In K. Apt, editor, Proc. of the Int. Joint Conf. and Symp. on Logic Programming, pages 177–191, 1992.

Equational Logic Programming, Actions, and Change

G.Große, S.Hölldobler, J.Schneeberger, U.Sigmund, M.Thielscher
Intellektik, Informatik, TH Darmstadt, Germany

Abstract
Recently three approaches for solving planning problems deductively were proposed each of which does not require to state frame axioms explicitly. These approaches are based on the linear connection method, an equational logic programming language, and on linear logic. In this paper, we briefly review these approaches and show that they are equivalent. Moreover, we illustrate that these approaches are not only restricted to deductive planning, but can be applied whenever actions are to be modelled in logic. We show that the approaches essentially amount on building predicates over the data structure multiset. Such multisets are interpreted as resources, which are consumed and produced by actions. We give a minimal and complete unification algorithm for the equational theory which defines the multisets. Finally, we discuss possible extensions of the equational logic programming approach.

1 Introduction

Logical approaches to computer science and artificial intelligence offer – among others – the advantage of a declarative representation of knowledge. Originally, classical logic was designed and used for the representation of static knowledge. More recently logic has also been applied to model actions, states, and changing situations (eg. [22, 21, 18, 27, 10, 19]). The very first approaches in this direction revealed some fundamental problems such as the *frame problem* [22], ie. the problem of how to represent the invariants of a situation with respect to a given action. To handle this problem J. McCarthy, P. Hayes [22], and C. Green [12] introduced *frame axioms*. However, they needed $n \times m$ frame axioms, where n is the number of actions and m is the number of fluents,[1] in a planning problem. This number was reduced to n by Kowalski [17], who introduced a different representation of fluents using a *Holds* predicate. However, these frame axioms still pose a considerable problem to automated theorem provers as they may lead to many redundant derivations and the application of these axioms must be deferred as long as possible. W. Bibel [3] used a modified version of his connection method to solve the frame problem without the need of any frame axioms. He considered only *linear* proofs, ie. proofs in which each literal is used at most once. Unfortunately, Bibel was unable to give a semantics for the linear connection method and as R. Kowalski states *if Bibel's system really works, then it deserves an explanation and it deserves a semantics* (see Discussion in [4]). Recently, M. Masseron etal. [20] applied the multiplicative fragment of linear logic [11] to planning and showed that in this framework planning problems can also be solved without frame axioms.

A different approach to deductive planning, which also avoids the frame axioms, was given in [15]. There, situations – viz. collections of fluents – are represented using a binary function

[1] Ie. essential properties describing situations.

symbol $\circ$, which is associative (A), commutative (C), and admits a unit element $\emptyset$ (1). For example, the situation in which two blocks a and b are on a table t and are clear can be described by the term $on(a,t) \circ on(b,t) \circ cl(a) \circ cl(b)$.[2] The planning process itself is specified using a predicate $plan(s,p,t)$ which is interpreted declaratively as *the execution of plan p transforms situation s into situation t*. Actions are defined by rules of the form

$$plan(preconditions \circ V, action(P), W) \ :- \ plan(postconditions \circ V, P, W),$$

where the pre- as well as the postconditions are collections of fluents connected by $\circ$. Such rules are applicable if the preconditions are part of the current situation and all remaining fluents are bound – via an AC1-unification procedure – to a variable V. For example, the *mv*-operator moves a block from one location to another one.

$$\begin{array}{l} plan(cl(X) \circ cl(Y) \circ on(X,Z) \circ V, mv(X,Y,P), W) \\ :- plan(cl(X) \circ cl(Z) \circ on(X,Y) \circ V, P, W). \end{array}$$

Finally, derivations are terminated with the help of a fact, which states that there is nothing to do if the goal situation is already contained in the current situation.

$$plan(V \circ W, \Lambda, V).$$

Queries to such a program can be answered using SLDE-resolution, where the equational theory AC1 is built into a special unification procedure. Moreover, the approach admits a standard semantics by applying the results from [16] or [14].

One should observe that the pre- as well as the postcondition of an action are just collections of fluents which can intuitively be understood as conjunctions of fluents. A more precise interpretation will be given in Section 4. This restriction holds also for [20]. W. Bibel allows a more general form, but all examples given in [3, 5] are restricted in precisely the same way.

The purpose of this paper is as follows.

1. We proved that the equational logic programming approach to deductive planning is equivalent to a the linear connection method and the linear logic approach to deductive planning. This result is obtained by transforming SLDE-refutations into linear connection proofs and linear logic proofs, respectively, and vice versa. With the help of these transformations, we do now obtain a semantics for the linear connection method as the standard semantics for logic programming modulo the equational theory AC1.
2. We show that the equational logic programming approach is not restricted to deductive planning, but can always be applied if situations are specified by conjunctions of fluents and if actions are defined over such situations. We illustrate this generality by specifying objects and database updates in this framework in analogy to [2] and [23].
3. We show that specifying conjunctions of fluents using an AC1-operator essentially amounts to defining predicates over the data structure multiset and that fluents represent resources which are consumed and produced by the actions.
4. We give an efficient AC1-unification algorithm for computing a complete and minimal set of unifiers modulo AC1.
5. We extend our approach by admitting a form of idempotent disjunction among fluents such that we can solve the 3-socks problem quite naturally. Moreover, we demonstrate that Mendel must have used some sort of non-idempotent disjunction (or linear logic) when he discovered his famous laws in genetics.

The paper concludes with a discussion and various ideas on future work.

[2]Throughout the paper we use PROLOG-syntax.

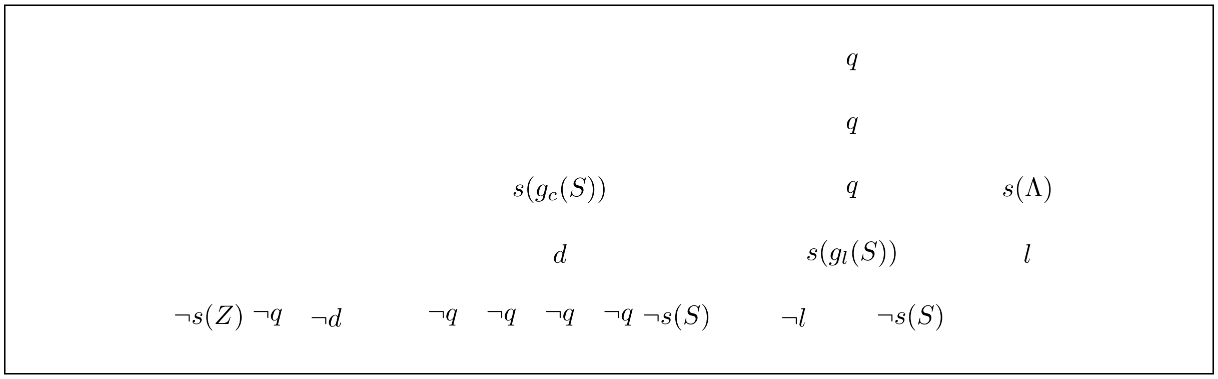
q
q
$s(g_c(S))$ q $s(\Lambda)$
d $s(g_l(S))$ l
$\neg s(Z)$ $\neg q$ $\neg d$ $\neg q$ $\neg q$ $\neg q$ $\neg q$ $\neg s(S)$ $\neg l$ $\neg s(S)$

Figure 1: A linear connection proof for the get lemonade example.

2 Deductive Planning

In this section we briefly repeat W. Bibel's [3], S. Hölldobler and J. Schneeberger's [15] as well as M. Masseron etal. approach [20] to deductive planning and show that these approaches are equivalent.

We illustrate the three approaches with the help of a little example. Suppose a thirsty person named Bert wants to get some lemonade from a vending machine. The lemonade costs 75 cents, which should be no problem since Bert has a one-dollar note as well as a quarter in his jacket. Unfortunately, the vending machine is kind of outdated and accepts only quarters. But there is also a cashier, which changes a dollar into four quarters. The problem of getting the lemonade can be described as a planning problem with the initial situation of Bert having a dollar note (d) and a quarter (q), the operators *get-change* (g_c) and *get-lemonade* (g_l), which allow him to change a dollar and to get a lemonade, respectively, and the goal where Bert has a lemonade (l) in his hand. Clearly, a solution to this problem is the plan with the two consecutive actions get-change and get-lemonade. One should observe that the pre- as well as the postconditions of both operators are conjunctions of fluents.

Linear Connection Method. W. Bibel's [3] approach to deductive planning is based on the connection method. Therein the initial situation is represented by the axiom $\exists Z\ [s(Z) \wedge d \wedge q]$, the operators get-change and get-lemonade by the axioms

$$\forall S\ [s(g_c(S)) \wedge d \ \rightarrow\ q \wedge q \wedge q \wedge q \wedge s(S)]$$
$$\forall S\ [s(g_l(S)) \wedge q \wedge q \wedge q \ \rightarrow\ l \wedge s(S)]$$

respectively, and the goal by $s(\Lambda) \wedge l$, where Λ is a constant denoting the empty plan. The predicate s is a so-called state literal, whose only role is to record the actions taken in order to achieve the goal. Figure 1 shows a valid connection proof for our example yielding the desired answer substitution $\{Z \mapsto g_c(g_l(\Lambda))\}$, ie. get-change first and, then get-lemonade.

The remarkable feature of the proof shown in Figure 1 is its *linearity*, ie. every literal is engaged in at most one connection. Without this linearity we might be able to connect the literal $\neg q$ occurring in the initial situation three times such that the conditions of the get-lemonade operator are fulfilled in the initial situation. In other words, one quarter would be enough to get a lemonade.

Equational Logic Programming. In the equational logic programming approach of S. Hölldobler and J. Schneeberger [15] already mentioned in the introduction the lemonade example can be expressed by the following program.

$$\begin{aligned}
&\mathit{plan}(d \circ V, g_c(P), W) \;:-\; \mathit{plan}(q \circ q \circ q \circ q \circ V, P, W).\\
&\mathit{plan}(q \circ q \circ q \circ V, g_l(P), W) \;:-\; \mathit{plan}(l \circ V, P, W).\\
&\mathit{plan}(V \circ W, \Lambda, V).
\end{aligned}$$

The first two clauses specify the actions get-lemonade and get-change, whereas the final clause states that the current situation already establishes the goal situation. The question of whether there exists a plan such that Bert can get a lemonade for a dollar and a quarter can now be answered as shown in Figure 2.

It is important to note that $\circ$ is not idempotent. Otherwise, $?-\,\mathit{plan}(d \circ q \circ q \circ q \circ q \circ q, P_1, l)$ would be an SLDE-resolvent of $?-\,\mathit{plan}(d \circ q, P, l)$ and the first program clause. But this would be like growing money on trees. Furthermore, the frame axioms are not needed as the variable V in the program clauses together with the unification computation built into SLDE-resolution take each fluent which is invariant under the action[3] to the next goal clause.

In [15] the semantics of the linear logic programming approach to deductive planning is defined as the standard semantics of a logic program with equality. It is shown that the approach is sound and complete.

Linear Logic. The use of linear logic for planning problems was proposed by M. Massaron etal. [20]. In this approach our running example is specified by the *current state axioms* $\vdash d$ and $\vdash q$ and by the following *transition axioms*.

$$\begin{aligned}
&d \vdash q \otimes q \otimes q \otimes q.\\
&q, q, q \vdash l.
\end{aligned}$$

A proof in this approach looks similar to Gentzen-like proofs. In Figure 3 a linear logic proof for our example is depicted. The plan which solves the given planning problem can be extracted from the proof by recording the used transition axioms together with their order in the proof.

[3]A fluent is *invariant* if it is not among the preconditions of an action.

$$\begin{aligned}
&?-\;\mathit{plan}(d \circ q, P, l).\\
&?-\;\mathit{plan}(q \circ q \circ q \circ q \circ q, P_1, l).\\
&?-\;\mathit{plan}(q \circ q \circ l, P_2, l).\\
&\square
\end{aligned}$$

Figure 2: The SLDE-resolution of $?-\,\mathit{plan}(d \circ q, P, l)$ obtained by applying – in this order – the first, second, and third program clause yields the answer substitution $\{P \mapsto g_c(g_l(\Lambda))\}$.

$$\dfrac{\dfrac{q \vdash q \qquad d \vdash q \otimes q \otimes q \otimes q}{q, d \vdash q \otimes q \otimes q \otimes q \otimes q}{\scriptstyle \otimes\_r} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{q \vdash q \quad q \vdash q}{q, q \vdash q \otimes q}{\scriptstyle \otimes\_r} \qquad q, q, q \vdash l}{q, q, q, q, q \vdash q \otimes q \otimes l}{\scriptstyle \otimes\_r}}{q, q, q, q \otimes q \vdash q \otimes q \otimes l}{\scriptstyle \otimes\_l}}{q, q, q \otimes q \otimes q \vdash q \otimes q \otimes l}{\scriptstyle \otimes\_l}}{q, q \otimes q \otimes q \otimes q \vdash q \otimes q \otimes l}{\scriptstyle \otimes\_l}}{q \otimes q \otimes q \otimes q \otimes q \vdash q \otimes q \otimes l}{\scriptstyle \otimes\_l}}{q, d \vdash q \otimes q \otimes l}{\scriptstyle cut}$$

Figure 3: A linear logic proof for the get lemonade example. For the definition of the rules $\otimes\_l$, $\otimes\_r$, and *cut* see [20].

One should observe that it is impossible to derive the sequent $q, d \vdash l$ instead of $q, d \vdash l \otimes q \otimes q$ in this framework. The linear logic approach as stated in [20] does not provide a mechanism for deleting essentials on the right side of a sequent.

The Equivalence Result. As the example depicted in Figures 1 and 2 already indicates, SLDE-refutations with respect to a linear logic program can be transformed into a linear connection proof and vice versa. The proof is by induction on the length of the SLDE-refutation. Similarly, as the example depicted in Figures 2 and 3 indicates, SLDE-refutations with respect to a linear logic program can be transformed into a linear logic proof. The proof is again by an induction on the length of SLDE-refutations. Conversely, it can be shown that a linear logic proof can be transformed into an SLDE-refutation by induction on the length of the linear proof. Due to lack of space, we have to omit the formal proofs. They can be found in [13] or [25].

Theorem 1 *Let $\mathcal{P}$ be a planning problem, where the pre- and postconditions of actions are conjunctions of fluents. Plan P is a solution for $\mathcal{P}$*

iff P is generated by a conjunctive equational logic program.

iff P is generated by a conjunctive linear connection proof.

iff P is generated by a conjunctive linear logic proof.

3 Objects and Database Updates

Although the approach in [15] was developed for deductive planning, it is not restricted to the planning domain. The basic ideas were to represent fluents on the object level with the help of an AC1-operator $\circ$, to specify rules for actions of the form

$$plan(preconditions \circ V, action(P), W) \ :- \ plan(postconditions \circ V, P, W),$$

and apply SLDE-resolution. The very same idea can be used to represent and manipulate objects and database items.

Objects. The combination of logic programming and object-oriented programming offers the advantage of both paradigms. From logic programming it inherits the declarative representation of data. From object-oriented programming it receives the structured representation of data in classes of objects, data structuring by inheritance among classes, and dynamic modification of data. In our approach the properties of an object are represented as fluents and connected by the AC1-operator $\circ$. For example, an object *point* which is at location $(7,3)$ and receives the input proj_x is represented by the term $\mathit{point} \circ \mathit{input}(\mathit{proj}_x(I)) \circ x(7) \circ y(3)$. Communication between objects is realized by shared variables as proposed in [26]. In our example, I is such a variable. Transitions are specified by rules. For example, a transition for projecting two-dimensional points on the x-axis can be specified by the following rule.

$$\mathit{obj}(\mathit{point} \circ \mathit{input}(\mathit{proj}_x(I')) \circ y(Y) \circ V) :- \ \mathit{obj}(\mathit{point} \circ \mathit{input}(I') \circ y(0) \circ V). \qquad (1)$$

Now the query

$$?- \ \mathit{obj}(\mathit{point} \circ \mathit{input}(\mathit{proj}_x(I)) \circ x(7) \circ y(3))$$

can be resolved with the transition rule (1) to

$$?- \ \mathit{obj}(\mathit{point} \circ \mathit{input}(I) \circ x(7) \circ y(0))$$

using the AC1-unifier $\{Y \mapsto 3,\ V \mapsto x(7),\ I' \mapsto I\}$. One should observe that the transition rule (1) is applicable to all objects which belong to the class of points and possess a y-coordinate or which belong to a subclass thereof. For example, a point which belongs to the subclass of coloured two-dimensional points and receives a proj_x–message, ie. the query

$$?- \ \mathit{obj}(\mathit{point} \circ \mathit{input}(\mathit{proj}_x(I)) \circ x(7) \circ y(3) \circ \mathit{colour}(\mathit{blue})),$$

is transformed to

$$?- \ \mathit{obj}(\mathit{point} \circ \mathit{input}(I) \circ x(7) \circ y(0) \circ \mathit{colour}(\mathit{blue}))$$

via SLDE-resolution with transition rule (1) using the AC1-unifier $\{Y \mapsto 3,\ V \mapsto x(7) \circ \mathit{colour}(\mathit{blue}),\ I' \mapsto I\}$. The only difference to the previous derivation is the binding of the variable V, which now contains also all additional (inherited) properties of the object.

There is no essential difference between the *plan* predicate and the *obj* predicate. One could easily rephrase objects and their transitions using a ternary predicate *object* such that $\mathit{object}(o,t,o')$ represents the fact that *object o is transformed into object o' after receiving messages t*. Since Theorem 1 tells us that we can transform SLDE-derivations into linear logic proofs and linear connection proofs and vice versa, we essentially incorporate objects into linear logic and the linear connection method as well.

There already exists a framework for representing objects in logic programs, which is very similar to our approach [2]. In fact, our example is taken from [2]. There, J.M. Andreoli and R. Pareschi represent the properties of objects as predicates which are connected via the binary connective @. @ plays the same role as $\circ$ in our approach except that @ is interpreted as *multiplicative disjunction* whereas we interpret $\circ$ as *multiplicative conjunction*. To think of @ as a disjunction is kind of counterintuitive as, then, an object like a point has an x-coordinate *or* has a y-coordinate. Formally, however, J.M. Andreoli and R. Pareschi treat @ as an AC1-operator and, thus, it corresponds precisely to $\circ$. There is also a technical difference between [2] and the work reported herein. In their approach proof rules are only defined for ground goals and rules rigorously, and it is stated that these rules can be lifted via unification. In our approach, the operational semantics is rigorously defined for general goals and rules.

Database Updates. In analogy to actions in planning scenario, database transactions are specified using a predicate $db(d, t, d')$, which is read declaratively as *the execution of the transaction sequence* t *transforms the database* d *into* d'. As an example consider a toy education database specifying relations about students, courses, and grades. $enr(St, C)$ tells us that the student St is enrolled in course C, $grd(St, C, G)$ that her grade in course C is G, and $pre(P, C)$ that P is a prerequisite course for course C. A transaction for a student St to be registered for a course C is only possible if she has obtained a grade G of at least 50 in all prerequisite courses.

$$\begin{array}{l} db(V, register(St, C, S), W) :- db(enr(St, C) \circ V, S, W),\ \neg low\_pre(St, C, V). \\ low\_pre(St, C, V) :- AC1\text{-}unify(V, Y \circ pre(P, C) \circ grd(St, P, G)),\ G \leq 50. \end{array}$$

The predicate *AC1-unify* unifies its arguments modulo AC1 and negation is handled by negation as failure. Now, if we want to know whether *sue* can register for course $m6$ in the situation $t = enr(sue, c1) \circ pre(m4, m6) \circ grd(bill, m1, 70) \circ grd(sue, m3, 85) \circ grd(sue, m4, 55)$ we ask the following query.

$$?-\ db(t, register(sue, m6, \Lambda), W).$$

Resolving this query with the clauses above and with the terminating fact $db(V, \Lambda, V)$ yields the empty clause and computed answer substitution $\{W \mapsto enr(sue, m6) \circ t\}$. This result is obtained because the proof of $low\_pre(sue, m6, t)$ fails.

This little example demonstrates how complex queries – including universally quantified variables here – can be handled using general concepts from logic programming. Similarly, we can state integrity constraints employing a modified termination clause together with the necessary predicates (see [15] for a more detailed discussion).

$$\begin{array}{l} db(V, \Lambda, V) :- consistent(V). \\ consistent(V) :- \neg inconsistent(V). \\ inconsistent(V \circ grd(St, C, G) \circ grd(St, C, G')), G' \neq G. \\ \ldots \end{array}$$

The little examples in this section were taken from [23]. However, it should be noted that R. Reiter's approach is more fundamental since it deals with standard situation calculus. Nevertheless, we observe that most of the examples he is able to solve are also feasible in our approach.

4 Predicates over Multisets

In our equational logic programming approach to actions and change we use terms of the form $s_1 \circ \ldots \circ s_n$, $n \geq 0$, to represent situations, where the s_i are fluents, ie. non-variable terms which do not contain the $\circ$ function symbol. If such terms are ground, then they can be mapped to multisets with the help of an interpretation $\mathcal{I}$ as follows.[4]

$$\begin{array}{lcl} \mathcal{I}(\emptyset) & = & \dot{\{}\dot{\}}. \\ \mathcal{I}(a) & = & \dot{\{}a\dot{\}} \text{ if } a \text{ is a fluent.} \\ \mathcal{I}(s \circ t) & = & \mathcal{I}(s) \,\dot{\cup}\, \mathcal{I}(t). \end{array}$$

[4]Multisets are depicted using the modified curly brackets $\dot{\{}$ and $\dot{\}}$. Furthermore, $\dot{\subseteq}$, $\dot{\cup}$, $\dot{\setminus}$, etc. denote the multiset extensions of the usual set operations $\subseteq$, $\cup$, $\setminus$, etc.

In other words, our predicates like *plan*, *obj*, or *db* are essentially predicates over the data-structure multiset. By the use of multisets changing situations is now like dealing with resources. By multiset operations we delete or add certain amounts of fluents to the current situation in order to yield a new situation. This update procedure is similar to the procedure used in Strips [8] but uses multisets instead of sets.

It is interesting to see that in all the three approaches to reasoning about change depicted in Section 2, the multiset representation for situations did not only solve the frame problem, but also led to a considerable gain of efficiency. This shall be illustrated in the following example. Suppose we had an arbitrary set of terms representing our domain in question. Certain subsets might represent certain situations. For instance, in a domain consisting of four quarters there are $16 = 2^4$ different situations. On the other hand, as common in practice, we are not really concerned with the question which quarter we got but more in how many quarters we got. The number of situations important for this knowledge is much less: just 5. Either we got none, or 1, or ... or 4 quarters. In fact what we have to do is just taking the identity of these quarters and forming the multiset of quarters. This reduction of the number of situations is not to underestimate as the following example shall demonstrate. Suppose we had two lemonades, four dollars and four quarters in our domain. Then the number of situations reduces from $1024 = 2^{10}$ to 75. This in turn affects the principle number of situation changes, which is the square of the number of situations. Instead of around 10^6 we can work with around 5×10^3; quite a difference. But as always there is no free dinner. If we use such a multiset representation we do not know with which object of the multiset we are left after some change. We cannot, for instance, say which of the 4 quarters remains in our pocket after spending three.

5 AC1-Unification

So far we have given an equational logic programming approach to reasoning about action and change, but have said nothing about computational aspects. We have only mentioned that the equational theory shall be built into the unification computation. In practice, however, an efficient implementation of a special E-unification procedure is indispensable and we will give such unification procedures in this section.

AC1-unification – as required in our approach – is finitary,[5] ie. there is always a finite and minimal set of unifiers for two terms. Therefore, the aim of building in an AC1-unification algorithm is to generate a complete and minimal set of unifiers.

There are a variety of AC1-unification algorithms (cf. [7]). However, the AC1-unification problems encountered in this paper are of a special kind. The general AC1-unification algorithms perform a lot of unnecessary and redundant computations if applied to these problems. More formally, the AC1-unification problems considered herein are defined as follows. Let a *fluent* be a non-variable term, which does not contain the $\circ$ function symbol. For example, $colour(blue)$, $point(I)$, and $\emptyset$ are fluents. In the remainder of this paragraph let s and t (possibly indexed) denote fluents. We will consider three different unification problems with increasing complexity.

- An *AC1-matching problem* consists of two terms of the form $s_1 \circ \ldots \circ s_n$ and $t_1 \circ \ldots \circ t_m \circ W$, where the s_i, $1 \leq i \leq n$, are ground and W does not occur in t_j.

[5]The notions and notations concerning unification under an equational theory are taken from [28].

- A *restricted AC1-unification problem* consists of two terms of the form $s_1 \circ \ldots \circ s_n$ and $t_1 \circ \ldots \circ t_m \circ W$, where W does neither occur in s_i nor in t_j.[6]
- An *AC1-unification problem* consists of two terms of the form $s_1 \circ \ldots \circ s_n \circ V$ and $t_1 \circ \ldots \circ t_m \circ W$, where V and W do neither occur in s_i nor in t_j.

The variables V and W which occur in the previous definitions are called *AC1-variables*.

AC1-matching. It turned out that in many applications of our equational logic programming approach to planning, one of the two terms to be unified modulo AC1 was ground. Thus, we have the chance to use an AC1-matching instead of an AC1-unification algorithm quite often. Therefore, we first describe an algorithm which generates a complete and minimal set of matchers for an AC1-matching problem. A substitution σ is a matcher for an AC1-matching problem iff $\sigma(W \circ t_1 \circ \ldots \circ t_m) =_{AC1} s_1 \circ \ldots \circ s_n$, where s_i, $1 \leq i \leq n$, are ground. It is easy to prove that if σ is a solution for the AC1-matching problem then $\dot{\{}\sigma t_1, \ldots, \sigma t_m\dot{\}} \dot{\subseteq} \dot{\{}s_1, \ldots, s_n\dot{\}}$. Conversely, if we find a substitution θ such that $\dot{\{}\theta t_1, \ldots, \theta t_m\dot{\}} \dot{\subseteq} \dot{\{}s_1, \ldots, s_n\dot{\}}$, then the AC1-matching problem consisting of the terms $s_1 \circ \ldots \circ s_n$ and $t_1 \circ \ldots \circ t_m \circ W$ is solvable and the matching substitution σ can be constructed from θ as follows. Let $\dot{\{}u_1, \ldots, u_k\dot{\}} = \dot{\{}s_1, \ldots, s_n\dot{\}} \dot{\setminus} \dot{\{}\theta t_1, \ldots, \theta t_m\dot{\}}$. Then, $\sigma = \theta|_{\mathcal{V}ar(t_1, \ldots, t_m)} \cup \{W \mapsto u_1 \circ \ldots \circ u_k\}$[7].

Let $\mathcal{S}$ and $\mathcal{T}$ be two multisets. With the previous discussion, we are now interested in computing a complete and minimal set Σ of substitutions such that for each $\sigma \in \Sigma$ we find that $\sigma\mathcal{T} \dot{\subseteq} \mathcal{S}$.[8] The following algorithm recursively generates this set.

If $\mathcal{T} = \dot{\{}\dot{\}}$ then $\Sigma = \{\varepsilon\}$.

If $\mathcal{T} = \dot{\{}t\dot{\}} \dot{\cup} \mathcal{T}'$ then $\Sigma = \{\theta\sigma' \mid \sigma' \in \Sigma' \wedge \exists s \in \mathcal{S} \dot{\setminus} \sigma'\mathcal{T}' : \theta = mgu(\sigma' t, s)\}$[9], where Σ' is a complete and minimal set of substitutions such that for each $\sigma' \in \Sigma'$ we find $\sigma'\mathcal{T}' \dot{\subseteq} \mathcal{S}$.

One should observe that for each σ' and each s the substitution $\theta\sigma' \in \Sigma$ is unique. Hence, the algorithm does no redundant computation.

Restricted AC1-Unification. Restricted AC1-unification can also be reduced to a subset problem over multisets, where substitutions may be applied to both multisets. Let θ be a substitution such that $\dot{\{}\theta t_1, \ldots, \theta t_m\dot{\}} \dot{\subseteq} \dot{\{}\theta s_1, \ldots, \theta s_n\dot{\}}$, then a unifier σ of $s_1 \circ \ldots \circ s_n$ and $t_1 \circ \ldots \circ t_m \circ W$ can be constructed from θ as shown within the AC1-matching. To generate a complete set of substitutions σ such that $\sigma\mathcal{T} \dot{\subseteq} \sigma\mathcal{S}$ holds, we modify the algorithm shown above in the following way.

If $\mathcal{T} = \dot{\{}\dot{\}}$ then $\Sigma = \{\varepsilon\}$.

If $\mathcal{T} = \dot{\{}t\dot{\}} \dot{\cup} \mathcal{T}'$ then $\Sigma = \{\theta\sigma' \mid \sigma' \in \Sigma' \wedge \exists s \in \sigma'\mathcal{S} \dot{\setminus} \sigma'\mathcal{T}' : \theta = mgu(\sigma' t, s)\}$, where Σ' is a complete and minimal set of substitutions such that for each $\sigma' \in \Sigma'$ we find $\sigma'\mathcal{T}' \dot{\subseteq} \sigma'\mathcal{S}$.

The resulting set Σ is complete but, unfortunately, it may contain non-minimal substitutions. Therefore, we have to test and, if necessary, to remove some substitutions. We use some heuristics to keep Σ as small as possible during computation.

[6] Note that the fluents s_i may now contain variables.

[7] $\mathcal{V}ar(X)$ denotes the set of variables occurring in the syntactic object X and $\sigma|_V$ denotes the restriction of the substitution σ to the variables in V.

[8] The notion of a minimal and complete set of substitutions is extended in the obvious way.

[9] $mgu(s, t)$ denotes the most general unifier of s and t.

AC1-Unification. The problem of unifying two terms that both include an AC1-variable cannot be reduced to a subset problem over multisets. However, each solution σ of the AC1-unification problem defined by the terms $s_1 \circ \ldots \circ s_n \circ V$ and $t_1 \circ \ldots \circ t_m \circ W$ can be interpreted as dividing the representing multisets $\mathcal{S} = \{s_1, \ldots, s_n\}$ and $\mathcal{T} = \{t_1, \ldots, t_m\}$ into two disjunctive parts $\mathcal{S}_1, \mathcal{S}_2$ and $\mathcal{T}_1, \mathcal{T}_2$, respectively, such that we find a most general substitution θ which unifies $\mathcal{S}_1$ and $\mathcal{T}_1$. Let $\theta\mathcal{S}_2 = \{u_1, \ldots, u_l\}$ and $\theta\mathcal{T}_2 = \{v_1, \ldots, v_k\}$. Then, $\sigma = \theta \cup \{V \mapsto v_1 \circ \ldots \circ v_k,\ W \mapsto u_1 \circ \ldots \circ u_l\}$.

The algorithms for AC1-matching, restricted AC1-unification, and AC1-unification are implemented and used successfully within a Prolog–implementation of our equational logic programming approach. They turned out to be very efficient, as they consider the characteristics of the AC1-terms occurring in all applications of our equational logic programming approach to actions and change. The details of the algorithms as well as a proof of their correctness, completeness, and minimality can be found in [30].

6 On Disjunction

So far we dealt only with conjunctions of fluents and we have shown how such conjunctions can be used for modelling planning problems, objects, and database updates. In certain domains, however, disjunctions arise naturally. Thus, we are faced with the problem of extending our approach to handle disjunction. In this section we will give two examples, the 3-socks problem and Mendel's law in genetics, which illustrate the need for two different forms of disjunction and sketch how these disjunctions can be modelled in an equational logic programming language.

The 3-socks problem. Imagine that we are standing in a dark room in front of a drawer which contains black and white socks. The 3-socks problem is now the question of how often we have to fetch a sock out of the drawer before we can be sure to have a pair of matching socks. As the name already says, the solution is 3. But how can we formalize the problem such that the solution is deduced? [3] as well as [15] give a solution within the framework of conjunctive planning, where the problem is mapped onto a pair of natural numbers. One of these numbers is incremented at each *fetch* operation until eventually one of them is 2. These solution are formally correct, but they are quite unnatural and unintuitive.

If we fetch a sock, then we know that we do have a sock, but we do not know whether this sock has the colour black or white. Hence, the result of the *fetch* operation should be formalized as a term $b \mid w$. What are the properties of $\mid$? As we intend to think of $\mid$ as an additive disjunction, $\mid$ should be associative (A) and commutative (C). But $\mid$ should also be idempotent (I), ie. $X \mid X = X$ should hold, because if in our example all socks are white we do hold a single white sock after a *fetch* operation. These properties in mind we can now specify the *fetch* operation.

$$\mathit{plan}(V, \mathit{fetch}(P), W).\ :-\ \mathit{plan}((b \mid w) \circ V, P, W). \tag{2}$$

The 3-socks problem can now be specified by the following query.

$$?-\ \mathit{plan}(\emptyset, P, (b \circ b) \mid (w \circ w)).$$

In other words, we are looking for a plan P such that its execution transforms the empty multiset into a multiset, where we find either two black or two white socks. Resolving this goal clause three times with (new variants of) rule (2) leads to

$$?- \mathit{plan}((b \mid w) \circ (b \mid w) \circ (b \mid w), P_3, (b \circ b) \mid (w \circ w)) \tag{3}$$

and to the binding $\{P \mapsto \mathit{fetch}(\mathit{fetch}(\mathit{fetch}(P_3)))\}$. In order to terminate the derivation we have to resolve (3) and the terminating fact $\mathit{plan}(V \circ W, \Lambda, V)$. But this requires to specify the interaction between $\circ$ and $\mid$. If we regard fluents as resources and a term of the form $X \mid Y$ as having either resource X or resource Y but not both, then it is natural to require that $\circ$ distributes over $\mid$, ie.

$$X \circ (Y \mid Z) \;=\; (X \circ Y) \mid (X \circ Z).^{10}$$

With the help of this axiom, the AC1 axioms for $\circ$, and the ACI axioms for $\mid$ we can now resolve (3) and the terminating fact leading to the binding $\{P \mapsto \mathit{fetch}(\mathit{fetch}(\mathit{fetch}(\Lambda)))\}$.

Mendel's law of genetics. Mendel wanted to understand how the colour of peas is determined. He crossed peas, counted the number of yellow and green peas in the first, second, etc. generation, and by a kind of backward reasoning discovered that the colour of peas is determined by two genes, each of which carries the hereditary factor for either yellow or green. Green is dominant and yellow is recessive, ie. peas are green if they have at least one green gene and peas are yellow if they have two yellow genes. During crossing the genes determining the colour are split and combined with a gene from another pea.

We will show how the crossing process can be modelled in an equational logic programming environment and how the ratio of yellow and green peas can be determined. We model peas with the help of a binary function symbol p, where $p(g_1, g_2)$ is interpreted as a pea with genes g_1 and g_2. The genes itself may be either yellow (y) or green (g). The colour of a pea, ie. whether the pea is *green* or *yellow*, can now be specified by the following equations.

$$\begin{array}{lcl} p(G_1,G_2) & = & p(G_2,G_1). \\ \mathit{green} & = & p(g,G). \\ \mathit{yellow} & = & p(y,y). \end{array} \tag{4}$$

The effect of crossing two peas is specified via a ternary predicate symbol *cp*, where $\mathit{cp}(p_1, p_2, p_3)$ is read declaratively as *crossing peas* p_1 *and* p_2 *yields* p_3. Of course, we do not know precisely how p_3 looks like; all we know is that its genes are a combination of the genes found in p_1 and p_2.

$$\mathit{cp}(p(X,Y), p(V,W),\; p(X,V) \,\#\, p(X,W) \,\#\, p(Y,V) \,\#\, p(Y,W)). \tag{5}$$

$\#$ is a binary, associative and commutative function symbol, whose intended meaning is disjunction. Because we are interested in determining the possibility of getting green and yellow peas, $\#$ must be *non*-idempotent and thus differs from the operator $\mid$ introduced in the previous section. The following example may illustrate this point.

In order to determine the outcome of crossing two green peas we ask the following query.

$$?- \mathit{cp}(\mathit{green}, \mathit{green}, Z). \tag{6}$$

[10] One should observe, that the law of distributivity of $\mid$ over $\circ$ does not meet our intuition. The term $X \mid (Y \circ Z)$ specifies that we have either X or $Y \circ Z$ but not both, whereas the term $(X \mid Y) \circ (X \mid Z)$ specifies that we have $X \mid Y$ and $X \mid Z$. In the latter case, we might have two resources of type X, whereas in the former case this is impossible.

(6) and (5) are unifiable modulo the equational theory (4) yielding the computed answer substitution

$$\{Z \mapsto \mathit{green} \# \mathit{green} \# \mathit{green} \# p(G_1,\ G_2)\}.$$

In other words, as we do not know whether the green peas were pure, we can only assume that one gene of the green peas has hereditary factor green and the other is undetermined which is indicated by the variables G_1 and G_2. It is straightforward to write a logic program to determine the possibility of receiving green and yellow peas in a crossing experiment. One simply has to count the number of occurrences of *green* and *yellow* in the binding for Z. In the example these numbers will be either 4 and 0 or 3 and 1, respectively, depending on the choices made for G_1 and G_2. If $\#$ were also idempotent – viz. an ACI-operator – then

$$\mathit{green} \# \mathit{green} \# \mathit{green} \# p(G_1,\ G_2) \ =_{ACI}\ \mathit{green} \# p(G_1,\ G_2)$$

and we would never be able to compute the correct possibilities for obtaining green and yellow peas.

We may speculate that Mendel must have used a non-idempotent (or multiplicative in the sense of [11]) disjunction, when he discovered his laws of genetics.

The two examples in this section illustrate that the equational logic programming approach to action and change can naturally be extended to cope with disjunction. This includes the semantics as given in [15].

7 Discussion

In this paper we have shown that three recent approaches to logic and change – viz. the linear connection method, linear logic, and equational logic programming – are equivalent if pre- and postconditions of actions are multiplicative conjunctions of fluents. This result does not only provide a standard semantics for (a fragment of) the linear connection method and linear logic, but also brings together three approaches, which were previously considered to be different. Moreover, this result allows to carry over insigths and results obtained in one approach to the other ones. However, there are still a variety of competing approaches to logic and change and their relation to the three approaches remains to be clarified. The situation calculus as used for example in [23] or labelled deductive systems [9] are just two of these approaches.

We have also shown that reasoning about action and change amounts in defining relations over the datastructure multiset and we have given complete and minimal unification algorithms for the equational theory which defines multisets. These algorithms are used in a PROLOG-implementation of the equational logic programming approach.

We believe that the equational logic programming approach to action and change has certain advantages over the linear connection method and linear logic. Equational logic admits a straightforward and well-understood standard semantics. The equational theory AC1 – used herein – can easily be added to a PROLOG system, which gives us a powerful and flexible implementation. As already demonstrated in the database example of Section 3 we can now combine reasoning over multisets with all other programming techniques in PROLOG, whereas for example in the linear logic approach Gentzen-style proofs have to be constructed.

Finally, we have outlined how the equational logic programming approach in [15] can be extended to handle two forms of disjunction – viz. idempotent and non-idempotent disjunction.

Although not mentioned so far, besides the non-idempotent conjunction $\circ$, we may also have an idempotent conjunction $\&$ on the object level. On the proof theoretic level incorporating these operators in the framework of [15] amounts in building in the equational theories for these operators into the unification computation. As unification algorithms for $|$, $\#$, and $\&$ are essentially variations of the AC1-unification algorithm for $\circ$ given in Section 5, the hard problem is to combine these algorithms (cf. [24]). On the model theoretic level, we have to understand the denotation of the operators $|$, $\#$, and $\&$. As disjunctions represent alternatives and $|$ as well as $\#$ shall denote disjunctions, one way to solve this problem might be by extending the interpretation $\mathcal{I}$ given in Section 4 to multisets of multisets. In this extension, a term $q \circ q$ might be interpreted as $\{\{q, q\}\}$, whereas the terms $b \mid b \mid w$ and $\mathit{green} \# \mathit{green} \# \mathit{yellow}$ used in Section 6 might be interpreted as $\{\{b\},\ \{w\}\}$ and $\{\{\mathit{green}\},\ \{\mathit{green}\},\ \{\mathit{yellow}\}\}$, respectively. An interpretation like $\{\{b\},\ \{w\}\}$ would tell us that the term $b \mid w$ denotes either a situation where we have a black sock or a situation where we have a white sock, whereas $\{\{\mathit{green}\},\ \{\mathit{green}\},\ \{\mathit{yellow}\}\}$ would tell us that the term $\mathit{green} \# \mathit{green} \# \mathit{yellow}$ denotes a situation where we have a green or a yellow pea, but also that it is more likely for the pea to be green. As on the proof theoretic level, the hard problem is to combine the various interpretations for the four operators. This problem is tackled in [29].

References

[1] J. Allen, J. Hendler, and A. Tate. *Readings in Planning.* Morgan Kaufmann, San Mateo, 1990.

[2] J-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3+4), 1991.

[3] W. Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.

[4] W. Bibel. A deductive solution for plan generation. In J. W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management*, pages 453 – 473. Springer, 1989. XII.

[5] W. Bibel, L. F. del Cerro, B. Fronhöfer, and A. Herzig. Plan generation by linear proofs: on semantics. In *Proceedings of the German Workshop on Artificial Intelligence*, pages 49 – 62. Springer Informatik Fachberichte *216*, 1989.

[6] R. S. Boyer, editor. *Automated Reasoning. Essays in Honor of Woody Bledsoe.* Automated Reasoning Series. Kluwer Academic Publishers, Dordrecht, Boston, London, 1991.

[7] H.-J. Bürckert, A. Herold, D. Kapur, J. H. Siekmann, M. E. Stickel, M. Tepp, and H. Zhang. Opening the AC-Unification race. *Journal of Automated Reasonsing*, 4:465–474, 1988.

[8] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189–208, 1971. Nachgedruckt in: [1].

[9] D. M. Gabbay. LDS — labeled deductive systems. Draft, July 1990.

[10] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the limitations of the situation calculus? In *[6]*, chapter 8, pages 167–179. Kluwer Academic Publishers, 1991.

[11] J. Y. Girard. Linear logic. *Journal of Theoretical Computer Science*, 50(1):1 – 102, 1987.

[12] C. Green. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 219–239. Morgan Kaufmann, 1969.

[13] G. Grosse, S. Hölldobler, and J. Schneeberger. On linear deductive planning. Internal Report, Technische Hochschule Darmstadt, Fachbereich Informatik, 1992.

[14] S. Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Computer Science*. Springer, 1989.

[15] S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990. A short version appeared in the Proceedings of the German Workshop on Artificial Intelligence, Informatik Fachberichte *216*, pages 63-73, 1989.

[16] J. Jaffar, J-L. Lassez, and M. J. Maher. A theory of complete logic programs with equality. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 175–184. ICOT, 1984.

[17] R. Kowalski. *Logic for Problem Solving*, volume 7 of *Artificial Intelligence*. North Holland, New York/Oxford, 1979.

[18] Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[19] V. Lifschitz. Toward a metatheory of action. In *Proceedings of the International Conference on Principles of Knowlege Representation and Reasoning*, pages 376–386, 1991.

[20] M. Masseron, C. Tollu, and J. Vauzielles. Generating plans in linear logic. In *Proceedings of the 10th FST-TCS*. Springer, LNCS *472*, 1990.

[21] J. McCarthy. Epistemological problems of artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1038–1044, 1977.

[22] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463 – 502. Edinburgh University Press, 1969.

[23] R. Reiter. On formalizing database updates: Preliminary report. In *Proceedings of the 3rd International Conference on Extending Database Technology*, 1992.

[24] M. Schmidt-Schauß. Unification in a combination of arbitary disjoint equational theories. In *Proceedings of the Conference on Automated Deduction*, pages 378–396, 1988.

[25] J. Schneeberger. *Plan Generation by Linear Deduction.* PhD thesis, Technische Hochschule Darmstadt, Fachbereich Informatik, 1992.

[26] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1:25–48, 1983.

[27] Y. Shoham. *Reasoning About Change.* MIT Press, 1988.

[28] J. H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7:207 – 274, 1989.

[29] Ute Sigmund. LLP - Lineare Logische Programmierung. Diplomarbeit, Technische Hochschule Darmstadt, Fachbereich Informatik, 1992. (in preparation).

[30] M. Thielscher. AC1-Unifikation in der linearen logischen Programmierung. Diplomarbeit, Technische Hochschule Darmstadt, Fachbereich Informatik, 1992. (in preparation).