Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 267 (2010) 15–28
www.elsevier.com/locate/entcs

Explicit Versus Symbolic Algorithms for Solving ALFP Constraints

Piotr Filipiuk Hanne Riis Nielson Flemming Nielson

DTU Informatics
Technical University of Denmark
Kongens Lyngby, Denmark
email: {pifi,riis,nielson}@imm.dtu.dk

Abstract

ALFP, Alternation-free Least Fixed Point logic, has successfully been used as an intermediate language in the implementation of static analysis and model checking problems. Clearly different analysis problems may give rise to ALFP clauses with different characteristics. There are also different approaches to solving ALFP clauses and some of those are better suited for certain kinds of clauses than others. The aim of this paper is to present two algorithms, one that is based on differential worklists and one based on BDD's, and experiment with them.

Keywords: Static analysis, constraint solving, Alternation-free Least Fixed Point logic, binary decision diagrams, differential worklist algorithm.

1 Introduction

Static analysis can be seen as a two-phase process where we first transform the analysis problem into a set of constraints that, in the second phase, is solved to produce the analysis result of interest. The constraints may be expressed in a language tailored to the problem at hand, or they may be expressed in a general purpose constraint language; in this paper we follow the latter approach and consider ALFP, Alternation-free Least Fixed Point Logic, which is an extension of Datalog. ALFP has successfully been used as the constraint language for sophisticated analyses of a wide variety of programming paradigms including imperative, functional, concurrent and mobile languages and more recently for model checking [4,12].

While a wide variety of analysis problems can be rephrased into the *same* constraint language it is not necessarily the case that they all will benefit from the *same* solver techniques. We can for example imagine that certain solver techniques are better at handling certain types of input formulae, due to the use of specialized data structures. It is therefore desirable to be able to experiment with different data structures and algorithms for solving the constraints. Similar considerations

doi:10.1016/j.entcs.2010.09.015

can also be found in the area of model checking where the same tool may support different combinations of data structures and algorithms thereby allowing the user to choose the most appropriate combination for the problem at hand.

In this paper we report on two solver algorithms being developed for ALFP: one being a differential worklist algorithm, originally developed in [14], that is based on a representation of relations as prefix trees [14] and the other being a continuation passing style algorithm that is based on a BDD representation of relations [5]. BDD's, originally designed for hardware verification, have already been used in a number of program analyses [19,3] and proven to be very efficient.

The rest of this paper is organized as follows. We first give the necessary background information on Alternation-free Least Fixed Point Logic in Section 2. In Section 3 we describe the overall structure of the algorithm that is shared between the two solver algorithms. Sections 4 and 5 describe the data structures and the algorithms of the differential and the BDD based algorithms respectively. We continue in Section 6 with experiments, employing model checking to verify a number of properties. We conclude in Section 7.

2 The logic ALFP

The alternation-free fragment of Least Fixed Point Logic (ALFP) [14] is an extension of Horn clauses allowing both existential and universal quantifications in preconditions, negative queries, disjunctions of preconditions, and conjunctions of conclusions. In order to deal with negative queries, we restrict ourselves to *alternation-free* formulae that are subject to a notion of *stratification* defined below.

Definition 2.1 Given a fixed countable set $\mathcal{X}$ of variables, a non-empty and finite universe $\mathcal{U}$ and a finite alphabet $\mathcal{R}$ of predicate symbols, we define the set of ALFP formulae (or clause sequences), cls, together with clauses, cl, and preconditions, pre, by the grammar:

$$\begin{array}{lcl} pre & ::= & R(v_1,\ldots,v_k) \mid \neg R(v_1,\ldots,v_k) \mid pre_1 \wedge pre_2 \\ & \mid & pre_1 \vee pre_2 \mid \exists x : pre \mid \forall x : pre \\ cl & ::= & R(v_1,\ldots,v_k) \mid \mathbf{1} \mid cl_1 \wedge cl_2 \mid pre \Rightarrow cl \mid \forall x : cl \\ cls & ::= & cl_1, \cdots, cl_s \end{array}$$

Here $v_i \in (\mathcal{X} \cup \mathcal{U})$, $x \in \mathcal{X}$, $R \in \mathcal{R}$, $k \geq 0$ and $s \geq 1$.

Occurrences of R and $\neg R$ in preconditions are called *positive queries* and *negative queries*, respectively, whereas the other occurrences of R are called *assertions*. We write $\mathbf{1}$ for the always true clause.

In order to ensure desirable theoretical and pragmatic properties in the presence of negation, we impose a notion of *stratification* similar to the one in Datalog [1,6]. Intuitively, stratification ensures that a negative query is not performed until the predicate has been fully asserted. This is important for ensuring that once a precondition evaluates to true it will continue to be true even after further assertions

of predicates.

Definition 2.2 The formula $cls = cl_1, \cdots, cl_s$ is stratified if there exists a function $rank : \mathcal{R} \to \{0, \cdots, s\}$ such that for all $i = 1, \cdots, s$:

- $rank(R) = i$ for every assertion R in cl_i;
- $rank(R) \leq i$ for every positive query R in cl_i; and
- $rank(R) < i$ for every negative query $\neg R$ in cl_i.

To specify the semantics of ALFP we shall introduce the interpretations $\varrho : \mathcal{R} \to \bigcup_k \mathcal{U}^k$ and $\varsigma : \mathcal{X} \to \mathcal{U}$ for predicate symbols and variables, respectively. We shall write $\varrho(R)$ for the set of k-tuples $(a_1, \ldots, a_k)$ from $\mathcal{U}^k$ associated with the k-ary predicate R and we write $\varsigma(x)$ for the atom of $\mathcal{U}$ bound to x. In the sequel we view the free variables occurring in a formula as constants from the finite universe $\mathcal{U}$. The satisfaction relations for preconditions pre, clauses cl and clause sequences cls are specified by:

$$(\varrho, \varsigma) \models pre, \quad (\varrho, \varsigma) \models cl \quad \text{and } (\varrho, \varsigma) \models cls$$

The formal definition is standard and is given in Appendix A.

3 Abstract algorithm

We shall present two algorithms for solving clause sequences; one being a differential worklist algorithm inspecting the tuples of the relations on an individual basis and another being a BDD based algorithm where the operations are performed at the level of relations. Although the underlying data structures of the two algorithms are very different they share the same overall structure and we shall present this in the present section and leave a more detailed discussion of the two algorithms to the next two sections.

Both algorithms operate with (intermediate) representations of the two interpretations ς and ϱ of the semantics; we shall call them `env` and `result`, respectively, in the following. In both algorithms `result` will be an imperative data structure that is updated as we progress. The data structure `env` is supplied as a parameter to the functions of the algorithms.

For both algorithms we have one function for each of the three syntactic categories. The function SOLVE takes a *clause sequence* as input and will call the function EXECUTE on each of the individual clauses. In the case of the differential worklist algorithm this takes the form

$$\text{SOLVE}(cl_1, \cdots, cl_s) = \text{EXECUTE}(cl_1)[\,]; \cdots ; \text{EXECUTE}(cl_s)[\,]$$

where we write $[\,]$ for the empty environment reflecting that we have no free variables in the clause sequences. In the case of the BDD based algorithm we shall use an explicit (imperative) stack keeping track of the clauses to be considered so here the

function takes the form

$$\begin{aligned}\textsc{solve}(cl_1, \cdots, cl_s) = {} & \textbf{let } \texttt{stack} = [cl_1, \cdots, cl_s] \\ & \textbf{while } \texttt{stack} \neq [\,] \textbf{ do } \textsc{execute}(\textsc{pop } \texttt{stack})[\,]\end{aligned}$$

where the function POP, as a side effect, pops the stack and returns its top element.

In both algorithms the function EXECUTE takes a *clause cl* as a parameter and a representation `env` of the interpretation of the variables. We have one case for each of the forms of *cl*:

$$\begin{aligned}\textsc{execute}(R(v_1, \cdots, v_k))\texttt{env} &= \cdots \\ \textsc{execute}(\mathbf{1})\texttt{env} &= () \\ \textsc{execute}(cl_1 \wedge cl_2)\texttt{env} &= \textsc{execute}(cl_1)\texttt{env}; \textsc{execute}(cl_2)\texttt{env} \\ \textsc{execute}(pre \Rightarrow cl)\texttt{env} &= \textsc{check}(pre, \textsc{execute}(cl))\texttt{env} \\ \textsc{execute}(\forall x : cl)\texttt{env} &= \textbf{let } \texttt{env}' = \cdots \textbf{ in } \textsc{execute}(cl)\texttt{env}'\end{aligned}$$

In the case of assertions the details depend on the actual algorithm and we shall return to those later. The case of conjunction is straightforward as we have to inspect both clauses. In the case of implication we make use of the function CHECK that in addition to the precondition and the environment also takes the continuation $\textsc{execute}(cl)$ as an argument. In the case of universal quantification we perform a recursive call using an updated environment; the details of which depend on the actual algorithm.

In both algorithms the function CHECK takes a *precondition*, a continuation and an environment as parameters. The treatment of queries depends on the actual algorithm and so does the treatment of disjunction and universal quantification; except from the fact that the overall structure is:

$$\begin{aligned}\textsc{check}(R(v_1, \cdots, v_k), next)\texttt{env} &= \cdots \\ \textsc{check}(\neg R(v_1, \cdots, v_k), next)\texttt{env} &= \cdots \\ \textsc{check}(pre_1 \wedge pre_2, next)\texttt{env} &= \textsc{check}(pre_1, \textsc{check}(pre_2, next))\texttt{env} \\ \textsc{check}(pre_1 \vee pre_2, next)\texttt{env} &= \cdots \\ \textsc{check}(\exists x : cl, next)\texttt{env} &= \textbf{let } next' = next \circ \cdots \\ & \quad\ \textbf{let } \texttt{env}' = \cdots \\ & \quad\ \textbf{in } \ \textsc{check}(cl, next')\texttt{env}' \\ \textsc{check}(\forall x : cl, next)\texttt{env} &= \cdots\end{aligned}$$

For conjunction we exploit a continuation passing programming style and for existential quantification we perform a recursive call using an updated environment

and an updated continuation; the details of which depend on the actual algorithm.

In the following two sections we give more details of the data structures used by the two algorithms and the missing cases in the above definitions.

4 Instantiation with differential worklist

In this section we present the main data structures and the details of the differential worklist algorithm [14,13]. The algorithm computes the relations in increasing order on their rank and therefore the negations present no obstacles. It combines the top-down solving approach of Le Charlier and van Hentenryck [7] with the propagation of differences [8], an optimization technique for distributive frameworks which is also known in the area of deductive databases [2] or as reduction of strength transformations for program optimization [15]. As mentioned above the main data structures are `env` and `result` representing the (partial) interpretation of variables and predicates, respectively.

Here `env` is implemented as an array indexed by the variables and returning either $None$, which means that the variable is undefined or $Some(a)$, which means that the variable is bound to $a \in \mathcal{U}$. By using arrays we ensure that the values of variables can be accessed in constant time. The main operation on `env` is the function `unify` given by

$$\texttt{unify}(\texttt{env}, v, a) = \begin{cases} \texttt{env} & \text{if } (v \in \mathcal{X} \wedge \texttt{env}[v] = Some(a)) \vee v = a \\ \texttt{env}[v \mapsto Some(a)] & \text{if } v \in \mathcal{X} \wedge \texttt{env}[v] = None \\ \text{fail} & \text{otherwise} \end{cases}$$

It is extended to k-tuples in a straightforward way. The function UNIFIABLE will, when applied to `env` and a tuple $(v_1, \cdots, v_k)$, return the subset of $\mathcal{U}^k$ for which `unify` will succeed.

The interpretation of the predicate symbols ϱ is given by the global data structure `result`, which is updated incrementally during computations. It is represented as a prefix tree that for each predicate R records the tuples currently known to belong to R. The prefix trees themselves are implemented using a dynamic array that for each node n of the tree keeps a list of all currently available successor atoms, together with a hash table that maps pairs (n, a) of nodes n and atoms a to successor nodes. There are three main operations on the data structure `result`: the operation `result.`HAS checks whether a tuple of atoms from the universe is associated with a given predicate, the operation `result.`SUB returns a list of the tuples associated with a given predicate and the operation `result.`ADD adds a tuple to the interpretation of a given predicate.

Since ϱ is not completely determined from the beginning it may happen that a query $R(v_1, \cdots, v_k)$ inside a precondition 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 consequences that

adding $(a_1, \cdots, a_k)$ to R will have on the contents of other predicates. This gives rise to introducing yet another global data structure `infl` that records computations that have to be resumed for the new tuples; these future computations will be called *consumers*. The `infl` data structure is also represented as a prefix tree that for each predicate R records consumers that have to be resumed when the interpretation of R is updated. There are two main operations on the data structure `infl`: the operation `infl.`REGISTER that adds a new consumer for a given predicate and `infl.`CONSUMERS that activates all the consumers currently associated with a given predicate.

Let us now return to the description of the function EXECUTE for the cases that are specific for the differential algorithm, that is, the case of assertion and the case of universal quantification. In case of assertions the algorithm is as follows

```
EXECUTE(R(v_1, ··· , v_k))env =
    let ITERFUN (a_1, ··· , a_k) =
        match result.HAS(R, (a_1, ··· , a_k)) with
            | true → ()
            | false →
                result.ADD(R, (a_1, ··· , a_k))
                ITER (fun f → f (a_1, ··· , a_k)) (infl.CONSUMERS R)
    in ITER ITERFUN (UNIFIABLE(env,(v_1, ··· , v_k)))
```

The function uses the auxiliary function ITER, which applies the function ITERFUN to each element of the list of k-tuples that can be unified with the argument $(v_1, \cdots, v_k)$. Given a tuple $(a_1, \cdots, a_k)$, the function ITERFUN adds the tuple to the interpretation of R stored in `result` if it is not already present. If the ADD operation succeeds, we first create a list of all the consumers currently registered for predicate R by calling the function `infl.`CONSUMERS. Thereafter, we resume the computations by iterating over the list of consumers and calling corresponding continuations.

In the case of universal quantification, we simply extend the environment to record that the value of the new variable is unknown and then we recurse:

```
EXECUTE(∀x : cl)env = EXECUTE(cl)((x, None) :: env)
```

Turning to the CHECK function let us first consider the algorithm in the case of positive queries:

```
CHECK(R(v_1, ··· , v_k), next)env =
    let CONSUMER (a_1, ··· , a_k) =
        match UNIFY(env, (v_1, ··· , v_k), (a_1, ··· , a_k)) with
            | fail → ()
            | env' → next env'
    in infl.REGISTER(R,CONSUMER); ITER CONSUMER (result.SUB R)
```

We first ensure that the consumer is registered in `infl`, by calling function REGISTER, so that future tuples associated with R will be processed. Thereafter, the function inspects the data structure `result` to obtain the list of tuples associated with the predicate R. Then, the auxiliary function CONSUMER unifies each tuple

with $(v_1, \cdots, v_k)$; and if the operation succeeds, the continuation *next* is invoked on the updated new environment.

In the case of negated query, the algorithm is of the form

$\textsc{check}(\neg R(v_1, \cdots, v_k), next)\texttt{env} =$
  let ITERFUN $(a_1, \cdots, a_k) =$
    match `result.`HAS$(R, (a_1, \cdots, a_k))$ **with**
      $\mid true \rightarrow ()$
      $\mid false \rightarrow next$ (UNIFY$(\texttt{env}, (v_1, \cdots, v_k), (a_1, \cdots, a_k))$)
  in ITER ITERFUN (UNIFIABLE$(\texttt{env}, (v_1, \cdots, v_k))$)

The function first computes the tuples unifiable with $(v_1, \cdots, v_k)$ in the environment `env`. Then, for each tuple it checks if the tuple is already in R and if not, the tuple is unified with $(v_1, \cdots, v_k)$ to produce new environment in which the continuation *next* is evaluated.

The CHECK function for disjunction of preconditions is as follows

$\textsc{check}(pre_1 \vee pre_2, next)\texttt{env} =$
  CHECK$(pre_1, next)$`env`; CHECK$(pre_2, next)$`env`

The function simply checks preconditions pre_1 and pre_2 respectively in the current environment `env`. In order to be efficient we use memoization; this means that if both checks yield the same bindings of variables, the second check does not need to consider the continuation, as it has already been done.

The algorithm for existential quantification checks the precondition *pre* in the environment extended with the quantified variable. The continuation that is passed is a composition of functions *next* and TAIL, where the function TAIL removes quantified variable from the environment passed as a parameter. The algorithm is as follows:

$\textsc{check}(\exists x : pre, next)\texttt{env} =$
  CHECK$(pre, next \circ \textsc{tail})((x, None) :: \texttt{env})$

In the case of universal quantification the function CHECK needs to inspect all atoms from the universe and find the extensions of `env` that are compatible with the precondition *pre*. In order to do that we iterate over the entire universe, successively binding the atoms to x and modifying the partial environments to be compatible with the precondition *pre*. We enumerate the universe using the auxiliary function LOOP, which is initially called with the complete list of atoms in the universe. The recursive structure of the function LOOP reflects the fact that universal quantification is a conjunction over the entire universe. The pseudo code for the case is as follows:

$\textsc{check}(\forall x : pre, next)\texttt{env} =$
  let LOOP U' `env'` $=$
    match U' **with**
      $\mid hd :: tl \rightarrow$ CHECK$(pre,$ LOOP $tl)$ $((x, Some(hd)) :: \texttt{env'})$
      $\mid [\,] \rightarrow next$ `env'`
  in LOOP U $((x, None) :: \texttt{env})$

5 Instantiation with BDD's

We now turn our attention to the BDD based algorithm. This algorithm also makes use of the data structures `env` and `result`, but this time they are represented as binary decision diagrams, or, to be more precise, by ordered binary decision diagrams (OBDD's). The use of BDD's allows us to operate on entire relations, rather than on individual tuples - as in the differential worklist algorithm. Furthermore, the cost of the BDD operations depends on the size of the BDD and not the number of tuples in the relation; hence dense relations can be computed efficiently as long as their encoded representations are compact.

Each BDD is defined over a finite sequence of distinct domain names. The main operations on BDD's, to be used in the following, are given by means of operations on the relations they represent. Given two relations with the same domain names, the operations *union*, $\cup$, and *proper subset testing*, $\subsetneq$, are defined as a corresponding operations on the set of their tuples. The *projection* operation, π, selects the subset of domains from the relation and removes all other domains. The *select* operation, σ_b, selects all tuples from the relation for which the given condition b holds. The *complement* operation, $\complement$, on the relation R returns a new relation containing tuples that are not in R. Given two relations with pairwise disjoint domain names, the *product* operation, $\times$, is defined as a Cartesian product of their tuples. The operation $\forall_{d_i}$ is the universal quantification of variables in domain d_i. It removes tuples from the relation by universal quantification over domain d_i.

The environment `env` and the interpretation of the predicates in `result` are represented as OBDD data structures. We need to keep track of the domain names of the BDD's so the environments and predicates will be annotated with subscript $[d_1, \cdots, d_k]$ denoting a list of pairwise disjoint domain names. In the case of environments $\texttt{env}_{[x_1,\cdots,x_n]}$ the domain names represent the variables currently in the scope.

Also in the BDD algorithm we shall use a data structure called `infl`, however, it is somewhat different from that of the differential algorithm. Given a relation R, $\texttt{infl}[R]$ will determine the set of clauses that queries R; that is when R occurs in a precondition. Assuming that R has rank i and that $cl_i = \{cl_{i1}, \cdots, cl_{in_i}\}$ we define

$$\mathit{infl}(R) = \{cl_{ij} \mid R \text{ is a query in } cl_{ij}\}$$

The data structure is implemented as a map. When new tuples are added to the interpretation of the predicate R, the clauses of $\texttt{infl}[R]$ are pushed on the stack; hence they will be processed later with the updated interpretation of the predicate R.

We shall now present the parts of the algorithm that are specific for the BDD based algorithm. We begin with the case of assertion for the EXECUTE function, which is defined as follows:

$$\begin{array}{l}\textsc{execute}(R_{[d_1,\cdots,d_k]}(v_1,\cdots,v_k))\texttt{env}_{[x_1,\cdots,x_n]} = \\ \quad \textbf{for } i = 1 \text{ to } k \textbf{ do} \\ \quad\quad \texttt{env}_{[x_1,\cdots,x_n,d_1,\cdots,d_i]} \leftarrow \sigma_{v_i=d_i}(\texttt{env}_{[x_1,\cdots,x_n,d_1,\cdots,d_{i-1}]} \times U_{[d_i]})\end{array}$$

$oldR_{[d_1,\cdots,d_k]} \leftarrow \texttt{result}[R]$
$\texttt{result}[R] \leftarrow oldR_{[d_1,\cdots,d_k]} \cup \pi_{[d_1,\cdots,d_k]}(\texttt{env}_{[x_1,\cdots,x_n,d_1,\cdots,d_k]})$
if $oldR_{[d_1,\cdots,d_k]} \subsetneq \texttt{result}[R]$ **then**
    $\texttt{stack}$.PUSH($\texttt{infl}[R]$)

In the *for* loop the function incrementally builds a product of the current environment and a relation representing the universe, and simultaneously selects the tuples compatible with the arguments $(v_1, \cdots, v_k)$. Then, the resulting relation is projected to the domain names of R, and the content of R is updated with the newly derived tuples. Additionally, if the interpretation of predicate R has changed, the influence set $\texttt{infl}$ of the predicate R is pushed on the stack.

The case of universal quantification is of the following form:

EXECUTE$(\forall x : cl)\texttt{env}_{[x_1,\cdots,x_n]} =$
    EXECUTE$(cl)(\texttt{env}_{[x_1,\cdots,x_n]} \times U_{[x]})$

The function extends the current environment with a domain for the quantified variable, and then executes the clause cl.

Turning to the CHECK function, we first present the case for the query, which is as follows:

CHECK$(R_{[d_1,\cdots,d_k]}(v_1, \cdots, v_k), next)\texttt{env}_{[x_1,\cdots,x_n]} =$
    $\texttt{env'}_{[x_1,\cdots,x_n,d_1,\cdots,d_k]} \leftarrow \texttt{env}_{[x_1,\cdots,x_n]} \times \texttt{result}[R]$
    for $i = 1$ to k **do**
        $\texttt{env'}_{[x_1,\cdots,x_n,d_1,\cdots,d_k]} \leftarrow \sigma_{v_i=d_i}(\texttt{env'}_{[x_1,\cdots,x_n,d_1,\cdots,d_k]})$
    $\texttt{env'}_{[x_1,\cdots,x_n]} \leftarrow \pi_{[x_1,\cdots,x_n]}(\texttt{env'}_{[x_1,\cdots,x_n,d_1,\cdots,d_k]})$
    $next(\texttt{env'}_{[x_1,\cdots,x_n]})$

First, the function creates an auxiliary relation, which is a product of the relations representing the current environment and the predicate R. The *for* loop selects tuples that are compatible with the arguments $(v_1, \cdots, v_k)$ producing a new relation that is then projected to the domain names of $\texttt{env}_{[x_1,\cdots,x_n]}$. The resulting relation is then applied to continuation $next$.

The case of negated query is similar, except that the predicate is complemented first. The algorithm for this case is of the following form:

CHECK$(\neg R_{[d_1,\cdots,d_k]}(v_1, \cdots, v_k), next)\texttt{env}_{[x_1,\cdots,x_n]} =$
    $\texttt{env'}_{[x_1,\cdots,x_n,d_1,\cdots,d_k]} \leftarrow \texttt{env}_{[x_1,\cdots,x_n]} \times (\complement\ \texttt{result}[R])$
    for $i = 1$ to k **do**
        $\texttt{env'}_{[x_1,\cdots,x_n,d_1,\cdots,d_k]} \leftarrow \sigma_{v_i=d_i}(\texttt{env'}_{[x_1,\cdots,x_n,d_1,\cdots,d_k]})$
    $\texttt{env'}_{[x_1,\cdots,x_n]} \leftarrow \pi_{[x_1,\cdots,x_n]}(\texttt{env'}_{[x_1,\cdots,x_n,d_1,\cdots,d_k]})$
    $next(\texttt{env'}_{[x_1,\cdots,x_n]})$

The CHECK function for disjunction of preconditions is as follows:

CHECK$(pre_1 \vee pre_2, next)\texttt{env}_{[x_1,\cdots,x_n]} =$
    CHECK$(pre_1, \lambda\texttt{env}^1_{[x_1,\cdots,x_n]}.$
        CHECK$(pre_2, \lambda\texttt{env}^2_{[x_1,\cdots,x_n]}.$
            $next(\texttt{env}^1_{[x_1,\cdots,x_n]} \cup \texttt{env}^2_{[x_1,\cdots,x_n]}))\texttt{env}_{[x_1,\cdots,x_n]})\texttt{env}_{[x_1,\cdots,x_n]}$

The function first checks both preconditions pre_1 and pre_2 in the current envi-

ronment $\mathtt{env}_{[x_1,\cdots,x_n]}$. Unlike in the differential algorithm, the continuation *next* is evaluated in the union of $\mathtt{env}^1_{[x_1,\cdots,x_n]}$ and $\mathtt{env}^2_{[x_1,\cdots,x_n]}$, which were produced by calls to the procedure CHECK for preconditions pre_1 and pre_2 respectively.

In the case of existential quantification in precondition, the algorithm is as follows:

$$\begin{aligned}&\textsc{check}((\exists x : pre, next)\mathtt{env}_{[x_1,\cdots,x_n]} = \\ &\quad \textsc{check}(pre, next \circ \pi_{[x_1,\cdots,x_n]})(\mathtt{env}_{[x_1,\cdots,x_n]} \times U_{[x]})\end{aligned}$$

The function first extends the current environment, in which the precondition is checked. Furthermore, before calling the continuation *next*, the domain for the quantified variable is projected out.

The universal quantification is dealt in the following way:

$$\begin{aligned}&\textsc{check}(\forall x : pre, next)\mathtt{env}_{[x_1,\cdots,x_n]} = \\ &\quad \textsc{check}(pre, next \circ (\forall_x))(\mathtt{env}_{[x_1,\cdots,x_n]} \times U_{[x]})\end{aligned}$$

The algorithm utilizes universal quantification of variables in a given domain, denoted by $\forall_x$, which is a standard BDD operation provided by the BDD package [10]. The operation removes tuples from the given relation by universal quantification over the given domain. Hence in the case of the BDD based algorithm it is enough to extend the current environment with a quantified variable, check the precondition in the extended environment and then perform universal quantification on the returned environment.

6 Experiments

The two algorithms have been implemented in $F\#$, thereby allowing us to perform experiments for clauses with different characteristics. In this section we report on our experiments on problems arising from a modal logic representation of analysis problems. We make use of Action Computation Tree Logic (ACTL) [11] model checking to validate properties of programs written in a Pascal-like imperative language. A number of papers [18,17,16,9] have already noted the connection between iterative data-flow analysis and model checking. As an example the following ACTL formula expresses whether a variable x may have been last assigned at label l:

$$wasLastAssignedAt_{(x,l)} = \mathbf{EF}_{(a|a\neq mod_x)}(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))$$

The formulation is based on abstractions of the actions that annotate the edges of a program model. Assignment statements, $x := e$, are abstracted to actions mod_x, whereas all other statements are abstracted to actions *use*. Intuitively, the formula verifies whether there exists a path, **EF**, not modifying variable x, leading to a state labeled l which has outgoing transition, **EX**, that modifies x. The ACTL formula is translated into the ALFP formula:

$$\begin{aligned}&[\forall s : [\exists a : \exists s' : T(s,a,s') \wedge \neg mod_x(a) \wedge R_1(s')] \Rightarrow R(s)] \wedge \\ &[\forall s : [\exists a : \exists s' : T(s,a,s') \wedge R(s')] \Rightarrow R(s)]\end{aligned}$$

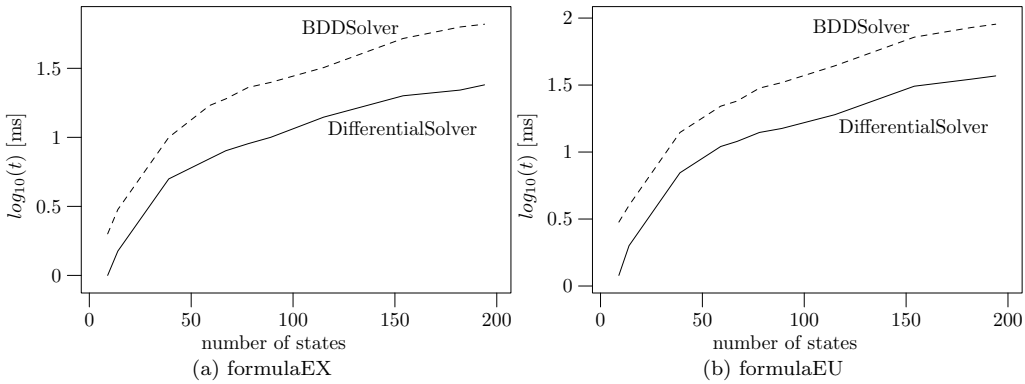

Fig. 1. Running time for modalities **EX** and **EU**.

where R_1 holds for all states satisfying $(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))$ and T is a transition relation.

The scalability of the solver algorithms is assessed by measuring the amount of time required to verify different ACTL properties as a number of program states increases. In the experiments we consider formulae presented in Appendix B, which are then translated into ALFP as shown in [12].

Figures 1 and 2 show the growth in milliseconds of time consumption from these experiments. The solving time is depicted by a dashed line for the BDD based algorithm and a solid line for the differential one.

From the experiment results for modalities **EX** and **EU** we observed that both algorithms were slightly slower for the second case. This is caused by two factors, greater length of the formula for modality **EU** and secondly the nesting of quantifiers, which in both cases is the same. Together, these factors determine the complexity of the algorithm [14]. What is more, the results show that the differential algorithm outperformed BDD implementation by a factor of two in both cases. This is caused by the fact that when resuming computations the differential algorithm only propagates the differences, whereas in the current implementation of the BDD based algorithm entire clauses are recomputed.

Turning to results for modalities **AX** and **AU** there was a large difference in performance between the differential and BDD based algorithm. The reason for this is that the ALFP formulae expressing these modalities make use of universal quantification in precondition, for which the algorithms use different solving strategies. The differential algorithm inspects all atoms from the universe and finds the extensions of the current environment that are compatible with the given precondition. The BDD based algorithm, on the other hand, utilizes universal quantification on BDD's that is computed in time $O(n^2)$, where n is the size of the BDD; hence it is much more efficient.

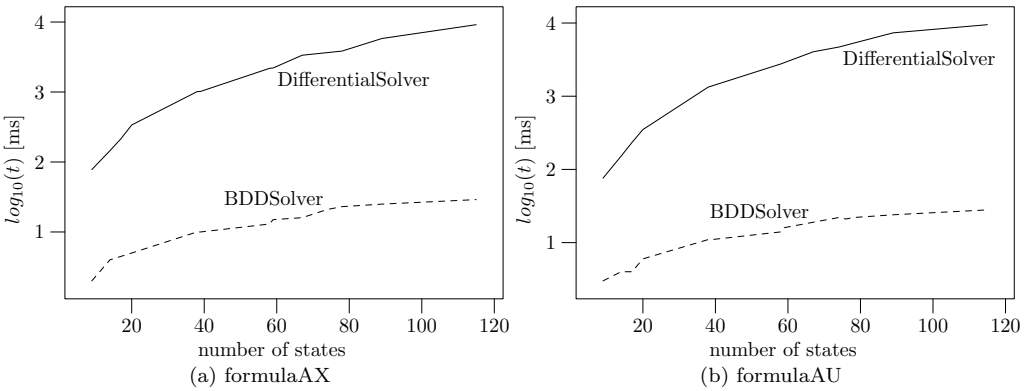

Fig. 2. Running time for modalities **AX** and **AU**.

7 Conclusion

This paper described two solver algorithms that use ALFP as the input language, one being a differential worklist algorithm based on a representation of relations as prefix trees and the other being a continuation passing style algorithm based on a BDD representation of relations. Currently no tuning of the clauses or variables is performed for any of the algorithms. Through the series of experiments we have shown how certain algorithms and data structures are better suited to certain kind of clauses than others. We have presented how, on the one hand, model checking problems benefit from the use of BDD's, and on the other, how static analysis problems take advantage of the differential solving approach. We have commented on the experiments, thus, explaining the coherence between the techniques used and the actual solving times. In future work we plan to employ heuristics for clause tuning and variable ordering, in order to improve solving times. We also hope to identify other data structures and methods that can be efficiently employed in constraint solving.

References

[1] Apt, K. R., H. A. Blair and A. Walker, *Towards a theory of declarative knowledge*, in: *Foundations of Deductive Databases and Logic Programming.*, Morgan Kaufmann, 1988 pp. 89–148.

[2] Balbin, I. and K. Ramamohanarao, *A generalization of the differential approach to recursive query evaluation*, Journal of Logic Programming **4** (1987), pp. 259–262.

[3] Berndl, M., O. Lhoták, F. Qian, L. J. Hendren and N. Umanee, *Points-to analysis using BDDs*, in: *PLDI*, 2003, pp. 103–114.

[4] Bodei, C., M. Buchholtz, P. Degano, F. Nielson and H. R. Nielson, *Static validation of security protocols*, Journal of Computer Security **13** (2005), pp. 347–390.

[5] Bryant, R. E., *Symbolic boolean manipulation with ordered binary-decision diagrams*, ACM Comput. Surv. **24** (1992), pp. 293–318.

[6] Chandra, A. K. and D. Harel, *Computable queries for relational data bases (preliminary report)*, in: *STOC*, 1979, pp. 309–318.

[7] Charlier, B. L. and P. V. Hentenryck, *A universal top-down fixpoint algorithm*, Technical report, CS-92-25, Brown University (1992).

[8] Fecht, C. and H. Seidl, *Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems*, Nordic Journal of Computing **5** (1998), pp. 304–329.

[9] Lamprecht, A.-L., T. Margaria and B. Steffen, *Data-flow analysis as model checking within the jABC*, in: *CC*, 2006, pp. 101–104.

[10] Lind-Nielsen, J., *Buddy, a binary decision diagram package* . URL http://sourceforge.net/projects/buddy/

[11] Nicola, R. D. and F. W. Vaandrager, *Action versus state based logics for transition systems*, in: *Semantics of Systems of Concurrent Processes*, 1990, pp. 407–419.

[12] Nielson, F. and H. R. Nielson, *Model checking is static analysis of modal logic*, in: *FOSSACS*, 2010, pp. 191–205.

[13] Nielson, F., H. R. Nielson, H. Sun, M. Buchholtz, R. R. Hansen, H. Pilegaard and H. Seidl, *The succinct solver suite*, in: *TACAS*, 2004, pp. 251–265.

[14] Nielson, F., H. Seidl and H. R. Nielson, *A Succinct Solver for ALFP*, Nord. J. Comput. **9** (2002), pp. 335–372.

[15] Paige, R., *Symbolic finite differencing - part i*, in: *ESOP'90*, LNCS **432** (1990), pp. 36–56.

[16] Schmidt, D. A., *Data flow analysis is model checking of abstract interpretations*, in: *POPL*, 1998, pp. 38–48.

[17] Schmidt, D. A. and B. Steffen, *Program analysis as model checking of abstract interpretations*, in: *SAS*, 1998, pp. 351–380.

[18] Steffen, B., *Data flow analysis as model checking*, in: *TACS*, 1991, pp. 346–365.

[19] Whaley, J. and M. S. Lam, *Cloning-based context-sensitive pointer alias analysis using binary decision diagrams*, in: *PLDI*, 2004, pp. 131–144.

A Semantics of ALFP

We write $\varsigma[x \mapsto a]$ for the mapping that is as ς except that x is mapped to a. The semantics of preconditions, clauses and clause sequences is given by:

$$
\begin{array}{ll}
(\varrho,\varsigma) \models R(v_1,\ldots,v_k) & \underline{\texttt{iff}}\ (\varsigma(v_1),\ldots,\varsigma(v_k)) \in \varrho(R) \\
(\varrho,\varsigma) \models \neg R(v_1,\ldots,v_k) & \underline{\texttt{iff}}\ (\varsigma(v_1),\ldots,\varsigma(v_k)) \notin \varrho(R) \\
(\varrho,\varsigma) \models pre_1 \wedge pre_2 & \underline{\texttt{iff}}\ (\varrho,\varsigma) \models pre_1 \text{ and } (\varrho,\varsigma) \models pre_2 \\
(\varrho,\varsigma) \models pre_1 \vee pre_2 & \underline{\texttt{iff}}\ (\varrho,\varsigma) \models pre_1 \text{ or } (\varrho,\varsigma) \models pre_2 \\
(\varrho,\varsigma) \models \exists x : pre & \underline{\texttt{iff}}\ (\varrho,\varsigma[x \mapsto a]) \models pre \text{ for some } a \in \mathcal{U} \\
(\varrho,\varsigma) \models \forall x : pre & \underline{\texttt{iff}}\ (\varrho,\varsigma[x \mapsto a]) \models pre \text{ for all } a \in \mathcal{U}
\end{array}
$$

$$
\begin{array}{ll}
(\varrho,\varsigma) \models R(v_1,\ldots,v_k) & \underline{\texttt{iff}}\ (\varsigma(v_1),\ldots,\varsigma(v_k)) \in \varrho(R) \\
(\varrho,\varsigma) \models \mathbf{1} & \underline{\texttt{iff}}\ \texttt{true} \\
(\varrho,\varsigma) \models cl_1 \wedge cl_2 & \underline{\texttt{iff}}\ (\varrho,\varsigma) \models cl_1 \text{ and } (\varrho,\varsigma) \models cl_2 \\
(\varrho,\varsigma) \models pre \Rightarrow cl & \underline{\texttt{iff}}\ (\varrho,\varsigma) \models cl \text{ whenever } (\varrho,\varsigma) \models pre \\
(\varrho,\varsigma) \models \forall x : cl & \underline{\texttt{iff}}\ (\varrho,\varsigma[x \mapsto a]) \models cl \text{ for all } a \in \mathcal{U}
\end{array}
$$

$$
(\varrho,\varsigma) \models cl_1,\cdots,cl_s \quad \underline{\texttt{iff}}\ (\varrho,\varsigma) \models cl_i \text{ for all } i, 1 \leq i \leq s
$$

B Expermiments input formulae

We present formulae used in experiments together with their translations to ALFP. For ease of presentation we introduce auxiliary formula $\phi = l \wedge \mathbf{EX}_{mod_x}(\mathsf{true})$, which is translated into following ALFP clauses defining relation $R_{(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))}$.

$$[\forall s : R_{\mathsf{true}}(s)] \wedge$$
$$[\forall s : [\exists a : \exists s' : T(s,a,s') \wedge mod_x(a) \wedge R_{\mathsf{true}}(s')] \Rightarrow R_{(\mathbf{EX}_{mod_x}(\mathsf{true}))}(s)] \wedge$$
$$[\forall s : R_l(s) \wedge R_{(\mathbf{EX}_{mod_x}(\mathsf{true}))}(s) \Rightarrow R_{(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))}(s)] \wedge$$

The relation is then used in the input clauses generated from ACTL formulae used in the experiments. The ACTL formulae along with corresponding ALFP clauses are given below.

$$\text{formulaEX} = \mathbf{EX}_{(a|a \neq mod_x)}(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))$$
$$[\forall s : [\exists a : \exists s' : T(s,a,s') \wedge mod_x(a) \wedge R_{(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))}(s')] \Rightarrow R_{(\mathbf{EX}_{mod_x}(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true})))}(s)]$$

$$\text{formulaAX} = \mathbf{AX}_{(a|a \neq mod_x)}(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))$$
$$[\forall s : [\forall a : \forall s' : \neg T(s,a,s') \vee (notmod_x(a) \wedge R_{\mathsf{true}}(s'))] \wedge$$
$$[\exists a : \exists s' : T(s,a,s')] \Rightarrow R_{(\mathbf{AX}_{notmod_x}(\mathsf{true}))}(s)]$$

$$\text{formulaEU} = \mathbf{E}[\mathsf{true}\ {}_{\mathcal{A}}\mathbf{U}_{\{a|a \neq mod_x\}}\ (l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))]$$
$$[\forall s : [\exists a : \exists s' : T(s,a,s') \wedge notmod_x(a) \wedge R_{(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))}(s')] \Rightarrow$$
$$R_{(\mathbf{E}[\mathsf{true}_A \mathbf{U}_{notmod_x}(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))])}(s)] \wedge$$
$$[\forall s : [\exists a : \exists s' : T(s,a,s') \wedge A(a) \wedge R_{\mathsf{true}}(s') \wedge R_{(\mathbf{E}[\mathsf{true}_A \mathbf{U}_{notmod_x}(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))])}(s')] \Rightarrow$$
$$R_{(\mathbf{E}[\mathsf{true}_A \mathbf{U}_{notmod_x}(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))])}(s)]$$

$$\text{formulaAU} = \mathbf{A}[\mathsf{true}\ {}_{\mathcal{A}}\mathbf{U}_{\{a|a \neq mod_x\}}\ (l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))]$$
$$[\forall s : [[\exists a : \exists s' : T(s,a,s')] \wedge$$
$$[\forall a : \forall s' : \neg T(s,a,s') \vee [notmod_x(a) \wedge R_{(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))}(s')] \vee$$
$$[A(a) \wedge R_{\mathsf{true}}(s') \wedge R_{(\mathbf{A}[\mathsf{true}_A \mathbf{U}_{notmod_x}(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))])}(s')]] \Rightarrow$$
$$R_{(\mathbf{A}[\mathsf{true}_A \mathbf{U}_{notmod_x}(l \wedge \mathbf{EX}_{mod_x}(\mathsf{true}))])}(s)]$$