Solving Planning Problems by Partial Deduction

Helko Lehmann and Michael Leuschel

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
{mal,hel99r}@ecs.soton.ac.uk

Abstract. We develop an abstract partial deduction method capable of solving planning problems in the Fluent Calculus. To this end, we extend "classical" partial deduction to accommodate both, equational theories and regular type information. We show that our new method is actually complete for conjunctive planning problems in the propositional Fluent Calculus. Furthermore, we believe that our approach can also be used for more complex systems, e.g., in cases where completeness can not be guaranteed due to general undecidability.

1 Introduction

One of the most widely used computational logic based formalism to reason about action and change is the situation calculus. In the situation calculus a situation of the world is represented by the sequence of actions $a_1, a_2, \ldots, a_k$ that have been performed since some initial situation s_0. Syntactically, a situation is represented by a term $do(a_k, do(a_{k-1}, \ldots, do(a_1, s_0)\ldots))$. There is no explicit representation of what properties hold in any particular situation: this information has to be derived using rules which define which properties are *initiated* and which ones are *terminated* by any particular action a_i.

The fluent calculus ($\mathcal{FC}$) "extends" the situation calculus by adding *explicit state* representations: every situation is assigned a *multi-set* of so called *fluents*. Every action a not only produces a new situation $do(a,\ldots)$ but also modifies this multi-set of fluents. This enables the fluent calculus to solve the (representational and inferential) frame problem in a simple and elegant way [10]. The fluent calculus can also more easily handle partial state descriptions and provides a solution to the explanation problem.

The multi-sets of fluents of $\mathcal{FC}$ are represented using an extended equational theory (EUNA, for extended unique name assumptions [11]) with an associated extended unification ($AC1$) which treats $\circ$ as a commutative and associative function symbol and 1° as the neutral element wrt $\circ$ (i.e., for any s, $s \circ 1^\circ =_{AC1} 1^\circ \circ s =_{AC1} s$). Syntactically, the empty multi-set is thus 1° and a multi-set of k fluents is thus represented as a term of the form $f_1 \circ \ldots \circ f_k$. This allows for a natural encoding of resources (à la linear logic) and it has the advantages that adding and removing a fluent f to a multi-set M can be very easily expressed using $AC1$ unification: $Add =_{AC1} M \circ f$ and $Del \circ f =_{AC1} M$.

In this light, the underlying execution mechanism of $\mathcal{FC}$ is the so-called SLDE–resolution, which extends ordinary SLD by adding support for an underlying equational theory, in this case $AC1$. An alternative way of implementing

$\mathcal{FC}$ would be to implement the equational theory as a rewrite system and then use narrowing. Unfortunately, both of these approaches are useless for many interesting applications of $\mathcal{FC}$. For example, both SLDE and narrowing, cannot solve the so-called *conjunctive planning problem*, which consists of finding a plan which transforms an initial situation i so as to arrive at a situation which contains (at least) all the fluents of some goal situation g, e.g. [3]. Indeed, for but the most trivial examples, both SLDE and narrowing will loop if no plan exists, and will (due to depth-first exploration) often fail to find a plan if one exists.

Part of the problem is the lack of detection of infinite failure (see [2]), but another problem is the incapability of producing a finite representation of infinitely many computed answers. In general, of course, these two problems are undecidable and so is the conjunctive planning problem. However, the conjunctive planning problem is not very different from the so called *coverability problem* in Petri nets: is it possible to fire a sequence of Petri net transitions so as to arrive at a marking which covers some goal marking. This problem *is* decidable (e.g., using the Karp-Miller procedure [12]) and in [13] it has been shown that a fragment of the fluent calculus has strong relations to Petri nets.

So, one might try to apply algorithms from the Petri net theory to $\mathcal{FC}$ in order to tackle the conjunctive planning problem. However, there is also a logic programming based approach which *can* solve these problems as well and which scales up to any extension expressible as a logic program. This approach is thus a more natural candidate, as it can not only handle the fragment of $\mathcal{FC}$ described in [13], but any $\mathcal{FC}$ domain which can be represented as a definite logic program. (although it will no longer be a decision procedure). Indeed, from [16] we know that *partial deduction* can be successfully applied to solve coverability problems of Petri nets. Thus, the idea of this paper is to apply partial deduction to fluent calculus specifications in order to decide the conjunctive planning problem for an interesting class of $\mathcal{FC}$ specifications (and to provide a useful procedure for more general $\mathcal{FC}$'s). There are several problems that still need to be solved in order for this approach to work:

- $\mathcal{FC}$ relies on $AC1$-unification, but unification under equational theory is not directly supported by partial deduction as used in [16] and one would have to apply partial deduction to a meta-interpreter implementing $AC1$-unification. Although this is theoretically feasible, this is still problematic in practice for efficiency and precision reasons. A more promising approach is to extend partial deduction so that it can handle an equational theory.
- Another problem lies with an inherent limitation of "classical" partial deduction, which relies on a rather crude domain for expressing calls: in essence a term represents all its instances. This was sufficient for handling Petri nets in [16] (where a term such as `[0,s(X)]` represents all Petri net markings with no tokens in place 1 and at least 1 token in place 2), but is not sufficient to handle $\mathcal{FC}$ whose state expressions are more involved. For example, given a $\mathcal{FC}$ specification with two fluents f_1 and f_2, it is impossible to represent a state which has one or more f_1's but no f_2's. Indeed, in "classical" partial deduction, a term such as $f_1 \circ X$ represents all its instances, and thus also

represents states which contain f_2's. To solve this we propose to use so called *abstract partial deduction* [14] with an abstract domain based upon regular types [24], and extend it to cope with equational theories.

Although in this paper we are mainly interested in applying partial deduction to the $\mathcal{FC}$ based upon $AC1$, we present the generalised partial deduction independently of the particular equational theory. However, the use of this general method in practice relies on an efficient unification procedure. If such a procedure can not be provided and/or one wishes to specialise the underlying equational theory, other approaches, e.g., based on narrowing [8] [1], should be considered. The reason we extend classical partial deduction for SLDE–resolution rather than building on top of [1], is that we actually do not wish to modify the underlying equational theory. As we will see later in the paper, this leads to a simpler theory with simpler correctness results, and also results in a tighter link with classical partial deduction used in [16]. This also means that it is more straightforward to integrate abstract domains as described in [14] (no abstract specialisation exists as of yet for narrowing-based approaches).

In the remainder of the paper, we thus develop a partial deduction method which considers both, equational theories and regular type information. The method will then enable us to solve conjunctive planning problems in the *simple Fluent Calculus*. In particular, we show that our method is actually complete for conjunctive planning problems in the *propositional Fluent Calculus*. We believe that our approach can also be used for more complex systems, without changing much of the algorithm, e.g., in cases where completeness can not be guaranteed due to general undecidability.

2 The Simple Fluent Calculus

The Fluent Calculus $\mathcal{FC}$ is a method for representation and reasoning about action and change [10]. In contrast to the Situation Calculus, states of the world are represented explicitly by terms of sort St. The solution of the frame problem in $\mathcal{FC}$ relies heavily on the use of the equational theory $AC1$ which defines $(St, \circ)$ to be a commutative monoid:

$$\begin{array}{l} \forall(x, y, z : St).\ (x \circ y) \circ z =_{AC1} x \circ (y \circ z) \\ \forall(x, y : St).\ x \circ y =_{AC1} y \circ x \\ \forall(x : St).\ x \circ 1^{\circ} =_{AC1} x \end{array} \qquad (AC1)$$

The operation $\circ$ is used to compose states by combining atomic elements, called *fluents*, which represent elementary propositions. Although in general the Fluent Calculus can be seen as an extension of the Situation Calculus [23], we restrict ourselves here to $\mathcal{FC}$ domains as introduced in [10], since they can be represented as definite logic programs. We call such $\mathcal{FC}$ domains *simple*. In simple $\mathcal{FC}$ domains actions are defined using the predicate $\texttt{action}(\mathcal{C}(\vec{x}), \mathcal{A}(\vec{x}), \mathcal{E}(\vec{x}))$ where $\mathcal{C}(\vec{x})$, $\mathcal{E}(\vec{x})$ are terms of sort St which might depend on variables in $\vec{x}$ and $\mathcal{A}(\vec{x})$ is a term of sort A which has the parameters $\vec{x}$, where the sort A represents the *actions*. Intuitively, executing an action $\mathcal{A}(\vec{x})\theta$ will consume the fluents in $\mathcal{C}(\vec{x})\theta$ and produce the fluents in $\mathcal{E}(\vec{x})\theta$. If all fluents appearing in

terms of type St in predicates `action` of a simple $\mathcal{FC}$ domain are constants, we call the domain *propositional*.

Example 1. (propositional $\mathcal{FC}$ domain) Let Σ_p be the following propositional $\mathcal{FC}$ domain with the fluents $f_1, \ldots, f_5$ and the actions $a_1, \ldots, a_6$.

$\texttt{action}(f_1, a_1, f_2)$. $\quad$ $\texttt{action}(f_3, a_4, f_2)$.
$\texttt{action}(f_1, a_2, f_4)$. $\quad$ $\texttt{action}(f_4, a_5, f_5 \circ f_5)$.
$\texttt{action}(f_2, a_3, f_3 \circ f_3)$. $\quad$ $\texttt{action}(f_5, a_6, f_4)$. □

Example 2. (simple $\mathcal{FC}$ domain) Let Σ_s be the simple $\mathcal{FC}$ domain with the fluents f_1, f_4, f_5, the actions a_2, a_5, a_6 as defined for Σ_p in example 1 and the following predicates for $a_1, a_3(X), a_4(X)$:

$\texttt{action}(f_1, a_1, f_2(0))$.
$\texttt{action}(f_2, a_3(X), f_3(foo(X)) \circ f_3(foo(foo(X))))$.
$\texttt{action}(f_3(X), a_4(X), f_2(X))$. □

The *conjunctive planning problem (CPP)* consists of deciding whether there is a finite sequence of actions such that its execution in a given initial state leads to a state which contains at least, i.e. *covers*, certain goal properties. The initial state and the goal properties are given as conjunctions of fluents, which can be represented as terms of sort St.

In the following, we consider the initial state to be completely known and represented by a ground term St_{init} of sort St.

To describe the execution of action sequences, we define the following predicate which describes all pairs of states, such that the second state can be reached from the first state by executing a finite sequence of actions:

$\texttt{reachable}(S, S)$.
$\texttt{reachable}(C \circ V, T) \leftarrow \texttt{action}(C, A, E) \wedge \texttt{reachable}(V \circ E, T)$.

Note that, in order to keep the representation simple, we do not keep explicitly track of the action sequence. Furthermore, since we propose to use program transformation techniques to solve the CPP, we do not encode the goal in the definition of `reachable/2`.[1] Also note that, for this interpreter to work correctly, it is important that $\circ$ is treated as a commutative and associative function symbol (e.g., $(f \circ g) \circ h$ should unify with $g \circ V$ with unifier $\{V/f \circ h\}$).

To specify and reason about equalities in a standard logical programming environment like Prolog, the particular underlying equational theory (e.g., $AC1$) has to be expressed by appropriate axioms. These axioms often cause trouble, e.g., if the solution to a unification problem is infinite, but it has been shown that equational theories can be successfully built into the unification procedure [20]. To allow a general treatment, SLD–resolution has been extended to SLDE–resolution which uses a universal unification procedure based on the proper ties common to all equational theories [9, 7]. In contrast to other techniques, SLDE–resolution allows to cut down the often tremendous search space by merging equation solving and standard resolution steps. (Narrowing [8] is an efficient approach to solve certain equational theories, and can be integrated as part of the unification into SLDE.)

[1] This is in contrast to [10] where containment of goal properties is encoded in the program.

3 SLDE–resolution

Formally, simple Fluent Calculus domains are (definite) *E-programs* (P, E), i.e. logic programs P with an equational theory E, [9, 7]. An equational theory E is defined as a set of universally closed formulas of the form $\forall(s{=_E}t)$ for some predicate $=_E$ complemented by the standard axioms of equality.[2] Consequently, if $E = \emptyset$ we obtain the *standard equational theory*, i.e. only syntactically identical terms are considered to be equal. In simple Fluent Calculus domains E is given by $AC1$. Many other equational theories have been investigated, see e.g. [21] for a review.

An *E–unification problem* consists of terms s, t and the question whether there exists a substitution σ with $\mathrm{Dom}(\sigma) \subseteq Vars(s) \cup Vars(t)$, s.t. $s\sigma{=_E}t\sigma$. If such a substitution σ exists s and t are called *E–unifiable* with *E–unifier* σ. For example, the terms $V \circ a$ and $a \circ b$ are $AC1$–unifiable with $\{V/b\}$.

A term s is an *E–instance* of a term t, denoted $s{\leq_E}t$, iff there is a substitution σ with $s{=_E}\sigma t$. Similarly, $\theta{\leq_E}\sigma$, for substitutions θ, σ, iff for all terms s: $s\theta{\leq_E}s\sigma$.

Let $U_E(s,t)$ denote the set of all E–unifiers of the terms s and t. Then, $\mathrm{U} \subseteq U_E(s,t)$ is called *complete* if for all $\theta \in U_E(s,t)$ there exists $\sigma \in \mathrm{U}$ and a substitution λ s.t. $\forall x \in Vars(s) \cup Vars(t)$: $x\theta{=_E}x\sigma\lambda$. If U is complete and for all $\theta, \sigma \in \mathrm{U}$, $\theta{\leq_E}\sigma$ implies $\theta = \sigma$, then it is called *minimal*. Correspondingly, an unification algorithm is called *complete* (*minimal*) if, for arbitrary terms s, t, it computes a complete (minimal) set of E–unifiers.

Note that minimal sets of E–unifiers are always unique if they exist. Hence, we denote the minimal E–unifier of s and t by $\mu U_E(s,t)$. We call a substitution in $\mu U_E(s,t)$ a *most general E–unifier* (*mgeu*) of s and t.

Based upon this, one can define the concepts of SLDE–resolution, SLDE–derivations, SLDE–refutations and computed answers in the classical way. One can also define SLDE–trees, where the only difference with SLD–trees is that resolution with a clause can lead to more than one child (as $\mu U_E(s,t)$ may contain more than one substitution)!

SLDE–trees are guaranteed to be finitely branching if the equational theory E is *finitary*, i.e. if the complete set of E unifiers $U_E(s,t)$ is finite. For example, it is well known that the equational theory $AC1$ is finitary.

In [10] it has been shown that SLDE–resolution is sound and complete for CPP in simple $\mathcal{FC}$ domains, i.e. every solution of a CPP is entailed by SLDE–resolution. However, even for propositional $\mathcal{FC}$ domains the SLDE–tree may contain infinite derivations and consequently, the search for a plan may not terminate.

Example 3. (Ex. 1 cont'd) If we repeatedly apply the actions a_3 and a_4 in alternation, we obtain an infinite derivation:

$$\leftarrow \underline{\texttt{reachable}}(f_2, S)$$
$$\leftarrow \underline{\texttt{action}}(f_2, A, E) \wedge \texttt{reachable}(1^\circ \circ E, S) \quad \{A/a_3, E/f_3 \circ f_3\}$$
$$\leftarrow \underline{\texttt{reachable}}(f_3 \circ f_3, S)$$

[2] These are reflexivity, symmetry, transitivity and substitutivity for all function and predicate symbols, respectively.

$\leftarrow \underline{\texttt{action}}(f_3, A', E') \wedge \texttt{reachable}(f_3 \circ E', S)$ $\quad \{A'/a_4, E'/f_2\}$
$\leftarrow \underline{\texttt{reachable}}(f_3 \circ f_2, S)$
$\leftarrow \underline{\texttt{action}}(f_2, A'', E'') \wedge \texttt{reachable}(f_3 \circ E'', S)$ $\quad \{A''/a_3, E''/f_3 \circ f_3\}$
$\ldots$ $\square$

To enable for solving the CPP or similar problems despite of potentially infinite SLDE–derivations we propose to use partial deduction techniques. To this end, we extend the partial deduction method used in [16, 15] to fit SLDE–resolution. Furthermore, we allow conjuncts to carry additional type information, thereby enabling for more precise specialisations.

4 A Partial Deduction Algorithm for E–Programs

The general idea of partial deduction of ordinary logic programs [18] is to construct, given a query $\leftarrow Q'$ of interest, a finite number of finite but possibly incomplete SLD–trees which "cover" the possibly infinite SLD–tree for $P \cup \{\leftarrow Q'\}$ (and thus also all SLD–trees for all instances of $\leftarrow Q'$). The derivation steps in these SLD–trees are the computations which have been pre-evaluated and the clauses of the specialised program are then extracted by constructing one specialised clause (called *resultant*) per branch.

While the initial motivation for partial deduction was program specialisation, one can also use partial deduction as a *top-down flow analysis* of the program under consideration. Indeed, partial deduction will unfold the initial query of interest until it spots a dangerous growth, at which point it will generalise the offending calls and restart the unfolding from the thus obtained more general call. Provided a suitably refined control technique is used (e.g., [17, 4]), one can guarantee termination as well as a precise flow analysis. As was shown in [15] such a partial deduction approach is powerful enough to provide a decision procedure for coverability problems for (reset) Petri nets and bears resemblance to the Karp–Miller procedure [12]. In the context of the CPP, the initial query of interest would be `reachable(init,goal)`, where `init` and `goal` are the initial and the goal state respectively and one would hope to obtain as a result a flow analysis from which it is clear whether the CPP has a solution.

Unfortunately, it has been demonstrated in [15] that "classical" partial deduction techniques may not be precise enough if state descriptions are complex. Similar problems occur if states are represented using non-empty equational theories, since abstractions just based on the "instance-of" relation and the associated most specific generalisation (*msg*) may be too crude (c.f., also [14]).

Example 4. (Ex. 1 cont'd) The *msg* of the atoms $\texttt{reachable}(f_3 \circ f_3, S)$ and $\texttt{reachable}(f_3 \circ f_3 \circ f_3, S)$ is $\texttt{reachable}(f_3 \circ f_3 \circ X, S)$. This is quite unsatisfactory, as X can represent *any* term, i.e., also terms containing other fluents such as f_4. In the context of CPP this means that any action can potentially be executed from $f_3 \circ f_3 \circ X$, and we have thrown away too much information for the generalisation to be useful. For example, if our goal state is f_4, we would not be able to prove that we cannot solve the CPP from the initial state $f_3 \circ f_3$. $\square$

In classical partial deduction there is no way of overcoming this problem, due to its inherent limitation that a call must represent all its instances (the same holds for narrowing-based partial evaluation [1]). Fortunately, this restriction has been lifted, e.g., in the abstract partial deduction framework of [14]. In essence, [14] extends partial deduction and conjunctive partial deduction [4] by working on *abstract conjunctions* on which *abstract unfolding* and *abstract resolution* operations are defined:

- An abstract conjunction is linked to the concrete domain of "ordinary" conjunctions via a concretisation function γ. In contrast to classical partial deduction, γ can be much more refined than the "instance-of" relation. For example, an abstract conjunction can be a couple (Q, τ) consisting of a concrete conjunction Q and some type[3] information τ, and $\gamma((Q, \tau))$ would be all the instances of Q which respect the type information τ. We could thus disallow f_4 to be an instance of X in Ex. 4.
- An abstract unfolding operation maps an abstract conjunction A to a set of *concrete* resultants $H_i \leftarrow B_i$, which have to be totally correct for all possible calls in $\gamma(A)$.
- For each such resultant $H_i \leftarrow B_i$ the abstract resolution will produce an *abstract* conjunction Q_i approximating all the possible resolvent goals which can occur after resolving an element of $\gamma(A)$ with $H_i \leftarrow B_i$.

It is to this framework, suitably adapted to cope with SLDE–resolution, that we turn to remedy our problems. We will actually only consider abstract atoms consisting of a concrete atom together with some type information. The latter will be represented by canonical (deterministic) regular unary logic (RUL) programs [24, 5]. To use a RUL program R in an SLDE–setting it must be ensured that every type t defined by R is "E–closed", i.e. if some term is of type t then all E–equivalent terms are of type t as well.

Definition 1. *A* canonical regular unary clause *is a clause of the form*

$$t_0(f(X_1, \ldots, X_n)) \leftarrow t_1(X_1) \wedge \ldots t_n(X_n)$$

where $n \geq 0$ and $X_1, \ldots, X_n$ are distinct variables. A canonical regular unary logic (RUL) program *is a program R where R is a finite set of regular unary clauses, in which no two different clause heads have a common instance.*

Let E be an equational theory. R is called E–closed *if the least Herbrand model of R and the least Herbrand model of (R, E) are identical.*

The set of ground terms r such that $R \models t(r)$ is denoted by $\tau_R(t)$. A ground term r is of type t in R iff $r \in \tau_R(t)$. Given a (possibly non-ground) conjunction T, we write $R \models \forall(T)$ iff for all ground instances T' of T, $R \cup \{\leftarrow T'\}$ has an SLD–refutation.

So, to solve Ex. 4 one could use the following (E–closed) RUL program, representing all states using just the fluent f_3, and give the variable X in Ex. 4 the type t_3:

$$t_3(1^\circ). \qquad t_3(f_3). \qquad t_3(X \circ Y) \leftarrow t_3(X) \wedge t_3(Y).$$

[3] A type is simply a decidable set of terms closed under substitution.

Given two canonical *RUL*-programs R_1, R_2, there exist efficient procedures for checking inclusion, computing the intersection and computing an upper bound using well known algorithms on corresponding automata [24]. Because of our definition, we can simply re-use the first two procedures to efficiently decide inclusion and compute the intersection of E–closed *RUL* programs. Furthermore, the intersection of two E–closed *RUL* programs is an E–closed *RUL* program. Given two *RUL* programs R_1, R_2 and two types t_1, t_2, we will denote by $(R_1, t_1) \cap (R_2, t_2)$ the couple (R_3, t_3) obtained by the latter procedure (i.e., we have $\tau_{R_3}(t_3) = \tau_{R_1}(t_1) \cap \tau_{R_2}(t_2)$). We will not make use of the upper bound and provide our own generalization mechanism.

Given some *RUL* program R, a *type conjunction (in R)* is simply a conjunction of the form $t_1(X_1) \wedge \ldots \wedge t_n(X_n)$, where all the X_i are variables (not necessarily distinct) and all the t_i are defined in R. We also define the notation $types_T(X) = \{t_j \mid t_j(X) \in T\}$ (where we allow $\in$ to be applied to conjunctions). E.g., $types_{t(X) \wedge t'(Z)}(Z) = \{t'\}$.

We now define the abstract domain used to instantiate the framework of [14]:

Definition 2. *We define the RULE* domain $(\mathcal{AQ}, \gamma, E)$ *to consist of an equational theory E,* abstract conjunctions *of the form $\langle Q, T, R\rangle \in \mathcal{AQ}$ where Q is a concrete conjunction, R an E–closed RUL program, and T a type conjunction in R such that $T = t_1(X_1) \wedge \ldots t_n(X_n)$, where $Vars(Q) = \{X_1, \ldots, X_n\}$.*[4] *The* concretisation function *γ is defined by: $\gamma(\langle Q, T, R\rangle) = \{Q\theta \mid R \models \forall(T\theta)\}$.*

We now define simplification and projection operations for type conjunctions. This will allow us to apply substitutions to abstract conjunctions as well as to define an (abstract) unfolding operation. As the above definition requires every variable to have exactly one type, the type of a variable Z occurring in a substitution such as $\{X/Z, Y/Z\}$ has to be determined by type intersection.

Definition 3. *Let R be some RUL program. The relation $\leadsto_R$ which maps type conjunctions to type conjunctions is defined as follows:*

- $t_1 \wedge t_2 \leadsto_R s_1 \wedge s_2$ *if* $t_1 \leadsto_R s_1$, $t_2 \leadsto_R s_2$, $s_1 \neq fail$, *and* $s_2 \neq fail$
- $t(X) \leadsto_R t(X)$ *if* X *is a variable*
- $t(c) \leadsto_R true$ *if* c *is a constant with* $c \in \tau_R(t)$
- $t(f(r_1, \ldots, r_n)) \leadsto_R s_1 \wedge \ldots \wedge s_n$ *if* $t(f(X_1, \ldots, X_n)) \leftarrow t_1(X_1) \wedge \ldots \wedge t_n(X_n) \in R$ *and* $t_i(r_i) \leadsto_R s_i$
- $t(r) \leadsto_R fail$ *otherwise*

We define a projection *which projects a type conjunction T in the context of a RUL program on a concrete conjunction Q, resulting in new abstract conjunction: $proj(Q, T, R) = \langle Q, S', R'\rangle$, where $T \leadsto_R S$ and*

- $S' = S$, $R' = \emptyset$ *if* $S = fail$ *or* $Vars(Q) = \emptyset$,
- *otherwise* $S' = t_1(X_1) \wedge \ldots \wedge t_n(X_n)$ *where* $Vars(Q) = \{X_1, \ldots, X_n\}$, $types_S(X_i) = \{t_{i_1}, \ldots, t_{i_k}\}$, $(R_i, t_i) = (R, t_{i_1}) \cap \ldots \cap (R, t_{i_k})$. *In this case* $R' = R_1 \cup \ldots \cup R_n$.

[4] Note that when writing, e.g., $Vars(Q) = \{X_1, \ldots, X_n\}$ all X_i are assumed to be distinct.

We now define applying substitutions on abstract conjunctions: $\langle Q,T,R\rangle\theta = proj(Q\theta,T,R)$.

For example, using the *RUL* program R above, we have $t_3(f_3 \circ Z \circ Z) \leadsto_R true \wedge t_3(Z) \wedge t_3(Z)$. We would thus have for $\theta = \{X/(f_3 \circ Z \circ Z)\}$ that $\langle p(X), t_3(X), R\rangle\theta = \langle p(f_3 \circ Z \circ Z), t_3(Z), R\rangle$.

To extend the notion of instantiation preorder to abstract conjunctions the subset relation between types has to be considered:

Definition 4. *Let* $A = \langle Q,T,R\rangle$, $A' = \langle Q',T',R'\rangle$ *be abstract conjunctions in the RULE domain* $(\mathcal{AQ},\gamma,E)$. *We call* A' *a* RULE–instance *of* A, *denoted by* $A' \leq_{RULE} A$ *iff*

1. *there exists a substitution* θ *such that* $A\theta = \langle Q',T'',R''\rangle$ *and*
2. *for all* $X \in Vars(Q')$ *with* $types_{T'}(X) = \{t'\}$ *and* $types_{T''}(X) = \{t''\}$, *we have* $\tau_{R'}(t') \subseteq \tau_{R''}(t'')$.

We define $<_{RULE}$ *and* $=_{RULE}$ *accordingly.*

In the above example, $\langle p(f_3 \circ Z \circ Z), t_3(Z), R\rangle <_{RULE} \langle p(X), t_3(X), R\rangle$.

Definition 5. *An* unfolding rule *is a function which, given a definite* E*–program* (P,E) *and a goal* $\leftarrow Q$, *returns a non-trivial*[5] *and possibly incomplete SLDE–tree for* (P,E) *and* $\leftarrow Q$.

Let τ *be a finite (possibly incomplete) SLDE–tree for* (P,E), $\leftarrow Q$. *Let* $\leftarrow G_1,\ldots,\leftarrow G_m$ *be the goals in the leaves of the non-failing branches of* τ. *Let* $\theta_1,\ldots,\theta_n$ *be the computed answer substitutions of the SLDE–derivations from* $\leftarrow Q$ *to* $\leftarrow G_1,\ldots,\leftarrow G_n$, *respectively. Then the set of* SLDE–resultants, *resultants*(τ), *is defined to be the set of clauses* $\{Q\theta_1 \leftarrow G_1,\ldots,Q\theta_n \leftarrow G_n\}$.

We can now define an *abstract unfolding* and an *abstract resolution* in the *RULE* domain. When a conjunction of the *RULE* domain is unfolded, the information concerning the types of variables can be used to reduce the number of resultants. Additionally, we will use Def. 3 to determine the types of leaf conjunctions.

Definition 6. *Let* (P,E) *be a definite* E*–program,* $\langle Q,T,R\rangle$ *an abstract conjunction in the RULE domain* $(\mathcal{AQ},\gamma,E)$, U *an unfolding rule. We define the abstract unfolding and resolution operations* $aunf(.), ares(.)$ *as follows:*

- $aunf(\langle Q,T,R\rangle) = \{Q\theta \leftarrow B \mid Q\theta \leftarrow B \in resultants(U(Q)) \wedge T\theta \not\leadsto_R fail\}$
- $ares(\langle Q,T,R\rangle) = \{proj(B,T\theta,R) \mid Q\theta \leftarrow B \in aunf(\langle Q,T,R\rangle)\}$

The following is a generic algorithm for abstract partial deduction, which structures the abstract conjunctions to be specialised in a *global tree* (see, e.g., [17]), and is parametrised by a covering test *covered*, a *whistle* detecting potential infinite loops, an a generalisation operation *abstract* and a function *partition* to separate conjunctions into sub-conjunctions.

[5] A trivial SLDE–tree has a single node where no literal has been selected for resolution.

Algorithm 4.1 (*generic partial deduction algorithm*)
Input: a definite E–program (P, E), an abstract conjunction A in $(\mathcal{AQ}, \gamma, E)$.
Output: a set of abstract conjunctions $\mathcal{A}$, a specialised program P, a global tree λ
Initialisation: $\lambda :=$ a "global" tree with a single unmarked node, labelled by A
repeat
  pick an unmarked or abstracted leaf node L in λ
  if $covered(L, \lambda)$ **then** mark L as processed
  else
    if $whistle(L, \lambda) = \mathtt{T}$ **then**
      mark L as abstracted
      $label(L) := abstract(L, \lambda)$
    else
      mark L as processed
      for all $A \in ares(label(L))$ **do**
        for all $A' \in partition(A)$ **do**
          add a new unmarked child C of L to λ
          $label(C) := A'$
until all nodes are processed
output $\mathcal{A} := \{label(A) \mid A \in \lambda\}$, $P := \{aunf(A) \mid A \in \mathcal{A}\}$, and λ

Algorithm 4.2 (a partial deduction algorithm for the Fluent Calculus) We define a particular instance of the above algorithm as follows:

Unfolding used by $aunf(.)$: Let $\langle Q, T, R\rangle$ be an abstract conjunction in the $RULE$ domain and (P, E) be a definite E–program. We define $U(Q)$ to be the maximal SLDE–tree τ such that every predicate p is selected at most once in every branch of τ.

covered Let L be a node labelled by an abstract conjunction in the $RULE$ domain $(\mathcal{AQ}, \gamma, E)$ and λ a tree labeled by elements of $\mathcal{AQ}$. Then we define $covered(L, \lambda)$ as true iff there is an ancestor L' of L such that $label(L') =_{RULE} label(L)$.

whistle We extend the well-established homeomorphic embedding relation [22], to take regular types and the $AC1$ equational theory into account. To simplify the presentation, we use $\circ$ as a variable arity functor to represent terms of sort St and disallow the use of 1° and nesting of $\circ$ (e.g., we represent $a \circ ((b \circ 1^\circ) \circ c)$ by $\circ(a, b, c)$ and $1^\circ \circ 1^\circ$ by $\circ()$).

Definition 7. *Let $A = \langle Q, T, R\rangle, A' = \langle Q', T', R'\rangle$ be abstract conjunctions in the $RULE$ domain $(\mathcal{AQ}, \gamma, E)$. For the purposes of this definition we suppose that $\wedge$ is handled by E as an associative and commutative function symbol. We say that A is homeomorphically embedded in A', $A \trianglelefteq_E A'$, iff $Q \trianglelefteq_E Q'$ where $\trianglelefteq_E$ on expressions is inductively defined as follows:*

1. *$X \trianglelefteq_E Y$ if X, Y variables with $\tau_R(types_T(X)) \subseteq \tau_{R'}(types_{T'}(Y))$*
2. *$r \trianglelefteq_E Y$ for all variables Y and ground terms r, with $r \in \tau_R(types_{T'}(Y))$*
3. *$r \trianglelefteq_E f(s_1, \ldots, s_n)$ if $f \neq \circ$ and $r \trianglelefteq_E s_i$ for some $1 \leq i \leq n$*
4. *$f(r_1, \ldots, r_n) \trianglelefteq_E f(s_1, \ldots, s_n)$ if $f \neq \circ$ and $\forall i \in \{1, \ldots, n\} : r_i \trianglelefteq_E s_i$.*
5. *$\circ(r_1, \ldots, r_m) \trianglelefteq_E \circ(s_1, \ldots, s_n)$ if there exists a permutation $s'_1, \ldots, s'_n$ of $s_1, \ldots, s_n$ such that $\forall i \in \{1, \ldots, m\} : r_i \trianglelefteq_E s'_i$.*

Note that for point 3. we may have $n = 0$, and for point 5. m, n can be 0. Intuitively, $s \trianglelefteq_E t$, means that we can obtain s from t by "striking out" certain sub-terms and by using the equational theory to re-write s and t. E.g., we have $f(0) \circ g \trianglelefteq_E g \circ h \circ f(s(0))$. In general, of course, $\trianglelefteq_E$ will be quite expensive to compute ([19]). However, one can introduce a lot of optimisations to obtain an efficient implementation (sorting fluents and defining a normal form for terms; one can also always use the classical homeomorphic embedding which ignores E).

The homeomorphic relation is a well-quasi order (provided $\subseteq$ is a well-quasi order on the possible regular types of variables; see below) and can thus be used to ensure termination of program specialisation techniques [22]. We use $\trianglelefteq_E$ as follows. Let L be a node labelled by an abstract conjunction in the *RULE* domain $(\mathcal{AQ}, \gamma, E)$ and λ a tree labeled by elements of $\mathcal{AQ}$. We define $whistle_{\trianglelefteq_E}(L, \lambda) =$ T iff L is not marked as abstracted and there is an ancestor L' of L such that $label(L') \trianglelefteq_E label(L)$.

abstract To ensure that abstractions of types may occur only finitely often, we require the use of a well founded type system.

Definition 8. *Let E be an equational theory and $\mathcal{T}$ a set of tuples (R, t) where R is a RUL program and t a predicate defined in R. We call $\mathcal{T}$ a* well founded type system *iff there is no infinite sequence $(R_1, t_1), (R_2, t_2), \ldots$ of elements of $\mathcal{T}$ such that $\tau_{t_i}(R_i) \subset \tau_{t_{i+1}}(R_{i+1})$ for all $i \geq 1$.*

Definition 9. *Let E be an equational theory, $\mathcal{T}$ be a well-founded type system and $\mathcal{A}$ a set of abstract conjunctions in $(\mathcal{AQ}, \gamma, E)$. The abstract conjunction $M = \langle Q, T, R\rangle$ is called a* RULE–generalisation *of $\mathcal{A}$ wrt $\mathcal{T}$ iff*

1. *for all $t(X) \in T$ we have $(t, R) \in \mathcal{T}$,*
2. *for all $A \in \mathcal{A}$, $A \leq_{RULE} M$.*

Furthermore, M is called a most specific *RULE–generalisation of $\mathcal{A}$ wrt $\mathcal{T}$, denoted by $M \in msg_{\mathcal{T}}(\mathcal{A})$, iff there exists no M' such that conditions 1, 2 hold for M' and $M' <_{RULE} M$.*

For example, $\mathcal{A} = \{\langle f_3, true, \emptyset\rangle, \langle f_3 \circ f_3, true, \emptyset\rangle\}$ and using the single type defined by the *RUL* program for t_3 we get $msg_{\mathcal{T}}(\mathcal{A}) = \{\langle f_3 \circ X, t_3(X), R\rangle\}$.

Again, for other equational theories than $(AC1)$ and more complicated type systems a most specific generalisation might be difficult to compute (and may not be unique). To accelerate convergence (and to simplify our completeness proof for CPP later on), we actually choose an element $M' = \langle Q, T, R\rangle$ of $msg_{\mathcal{T}}(\mathcal{A})$ and then remove the maximum number of subterms from Q so that the resulting abstract conjunction is still more general than M' (in the sense of $\leq_{RULE}$). We will denote the result by $nmsg_{\mathcal{T}}(\mathcal{A})$. For example, we would instead of using $M' = \langle f_3 \circ X, t_3(X), R\rangle$ use the more general $nmsg_{\mathcal{T}}(\mathcal{A}) = \langle X, t_3(X), R\rangle$. This loses some precision, but convergence is accelerated, and actually no vital information for the CPP is lost!

Let L be a node labelled by an abstract conjunction in the *RULE* domain $(\mathcal{AQ}, \gamma, E)$ and λ a tree labeled by elements of $\mathcal{AQ}$. Let $\mathcal{L}$ denote the set of all ancestors of L in λ such that $L' \in \mathcal{L}$ iff $label(L') \trianglelefteq_E label(L)$. Furthermore, let $\mathcal{A}$ denote the set of labels of $\mathcal{L}$. Then we define $abstract(L, \lambda) = nmsg_{\mathcal{T}}(\mathcal{A})$.

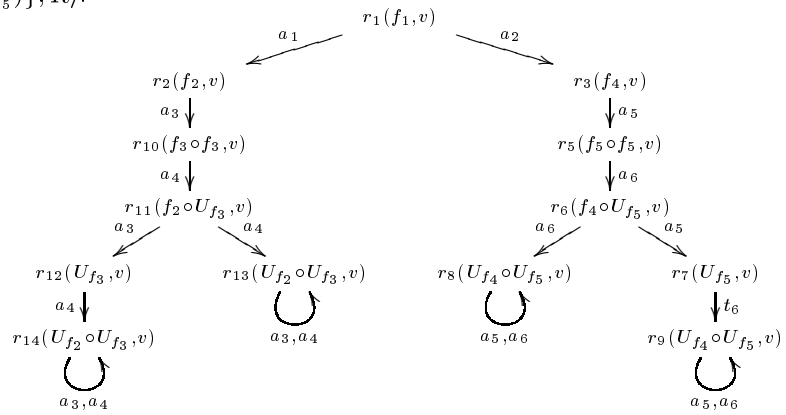
partition Let $A = \langle Q, T, R\rangle$ be an abstract conjunction in $(\mathcal{AQ}, \gamma, E)$ and $\mathcal{T}$ a well-founded type system. We define $partition(A) = atoms(nmsg_{\mathcal{T}}(\{A\}))$.

Example 5. (Ex. 1 cont'd) Additionally to the actions of Ex. 1 and the domain independent `reachable/2`, let the initial state be defined as $St_{init} = f_1$.

In this example every abstract conjunction $C \in \mathcal{AQ}$ will be of the form $\langle$`reachable(`u`,`v`)`$, T, R\rangle$ where $v = V_{f_1} \circ \ldots \circ V_{f_5}$ and $\forall 1 \leq i \leq 5 : t_{f_i}(V_{f_i}) \in T$ (representing that $f_1, \ldots, f_5$ may occur arbitrarily often in the final state). Furthermore, u is of sort St where $U \in Vars(u) \Rightarrow \exists 1 \leq i \leq 5$ s.t. $t_{f_i}(U) \in T$. Finally, R consists of predicates (t_{f_i}) for each fluent f_i:

$$t_{f_i}(1^\circ). \qquad t_{f_i}(f_i). \qquad t_{f_i}(X \circ Y) \leftarrow t_{f_i}(X) \wedge t_{f_i}(Y).$$

We define the type system as $\mathcal{T} = \{(R, t_{f_i}) \mid 1 \leq i \leq 5\}$. Then, the following tree is generated by our partial deduction algorithm with input Σ_p and initial abstract conjunction $\langle$`reachable(`St_{init}`,`$V_{f_1} \circ \ldots \circ V_{f_5}$`)`$, \{t_{f_1}(V_{f_1}) \wedge \ldots \wedge t_{f_5}(V_{f_5})\}, R\rangle$:



To simplify the picture the *RUL* programs and type conjunctions have not been represented. The *RUL* programs do not change in this example and the type information has been depicted as follows: v represents $V_{f_1} \circ \ldots \circ V_{f_5}$ and the type conjunction T_j for each node $r_j(u, v)$ contains atoms $t_{f_i}(V_{f_i})$, $i = 1, \ldots, 5$, and $t_{f_i}(U_{f_i})$ if they are used. t_{f_i} is defined by the corresponding *RUL* program (t_{f_i}). Finally, $r_j(u, v)$ is the jth node with label $\langle$`reachable(`u`,`v`)`$, T_j, R\rangle$.

For example, we can conclude from the tree that every fluent can be generated arbitrarily often. But, e.g., it is impossible to reach a state containing both, f_2 and f_4. □

Example 6. (Ex. 2 cont'd) Additionally to the actions defined in Ex. 2 and the domain independent `reachable/2`, let the initial state be defined as $St_{init} = f_1$.

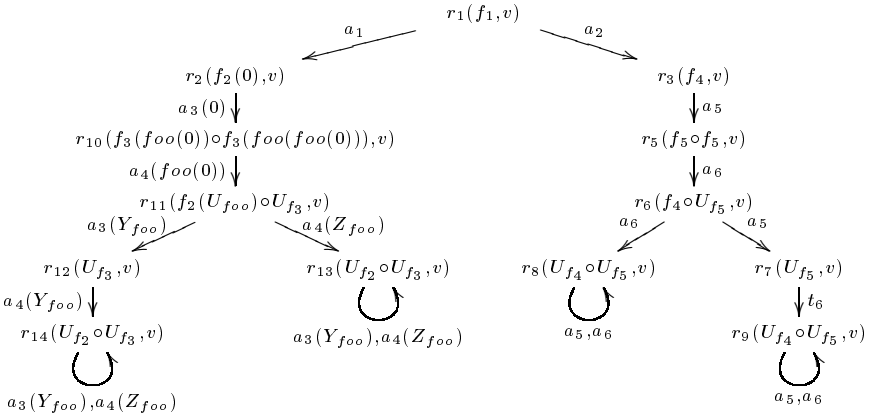
In this example every abstract conjunction $C \in \mathcal{AQ}$ will be of the form $\langle$`reachable(`u`,`v`)`$, T, R\rangle$ where again $v = V_{f_1} \circ \ldots \circ V_{f_5}$ and $t_{f_i}(V_{f_i}) \in T$. Also, u is of sort St where for all $U \in Vars(u)$ either $\exists 1 \leq i \leq 5$ s.t. $t_{f_i}(U) \in T$ or

$t_{foo}(U) \in T$. R contains (t_{f_i}) of Ex. 5 for $i = 1, 4, 5$, and for $i = 2, 3$ and t_{foo}, respectively:

$t_{f_i}(1^\circ)$.
$t_{f_i}(f_i(X))$:- $t_{foo}(X)$.
$t_{f_i}(Y \circ X)$:- $t_{f_i}(Y)$, $t_{f_i}(X)$.

$t_{foo}(0)$.
$t_{foo}(foo(X))$:- $t_{foo}(X)$.

We define the type system $\mathcal{T} = \{(R, t_{f_i}) \mid 1 \leq i \leq 5\} \cup \{(R, t_{foo})\}$. Then, the following tree is generated by our partial deduction algorithm with input Σ_s and initial abstract conjunction $\langle \texttt{reachable}(St_{init}, V_{f_1} \circ \ldots \circ V_{f_5}), \{t_{f_1}(V_{f_1}) \wedge \ldots \wedge t_{f_5}(V_{f_5})\}, R\rangle$:



Again, to simplify the picture the *RUL* programs and type conjunctions have not been represented. *RUL* programs do not change in this example and the type information has been depicted as follows: v represents $V_{f_1} \circ \ldots \circ V_{f_5}$ and the type conjunction T_j for each node $r_j(u, v)$ contains atoms $t_{f_i}(V_{f_i})$, $i = 1, \ldots, 5$, and $t_{f_i}(U_{f_i})$, $t_{foo}(U_{foo})$ if they are used. t_{f_i}, t_{foo} is defined by the corresponding *RUL* programs (t_{f_i}) and (t_{foo}). Finally, $r_j(u, v)$ is the jth node with label $\langle \texttt{reachable}(u, v), T_j, R\rangle$.

For example, we can conclude from the tree that it is possible to generate a state containing arbitrary many instances of the fluent f_2. But we cannot conclude whether we can generate a state containing arbitrary many copies of one particular instance of f_2. □

5 Completeness wrt. $\mathcal{FC}_{PL}$

In [13] it has been shown that Petri net algorithms can be used to decide temporal properties of propositional Fluent Calculus domains. In particular, to every propositional $\mathcal{FC}$ domain with completely defined initial state exists a bisimilar Petri net. Furthermore, the conjunctive planning problem for the propositional $\mathcal{FC}$ can be expressed as a formula in the temporal logic CTL (CTL respects bisimulation). The same formula is known to describe *coverability* properties of Petri nets. Coverability problems can be decided using the Karp-Miller tree [12]. This tree can also be generated by the partial deduction algorithm 4.1 using the instantiations of section 4. By doing so, we show that the proposed partial deduction method is complete wrt. conjunctive planning problems.

Theorem 1. *Let Σ be a propositional $\mathcal{FC}$ domain, Δ_Σ the RULE domain defined as in example 5 and St_{init} some ground term of sort St. Then the partial deduction algorithm applied to Σ, Δ_Σ and $A = \langle \texttt{reachable}(St_{init}, V_{f_1} \circ \ldots \circ V_{f_n}), \{t_{f_1}(V_{f_1}) \wedge \ldots \wedge t_{f_n}(V_{f_n})\}, R\rangle$ will produce a global tree λ which is isomorphic to a Karp-Miller coverability tree of the corresponding Petri net Π.*

6 Conclusion

We have presented a generic and a more specific abstract partial deduction method for equational logic programs, based upon an abstract domain with regular types. This is one of the first full instantiations of the framework in [14] (see also the independently developed [6]). The main motivation was to obtain a useful method for tackling the conjunctive planning problem in the fluent calculus, stimulated by earlier success of partial deduction for solving coverability problems in the Petri net area. We were able to prove that our more specific method *is* a decision procedure for the conjunctive planning problem in the propositional fluent calculus. However, the method can also be applied to more expressive fragments of the fluent calculus or extended to cope with other formalisms such as process algebras (where, contrary to Petri nets, type information is also vital), and we believe that it will be able to provide useful results in that setting. Finally, the methods can of course also be used to *specialise* fluent calculus descriptions, and can also be applied to "ordinary" logic programs, where the additional precision of the regular types should pay off in terms of improved specialisation. In the future we hope to produce a full-fledged implementation to test these claims.

References

1. M. Alpuente, M. Falaschi, and G. Vidal. Partial evaluation of functional logic programs. *ACM Trans. Program. Lang. Syst.*, 20(4):768–844, 1998.
2. M. Bruynooghe, H. Vandecasteele, D. A. de Waal, and M. Denecker. Detecting unsolvable queries for definite logic programs. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of ALP/PLILP'98*, LNCS 1490, pages 118–133. Springer, 1998.
3. D. Chapman. Planning for conjunctive goals. *AIJ*, 32(3):333–377, 1985.
4. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *J. Logic Program.*, 41(2 & 3):231–277, 1999.
5. J.P. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of ICLP'94*, pages 599–613. The MIT Press, 1994.
6. J. P. Gallagher and J. C. Peralta. Using regular approximations for generalisation durign partial evaluation. In *Proceedings of PEPM'00*, pages 44–51. ACM Press.
7. J. H. Gallier and S. Raatz. Extending SLD resolution to equational horn clauses using E-unification. *J. Logic Program.*, 6(1-2):3–43, 1989.
8. M. Hanus. The integration of functions into logic programming. *J. Logic Program.*, 19 & 20:583–628, May 1994.
9. S. Hölldobler. *Foundations of Equational Logic Programming*, LNAI 353. Springer, 1989.

10. S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Gen. Comput.*, 8:225–244, 1990.
11. S. Hölldobler and M. Thielscher. Computing change and specificity with equational logic programs. *Annals of Mathematics and Artificial Intelligence*, 14:99–133, 1995.
12. R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.
13. H. Lehmann and M. Leuschel. Decidability results for the propositional fluent calculus. In J. Lloyd et al., editor, *Proceedings of CL'2000*, LNAI 1861, pages 762–776, London, UK, 2000. Springer.
14. M. Leuschel. Program specialisation and abstract interpretation reconciled. In Joxan Jaffar, editor, *Proceedings of JICSLP'98*, pages 220–234, Manchester, UK, 1998. MIT Press.
15. M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In J. Lloyd et al., editor, *Proceedings of CL'2000*, LNAI 1861, pages 101–115, London, UK, 2000. Springer.
16. M. Leuschel and H. Lehmann. Solving coverability problems of Petri nets by partial deduction. In *Proceedings of PPDP'2000*. ACM Press, 2000. To appear.
17. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Trans. Program. Lang. Syst.*, 20(1):208–258, 1998.
18. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *J. Logic Program.*, 11(3&4):217–242, 1991.
19. R. Marlet. *Vers une Formalisation de l'Évaluation Partielle*. PhD thesis, Université de Nice - Sophia Antipolis, 1994.
20. G. Plotkin. Building in equational theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, number 7, pages 73–90, Edinburgh, Scotland, 1972. Edinburgh University Press.
21. J. H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7(3–4):207–274, 1989.
22. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. Lloyd, editor, *Proceedings of ILPS'95*, pages 465–479, Portland, USA, 1995. MIT Press.
23. M. Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *AIJ*, 111(1–2):277–299, 1999.
24. E. Yardeni and E. Shapiro. A type system for logic programs. *J. Logic Program.*, 10(2):125–154, 1990.

A Completeness of Partial Deduction

Definition 10. *A* Petri net *Π is a tuple (S, T, F, M_0) consisting of a finite set of places S, a finite set of transitions T with $S \cap T = \emptyset$ and a flow relation F which is a function from $(S \times T) \cup (T \times S)$ to $\mathbb{N}$. A* marking *m for Π is a mapping $S \mapsto \mathbb{N}$. M_0 is a marking called* initial.

A transition $t \in T$ is enabled *in a marking M iff $\forall s \in S : M(s) \geq F(s,t)$. An enabled transition can be fired, resulting in a new marking M' defined by $\forall s : M'(s) = M(s) - F(s,t) + F(t,s)$. We will denote this by $M[t\rangle M'$. By $M[t_1, \ldots, t_k\rangle M'$ we denote the fact that for some intermediate markings $M_1, \ldots, M_{k-1}$ we have $M[t_1\rangle M_1$, $\ldots$, $M_{k-1}[t_k\rangle M'$.*

We define the reachability tree $RT(\Pi)$ inductively as follows: Let M_0 be the label of the root node. For every node n of $RT(\Pi)$ labelled by some marking

M and for every transition t which is enabled in M, add a node n' labelled M' such that $M[t\rangle M'$ and add an arc from n to n' labelled t. The set of all labels of $RT(\Pi)$ is called the reachability set *of Π, denoted $RS(\Pi)$.*

For convenience, we denote $M \geq M'$ iff $M(s) \geq M'(s)$ for all places $s \in S$. We also introduce *pseudo-markings*, which are functions from S to $\mathbb{N} \cup \{\omega\}$ where we also define $\forall n \in \mathbb{N} : \omega > n$ and $\omega + n = \omega - n = \omega + \omega = \omega$. Using this we also extend the notation $M_{k-1}[t_1, \ldots, t_k\rangle M'$ for such markings.

Many interesting properties of Petri nets can be investigated using the so-called *Karp-Miller tree* resulting of the following algorithm[6], first defined in [12]. The Karp-Miller tree is a finite abstraction of the set of reachable markings $RS(\Pi)$ with which we can decide whether it is possible to "cover" some arbitrary marking M' (i.e., $\exists M'' \in RT(\Pi) \mid M'' \geq M'$) simply by checking whether a node in the tree covers M'.

Algorithm A.1 (*Karp–Miller–Tree*)

Input: a Petri net $\Pi = (S, T, F, M_0)$
Output: a tree $KM(\Pi)$ of nodes labelled by pseudo-markings
Initialisation: set $U := \{node(r, M_0)\}$ of unprocessed nodes
while $U \neq \emptyset$
    select some $(k, M) \in U$;
    $U := U \setminus \{(k, M)\}$;
    if there is no ancestor node (k_1, M_1) of (k, M) with $M = M_1$ **then**
        $M_2 = M$;
        for all ancestors (k_1, M_1) of (k, M) such that $M_1 < M$ **do**
            for all places $p \in S$ such that $M_1(p) < M(p)$ **do** $M_2(p) = \omega$;
        $M := M_2$;
        for every transition t such that $M[t\rangle M'$ **do**
            create node (k', M');
            create arc labelled t from (k, M) to (k', M');
            $U := U \cup (k', M')$;

We will now formally prove that the algorithm 4.1 with the instantiation of section 4.2 can be used to decide coverability problems. For this we need to establish a link between pseudo-markings in the Karp-Miller tree and abstract conjunctions produced by partial deduction.

Let Σ be some propositional $\mathcal{FC}$ domain. Let $F_\Sigma = \{f_1, \ldots, f_n\}$ be the fluents and $A_\Sigma = \{a_1, \ldots, a_m\}$ the actions defined in Σ. We define $|t, f|$ as the number of occurrences of $f \in F_\Sigma$ in the ground term t. According to [13], the Petri net (S, T, F, M_0) corresponding to a propositional $\mathcal{FC}$ domain Σ is given by associating an unique place $S(f) \in S$ to each $f \in F_\Sigma$. Every clause `action(`$\mathcal{C}$`,`a`,`$\mathcal{E}$`)` in Σ with $a \in A_\Sigma$ is associated a transition $T(a) \in T$. The flow relation is defined by $F(T(a), S(f)) = |\mathcal{C}, f|$ and $F(S(f), T(a)) = |\mathcal{E}, f|$ for every $f \in F_\Sigma$ and $a \in A_\Sigma$. Let Δ_Σ be the $RULE$ domain $(\mathcal{AQ}, \gamma, AC1)$ where every abstract conjunction $C \in \mathcal{AQ}$ is of the form $\langle$`reachable(`u`,`v`)`$, H, R\rangle$ where $v = V_{f_1} \circ \ldots \circ V_{f_n}$, u are terms of sort St and for all variables $U_f \in Vars(u)$, where $f \in F_\Sigma$, $t_f(U) \in H$ and for all $V_f \in Vars(v)$, where $f \in F_\Sigma$, $t_f(V) \in H$.

[6] The algorithm presented here differs slightly from the original.

R consists of the predicates (t_f) for each fluent $f \in F_\Sigma$ as defined in example 5 for (t_{f_i}). Then, we define the pseudo-marking C^μ for each $f \in F_\Sigma$:

$$C^\mu(f) = \begin{cases} \omega & \text{if} \quad X \in Vars(u) \wedge t_f(X) \in H \\ |u, f| & \text{otherwise} \end{cases}$$

Accordingly, the initial marking corresponding to some St_{init} is given by

$$\langle \texttt{reachable}(St_{init}, V_{f_1} \circ \ldots \circ V_{f_n}), \{t_{f_1}(V_{f_1}) \wedge \ldots \wedge t_{f_n}(V_{f_n})\}, R \rangle^\mu$$

Additionally, we associate with every pseudo-marking M and RUL program R as defined above an abstract conjunction $M^\alpha = \langle \texttt{reachable}(u, v), H, R \rangle$ s.t. for every fluent $f \in F_\Sigma$, the term u contains f exactly $M(S(f))$ times if $M(S(f)) \neq \omega$. For every $f \in F_\Sigma$ with $M(S(f)) = \omega$, u contains a variable X and H a type declaration $t_f(X)$, and $v = V_{f_1} \circ \ldots \circ V_{f_n}$ with $t_{f_i}(V_{f_i}) \in H$ for all $1 \leq i \leq n$.

To prove that the tree generated by our PD algorithm is isomorphic to the Karp-Miller tree, we use the following propositions establishing links between the algorithms 4.2 and A.1.

Lemma 1. *Let L_1, L_2 be some nodes of the tree λ which is labelled by abstract conjunctions of Δ_Σ. Let $C_1 = \langle \texttt{reachable}(u_1, v), T_1, R \rangle = label(L_1)$ and $C_2 = \langle \texttt{reachable}(u_2, v), T_2, R \rangle = label(L_2)$. Then $C_1 =_{RULE} C_2$ iff ${C_1}^\mu = {C_2}^\mu$.*

Proof. This follows using the mappings $\_^\alpha$ and $\_^\mu$ between markings and abstract conjunctions as defined above and the fact that $(C^\mu)^\alpha =_{RULE} C$ for markings $C = \langle \texttt{reachable}(u, v), T, R \rangle$: from the definition, $C_1 =_{RULE} C_2$ iff for all fluents f holds either 1. the number of f in u_1 and u_2 must be equal, or 2. there are variables X in u_1 and Y in u_2 s.t. $t_f(X) \in T_1$ and $t_f(Y) \in T_2$. □

Lemma 2. *Let L be some node of the tree λ which is labelled by abstract conjunctions of Δ_Σ. Let $C = \langle \texttt{reachable}(u, v), T, R \rangle = label(L)$ and $C_0, C_1, \ldots, C_n$ is the sequence of labels of the ancestors of L in λ where C_0 is the label of the root node. $whistle(L, \lambda) = \texttt{T}$ iff there is some L_k labelled C_k, $0 \leq k \leq n$, with ${C_k}^\mu \leq C^\mu$.*

Proof. According to the definition, *whistle* returns T iff there is some ancestor L_k of L labelled C_k s.t. $C_k \trianglelefteq_E C$. If u does not contain a variable of type t_f for fluent $f \in F_\Sigma$ and $C_k \trianglelefteq_E C$, then by case 5 of the definition of $\trianglelefteq_E$ follows ${C_k}^\mu(S(f)) \leq C^\mu(S(f))$. Otherwise, by case 1 follows ${C_k}^\mu(S(f)) \leq C^\mu(S(f))$ if C_k contains a variable of type t_f as well, or by case 2, ${C_k}^\mu(S(f)) \leq C^\mu(S(f))$ if C_k contains any number of copies of f. Note that due to the use of *nmsg* a label C' in λ may never contain both, copies of f and a variable of type t_f. Now, let $C_k = \langle \texttt{reachable}(u_k, v_k), T_k, R \rangle$ be an abstract conjunction in Δ_Σ and ${C_k}^\mu(S(f)) \leq C^\mu(S(f))$ for all fluents f. If C_k contains a variable of type t_f, case 1 of $\trianglelefteq_E$ applies iff C contains an appropriate variable, i.e. iff $C^\mu(S(f)) = \omega$. Otherwise, if C_k contains copies of f then case 2 applies iff $C^\mu(S(f)) = \omega$ and case 5 applies iff ${C_k}^\mu(S(f)) \leq C^\mu(S(f)) \neq \omega$. □

Lemma 3. *Let L be some node of the tree λ which is labelled by abstract conjunctions of Δ_Σ. Let $C = \langle \texttt{reachable}(u, v), T, R \rangle = label(L)$ and $\{C_1, \ldots, C_n\}$ is the sequence of labels of ancestors of L in λ s.t. $C_i \trianglelefteq_E C$ for all $1 \leq i \leq n$. Let $\mathcal{T}$ consist of all pairs (t, R) s.t. t is a predicate in R. $C' = abstract(L, \lambda)$ iff ${C'}^\mu = M'$ and M' is defined as follows: if for some fluent f there exists an ancestor C_k of C in λ s.t. ${C_k}^\mu < C^\mu$ and ${C_k}^\mu(f) < C^\mu(f)$, $M'(f) = \omega$, otherwise $M'(f) = C^\mu(f)$.*

Proof. Note that $\mathcal{T}$ is a finite set s.t. for any two $(R,t_1),(R,t_2) \in \mathcal{T}$, $\tau_R(t_1) \cap \tau_R(t_2) = \emptyset$. Furthermore, every $\tau_R(t)$ with $(R,t) \in \mathcal{T}$ consists only of terms constructed by combining copies of one particular fluent $f \in F_\Sigma$ using $\circ$. Hence, from the definition of *abstract* and $nmsg_\mathcal{T}$ follows that C' contains a variable of type t_f iff there is some ancestor L_k of L labelled C_k s.t. $C_k \trianglelefteq_E C$ and ${C_k}^\mu(f) < C^\mu(f)$: on one hand, condition 2 in definition 9 ensures that C and C_k are both instances of $C' = \langle \texttt{reachable(}u'\texttt{,}v\texttt{)}, T', R\rangle$. This can only be the case if ${C_k}^\mu(f) < {C'}^\mu(f)$ and $C^\mu(f) \leq {C'}^\mu(f)$ and hence, u' must contain a variable of type t_f representing ω. Furthermore, from definition of $nmsg_\mathcal{T}$, a fluent f must not occur in u' if u' contains a variable of type t_f. On the other hand, from definition 9 follows, that u' must not contain a variable of type t_f if ${C_k}^\mu(f) = C^\mu(f)$ for all ancestors with label C_k and ${C_k}^\mu < C^\mu$. In this case, since $C \trianglelefteq_E C'$, the same number of copies of fluent f occurs in u' as in u. □

Lemma 4. *Let L_1, L_2 be some nodes of the tree λ which is labelled by abstract conjunctions of Δ_Σ. Let $C_1 = \langle \texttt{reachable(}u_1\texttt{,}v\texttt{)}, T_1, R\rangle = label(L_1)$ and $C_2 = \langle \texttt{reachable(}u_2\texttt{,}v\texttt{)}, T_2, R\rangle = label(L_2)$. $C_2 \in partition(ares(aunf(L_1)))$ iff there is an action $\mathcal{A}$ s.t. ${C_1}^\mu[T(\mathcal{A})\rangle {C_2}^\mu$.*

Proof. The procedures *ares*() and *aunf*() can be simplified, since a variable may never have two or more types, conjunctions of types do not have to be computed. *ares*() and *aunf*() unfold and ensure type of variables, only. According to the used unfolding rule an atom $\texttt{reachable(}u_1\texttt{,}v\texttt{)}$ is unfolded s.t. every occuring predicate is unfolded once, i.e. into the subgoals $\texttt{action(}\mathcal{C}\texttt{,}\mathcal{A}\texttt{,}\mathcal{E}\texttt{)}$ where $\mathcal{C}$, $\mathcal{A}$, $\mathcal{E}$ are ground and $\texttt{reachable(}u_1'\texttt{,}v\texttt{)}$ where $u_1' =_{AC1} V \circ \mathcal{C}$ and $u_1' =_{AC1} V \circ \mathcal{E}$. According to the $AC1$ unification, if $u_1 =_{AC1} V \circ \mathcal{C}$ either $|\mathcal{C}, f| \leq |u_1, f|$ or there is a variable of type t_f in u_1. Consequently, $u_1 =_{AC1} V \circ \mathcal{C}$ iff $T(\mathcal{A})$ is enabled in ${C_1}^\mu$. Furthermore, if u_1 does not contain a variable of type t_f, it holds $|u_1', f| = |u_1, f| - |\mathcal{C}, f| + |\mathcal{E}, f|$. Otherwise, the codomain of any *mgeu* for u_1 and $V \circ \mathcal{C}$ must contain a variable X s.t. $t_f(X)$. Let T_1' be the set of such type declarations. Then, with $C_1' = \langle \texttt{reachable(}u_1'\texttt{,}v\texttt{)}, T_1', R\rangle$, it follows ${C_1}^\mu[T(\mathcal{A})\rangle {C_1'}^\mu$. However, u_1' may contain copies of a fluent f even if there is a variable X in u_1' with $t_f(X) \in T$. Using the partition function with $nmsg_\mathcal{T}$, u_2 is defined as u_1' where such additional copies are removed. By this it is ensured that for every marking M with ${C_1}^\mu[T(\mathcal{A})\rangle M$, $M^\alpha =_{RULE} C_2$. □

Proof. (**theorem 1**) Per definition $U = \{node(r, M_0)\}$ where $M_0 = A^\mu$. Now, we show the correspondence between each step in algorithm A.1 and algorithm 4.1. First, both algorithms terminate if no unprocessed nodes remain. Second, in algorithm 4.1 a selected node L is marked processed if $covered(L, \lambda)$ is true. Let (k, M) be the selected node by algorithm A.1 with $M = label(L)^\mu$. According to lemma 1, $covered(L, \lambda)$ iff there is an ancestor node (k_1, M_1) with $M = M_1$. In this case (k, M) is marked processed by algorithm A.1 (i.e. removed from the list of unprocessed nodes). Third, algorithm 4.1 calls $abstract(L, \lambda)$ if $whistle(L, \lambda) = \texttt{T}$. Using lemma 2 $whistle(L, \lambda) = \texttt{T}$ iff there is some ancestor L_k of L s.t. $label(L_k)^\mu \leq label(L)^\mu$. In algorithm A.1, abstraction is performed for every ancestor (k_1, M_1) of (k, M) with $M_1 < M$. Since the case $M_1 = M$ and $label(L_k) =_{RULE} label(L)$, respectively, has already been checked, it remains to be shown, that $C' = abstract(L, \lambda)$ iff ${C'}^\mu = M'$ and M' is defined as follows: if for some fluent f there exists an ancestor L_k of L in λ s.t. $label(L_k)^\mu < label(L)^\mu$ and $label(L_k)^\mu(f) < label(L)^\mu(f)$, $M(f) = \omega$, otherwise $M(f) = label(L)^\mu(f)$. This has been shown in lemma 3. Finally, from lemma 4 follows that $C_2 \in partition(ares(aunf(L_1)))$ iff there is an action $\mathcal{A}$ s.t. ${C_1}^\mu[T(\mathcal{A})\rangle {C_2}^\mu$. □