Nordic Journal of Computing

A Succinct Solver for ALFP

Flemming Nielson[1], Helmut Seidl[2] and Hanne Riis Nielson[1]

[1] *Informatics and Mathematical Modelling, Technical University of Denmark*
Email: {nielson,riis}@imm.dtu.dk

[2] *Universität Trier, FB IV – Informatik*
Email: seidl@uni-trier.de

Abstract. We develop a solver algorithm which allows to efficiently compute the stable model of a very expressive fragment of predicate logic. The succinct formulation of the algorithm is due to the disciplined use of continuations and memoisation. This facilitates giving a precise characterisation of the behaviour of the solver and to develop a complexity calculation which allows to obtain its formal complexity. Practical experiments on a control-flow analysis of the ambient calculus show that the solver frequently performs better than the worst-case complexity estimates.

CR Classification: D.3, F.3, F.4.

Key words: Program analysis, Alternation-free Least Fixpoint Logic, topdown fixed point computation.

1. Introduction

Functional languages are seldom used for expressing algorithmic insights because most algorithm designers prefer to express their insights in imperative languages and find it easier to perform the analysis of worst-case and average-case complexity with respect to a first-order imperative model of computation. As is clear to any functional programmer such algorithms can be coded in functional languages but this is somewhat besides the point of the algorithm designer. Indeed it is hard to criticise the algorithm designer for the choices made: The use of imperative languages makes it easier to communicate with colleagues working on algorithms (simply because this is the common language of the field) and it makes it easier to perform the complexity analysis due to the simpler and more direct layout of data in the store (thus avoiding the need to consider run-time stacks, closures and garbage collection) in accordance with well-established models of computation (like Turing Machines, the RAM model with uniform or logarithmic cost etc.).

Despite our sympathy with the traditional viewpoints of algorithm designers we feel they miss an important consideration: Algorithms need to be used by well-educated programmers and there is little value in developing

Received January 30, 2003.

an advanced algorithm if the prospective programmers cannot understand it or cannot see how to adapt it to their needs. Given the limited abilities of humans to grasp many concepts at the same time it is important to pay attention to the notation used for expressing the algorithmic insights. Indeed, "a notation is important for what it leaves out" [27, page 33] because more succinct specifications only need to focus on those key insights that have to be grasped at the same time. This is where functional programming enters the picture. More specifically we consider an eager functional language (since lazy languages are even further removed from the RAM model etc.) with imperative features (to maintain control over the sharing of results of computations) and continuations.

This paper takes the approach outlined to develop a state-of-the-art constraint solver. We consider the alternation-free fragment of *Least Fixpoint Logic* (ALFP) in clausal form. This logic is more expressive than, e.g., Datalog [12, 17] but still allows for polynomial model-checking routines. Formulae in this fragment naturally arise in the specification of static analyses of programs (see, e.g. [19, 23]). Here, we consider the systematic design of a fixpoint engine for this logic. We build on known approaches for sub-logics like [19, 23]. Following McAllester [19], we aim at an algorithm which makes computing the result as cheap as checking the result. Furthermore, the algorithm should be *simple*, i.e., work almost without pre-processing of formulae, and *predictable*, i.e., its complexity behaviour should be easily computable.

More specifically we build on previous insights on using functional programming for implementing state-of-the-art solvers [8, 15, 14] and we claim the following general advantages of our approach:

- The specification logic is much more expressive than the fragment of set constraints as provided by BANE [1] or the formulae of Datalog as, e.g., advocated by McAllester — yet our solver can be presented in less than a page of SML [20] pseudo-code.
- On the Datalog fragment, our solver achieves the best known theoretical bounds for corresponding solvers.
- The solver has a very modular design — allowing for a rapid implementation and simple explanation. In particular, the use of recursion and continuations allowed us to disregard a number of classical techniques (the use of work-lists, the identification of strong components etc.) without penalties in the performance observed.
- The solver has a very regular structure allowing an abstract characterisation of its behaviour — thereby paving the way for predicting also its best-case computational performance when solving formulae.

In our view it would not be possible to obtain as good results on obtaining precise estimates of the worst-case complexity of the solver if we had merely based our work on an off-the-shelf set constraint or Datalog solver. This is important because logically equivalent clauses may exhibit very different running times and it is important to be able to predict this.

In Section 2 we present the fragment of predicate logic considered, define its semantics and illustrate its usability in the specification of program analyses. To prepare for the development of the solver, we give in Section 3 a formal characterisation of a checker for the logic and we show how it can be implemented in SML. In Section 4 we then develop the solver itself, we explain some of the algorithmic techniques needed for obtaining good performance and also we develop a formal complexity result for the solver. Finally, in Section 5 we present some further optimisation techniques and we present benchmarks from a series of practical experiments. Section 6 contains the concluding remarks.

2. Alternation-free Least Fixpoint Logic

In this section, we introduce alternation-free *Least Fixpoint Logic* in clausal form (abbreviated ALFP) as our constraint formalism. Here, we build on Horn clauses *with sharing* as considered in [23] and extend them further by allowing also universal quantification in pre-conditions and negation. In summary, ALFP formulae extend Horn clauses (with explicit quantification) in that we additionally allow

- both existential and universal quantification in pre-conditions;
- negated queries (subject to a notion of stratification);
- disjunctions of pre-conditions; and
- conjunctions of conclusions.

In this section we present the syntax and the semantics of the logic and illustrate how it may be used to specify a simple program analysis.

2.1 Syntax

Assume we are given a fixed countable set $\mathcal{X}$ of (auxiliary) variables and a finite ranked alphabet $\mathcal{R}$ of predicate symbols. Then the set of clauses, *cl*, is given by the following grammar

$$\begin{array}{lcl} pre & ::= & R(x_1,\cdots,x_k) \quad | \quad \neg R(x_1,\cdots,x_k) \quad | \quad pre_1 \wedge pre_2 \\ & | & pre_1 \vee pre_2 \quad | \quad \exists x : pre \quad | \quad \forall x : pre \\ cl & ::= & R(x_1,\cdots,x_k) \quad | \quad \mathbf{1} \quad | \quad cl_1 \wedge cl_2 \\ & | & pre \Rightarrow cl \quad | \quad \forall x : cl \end{array}$$

where $R \in \mathcal{R}$ is a k-ary predicate symbol for $k \geq 1$, $x, x_1, \cdots \in \mathcal{X}$ denote arbitrary variables, and $\mathbf{1}$ is the always true clause. Occurrences of $R(\cdots)$ and $\neg R(\cdots)$ in pre-conditions are also called *queries* and *negative queries*, respectively, whereas the other occurrences are called *assertions* of the predicate R.

Example 1. Let E be a binary predicate defining the edges of a graph. Then the transitive closure T of the graph may be defined by the clause

$$(\forall x : \forall y : \mathsf{E}(x,y) \Rightarrow \mathsf{T}(x,y)) \quad \wedge \quad (\forall x : \forall y : \forall z : \mathsf{T}(x,y) \wedge \mathsf{T}(y,z) \Rightarrow \mathsf{T}(x,z))$$

A logically equivalent formulation is

$$\forall x : \forall y : \mathsf{E}(x,y) \Rightarrow (\mathsf{T}(x,y) \wedge (\forall z : \mathsf{T}(y,z) \Rightarrow \mathsf{T}(x,z)))$$

where we exploit the possibility of sharing of pre-conditions. □

In order to deal with negations conveniently, we restrict ourselves to *alternation-free* formulae. We introduce a notion of stratification similar to the one which is known from *Datalog* [7, 3]. A clause *cl* is an *alternation-free Least Fixpoint formula* (ALFP formula for short) if it has the form $cl = cl_1 \wedge \cdots \wedge cl_s$, and there is a function $rank : \mathcal{R} \to \mathbb{N}$ such that for all $j = 1, \cdots, s$, the following properties hold:

- ∘ all predicates of assertions in cl_j have rank j;
- ∘ all predicates of queries in cl_j have ranks at most j; and
- ∘ all predicates of negated queries in cl_j have ranks strictly less than j.

Each j is said to be a stratum and s is the number of strata.

Example 2. This notion of stratification allows us to define an equality predicate eq and a non-equality predicate neq by the clause:

$$(\forall x : \mathsf{eq}(x,x)) \wedge (\forall x : \forall y : \neg\mathsf{eq}(x,y) \Rightarrow \mathsf{neq}(x,y))$$

Taking the function *rank* to have $rank(\mathsf{eq}) = 1$ and $rank(\mathsf{neq}) = 2$ it is clear that the conditions are fulfilled. Consequently we may dispense with an explicit treatment of equality and non-equality predicates in the development of the present paper.

The following formula is ruled out by our notion of stratification:

$$(\forall x : \neg P(x) \Rightarrow Q(x)) \wedge (\forall x : Q(x) \Rightarrow P(x))$$

This is because it is impossible to have $rank(P) < rank(Q)$ and $rank(Q) \leq rank(P)$. □

2.2 Semantics

Given a non-empty and finite[1] universe $\mathcal{U}$ of atomic values (or atoms) together with interpretations ρ and σ for predicate symbols R and free variables x, respectively, we define the satisfaction relations

$$(\rho, \sigma) \models pre \quad \text{and} \quad (\rho, \sigma) \models cl$$

for pre-conditions and clauses as in Table I. Here we write $\rho(R)$ for the set of k-tuples $(a_1, \cdots, a_k)$ from $\mathcal{U}$ associated with the k-ary predicate R, we

$(\rho,\sigma) \models R\,(x_1,\cdots,x_k)$	iff	$(\sigma(x_1),\cdots,\sigma(x_k)) \in \rho(R)$
$(\rho,\sigma) \models \neg R\,(x_1,\cdots,x_k)$	iff	$(\sigma(x_1),\cdots,\sigma(x_k)) \notin \rho(R)$
$(\rho,\sigma) \models pre_1 \wedge pre_2$	iff	$(\rho,\sigma) \models pre_1$ and $(\rho,\sigma) \models pre_2$
$(\rho,\sigma) \models pre_1 \vee pre_2$	iff	$(\rho,\sigma) \models pre_1$ or $(\rho,\sigma) \models pre_2$
$(\rho,\sigma) \models \exists x : pre$	iff	$(\rho,\sigma[x \mapsto a]) \models pre$ for some $a \in \mathcal{U}$
$(\rho,\sigma) \models \forall x : pre$	iff	$(\rho,\sigma[x \mapsto a]) \models pre$ for all $a \in \mathcal{U}$
$(\rho,\sigma) \models R\,(x_1,\cdots,x_k)$	iff	$(\sigma(x_1),\cdots,\sigma(x_k)) \in \rho(R)$
$(\rho,\sigma) \models \mathbf{1}$		always
$(\rho,\sigma) \models cl_1 \wedge cl_2$	iff	$(\rho,\sigma) \models cl_1$ and $(\rho,\sigma) \models cl_2$
$(\rho,\sigma) \models pre \Rightarrow cl$	iff	$(\rho,\sigma) \models cl$ whenever $(\rho,\sigma) \models pre$
$(\rho,\sigma) \models \forall x : cl$	iff	$(\rho,\sigma[x \mapsto a]) \models cl$ for all $a \in \mathcal{U}$

TABLE I: Semantics of pre-conditions and clauses.

write $\sigma(x)$ for the atom of $\mathcal{U}$ bound to x and finally $\sigma[x \mapsto a]$ stands for the mapping that is as σ except that x is mapped to a.

In the sequel, we view the free variables occurring in a formula as constants (i.e. atoms) from the finite universe $\mathcal{U}$. Thus, given an interpretation σ_0 of the constant symbols, in the clause *cl*, we call an interpretation ρ of the predicate symbols $\mathcal{R}$ a *solution* to the clause provided $(\rho,\sigma_0) \models cl$.

Let Δ denote the set of interpretations ρ of predicate symbols in $\mathcal{R}$ over $\mathcal{U}$. Then Δ is a complete lattice w.r.t. the lexicographical ordering "$\sqsubseteq$" defined by $\rho_1 \sqsubseteq \rho_2$ if and only if there is some $1 \le j \le s$ such that the following properties hold:

- $\rho_1(R) = \rho_2(R)$ for all $R \in \mathcal{R}$ with $rank(R) < j$;
- $\rho_1(R) \subseteq \rho_2(R)$ for all $R \in \mathcal{R}$ with $rank(R) = j$ and
- either $j = s$ or $\rho_1(R) \subset \rho_2(R)$ for at least one $R \in \mathcal{R}$ with $rank(R) = j$.

Perhaps the more familiar ordering is "$\subseteq$" defined by $\rho_1 \subseteq \rho_2$ if and only if $\rho_1(R) \subseteq \rho_2(R)$ for all $R \in \mathcal{R}$. The two orderings "$\sqsubseteq$" and "$\subseteq$" coincide when there is only one stratum (i.e. $s = 1$); slightly more generally we have that if ρ_0 is an *input interpretation*, meaning that it gives non-empty interpretations to predicates in the first stratum only, then for all ρ we have $\rho_0 \sqsubseteq \rho$ if and only if $\rho_0 \subseteq \rho$. We have:

PROPOSITION 1. *Assume cl is an ALFP formula and σ_0 is an interpretation of the free variables in cl. Then the set of all ρ with $(\rho,\sigma_0) \models cl$ forms a* Moore *family, i.e., is closed under greatest lower bounds (wrt. $\sqsubseteq$).*

A proof of Proposition 1 can be found in Appendix A. We conclude that for every initial interpretation ρ_0 of the predicate symbols there is a

[1] All of the development of Section 2 will go through for an infinite universe as well.

lexicographically least solution ρ of *cl* (given a fixed σ_0) such that $\rho_0 \subseteq \rho$; this solution is called the *optimal* solution of *cl* exceeding ρ_0. The proof simply amounts to observing that there is a formula cl_{ρ_0} (assuming that σ_0 can be arranged to be surjective) such that $\rho_0 \subseteq \rho$ is equivalent to $(\rho, \sigma_0) \models cl_{\rho_0}$ and then use that the set of all solutions of $cl_{\rho_0} \wedge cl$ is a Moore family.

We should point out that a key design decision for considering the fragment ALFP of first order logic is that it facilitates establishing a Moore Family result as in Proposition 1. In more general terms this ensures that the approach taken falls within the general framework of Abstract Interpretation [9, 10, 21] so that we can be sure that there always is a single best solution to the analysis problem considered. For this reason we decided not to allow a precondition of the form $cl \Rightarrow pre$ as may be found in Hereditary Harrop Formulae [16].

2.3 Expressiveness of ALFP

The extra features of ALFP compared with classical Horn clauses increase the ease with which various properties can be expressed. The use of conjunctions in conclusions does not add further expressiveness as the same property can be expressed by duplicating the pre-condition or by introducing an auxiliary predicate for it. Similarly, the use of disjunction and existential quantification in the pre-conditions does not add further expressiveness as the same effect can be obtained using auxiliary predicates. However, the combination of universal quantification in pre-conditions and negated queries do add expressiveness as is illustrated by the following example:

EXAMPLE 3. Continuing Example 1 consider the formula:

$$\forall x.(\forall y.\neg \mathsf{E}(x,y) \vee \mathsf{A}(y)) \Rightarrow \mathsf{A}(x)$$

Taking $rank(\mathsf{E}) = 1$ and $rank(\mathsf{A}) = 2$ it is easy to see that this is an ALFP formula defining a predicate A that holds on the set of all acyclic nodes in a graph, i.e., all nodes from which no cycle can be reached. Without fixing a finite universe in advance, this predicate is not definable in Datalog (even with stratified negation [12, 17]). □

2.4 Worked Example: Discretionary Ambients

To illustrate the use of ALFP for program analysis we specify in this subsection a simple control flow analysis for Discretionary Ambients [24], a variant of the Mobile Ambients [6] that directly supports discretionary access control.

The syntax of processes P and capabilities M are given by the following abstract syntax

$$P \ ::= \ (\nu\mu)P \ | \ (\nu n\!:\!\mu)\,P \ | \ \mathbf{0} \ | \ P_1 \,|\, P_2 \ | \ !P \ | \ n[P] \ | \ M.P$$
$$M \ ::= \ \texttt{in}\, n \ | \ \texttt{out}\, n \ | \ \texttt{open}\, n \ | \ \overline{\texttt{in}}_\mu\, n \ | \ \overline{\texttt{out}}_\mu\, n \ | \ \overline{\texttt{open}}_\mu\, n$$

where we use n to range over names and μ to range over group names. The construct $(\nu\mu)P$ introduces a new group μ and its scope P; the construct $(\nu n : \mu)\,P$ then introduces a new name n of the already existing group μ and its scope P. The remaining constructs for processes are as for Mobile Ambients [6]. The capabilities $\mathtt{in}\ n$, $\mathtt{out}\ n$ and $\mathtt{open}\ n$ are as in Mobile Ambients and the co-capabilities $\overline{\mathtt{in}}_\mu\ n$, $\overline{\mathtt{out}}_\mu\ n$ and $\overline{\mathtt{open}}_\mu\ n$ extend those of Safe Ambients [18]: n is the name of the object granting the access right and μ is the group of the subject that is allowed to perform the operation.

We consider a simple control flow analysis approximating the behavior of a process by a *single* abstract configuration that describes which subambients and (co-)capabilities an ambient group *may* possess in *any* of the possible derivatives that the overall process may have. It amounts to systematically performing the following approximations:

- The analysis distinguishes between the various groups of ambients but not between the individual ambients.
- The analysis does not keep track of the exact order of the capabilities inside an ambient nor of their multiplicities.
- The analysis represents the tree structure of the processes by a binary predicate $\mathcal{I}$ modeling the father-son relationship.

Formally, the analysis defines a binary predicate $\mathcal{I}$ with

$$\mathcal{I} \subseteq \mathbf{Group} \times (\mathbf{Group} \cup \mathbf{Cap})$$

where **Group** is the set of groups, and **Cap** is the set of group capabilities and group co-capabilities (i.e. built from groups rather than ambient names). Here $\mathcal{I}(\mu, \mu')$ should hold whenever an ambient of the group μ' is a subambient of an ambient of group μ and $\mathcal{I}(\mu, m)$ should hold whenever the group capability or group co-capability m occurs in an ambient in the group μ.

We assume that the syntax of the process of interest is given by a binary predicate PRG relating the ambient groups (denoted by the variable '$\star$' below) to their initial subambients and capabilities. Furthermore, we assume that there are predicates AMB, IN, $\overline{\text{IN}}$, OUT, $\overline{\text{OUT}}$, OPEN and $\overline{\text{OPEN}}$ which relate ambients to their groups and capabilities and co-capabilities to their group restrictions, respectively. These predicates jointly constitute the initial interpretation ρ_0 considered above; hence it can be represented by a clause cl_{ρ_0} although we shall prefer not to do so.

In order to enhance readability we will use *structured terms*. For example, we write

$$\forall *, \mu \ :\ \text{PRG}(*, \mathsf{amb}(\mu)) \Rightarrow \mathcal{I}(*, \mu)$$

instead of:

$$\forall *, \mu \ :\ (\exists a \ :\ \text{PRG}(*, a) \wedge \text{AMB}(a, \mu)) \Rightarrow \mathcal{I}(*, \mu)$$

In order to remove the structured terms in the specification below, we thus have to *invert* the occurring constructor applications by introducing a subsequent query of the predicate corresponding to the constructor. Note that

$$\forall \star, \mu : \textsc{prg}(\star, \mathsf{amb}(\mu)) \Rightarrow \mathcal{I}(\star, \mu)$$

$$\forall \star, \mu : \textsc{prg}(\star, \mathsf{in}(\mu)) \Rightarrow \left[\begin{array}{l} \mathcal{I}(\star, \mathsf{in}(\mu)) \wedge \\ \forall \mu_a, \mu_p : \left[\begin{array}{l} \mathcal{I}(\mu, \overline{\mathsf{in}}(\mu_a, \mu)) \wedge \\ \mathcal{I}(\mu_p, \mu) \wedge \\ \mathcal{I}(\mu_a, \mathsf{in}(\mu)) \wedge \\ \mathcal{I}(\mu_p, \mu_a) \end{array}\right] \Rightarrow \mathcal{I}(\mu, \mu_a) \end{array}\right]$$

$$\forall \star, \mu : \textsc{prg}(\star, \mathsf{out}(\mu)) \Rightarrow \left[\begin{array}{l} \mathcal{I}(\star, \mathsf{out}(\mu)) \wedge \\ \forall \mu_a, \mu_g : \left[\begin{array}{l} \mathcal{I}(\mu, \overline{\mathsf{out}}(\mu_a, \mu)) \wedge \\ \mathcal{I}(\mu_g, \mu) \wedge \\ \mathcal{I}(\mu_a, \mathsf{out}(\mu)) \wedge \\ \mathcal{I}(\mu, \mu_a) \end{array}\right] \Rightarrow \mathcal{I}(\mu_g, \mu_a) \end{array}\right]$$

$$\forall \star, \mu : \textsc{prg}(\star, \mathsf{open}(\mu)) \Rightarrow \left[\begin{array}{l} \mathcal{I}(\star, \mathsf{open}(\mu)) \wedge \\ \forall \mu_p : \left[\begin{array}{l} \mathcal{I}(\mu, \overline{\mathsf{open}}(\mu_p, \mu)) \wedge \\ \mathcal{I}(\mu_p, \mu) \wedge \\ \mathcal{I}(\mu_p, \mathsf{open}(\mu)) \end{array}\right] \Rightarrow \begin{array}{l} \forall u : \mathcal{I}(\mu, u) \\ \qquad \Rightarrow \mathcal{I}(\mu_p, u) \end{array} \end{array}\right]$$

$$\forall \star, \mu, \mu' : \textsc{prg}(\star, \overline{\mathsf{in}}(\mu, \mu')) \Rightarrow \mathcal{I}(\star, \overline{\mathsf{in}}(\mu, \mu'))$$

$$\forall \star, \mu, \mu' : \textsc{prg}(\star, \overline{\mathsf{out}}(\mu, \mu')) \Rightarrow \mathcal{I}(\star, \overline{\mathsf{out}}(\mu, \mu'))$$

$$\forall \star, \mu, \mu' : \textsc{prg}(\star, \overline{\mathsf{open}}(\mu, \mu')) \Rightarrow \mathcal{I}(\star, \overline{\mathsf{open}}(\mu, \mu'))$$

TABLE II: Analysis of Discretionary Ambients.

this removal introduces one extra clause variable (as a above) whose scope is restricted by an existential quantification.

Given this convention, the analysis is specified by the conjunction of the set of clauses of Table II. We refer to [24] for a detailed explanation of the analysis and a discussion of its properties and only offer a few explanations here. The first clause merely records the fact that if some ambient group μ occurs inside some other ambient group $\star$ in the original program then this should be recorded in the $\mathcal{I}$ relation. The last three clauses are analogous. For the remaining three clauses observe that the first conjunct of each conclusion is analogous to above. The second conjunct of each conclusion takes care of the operational semantics: the condition in the inner square brackets indicates when a reduction is possible and the conclusion records the result of the reduction.

It is important to observe that the clauses of Table II have constant size irrespectively of the size of the ambient process P considered: the latter only influences the size of the relations PRG etc. in the initial interpretation ρ_0. Also one may check that the clause is indeed a clause in ALFP except for the omission of conjunctions between the lines; in particular there is no use of negation and all relations are in the same stratum.

2.5 Asymptotic Complexity

There is a rather straight-forward method for solving an ALFP formula *cl* that gives a good indication of the worst-case complexity. It proceeds by instantiating all variables occurring in *cl* in all possible ways. The resulting system has no free variables and can be solved by classical solvers for alternation-free Boolean equation systems [13]. Here it is natural to let the universe $\mathcal{U}$ consist of the set of atoms occurring in the range of σ_0 or ρ_0.

PROPOSITION 2. *The optimal solution ρ exceeding an interpretation ρ_0 (i.e. $\rho \supseteq \rho_0$) of a ALFP formula* cl *(w.r.t. an interpretation σ_0 of the constant symbols) can be computed in time*

$$\mathcal{O}(\#\rho + N^r \cdot n)$$

where N is the size of the universe, n is the size of cl*, and r is the maximal nesting depth of quantifiers in* cl *and $\#\rho$ is the sum of cardinalities of predicates $\rho(R)$.*

A similar result holds for checking that a candidate solution ρ is indeed a solution to cl*.*

PROOF. The first part of the result is a straight-forward extension of the proof of Proposition 1 of [23] (itself extending Theorem 1 of [19]) and is omitted. The second part of the result is achieved by computing the optimal solution ρ' to *cl* that exceeds ρ and checking that $\rho' = \rho$. □

As observed by McAllester [19] for conventional Horn clauses, such an approach and such a complexity bound is unsatisfactory for program analysis, because it suggests an implementation that always achieves the worst-case complexity. Rather we would like an implementation that would produce better complexity in benign cases. This motivates the development of the subsequent sections.

To make the presentation more accessible we shall subsequently *assume* that the clauses considered have constant size, i.e. the length needed to write the clause is $\mathcal{O}(1)$. This is in line with the development of Subsection 2.4 where the program dependent information is directly coded as relations (like PRG) in the initial interpretation ρ_0 and where the analysis itself is represented by a clause whose size is independent of the size of the program.

3. The ALFP Checker

To make it easier to present the ALFP solver in the next section we shall introduce a simplified version of the algorithm that can be used to *check* whether a formula *cl* is true for fixed interpretations σ_0 and ρ of the constant symbols and the predicate symbols — the solver will additionally compute the interpretation of the predicate symbols.

3.1 Partial Environments

To mimic the modifications of the interpretation of the variables in the semantics in Table I the checker (and the solver) will operate on *partial environments* mapping variables to atoms of the universe. We shall construct the environments in a lazy fashion meaning that variables may not have been given their values when introduced by the quantifiers, hence the use of partial environments. It is convenient to represent the partial environments as lists of pairs

$$\eta = (x_1, d_1) :: \cdots :: (x_n, d_n)$$

where the x_i are (not necessarily distinct) variables and the d_i are *optional values*, i.e., either atoms or the special value $\Box$ denoting that so far the variable is "unbound". Two partial environments $\eta = (x_1, d_1) :: \cdots :: (x_n, d_n)$ and $\eta' = (x'_1, d'_1) :: \cdots :: (x'_{n'}, d'_{n'})$ are related with the ordering

$$\eta \sqsubseteq \eta'$$

if $n = n'$ and for all i, $x_i = x'_i$ and whenever $d_i \neq \Box$ then $d_i = d'_i$; this expresses that η' is more instantiated than η. Note that the two environments are only allowed to differ in the second components of the pairs.

A partial environment η gives rise to an interpretation $[\eta] : \mathcal{X} \to \mathcal{U}$ defined as follows:

$$\begin{array}{rcl} [(x,d) :: \eta](y) & = & \left\{ \begin{array}{ll} d & \text{if } x = y \\ [\eta](x) & \text{otherwise} \end{array} \right. \\ [\epsilon](y) & = & \sigma_0(y) \end{array}$$

We shall use this to formalise the relationship between partial environments and interpretations. An interpretation $\sigma : \mathcal{X} \to \mathcal{U}$ (as used in Table I) is *unifiable* with a partial environment η, written

$$\eta \preceq \sigma$$

if for all variables x either $\sigma(x) = [\eta](x)$ or $[\eta](x) = \Box$. Note that if $\eta \sqsubseteq \eta'$ and $\eta' \preceq \sigma$ then also $\eta \preceq \sigma$.

The main auxiliary operation on environments is the function unify which, when given a partial environment η, a tuple $(y_1, \cdots, y_k)$ of variables and a tuple $(a_1, ..., a_k)$ of atoms (representing a tuple of some k-ary predicate), determines the minimal modification η' of η with $[\eta']\, y_i = a_i$ for all i and $\eta \sqsubseteq \eta'$; if η' does not exist then the call to unify fails. Formally we define the operation as follows:

$$\mathsf{unify}(\eta, (y_1, \cdots, y_k), (a_1, \cdots, a_k)) = \eta[y_1 \mapsto a_1] \cdots [y_k \mapsto a_k]$$

where

$$\begin{array}{rcl} ((x,d) :: \eta)[y \mapsto a] & = & \left\{ \begin{array}{ll} (x,a) :: \eta & \text{if } x = y \text{ and } d = a \vee d = \Box \\ \text{fail} & \text{if } x = y \text{ and } d \neq a \wedge d \neq \Box \\ (x,d) :: (\eta[y \mapsto a]) & \text{otherwise} \end{array} \right. \\ \epsilon[y \mapsto a] & = & \left\{ \begin{array}{ll} \text{fail} & \text{if } \sigma_0(y) \neq a \\ \epsilon & \text{otherwise} \end{array} \right. \end{array}$$

Clearly $\eta \sqsubseteq \mathsf{unify}(\eta, (y_1, \cdots, y_k), (a_1, \cdots, a_k))$ whenever the latter does not fail. Hence $\mathsf{unify}(\eta, (y_1, \cdots, y_k), (a_1, \cdots, a_k))$ is the most general unifier of η and $(y_1, a_1) :: \cdots :: (y_k, a_k)$.

3.2 Abstract Characterisation of the Checker

Given a clause and a set $\mathcal{E}$ of partial environments the checker will determine whether or not the clause will evaluate to true for all interpretations σ that are unifiable with a partial environment from $\mathcal{E}$. It does so by recursively processing the clause and by updating the set of partial environments by collecting the bindings of the instantiated variables. Whenever a quantified subexpression is encountered which introduces variable x, we extend each of the partial environments with $(x, \Box)$ indicating that the variable has been introduced but its value has not yet been determined. Accordingly, whenever the quantified subexpression is left, the top-most occurrence of the variable x is deleted from the partial environments and since the environments are presented as lists, the previous (global) binding of x will become visible again. New bindings for variables are obtained at the queries $R(x_1, \cdots, x_k)$: here the auxiliary function unify is applied to every partial environment η of interest, the tuple $(x_1, \cdots, x_k)$ and every tuple $(a_1, \cdots, a_k)$ belonging to the interpretation $\rho(R)$ of R. The resulting set of most general unifiers is then propagated further through the clause.

The details of the functions are given in Table III and are explained below. The function $\mathcal{T}^{\bullet}_{\rho}[\![cl]\!]$ is concerned with the clauses whereas $\mathcal{T}_{\rho}[\![pre]\!]$ takes care of the pre-conditions. Both functions make use of the prefixing operation :: that is extended in a pointwise manner to sets of partial environments:

$$(x, d) \mathbin{\widehat{::}} \mathcal{E} = \{(x, d) :: \eta \mid \eta \in \mathcal{E}\}$$

Observe that if $\eta \preceq \sigma$ then $(x, d) :: \eta \preceq \sigma[x \mapsto a]$ for some $a \in \mathcal{U}$: if $d = \Box$ then any a will do and otherwise take $a = d$.

Similarly, the operation **tl** returning the tail of a list is extended to sets in a pointwise manner; more specifically:

$$\begin{array}{rcl} \mathbf{tl}(\eta) & = & \begin{cases} \eta' & \text{if } \eta = (x, d) :: \eta' \\ \text{fail} & \text{otherwise} \end{cases} \\ \widehat{\mathbf{tl}}(\mathcal{E}) & = & \{\mathbf{tl}(\eta) \mid \eta \in \mathcal{E} \wedge \mathbf{tl}(\eta) \neq \text{fail}\} \end{array}$$

Note that $\mathbf{tl}(\eta) \preceq \sigma$ does not imply that $\eta \preceq \sigma$ since η may impose requirements on the value of some variable x. However, it will be the case that $\eta \preceq \sigma[x \mapsto a]$ for some $a \in \mathcal{U}$ (in particular $\eta[x]$ in case it is not $\Box$).

The clauses defining $\mathcal{T}_{\rho}[\![pre]\!]$ are mostly straightforward: given a precondition *pre* and a set $\mathcal{E}$ of partial environments it will return a set of further instantiations of these environments that is compatible with the requirements of *pre*. The clause for conjunction expresses that first we modify the set of partial environments as required by pre_1 and this set is next modified as required by pre_2. In the case of disjunction the same set $\mathcal{E}$ of partial

$$
\begin{array}{lcl}
\mathcal{T}_\rho[\![R(\vec{x})]\!]\,\mathcal{E} & = & \{\eta' \mid \eta \in \mathcal{E}, \vec{a} \in \rho(R), \eta' = \mathsf{unify}\,(\eta,\vec{x},\vec{a}) \neq \mathsf{fail}\} \\
\mathcal{T}_\rho[\![\neg R(\vec{x})]\!]\,\mathcal{E} & = & \{\eta' \mid \eta \in \mathcal{E}, \vec{a} \notin \rho(R), \eta' = \mathsf{unify}\,(\eta,\vec{x},\vec{a}) \neq \mathsf{fail}\} \\
\mathcal{T}_\rho[\![pre_1 \wedge pre_2]\!]\,\mathcal{E} & = & \mathcal{T}_\rho[\![pre_2]\!]\,(\mathcal{T}_\rho[\![pre_1]\!]\,\mathcal{E}) \\
\mathcal{T}_\rho[\![pre_1 \vee pre_2]\!]\,\mathcal{E} & = & \mathcal{T}_\rho[\![pre_1]\!]\,\mathcal{E} \cup \mathcal{T}_\rho[\![pre_2]\!]\,\mathcal{E} \\
\mathcal{T}_\rho[\![\exists x : pre]\!]\,\mathcal{E} & = & \widehat{\mathbf{tl}}\,(\mathcal{T}_\rho[\![pre]\!]\,((x,\Box)\,\widehat{::}\,\mathcal{E})) \\
\mathcal{T}_\rho[\![\forall x : pre]\!]\,\mathcal{E} & = & \widehat{\mathbf{tl}}\,(\mathcal{T}_{\rho,\mathcal{U}}[\![pre]\!]\,((x,\Box)\,\widehat{::}\,\mathcal{E})) \\
 & & \text{where } \mathcal{T}_{\rho,a::U}[\![pre]\!]\,((x,d)\,\widehat{::}\,\mathcal{E}) \\
 & & \qquad = \mathcal{T}_{\rho,U}[\![pre]\!]\,(\mathcal{T}_\rho[\![pre]\!]\,((x,a)\,\widehat{::}\,\mathcal{E})) \\
 & & \quad \mathcal{T}_{\rho,[\,]}[\![pre]\!]\,\mathcal{E} = \mathcal{E} \\
\hline
\mathcal{T}^\bullet_\rho[\![R(\vec{x})]\!]\,\mathcal{E} & = & \forall \eta \in \mathcal{E} : \forall \vec{a} : \mathsf{unify}\,(\eta,\vec{x},\vec{a}) \neq \mathsf{fail} \Rightarrow \vec{a} \in \rho(R) \\
\mathcal{T}^\bullet_\rho[\![\mathbf{1}]\!]\,\mathcal{E} & = & \mathsf{true} \\
\mathcal{T}^\bullet_\rho[\![cl_1 \wedge cl_2]\!]\,\mathcal{E} & = & \mathcal{T}^\bullet_\rho[\![cl_1]\!]\,\mathcal{E} \wedge \mathcal{T}^\bullet_\rho[\![cl_2]\!]\,\mathcal{E} \\
\mathcal{T}^\bullet_\rho[\![pre \Rightarrow cl]\!]\,\mathcal{E} & = & \mathcal{T}^\bullet_\rho[\![cl]\!]\,(\mathcal{T}_\rho[\![pre]\!]\,\mathcal{E}) \\
\mathcal{T}^\bullet_\rho[\![\forall x.cl]\!]\,\mathcal{E} & = & \mathcal{T}^\bullet_\rho[\![cl]\!]\,((x,\Box)\,\widehat{::}\,\mathcal{E})
\end{array}
$$

TABLE III: Abstract checking functions.

environments are modified by pre_1 and pre_2 and we return the union of the resulting sets.

In the case of existential quantification we first determine the potential extensions of the given partial environments; then we return the relevant parts of the extended environments. In the case of universal quantification we have to inspect *all* the atoms of the universe and find the extensions of $\mathcal{E}$ that are compatible with the requirements of *pre*. To do so we rely on an enumeration of the universe $\mathcal{U}$ and we introduce the auxiliary function $\mathcal{T}_{\rho,U}$ to traverse a list U of atoms and successively bind the atoms to x and modify the set of partial environments to be compatible with the requirements of *pre*. Initially we call the function with the complete listing $\mathcal{U}$ of the universe (i.e. $\mathcal{T}_{\rho,\mathcal{U}}$). Note that the recursive structure of $\mathcal{T}_{\rho,U}$ reflects that the universal quantification basically is a conjunction over the full universe.

The following fact can be proved using structural induction and is essential for understanding the details of Table III:

FACT 1. If $\eta' \in \mathcal{T}_\rho[\![pre]\!]\,\mathcal{E}$ then there exists $\eta \in \mathcal{E}$ such that $\eta \sqsubseteq \eta'$.

This means that $\mathcal{T}_\rho[\![pre]\!]\,\mathcal{E}$ always makes further instantiations of the partial environments of $\mathcal{E}$. Writing $\overline{\mathcal{E}}$ for the set of interpretations that are unifiable with a partial environment from $\mathcal{E}$

$$\overline{\mathcal{E}} = \{\sigma \mid \exists \eta \in \mathcal{E} : \eta \preceq \sigma\}$$

we can conclude that $\overline{\mathcal{T}_\rho[\![pre]\!]\,\mathcal{E}} \subseteq \overline{\mathcal{E}}$.

The correctness of the specification can be stated as follows:

PROPOSITION 3. *For all pre-conditions pre, clauses cl, and sets $\mathcal{E}$ of partial environments, the following holds:*

(1) $\overline{\mathcal{T}_\rho[\![pre]\!]\,\mathcal{E}} = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge (\rho, \sigma) \models pre\}$.

(2) $\mathcal{T}^\bullet_\rho[\![cl]\!]\,\mathcal{E}$ *if and only if* $\forall \sigma \in \overline{\mathcal{E}} : (\rho, \sigma) \models cl$.

The proofs are by structural induction and may be found in Appendix B.

EXAMPLE 4. Consider the formula *cl* of Example 1 specifying the transitive closure T of a predicate E

$$\forall x : \forall y : \mathsf{E}(x, y) \Rightarrow (\mathsf{T}(x, y) \wedge (\forall z : \mathsf{T}(y, z) \Rightarrow \mathsf{T}(x, z)))$$

and assume that the predicate symbols E and T have the following interpretations:

$$\begin{aligned} \rho(\mathsf{E}) &= \{(a, b), (b, c), (c, b)\} \\ \rho(\mathsf{T}) &= \{(a, b), (b, c), (c, b), (a, c), (b, b), (c, c)\} \end{aligned}$$

Using the semantics of Table I it is easy to see that the formula *cl* is true. Let us see how the same result is obtained using the specification of Table III.

We are interested in the value of $\mathcal{T}^\bullet_\rho[\![cl]\!]\{[\,]\}$ where $[\,]$ is the empty partial environment. Using the equations for $\mathcal{T}^\bullet_\rho$ we see that we are interested in

$$\mathcal{T}^\bullet_\rho[\![\mathsf{T}(x, y) \wedge (\forall z : \mathsf{T}(y, z) \Rightarrow \mathsf{T}(x, z))]\!]\mathcal{E}$$

where $\mathcal{E} = \mathcal{T}_\rho[\![\mathsf{E}(x, y)]\!]\{(y, \Box) :: (x, \Box)\}$. The equations for $\mathcal{T}_\rho$ give

$$\mathcal{E} = \{(y, b) :: (x, a),\ (y, c) :: (x, b),\ (y, b) :: (x, c)\}$$

It is now easy to verify that $\mathcal{T}^\bullet_\rho[\![\mathsf{T}(x, y)]\!]\mathcal{E} = \mathsf{true}$ by simply inspecting the definition of $\rho(\mathsf{T})$. Turning to $\mathcal{T}^\bullet_\rho[\![\forall z : \mathsf{T}(y, z) \Rightarrow \mathsf{T}(x, z)]\!]\mathcal{E}$ we have to compute $\mathcal{T}^\bullet_\rho[\![\mathsf{T}(x, z)]\!]\mathcal{E}'$ where $\mathcal{E}' = \mathcal{T}_\rho[\![\mathsf{T}(y, z)]\!]((z, \Box) \hat{::} \mathcal{E})$. Since

$$(z, \Box) \hat{::} \mathcal{E} = \{(z, \Box) :: (y, b) :: (x, a),\ (z, \Box) :: (y, c) :: (x, b),\ (z, \Box) :: (y, b) :: (x, c)\}$$

we obtain:

$$\begin{aligned} \mathcal{E}' = \{ & (z, b) :: (y, b) :: (x, a),\ (z, b) :: (y, c) :: (x, b),\ (z, b) :: (y, b) :: (x, c), \\ & (z, c) :: (y, b) :: (x, a),\ (z, c) :: (y, c) :: (x, b),\ (z, c) :: (y, b) :: (x, c)\} \end{aligned}$$

It is now easy to verify that $\mathcal{T}^\bullet_\rho[\![\mathsf{T}(x, z)]\!]\mathcal{E}' = \mathsf{true}$. $\Box$

3.3 The Implementation

We shall now describe the main characteristics of an SML implementation of the checker.

The *interpretation of the predicate symbols* is given by a global data structure `rho`. The operation `rho.has` checks whether a tuple of atoms from the universe is associated with a given predicate symbol and the operation `rho.sub` returns a list of the tuples associated with a given predicate symbol:

```
type univ   =  (* the universe *)
type rho    =  (* the interpretation of predicates *)

rho.has:       predicate * univ list -> bool
rho.sub:       predicate -> (univ list) list
```

The *partial environments* are implemented using the option data type of SML:

```
type env     =   (var * univ option) list

unify:           env * var list * univ list -> env option
unifiable:       env * var list -> (univ list) list
```

The function `unify` is as described in the previous subsection and again the option data type is used to signal potential failure. The function `unifiable` returns the set of tuples for which a call of `unify` will succeed.

To prepare for the formulation of the solver in the next section we shall use a non-standard representation of the *truth values*: *true* is represented by normal termination with the unit value `()` whereas *false* is represented by raising the exception `FF`.

The function `check` implementing $\mathcal{T}_\rho[\![pre]\!]$ is written in continuation passing style

```
type cont  =  env -> unit
check:        pre * cont -> cont
```

and thus is applied to a pre-condition *pre* and a continuation `K` together with a partial environment η. The pseudo code is presented in Table IV and is explained below. In the case of queries $R(\vec{x})$ we first construct a continuation `K'` that given a tuple $\vec{a}$ of R determines the most general unifier of $\vec{x}$ with $\vec{a}$ relative to η and propagates it to `K` if successful; then `K'` is applied to *all* the tuples of R and to express this we make use of the function `app` that applies a function to each element of a list and returns the value `()`:

```
fun app f []      = ()
  | app f (x::xs) = (f x; app f xs)
```

In the case of negated queries $\neg R(\vec{x})$ we construct a continuation `K'` that is applied to all tuples that are unifiable with $\vec{x}$ relative to η: given a tuple $\vec{a}$ *not* in R it constructs the most general unifier of $\vec{x}$ with $\vec{a}$ relative to η and propagates it to the continuation `K`. The remaining cases of the function `check` follow the overall pattern of Table III.

The function `execute` implementing $\mathcal{T}^\bullet_\rho[\![cl]\!]$ will construct a continuation

```
execute:  cl -> cont
```

```
fun check (R(x⃗), K) η         =  let fun K' a⃗ =  case unify(η, x⃗, a⃗) of
                                                     NONE -> ()
                                                   | SOME η' -> K(η')
                                   in app K' (rho.sub R)
                                   end
  | check (¬R(x⃗), K) η         =  let fun K' a⃗ =  if rho.has(R, a⃗)
                                                   then ()
                                                   else K (unify(η, x⃗, a⃗))
                                   in app K' (unifiable (η, x⃗))
                                   end
  | check (pre₁ ∧ pre₂, K) η   =  check (pre₁, check (pre₂, K)) η
  | check (pre₁ ∨ pre₂, K) η   =  check (pre₁, K) η; check (pre₂, K) η
  | check (∃x : pre, K) η      =  check (pre, K ∘ tl) ((x, NONE)::η)
  | check (∀x : pre, K) η      =  let fun check' [] ((x, _) :: η') = K(η')
                                           | check' (a :: U) ((x, _) :: η')
                                               = check (pre, check' U) ((x, SOME a) :: η')
                                   in check' 𝒰 ((x, NONE) :: η)
                                   end
--------------------------------------------------------------------------------------------
fun execute (R(x⃗)) η           =  let fun K a⃗ =  if rho.has (R, a⃗)
                                                  then ()
                                                  else raise FF
                                   in app K (unifiable (η, x⃗))
                                   end
  | execute 1 η                =  ()
  | execute (cl₁ ∧ cl₂) η      =  execute cl₁ η; execute cl₂ η
  | execute (pre ⇒ cl) η       =  check (pre, execute cl) η
  | execute (∀x : cl) η        =  execute cl ((x, NONE)::η)
```

TABLE IV: Pseudo SML code for the ALFP checker.

and thus is applied to a clause cl and a partial environment η. Once more the pseudo code is defined in Table IV and again most of the cases follow the overall pattern of Table III so we shall only comment on the case of assertions $R(\vec{x})$. Here we first construct a continuation `K` that checks whether a tuple belongs to R and then, using the `app` function, it is applied to all the tuples that are unifiable with $\vec{x}$ relative to η.

The relationship between Tables III and IV can be stated as follows:

PROPOSITION 4. *Assume that the global data structure* `rho` *contains the solution ρ. Then for all pre-conditions pre, clauses cl and sets $\mathcal{E}$ of partial environments, the following hold:*

- $\mathcal{T}_\rho[\![pre]\!]\ \mathcal{E}$ *equals the set of partial environments which on some call* `check` $(pre, \mathtt{K})\,\eta$, $\eta \in \mathcal{E}$, *are propagated to the continuation* `K`*; and*
- $\mathcal{T}^\bullet_\rho[\![cl]\!]\mathcal{E}$ *holds if and only if all calls* `execute` $cl\ \eta$, $\eta \in \mathcal{E}$, *terminate with* `()`.

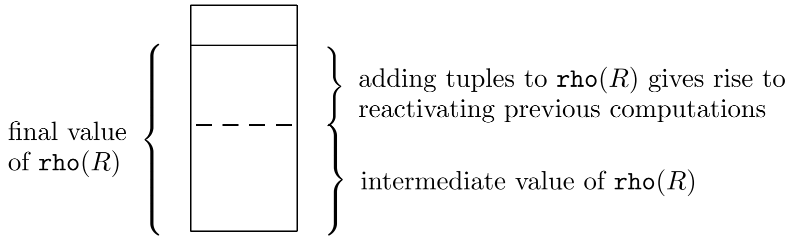

Fig. 1: Previous computations are resumed when new tuples are added.

4. The ALFP Solver

The solver differs from the checker in that it additionally constructs the interpretation ρ of the predicate symbols. As before we maintain a global data structure `rho` corresponding to ρ; it is updated by means of side effects and upon termination of the solver it will be guaranteed that the clause is satisfied (with respect to the fixed interpretation σ_0 of variables). However, since ρ is not completely determined from the beginning it may happen that some query $R(x_1, \cdots, x_k)$ inside a pre-condition fails to be satisfied at the given point in time, but may hold in the future when a new tuple $(a_1, \cdots, a_k)$ has been added to the interpretation of R. If we are not careful we will then lose the influence that adding $(a_1, \cdots, a_k)$ to R will have on the contents of other relations. This is illustrated in Figure 1 and gives rise to introducing yet another global data structure `infl` recording computations that have to be resumed for the new tuples. We explain the details below.

4.1 The Implementation

The global data structure `rho` is used for recording the currently known tuples of predicates. Accordingly, it allows adding new tuples to predicates and, as in the previous section, checking for the presence of tuples as well as retrieving whole relations:

```
rho.add:    predicate * univ list -> unit
rho.has:    predicate * univ list -> bool
rho.sub:    predicate -> (univ list) list
```

As mentioned above it may happen that some query $R(x_1, \cdots, x_k)$ inside a pre-condition fails to to be satisfied in the current environment, but may hold in the future when some new tuples have been added to the interpretation of R. In this case, we residualise the current computation by constructing a *consumer* for R and by recording it in a global data structure called `infl`. When activated, this *consumer* will perform actions corresponding to those of the checking routine: it will call the function `unify` followed by the corresponding continuation of the `check` routine. Accordingly, whenever the function `execute` adds a new tuple $(a_1, \cdots, a_k)$ to the interpretation of R,

```
fun check (R(x⃗), K) η       =  let fun K' a⃗ =  case unify(η, x⃗, a⃗) of
                                                   NONE -> ()
                                                 | SOME η' -> K(η')
                                in (infl.register(R, K');
                                      app K' (rho.sub R))
                                end
  | check (¬R(x⃗), K) η       =  let fun K' a⃗ =  if rho.has(R, a⃗)
                                                 then ()
                                                 else K (unify(η, x⃗, a⃗))
                                in app K' (unifiable (η, x⃗))
                                end
  | check (pre1 ∧ pre2, K) η  =  check (pre1, check (pre2, K)) η
  | check (pre1 ∨ pre2, K) η  =  check (pre1, K) η; check (pre2, K) η
  | check (∃x : pre, K) η     =  check (pre, K ∘ tl) ((x, NONE) :: η)
  | check (∀x : pre, K) η     =  let fun check' [] ((x, _) :: η') = K(η')
                                       | check' (a :: U) ((x, _) :: η')
                                          = check (pre, check' U) ((x, SOME a) :: η')
                                in check' 𝒰 ((x, NONE) :: η)
                                end
-------------------------------------------------------------------------------
fun execute (R(x⃗)) η         =  let fun K a⃗ =  if rho.has(R, a⃗)
                                                then ()
                                                else (rho.add(R, a⃗);
                                                      app (fn K' => K' a⃗)
                                                          (infl.consumers R))
                                in app K (unifiable (η, x⃗))
                                end
  | execute 1 η               =  ()
  | execute (cl1 ∧ cl2) η     =  execute cl1 η; execute cl2 η
  | execute (pre ⇒ cl) η      =  check (pre, execute cl) η
  | execute (∀x : cl) η       =  execute cl ((x, NONE) :: η)
```

TABLE V: Pseudo SML code for the ALFP solver.

the list of consumers waiting for new tuples of R is activated, and thereby the corresponding computations are resumed. The operations on the data structure `infl` are as follows:

```
type infl         =  (* the data structure of waiting consumers *)
type consumer     =  univ list -> unit

infl.register     :  predicate * consumer -> unit
infl.consumers    :  predicate -> consumer list
```

Both `rho` and `infl` will grow monotonically as time progresses. As was the case for the checker in the previous section, the function `check` is applied to a pair of a pre-condition *pre* and a continuation `K` together with a partial environment η. Most cases of the definition in Table V are as before. For a query $R(\vec{x})$ we construct a continuation $\mathtt{K}'$ as before; however, it is registered as a potential consumer of new tuples of R before being called for all tuples which already are included in the interpretation of R.

Before turning to negated queries, we shall observe that according to the last-in-first-out discipline of the runtime stack of SML function calls, the solver only proceeds to the execution of the next stratum of a clause, when the execution of the previous strata together with all triggered calls to continuations have finished. Consequently, the solver stabilises all predicates relative to all clauses considered so far, implying that the clauses of the next stratum in the input clause are not processed before all predicates in the previous ones have definitely stabilised. This means that our solving strategy naturally respects the stratification in the sense that no tuple is added to a relation of rank j before all relations of rank less than j have obtained their final content. Thus, for dealing with negated queries $\neg R(\vec{x})$, no registration of consumers is necessary.

Turning to the function `execute`, it is applied to a clause *cl* and a partial environment η as was also the case for the checker in the previous section. Most cases of the definition in Table V are as before. For an assertion $R(\vec{x})$ we consider as before all tuples $\vec{a}$ that are unifiable with $\vec{x}$ in the environment η; however, if $\vec{a}$ is *new* then we add it to the relation for R and we make sure to apply each of the consumers $\mathtt{K}'$ that have been registered for R to $\vec{a}$.

Example 5. Returning to Example 4 let us see how the solver computes the transitive closure T of a predicate E. Assume that initially the data structures `rho` and `infl` are as follows

$$\begin{array}{llll} \texttt{rho}(\mathsf{E}): & [(a,b),(b,c),(c,b)] & \texttt{infl}(\mathsf{E}): & [\,] \\ \texttt{rho}(\mathsf{T}): & [\,] & \texttt{infl}(\mathsf{T}): & [\,] \end{array}$$

so that in particular there are no consumers registered with any of the predicates. We are interested in the result of the call:

$$\texttt{execute}\ (\ \forall x : \forall y : \mathsf{E}(x,y) \Rightarrow (\mathsf{T}(x,y) \wedge (\forall z : \mathsf{T}(y,z) \Rightarrow \mathsf{T}(x,z)))\ )\ [\,]$$

This gives rise to the call:

$$\begin{array}{l} \texttt{check}\ (\ \mathsf{E}(x,y),\ \mathtt{K}_{\mathsf{E}}\ )\ ((y,\Box)::(x,\Box)) \\ \text{where } \mathtt{K}_{\mathsf{E}} = \texttt{execute}(\ \mathsf{T}(x,y) \wedge (\forall z : \mathsf{T}(y,z) \Rightarrow \mathsf{T}(x,z))\ ) \end{array}$$

For the sake of readability we write $\Box$ for `NONE` and omit the constructor `SOME` below. At this stage we register a consumer for E

$$\texttt{infl}(\mathsf{E}):\quad [\ \texttt{fn}\ (a_x,a_y)\ \texttt{=>}\ \mathtt{K}_{\mathsf{E}}\ ((y,a_y)::(x,a_x))\ ]$$

to take care of potential future additions to E and furthermore, we call this consumer on the pairs of `rho`(E) one at a time.

Let us consider the pair (a,b). First the conjunct $\mathsf{T}(x,y)$ gives rise to updating the `rho` data structure for T with the pair (a,b) and since no consumers have been registered with T so far no further actions are triggered. Next the conjunct $\forall z : \mathsf{T}(y,z) \Rightarrow \mathsf{T}(x,z)$ gives rise to updating the `infl` data structure so we get

$$\texttt{rho}(\mathsf{T}):\quad [(a,b)] \qquad\qquad \texttt{infl}(\mathsf{T}):\quad [\,\mathtt{K}_{ab}\,]$$

where:

$$\mathtt{K}_{ab} \;=\; \mathtt{fn}\ (a_y, a_z)\ \mathtt{=>}\ \mathtt{if}\ a_y = b\ \mathtt{then\ execute}\,(\mathsf{T}(x,z))\,((z,a_z)::(y,b)::(x,a))\ \mathtt{else}\ ()$$

Informally, when $\mathtt{K}_{ab}$ is called on a pair (b,z) it constructs a new pair (a,z) to be included in $\mathtt{rho}(\mathsf{T})$. At this point $\mathtt{K}_{ab}$ is called on the current pairs of $\mathtt{rho}(\mathsf{T})$ but this has no effect since $\mathtt{rho}(\mathsf{T})$ does not yet contain any pairs with b as first component.

Consider now the pair (b,c) in $\mathtt{rho}(\mathsf{E})$. First the conjunct $\mathsf{T}(x,y)$ of the formula gives rise to updating the `rho` data structure for T with (b,c), this triggers the application of the consumer $\mathtt{K}_{ab}$ associated with T on (b,c) and this causes the pair (a,c) to be added to $\mathtt{rho}(\mathsf{T})$ as well. The consumer $\mathtt{K}_{ab}$ is also called on (a,c) but this has no effect. Next the conjunct $\forall z : \mathsf{T}(y,z) \Rightarrow \mathsf{T}(x,z)$ gives rise to the construction of a new consumer $\mathtt{K}_{bc}$ associated with T in the `infl` data structure so we get

$$\mathtt{rho}(\mathsf{T}) :\ [(a,b),(b,c),(a,c)] \qquad \mathtt{infl}(\mathsf{T}) :\ [\mathtt{K}_{ab},\mathtt{K}_{bc}]$$

where:

$$\mathtt{K}_{bc} \;=\; \mathtt{fn}\ (a_y, a_z)\ \mathtt{=>}\ \mathtt{if}\ a_y = c\ \mathtt{then\ execute}\,(\mathsf{T}(x,z))\,((z,a_z)::(y,c)::(x,b))\ \mathtt{else}\ ()$$

As before, when $\mathtt{K}_{bc}$ is called on a pair (c,z) it constructs a new pair (b,z) to be included in $\mathtt{rho}(\mathsf{T})$. At this point $\mathtt{K}_{bc}$ is called on the current pairs of $\mathtt{rho}(\mathsf{T})$ but has no effect since $\mathtt{rho}(\mathsf{T})$ does not yet contain any pairs with c as first component.

Finally consider the pair (c,b) in $\mathtt{rho}(\mathsf{E})$. First the conjunct $\mathsf{T}(x,y)$ of the formula ensures that (c,b) is added to $\mathtt{rho}(\mathsf{T})$ and the consumers $\mathtt{K}_{ab}$ and $\mathtt{K}_{bc}$ currently associated with T will ensure that also (b,b) is added to $\mathtt{rho}(\mathsf{T})$. The two consumers are also called on (b,b) but this has no effect. Next the second conjunct $\forall z : \mathsf{T}(y,z) \Rightarrow \mathsf{T}(x,z)$ gives rise to the construction of a new consumer $\mathtt{K}_{cb}$ associated with T; at this stage the `rho` and `infl` data structures are

$$\mathtt{rho}(\mathsf{T}) :\ [(a,b),(b,c),(a,c),(c,b),(b,b)] \qquad \mathtt{infl}(\mathsf{T}) :\ [\mathtt{K}_{ab},\mathtt{K}_{bc},\mathtt{K}_{cb}]$$

where:

$$\mathtt{K}_{cb} \;=\; \mathtt{fn}\ (a_y, a_z)\ \mathtt{=>}\ \mathtt{if}\ a_y = b\ \mathtt{then\ execute}\,(\mathsf{T}(x,z))\,((z,a_z)::(y,b)::(x,c))\ \mathtt{else}\ ()$$

As before, when $\mathtt{K}_{cb}$ is called on a pair (b,z) it constructs a new pair (c,z) to be included in $\mathtt{rho}(\mathsf{T})$. At this point $\mathtt{K}_{cb}$ is called on the current pairs of $\mathtt{rho}(\mathsf{T})$ and this gives rise to adding (c,c) to $\mathtt{rho}(\mathsf{T})$:

$$\mathtt{rho}(\mathsf{T}) :\ [(a,b),(b,c),(a,c),(c,b),(b,b),(c,c)]$$

The three consumers $\mathtt{K}_{ab}$, $\mathtt{K}_{bc}$ and $\mathtt{K}_{cb}$ of $\mathtt{infl}(\mathsf{T})$ are now called on the new pair (c,c) but this has no effect. This completes the computation of the transitive closure of the predicate E. □

4.2 Optimising Tail Recursion

To obtain good running times of the solver it is essential to exploit the use of tail recursion to the fullest. Indeed, at queries $R(\vec{x})$, the function `check` calls the function `app` *tail-recursively* — meaning that the stack frame for an original call of `check` can be re-used by this final call. The implementation of `app`, however, as provided by version 110.0.7 of the SML standard library is given by:

```
fun app f []      = ()
  | app f (x::xs) = (f x; app f xs)
```

Thus, the call of f for the last element in the list is *not* a tail call — which means that the stack is not trimmed as early as possible. The better implementation of `app` which is therefore appropriate here adds an extra case for one-element lists:

```
fun app f []      = ()
  | app f [x]     = f x
  | app f (x::xs) = (f x; app f xs)
```

Now, the last call of `f` for a (non-empty) list is a tail call — implying that the stack frame for `app` (which was taken from the call to the function `check`) is now re-used by the last call to the function `f`. A similar argument shows that the modified `app` also improves the consumption of stack space in the nested application of `app` at assertions. Experiments indicate that this optimisation reduces runtimes considerable in those cases where space consumption is critical.

4.3 Prefix Trees

Two problems withstand a rapid propagation of new tuples of a predicate R to all places where they are queried:

- Given an argument list $(x_1, \cdots, x_k)$ together with a partial environment η how do we efficiently find all the tuples of a predicate that possibly are unifiable with $(x_1, \cdots, x_k)$ relative to η?
- How do we maintain the consumers in such a way that they are easy to register and such that all registered consumers that are potentially applicable to new tuples can be accessed efficiently?

The first problem is concerned with the efficiency of the application of the `app`-function in the definition of the `check`-function on queries of the form $R(x_1, \cdots, x_k)$ in Table V. The second problem is concerned with the efficiency of the inner occurrence of the `app`-function in the definition of the `execute`-function on assertions of the form $R(x_1, \cdots, x_k)$.

Different consumers for the same tuple $(a_1, \cdots, a_k)$ of a k-ary predicate may correspond to queries where different subsets of argument positions are bound in the environment. An obvious idea therefore consists in grouping the consumers according to these subsets. There could, however, be as many as 2^k non-empty subsets of consumers waiting to unify with $(a_1, \cdots, a_k)$. Organizing and maintaining access to these is awkward and incurs overhead.

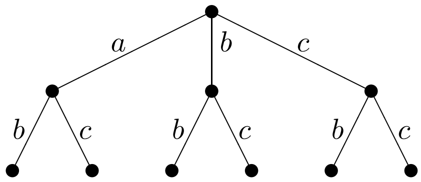

Fig. 2: Prefix tree for $\texttt{rho}(\mathsf{T}) = [(a,b),(b,c),(a,c),(c,b),(b,b),(c,c)]$.

In order to solve the two problems, let us for a moment trade efficiency against generality and abandon efficient support for *all* possible query patterns. Instead, we arrange the set of all currently known tuples of the k-ary predicate R into a *prefix tree.* The prefix tree can be implemented, e.g., by using an *extensible array* to store for each node v of the tree the list of all currently available successor atoms, together with a *hash table* to map pairs $\langle v, a\rangle$ of nodes v and atoms a to successor nodes.

Example 6. Returning to Example 5 we represent the predicate $\texttt{rho}(\mathsf{T}) = [(a,b),(b,c),(a,c),(c,b),(b,b),(c,c)]$ using the prefix tree of Figure 2. □

The tree representation allows us to implement a function `rho.subS` which, given a predicate R and a prefix $(a_1,\cdots,a_i)$, rapidly enumerates all suffixes $(a_{i+1},\cdots,a_k)$ satisfying that $(a_1,\cdots,a_i,a_{i+1},\cdots,a_k)$ is contained in $\texttt{rho}(R)$:

```
rho.subS :   predicate * univ list -> univ list
```

Given this function, we can replace the occurrence of

```
app K′ (rho.sub  R)
```

in the definition of the `check`-function for $R(\vec{x})$ with

```
let val a⃗₁ = first (η, x⃗)
in app (fn a⃗₂ => K′ (a⃗₁ @ a⃗₂)) (rho.subS (R, a⃗₁))
end
```

where we write $\vec{a}_1 @ \vec{a}_2$ for the concatenation of two tuples and where the function `first` takes a partial environment η and a tuple $\vec{x}$ of arguments and returns the unique tuple $\vec{a}_1$ of atoms corresponding to the maximal prefix of $\vec{x}$ whose variables are all instantiated in η.

Example 7. Returning to Example 5 let us assume that $\texttt{rho}(\mathsf{E})$ additionally contains the pair (d,a). The first conjunct $\mathsf{T}(x,y)$ of the formula will add (d,a) to $\texttt{rho}(\mathsf{T})$ and then call the consumers $\mathtt{K}_{ab}$, $\mathtt{K}_{bc}$ and $\mathtt{K}_{cb}$ on (d,a) but they will have no effect. Next the second conjunct $\forall z : \mathsf{T}(y,z) \Rightarrow \mathsf{T}(x,z)$ will construct a new consumer $\mathtt{K}_{da}$

```
K_da  =  fn (a_y, a_z) => if a_y = a
                          then execute (T(x,z)) ((z,a_z) :: (y,a) :: (x,d))
                          else ()
```

to be associated with T and $\mathtt{K}_{da}$ will be applied to all the pairs currently in rho(T). The optimised algorithm realises that only pairs with a as first component are of interest and hence `rho.subS` only returns the list $[b, c]$ of potential second components so that the consumer $\mathtt{K}_{da}$ is only applied to the pairs (a, b) and (a, c) (thereby adding (d, b) and (d, c) to rho(T)). □

Turning to the consumers we now observe that we also need to let the consumers of a k-ary predicate R register for the set of *suffixes* for a given prefix. This implies that there are at most $k+1$ sets of consumers waiting for each new tuple $(a_1, \cdots, a_k)$ of R. Also, the waiting consumers are maintained in a *prefix tree* of depth at most k whose nodes correspond to the possible prefixes $(a_1, \cdots, a_i)$. In particular, all potential consumers of a single tuple $(a_1, \cdots, a_k)$ can be collected by traversing one branch in the tree. This data structure of waiting consumers can be implemented again by an *extensible array* and a *hash table*. As before, the hash table maps pairs $\langle v, a\rangle$ of nodes v and atoms a to successor nodes in the tree. The extensible array, however, now returns for each node in the tree the set of consumers waiting for suffixes. This functionality is provided by the two functions `infl.registerS` and `infl.consumersS` whose types are:

```
infl.registerS   :  predicate * consumer * univ list -> unit
infl.consumersS  :  predicate * univ list -> consumer list
```

Given these functions, we can replace the occurrence of

```
infl.register (R,K')
```

in the definition of the `check`-function for $R(\vec{x})$ with

```
let val a⃗1 = first (η, x⃗)
in infl.registerS (R, K', a⃗1)
end
```

Furthermore, the occurrence of

```
app (fn K' => K' a⃗) (infl.consumers R)
```

in the definition of the `execute`-function for $R(\vec{x})$ can be replaced with

```
app (fn K' => K' a⃗) (infl.consumersS (R, a⃗))
```

Assuming that hash table lookups and insertions can be done in time $\mathcal{O}(1)$, we then conclude that collecting all consumers for a tuple $(a_1, \cdots, a_k)$ of a predicate R amounts to $k+1$ lookups in the prefix tree and a concatenation of $k+1$ sets of consumers. Thus, it can be executed in time $\mathcal{O}(k+1+n)$ where k is the arity of the predicate R, and n is the size of the resulting set of consumers.

EXAMPLE 8. The consumers of the final version of infl(T) in Example 5 can be arranged into the prefix tree of Figure 3. Let us now assume that rho(E) additionally contains the pair (b, d). The clause $\mathsf{T}(x, y)$ then ensures that it is added to rho(T) and in the optimised algorithm only consumers corresponding to the prefix b will be called, i.e. only $\mathtt{K}_{ab}$ and $\mathtt{K}_{cb}$. □

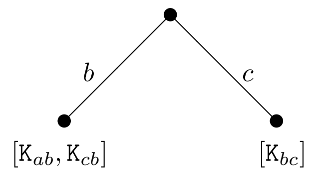

Fig. 3: Prefix tree for `infl`(T) = $[\mathtt{K}_{ab}, \mathtt{K}_{bc}, \mathtt{K}_{cb}]$.

4.4 Memoisation

While evaluating a call `check` $(pre, \mathtt{K})\,\eta$, the continuation K may be called more than once for the same partial environment η'. This may happen at disjunctions $pre_1 \vee pre_2$ where η' satisfies both pre_1 and pre_2. It may also happen at existential quantifications $\exists x : pre$ where pre can be satisfied for different values of x in $(x, \cdots) :: \eta'$.

In order to avoid this unnecessary duplication of work, we may add memoisation at every occurrence of a pre-condition pre; this amounts to recording the set of environments η' for which the given continuation has already been called. This scheme then can be optimised (for space) in such a way that we avoid the extra book-keeping in as many places as is safely possible. In practice, we place memoisation only at outermost occurrences of disjunctions or existential quantifications.

In particular, we omit memoisation at conjunctions. The reason is twofold:

- In absence of disjunctions or existential quantifications, no duplication of work can occur.
- If memoisation occurs at all disjunctions and existential quantifications, then every continuation K is called with the same environment η' only a bounded number of times.

As an illustrating example, consider the clause:

$$R(a) \wedge S(a,a) \wedge \forall x : \forall y : (R(x) \vee R(y)) \wedge S(x,y) \Rightarrow T(x,y)$$

and assume we have memoisation at existentials and disjunctions. The two environments generated by checking the disjunction are:

$$\eta_1 = (x,a) :: (y,\Box) \quad \text{and} \quad \eta_2 = (x,\Box) :: (y,a)$$

When propagated to the query $S(x,y)$ we perform unification with the tuple (a,a) and in both cases obtain the environment:

$$\eta = (x,a) :: (y,a)$$

In general, all η' which, after a query, result in the same environment η must agree on all jointly instantiated variables and thus may differ only on the set of variables so far uninstantiated. As the formula has bounded size, these are just boundedly many.

4.5 Estimating the Complexity

For the following, we assume that the algorithm is equipped with a memoisation scheme such that no continuation is called twice with the same argument. We also assume that the data structures explained in this section for implementing `rho` are also used by the checker.

We now state the fundamental observation that computing the solution to a clause is no more expensive, asymptotically, than checking the result:

PROPOSITION 5. *Assume that* cl *is an ALFP formula of constant size. Let* ρ_0 *denote an interpretation and* ρ *the optimal solution of* cl *exceeding* ρ_0 *(i.e.* $\rho_0 \subseteq \rho$*). Let* t_0 *and* t *be the runtimes of the solver when started on* ρ_0 *and* ρ*, respectively, and let* t' *be the runtime of the checker when started on* ρ*.*

Then $t_0 \leq c \cdot t$ *and* $t \leq c' \cdot t'$ *for constants* c *and* c' *independent of the relations in* ρ*.*

PROOF. Let π_0 and π be the fixpoint computations of the solver when started on ρ_0 and ρ, respectively, and let π' be the computations of the checker when started on ρ.

The only actions of π not performed by π' are the registration of consumers; however, these are dominated by the number of calls on `check` and hence by the runtime of π'. Hence t may exceed t' only by a constant factor.

Concerning π and π_0 the following holds:

- ∘ The fixpoint computation π_0 performs the same calls `check` $(pre, \texttt{K})\,\eta$, `execute` $cl\,\eta$, `infl.register` $(\eta, \vec{x}, \vec{a})$, and `unify` $(\eta, \vec{x}, \vec{a})$ as π — only, perhaps, in a different ordering.
- ∘ The only additional work of π_0 consists in determining the sets of consumers for the tuples newly added to the relations.

According to the implementation of `infl.consumersS` as described in subsection 4.3 and our assumption that the size of cl, and therefore also all arities of occurring predicates, are $\mathcal{O}(1)$, we know that the runtime of each individual call `infl.consumersS` $(R, \vec{a})$ is proportional to 1 plus the number of returned consumers. Thus, the total additional time spent by π_0 is $\mathcal{O}(P+Q)$ where P equals the total number of calls `infl.consumersS` $(R, \vec{a})$, and Q equals the total number of returned consumers. The value P is bounded by the total number of calls `execute` $cl'\,\eta$ where cl' is an assertion. As each consumer causes a call to `unify`, we deduce that Q is bounded by the total number of calls to `unify`. Consequently, the sum $P+Q$ is bounded by the total runtime of the solver on ρ — implying that the time t_0 may exceed the time t only by a constant factor. □

A proposition similar to Proposition 5 has been stated by McAllester for his algorithm and ordinary Horn clauses [19]. Thanks to this proposition, it suffices to determine the complexity of the solver for *checking* a solution. And in fact, the corresponding work incurred by different syntactical components of the clause can be calculated in a compositional manner.

$$
\begin{array}{lcl}
\mathcal{C}_\rho[\![R(\vec{x})]\!]\,\mathcal{E} & = & \#\{\langle \eta, \vec{a}\rangle \mid \eta \in \mathcal{E}, (\texttt{first}\,(\eta,\vec{x}))@\vec{a} \in \rho(R)\} \\
\mathcal{C}_\rho[\![\neg R(\vec{x})]\!]\,\mathcal{E} & = & \sum_{\eta\in\mathcal{E}} (\#\mathcal{U})^{\#\texttt{free}(\eta,\vec{x})} \\
\mathcal{C}_\rho[\![pre_1 \wedge pre_2]\!]\,\mathcal{E} & = & \mathcal{C}_\rho[\![pre_1]\!]\,\mathcal{E} + \mathcal{C}_\rho[\![pre_2]\!]\,(\mathcal{T}_\rho[\![pre_1]\!]\,\mathcal{E}) \\
\mathcal{C}_\rho[\![pre_1 \vee pre_2]\!]\,\mathcal{E} & = & \mathcal{C}_\rho[\![pre_1]\!]\,\mathcal{E} + \mathcal{C}_\rho[\![pre_2]\!]\,\mathcal{E} \\
\mathcal{C}_\rho[\![\exists x : pre]\!]\,\mathcal{E} & = & \mathcal{C}_\rho[\![pre]\!]\,((x,\Box)\,\hat{::}\,\mathcal{E}) \\
\mathcal{C}_\rho[\![\forall x : pre]\!]\,\mathcal{E} & = & \mathcal{C}_{\rho,\mathcal{U}}[\![pre]\!]\,((x,\Box)\,\hat{::}\,\mathcal{E}) \\
 & & \text{where}\quad \mathcal{C}_{\rho,a::U}[\![pre]\!]\,((x,d)\,\hat{::}\,\mathcal{E}) \\
 & & \qquad = \mathcal{C}_\rho[\![pre]\!]\,((x,a)\,\hat{::}\,\mathcal{E}) \\
 & & \qquad\quad + \mathcal{C}_{\rho,U}[\![pre]\!]\,(\mathcal{T}_\rho[\![pre]\!]\,((x,a)\,\hat{::}\,\mathcal{E})) \\
 & & \quad\;\; \mathcal{C}_{\rho,[\,]}[\![pre]\!]\,\mathcal{E} = \#\mathcal{E} \\
\hline
\mathcal{C}^\bullet_\rho[\![R(\vec{x})]\!]\,\mathcal{E} & = & \#\{\langle \eta, \vec{a}\rangle \mid \eta \in \mathcal{E}, \vec{a} \in \rho(R), \texttt{unify}\,(\eta,\vec{x},\vec{a}) \neq \mathsf{fail}\} \\
\mathcal{C}^\bullet_\rho[\![\mathbf{1}]\!]\,\mathcal{E} & = & \#\mathcal{E} \\
\mathcal{C}^\bullet_\rho[\![cl_1 \wedge cl_2]\!]\,\mathcal{E} & = & \mathcal{C}^\bullet_\rho[\![cl_1]\!]\,\mathcal{E} + \mathcal{C}^\bullet_\rho[\![cl_2]\!]\,\mathcal{E} \\
\mathcal{C}^\bullet_\rho[\![pre \Rightarrow cl]\!] & = & \mathcal{C}_\rho[\![pre]\!]\,\mathcal{E} + \mathcal{C}^\bullet_\rho[\![cl]\!]\,(\mathcal{T}_\rho[\![p]\!]\,\mathcal{E}) \\
\mathcal{C}^\bullet_\rho[\![\forall x : cl]\!]\,\mathcal{E} & = & \mathcal{C}^\bullet_\rho[\![cl]\!]\,((x,\Box)\hat{::}\mathcal{E})
\end{array}
$$

TABLE VI: The costs of pre-conditions and clauses.

Based on the functions $\mathcal{T}_\rho$ and $\mathcal{T}^\bullet_\rho$ of Section 3, we define in Table VI the function $\mathcal{C}_\rho$ for calculating the costs incurred by pre-conditions, and the function $\mathcal{C}^\bullet_\rho$ for calculating the costs incurred by clauses, on any set of arriving environments. Here, `first` is as before and $\texttt{free}\,(\eta, \vec{x})$ returns the set of variables from $\vec{x}$ which are not instantiated in η.

Note that we essentially count for every pre-condition, the number of satisfying environments. This generalises the intuitive idea of counting *prefix firings* in the manner used by McAllester for his algorithm and the special case of Horn clauses [19]. In case of queries, our estimation additionally takes into account that our relations are stored in prefix trees — thus we only support rapid enumeration of sub-relations for given prefixes of tuples. In case of negated queries, we take into account that the solver has to enumerate all possible instantiations of the variables which are not already instantiated in the environment. In case of universal quantifications, we calculate the costs of iterating through all possible bindings for the bound variable.

Our first main theorem concerns the checker; note that it could not be stated until the introduction of prefix trees:

THEOREM 1. *Assume that cl is an ALFP formula of constant size. Then* *the solution ρ can be checked by the checker of Table IV in time*

$$\mathcal{O}(\#\rho + \mathcal{C}^{\bullet}_{\rho}[\![cl]\!]\,\mathcal{E}_0)$$

where the set $\mathcal{E}_0$ consists of the single initial environment [ ].

For a formal proof see Appendix C. We now state a similar theorem for the solver:

THEOREM 2. *Assume that cl is an ALFP formula of constant size. Then the least solution ρ exceeding some interpretation ρ_0 (i.e. $\rho_0 \subseteq \rho$) can be computed by the solver of Table V in time*

$$\mathcal{O}(\#\rho + \mathcal{C}^{\bullet}_{\rho}[\![cl]\!]\,\mathcal{E}_0)$$

where the set $\mathcal{E}_0$ consists of the single initial environment [ ].

This result is an immediate corollary of Theorem 1 and Proposition 5. These theorems demonstrate that it is not only possible to succinctly express algorithmic insights in functional languages but that it is also possible to elegantly perform the necessary complexity calculations.

5. Pragmatics

We have implemented the solver for ALFP formulae using SML of New Jersey. In order to experiment with the solver we have implemented a number of frontends for various programming calculi. Here, we shall report on the highlights of some experiments performed for Discretionary Ambients [24] based on the analysis of Subsection 2.4; we refer to [5] for the full set of experiments.

We shall report on three groups of experiments: (1) rearrangement of arguments of predicates, (2) rearrangements of conjunctions in pre-conditions and (3) limiting the scope of variables. Similar kinds of "optimisations" have been studied quite extensively also related to BDD- and SAT-based model checking [2], logic programming and deductive databases [11, 28].

5.1 Experimental Set-up

The analyses are based on the specification of Subsection 2.4 and here we consider a scalable program where a single packet `p` is routed through a network of $m \times m$ sites, each site $\mathtt{s}_{i,j}$ contains a routing table that non-deterministically directs the packet to site $\mathtt{s}_{i+1,j}$ (if $i < m$) or to site $\mathtt{s}_{i,j+1}$ (if $j < m$). The cardinality of the universe of interest for a process with an $m \times m$ network is $\mathcal{O}(m^2)$ and this is also the size of the process and the size of the solution to the analysis problem. The theoretical runtimes for the analysis is cubic in the size of the universe as may be calculated using Theorem 1; this corresponds to $\mathcal{O}(m^6)$. As we shall see we can obtain substantially better results in practice by carefully tuning the clauses.

The time used by the solver is measured as the CPU-time used by the SML implementation. The execution time is split into two contributions:

- The time for the *initialisation* phase: this includes the time it takes to load the analysis clause files and to generate and represent the PRG-relation of the example process in the data structures of the solver.
- The time for *solving* the closure condition of the analysis.

We only report on the solving time *excluding* the time spent on garbage collection; in our view the execution times including garbage collection are less informative as they show great variation when the same experiment is repeated.

Having measured the execution time of the solver it is of interest to see how it relates to the size of the universe and thereby the size of the ambient process. For simplicity we assume that the execution time t for the *solving phase* can be expressed as

$$t = c \cdot m^r$$

for some number r — thereby ignoring that the measured times may contain contributions from polynomials of lower degree. In subsequent figures we shall plot corresponding values of t and m on a double logarithmic scale; in this form the formula amounts to $\log t = \log c + r \cdot \log m$. In order to estimate c and r, the measured times are fitted to this doubly algorithmic model by a least-square fit.

5.2 *The Order of Arguments of Predicates*

Recall that the solver uses prefix trees to represent relations meaning that preferences are given to certain query patterns and imposing a potential large penalty on other query patterns. If support for a further *query* pattern of a k-ary predicate R is demanded, then it can be obtained through a source-to-source transformation — similar in spirit to the second last rule of McAllester in the proof of Theorem 3 of [19]. For example, if we would like to query the ternary predicate R where the last argument is bound, we introduce the new clause

$$\forall x_1 : \forall x_2 : \forall x_3 : R(x_2, x_3, x_1) \Rightarrow R'(x_1, x_2, x_3)$$

that defines R' in terms of R and we replace the corresponding query of $R(a_2, a_3, a_1)$ by $R'(a_1, a_2, a_3)$; no other changes are necessary.

Clearly, we could tabulate for *each* occurrence of a query $R(x_1, \cdots, x_k)$ a separate copy of the relation for R where the subset of instantiated variables does not form a prefix of the sequence $x_1, \cdots, x_k$. We might, however, exploit the prefix tree implementation and try to cover several non-prefix query patterns by one suitably permuted representation of the relation. In general, we have the following optimisation problem:

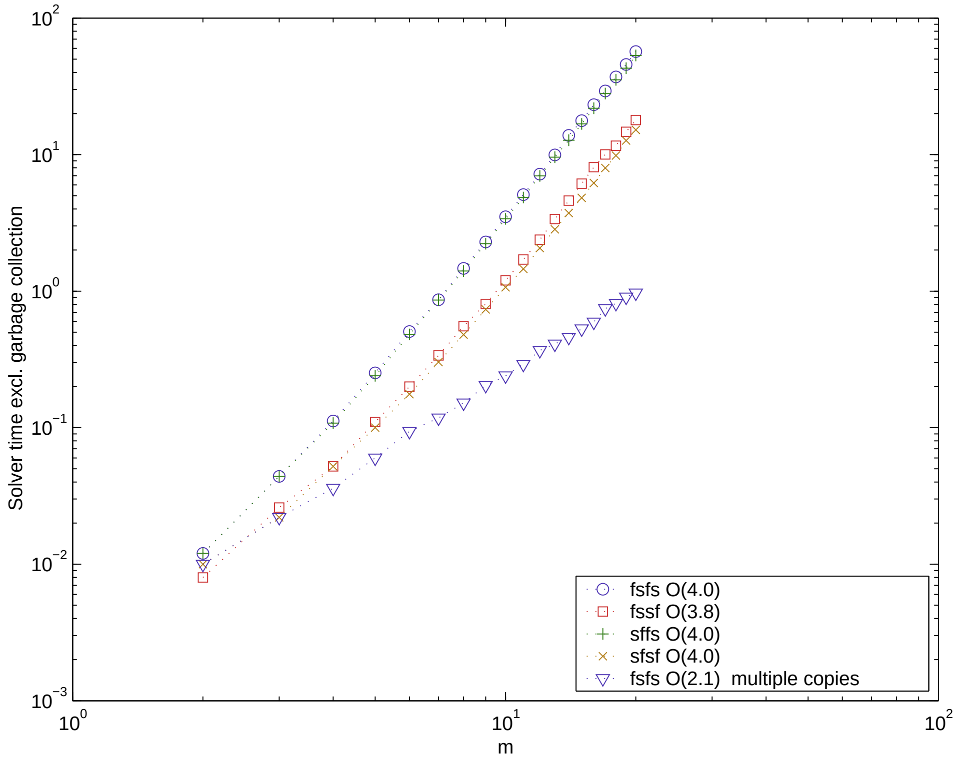

Fig. 4: The impact of reordering arguments of predicates.

> Given a set $\mathcal{S}$ of subsets $S \subseteq \{1, \cdots, k\}$ of instantiated components how do we construct a minimal set Π of permutations π of $\{1, \cdots, k\}$ such that each subset $S \in \mathcal{S}$ is a prefix of some $\pi \in \Pi$.

We use a simple greedy heuristics: We iteratively extract a maximal chain $S_1 \subset \ldots \subset S_f$ of sets in $\mathcal{S}$ for which a supporting permutation is derived. Then the chain is removed from $\mathcal{S}$. We repeat these steps until $\mathcal{S}$ is exhausted.

EXAMPLE 9. To illustrate the importance of the ordering of the arguments of predicates we have experimented with a version of the solver *without* an implementation of the above heuristics. We have considered four different versions of the analysis of Table II corresponding to permutations of the arguments of the two predicates PRG and $\mathcal{I}$. Both predicates express a father-son relationship so we have four variations:

	$\mathcal{I}$	PRG
`fsfs`	(father, son)	(father, son)
`fssf`	(father, son)	(son, father)
`sffs`	(son, father)	(father, son)
`sfsf`	(son, father)	(son, father)

For comparison we have also performed the experiment on the `fsfs` version of the analysis (corresponding to Table II) with the solver *including* the above heuristics. Figure 4 shows the result of the experiments on a double logarithmic scale: the time t is on the y-axis and the parameter m is on the x-axis. We see from Figure 4 that considerable improvements are obtained when using the solver's ability to generate and maintain multiple copies of predicates. □

5.3 The Order of Conjuncts in Pre-conditions

Pre-conditions are evaluated from left to right and in the context of an environment η. When checking a query to a predicate R the evaluation of the remainder of the pre-condition is performed for all the new environments η' obtained by unifying η with a tuple currently in R. If the unification fails then no further work is done. Thus we should try to arrange the conjuncts of a pre-conditions such that the unifications fail as early as possible.

To illustrate this consider the clause

$$\forall x : \; R(x, a) \wedge R(b, x) \Rightarrow Q(x)$$

where a and b are constants. Initially, the query $R(x, a)$ is evaluated and unification is performed with every pair in R; this will succeed for all pairs of R with a as the second component. For each of these environments $R(b, x)$ is evaluated.

Now, suppose we have a priori knowledge that the predicate R contains many pairs with a as second component but only few with b as the first component. In this case, rearranging the conjuncts in the pre-condition, i.e. the clause

$$\forall x : \; R(b, x) \wedge R(x, a) \Rightarrow Q(x)$$

will be more efficient as fewer environments are propagated from the first query to the second. This observation leads to the general optimisation strategy that queries, which restrict the variable binding most, should be performed early in order to increase efficiency. Note that this strategy may require a priori knowledge of the contents of the predicates and this knowledge may not always be available.

Example 10. To investigate this we have studied three versions of the analysis of Table II where we have varied the order in which the various parts of the redex is recognised; for the `in`-action this amounts to the following:

	first	then	and	finally
`craco`	$\mathcal{I}(\mu_a, \mathsf{in}(\mu))$	$\mathcal{I}(\mu_p, \mu_a)$	$\mathcal{I}(\mu_p, \mu)$	$\mathcal{I}(\mu, \overline{\mathsf{in}}(\mu_a, \mu))$
`coacr`	$\mathcal{I}(\mu, \overline{\mathsf{in}}(\mu_a, \mu))$	$\mathcal{I}(\mu_p, \mu)$	$\mathcal{I}(\mu_a, \mathsf{in}(\mu))$	$\mathcal{I}(\mu_p, \mu_a)$
`cocra`	$\mathcal{I}(\mu, \overline{\mathsf{in}}(\mu_a, \mu))$	$\mathcal{I}(\mu_a, \mathsf{in}(\mu))$	$\mathcal{I}(\mu_p, \mu_a)$	$\mathcal{I}(\mu_p, \mu)$

Here `coacr` is the specification of Table II and we see from Figure 5 that `craco` is a constant factor better than `cocra` but `coacr` yields a significantly

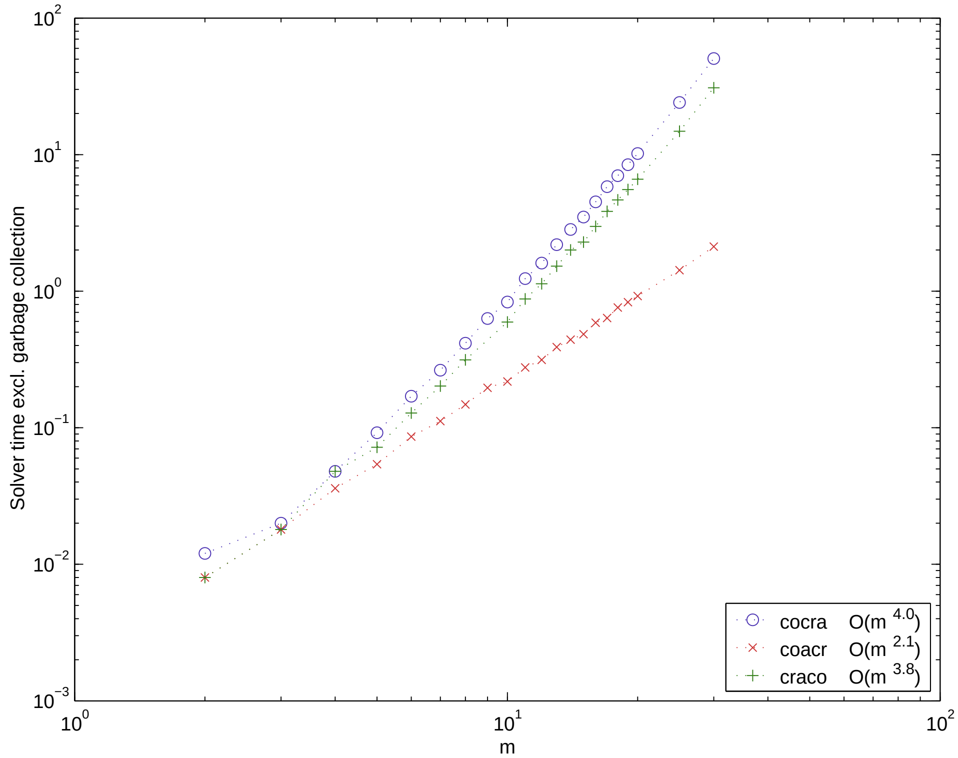

Fig. 5: The impact of reordering the conjuncts of pre-conditions.

better result. The overall results can be confirmed by manual calculations of the number of environments generated when analysing example processes.

□

5.4 Limiting the Scope of Variables

Even though a continuation K only is called once for each partial environment η we still may end up duplicating work. As an example consider the clause:

$$\forall x : \forall y : \forall z : \ P(x,y) \wedge Q(y,z) \Rightarrow R(y,z)$$

Checking $P(x,y)$ will bind x as well as y. Assume now that P contains several pairs with the same second component as e.g. (a,c) and (b,c). We are then going to propagate two environments $(z,\square) :: (y,c) :: (x,a)$ and $(z,\square) :: (y,c) :: (x,b)$ to the rest of the formula and since x does not occur in the rest of the formula we are going to do the same work twice. Hence we may consider approaches for pruning partial environments so that only the relevant parts are propagated — in the example above we only need to propagate the environment $(z,\square) :: (y,c)$.

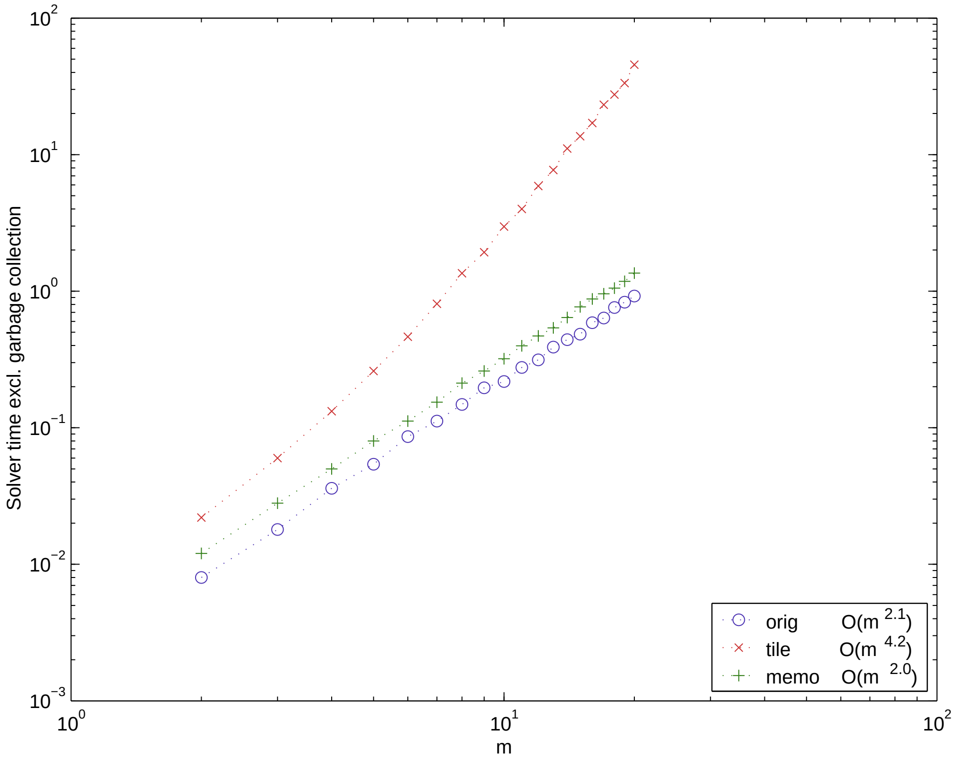

Fig. 6: The impact of tiling and memoisation.

One way of syntactically enforcing this is to introduce an auxiliary predicate P' and replace the above clause by:

$$\begin{array}{l}\forall x : \forall y : P(x,y) \Rightarrow P'(y) \\ \forall y : \forall z : \ \ P'(y) \wedge Q(y,z) \Rightarrow R(y,z)\end{array}$$

This is an instance of the tiling transformation of [23]; this is a general transformation technique on Horn clauses that may improve worst-case complexity by reducing nesting depth of quantifiers whenever there are queries in a pre-condition using variables that do not occur in the conclusion of the clause.

An alternative and logically equivalent technique is to rewrite the above clause to use an existential quantifier as in

$$\forall y : \forall z : \ \ (\exists x : P(x,y)) \wedge Q(y,z) \Rightarrow R(y,z)$$

and rely on the memoisation technique of Subsection 4.4.

Example 11. We have applied the two techniques to the analysis of Table II. In Figure 6 we display the result of the experiments with:

`orig`	the analysis of Table II
`tile`	the analysis obtained by the tiling transformation
`memo`	the analysis obtained by introducing existentials

The figure shows that the benefit of using existentials is negligible for the analysis of Table II: the slight reduction in the exponent is offset by a larger constant factor. Perhaps surprisingly the figure shows that tiling is costly in comparison. This is because in `tile` we compute the `sibling` predicate that turns out to be very large. In a sense we do the same in `memo` but here it is done in a "lazy" way by only computing the pairs of potential interest. □

6. Conclusion

Our solver algorithm is clearly based on classical work on efficient fixpoint algorithms [15]. In particular, it combines the topdown solving approach of Le Charlier and van Hentenryck [8] with the propagation of differences [14], an optimisation technique for distributive frameworks which is also known in the area of deductive databases [4] or as *reduction of strength* transformations for program optimisation [25]. For these ideas to work we had to provide arbitrarily branching *prefix trees* as a universal data-structure for storing relations as well as for organising sets of waiting consumers. The efficiency, simplicity and expressiveness of the logic made it our favourite choice over the transformational approach of McAllester [19].

The analysis of the complexity has benefitted from the pioneering ideas of McAllester [19] on the complexity of solving (classical) Horn clauses. Here, we generalised these techniques to a richer class of input formulae and adapted it to the specific properties of our solver. In doing this, we were greatly assisted by the abstract characterisation of the behaviour of the solver which again was made possible thanks to the specific programming style (in particular continuations and memoisation) being used.

As an alternative to our approach we considered using off-the-shelf implementations of deductive databases, e.g. the Coral system [26], or logic programming systems tuned to find all solutions, e.g. XSB Prolog [11]. However we found this to be a less viable approach in order to sustain our overall objective of automatic complexity analysis [22].

Acknowledgements. The experiments reported in Section 5 were carried out in collaboration with Mikael Buchholtz and were partially supported by the European Union through the SecSafe project.

References

[1] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming (SCP)*, 35(2):79–111, 1999.

[2] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Faster SAT and smaller BDDs via common function structure. In *IEEE/ACM International Conference on Computer Aided Design*, pages 443–448. IEEE Press, 2001.

[3] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative programming. In J. Minsker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan-Kaufman, Los Altos, CA, 1988.

[4] I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Programming (JLP)*, 4(3):259–262, 1987.

[5] M. Buchholtz, H. Riis Nielson, and F. Nielson. Experiments with Succinct Solvers. Technical Report IMM-TR-2002-4, Technical University of Denmark, 2002.

[6] L. Cardelli and A.D. Gordon. Mobile ambients. In *Proceedings of FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

[7] A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 25(2):156–178, 1980.

[8] B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.

[9] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.

[10] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of 6th ACM Symposium on Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[11] B. Cui and D. S. Warren. A system for tabled constraint logic programming. In *Computational Logic 2000*, volume 1861 of *LNAI*, pages 478–492. Springer-Verlag, 2000.

[12] E. Dahlhaus. Skolem normal forms concerning the least fixpoint. In *Computation Theory and Logic*, pages 101–106. LNCS 270, Springer Verlag, 1987.

[13] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[14] C. Fecht and H. Seidl. Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems. In *European Symposium on Programming (ESOP)*, pages 90–104. LNCS 1381, Springer Verlag, 1998. Long version in *Nordic Journal of Computing 5, 304-329, 1998.*

[15] C. Fecht and H. Seidl. A Faster Solver for General Systems of Equations. *Science of Computer Programming (SCP)*, 35(2-3):137–162, 1999.

[16] J. Harland. Success and failure for Hereditary Harrop formulae. *Journal of Logic Programming*, 17(1):1–29, 1993.

[17] P.G. Kolaitis. Implicit definability on finite structures and unambiguous computations (preliminary report). In *5th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 168–180, 1990.

[18] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of POPL'00*, pages 352–364. ACM Press, 2000.

[19] D. McAllester. On the Complexity Analysis of Static Analyses. In *6th Static Analysis Symposium (SAS)*, pages 312–329. LNCS 1694, Springer Verlag, 1999.

[20] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML.* MIT Press, 1997.

[21] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis.* Springer, 1999.

[22] F. Nielson, H. Riis Nielson, and H. Seidl. Automatic complexity analysis. In *European Symposium on Programming*, pages 243–261, 2002.

[23] F. Nielson and H. Seidl. Control-Flow Analysis in Cubic Time. In *European Symposium on Programming (ESOP)*, volume 2028 of *LNCS*, pages 252–268, 2001.

[24] H. Riis Nielson, F. Nielson, and M. Buchholtz. Security for Mobility. 2003. To appear.

[25] R. Paige. Symbolic Finite Differencing – Part I. In *Proceedings of 3rd European Symposium on Programming (ESOP)*, pages 36–56. LNCS 432, 1990.

[26] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL Deductive System. *VLDB Journal*, 3(2):161–210, 1994.

[27] Joseph E. Stoy. *Denotational Semantics: The Scott–Strachey approach to programming language semantics.* MIT Press, 1977.

[28] S. Sudarshan. *Optimizing Bottom-up Query Evaluation for Deductive Databases.* PhD thesis, University of Wisconson-Madison, 1992.

Appendix A. Proof of Proposition 1

Assume cl has the form $cl_1 \wedge \cdots \wedge cl_s$ where cl_j is the clause corresponding to stratum j, and let $\mathcal{R}_j$ denote the set of all relation symbols R defined in $cl_1 \wedge \cdots \wedge cl_j$ taking $\mathcal{R}_0 = \emptyset$.

Let M denote a set of assignments which map relation symbols to relations; then $\rho = \bigsqcap M$ is given by the formula

$$\rho(R) = \bigcap\{\rho'(R) \mid \rho' \in M \wedge \forall R' \in \mathcal{R}_{rank(R)-1} : \rho(R') = \rho'(R')\}$$

which is well-defined by induction on the value of $rank(R)$.

Example 12. Consider the setting where $\mathcal{U} = \{a,b,c,d\}$, $\mathcal{R}_1 = \{R\}$, $\mathcal{R}_2 = \{S\}$ and $\rho_1(R) = \{a,b\}$, $\rho_1(S) = \{a,b\}$, $\rho_2(R) = \{a,c\}$ and $\rho_2(S) = \{a,c\}$. Then $(\rho_1 \sqcap \rho_2)(R) = \{a\}$ and $(\rho_1 \sqcap \rho_2)(S) = \{a,b,c,d\}$. □

We prove that for all j, all M and all variable environments σ:

Lemma 1. *If $\rho = \bigsqcap M$, c occurs in cl_j and $(\rho',\sigma) \models c$ for all $\rho' \in M$ then also $(\rho,\sigma) \models c$.*

Proof We proceed by induction on j and in each case distinguish between two cases.

- ∘ The first case is when $\rho(R) = \mathcal{U}^k$ for *all* relations R of rank j and appropriate arity k. In this case a straightforward induction on c suffices for proving that $(\rho,\sigma) \models c$ holds for all c occurring in cl_j. We consider two illustrative cases:
  - If c is $R(x_1,\cdots,x_k)$ then the result is immediate since we assumed that $\rho(R) = \mathcal{U}^k$.
  - If c is $p \Rightarrow c'$ then the induction hypothesis gives $(\rho,\sigma) \models c'$ and hence it is immediate that also $(\rho,\sigma) \models c$.
- ∘ The second case is when $\rho(R) \neq \mathcal{U}^k$ for *some* relation R of rank j and some arity k. Then the set

  $$M_j = \{\rho' \in M \mid \forall R' \in \mathcal{R}_{j-1} : \rho(R') = \rho'(R')\}$$

  is non-empty and we have:

  $$\begin{array}{lcll} \rho(R) & = & \bigcap\{\rho'(R) \mid \rho' \in M_j\} & \text{if } rank(R) = j \\ \rho(R) & = & \rho'(R) & \text{if } rank(R) < j \text{ and } \rho' \in M_j \end{array}$$

We proceed by structural induction on c (recalling that c occurs in cl_j) and establish an auxiliary result for pre-conditions.

- If c is $R(x_1, \cdots, x_k)$ then, using that $rank(R) = j$, we have that $\rho(R)$ equals the intersection of all relations $\rho'(R)$ for $\rho' \in M_j$. Given the assumption that $t = (\sigma(x_1), \cdots, \sigma(x_k)) \in \rho'(R)$ for all $\rho' \in M_j \subseteq M$ we have that $t \in \rho(R)$ as desired.
- If c is $p \Rightarrow c'$ then we have two cases. In the first case, $(\rho, \sigma) \models p$ is *false* in which case $(\rho, \sigma) \models c$ is immediate. In the second case $(\rho, \sigma) \models p$ is *true* and from the Lemma 2 below we get that $(\rho', \sigma) \models p$ for all $\rho' \in M_j$ and hence the result follows by induction hypothesis.

LEMMA 2. *If $\rho = \bigsqcap M$, pre occurs in cl_j and $(\rho, \sigma) \models pre$ then also $(\rho', \sigma) \models pre$ for all $\rho' \in M_j$.*

Proof We proceed by induction on j and in each case perform a structural induction on the form of the precondition *pre* occurring in cl_j. Most cases are straightforward since $\rho(R) \subseteq \rho'(R)$ for all $\rho' \in M_j$. The only non-trivial case is when *pre* has the form $\neg R(x_1, \cdots, x_k)$. Here the result follows because $rank(R) < j$ and hence $\rho'(R) = \rho(R)$ for all $\rho' \in M_j$. □

Appendix B. Proof of Proposition 3

Appendix B.1 Proof of (1)

By structural induction on *pre* we shall show that

$$\overline{\mathcal{T}_\rho[\![pre]\!]\ \mathcal{E}} = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge (\rho, \sigma) \models pre\}$$

Case $R(\vec{x})$**:** We have to show

$$\overline{\{\eta' \mid \eta \in \mathcal{E}, \vec{a} \in \rho(R), \eta' = \mathsf{unify}\,(\eta, \vec{x}, \vec{a}) \neq \mathsf{fail}\}} = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge \sigma(\vec{x}) \in \rho(R)\}$$

($\subseteq$) Assume $\mathsf{unify}\,(\eta, \vec{x}, \vec{a}) \preceq \sigma$ for some $\eta \in \mathcal{E}$ and $\vec{a} \in \rho(R)$. Clearly $\eta \preceq \sigma$ and $\sigma(\vec{x}) = \vec{a}$ and the result follows.

($\supseteq$) Assume $\eta \preceq \sigma$ for $\eta \in \mathcal{E}$ and $\sigma(\vec{x}) \in \rho(R)$. Take $\vec{a} = \sigma(\vec{x})$ so $\vec{a} \in \rho(R)$; clearly $\mathsf{unify}\,(\eta, \vec{x}, \vec{a})$ will succeed and $\mathsf{unify}\,(\eta, \vec{x}, \vec{a}) \preceq \sigma$.

Case $\neg R(\vec{x})$**:** Similar.

Case $pre_1 \wedge pre_2$**:** It follows directly from the induction hypothesis that

$$\overline{\mathcal{T}_\rho[\![pre_2]\!]\ (\mathcal{T}_\rho[\![pre_1]\!]\ \mathcal{E})} = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge (\rho, \sigma) \models pre_1 \wedge (\rho, \sigma) \models pre_2\}$$

Case $pre_1 \vee pre_2$**:** It follows directly from the induction hypothesis that

$$\overline{\mathcal{T}_\rho[\![pre_1]\!]\ \mathcal{E}} \cup \overline{\mathcal{T}_\rho[\![pre_2]\!]\ \mathcal{E}} = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge ((\rho, \sigma) \models pre_1 \vee (\rho, \sigma) \models pre_2)\}$$

Case $\exists x.pre$**:** We shall show

$$\overline{\widehat{\mathbf{tl}}\,(\mathcal{T}_\rho[\![pre]\!]\,((x,\Box)\mathbin{\hat{::}}\mathcal{E}))} = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge \exists a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models pre\}$$

For the right-hand side we have:

$$\begin{aligned}&\{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge \exists a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models pre\}\\ &\quad = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge \exists a \in \mathcal{U} : \sigma[x \mapsto a] \in \overline{(x,\Box)\mathbin{\hat{::}}\mathcal{E}} \wedge (\rho, \sigma[x \mapsto a]) \models pre\}\\ &\quad = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge \exists a \in \mathcal{U} : \sigma[x \mapsto a] \in \overline{\mathcal{T}_\rho[\![pre]\!]\,((x,\Box)\mathbin{\hat{::}}\mathcal{E})}\}\end{aligned}$$

The first equality follows because $\eta \preceq \sigma$ implies $(x,\Box) :: \eta \preceq \sigma[x \mapsto a]$. The second equality follows from the induction hypothesis. For the left-hand side we have:

$$\begin{aligned}&\overline{\widehat{\mathbf{tl}}\,(\mathcal{T}_\rho[\![pre]\!]\,((x,\Box)\mathbin{\hat{::}}\mathcal{E}))}\\ &\quad = \{\sigma \mid \exists \eta, d : (x,d) :: \eta \in \mathcal{T}_\rho[\![pre]\!]\,((x,\Box)\mathbin{\hat{::}}\mathcal{E}) \wedge \eta \preceq \sigma \wedge \sigma \in \overline{\mathcal{E}}\}\\ &\quad = \{\sigma \mid \exists a : \sigma[x \mapsto a] \in \overline{\mathcal{T}_\rho[\![pre]\!]\,((x,\Box)\mathbin{\hat{::}}\mathcal{E})} \wedge \sigma \in \overline{\mathcal{E}}\}\end{aligned}$$

Here the first equality follows from Fact 1 and the second follows from the definitions.

Case $\forall x.pre$**:** We shall show

$$\overline{\widehat{\mathbf{tl}}\,(\mathcal{T}_{\rho,\mathcal{U}}[\![pre]\!]\,((x,\Box)\mathbin{\hat{::}}\mathcal{E}))} = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge \forall a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models pre\}$$

By induction on the size of the list U we shall prove

$$\overline{\widehat{\mathbf{tl}}\,(\mathcal{T}_{\rho,U}[\![pre]\!]\,((x,d)\mathbin{\hat{::}}\mathcal{E}))} = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge \forall a \in U : (\rho, \sigma[x \mapsto a]) \models pre\}$$

and the result will follow. In the case where U is empty it amounts to

$$\overline{\widehat{\mathbf{tl}}\,((x,d)\mathbin{\hat{::}}\mathcal{E})} = \overline{\mathcal{E}}$$

which clearly holds. For the induction step the left-hand side amounts to

$$\overline{\widehat{\mathbf{tl}}\,(\mathcal{T}_{\rho,U}[\![pre]\!]\,(\mathcal{T}_\rho[\![pre]\!]\,((x,a')\mathbin{\hat{::}}\mathcal{E})))}$$

which, by the current induction hypothesis (and Fact 1), equals:

$$\{\sigma \mid \sigma \in \overline{\widehat{\mathbf{tl}}(\mathcal{T}_\rho[\![pre]\!]\,((x,a')\mathbin{\hat{::}}\mathcal{E}))} \wedge \forall a \in U : (\rho, \sigma[x \mapsto a]) \models pre\}$$

Now the overall induction hypothesis applied to *pre* (together with Fact 1) gives

$$\overline{\widehat{\mathbf{tl}}\,(\mathcal{T}_\rho[\![pre]\!]\,((x,a')\mathbin{\hat{::}}\mathcal{E}))} = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge (\rho, \sigma[x \mapsto a']) \models pre\}$$

such that the left-hand side becomes

$$\{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge (\rho, \sigma[x \mapsto a']) \models pre \;\wedge\; \forall a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models pre\}$$

which amounts to the right-hand side of the induction step. This concludes the proof. □

Appendix B.2 Proof of (2)

By structural induction on *cl* we shall show that

$$\mathcal{T}^{\bullet}_{\rho}[\![cl]\!]\ \mathcal{E} \quad \textit{if and only if} \quad \forall\sigma \in \overline{\mathcal{E}} : (\rho, \sigma) \models cl$$

Case $R(\vec{x})$**:** We shall show

$$\forall\eta \in \mathcal{E} : \forall\vec{a} : \mathsf{unify}\,(\eta, \vec{x}, \vec{a}) \neq \mathsf{fail} \Rightarrow \vec{a} \in \rho(R) \quad \textit{iff} \quad \forall\sigma \in \overline{\mathcal{E}} : (\rho, \sigma) \models R(\vec{x})$$

($\Rightarrow$) Assume $\sigma \in \overline{\mathcal{E}}$ i.e. $\eta \preceq \sigma$ for some $\eta \in \mathcal{E}$. Take $\vec{a} = \sigma(\vec{x})$ and observe that $\mathsf{unify}\,(\eta, \vec{x}, \vec{a})$ cannot fail since $\eta \preceq \sigma$. Thus $\vec{a} \in \rho(R)$ as required.

($\Leftarrow$) Let $\eta \in \mathcal{E}$ and let $\vec{a}$ be such that $\mathsf{unify}\,(\eta, \vec{x}, \vec{a})$ does not fail. Then there exists σ such that $\mathsf{unify}\,(\eta, \vec{x}, \vec{a}) \preceq \sigma$ and $\sigma(\vec{x}) = \vec{a}$. Clearly $\eta \preceq \sigma$ so $\sigma \in \overline{\mathcal{E}}$ and hence $(\rho, \sigma) \models R(\vec{x})$. It now follows that $\vec{a} \in \rho(R)$.

Case 1: Trivial

Case $cl_1 \wedge cl_2$**:** It follows directly from the induction hypothesis that

$$\mathcal{T}^{\bullet}_{\rho}[\![cl_1]\!]\ \mathcal{E} \wedge \mathcal{T}^{\bullet}_{\rho}[\![cl_2]\!]\ \mathcal{E} \quad \textit{iff} \quad \forall\sigma \in \overline{\mathcal{E}} : (\rho, \sigma) \models cl_1 \wedge (\rho, \sigma) \models cl_2$$

Case $pre \Rightarrow cl$**:** We shall show

$$\mathcal{T}^{\bullet}_{\rho}[\![cl]\!]\ (\mathcal{T}_{\rho}[\![pre]\!]\ \mathcal{E}) \quad \textit{iff} \quad \forall\sigma \in \overline{\mathcal{E}} : (\rho, \sigma) \models pre \Rightarrow (\rho, \sigma) \models cl$$

The induction hypothesis gives

$$\mathcal{T}^{\bullet}_{\rho}[\![cl]\!]\ (\mathcal{T}_{\rho}[\![pre]\!]\ \mathcal{E}) \quad \textit{iff} \quad \forall\sigma \in \overline{\mathcal{T}_{\rho}[\![pre]\!]\ \mathcal{E}} : (\rho, \sigma) \models cl$$

From *(1)* we have

$$\overline{\mathcal{T}_{\rho}[\![pre]\!]\ \mathcal{E}} = \{\sigma \mid \sigma \in \overline{\mathcal{E}} \wedge (\rho, \sigma) \models pre\}$$

This concludes the proof.

Case $\forall x.cl$**:** We shall show

$$\mathcal{T}^{\bullet}_{\rho}[\![cl]\!]\ ((x, \Box) \mathbin{\hat{::}} \mathcal{E}) \quad \textit{iff} \quad \forall\sigma \in \overline{\mathcal{E}} : \forall a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models cl$$

The induction hypothesis gives

$$\mathcal{T}^{\bullet}_{\rho}[\![cl]\!]\ ((x, \Box) \mathbin{\hat{::}} \mathcal{E}) \quad \textit{iff} \quad \forall\sigma' \in \overline{(x, \Box) \mathbin{\hat{::}} \mathcal{E}} : (\rho, \sigma') \models cl$$

($\Rightarrow$) Given $\sigma \in \overline{\mathcal{E}}$ and $a \in \mathcal{U}$ we have $\eta \preceq \sigma$ for some $\eta \in \mathcal{E}$ so $(x, \Box) :: \eta \preceq \sigma[x \mapsto a]$ and $(x, \Box) :: \eta \in (x, \Box) \mathbin{\hat{::}} \mathcal{E}$. Thus $\sigma[x \mapsto a] \in \overline{(x, \Box) \mathbin{\hat{::}} \mathcal{E}}$ and hence $(\rho, \sigma[x \mapsto a]) \models cl$.

($\Leftarrow$) Let $\sigma' \in \overline{(x, \Box) \mathbin{\hat{::}} \mathcal{E}}$ and observe that $(x, \Box) :: \eta \preceq \sigma'$ (for some $\eta \in \mathcal{E}$) so that $\eta \preceq \sigma$ for some σ and $a \in \mathcal{U}$ satisfying $\sigma' = \sigma[x \mapsto a]$. Since $\sigma \in \overline{\mathcal{E}}$ it follows that $(\rho, \sigma[x \mapsto a]) \models cl$. □

Appendix C. Proof of Theorem 1

In the sequel we make use of the following companion to Proposition 3:

PROPOSITION 6. *For all solutions ρ, pre-conditions pre, and sets $\mathcal{E}$ of partial environments, the following holds:*

$$\#\,(\mathcal{T}_\rho[\![\text{pre}]\!]\,\mathcal{E}) \leq \mathcal{C}_\rho[\![\text{pre}]\!]\,\mathcal{E}$$

When started on a solution ρ placed in `rho`, the checker of Table IV can be thought of as contiguously evaluating the sub-terms of the clause — thus allowing us to perform an induction on the structure of clauses. Let *pre* and *cl* denote a pre-condition and a clause, respectively. Let $T[pre, \mathcal{E}]$ denote the maximal time spent by the algorithm on the call $\texttt{check}\,(pre, \texttt{K})\,\eta$ for any partial environment η from $\mathcal{E}$ before calling continuation `K`. Accordingly, let $T[cl, \mathcal{E}]$ denote the total time spent by the algorithm on calls $\texttt{execute}\,cl\,\eta$ for $\eta \in \mathcal{E}$. We estimate these complexities as follows:

PROPOSITION 7. *There are constants $d_{pre}, d_{cl} > 0$ only depending on pre and cl, respectively, such that:*

$$\begin{array}{lcl} T[pre, \mathcal{E}] & \leq & d_{pre} \cdot (\#\mathcal{E} + \mathcal{C}_\rho[\![pre]\!]\,\mathcal{E}) \\ T[cl, \mathcal{E}] & \leq & d_{cl} \cdot (\#\mathcal{E} + \mathcal{C}^\bullet_\rho[\![cl]\!]\,\mathcal{E}) \end{array}$$

Before giving the proof we return to Theorem 1: Since $\#\mathcal{E}_0 = 1$, we conclude from Proposition 7, that the checker uses time $\mathcal{O}(\#\rho + \mathcal{C}^\bullet_\rho[\![cl]\!]\,\mathcal{E}_0)$ as stated in the theorem — where the extra time $\mathcal{O}(\#\rho)$ is needed to read the relations $\rho(R)$ and insert them into the data structure `rho`.

Appendix C.1 Proof of Proposition 7

It remains to prove Proposition 7. Here, we only consider the cases where *pre* is a query or a conjunction, and where *cl* is an assertion or an implication.

Case *pre* is $R(\vec{x})$: For $\eta \in \mathcal{E}$ and $\vec{a}_1 = \texttt{first}\,(\eta, \vec{x})$, let T_η denote the set of all tuples $\vec{a}_1 @ \vec{a}_2 \in \rho(R)$. The work of calls $\texttt{check}\,(R(\vec{x}), \texttt{K})\,\eta$ where $\eta \in \mathcal{E}$ consists of:

- ∘ determining for each $\eta \in \mathcal{E}$, the set T_η and computing $\texttt{unify}\,(\eta, \vec{x}, \vec{a}')$ for all $\vec{a}' \in T_\eta$.

According to our tree-like representation of relations, the checker will need time $\mathcal{O}(1 + \#T_\eta)$ for each η. Therefore, we obtain:

$$T[R(\vec{x}), \mathcal{E}] \leq d \cdot \textstyle\sum_{\eta \in \mathcal{E}} (1 + \#T_\eta) = d \cdot (\#\mathcal{E} + \mathcal{C}_\rho[\![R(\vec{x})]\!]\,\mathcal{E})$$

for some constant $d > 0$ — giving the claim of Proposition 7 for this case.

Case *pre* is $pre_1 \wedge pre_2$: In this case the work of calls $\texttt{check}\,(pre, \texttt{K})\,\eta$ where $\eta \in \mathcal{E}$ consists of

- the work of all the calls $\texttt{check}\,(pre_1, \texttt{K}')\,\eta$ for $\eta \in \mathcal{E}$ where $\texttt{K}' = \texttt{check}\,(pre_2, \texttt{K})$ — thereby computing the set of partial environments $\mathcal{E}' = \mathcal{T}_\rho[\![pre_1]\!]\,\mathcal{E}$;
- the work of all the calls $\texttt{check}\,(pre_2, \texttt{K})\,\eta'$ for $\eta' \in \mathcal{E}'$.

By the inductive hypothesis, the first task consumes time $\mathcal{O}(\#\mathcal{E}+\mathcal{C}_\rho[\![pre_1]\!]\,\mathcal{E})$ whereas the second one takes time $\mathcal{O}(\#\mathcal{E}' + \mathcal{C}_\rho[\![pre_2]\!]\,\mathcal{E}')$. Now, by Proposition 6, we get:

$$\#\mathcal{E}' = \#\mathcal{T}_\rho[\![pre_1]\!]\,\mathcal{E} \leq \mathcal{C}_\rho[\![pre_1]\!]\,\mathcal{E}$$

Therefore

$$T[pre, \mathcal{E}] \leq d \cdot (\#\mathcal{E} + \mathcal{C}_\rho[\![pre_1]\!]\,\mathcal{E} + \mathcal{C}_\rho[\![pre_2]\!]\,(\mathcal{T}_\rho[\![pre_1]\!]\,\mathcal{E}))$$

for some constant $d > 0$ — thereby establishing our claim.

Case *cl* is $R(\vec{x})$: The work of the checker on the assertion $R(\vec{x})$ when executing the calls $\mathsf{execute}\,(R(\vec{x}))\,\eta$ for $\eta \in \mathcal{E}$ amounts to:

- determining for every $\eta \in \mathcal{E}$, the set T_η of all $\vec{a}$ with $\mathsf{unify}\,(\eta, \vec{x}, \vec{a}) \neq \mathsf{fail}$;
- for each $\vec{a} \in T_\eta$ checking that $\texttt{rho.has}(R, \vec{a})$.

According to our assumptions, this work requires time $\mathcal{O}(\sum_{\eta\in\mathcal{E}}(1 + \#T_\eta))$. We have:

$$\begin{array}{rcl} \sum_{\eta\in\mathcal{E}}(1 + \#T_\eta) & \leq & \#\mathcal{E} + \#\{(\eta, \vec{a}) \mid \eta \in \mathcal{E}, \vec{a} \in \rho R, \texttt{unify}\,(\eta, \vec{x}, \vec{a}) \neq \mathsf{fail}\} \\ & = & \#\mathcal{E} + \mathcal{C}^\bullet_\rho[\![R(\vec{x})]\!]\,\mathcal{E} \end{array}$$

— giving the assertion of Proposition 7 for this case.

Case *cl* is $pre \Rightarrow cl_0$: The work of the checker on calls $\texttt{execute}\,(pre \Rightarrow cl_0)\,\eta$ for $\eta \in \mathcal{E}$ amounts to:

- the work on calls $\texttt{check}\,(pre, \texttt{execute}\,cl_0)\,\eta$ for $\eta \in \mathcal{E}$; together with
- the work on calls $\texttt{execute}\,cl_0\;\eta'$ for $\eta' \in \mathcal{T}_\rho[\![pre]\!]\,\mathcal{E}$, i.e., those η' which are obtained by the calls $\texttt{check}\,(pre, \texttt{execute}\,cl_0)\,\eta$, $\eta \in \mathcal{E}$.

Therefore by inductive hypothesis and Proposition 6

$$\begin{array}{rcl} T[pre \Rightarrow cl_0, \mathcal{E}] & \leq & d_0 + T[pre, \mathcal{E}] + T[cl_0, \mathcal{T}_\rho[\![pre]\!]\,\mathcal{E}] \\ & \leq & d \cdot (\#\mathcal{E} + \mathcal{C}_\rho[\![pre]\!]\,\mathcal{E} + \#\,(\mathcal{T}_\rho[\![pre]\!]\,\mathcal{E}) + \mathcal{C}^\bullet_\rho[\![cl_0]\!]\,(\mathcal{T}_\rho[\![pre]\!]\,\mathcal{E})) \\ & \leq & d \cdot (\#\mathcal{E} + 2 \cdot \mathcal{C}_\rho[\![pre]\!]\,\mathcal{E} + \mathcal{C}^\bullet_\rho[\![cl_0]\!]\,(\mathcal{T}_\rho[\![pre]\!]\,\mathcal{E})) \\ & \leq & 2 \cdot d \cdot (\#\mathcal{E} + \mathcal{C}^\bullet_\rho[\![pre \Rightarrow cl_0]\!]\,\mathcal{E}) \end{array}$$

for suitable constants $d_0, d > 0$. This completes the proof. □