Solving Deductive Planning Problems Using Program Analysis and Transformation

D.A. de Waal* and M. Thielscher

FG Intellektik, FB Informatik, Technische Hochschule Darmstadt, Germany
{andre,mit}@intellektik.informatik.th-darmstadt.de

Abstract. Two general, problematic aspects of deductive planning, namely, detecting unsolvable planning problems and solving a certain kind of postdiction problem, are investigated. The work is based on a resource oriented approach to reasoning about actions and change using a logic programming paradigm. We show that ordinary resolution methods are insufficient for solving these problems and propose program analysis and transformation as a more promising and successful way to solve them.

1 Introduction

Understanding and modeling the ability of humans to reason about actions, change, and causality is one of the key issues in Artificial Intelligence and Cognitive Science. Since logic appears to play a fundamental rôle for intelligent behavior, many deductive methods for reasoning about change were developed and thoroughly investigated. It became apparent that a straightforward use of classical logic lacks the essential property that facts describing a world state may change in the course of time. To overcome this problem, the truth value of a particular fact (called *fluent* due to its dynamic nature) has to be associated with a particular state. This solution brings along the famous technical frame problem which captures the difficulty of expressing that the truth values of facts not affected by some action are not changed by the execution of this action.

Many deductive methods for reasoning about change are based on the ideas underlying the situation calculus [20, 21]. Yet in recent years new deductive approaches have been developed which enable us to model situations, actions, and causality without the need to employ extra axioms due to the frame problem [1, 19, 14]. Instead of representing the atomic facts used to describe situations as fluents, these approaches take the facts as *resources*. Resources do not hold forever—they are consumed and produced by actions. Consequently, resources which are not affected by an action remain as they are and need not be updated.

In particular, the approach developed in [14] is based on logic programming with an associated equational theory. Although previous results illustrate the expressiveness of the equational logic programming approach (ELP, for short)

* This author was supported by HCM Project: Compulog Group—Cooperation Group in Computational Logic under contract no. ERBCHBGCT930365.

in principle, the applicability of concrete proof strategies such as Prolog has not yet been assessed. A major difficulty is caused by the use of an underlying equational theory, which requires a non-standard unification procedure in conjunction with an extended resolution principle called *SLDE-resolution* [8, 13]. In this paper, we follow an alternative direction and investigate a particular program where a unification algorithm for our special equational theory is integrated by means of additional program clauses while otherwise standard unification is used.

On the basis of this logic program, we illustrate two general classes of problems which deserve a successful treatment yet turn out to be unsolvable using ordinary resolution methods. First, non-terminating sequences of actions usually prevent us from deciding unsolvable planning problems. More precisely, a planning problem consists of an underlying set of action descriptions, a collection of initially available resources, and a goal specification (consisting of the resources we strive for). If no action sequence can be found that transforms the initial situation into a situation containing the goal, such a planning problem is called unsolvable; detecting this, however, is problematic as soon as infinite sequences of actions have to be considered. Second, we investigate a particular kind of so-called *postdiction* problem where we try to determine which resources can possibly be used to obtain a certain goal situation. Although our deductive planning approach can successfully model this problem in principle, it is not practical in any real implementation as there are possibly an infinite number of combinations of resources that may lead to one specific resource being produced. Moreover, since our logic program, and especially the encoding of the special unification algorithm, was not designed to reason backwards, it loops even in case of finite action sequences.

In this paper we propose elegant solutions to these two problems, detecting unsolvable planning problems and postdiction, based on logic program analysis and transformation. The solutions are based on the approach by de Waal and Gallagher [3, 2]. In their approach a proof procedure for some logic is specialized with respect to a specific theorem proving problem. The result of the specialization process is an optimized proof procedure that can only prove formulas in the given theorem proving problem. One of the effects of the specialization may be that one or more infinitely failed deductions may be detected and are then deleted. It is this property that makes the developed specialization process so attractive for optimizing difficult planning problems.

In the context of this paper, the particular proof procedure and theorem proving problem is the logic program to model actions and change along with a set of action descriptions defining the various feasible actions, their conditions and their effects. The aim is to detect unsolvable planning problems and derive finite descriptions of a possible infinite number of resources. However, we have found that the procedure suggested in [2] is not precise enough for the optimization of this logic program and needs improvement. The first problem sketched above may be solved by refining the specialization procedures developed in [3, 2]. Nonetheless it is not feasible to give an exact solution to the second problem as we pointed out earlier. An approximation of the resources needed is therefore computed.

The layout of the rest of the paper is as follows. In the next section we introduce deductive planning problems. Furthermore, we introduce two exemplary classes of problems we aim to solve with our improved analysis and transformation techniques. In Section 3 we give an improved specialization procedure that better exploits the approximation results than was proposed in [2]. In Section 4, this refined technique is applied to the exemplary problems discussed in Section 2. This paper concludes with a comparison with related work and a short discussion of how to further improve the proposed specialization method.

2 Deductive Planning Problems

2.1 The Equational Logic Programming Approach

The completely reified representation of situations is the distinguishing feature of the ELP-based approach [14]. To this end, the *resources* being available in a situation are treated as terms and are connected using a special binary function symbol, denoted by $\circ$ and written in infix notation. As an example, consider the term[2]

$$d \circ q \circ f \circ f \circ dm \tag{1}$$

which shall represent a situation where we possess a dollar (d), a quarter (q), two fünfziger (f—fifty pfennige), and one deutschmark (dm). Intuitively, the order of the various resources occurring in a situation should be irrelevant, which is why we employ a particular equational theory, viz

$$(X \circ Y) \circ Z =_{AC1} X \circ (Y \circ Z) \tag{2}$$

$$X \circ Y =_{AC1} Y \circ X \tag{3}$$

$$X \circ \emptyset =_{AC1} X \tag{4}$$

where $\emptyset$ is a special constant denoting a unit element for $\circ$. This equational theory, written AC1, will be used as the underlying theory of our equational logic program modeling actions and change.

Based on this representation, actions are defined in a STRIPS-like fashion [4, 16] by stating a collection of resources to be removed from along with a collection of resources to be added to the situation at hand. Such an action is applicable if all resources to be removed are contained in the current situation. For instance, a machine that changes two fünfziger into a deutschmark can be specified by an action with condition $f \circ f$ and effect dm. This action is applicable in (1) since two resources of type f are included; the result of applying this action is

[2] Throughout this paper, we use a PROLOG-like syntax, i.e., constants and predicates are in lower cases whereas variables are denoted by upper case letters. Moreover, free variables are implicitly assumed to be universally quantified and, as usual, the term $[h \mid t]$ denotes a list with head h and tail t.

computed by removing two fünfziger from and adding a deutschmark to (1), i.e., $dm \circ d \circ q \circ dm$ which is exactly the expected outcome.

In what follows, we describe an equational logic program that formalizes the above concepts. First of all, actions are described by means of unit clauses using the ternary predicate $action(c, a, e)$ where c and e are the condition and effect, respectively, and a is a symbol denoting the name of the action. E.g., our exemplary change action is encoded as

$$action(f \circ f, gdm, dm) \leftarrow \tag{5}$$

where gdm is meant as an abbreviation of *get-deutschmark*.

Next, we have to find a formalization of testing whether the resources contained in a term c, denoting the condition of some action, are each contained in a term s, denoting the situation at hand. This can be achieved by stating an AC1-unification problem of the form $c \circ Z =_{AC1} s$, where Z is a new variable. It is easy to see that if this problem is solvable, i.e., if a substitution θ can be found such that $(c \circ Z)\theta =_{AC1} s\theta$, then all subterms occurring in c are also contained in s. For instance, the unification problem $f \circ f \circ Z =_{AC1} \; d \circ q \circ f \circ f \circ dm$ is solvable by taking $\theta = \{Z \mapsto d \circ q \circ dm\}$. Moreover, a side effect of solving such a unification problem is that the variable Z becomes bound to exactly those resources which are obtained by removing the elements in c from s. Hence, to obtain the resulting situation, we finally have to add the effect e of the action under consideration to $Z\theta$, i.e., the term $e \circ Z\theta$ represents the intended outcome. The reader should note that no additional axioms are needed here for solving the technical frame problem since all resources which are not affected by performing an action are automatically available in the resulting situation (e.g., the resources $d \circ q \circ dm$ in our example above). By means of logic program clauses, the application of actions is encoded using the ternary predicate $causes(i, p, g)$ where i and g are situation terms (called *initial* and *goal* situation, respectively) and p (called *plan*) is a sequence of action names:

$$\begin{array}{ll} causes(I, [\,], I) \leftarrow & \\ causes(I, [A|P], G) \leftarrow & action(C, A, E), \\ & C \circ Z =_{AC1} I, \\ & causes(E \circ Z, P, G) \end{array} \tag{6}$$

In words, the empty sequence of actions, $[\,]$, changes nothing while an action a followed by a sequence of actions p applied to i yields g if a is applied as described above and, afterwards, applying p to the resulting situation, $(E \circ Z)\theta$, yields g[3].

A major difficulty as regards practical implementations of this approach is caused by the underlying equational theory, which is assumed to be built into the unification procedure. In [10] we argued that the AC1-unification problems that

[3] For the sake of an appropriate treatment of equality subgoals, we implicitly add the clause $X =_{AC1} X$ encoding *reflexivity*. Note that each SLDE-step is intended to be performed with respect to our equational theory, AC1.

occur when computing with our program are of a special kind, and we proposed a unification algorithm designed for these particular cases. In the rest of this paper, we investigate a standard logic program where this algorithm is modeled by means of additional program clauses rather than by means of an extended unification procedure. To this end, terms using our special connection function, $\circ$, are represented as lists containing the available resources. For instance, (1) is encoded as $[d, q, f, f, dm]$. Furthermore, we introduce a new predicate $ac1\_match$ to model AC1-matching problems of the form $s \circ V =_{AC1} t$ where t is variable-free while s might contain variables but not on the level of the binary function $\circ$:

$$
\begin{array}{l}
causes(I, [\,], I) \;\leftarrow \\
causes(I, [A|P], G) \;\leftarrow action(C, A, E), ac1\_match(C, I, Z), \\
\qquad\qquad\qquad\qquad\quad append(E, Z, S), causes(S, P, G) \\
\\
ac1\_match(S, T, Z) \leftarrow \; mult\_subset(S, T, Z) \\
\\
mult\_subset([\,], T, T) \;\leftarrow \\
mult\_subset([E|S], T, R) \;\leftarrow mult\_minus(T, E, T2), \\
\qquad\qquad\qquad\qquad\qquad mult\_subset(S, T2, R) \\
\\
mult\_minus([E|R], E, R) \;\leftarrow \\
mult\_minus([E1|R1], E, [E1|R2]) \;\leftarrow\; mult\_minus(R1, E, R2)
\end{array}
\tag{7}
$$

In words, $ac1\_match(s, t, z)$ shall be true iff s represents a multiset[4] of resources that are all contained in t; furthermore, z contains all resources occurring in t but not in s. The definition of the corresponding predicate $mult\_subset$ is based on a predicate named $mult\_minus(s, e, t)$ with the intended meaning that removing an element e of the multiset corresponding to s yields a multiset corresponding to t. Finally, we need the standard *append* predicate to model adding the effect of an action to the remaining resources after having removed the condition.

2.2 Unsolvable Planning Problems

In this and the following subsection, we use a combined change/vending machine as the exemplary action scenario. We have already considered the action of changing two fünfziger into a deutschmark; furthermore, the machine shall change a deutschmark into two fünfziger (action gf) and also a dollar into four quarters (action gq) and vice versa (action gd); finally, two fünfziger are the price for a can of lemonade (l; action gl). To summarize, the following clauses specify

[4] The reader should observe that the axioms (2),(3) and (4) essentially model the datastructure multiset.

our exemplary domain:

$$\begin{array}{l} action([dm], gf, [f,f]) \leftarrow \\ action([f,f], gdm, [dm]) \leftarrow \\ action([d], gq, [q,q,q,q]) \leftarrow \\ action([q,q,q,q], gd, [d]) \leftarrow \\ action([f,f], gl, [l]) \leftarrow \end{array} \quad (8)$$

Now, consider the following query, which is used to ask for a plan whose execution yields a can of lemonade given a dollar plus a quarter:

$$\leftarrow \; causes([q,d], P, G), \; ac1\_match([l], G, Z). \quad (9)$$

If this query succeeds then the answer substitution for P is a sequence of actions transforming the initially available collection of resources, $[q,d]$, into a situation G which includes a can of lemonade (and possibly other resources, bound to Z). Note that this way of formalizing a planning problem enables us to specify goal situations only partially.

It seems obvious that (9) cannot succeed with respect to the program depicted in (7) given the action descriptions (8) since we need some unaccessible German money to buy lemonade. Hence, our exemplary planning problem is unsolvable. Our logic program, however, loops when faced with this query since it computes alternate changes of a dollar into four quarters and back into a dollar forever. Thus, simply using SLD-resolution does not suffice to detect this kind of insolubility. Correspondingly, there is no finite failure SLDE-tree for the query

$$\leftarrow causes(q \circ d, P, l \circ Z) \quad (10)$$

with respect to the equational logic program depicted in (6) given a collection of action descriptions corresponding to (8)[5].

One might argue that a loop checking mechanism, detecting identical or subsumed goals in the same branch of the search tree, solves this problem. This is not true as we now illustrate. Consider the changing machine being partly out of order in so far as it changes a dollar into four quarters as before, but now just three quarters into a dollar. Again, it is impossible to find a refutation for our query (9). Yet we can use the machine to produce more and more resources by alternately changing the dollar into four quarters and using three quarters to reproduce the dollar. No ordinary loop checking is applicable here because no two subgoals match during the infinite derivation.

In Section 4, we show how our program analysis and transformation techniques provide a more general way of tackling the insolubility problem in planning.

---

[5] The third argument in (10), $l \circ Z$, encodes what is expressed by the second literal in (9), namely the fact that the goal situation might contain other resources aside from the required lemonade.

2.3 The Postcondition Problem in Deductive Planning

Apart from solving temporal projection (prediction) and planning problems, the ELP based approach is also suitable for a certain kind of postdiction problems, as has been argued in [15]. Postdiction means given a goal situation, what can be deduced about the initial situation, i.e., which resources are needed to obtain a specific goal. For example, suppose we want to buy a can of lemonade, what do we need to achieve this goal? To answer this question, the query $\leftarrow\ causes(I, P, l)$ can be executed with respect to the equational logic program depicted in (6) and a set of action descriptions corresponding to (8). The simplest answer to our question is that already possessing the lemonade clearly is sufficient to obtain it; but two fünfziger as well as a deutschmark too can be used to achieve the goal.

Although the set of initial situations is limited, there is an infinite number of ways generating the resulting situation, l. Due to the infiniteness of the corresponding SLDE-tree, it seems difficult to infer that the resources needed to obtain a can of lemonade are a can of lemonade itself, fünfziger, or deutschmarks, whereas dollars and quarters are needless. Even worse, if we run our logic program (7) given the following query $\leftarrow\ causes(I, P, [l])$ then we first obtain the answer $I = [l], P = []$ as intended but then the program loops without providing us with additional solutions.

3 Specialization

Logic program transformation and analysis provide a wide variety of techniques that can be used for program specialization. These techniques include for instance: partial evaluation, type checking, mode analysis and termination analysis. However, it was realized in [3] that a combination of analysis and transformation techniques provides the best specialization potential. Such a combination based on partial evaluation and regular approximation was then further developed in [3, 2]. They developed a problem specific optimization technique: a proof procedure for some logic is specialized with respect to a specific theorem proving problem. The theorem proving problem is normally a set of axioms and hypotheses of some theory and includes a specific formula that may or may not be a theorem of the given theory.

The partial evaluation step creates, amongst other specializations, renamed versions of definitions according to some criteria (e.g. a specialized version of each inference rule in the proof procedure is created with respect to each predicate symbol appearing in the axioms and hypotheses). The approximation step then computes a safe approximation of the partially evaluated program. As a last step, a simple decision procedure is used to delete clauses in the partially evaluated program based of information contained in the regular approximation. The result of the specialization process is an optimized proof procedure that only proves theorems (or disproves non-theorems) in the given theory. Alternatively, the result may be a table of analysis information about the behavior of the proof procedure on this theorem proving problem.

Two criticisms against these techniques are: they are too problem specific and they do not use all the derived analysis information effectively. The first point criticizes the use of a specific formula (theorem or non-theorem) as a goal in the analysis. The method in [2] computes a new approximation for each query we are interested in. If the analysis could be done generally just with respect to a set of axioms and hypotheses, the analysis information will hold for any theorem or non-theorem we wish to analyze with respect to. The second point criticizes the use of the decision procedure used in [2]:

> "Does a definition for some predicate $p(\ldots)$ exist in P (the source program), but not in A (the approximation of P)? If the answer is yes, all clauses containing positive occurrences of $p(\ldots)$ in the body of a clause in P are useless and can be deleted."

This procedure ignores all of the information contained within the approximation definitions (it just tests for the existence of an approximation).

Our aim is to extend the developed techniques taking the above criticisms into account. We will therefore try to keep the goal or query as general as possible so that the analysis does not need to be redone for every query we wish to investigate. Furthermore, in addition to using the above decision procedure, we will make use of information in the approximation for predicates that do not get deleted using the above criterion. In the next section we develop such a procedure.

We assume the reader is familiar with the basic concepts used in logic programming [17]. A logic meta-programming paradigm as defined by Lloyd [12] is used to state our proof procedure and theorem proving problem in. However, type definitions are not given as our specialization method does not depend on them. A partial evaluator and a regular approximation procedure capable of specializing and approximating pure logic programs are also assumed. Complete descriptions of one such partial evaluator and regular approximation procedure can be found in [5, 7].

3.1 An Improved Decision Procedure

In this section we show how approximate information derived through a regular approximation may be used to achieve useful specializations. The class of Regular Unary Logic Programs was defined by Yardeni and Shapiro [24]. It is attractive to represent approximations of programs as Regular Unary Logic (RUL) Programs, as regular languages have a number of decidable properties and can conveniently be analyzed and manipulated for the use in program specialization. Due to lack of space we refer the interested reader to [7] for definitions of canonical regular unary clause, canonical regular unary logic program and regular definition of predicates.

A RUL program can now be obtained from a regular definition of predicates by replacing each clause $p_i(x_{1^i}, \ldots, x_{n^i}) \leftarrow B$ by a clause

$$approx(p_i(x_{1^i}, \ldots, x_{n^i})) \leftarrow B$$

where *approx* is a unique predicate symbol not used elsewhere in the RUL program. In this case the functor p_i denotes the predicate p_i. The predicate $any(X)$ denotes any term in the Herbrand Universe of the program. A regular safe approximation can now be defined in terms of a regular definition of predicates.

Definition 1. regular safe approximation
Let P be a definite program and A a regular definition of predicates in P. Then A is a **regular safe approximation** of P if the least Herbrand model of P is contained in the least Herbrand model of A.

This definition states that all logical consequences of P are contained in A. We now define a useless clause, that is a clause that never contributes to any solution.

Definition 2. useless clause with respect to a computation
Let P be a definite program, G a definite goal, R a safe computation rule and $C \in P$ be a clause. Let T be an SLD-tree of $P \cup \{G\}$ via R. Then C is useless with respect to T if C is not used in any refutations in T.

The above definition is restricted to definite programs as this simplifies the presentation (see [2] for a definition regarding normal programs) and as definite logic programs are sufficient for representing our equational logic programs.

Given a RUL program A that is a regular safe approximation (called a safe approximation from now on) of a program P, we want to use the information present in A to further optimize program P, that is to detect and delete more useless clauses.

The regular approximation system described in [7] contains a decision procedure for regular languages that can be used to detect useless clauses. However, if this system is not used to compute the approximation, the user has to implement such a procedure. The following condition from [2, 3] is adequate for detecting useless clauses.

> If $A \cup \{G\}$ has a finitely failed SLD-tree then $P \cup \{G\}$ has no SLD-refutation.

A procedure implementing this condition is given in Figure 1. Note that only a subset of possibly failed SLD-trees are detected. The result of the given transformation is a specialized version of P with zero or more clauses deleted. Although the procedure as stated may be inefficient, it can be efficiently implemented as we can have the approximation output its result in the format required by *Step 4*. The original decision procedure will then still be valid and we only need to check *Step 4* which can be done efficiently using any of the currently available logic programming language implementations.

The following theorem states the result of the above specialization precisely.

Theorem 3. preservation of all finite computations
Let P be a definite logic program, $\leftarrow G$ a definite goal and A a regular approximation of P with respect to $\leftarrow G$. Let P' be the result of applying the transformation given in Figure 1 to P.

Given a definite logic program P, a definite goal $\leftarrow G$ and a regular approximation A of P with respect to the goal $\leftarrow G$, a procedure for deciding if a clause $p \in P$ is useless with respect to the goal $\leftarrow G$ is:

1. **Identify the arguments x_i, $(1 \leq i \leq n)$ in every approximation definition $approx(p_j(x_1, \ldots, x_n))$ that is approximated by only a finite number of facts.**
2. **Delete the approximation definitions for all other arguments that were not identified in *Step 1* (only a unique variable in each such argument position will remain).**
3. **Unfold with respect to the definitions of arguments identified in *Step 1* (this gives us a finite number of facts $approx(p_j(\ldots, x_i, \ldots))$, with zero or more arguments x_i instantiated to a ground term).**
4. **Delete all clauses in P that have a literal that can not possibly unify with at least one literal $p_j(x_1, \ldots, x_n)$ contained in an approximation definition $approx(p_j(\ldots))$.**

Fig. 1. Specialization Procedure Exploiting Approximation Information

1. If $P \cup \{G\}$ has an SLD-refutation with computed answer θ, then $P' \cup \{G\}$ has an SLD-refutation with computed answer θ.
2. If $P \cup \{G\}$ has a finitely failed SLD-tree then $P' \cup \{G\}$ has a finitely failed SLD-tree.

Proof.
The proof is similar to that given in [3].

Our experiments with equational logic programs (of which some are given in the following section) showed that the improved decision procedure strikes a good balance between precision and efficiency (which should be one of the aims of every specialization/approximation system).

3.2 Interpreting Analysis Results

In the previous section we used part of the derived approximation to further specialize our program. However, there are many cases in which we are only interested in getting a finite description of the success set of a program and not in individual answers. It is also impractical to try to collect an infinite number of solutions by any means other than by approximation. Furthermore, we might have procedures designed to be used only with some arguments instantiated and others uninstantiated. Changing the mode of an argument will most certainly lead to nonterminating behavior of our program. In such cases an approximation tool may be very useful as it will in finite time give us a finite description of the success set of a program. We argue that a regular approximation is a useful description that may in many cases also be very informative.

One of the most useful properties of the regular approximation derived by procedures such as [7, 11], is that the concrete and abstract domains (see [18] for further details) share the same constants and function symbols (except for the variable X in $any(X)$ in the approximation representing any term in the Herbrand universe of a program). A direct interpretation of the information given in the approximation is therefore possible without referring to abstraction and concretization functions as is usually the case in abstract interpretation. A constant a occurring in our approximation indicates that this constant may possibly occur in one or more solutions of the source program. The same also holds for any function symbol f. However, the number of solutions containing these constants and functions can not be deduced from the approximation.

In the next section we give a planning example where just such an approximation allows us to infer very useful results not possible with any other method known to us.

4 Solving Deductive Planning Problems

Two example problems, all related to the lemonade dispensing machine described in the previous section are specialized and analyzed in this section. Our aim with the first problem is to illustrate how an unsolvable planning problem may be detected. With the second problem, we illustrate how an approximate solution to a postdiction problem may be computed.

The five action descriptions in Program 1 together with the meta-program described in Section 2.1 describe a lemonade dispensing machine. The resources that can be consumed and produced are given by the actions stated by the five clauses.

$$
\begin{array}{ll}
action([d], gc, [q,q,q,q]) \leftarrow & \%\ \textit{change dollar into 4 quarters} \\
action([q,q,q], gd, [d]) \leftarrow & \%\ \textit{change 3 quarters into dollar} \\
action([dm], gf, [f,f]) \leftarrow & \%\ \textit{change deutschmark into 2 fünfziger} \\
action([f,f], gdm, [dm]) \leftarrow & \%\ \textit{change 2 fünfziger into deutschmark} \\
action([f,f], gl, [l]) \leftarrow & \%\ \textit{change 2 fünfziger into a lemonade}
\end{array}
$$

Program 1: Action Descriptions of a Lemonade Dispensing Machine

As a first problem, we want to get **ONLY** a can of lemonade from the machine using a dollar and a quarter. This can be expressed by the following query $\leftarrow causes([d,q], Plan, [l])$. Specialization of the above program with the technique described in [3, 2] gives the specialized program in Program 2. Note that we now only approximate Program 2 with respect to the query $\leftarrow causes(Resources, Plan, [l])$ as we will have another instance of this query in our second example and want to keep the specialization as general as possible (we therefore will not need to approximate again for our second example). This is in keeping with our aim stated at the beginning of the previous section to keep the query we specialize with respect to as general as possible.

$$
\begin{array}{l}
causes(X1, [\,], X1) \leftarrow \\
causes(X1, [gc|X2], X3) \leftarrow mult\_minus\_1(X1, d, X4), \\
\quad\quad causes([q, q, q, q|X4], X2, X3) \\
causes(X1, [gd|X2], X3) \leftarrow mult\_minus\_1(X1, q, X4), \\
\quad\quad mult\_minus\_1(X4, q, X5), mult\_minus\_1(X5, q, X6), \\
\quad\quad causes([d|X6], X2, X3) \\
causes(X1, [gf|X2], X3) \leftarrow mult\_minus\_1(X1, dm, X4), \\
\quad\quad causes([f, f|X4], X2, X3) \\
causes(X1, [gdm|X2], X3) \leftarrow mult\_minus\_1(X1, f, X4), \\
\quad\quad mult\_minus\_1(X4, f, X5), causes([dm|X5], X2, X3) \\
causes(X1, [gl|X2], X3) \leftarrow mult\_minus\_1(X1, f, X4), \\
\quad\quad mult\_minus\_1(X4, f, X5), causes\_1(X5, X2, X3)
\end{array}
$$

$$
\begin{array}{l}
mult\_minus\_1([X1|X2], X1, X2) \leftarrow \\
mult\_minus\_1([X1|X2], X3, [X1|X4]) \leftarrow mult\_minus\_1(X2, X3, X4)
\end{array}
$$

$$
\begin{array}{l}
causes\_1(X1, [\,], [l|X1]) \leftarrow \\
causes\_1(X1, [gc|X2], X3) \leftarrow mult\_minus\_1(X1, d, X4), \\
\quad\quad causes([q, q, q, q, l|X4], X2, X3) \\
causes\_1(X1, [gd|X2], X3) \leftarrow mult\_minus\_1(X1, q, X4), \\
\quad\quad mult\_minus\_1(X4, q, X5), mult\_minus\_1(X5, q, X6), \\
\quad\quad causes([d, l|X6], X2, X3) \\
causes\_1(X1, [gf|X2], X3) \leftarrow mult\_minus\_1(X1, dm, X4), \\
\quad\quad causes([f, f, l|X4], X2, X3) \\
causes\_1(X1, [gdm|X2], X3) \leftarrow mult\_minus\_1(X1, f, X4), \\
\quad\quad mult\_minus\_1(X4, f, X5), causes([dm, l|X5], X2, X3) \\
causes\_1(X1, [gl|X2], X3) \leftarrow mult\_minus\_1(X1, f, X4), \\
\quad\quad mult\_minus\_1(X4, f, X5), causes\_1([l|X5], X2, X3)
\end{array}
$$

Program 2: Program Specialized with respect to $\leftarrow causes(Res, Plan, [l])$

A non-empty approximation is computed (no clauses may be deleted) and the query $\leftarrow causes([d, q], Plan, [l])$ still fails to terminate and we are unable to detect that we will never be able to obtain a can of lemonade from the machine. However, by incorporating the refinement of the improved decision procedure into Program 2, we get a finitely failed computation. The transformed approximation as described in the previous section taking part in the specialization is given in Program 3.

$$
\begin{array}{l}
approx(causes(X1, X2, X3)) \leftarrow \\
approx(mult\_minus\_1(X1, dm, X2)) \leftarrow \\
approx(mult\_minus\_1(X1, f, X2)) \leftarrow \\
approx(causes\_1\_ans([\,], [\,], X1)) \leftarrow
\end{array}
$$

Program 3: Approximation Information Derived from Program 2

The program after further specialization is given in Program 4. Six clauses could be deleted.

$causes(X1, [\,], X1) \leftarrow$
$causes(X1, [gf|X2], X3) \leftarrow mult\_minus\_1(X1, dm, X4),$
$\quad causes([f, f|X4], X2, X3)$
$causes(X1, [gl|X2], X3) \leftarrow mult\_minus\_1(X1, f, X4),$
$\quad mult\_minus\_1(X4, f, X5), causes\_1(X5, X2, X3)$

$mult\_minus\_1([X1|X2], X1, X2) \leftarrow$
$mult\_minus\_1([X1|X2], X3, [X1|X4]) \leftarrow mult\_minus\_1(X2, X3, X4)$

$causes\_1(X1, [\,], [l|X1]) \leftarrow$

Program 4: Program After Further Specialization

The query $\leftarrow causes([d, q], Plan, [l])$ now fails finitely. We have therefore proved that it is impossible to get only a can of lemonade with a dollar and a quarter.

As a second problem we want to deduce what resources are needed to get lemonade. This can be expressed by the query $\leftarrow causes(Resources, Plan, [l])$. Running this query using our original program ((7) with Program 1) only tells us that if we start with a can of lemonade, we have achieved our goal and then the program goes into an infinitely failed deduction. One of the reasons for this unsatisfactory result is that the procedure in (7) was designed to run "forward" and not "backward" as we are trying to do in this example. If the query was stated slightly more generally in that we only require a can of lemonade to be included in the result of the deduction (not to be the only result), there may also be an infinite number of combinations of resources that may lead to this goal situation. Obviously, it is impossible to run the procedure and an approximation of the resources is the best answer we can give.

Applying our improved specialization method to this query yields the result that a combination of deutschmarks and fünfziger (we do not know how many of each) are needed to get a can of lemonade. Part of the approximation result is given in Program 5.

$approx(causes(X1, X2, X3) \leftarrow t1(X1), t5(X2), t7(X3)$
$t1([X1|X2]) \leftarrow t2(X1), t3(X2)$
$t5([\,]) \leftarrow$
$t5([X1|X2]) \leftarrow t6(X1), t5(X2)$
$t7([X1|X2]) \leftarrow t8(X1), t9(X2)$
$t3([\,]) \leftarrow$
$t3([X1|X2]) \leftarrow t4(X1), t3(X2)$
$t8(l) \leftarrow$
$t9([\,]) \leftarrow$
$t6(gl) \leftarrow \quad t6(gf) \leftarrow \quad t6(gdm) \leftarrow$
$t2(l) \leftarrow \quad t2(f) \leftarrow \quad t2(dm) \leftarrow$
$t4(dm) \leftarrow \quad t4(f) \leftarrow$

Program 5: Approximation of Resources

We detected that dollars and quarters play no role in getting a can of lemonade. This is a satisfactory result with enough precision to be useful. This indicates that there are some redundant states in our machine that can never lead to a successful purchase of only a can of lemonade.

5 Discussion

The specialization procedure proposed in Section 3.1 bears some resemblance to the *v-reduction* rule for the connection graph proof procedure proposed by Munch in [22]. This rule considers sets of constants which certain clause variables may be instantiated to due to resolution. Links in the connection graph may be deleted if it is found that the *value_sets* of two corresponding arguments in two connected literals have no elements in common. These *value_sets* can be regarded as an approximation of the possible values that an argument position may take. Information inside recursive structures (such as arguments to functions) are also ignored similarly to our approach where we use information represented by a finite number of facts.

The work on approximation in automated theorem proving by for instance Giunchiglia and Walsh [9] and Plaisted [23] may also be applicable to the optimization of planning problems. However, their viewpoint is that it is the object theory that needs changing and not the logic (they would therefore not approximate the proof procedure for solving planning problems, but concentrate on approximating the action descriptions). Our approximation does not make this distinction. Furthermore, our general approximation procedure, namely a regular approximation, is fixed. This makes our method easier to adapt to new domains as we do not have to develop a new approximation for each new domain we want to investigate.

The proposed method may obviously be further improved by also using information represented by parts of the approximation other than only that represented by facts. However, the decision procedure will then be more complicated and we may then not rely any more on only SLD-resolution to test failure in the approximation. When more complicated action descriptions containing variables in resources are analyzed, we may need to take advantage of all the useful specializations. However, the partial evaluation step may assist in overcoming some of the problems posed by more complicated resource descriptions as it can factor out common structure at argument level (see [6, 5] for further details).

References

1. W. Bibel. A Deductive Solution for Plan Generation. *New Generation Computing*, 4:115–132, 1986.
2. D.A. de Waal. *Analysis and Transformation of Proof Procedures.* PhD thesis, University of Bristol, October 1994.
3. D.A. de Waal and J.P. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221. Springer-Verlag, 1994.

4. R. Fikes and N. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence Journal*, 2:189–208, 1971.
5. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
6. J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic, Leuven, Belgium*, 1990.
7. J. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.
8. J. H. Gallier and S. Raatz. Extending SLD-Resolution to Equational Horn Clauses Using E-Unification. *Journal of Logic Programming*, 6:3–44, 1989.
9. F. Giunchiglia and T. Walsh. A Theory of Abstraction. *Artificial Intelligence Journal*, 56(2–3):323–390, October 1992.
10. G. Große, S. Hölldobler, J. Schneeberger, U. Sigmund and M. Thielscher. Equational Logic Programming, Actions and Change. In K. Apt, editor, *Proceedings of IJCSLP*, pages 177–191, Washington, 1992. The MIT Press.
11. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. Technical report, Brown University, Department of Computer Science, December 1993.
12. P.M. Hill and J.W. Lloyd. Analysis of meta-programs. In *Meta-Programming in Logic Programming*, pages 23–52. The MIT Press, 1989.
13. S. Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *LNAI*. Springer-Verlag, 1989.
14. S. Hölldobler and J. Schneeberger. A New Deductive Approach to Planning. *New Generation Computing*, 8:225–244, 1990.
15. S. Hölldobler and M. Thielscher. Computing change and specificity with equational logic programs. *Annals of Mathematics and Artificial Intelligence*, 14(1):99-133, 1995.
16. V. Lifschitz. On the Semantics of STRIPS. In M. P. Georgeff and A. L. Lansky, editors, *Proceedings of the Workshop on Reasoning about Actions & Plans*. Morgan Kaufmann, 1986.
17. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
18. K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of logic programs. *Journal of Logic Programming*, 13:181–204, 1992.
19. M. Masseron, C. Tollu, and J. Vauzielles. Generating Plans in Linear Logic I. Actions as Proofs. *Theoretical Computer Science*, 113:349–370, 1993.
20. J. McCarthy. Situations and Actions and Causal Laws. Stanford Artificial Intelligence Project, Memo 2, 1963.
21. J. McCarthy and P.J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.
22. K.H. Munch. A new reduction rule for the connection graph proof procedure. *Journal of Automated Reasoning*, 4:425–444, 1988.
23. D. Plaisted. Abstraction mappings in mechanical theorem proving. *Automated Deduction—CADE-5*, pages 264–280. Springer-Verlag, 1980.
24. E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.

This article was processed using the LaTeX macro package with LLNCS style