Succinct Solvers

Flemming Nielson[1] and Helmut Seidl[2]

[1] Informatics & Mathematical Modelling, Technical University of Denmark, DK-2800 Kongens Lyngby, Denmark, nielson@imm.dtu.dk
[2] Universität Trier, FB IV – Informatik, D-54286 Trier, Germany, seidl@uni-trier.de

Abstract. We develop a solver algorithm which allows to efficiently compute the optimal 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 evaluations on a control-flow analysis of the ambient calculus shows a good match between theory and practice.

Keywords: Program analysis, Least Fixpoint Logic, topdown fixpoint computation, ambient calculus.

1 Introduction

Functional languages are seldomly 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 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 cost 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 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. (Behavioural psychologists typically find that a human is only able to grasp the interplay of about seven concepts at any point in time.) Indeed, "a notation is important for what it leaves out" [15, page 33] because then more succinct specifications can be developed that only 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* [7, 10] but still allows for polynomial model-checking routines. Formulas in this fragment naturally arise in the specification of static analyses of programs (see, e.g. [11, 12]). Here, we consider the systematic design of a fixpoint engine for this logic. We build on known approaches for sub-logics like [11, 12] for Datalog. Like McAllester in [11], we aim at an algorithm which makes computing the result as cheap as checking of the result. Furthermore, the algorithm should be *simple*, i.e., work almost without pre-processing of formulas, and *predictable*, i.e., its complexity behaviour should be easily computable.

More specifically 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 Datalog as, e.g., advocated by McAllester — yet our solver can be presented in less than a page of SML 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 performance.
- 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 formulas.

The development builds on previous insights on using functional programming for implementing state-of-the-art solvers [6, 9, 8] achieving only a few of the above advantages.

In Section 2 we present the fragment of predicate logic considered and define its semantics. In Section 3 we then develop the solver, explain the algorithmic techniques needed for obtaining good performance, and illustrate the modularity of its design. In Section 5 we give an overall characterisation of its behaviour and use this to obtain a formal complexity result that is validated practically in Section 6. Section 7 concludes and the appendices contain proofs of the main results.

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 [12] and extend them further by allowing also universal quantification in pre-conditions and negation. Thus, ALFP formulas extend Horn clauses (with explicit quantification) in that we additionally allow

- both existential and universal quantification in pre-conditions;
- negated queries;
- disjunctions of preconditions as well as
- conjunctions of conclusions.

The extra features increase convenience in handling in that they allow to restrict the scopes of variables and merge identical conclusions as well as identical pre-conditions without the technical inconvenience of introducing auxiliary predicates. Existential quantification in pre-conditions does not enhance succinctness or expressiveness of Horn formulas. By using universal quantification

in pre-conditions, and through negation, we may express queries which are otherwise impossible to state. Consider, e.g., the formula:

$$\forall x.(\forall y.\neg E(x,y) \vee T(y)) \Rightarrow T(x)$$

This formula defines a predicate T which is satisfied by the set of all acyclic nodes in a graph, i.e., all nodes from which no cycle can be reached. This predicate T is not definable in Datalog – even with stratified negation [7, 10].

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

$$\begin{array}{lcl}
\mathsf{pre} & ::= & R(x_1,\ldots,x_k) \mid \neg R(x_1,\ldots,x_k) \mid x = y \mid x \neq y \\
 & & \mathsf{pre}_1 \wedge \mathsf{pre}_2 \mid \mathsf{pre}_1 \vee \mathsf{pre}_2 \mid \forall x : \mathsf{pre} \mid \exists x : \mathsf{pre} \\
\mathsf{clause} & ::= & R(x_1,\ldots,x_k) \mid \mathbf{1} \mid \mathsf{clause}_1 \wedge \mathsf{clause}_2 \mid \\
 & & \mathsf{pre} \Rightarrow \mathsf{clause} \mid \forall x : \mathsf{clause}
\end{array}$$

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

Given a universe $\mathcal{U}$ of atomic values (or atoms) together with interpretations ρ and σ for predicate symbols and free variables, respectively, we define the satisfaction relation

$$(\rho, \sigma) \models t$$

(t a pre-condition or clause) as follows.

$(\rho, \sigma) \models \mathbf{1}$ holds for all ρ and σ. Otherwise, we have:

$$\begin{array}{lll}
(\rho,\sigma) \models R(x_1,\ldots,x_k) & \text{iff} & (\sigma\, x_1,\ldots,\sigma\, x_k) \in \rho\, R \\
(\rho,\sigma) \models \neg R(x_1,\ldots,x_k) & \text{iff} & (\sigma\, x_1,\ldots,\sigma\, x_k) \notin \rho\, R \\
(\rho,\sigma) \models x = y & \text{iff} & \sigma\, x = \sigma\, y \\
(\rho,\sigma) \models x \neq y & \text{iff} & \sigma\, x \neq \sigma\, y \\
(\rho,\sigma) \models \exists x : \mathsf{pre} & \text{iff} & (\rho, \sigma \oplus \{x \mapsto a\}) \models \mathsf{pre} \quad \text{for some } a \in \mathcal{U} \\
(\rho,\sigma) \models \forall x : t & \text{iff} & (\rho, \sigma \oplus \{x \mapsto a\}) \models t \quad \text{for all } a \in \mathcal{U} \\
(\rho,\sigma) \models t_1 \wedge t_2 & \text{iff} & (\rho,\sigma) \models t_1 \text{ and } (\rho,\sigma) \models t_2 \\
(\rho,\sigma) \models \mathsf{pre}_1 \vee \mathsf{pre}_2 & \text{iff} & (\rho,\sigma) \models \mathsf{pre}_1 \text{ or } (\rho,\sigma) \models \mathsf{pre}_2 \\
(\rho,\sigma) \models \mathsf{pre} \Rightarrow \mathsf{clause} & \text{iff} & (\rho,\sigma) \models \mathsf{clause} \text{ whenever } (\rho,\sigma) \models \mathsf{pre}
\end{array}$$

In the sequel, we view the free variables occurring in an formula as constant symbols or atoms from the finite universe $\mathcal{U}$. Thus, given an interpretation σ of the constant symbols, in the clause clause, we call an interpretation ρ of the predicate symbols $\mathcal{R}$ a *solution* provided $(\rho, \sigma) \models \mathsf{clause}$.

Since our logic provides explicit checks for equality and in-equality in pre-conditions, we abandon corresponding checks in queries, i.e., demand that all variables (which are not constants) occurring in an argument list are pairwise distinct. In order to deal with negations conveniently, we restrict ourselves to *alternation-free* formulas. We introduce a notion of stratification similar to the one which is known from Datalog [5, 2]. A clause c is an *alternation-free Least Fixpoint formula* (ALFP formula for short) iff c is of the form $c = s_1 \wedge \ldots \wedge s_k$, and there is a rank function $r : \mathcal{R} \to \mathbb{N}$ such that for all $j = 1, \ldots, k$, the following properties hold:

1. All predicates of assertions in s_j have rank j;
2. All predicates of queries in s_j have ranks at most j;
3. All predicates of negated queries in s_j have ranks strictly less than j.

This allows us to define an equality predicate E and a non-equality predicate N by the clause

$$(\forall x : E(x,x)) \wedge (\forall x : \forall y : \neg E(x,y) \Rightarrow N(x,y))$$

and hence we may dispense with an explicit treatment of $=$ and $\neq$ in the development that follows. On the other hand this rules out:

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

Let Δ denote the set of interpretations of predicate symbols in $\mathcal{R}$ over $\mathcal{U}$. Then Δ is a complete lattice w.r.t. the lexicographical ordering "$\sqsubseteq$", i.e., $\rho_1 \sqsubseteq \rho_2$ iff there is some $1 \leq j \leq k$ such that the following properties hold:

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

We have:

Proposition 1. *Assume* clause *is an ALFP formula and σ is an interpretation of the free variables in* clause. *Then the set of all ρ with $(\rho, \sigma) \models$ clause forms a* Moore *family, i.e., is closed under greatest lower bounds.*

A proof of proposition 1 can be found in appendix A. Since the set of all solutions of clause (given a fixed σ) is a Moore family, we conclude that for every interpretation ρ_0 of the predicate symbols, there is a lexicographically least solution ρ of clause such that $\rho_0 \subseteq \rho$ (i.e., $\rho_0\, R \subseteq \rho\, R$ for all R) — which we call the *optimal* solution of clause exceeding ρ_0.

3 The ALFP Solver

A first and rather naive method of solving an ALFP formula c consists in instantiating all variables occurring in c in all possible ways. The resulting system can then be solved by classical solvers for alternation-free Boolean equation systems. Extending the proof of the corresponding proposition in [12] (itself extending [11, Theorem 1]), we have:

Proposition 2. *The optimal solution ρ of a ALFP formula c exceeding an input interpretation can be computed in time $\mathcal{O}(\#\rho + N^r \cdot n)$ where N is the number of occurring atoms, n is the size of c, and r is the maximal nesting depth of quantifiers in c and $\#\rho$ is the sum of cardinalities of relations $\rho\, R$.* □

As observed by McAllester in [11] for conventional Horn clauses, such an approach and such a complexity bound is unsatisfactory for program analysis, since the universe $\mathcal{U}$ may have cardinality as large as the program itself – implying that even analyses using as few as two variables would not be likely to scale up to large programs. Therefore, McAllester presents an algorithm for (a special class of) Horn clauses which allows a computation of the least solution of the original clause which often is faster than the naive approach from proposition 2. The key idea of McAllester is to bring clauses into a specific canonical form. In order to do so, he introduces auxiliary predicates for prefixes of pre-conditions. Furthermore, he employs constructor applications for collecting bound variables into environments.

McAllester's algorithm is rather indirect as it amounts to a major amount of pre-processing before actual solving. Moreover, the new auxiliary predicates may be a source of unnecessary extra space consumption. The main objective, however, is that McAllester's solving method

does no longer suffice for the richer logic ALFP. Therefore, we developed and implemented an alternative solving algorithm. In contrast to McAllester's method, our solving procedure does not rely on pre-processing. It also completely abandons special worklist-like data-structures as are typical for most classical iterative fixpoint algorithms [9]. Instead, we adapt the recursive topdown approach of Le Charlier and van Hentenryck [6] which is enhanced by continuation-based semi-naive iteration [3, 8].

3.1 The Basic Solver

In essence, our solver for ALFP formulas consists of two recursive functions execute and check that operate on clauses and preconditions, respectively. The interpretation of predicate symbols, corresponding to ρ, is maintained in imperative datastructures that are updated by means of side effects. The interpretation of variables, corresponding to σ, is maintained in an environment that is passed around explicitly (in a continuation based manner in the case of check). Thus upon termination of the solver, the interpretation of predicate symbols in the global datastructures will have been assigned values that makes the clauses satisfiable (for a suitable interpretation of variables).

```
type U     = (* the universe *)
type tuple = U list
type env   = (var * U option) list

unify      : env * var list * tuple → env option
unifiable  : env * var list → tuple list
```

Fig. 1. Operations on environments.

Before explaining the operation of execute and check we need to take a closer look at the types and auxiliary functions used in the solver (see figure 1). The universe of atoms is an unspecified type U and we model a tuple of a relation as a list of atoms (in order to deal with relations of different arities) and a relation then conceptually is a global data structure containing a list of tuples (although represented as a prefix tree as explained below) — this takes care of the interpretation of predicate symbols, ρ. Environments map variables to atoms but is constructed in a lazy fashion meaning that variables may not have been given their values when introduced by quantifiers; hence environments are partial environments that are list of pairs $x \mapsto a$ where x is a variable and a is an optional value, i.e., either an atom or $\square$ (denoting "unbound" or NONE in SML) — this takes care of the interpretation of variables, σ (that is called η in the sequel to indicate that it is now a partial environment).

The main auxiliary operation on environments is the function unify which when called for a partial environment η, a list $[x_1, \ldots, x_k]$ of variables x_i and a tuple $[a_1, ..., a_k]$ of atoms (representing a tuple of some k-ary relation), determines the minimal extension η' of η with $\eta'\, x_i = a_i$ for all i if it exists; otherwise, it returns $\square$ ("fail"). Thus, η' is the most general unifier of the two lists relative to η. Moreover, we need a function unifiable which, for a given pair $(\eta, args)$ of a partial environment η and a list of variables $args$ determines the list of all tuples t for which unify $(\eta, args, t)$ succeeds.

Both functions execute and check operate by recursively processing the original clause or pre-condition and by propagating an environment which collects bindings of the instantiated variables. Whenever during propagation of a partial environment η, a quantified subexpression is encountered which introduces variable x, we extend η with $x \mapsto \square$. Accordingly, whenever the

quantified subexpression is left, the variable x is popped from the environment. New bindings for variables are obtained at a query $R(args)$. There we compute unify $(\eta, args, t)$ for every list t so far known to belong to the interpretation of R. The resulting most general unifiers are then propagated further through the clause.

The functions execute and check are presented in SML-style notation in figures 3 and 4 and are explained in detail below. Function check operates on pre-conditions in order to find new bindings for the (partial) variable environment, whereas function execute operates on clauses in order to generate new elements for predicates. Pre-conditions may trigger further executions. These future computations are passed to the function check as continuations. The type of continuations is defined in figure 2 as are a number of other auxiliary definitions. We also recall the SML-function app that applies a function to each element in a list, in order to perform the side effects.

The global data structure rho is used for recording the currently known elements of predicates. Accordingly, it allows adding new tuples to predicates, checking for the presence of tuples, as well as retrieving whole relations. The use of data structure infl is slightly more subtle.

```
type cont = env → unit
type consumer = tuple → unit

global datastructure rho
add       : rho * predicate * tuple → unit
has       : rho * predicate * tuple → bool
sub       : rho * predicate → tuple list

global datastructure infl
register  : infl * predicate * consumer → unit
consumers : infl * predicate → consumer list
```

Fig. 2. Operations on predicates.

It may happen that some query $R(args)$ inside a pre-condition fails to to be satisfiable in the current environment, but may hold in the future when some new elements for R have been revealed. In this case, we residualise the current computation by constructing a *consumer* for R and recording it in the data structure infl for predicate R. This *consumer* is nothing but the subsequent call to function unify followed by the corresponding continuation of the check routine. Accordingly, whenever the function execute adds a new elements t to the interpretation of R, the list of consumers waiting for new elements of R is activated, and the corresponding computations are resumed.

```
check : pre * cont → cont

fun check (p, n) η = case p
    of R(args) → let  fun f t = case unify (η, args, t)
                                of □   →  ()
                                 | η'  →  n η'
                  in   register (infl, R, f); app f (sub (rho, R))
                  end
     | p1 ∧ p2  → check (p1, check (p2, n)) η
     | ...
```

Fig. 3. The key elements of function check.

Our algorithmic idea here abandons an explicit worklist-like data-structure for the fixpoint iteration. Instead, it employs the recursion stack itself for this purpose. Re-activating residualized continuations corresponds to an *eager* iteration strategy where all consequences of the first of several new facts are tracked before the next one is considered.

The function check is applied to a pair of a pre-condition p and a continuation n together with a partial environment η for the global variables. As equality "=" (and un-equality "$\neq$" as soon as we have negation) can be defined, we have omitted corresponding rules in the treatment of pre-conditions. If p is a query *R(args)* then a continuation f is constructed which given an element t for R, determines the most general unifier of *args* with t relative to η and, if it exists propagates it to n. The continuation f is registered as a potential consumer of new elements of R. Then it is called for all elements which already have been revealed for R. If p is a conjunction $p_1 \wedge p_2$, then the check of the second pre-condition check (p_2, n) is supplied as a continuation for the call of check on the first pre-condition p_1.

The function execute is applied to a clause c and a partial environment η for global variables. The intention of execute is to add new elements to predicates. If c equals **1**, nothing has to be done. If c is an assertion *R(args)* then all elements t are considered which are unifiable with *args* in environment η. If such a element t is *new*, we add it to the relation for R. Moreover, we determine the set of consumers which have been registered for R and apply each of them to t. If c is a conjunction of clauses c_1 and c_2, then execute is called on c_1 and c_2 in sequence – both with the same environment η since no new bindings can result. If c is an implication $p \Rightarrow c'$, then the function check is called on the pre-condition p with continuation execute c'. If c is a universal quantification $\forall x : c'$, the new binding $x \mapsto \square$ is added to η followed by an execution of c'.

```
execute : clause → cont

fun execute c η = case c
 of 1          → ()
  | R(args)    → app (fn t → if   has(rho, R, t) then ()
                             else (add (rho, R, t);
                                   app (fn f → f t) (consumers (infl, R))))
                     (unifiable (η, args))
  | c1 ∧ c2    → (execute c1 η; execute c2 η)
  | p ⇒ c'     → check (p, execute c') η
  | ∀x : c'    → execute c' ((x ↦ □) :: η)
```

Fig. 4. The function execute.

3.2 Prefix Trees

Two problems withstand a rapid propagation of new elements of a predicate R to all places where these elements are queried.

The first problem is, given an argument list *args* together with a partial environment η, to *find* all elements of a predicate which possibly are unifiable with *args* relative to η: this is to make the app loop at queries fast (see fig. 3).

The second problem is, to maintain the consumers in such a way that consumers are easy to register, and all registered consumers which are potentially applicable to new elements can be revealed efficiently: this is to make the inner app loop at assertions $R(args)$ fast (see fig. 4).

Different consumers for the same element $t = [a_1, \ldots, a_k]$ of a k-ary predicate may correspond to queries where different subsets of argument positions are bound in the environment. An ob-

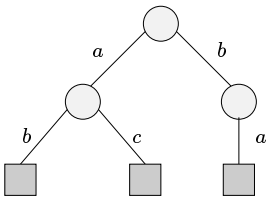

Fig. 5. Representation of the relation $\{(a,b),(a,c),(b,a)\}$.

vious 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 t. Organizing and maintaining access to these is awkward and incurs overhead.

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 elements of the k-ary predicate R into a *prefix tree* as in fig. 5. The tree representation allows us to implement a function subS which, given the data structure rho, a predicate R and a prefix t_1, rapidly enumerates all suffices t_2 with $t_1@t_2$ contained in R. Given this function, we implement the app-loop in figure 3 as in figure 6. Here, the function first in the first line takes a pair of an environment η and an argument list *args* and returns the unique sequence of atoms corresponding to the maximal prefix of *args* whose variables are all instantiated in η.

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.

Accordingly, we also allow consumers of a k-ary predicate R only to register for the set of *suffices* t_2 for a given prefix t_1. This implies that there are at most $k+1$ sets of consumers waiting for each new element t of R. Again, the waiting consumers are maintained in a prefix tree of depth at most k whose nodes correspond to the possible prefixes t_1. In particular, all potential consumers of a single element t 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 suffices. This functionality is provided by the two functions registerS and consumersS whose types are given in figure 7.

So assuming that hash table lookups can be done in time $\mathcal{O}(1)$, we conclude that collecting all consumers for an element t of predicate R amounts to $k+1$ calls to the function consumersS and a concatenation of $k+1$ lists. 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 result.

It remains to explain that our restriction in generality is not as severe as it might seem. If support for a further query pattern of the k-ary predicate R is demanded, it could be obtained through a source-to-source transformation – similar in spirit to the second last rule of McAllester

```
let  val t1 = first (η, args)
in   app (fn t2 → f (t1 @ t2)) (subS (rho, R, t1))
end
```

Fig. 6. The improved app-loop of check.

```
subS        : rho * predicate * tuple → tuple list
registerS   : infl * predicate * consumer * tuple → unit
consumersS  : infl * predicate * tuple → consumer list
```

Fig. 7. Improved operations on predicates.

```
fun check (p, n) η = case p
    of ...
     | ¬R(args) → app (fn t → if has(rho, R, t) then ()
                              else n (unify (η, args, t)))
                      (unifiable (η, args))
     | p1 ∨ p2   → (check (p1, f) η; check (p2, f) η)
     | ∃x : p'   → check (p', n ∘ tl) ((x ↦ □) :: η)
     | ∀x : p'   → let fun f []       ((x ↦ _) :: η) = n η
                         | f (a::U)   ((x ↦ _) :: η) = check (p', f U) ((x ↦ a) :: η)
                   in f U ((x ↦ □) :: η)
                   end
```

Fig. 8. The advanced elements of function check.

in the proof of [11, Theorem 3], however, without introduction of non-atomic terms. Here, it suffices to introduce an auxiliary predicate R' which is obtained from R through re-arrangement of argument positions. For example, if we would like to query the ternary predicate R where the last argument is bound, we introduce the new clause:

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

Furthermore, we replace the corresponding query of R in a pre-condition by the corresponding query of R'.

Against the exponential overhead of blindly introducing support for all exponentially many query patterns, this source-to-source transformation has the advantage that the overhead for the additional query pattern is paid only if it really occurs in the current clause.

3.3 Extensions

Let us now extend the basic solving algorithm for handling the advanced features negation, disjunction, and existential and universal quantification in queries that we dispensed with in figure 3. It turns out that it suffices to add further rules to the function check. The necessary additions are summarised in figure 8.

Technically, the simplest extension is negation. According to the last-in-first-out discipline of the runtime stack, the solver only proceeds to the execution of the second conjunct of two clauses, when the execution of the first one together with all triggered calls to continuations have been 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 former ones have definitely stabilised. This means that our solving strategy naturally respects the stratification and strong components. Thus, for dealing with negated queries $\neg R(args)$, we only have to add a new rule to the function check. This rule determines all tuples t *not* belonging to R and then propagates unify $(\eta, args, t)$ to the current continuation. Since the relation for R has already stabilised (and therefore also its complement), no registration of consumers is necessary.

The basic idea for disjunctions $p \equiv p_1 \vee p_2$ is to call check for both pre-conditions p_i — but now with the same continuation.

If p is an existential quantification $\exists x : p'$, we check the condition p' for the extended environment $(x \mapsto \square)::\eta$ where the resulting environments are truncated before propagation to the continuation n.

If p is a universal quantification $\forall x : p'$, we add the binding $x \mapsto \square$ to η and then propagate it to the continuation $f\,\mathcal{U}$. Here, $\mathcal{U}$ denotes a list of all atoms in the universe. The function f evaluates to the continuation n when applied to an empty list of atoms. When applied to a non-empty list $a::U$ and some environment f updates the topmost binding in the environment to a and then calls the function check on the body p' and $f\,U$ as the remaining continuation.

4 Practical Issues

Here, we mention just three further issues which are important in order to arrive at an efficient implementation.

4.1 Avoiding Duplication of Work

It may happen that, while evaluating a call check $(p, n)\,\eta$, the continuation n is called more than once for an environment η'. This may happen at disjunctions $p \equiv p_1 \vee p_2$ where η' satisfies both p_1 and p_2. Another source of such duplication of work are existential quantifications $\exists x : p'$ where p' can be satisfied for different values of x.

In order to avoid this unnecessary duplication of work, we add memoisation for every occurrence of a pre-condition p which records the set of environments η' for which the given continuation has already been called. This principal 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. So, we place memoisation only at an outermost occurrences of disjunctions or existential quantifications. In particular, we omit memoisation at conjunctions. The reason is twofold:

1. In absence of disjunctions or existential quantifications, no duplication of work can occur.
2. If memoisation occurs at all disjunctions and existential quantifications, then every continuation n 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 caching at existentials and disjunctions. The two environments generated through checking of the disjunction are:

$$\eta_1 = [x \to a, y \to \square] \quad \text{and} \quad \eta_2 = [x \to \square, y \to a]$$

These two are different and propagated to the query $S(x, y)$. Unification with the tuple $[a, a]$ there will result in the environment:

$$\eta = [x \to a, y \to a]$$

being generated *twice*. 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 in the sets of so far unbound variables. As the formula has constant size, these are just constantly many.

4.2 Optimising Auxiliary Predicates

Clearly, we could tabulate for *each* occurrence of a query $R(x_1, \ldots, x_k)$ a separate (permuted) copy of the relation for R where the subset of instantiated variables does not form a prefix of the sequence $x_1, \ldots, x_k$. 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. Consider, e.g., a ternary predicate R which is queried once with the first and last component instantiated and another time just with the last component instantiated. Then it suffices to introduce just one auxiliary predicate R' which is defined by:

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

Thus, the predicate R' serves both requirements, and we have saved time and space for one extra copy of R. In general, we arrive here at the following optimisation problem:

Given: a set $\mathcal{S}$ of subsets $S \subseteq \{1, \ldots, k\}$ of instantiated components;
Wanted: a set Π of permutations π of the sequence: $1, \ldots, k$ such that each subset $S \in \mathcal{S}$ is a prefix of some $\pi \in \Pi$.

We conjecture that computing such a set Π of *minimal cardinality* in general is hard. For our purposes here, however, *any* set Π of cardinality at most $\#\mathcal{S}$ will do: smaller sets just decrease resource consumption. Therefore, we use a simple greedy algorithm. 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.

This algorithm still gives the optimal result for $k = 3$ — and was good enough for all our applications so far.

4.3 Optimising Tail Recursion

At queries $p \equiv R(\ldots)$, 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 the to-date 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* tail-recursive in the definition of app — 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 tail-recursive in app as well — implying that the stack frame for app (which was taken from the call to function check) is now re-used by the last call to 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.

5 Estimating the Complexity

For the following, we assume that the algorithm is equipped with a caching scheme such that no continuation n is called twice with the same argument.

We now state the fundamental observation that computing the solution to a clause is as expensive, asymptotically, as checking the result:

Proposition 3. *Assume that c is an ALFP formula of constant size. Let ρ_0 denote an input interpretation and ρ the optimal solution of c exceeding ρ_0. Let t_0 and t be the runtimes of the solver when started on ρ_0 and ρ, respectively.*

Then $t_0 \leq d \cdot t$ for some constant d 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. Then the following holds:

1. The fixpoint computation π_0 performs the same calls $\mathsf{check}\ (p, n)\ \eta$, $\mathsf{execute}\ c\ \eta$, $\mathsf{register}\ (\eta, \mathit{args}, t)$, and $\mathsf{unify}\ (\eta, \mathit{args}, t)$ as π — only, perhaps, in a different ordering.
2. 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 $\mathsf{consumersS}$ as described in subsection 3.2 and our assumption that the size of c, and therefore also all arities of occurring predicates, are $\mathcal{O}(1)$, we know that the runtime of each individual call $\mathsf{consumersS}\ (\mathsf{infl}, R, t)$ 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 $\mathsf{consumersS}\ (\mathsf{infl}, R, t)$, and Q equals the total number of returned consumers. The value P is bounded by the total number of calls $\mathsf{execute}\ c'\ \eta$ where c' 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 prop. 3 has been stated by McAllester for his algorithm and ordinary Horn clauses [11]. 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 "denotational" way.

Assume for the following that we are given a fixed interpretation ρ of predicate symbols. Every pre-condition p gives rise to a transformer $\mathcal{T}_\rho[p]$ mapping a set $\mathcal{E}$ of reaching environments for the variables in B to the set of environments after checking p, i.e., those which are propagated by calls $\mathsf{check}\ (p, n)\ \eta$, $\eta \in \mathcal{E}$ to the continuation n. All environments η in $\mathcal{E}$ are of the form

$$\eta = [x_1 \mapsto d_1, \ldots, x_m \mapsto d_m]$$

for the same sequence $x_1, \ldots, x_m$ of variables but potentially different sequences $d_1, \ldots, d_m$ of values (including $\square$). Furthermore, all free variables from p are among the x_j. A formal definition of this transformation is given in figure 9. Here, $\bar{\mathsf{tl}}$ and $\lambda E.a :: E$ denote the elementwise extension of the corresponding un-barred functions to sets of partial environments. Moreover, $\mathcal{T}_{\rho,U}[p]$ is an auxiliary function implementing the iteration on all possible bindings for x.

Based on the transformation $\mathcal{T}_\rho$, we define in figure 10 the function $\mathcal{C}_\rho$ for calculating the costs incurred by preconditions, 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 $\mathsf{free}\ (\eta, \mathit{args})$ in the second line returns the set of variables from *args* which are not instantiated in η. Finally, we introduced an auxiliary function for evaluating the costs of universal quantification.

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

$$
\begin{array}{ll}
\mathcal{T}_\rho[R(args)]\ \mathcal{E} & = \{\mathsf{unify}\,(\eta, args, t) \mid \mathsf{unify}\,(\eta, args, t) \neq \square, \eta \in \mathcal{E}, t \in \rho R\} \\
\mathcal{T}_\rho[\neg R(args)]\ \mathcal{E} & = \{\mathsf{unify}\,(\eta, args, t) \mid \mathsf{unify}\,(\eta, args, t) \neq \square, \eta \in \mathcal{E}, t \notin \rho R\} \\
\mathcal{T}_\rho[p_1 \wedge p_2]\ \mathcal{E} & = \mathcal{T}_\rho[p_2]\ (\mathcal{T}_\rho[p_1]\ \mathcal{E}) \\
\mathcal{T}_\rho[p_1 \vee p_2]\ \mathcal{E} & = \mathcal{T}_\rho[p_1]\ \mathcal{E} \cup \mathcal{T}_\rho[p_2]\ \mathcal{E} \\
\mathcal{T}_\rho[\exists x : p]\ \mathcal{E} & = \mathbf{t\bar{l}}\,(\mathcal{T}_\rho[p]\ ((x \mapsto \square) :: \mathcal{E})) \\
\mathcal{T}_\rho[\forall x : p]\ \mathcal{E} & = \mathcal{T}_{\rho,\mathcal{U}}[p]\ ((x \mapsto \square) :: \mathcal{E})) \qquad \text{where} \\
 & \\
\mathcal{T}_{\rho,[\,]}[p]\ \mathcal{E} & = \mathbf{t\bar{l}}\,\mathcal{E} \\
\mathcal{T}_{\rho,a::U}[p]\ ((x \mapsto \_) :: \mathcal{E}) & = \mathcal{T}_{\rho,U}[p]\ (\mathcal{T}_\rho[p]\ ((x \mapsto a) :: \mathcal{E}))
\end{array}
$$

Fig. 9. The transformation induced by check.

$$
\begin{array}{ll}
\mathcal{C}_\rho[R(args)]\ \mathcal{E} & = \#\{\langle \eta, t\rangle \mid \eta \in \mathcal{E}, (\mathsf{first}\,(\eta, args))\ t \in \rho R\} \\
\mathcal{C}_\rho[\neg R(args)]\ \mathcal{E} & = \sum_{\eta \in \mathcal{E}} (\#\mathcal{U})^{\#\mathsf{free}(\eta, args)} \\
\mathcal{C}_\rho[p_1 \wedge p_2]\ \mathcal{E} & = \mathcal{C}_\rho[p_1]\ \mathcal{E} + \mathcal{C}_\rho[p_2]\ (\mathcal{T}_\rho[p_1]\ \mathcal{E}) \\
\mathcal{C}_\rho[p_1 \vee p_2]\ \mathcal{E} & = \mathcal{C}_\rho[p_1]\ \mathcal{E} + \mathcal{C}_\rho[p_2]\ \mathcal{E} \\
\mathcal{C}_\rho[\exists x : p]\ \mathcal{E} & = \mathcal{C}_\rho[p]\ ((x \mapsto \square) :: \mathcal{E}) \\
\mathcal{C}_\rho[\forall x : p]\ \mathcal{E} & = \mathcal{C}_{\rho,\mathcal{U}}[p]\ ((x \mapsto \square) :: \mathcal{E}) \qquad \text{where} \\
 & \\
\mathcal{C}_{\rho,[\,]}[p]\ \mathcal{E} & = \#\mathcal{E} \\
\mathcal{C}_{\rho,a::U}[p]\ ((x \mapsto \_) :: \mathcal{E}) & = \mathcal{C}_\rho[p]\ ((x \mapsto a) :: \mathcal{E})\ +\ \mathcal{C}_{\rho,U}[p]\ (\mathcal{T}_\rho[p]\ ((x \mapsto a) :: \mathcal{E})) \\
 & \\
\mathcal{C}^\bullet_\rho[\mathbf{1}]\ \mathcal{E} & = \#\mathcal{E} \\
\mathcal{C}^\bullet_\rho[R(args)]\ \mathcal{E} & = \#\{\langle \eta, t\rangle \mid \eta \in \mathcal{E}, t \in \rho R, \mathsf{unify}\,(\eta, args, t) \neq \square\} \\
\mathcal{C}^\bullet_\rho[p \Rightarrow c] & = \mathcal{C}_\rho[p]\ \mathcal{E} + \mathcal{C}^\bullet_\rho[c]\ (\mathcal{T}_\rho[p]\ \mathcal{E}) \\
\mathcal{C}^\bullet_\rho[c_1 \wedge c_2]\ \mathcal{E} & = \mathcal{C}^\bullet_\rho[c_1]\ \mathcal{E} + \mathcal{C}^\bullet_\rho[c_2]\ \mathcal{E} \\
\mathcal{C}^\bullet_\rho[\forall x : c]\ \mathcal{E} & = \mathcal{C}^\bullet_\rho[c]\ ((x \mapsto \square) :: \mathcal{E})
\end{array}
$$

Fig. 10. The costs of pre-conditions and clauses.

for his algorithm and the special case of Horn clauses [11]. 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. We then obtain our main theorem thereby demonstrating 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:

Theorem 1. *Assume that c is a alternation-free LFP formula of constant size. Then the solution ρ can be checked by the solver in time $\mathcal{O}(\#\rho + \mathcal{C}^\bullet_\rho[c]\ \mathcal{E}_0)$ where the set $\mathcal{E}_0$ consists of the single initial environment for the constants.*

For a formal proof see appendix B.

6 Experimental Evaluation

We have implemented the solver for ALFP formulae using SML of New Jersey. Furthermore, we have implemented several frontends for practically applying the solver to various control-flow analyses. Here, we exemplify the efficiency of the resulting system by two scalable benchmarks (executed on an 800 MHz Pentium III with 1 GB of main memory under Linux with SMLNJ, version 110.0.7).

The benchmark `trans2-n` computes the transitive closure $T_2 = E^+$ of a directed line graph with n vertices and $n-1$ edges (denoted E). It uses the clause:

$$\forall x : \forall y : E(x,y) \Rightarrow (T_2(x,y) \wedge \forall z : T_2(y,z) \Rightarrow T_2(x,z))$$

Thus, the universe contains $N = n$ elements, whereas the resulting relation for T_2 contains $\frac{1}{2}n \cdot (n+1)$ elements. The logically equivalent clause for $T_1 = E^+$

$$\forall x : \forall y : (E(x,y) \Rightarrow T_1(x,y)) \wedge (\forall z : T_1(x,z) \wedge T_1(z,y) \Rightarrow T_1(x,y))$$

turns out to yield a benchmark `trans1-n` that is much less efficient (see Fig. 11).

The theoretical runtime for the naive computation of transitive closure is cubic. However, in the case of the optimised formula, `trans2-n`, we can in fact use Theorem 1 to obtain a quadratic bound: given a universe of size n and a relation E of cardinality e (equal to $n-1$ in the benchmark for transitive closure), there are exactly e partial environments satisfying the query $E(x,y)$, giving us cost e for this query and the same cost also for the assertion $T_2(x,y)$. All these e environments (extended by $(z \mapsto \square)$) reach the query $T_2(y,z)$. Since there are at most n possible values for z, we conclude that $T_2(y,z)$ contributes at most $e \cdot n$ to the cost. Since the assertion $T_2(x,z)$ contributes the same cost, we obtain the complexity estimation $\mathcal{O}(e \cdot n)$. Indeed, the relative increase in the runtimes of `trans2-n` roughly shows the expected quadratic behaviour.

n	trans1-n	trans2-n
200	15.9	1.05
400	116.4	4.4
600	386.7	11.6
800	∞	16.2
1000	∞	29.1
1200	∞	50.0
1400	∞	57.5
1600	∞	76.2
1800	∞	128.0

n	N	(rel.)	router1-n	router2-n
5	216	1.0	2.99	0.16
6	318	1.47	6.89	0.21
7	440	2.04	13.4	0.39
8	582	2.69	22.4	0.57
9	744	3.44	35.1	0.79
10	926	4.3	56.8	1.41
15	2136	9.9	∞	5.01
20	3846	17.8	∞	13.0
25	6056	28.0	∞	26.7
30	8766	40.6	∞	55.9
35	11976	55.4	∞	116.9

Fig. 11. Runtimes on transitive closure and 0cfa on the ambient benchmark in seconds (including garbage collection).

The benchmarks `router1-n` and `router2-n` consist of an ambient program [4] describing a square array of n^2 servers and a packet to be routed through the array: it starts at the top left corner, then nondeterministically moves either right or down, to end in the lower right corner. On this program we perform the 0cfa control-flow analysis as presented in [12] giving the numbers in `router1-n` as well as an optimised version of this analysis – giving the numbers in `router2-n`. The optimisation basically amounted to avoiding large auxiliary relations and replacing filtering of larger relations by creating auxiliary relations which directly provide the desired (few) tuples. The two versions of the analysis can be found in appendix C. The router example is typical for related applications in program analysis as the computed relations turn out to be sparse, i.e.,

have approximately linear cardinalities. Here, the cardinality of the universe is $N = 10n^2 - 8n + 6$ but the computed relations are approximately of the same size.

The theoretical runtime for 0cfa on ambient programs is cubic in N (as reported in [12] and as may be obtained using Theorem 1). However, even the original analysis `router1-n` from [12] seems to behave significantly better in practice, but still it uses about a minute for `router1-10`, and no termination in reasonable time could be obtained for `router1-15`. More than 30 times faster runtimes could be observed when scaling the optimised analysis `router2-n`. Also, the relative increase is perhaps even less than quadratic when compared to the relative increase of the respective universes. Accordingly, more than ten times larger input ambient programs could be analyzed in reasonable time. The absolute runtimes both on transitive closure and the ambient analysis indicate that the fixpoint algorithm – even in its functional implementation – is indeed quite efficient. (We coded an implementation of the transitive closure in `C` and obtained only about one order of magnitude speed-up.) The largest examples we could do in reasonable time (a few minutes) were `trans2-1800` and `router2-35`. The latter is an ambient program of around 100 KB size on which the analyzer produces about 1.4 MB output. We also have tried the analyzer on even larger examples but there the runtimes increased dramatically – due to growing space consumption and therefore exploding garbage collection and paging times.

7 Conclusion

Our solver algorithm is clearly based on classical work on efficient fixpoint algorithms [9]. In particular, it combines the topdown solving approach of Le Charlier and van Hentenryck [6] with the propagation of differences [8], an optimisation technique for distributive frameworks which is also known in the area of deductive databases [3] or as *reduction of strength* transformations for program optimisation [13]. 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 [11] or off-the-shelf implementations of deductive databases as, e.g., the CORAL system [14].

The complexity analysis has benefitted from the pioneering ideas of McAllester [11] on the complexity of solving (classical) Horn clauses. Here, we generalised these techniques to a richer class of input formulas 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.

References

1. A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming (SCP)*, 35(2):79–111, 1999.
2. 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.
3. 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.
4. L. Cardelli and A.D. Gordon. Mobile ambients. In *Proceedings of FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.
5. A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 25(2):156–178, 1980.

6. B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.
7. E. Dahlhaus. Skolem normal forms concerning the least fixpoint. In *Computation Theory and Logic*, pages 101–106. LNCS 270, Springer Verlag, 1987.
8. 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.*
9. C. Fecht and H. Seidl. A Faster Solver for General Systems of Equations. *Science of Computer Programming (SCP)*, 35(2-3):137–162, 1999.
10. 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.
11. D. McAllester. On the Complexity Analysis of Static Analyses. In *6th Static Analysis Symposium (SAS)*, pages 312–329. LNCS 1694, Springer Verlag, 1999.
12. F. Nielson and H. Seidl. Control-Flow Analysis in Cubic Time. In *European Symposium on Programming (ESOP)*. LNCS, Springer Verlag, 2001. To appear.
13. R. Paige. Symbolic Finite Differencing – Part I. In *Proceedings of 3rd European Symposium on Programming (ESOP)*, pages 36–56. LNCS 432, 1990.
14. R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL Deductive System. *VLDB Journal*, 3(2):161–210, 1994.
15. Joseph E. Stoy. *Denotational Semantics: The Scott–Strachey approach to programming language semantics.* MIT Press, 1977.

A Proof of Proposition 1

Assume $\mathsf{clause} \equiv s_1 \wedge \ldots \wedge s_m$ where s_j is the clause corresponding to stratum j, and let $\mathcal{R}_j$ denote the set of all relation symbols R defined in $s_1 \wedge \ldots \wedge s_j$ taking $\mathcal{R}_0 = \emptyset$, and recall the rank function r. Let M denote a set of assignments which map relation symbols to relations; then $\rho = \bigsqcap M$ is given by the formula

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

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

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

Claim: If $\rho = \bigsqcap M$, c occurs in s_j and $(\rho', \sigma) \models c$ for all $\rho' \in M$ then also $(\rho, \sigma) \models c$.

In order to prove the Claim, we proceed by complete induction on j and in each case distinuish two cases.

Case 1: 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 occuring in s_j. We consider two illustrative cases.

$\underline{c \equiv R(x_1, \ldots, x_k)}$ This case is immediate since we assumed that $\rho(R) = \mathcal{U}^k$.

$\underline{c \equiv (p \Rightarrow c')}$ / $(\rho, \sigma) \models c.$ By the induction hypothesis $(\rho, \sigma) \models c'$ and hence it is immediate that also

Case 2: 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:

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

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

$\underline{c \equiv R(x_1, \ldots, x_k)}$ Since $r(R) = j$ we have that $\rho(R)$ equals the intersection of all relations $\rho'(R)$ for $\rho' \in M_j$. Given the assumption that $t = (\sigma\, x_1, \ldots, \sigma\, x_k) \in \rho'(R)$ for all $\rho' \in M_j \subseteq M$ we have that $t \in \rho(R)$ as desired.

$\underline{c \equiv (p \Rightarrow c')}$ We consider 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 Auxiliary Claim below we get that $(\rho', \sigma) \models p$ for all $\rho' \in M_j$ and hence the result follows by induction hypothesis.

Auxiliary Claim: If $\rho = \bigsqcap M$, p occurs in s_j and $(\rho, \sigma) \models p$ then also $(\rho', \sigma) \models p$ for all $\rho' \in M_j$.

We proceed by induction in j and in each case perform a structural induction in p (recalling that p occurs in s_j). Most cases are straightforward since $\rho(R) \subseteq \rho'(R)$ for all $\rho' \in M_j$. The only non-trivial case is:

$\underline{p \equiv \neg R(x_1, \ldots, x_k)}$ Here the result follows because the rank of R is strictly less than j and hence $\rho'(R) = \rho(R)$ for all $\rho' \in M_j$. □

B Proof of Theorem 1

In the sequel we make use of the following technical proposition:

Proposition 4. *Assume that the current interpretation of predicate symbols is given by ρ. Then for all pre-conditions p and sets $\mathcal{E}$ of partial environments, the following holds:*

1. *$\mathcal{T}_\rho[p]\ \mathcal{E}$ equals the set of partial environments η' which on calls $\mathsf{check}\,(p, n)\,\eta$, $\eta \in \mathcal{E}$, are propagated to the continuation n;*
2. *$\#\,(\mathcal{T}_\rho[p]\ \mathcal{E}) \leq \mathcal{C}_\rho[p]\ \mathcal{E}$.* □

When started on a solution ρ, the solver 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 p and c denote a pre-condition and a clause, respectively. Let $T[p, \mathcal{E}]$ denote the maximal time spent by the algorithm on the call $\mathsf{check}\,(p, n)\,\eta$ for any partial environment η from $\mathcal{E}$ before calling continuation n. Accordingly, let $T[c, \mathcal{E}]$ denote the total time spent by the algorithm on calls $\mathsf{execute}\,c\,\eta$, $\eta \in \mathcal{E}$. We estimate these complexities as follows:

Proposition 5. *There are constants $d_p, d_c > 0$ only depending on p and c, respectively, such that*

$$T[p, \mathcal{E}] \leq d_p \cdot (\#\mathcal{E} + \mathcal{C}_\rho[p]\ \mathcal{E})$$
$$T[c, \mathcal{E}] \leq d_c \cdot (\#\mathcal{E} + \mathcal{C}^\bullet_\rho[c]\ \mathcal{E})$$

As $\#\mathcal{E}_0 = 1$, we conclude from proposition 5, that the solver uses time $\mathcal{O}(\#\rho + \mathcal{C}^\bullet_\rho[c]\ \mathcal{E}_0)$ as stated in theorem 1 — where the extra time $\mathcal{O}(\#\rho)$ is needed to read the relations $\rho\,R$ and insert them into the datastructure rho.

Proof (Proposition 5). Here, we only consider the cases where

- p is a query or a conjunction, and where
- c is an assertion or an implication.

$\underline{p \equiv R(args)}$ For $\eta \in \mathcal{E}$ and $t_1 = \mathsf{first}\,(\eta, args)$, let T_η denote the set of all tuples $t_1 @ t_2 \in \rho R$. The work of calls $\mathsf{check}\,(p, n)\,\eta$, $\eta \in \mathcal{E}$, consists of

1. registering for R and t_1 certain functions $f_\eta, \eta \in \mathcal{E}$;
2. determining for each $\eta \in \mathcal{E}$, the set T_η and computing $\mathsf{unify}\,(\eta, \mathit{args}, t')$ for all $t' \in T_\eta$.

The first task consumes time $\mathcal{O}(\#\mathcal{E})$. For the latter, according to our tree-like representation of relations, the solver will need time $\mathcal{O}(\#T_\eta)$ for each η. Therefore, we obtain:

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

for some $d > 0$ – giving the claim of proposition 5 for this case.

$\underline{p \equiv p_1 \wedge p_2}$ Then the work of calls $\mathsf{check}\,(p, n)\,\eta$, $\eta \in \mathcal{E}$, consists of

1. the work of all calls $\mathsf{check}\,(p_1, n')\,\eta$, $\eta \in \mathcal{E}$, where $n' = \mathsf{check}\,(p_2, n)$ – thereby computing the set of partial environments $\mathcal{E}' = \mathcal{T}_\rho[p_1]\,\mathcal{E}$;
2. the work of all calls $\mathsf{check}\,(p_2, n)\,\eta'$, $\eta' \in \mathcal{E}'$.

By inductive hypothesis, the first task consumes time $\mathcal{O}(\#\mathcal{E} + \mathcal{C}_\rho[p_1]\,\mathcal{E})$ whereas the second one takes time $\mathcal{O}(\#\mathcal{E}' + \mathcal{C}_\rho[p_2]\,\mathcal{E}')$. Now, by proposition 4,

$$\#\mathcal{E}' = \#\mathcal{T}_\rho[p_1]\,\mathcal{E} \leq \mathcal{C}_\rho[p_1]\,\mathcal{E}$$

Therefore,

$$T[p, \mathcal{E}] \leq d \cdot (\#\mathcal{E} + \mathcal{C}_\rho[p_1]\,\mathcal{E} + \mathcal{C}_\rho[p_2]\,(\mathcal{T}_\rho[p_1]\,\mathcal{E}))$$

according to our claim.

$\underline{c \equiv R(\mathit{args})}$ The work of the solver on the assertion c when executing the calls $\mathsf{execute}\,c\,\eta$, $\eta \in \mathcal{E}$, amounts to:

- determining for every $\eta \in \mathcal{E}$, the set T_η of all t with $\mathsf{unify}\,(\eta, \mathit{args}, t) \neq \square$;
- adding the sets T_η to the current value for the predicate R.

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

$$\begin{aligned}\textstyle\sum_{\eta \in \mathcal{E}}(1 + \#T_\eta) &\leq \#\mathcal{E} + \#\{(\eta, t) \mid \eta \in \mathcal{E}, t \in \rho R, \mathsf{unify}\,(\eta, \mathit{args}, t) \neq \square\} \\ &= \#\mathcal{E} + \mathcal{C}^\bullet_\rho[c]\,\mathcal{E}\end{aligned}$$

— giving the assertion of proposition 5 for this case.

$\underline{c \equiv (p \Rightarrow c')}$ The work of the solver on calls $\mathsf{execute}\,c\,\eta$, $\eta \in \mathcal{E}$, amounts to:

1. the work on calls $\mathsf{check}\,(p, \mathsf{execute}\,c')\,\eta$, $\eta \in \mathcal{E}$; together with
2. the work on calls $\mathsf{execute}\,c'\,\eta'$ for $\eta' \in \mathcal{T}_\rho[p]\,\mathcal{E}$, i.e., those η' which are obtained by the calls $\mathsf{check}\,(p, \mathsf{execute}\,c')\,\eta$, $\eta \in \mathcal{E}$.

Therefore by inductive hypothesis and prop. 4,

$$\begin{aligned}T[c, \mathcal{E}] &\leq d_0 + T[p, \mathcal{E}] + T[c', \mathcal{T}_\rho[p]\,\mathcal{E}] \\ &\leq d \cdot (\#\mathcal{E} + \mathcal{C}_\rho[p]\,\mathcal{E} + \#\,(\mathcal{T}_\rho[p]\,\mathcal{E}) + \mathcal{C}^\bullet_\rho[c']\,(\mathcal{T}_\rho[p]\,\mathcal{E})) \\ &\leq d \cdot (\#\mathcal{E} + 2 \cdot \mathcal{C}_\rho[p]\,\mathcal{E} + \mathcal{C}^\bullet_\rho[c']\,(\mathcal{T}_\rho[p]\,\mathcal{E})) \\ &\leq 2 \cdot d \cdot (\#\mathcal{E} + \mathcal{C}^\bullet_\rho[c]\,\mathcal{E})\end{aligned}$$

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

C The Benchmark Clauses for Ambient Analysis

Our frontend extracts from the ambient program the ternary relations In/3, Out/3, Open/3 and Name/3. The first component always contains the label of the statically enclosing ambient. In case of the first three relations, the last component holds the labels of corresponding capabilities, and the second component the names of ambients the capabilities possibly refer to. In case of the relation Name/3, the third component holds the labels of ambients which are named by the names given in the second component.

Then the analysis from [12] is equivalent to the clause:

$$
\begin{aligned}
&(\forall l, n, x : (\mathsf{In}(l,n,x) \Rightarrow (\mathsf{HasFather}(x,l) \wedge \\
&\qquad (\forall y, z : \mathsf{HasFather}(x,y) \wedge \mathsf{Sibling}(y,z) \wedge \mathsf{HasName}(z,n) \Rightarrow \mathsf{HasFather}(y,z)))) \wedge \\
&\qquad (\mathsf{Out}(l,n,x) \Rightarrow (\mathsf{HasFather}(x,l) \wedge \\
&\qquad (\forall y, z : \mathsf{HasFather}(x,y) \wedge \mathsf{HasFather}(y,z) \wedge \mathsf{HasName}(z,n) \Rightarrow \mathsf{OutPair}(y,z)))) \wedge \\
&\qquad (\mathsf{Open}(l,n,x) \Rightarrow (\mathsf{HasFather}(x,l) \wedge \\
&\qquad (\forall y, z : \mathsf{HasFather}(x,y) \wedge \mathsf{HasFather}(z,y) \wedge \mathsf{HasName}(z,n) \Rightarrow \mathsf{OpenPair}(z,y)))) \wedge \\
&\qquad (\mathsf{Name}(l,n,x) \Rightarrow (\mathsf{HasName}(x,n) \wedge \mathsf{HasFather}(x,l)))) \wedge
\end{aligned}
$$

$$(\forall y, z, t : \mathsf{HasFather}(y,t) \wedge \mathsf{HasFather}(z,t) \Rightarrow \mathsf{Sibling}(y,z)) \wedge$$

$$(\forall x, y, z : \mathsf{OutPair}(x,y) \wedge \mathsf{HasFather}(y,z) \Rightarrow \mathsf{HasFather}(x,z)) \wedge$$

$$(\forall x, y, z : \mathsf{OpenPair}(y,z) \wedge \mathsf{HasFather}(x,y) \Rightarrow \mathsf{HasFather}(x,z))$$

Evaluating this clause on the router programs gives the numbers in column `router1-n` of figure 11.

This clause turns out to be not as efficient as one might have hoped. In particular, the Sibling-relation becomes very large: for $n = 10$, it contains no less than 568929 tuples – although the relation HasFather/2 only comprises 2004 tuples. In order to optimise the clause further, we make therefore the following observations:

- It is not necessary to consider each capability individually. The only property which matters for the analysis is whether an ambient contains an In-operation for a specific name or not (similar for Out and Open). Therefore, we introduce new auxiliary binary predicates HasIn/2, HasOut/2 and HasOpen/2.
- Instead of determining all fathers of an ambient and then selecting from these the fathers having a specific name, it is much more efficient to access the fathers with a given name directly. This gives us the ternary relation HasFather/3. A similar trick is also played to rapidly access the ambient sons with a given name.
- Using the latter ternary relation, we replace the conjunction

  $$\mathsf{Sibling}(y,z) \wedge \mathsf{HasName}(z,n)$$

  with the conjunction:

  $$\mathsf{HasFather}(y,t) \wedge \mathsf{HasSon}(t,n,z)$$

  for some new auxiliary variable t – thus avoiding the large relation Sibling/2.

By application of these ideas, we arrive at:

$$
\begin{array}{ll}
(\forall l, n, x: & (\mathsf{In}(l,n,x) \Rightarrow (\mathsf{HasFather}(x,l) \wedge \\
& \quad (\forall y : \mathsf{HasFather}(x,y) \Rightarrow \mathsf{HasIn}(y,n)))) \wedge \\
& (\mathsf{Out}(l,n,x) \Rightarrow (\mathsf{HasFather}(x,l) \wedge \\
& \quad (\forall y : \mathsf{HasFather}(x,y) \Rightarrow \mathsf{HasOut}(y,n)))) \wedge \\
& (\mathsf{Open}(l,n,x) \Rightarrow (\mathsf{HasFather}(x,l) \wedge \\
& \quad (\forall y : \mathsf{HasFather}(x,y) \Rightarrow \mathsf{HasOpen}(y,n)))) \wedge \\
& (\mathsf{Name}(l,n,x) \Rightarrow (\mathsf{HasName}(x,n) \wedge \mathsf{HasFather}(x,l) \wedge \\
& \quad (\forall y : \mathsf{HasFather}(x,y) \Rightarrow (\mathsf{HasSon}(y,n,x) \wedge \\
& \quad (\forall n : \mathsf{HasName}(y,n) \Rightarrow \mathsf{HasFather}(x,n,y)))))) \wedge \\
(\forall y, n, z, t : & \mathsf{HasIn}(y,n) \wedge \mathsf{HasFather}(y,t) \wedge \\
& \mathsf{HasSon}(t,n,z) \Rightarrow \mathsf{HasFather}(y,z))) \wedge \\
(\forall x, y, n : & \mathsf{HasOut}(x,n) \wedge \mathsf{HasFather}(x,n,y) \Rightarrow \mathsf{OutPair}(x,y)) \wedge \\
(\forall x, y, z : & \mathsf{OutPair}(x,y) \wedge \mathsf{HasFather}(y,z) \Rightarrow \mathsf{HasFather}(x,z)) \wedge \\
(\forall x, y, n : & \mathsf{HasOpen}(x,n) \wedge \mathsf{HasSon}(x,n,y) \Rightarrow \mathsf{OpenPair}(y,x)) \wedge \\
(\forall x, y, z : & \mathsf{HasFather}(x,y) \wedge \mathsf{OpenPair}(y,z) \Rightarrow \mathsf{HasFather}(x,z))
\end{array}
$$

The numbers for this clause are listed in column `router2-n` of figure 11.