namely, that of *Conditional Answer*. In both cases, the expected result of a computation consists of two parts: (*i*) a *binding* of the logical variables (as in Logic

* This work is partially funded by the Information Society Technologies programme of the European Commission under the IST-2001-32530 project

Programming); (*ii*) some further *conditions* that should be satisfied in order for the solution to be correct.

A constraint program provides as answer a set of *constraints*, i.e., interpreted atoms that should be proven true in order for the answer to be correct. For example, in CLP(FD), given the query

$$A :: 1..5, B :: 1..10, C :: 5..10, A < B, B < C, A > 4$$

the constraint solver can provide as an answer a binding $A/5$ plus some further constraints $B :: 6..9$, $C :: 7..10$ and $B < C$. The answer is correct provided that the constraints are satisfied.

An abductive program provides a set Δ of abduced atoms; the answer is correct if the set of abduced atoms is true. For example, given that

$$\begin{aligned} grass\_is\_wet &\leftarrow rained\_last\_night. \\ grass\_is\_wet &\leftarrow sprinkler\_was\_on. \\ shoes\_are\_wet &\leftarrow grass\_is\_wet. \end{aligned}$$

a possible answer to the query $? - shoes\_are\_wet$ is *true*, provided that the unknown *rained_last_night* is assumed true as well.

From this observation, we could naturally map abduction through a constraint program, where abduced literals are posted in a constraint store as constraints are in CLP. This could help the development of a very efficient abductive system, that would not need meta-interpretation, and that could exploit the efficient propagation algorithms of CLP.

Abduction and CLP have been integrated from different viewpoints. Kowalski et al. [1] propose a framework where abduction and constraints are treated uniformly, but they are not very focussed on efficiency issues. Kakas et al. [2] [4] implement abduction on top of a CLP system, exploiting constraints to limit the search space. However, their implementations of abduction are still based on meta-interpretation, and the constraint solver is not used to reason upon abduced literals, but on the constrained variables appearing in them.

In this work, we propose two possible implementations of an abductive proof procedure based on CLP: one is more efficient, while the other can be extended for non ground abducibles. The implementations are based on Constraint Handling Rules [5], a rewriting system to build constraint solvers on top of another CLP solver. Both implementations are on top of the Boolean domain, CLP(Bool). In the first one, we build a solver for a new CLP domain, called CLP($\mathcal{A}bd$), in which the *definition* of the solver is based on integrity constraints. In the second, the solver relates abducible literals and integrity constraints both as constraints of the CLP

domain. We extend the second implementation to deal with variables in the abduced literals.

Our work is related to the work of Kowalski et al. [1], where the idea of treating uniformly constraints and abducibles is presented. Abdennadher and Christiansen [6] propose a CHR implementation that is strongly related both to [1] and to the first of the implementations proposed in this paper. The main difference between [6] and the first of our implementations is that we give a CLP flavor of the abductive sort and propose to exploit a boolean solver. Also, we propose a second implementation where the integrity constraints are treated as constraints à la CLP.

2 Notation and Preliminaries

Definition 1. *An* Abductive Logic Program *[7] is a triple* $\langle KB, Ab, IC\rangle$*:*

- KB *is a (normal) logic program, i.e., a set of clauses* $A \leftarrow L_1, \ldots, L_m$*, where* L_i $(i = 1, \ldots, m)$ *are literals and* A *is an atom;*
- Ab *a set of* abducible predicates*,* p*, such that* p *does not occur in the head of any clause of* KB*;*
- IC *is a set of integrity constraints, that is, a set of closed formulae.*

Given an abductive program and a goal G*, an* abductive explanation *for* G *is a set* Δ *(such that* $\Delta \subseteq Ab$*) with a substitution* θ *such that* $KB \cup \Delta$ *is consistent,* $KB \cup \Delta \models \tilde{\forall}(G/\theta)$ *and* $KB \cup \Delta \models IC$*.*

We suppose that each IC has the syntax (where $L_1, \ldots, L_n$ is a conjunction of literals):

$$(\bot) \leftarrow L_1, \ldots, L_n.$$

thus, the previous definition $KB \cup \Delta \models IC$ is equivalent to saying that the literals appearing in an integrity constraint cannot be all true in order for the program (with the Δ) to be consistent.

Definition 2. Constraint Logic Programming (CLP) *[8] is a general framework; its instances are languages CLP(*$\mathcal{X}$*) where* $\mathcal{X}$ *represents the domain of the computation.* $\mathcal{X}$ *is the quadruple* $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T}\rangle$*, where*

- Σ *is a signature*
- $\mathcal{D}$ *is a* Σ*-structure that gives the interpretation of the symbols in* Σ
- $\mathcal{L}$ *is a class of* Σ*-formulas that defines the language of constraints*
- $\mathcal{T}$ *is a* Σ*-theory that defines the logical semantics of constraints.*

A solver solv for the class of constraints $\mathcal{L}$ *is a function that maps each formula into* true*,* false *or* unknown*.*

Definition 3. Constraint Handling Rules *(CHR) [5] are a term-rewriting system that handles constraints in a store. There are three basic rules, called simplification, propagation and simpagation rules.*

A simplification *rule has the following syntax:*

$$[RuleName@]Head[, Head] \Leftrightarrow [Guard|]Body.$$

Declaratively, a rule relates heads and body provided the guard is true. A simplification rule means that the heads are true if and only if the body is true. A propagation *rule, like:*

$$[RuleName@]Head[, Head] \Rightarrow [Guard|]Body.$$

means that the body is true if the heads are true. A simpagation *rule*

$$[RuleName@]Head \setminus Head \Leftrightarrow [Guard|]Body.$$

is a combination of a simplification and propagation rule. The rule $H1 \setminus H2 \Leftrightarrow Body$ *is equivalent to the simplification rule* $H1, H2 \Leftrightarrow Body, H1$*. However, the simpagation rule is more compact to write, more efficient to execute, and has better termination behavior than the corresponding simplification rule. Operationally, when the heads of a simplification rule can be unified with constraints in the store they are removed from the store and Body is executed. When the heads of a propagation rule are unified with constraints in the store, the body is executed. In a simpagation rule, the first head is kept in the store, while the second one is removed.*

3 An abductive solver on top of CLP(Bool)

Abduction can be considered as a CLP sort defined by means of Integrity Constraints. Each abduced literal is mapped to a (CLP) constraint together with a boolean that carries information about the truth of the abduced. If A is a literal, A^{B_A} is true iff $(A \Leftrightarrow B_A)$, i.e., the boolean B_A assumes value *true* iff A is true. For example, A^{false} is equivalent to $\neg A$. In this section, we focus on ground literals; non ground abduced will be considered in Section 4.3.

We first give formal definitions, then we show the implementation together with motivating examples.

3.1 Abduction as a CLP sort: CLP($\mathcal{A}bd$)

The concept of abduction could be considered as a language in the CLP class, in which abduced literals are mapped to (CLP) constraints, and integrity constraints define the solver.

In particular, we define a CLP sort $\mathcal{A}bd$. The class of constraints, $\mathcal{L}_{\mathcal{A}bd}$, is simply the set of abducible predicates, the Σ-structure $\mathcal{D}_{\mathcal{A}bd}$ (the domain of the computation) maps the terms in abducibles into Herbrand interpretations, and the theory $\mathcal{T}_{\mathcal{A}bd}$ contains the integrity constraints. We will suppose that in $\mathcal{T}_{\mathcal{A}bd}$ constraints are idempotent; this will ensure that abducing twice the same term is equivalent to abducing it once.

A solver, $solv_{\mathcal{A}bd}$, on a particular set of abducible predicates, will be defined by the user by means of integrity constraints. The solver will give a failure if the constraint store (which coincides with the set Δ of abduced literals) is inconsistent with the theory $\mathcal{T}_{\mathcal{A}bd}$. For example, if $(\bot \leftarrow a, b) \in \mathcal{T}_{\mathcal{A}bd}$, $solv_{\mathcal{A}bd}(\{a, b\}) = false$ (i.e., if the literals a and b are both abduced, the solver fails). A possible implementation is given in the following.

3.2 Implementation of $solv_{\mathcal{A}bd}$

The implementation of the abductive proof procedure can be based on a boolean solver. To each abduced literal we associate a boolean variable: if the boolean has the logic value *true*, then the literal is *positive*, otherwise it is *negative*. In this way, ICs can be easily handled by the boolean solver. Consider the IC (where a, b and c are abducibles):

$$\leftarrow a, b, \neg c. \tag{1}$$

If the atom a is abduced, a (passive) constraint $abd(a, true)$ is inserted in the constraint store. Now, all the other literals in IC (1) are abduced (if not already in the store), with unknown (variable) truth status; i.e., the constraint $abd(b, B_b)$ and $abd(c, B_c)$ are inserted in the store. The integrity constraint (1) can now be simply turned into a boolean constraint:

$$\neg(true \wedge B_b \wedge \neg B_c) \tag{2}$$

and passed to the boolean solver. In this way, the boolean solver could be able to detect inconsistencies early, avoiding many backtracking steps, and perform powerful propagation, and could infer the truth status of some of the variables. Modern CLP languages include very smart boolean solvers, based on efficient representation techniques of the boolean formulas [9]. For example, suppose to have the following set of integrity constraints:

$$\leftarrow b, c. \qquad \leftarrow \neg c, d. \qquad \leftarrow b, \neg d.$$

where all the literals are abducible. The corresponding boolean formula, in our representation, would be

$$\neg(B_b \wedge B_c) \wedge \neg(\neg B_c \wedge B_d) \wedge (B_b \wedge \neg B_d).$$

The boolean solver clp(B) embedded in SICStus [10] infers immediately that B_b must be false, in order for the ICs to be satisfiable, thus we can immediately abduce $\neg b$. From this viewpoint, most abductive proof procedures are based on a check *a posteriori* of the ICs, and on an early commitment which "labels" an abducible atom as true or false before it is actually needed. By translating ICs into constraints of efficient solvers and commit to choices only when really needed, we may a priori remove inconsistent configurations, and increase the efficiency of the proof.

If the IC also contains non-abducible literals, they can be handled as in the KM proof procedure [11]. E.g, given an Integrity Constraint:

$$\leftarrow a, b, \neg c, p. \tag{3}$$

where p is not abducible, one can impose the disjuction of the boolean constraint (2) and, as an alternative, impose that p fails:

$$\neg(true \wedge B_b \wedge \neg B_c) \vee not\ p.$$

If p is a predicate that does not perform abduction, imposing that it fails is equivalent to negation as failure. In general one may think to (i) either perform a *consistency derivation* [11], that would need meta-interpretation, or (ii) use negation as failure without abduction, i.e., calling the predicate p with the current set Δ. In case (ii), however, one has to perform this check each time a new literal is abduced.

Example 1. Consider the following program (a and b are abducible):

$$q \leftarrow a. \qquad\qquad p \leftarrow b. \qquad\qquad \leftarrow a, p.$$

The derivation for the goal $? - q$ would abduce a; now one can try to falsify p by (i) abducing $\neg b$ or by (ii) invoking p without abducing new literals. In case (ii) one should also ensure that whenever a new literal is abduced, the falsity of p is checked again, otherwise the goal $? - q, b$ would, erroneously, succeed.

We are not committed to one choice or the other. In the following, we will use the symbol *not* as an implementation of one of the two methods.

Integrity Constraints as Rules Since abducibles are constraints in the store, CHR seems a perfect means for implementing integrity constraints. As in [6], integrity constraints could be translated into rules. For example, the IC in Eq. 3 could be translated into a propagation rule:

$$abd(a, true), abd(b, true), abd(c, false) \Rightarrow not(p) \tag{4}$$

i.e., whenever a and b are true and c is false, one should check that p fails. A more effective solution would exploit also the boolean solver. We could rewrite the rule in Eq. 4 as follows:

$$abd(a, B_a), abd(b, B_b), abd(c, B_c) \Rightarrow (\neg(B_a \wedge B_b \wedge \neg B_c); not(p)) \,. \quad (5)$$

where the semicolon stands (as in Prolog) for nondeterministic disjunction and the constraint $\neg(B_a \wedge B_b \wedge \neg B_c)$ is passed to the boolean solver.

In general, given an integrity constraint

$$\leftarrow A_1, \ldots, A_n, P_1, \ldots, P_k$$

where A_i are literals of abducible predicates and P_j are literals of defined predicates, a corresponding propagation rule defining $solv_{\mathcal{A}bd}$ is

$$abd(A_1, B_1), \ldots, abd(A_n, B_n) \Rightarrow (\neg(B_1 \wedge \ldots \wedge B_n); not((P_1, \ldots, P_k))).$$

Of course, we should also ensure that an abducible is not true and false at the same time (idempotence of the theory $\mathcal{T}_{\mathcal{A}bd}$):

$$abd(X, B_1) \setminus abd(X, B_2) \Rightarrow B_1 = B_2$$

With this idea, the check of ICs is activated only after *all* the abducibles in the IC have been abduced (true or false). This can be considered as a *lazy* evaluation of integrity constraints. An *eager* strategy would activate the constraints when *one* of the abducibles in the ICs is in Δ:

$$\begin{array}{l} abd(a, B_a) \Rightarrow abd(b, B_b), abd(c, B_c), (\neg(B_a \wedge B_b \wedge \neg B_c); not(p)) \,. \\ abd(b, B_b) \Rightarrow abd(a, B_a), abd(c, B_c), (\neg(B_a \wedge B_b \wedge \neg B_c); not(p)) \,. \\ abd(c, B_c) \Rightarrow abd(a, B_a), abd(b, B_b), (\neg(B_a \wedge B_b \wedge \neg B_c); not(p)) \,. \end{array} \quad (6)$$

It is worth noting that the translation from the usual notation of Eq. 3 to those in Eq. 5 or Eq. 6 can be performed syntactically, as a preprocessing. Another possibility, that does not need preprocessing, is shown in the next section.

4 Integrity Constraints as CLP Constraints

Another implementation would consider ICs as CLP constraints and the CHR rules would define the general propagation of ICs. In other words, one could write the ICs as constraints in the CLP sense. Thus, the abductive program would have a form similar to the typical constraint program:

```
abductive_program :- impose_integrity_constraints,  search.
```

In our syntax, an integrity constraint contains a list of (possibly abducible) literals. Abducible literals are represented with the same functor $abd/2$, where the first argument is the atom, and the second is a boolean stating if the literal is positive or negative[3].

In our example, the user would write something like:

```
impose_integrity_constraints :-
   ic([abd(a,true), abd(b,true), abd(c,false), p]),
abductive_program :-
   impose_integrity_constraints, abd(a,true), ...
```

for an abductive program with an integrity constraint $\leftarrow a, b, \neg c, p$ that abduces the atom a.

Again, we first formalize the sort, then we propose an implementation of the solver based on CHR, on top of CLP(Bool).

4.1 Abduction as a sort with ICs as constraints: CLP($\mathcal{A}bd2$)

In this implementation, both abducibles and integrity constraints are mapped to constraints à la CLP. We formalize the language as an instance of the CLP framework, that we call CLP($\mathcal{A}bd2$), in which the CLP language contains both abducibles and ICs as constraints. In the quadruple defining the sort $\mathcal{A}bd2$ we have that:

- the class of constraints, $\mathcal{L}_{\mathcal{A}bd2}$, contains both the abducible predicates (indicated with $abd/2$) and the integrity constraints ($ic/1$);
- the domain of the computation, the Σ-structure $\mathcal{D}_{\mathcal{A}bd2}$, maps abducibles and integrity constraints into Herbrand interpretations;
- the theory $\mathcal{T}_{\mathcal{A}bd2}$ contains the basic rule that defines interaction between integrity constraints and abducibles: if all the literals in an integrity constraint are true, inconsistency arises, i.e.:

$$ic([abd(A_1, B_1), \ldots, abd(A_n, B_n), p_1, \ldots, p_k]),$$
$$abd(A_1, B_1), \ldots, abd(A_n, B_n), p_1, \ldots, p_k \rightarrow \bot$$

In the next section we propose a solver, $solv_{\mathcal{A}bd2}$, for the language CLP($\mathcal{A}bd2$) defined through CHR.

[3] Note the abuse of notation: we use the same symbol $abd/2$ to indicate both the *constraint* identifying an abducible and the *terms* indicating abducibles inside an IC.

4.2 Implementation: $solv_{\mathcal{A}bd2}$

The following rules would perform the propagation of ICs in a lazy style:

$$\begin{array}{l} lazy \ @ \ ic(L) \setminus abd(X,B) \Leftrightarrow delete(L, abd(X,B), R) | ic(R). \\ goal \ @ \ ic(L) \Leftrightarrow no\_abducibles(L) | not(and\_call(L)). \end{array} \quad (7)$$

The first rule performs propagation. In order to understand its meaning, let us first consider the following case: $abd(X,B)$ has been abduced and $abd(X,\neg B)$ is part of the IC. In this case, the IC is satisfied: in fact, one of the literals in the body is false. In particular, if the IC contains $abd(X,true)$, then the IC is satisfied if X is abduced false. If the IC contains $abd(X,false)$, this means that the negation of X is in the IC, thus if X is abduced $true$, the IC is satisfied. Viceversa, if $abd(X,B)$ has been abduced and $abd(X,B)$ is in the IC, we must prove that the rest of the (body of the) IC is false. Thus, we remove from the IC $abd(X,B)$.

In this semantics, an empty IC means failure; thus we could have:

$$ic([]) \Leftrightarrow fail$$

However, this rule is redundant, as it is already contained in the second rule of Eq. 7. The second rule, in fact, considers the case in which no abducibles are left in the IC. In this case, we should try to prove that the conjunction of the remaining literals in the body is false: this is performed by the negation of the conjunction of the literals.

We could also decide to have eager propagation (rule *goal* is the same as in Eq. 7):

$$eager \ @ \ ic(L) \setminus abd(X,B) \Leftrightarrow delete(L, abd(X,B), R) | eager\_prop(R).$$

Predicate *eager_prop* is defined as follows:

```
eager_prop(L) :-
 divide_abducibles(L,Abducibles,NonAbducibles),
 (abduce_eager(Abducibles),
  get_booleans(Abducibles,Bools), impose_boolean_nand(Bools)
 ; ic(NonAbducibles)).
```

first of all, we separate the abducibles from the non abducibles in the IC. We try to satisfy the constraint only with the abducibles: we abduce the predicates with a boolean variable as truth value, then we impose a boolean constraint that states that not all the literals can be true. If this attempt fails, then we impose the integrity constraint consisting only of the non abducibles; the second rule of Eq. 7 will take care of the rest.

These implementations work in the propositional case. If we allow for generic literals, even with variables, in the Δ, then we should consider further issues, depicted in the following section.

4.3 CHR and abduction with variables

In the non-propositional case, it is not safe to remove an IC (or change it) when a literal that it contains is abduced. E.g., if we have an IC like:

$$\leftarrow a(X,Y), b(X) \tag{8}$$

and we abduce $a(1,2)$, we can, of course, say that $b(1)$ is false, but we cannot remove the IC. Thus, many of the simpagation rules will be converted into propagation rules.

Moreover, we have to consider that variables in abduced literals are existentially quantified, while in integrity constraints they are universally quantified. Thus, given the IC (8), abducing $b(T)$ means that $\exists T, b(T)$, and propagation of the integrity constraint produces:

$$\exists T, \Delta = \{b(T)\} \ \wedge \ \forall Y[\bot \leftarrow a(T,Y)].$$

Notice that after propagation we have implications with some of the variables existentially quantified, and some universally; the scope of existential variables includes also the set Δ.

We decided to explicitly tag each variable with its quantification. Many proof procedures avoid the explicit quantifications by introducing syntactic restrictions [12].

We recognize two possibilities: we could

1. abduce new literals only when all of the variables are existentially quantified.
2. abduce literals where some of the variables are universally quantified

Again, the first choice is more "lazy", while the second is more "eager" in inferring information to be passed to the boolean solver. In the first idea, one accepts, in the set Δ, only existentially quantified literals; this lessens the information passed to the boolean solver, thus propagation would be less efficient. For example, consider the following IC:

$$\leftarrow a(X), b(X,Y), c(X)$$

where $a/1$, $b/2$ and $c/1$ are abducibles. If one abduces $a(f(A))$, propagation will give that:

$$\exists A, \Delta = \{a(f(A))\}, \forall Y[\bot \leftarrow b(f(A),Y), c(f(A))]$$

Since the term $c(f(A))$ is existentially quantified, one could abduce it (with a variable B_c as truth status), and obtain that

$$\exists A, \Delta = \{a(f(A)), c^{B_c}(f(A))\}, [\neg(true \wedge B_c) \vee \forall Y \bot \leftarrow b(f(A),Y)]$$

i.e., we can pass the condition $\neg(true \wedge B_c)$ to the boolean solver, and in case of backtracking, impose that $\forall Y \bot \leftarrow b(f(A), Y)$.

The second choice would allow for more powerful propagation: since we exploit an efficient boolean solver, abducing as many literals as possible also means inferring as much information as possible for the boolean solver. In our example, we would also abduce the atom $(\forall Y)b(f(A), Y)$, with an unknown truth status B_b; i.e., we reach the following state:

$$\exists A, \forall Y, \Delta = \{a(f(A)), b^{B_b}(f(A), Y), c^{B_c}(f(A))\}, [\neg(true \wedge B_b \wedge B_c)]$$

The implementation of the proof with this second choice becomes much harder; we plan to develop it in the future.

The only propagation rule we use is the following:

```
propagation @ ic(L), abd(X,B) ==>
 pat_delete(L,abd(X,B),Rest,A) |
 copy_term_universal(t(Rest,A),t(R,Y)),
 unify_considering_quantification(abd(X,B),Y,Unify),
 (Unify=true,
   separate_existential_terms(R,ExistentialTerms,Others),
   abduce_and_propagate(ExistentialTerms,Others)
 ; Unify=false).
```

pat_delete finds in the list L a literal A that matches with $abd(X, B)$ and provides also the *Rest* of the list. Note that the predicates in the guards should not instantiate variables in the abduced literals (in Δ); thus predicate *pat_delete* does not perform usual unification, but only pattern-matching. A copy of the integrity constraint is made, with fresh new variables in place of universally quantified variables; existentially quantified variables are not copied. Unification is then performed between the abduced literal and its matching literal in the copy of the IC. Notice that unification is, in general, non-deterministic in presence of existentially quantified variables (if we have two literals $\exists A, p(A)$ $\exists B, p(B)$, one possibility is that $A = B$, the other is that $A \neq B$). If unification succeeds (the boolean $Unify$ is true), the existential terms are divided from the others and abduced, while the others are imposed, upon backtracking, as a new IC (eventually with some existentially quantified variables).

5 Conclusions and Future Work

In this paper we proposed two interpretations of abduction in a CLP setting: one with integrity constraints defining a CLP solver for the domain of abducibles, the other with both abducibles and integrity constraints as

constraints à la CLP. In both proposals, we rely on a CLP(Bool) solver to efficiently propagate integrity constraints.

In future work we would like to embed CLP constraints also on the variables of abduced literals: since many CLP(FD) languages contain *reified* constraints (i.e., constraints whose truth value is a boolean variable), they should integrate smoothly in our framework: we map abducibles into constraints $abd(Atom, Bool)$ that can be considered as a particular instance of reified constraints.

Also, we would like to extend the framework to accept universally quantified variables in abduced literals: this could exploit more deeply the propagation performed by the CLP(Bool) solver.

A very important issue will be experimentation and comparison with other available implementations of abductive proof procedures.

References

1. Kowalski, R., Toni, F., Wetzel, G.: Executing suspended logic programs. Fundamenta Informaticae **34** (1998) 203–224
2. Kakas, A.C., Michael, A., Mourlas, C.: ACLP: Abductive Constraint Logic Programming. Journal of Logic Programming **44** (2000) 129–177
3. Wetzel, G.: A unifying framework for abductive and constraint logic programming. In Franois Bry, Burkhard Freitag, D.S., ed.: Twelfth Workshop Logic Programming, WLP, Mnchen (1997)
4. Kakas, A.C., Nuffelen, B.V., Denecker, M.: $\mathcal{A}$-System: Problem solving through abduction. In Nebel, B., ed.: Proceedings of IJCAI 2001, Seattle, Washington, USA, Morgan Kaufmann (2001) 591–596
5. Frühwirth, T.: Constraint Handling Rules. In Podelski, A., ed.: Constraint Programming: Basics and Trends. Number 910 in Lecture Notes in Computer Science. Springer Verlag (1995) 90–107
6. Abdennadher, S., Christiansen, H.: An experimental CLP platform for integrity constraints and abduction. In Larsen, H., J.Kacprzyk, Zadrozny, S., Andreasen, T., Christiansen, H., eds.: FQAS, Flexible Query Answering Systems. LNCS, Warsaw, Poland, Springer-Verlag (2000) 141–152
7. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. Handbook of Logic in AI and Logic Programming **5** (1998) 235–324
8. Jaffar, J., Maher, M., Marriott, K., Stuckey, P.: The semantics of constraint logic programs. Journal of Logic Programming **37(1-3)** (1998) 1–46
9. Bryant, R.: Graph-based logarithms for boolean function manipulation. IEEE Transactions on Computers (1996)
10. Carlsson, M., Widén, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., Sjöland, T.: SICStus prolog user's manual. Technical Report T91:15, Swedish Institute of Computer Science (1995)
11. Kakas, A.C., Mancarella, P.: Abductive logic programming. In Marek, V.W., Nerode, A., Pedreschi, D., Subrahmanian, V.S., eds.: Proceedings of the Workshop Logic Programming and Non-Monotonic Logic, Austin, TX (1990) 49–61
12. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming **33** (1997) 151–165