Symbolic Resource Bound Inference for Functional Programs
Technical Report EPFL-REPORT-190578

Ravichandhran Madhavan[1] and Viktor Kuncak[2]

[1] ravi.kandhadai@epfl.ch
[2] viktor.kuncak@epfl.ch
EPFL

Abstract. We present an approach for inferring symbolic resource bounds for purely functional programs consisting of recursive functions, algebraic data types and nonlinear arithmetic operations. In our approach, the developer specifies the desired shape of the bound as a program expression containing numerical holes which we refer to as *templates*. For e.g, $\mathsf{time} \leq a * \mathsf{height}(\mathsf{tree}) + b$ where a, b are unknowns, is a template that specifies a bound on the execution time. We present a scalable algorithm for computing tight bounds for sequential and parallel execution times by solving for the unknowns in the template. We empirically evaluate our approach on several benchmarks that manipulate complex data structures such as binomial heap, lefitist heap, red-black tree and AVL tree. Our implementation is able to infer hard, nonlinear symbolic time bounds for our benchmarks that are beyond the capability of the existing approaches.

1 Introduction

This paper presents a new algorithm and a publicly available tool for inferring resource bounds of functional programs.[3] We focus on functional languages because they eliminate by construction low-level memory errors and allow the developer to focus on functional correctness and performance properties. Our tool is designed to automate reasoning about such high-level properties. We expect this research direction to be relevant both for improving the reliability of functional programming infrastructure used in many enterprises (e.g. LinkedIn, Twitter, several banks), as well as for reasoning about software and hardware systems within interactive theorem provers [17], [21], [29], [12], [19], which often model stateful and distributed systems using functional descriptions.

The analysis we present in this paper aims to discover invariants (e.g. function postconditions) that establish program correctness as well as bounds on parallel and sequential program execution time. Such invariants often contain invocations of user-defined recursive functions specific to the program being verified, such as size or height functions on a tree structure. We therefore need a verification

[3] To download the tool please see `http://lara.epfl.ch/w/software`

technique that can prove invariants that are expressed in terms of user-defined functions. To the best of our knowledge, our tool is the first available system that can establish such complex resource bounds with this degree of automation.

Our tool can show, for example, that a function converting a propositional formula into negation-normal form takes no more than $44 \cdot \mathsf{size}(f) - 20$ operations, where $\mathsf{size}(f)$ is the number of nodes in the formula f. The tool also proves that the depth of the computation graph (time in an infinitely parallel implementation) is bounded by $5 \cdot h(f) - 2$, where $h(f) \geq 1$ is the height of the formula tree. As another example, it shows that deleting from an AVL tree requires the number of operations given by $145 \cdot h(t) + 19$, where $h(t) \geq 0$ is the height of the tree t, whereas the depth of the computation graph is $51 \cdot h(t) + 4$.

Our tool takes as input the program, as well as the desired shapes of invariants, which we call *templates*. The goal of the analysis becomes finding coefficients in the templates. The coefficients in practice tend to be sufficiently large that simply trying out small values does not scale. We therefore turn to one of the most useful techniques for finding unknown coefficients in invariants: Farkas' lemma. This method converts a $\exists\forall$ problem on linear constraints into a purely existential problem over non-linear constraints.

The challenge that we address is developing a practical technique that makes such expensive non-linear reasoning work on programs and templates that contain invocations of user-defined recursive functions, that use algebraic data types (such as trees and lists), and that have complex control flow with many disjunctions.

We present a publicly available tool that handles these difficulties through an incremental and counterexample-driven algorithm that soundly encodes algebraic data types and recursive functions and that fully leverages the ability of an SMT solver to handle disjunctions efficiently. We show that our technique is effective for the problem of discovering highly application-specific inductive resource bounds in functional programs.

2 Background and Enabling Techniques

We first present key existing technology on which our tool builds.

2.1 Instrumenting Programs to Track Resource Bounds

Our approach decouples the semantics of resources such as execution time from their static analysis. We start with the exact instrumentation of programs with resource bounds, without approximating e.g. conditionals or recursive invocations. To illustrate our approach, consider a simple Scala [22] program shown in Fig. 1, which appends a list l2 to the reverse of l1. We use this program as our running example. The recursive function size counts the length of its list argument; it is user-defined and omitted for brevity.

Fig. 2 illustrates the instrumentation for tracking execution time on this example. For every expression e in the program the resource consumed by e is

```
def revRec(l1:List, l2:List) : List =
(l1 match {
  case Nil() ⇒ l2
  case Cons(x,xs) ⇒
    revRec(xs, Cons(x, l2))
})
ensuring(res ⇒ time ≤ a∗size(l1) + b))
```

Fig. 1. Appending $l2$ to the reverse of $l1$

```
def revRec(l1:List,l2:List):(List,Int) =
  (l1 match {
   case Nil() ⇒ (l2, 1)
   case Cons(x,xs) ⇒
     val (e, t) = revRec(xs, Cons(x,l2))
     (e, 5 + t) })
ensuring(res ⇒ res._2 ≤ a∗size(l1) + b))
```

Fig. 2. After time instrumentation

computed as a function of the resources consumed by its sub-expressions. For instance, the *execution time* of an expression (such as $e_1 * e_2$) is *the sum* of the execution times of its arguments (e_1 and e_2) plus the time taken by the operation (here, $*$) performed by the expression (in this case, 1). We expose the resource usage of a procedure to its callers by augmenting the return value of the procedure with its resource usage. The resource consumption of a function call is determined as the sum of the resources consumed by the called function (which is exposed through its augmented return value) plus the cost of invoking the function. The cost of primitive operations, such as +, variable access, etc., are parametrized by a cost model which is, by default, 1 for all primitive operations.

Another resource that we consider in this paper is *depth*, which is a measure of parallelism in an expression. *Depth* [6] is the longest chain of dependencies between the operations of an expression. The *depth* and *work* (the sequential execution time) of programs have been used by the previous works to accurately estimate the parallel running times on a given parallel system [6]. Fig. 3 and Fig. 4 illustrate the instrumentation our tool perform to compute the *depth* of a procedure that traverses a tree. We compute the *depth* of an expression similarly to its execution time, but instead of adding the resource usages of the sub-expressions, we compute their maximum.

```
def traverse(t: Tree) = (t match {
  case Leaf() ⇒ f(t)
  case Node(l,v,r) ⇒
    traverse(l) + traverse(r) + f(t)
)
ensuring(res ⇒ depth ≤ a∗height(t) + b)
```

Fig. 3. A tree traversal procedure

```
def traverse(t: Tree):(Tree,Int)= (t match{
 case Leaf() ⇒ f(t)
 case Node(l,v,r) ⇒
   val (el, dl) = traverse(l)
   val (er, dr) = traverse(r)
   val (e, d) = f(t)
   (el+er+e, max(max(dl,dr)+1,d)+5)) })
ensuring(res ⇒ res._2 ≤ a∗height(t) + b)
```

Fig. 4. After *depth* instrumentation

Every inductive invariant for the instrumented procedure obtained by solving for the unknowns a, b is a valid bound for the resource consumed by the original procedure. Moreover, the strongest invariant is also the strongest bound on the resource. Notice that the instrumentation increases the program sizes, introduces tuples and, in the case of *depth* instrumentation, creates numerous max operations.

2.2 Solving Numerical Parametric Formulas

Our approach requires deciding validity of formulas of the form $\exists \mathbf{a}.\forall \mathbf{x}.\neg\phi$, where $\mathbf{a}$ is a vector of variables. The formulas have a single quantifier alternation. We thus need to find values for $\mathbf{a}$ that will make ϕ unsatisfiable. We refer to ϕ as a *parametric formula* whose parameters are the variables $\mathbf{a}$. When the formula ϕ consists only of *linear inequalities*, finding values for the parameters $\mathbf{a}$ can be converted to that of satisfying a quantifier-free nonlinear constraint (Farkas' constraint) using a known reduction, sketched below.

A conjunction of linear inequalities is unsatisfiable if one can derive a contradiction $1 \leq 0$ by multiplying the inequalities by non-negative values, subtracting the smaller terms by non-negative values and adding the coefficients in the inequalities. E.g, $ax+by+c \leq 0 \wedge x-1 \leq 0$ is unsatisfiable if there exist non-negative real numbers $\lambda_0, \lambda_1, \lambda_2$ such that $\lambda_1 \cdot (ax+by+c) + \lambda_2 \cdot (x-1) - \lambda_0 \leq 0$ reduces to $1 \leq 0$. Hence, the coefficients of x and y should become 0 and the constant term should become 1. This yields a nonlinear constraint $\lambda_1 a + \lambda_2 = 0 \wedge \lambda_1 b = 0 \wedge \lambda_1 c - \lambda_2 - \lambda_0 = 1 \wedge \lambda_0 \geq 0 \wedge \lambda_1 \geq 0 \wedge \lambda_2 \geq 0$. The values of a and b in every model for this nonlinear constraint will make the inequalities unsatisfiable.

This approach has been used by previous works [7,9,15] to infer linear invariants for numerical programs. There are two important points to note about this approach: (a) In the presence of real valued variables, handling strict inequalities in the parametric formula requires an extension based on *Motzkin's transposition theorem* as discussed in [24]. (b) This approach is complete for linear real formulas by Farkas' Lemma, but not for linear integer formulas. However, the incompleteness did not manifest in any of our experiments. Similar observation has also been documented in the previous works such as [15].

2.3 Successive Function Approximation by Unfolding

To construct verification conditions (VCs) in the presence of algebraic data-types (ADTs) and recursive functions we use the approach employed in the Leon verifier [5, 28]. The approach constructs VCs incrementally wherein each increment makes the VC more precise by unrolling the function calls that have not been unrolled in the earlier increments (referred to as VC refinement). The functions in the VCs at any given step are treated as uninterpreted functions. Hence, every VC created is a sufficient but not necessary condition for the postcondition to be inductive. The postcondition is inductive if any of the generated VCs are valid. The refinement of VCs continues forever until the postcondition is proven. In our implementation, we enforce termination by bounding the number of times a recursive function call is unrolled (fixed as 2 in our experiments).

We explain the VC generation and refinement on the revRec function shown in Fig. 2. The initial VC that we create for revRec is shown below

$$\forall l1, l2, res, x, xs, e, t, r, f1, f2, \mathsf{size}, \mathsf{revRec}.\ \neg\phi$$
$$\phi \equiv ((l1 = Nil() \wedge res = (l2, 1)) \vee (l1 = Cons(x, xs) \wedge res = (e, 5 + t) \wedge (e, t) = \mathsf{revRec}(xs, Cons(x, l2))) \wedge f2 > ar + b \wedge r = \mathsf{size}(l1) \wedge res = (f1, f2) \qquad (1)$$

The function symbols in the VC are universally quantified as they are treated as uninterpreted functions. The combined algorithm presented in the next section solves for the parameters a, b so that the VC holds for any definition of size and revRec. If the formula (1) has no solution, it then refines the VC by unrolling the calls to size and revRec. For instance, unrolling $r = \mathsf{size}(l1)$ in the above formula will conjoin the predicate with the formula $(l1 = Nil() \wedge r = 0) \vee (l1 = Cons(x1, xs1) \wedge r = 1 + r2 \wedge r2 = \mathsf{size}(xs1))$ that corresponds to the body of size. The subsequent refinements will unroll the call $r2 = \mathsf{size}(xs1)$ and so on. Note that, whereas unfolding is the key mechanism in Leon [5, 28], here it is used in a new combination, with the inference of numerical parameters.

3 Invariant Inference Algorithm

We next present core techniques of our algorithm for inferring resource bounds. The algorithm introduces new techniques and combines the existing techniques to overcome their individual weaknesses.

3.1 Solving Formulas with Algebraic Data Types and Recursion

We first describe our approach for solving parametric formulas that are similar to constraint (1) with ADTs, uninterpreted functions, linear and nonlinear arithmetic operations.

Eliminating Uninterpreted Functions and ADT Constructors from Parametric Disjuncts. Let d be a parametric formula with parameters param defined over a set of variables X and uninterpreted function symbols X_f. We reduce this to a formula d' that does not have any uninterpreted functions and ADT constructors using the axioms of uninterpreted functions and ADTs as described below. We convert d to negation normal form and normalize the resulting formula so that every atomic predicate (atom) referring to uninterpreted functions or ADTs is of the form $r = f(v_1, v_2, \ldots, v_n)$ or $r = cons(v_1, v_2, \ldots, v_n)$ where f is a function symbol, *cons* is the constructor of an ADT and $r, v_1, \ldots, v_n$ are variables. We refer to this process as *purification.* Let F and T be the set of function atoms and ADT atoms in the purified formula.

$$\begin{aligned}
&\text{let } \delta_1 = \bigwedge\{(\bigwedge_{i=1}^{n} v_i = u_i) \Rightarrow (r = r') \mid r = f(v_1, \ldots, v_n),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad r' = f(u_1, \ldots, u_n) \in F\}\\
&\text{let } \delta_2 = \bigwedge\{(\bigwedge_{i=1}^{n} v_i = u_i) \Leftrightarrow (r = r') \mid r = cons(v_1, \ldots, v_n),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad r' = cons(u_1, \ldots, u_n) \in T\}\\
&\text{let } \delta = (purify(d) \setminus (F \cup T)) \wedge \delta_1 \wedge \delta_2
\end{aligned}$$

where $\delta \setminus (F \cup T)$ is a formula obtained by substituting with true every atomic predicate in F or T. Notice that the above elimination procedure uses only the fact that the ADT constructors are *injective*. Due to this the completeness of our approach may not be immediately obvious. In section 3.2 we formalize the completeness property of our approach.

Applying the above reduction to the disjunct d_{ex} of Constraint (1) along which $l = \mathsf{Nil()}$, results in a constraint of the form sketched below. We consider tuples also as ADTs.

$$purify(d_{ex}) = \begin{cases} (l1 = Nil() \wedge res = (l2, 1) \wedge f2 > ar + b \\ \wedge r = \mathsf{size}(n1) \wedge res = (f1, f2) \end{cases}$$

$$\delta_{ex} = (f2 > ar + b \wedge ((l2 = f1 \wedge f2 = 1) \Leftrightarrow res = res)) \qquad (2)$$

The formula δ obtained by eliminating uninterpreted function symbols and ADTs typically has several disjunctions. In fact, if there are n function symbols and ADT constructors in d then d' could potentially have $O(n^2)$ disjunctions and $O(2^{n^2})$ disjuncts. Our approach described in the next subsection solves the parametric formulas incrementally based on counter-examples.

3.2 Incrementally Solving Parametric Formulas

Figure 5 presents our algorithm for solving an alternating satisfiability problem. Given a parametric formula, the goal is to find an assignment ι for params such that replacing params according to ι results in unsatisfiable formula. We explain our algorithm using the example presented in the earlier section. Consider the VC given by constraint (1). Initially, we start with some arbitrary assignment ι for the parameters a and b (line 5 of the algorithm). Say $\iota(a) = \iota(b) = 0$ initially. Next, we instantiate (1) by replacing a and b by 0 (line 8), which results in the non-parametric constraint: $\phi_{ex} : ((l1 = Nil() \wedge res = (l2, 1)) \vee (l1 = Cons(x, xs) \wedge res = (e, 5 + t) \wedge (e, t) = \mathsf{revRec}(xs, Cons(x, l2)) \wedge f2 > 0 \wedge r = \mathsf{size}(l1) \wedge res = (f1, f2)$.

If the constraint becomes unsatisfiable because of the instantiation then we have found a solution. Otherwise, we construct a model σ for the instantiated formula as shown in line 11. For the constraint ϕ_{ex} shown above, $l1 \mapsto Nil(), l2 \mapsto Nil(), res \mapsto (Nil(), 1), r \mapsto -1$ and $\mathsf{size} \mapsto \lambda x.(x = Nil() \to -1 \mid 0)$ is a model. In the next step, we combine the models ι and σ and construct σ'. Note that ι is an assignment for parameters and σ is an assignment for universally quantified variables. Using the model σ' we choose a disjunct of the parametric formula (1) that is satisfied by σ'. For our example, the disjunct chosen will be $d_{ex} : l1 = Nil() \wedge res = (l2, 1) \wedge f2 > ar + b \wedge r = \mathsf{size}(l1) \wedge res = (f1, f2)$. This operation of choosing a disjunct satisfying a given model can be performed efficiently in time linear in the size of the formula without explicitly constructing a disjunctive normal form.

The function $\mathsf{elimFunctions}$ invoked at line 14 eliminates the function symbols and ADT constructors from the disjunct d using the approach described in section 3.1. Applying $\mathsf{elimFunctions}$ on d_{ex} results in the formula δ_{ex} given by (2). We

```
1  input : A parametric linear formula $\phi$ with parameters 'params'
2  output : Assignments for params such that $\phi(params)$ is unsatisfiable
3  or $\emptyset$ if no such assignment exists
4  def solveUNSAT(params, $\phi$) {
5    construct an arbitrary initial mapping $\iota$ : params $\mapsto \mathbb{R}$
6    var C = true
7    while(true) {
8     let $\phi_{inst}$ be obtained from $\phi$ by replacing every $t \in$ params by $\iota(t)$
9     if ($\phi_{inst}$ is unsatisfiable) return $\iota$
10    else {
11     choose $\sigma$ such that $\sigma \models \phi_{inst}$
12     let $\sigma'$ be $\iota \uplus \sigma$
13     choose a disjunct $d$ of $\phi$ such that $\sigma' \models d$
14     let $\delta$ be elimFunctions($d$)
15     choose a disjunct $d'$ of $\delta$ such that $\sigma' \models d'$
16     let $d_{num}$ be elim($d'$)
17     let $C_d$ be unsatConstraints($d_{num}$)
18     $C = C \wedge C_d$
19     if ($C$ is unsatisfiable) return $\emptyset$
20     else {
21      choose $m$ such that $m \models C$
22      let $\iota$ be the projection of $m$ onto params }}}}
```

Fig. 5. A procedure for finding parameters for a formula to make it unsatisfiable. unsatConstraints generates nonlinear constraints for unsatisfiability of a disjunct as illustrated in section 2.2.

choose a disjunct d' of δ that satisfies the model σ'. For our example, the disjunct of δ_{ex} that will be chosen is $d'_{ex} : l2 = f1 \wedge f2 = 1 \wedge res = res \wedge f2 > ar + b$.

Eliminating non-numerical predicates from a disjunct (elim). We now describe the operation elim at line 16. Let d' be the parametric disjunct chosen in the previous step. d' is a conjunction of atomic predicates (atoms). Let d_t denote the atoms that consist of variables of ADT type or boolean type. Let d_n denote the atoms that do not contain any parameters and only contain variables of numerical type. Let d_p denote the remaining atoms that has parameters and numerical variables.

For the example disjunct d'_{ex}, d_t is $l2 = f1$, d_n is $f2 = 1$ and d_p is $f2 > ar+b$. The disjunct d_t can be dropped as d_t cannot be falsified by any instantiation of the parameters. This is because d_p and d_t will have no common variables. The remaining disjunct $d_n \wedge d_p$ is completely numerical. However, we simplify $d_n \wedge d_p$ further as explained below. We construct a simplified formula d'_n by eliminating variables in d_n that do not appear in d_p by applying the quantifier elimination rules of Presburger arithmetic on d_n [23]. In particular, we apply the one-point rule that uses equalities to eliminate variables and the rule that eliminates relations over variables for which only upper or lower bounds exist. $d_n \wedge d_p$ is unsatisfiable *iff* $d'_n \wedge d_p$ is unsatisfiable.

Typically, d_n has several variables that do not appear in d_p. This elimination helps reduce the sizes of the disjuncts and in turn the sizes of the nonlinear constraints generated from the disjunct. Our experiments indicate that the sizes of the disjuncts are reduced by 70% or more.

We construct nonlinear Farkas' constraints (line 17) for falsifying the disjunct d_{num}, obtained after elimination phase, as described in section 2.2. We conjoin the nonlinear constraint with previously generated constraints, if any (lines 17,18). A satisfying assignment to the new constraint will falsify every disjunct explored thus far. We consider the satisfying assignment as the next candidate model ι for the parameters and repeat the above process.

If the nonlinear constraint C is unsatisfiable at any given step then we conclude that there exists no solution that would make ϕ unsatisfiable. In this case, we refine the VC by unrolling the functions calls as explained in section 2.3 and reapply the algorithm solveUNSAT on the refined VC.

Correctness, Completeness and Termination of solveUNSAT.

Let $\mathcal{F}$ denote parametric linear formulas belonging to the theory of real arithmetic, uninterpreted functions and ADTs, in which parameters are real valued and appear only as coefficients of variables.

Theorem 1. *Let $\phi \in \mathcal{F}$ be a linear parametric formula with parameters params.*

1. *The procedure solveUNSAT is correct for $\mathcal{F}$. That is, if $\iota \neq \emptyset$ then ι is an assignment for parameters that will make ϕ unsatisfiable.*
2. *The procedure solveUNSAT is complete for $\mathcal{F}$. That is, if $\iota = \emptyset$ then there does not exist an assignment for params that will make ϕ unsatisfiable.*
3. *The procedure solveUNSAT terminates.*

The correctness of procedure solveUNSAT is obvious as the procedure returns a model ι iff ι makes the formula ϕ unsatisfiable. The algorithm terminates since, in every iteration of the solveUNSAT algorithm, at least one satisfiable disjunct of elimFunctions(d) is made unsatisfiable, where d is a disjunct of ϕ. The number of disjuncts that can be falsified by the solveUNSAT procedure is bounded by $O(2^{n^2})$, where n is the number of atoms in ϕ. Note that, in practice, our tool explores a very small fraction of the disjuncts (see section 4). The proof of completeness of the procedure is detailed in [20]. An important property that ensures completeness is that the operation elimFunctions is applied only on a satisfiable disjunct d. This guarantees that the predicates in d involving ADT variables do not have any inconsistencies. Since the parameters can only influence the values of numerical variables, axioms that check for inconsistencies among the ADT predicates can be omitted.

Theorem 1 implies that the procedure we described in the previous sections for solving parametric VCs, in the presence of recursive functions, ADTs and arithmetic operations, that iteratively unrolls the recursive functions in the VC and applies the solveUNSAT procedure in each iteration is complete when the recursive functions are *sufficiently surjective* [27, 28] and when the arithmetic operations in the VCs are parametric linear operations over reals.

3.3 Solving Nonlinear Parametric Formulas

Nonlinearity is common in resource bounds. In this section, we discuss our approach for handling nonlinear parametric formulas like $\phi_{ex} : wz < xy \wedge x < w - 1 \wedge y < z - 1 \wedge ax + b \leq 0 \wedge ay + b \leq 0$ where a, b are parameters. Our approach is based on axiomatizing the nonlinearity operations. We handle multiplication by using axioms such as $\forall x, y.\ xy = (x-1)y+y$, $\forall x, y.\ xy = x(y-1)+x$ and monotonicity properties like $(x \geq 0 \wedge y \geq 0 \wedge w \geq x \wedge z \geq y) \Rightarrow xy \leq wz$. Similarly, we axiomatize exponential functions of the form C^x, where C is a constant. For example, we use the axiom $\forall x.\ 2^x = 2 \cdot 2^{x-1}$ together with the monotonicity axiom for modelling 2^x. The axioms are incorporated into the verification conditions by recursive instantiation as explained below.

Axioms such as $xy = (x-1)y+y$ that are recursively defined are instantiated similar to unrolling a recursive function during VC refinements. For example, in each VC refinement, for every atomic predicate $r = xy$ that occurs in the VC, we add a new predicate $r = (x-1)y + y$ if it does not exist. We instantiate a binary axiom, such as monotonicity, on every pair of terms in the VC on which it is applicable. For instance, if $r = f(x)$, $r' = f(x')$ are two atoms in the VC and if f has a monotonicity axiom, then we conjoin the predicate $(x \leq x' \Rightarrow r \leq r') \wedge (x' \leq x \Rightarrow r' \leq r)$ to the VC. This approach can be extended to N-ary axioms. If the axioms define a *Local Theory Extension* [16] (like monotonicity) then the instantiation described above is complete.

Consider the example formula ϕ_{ex} shown above. Instantiating the multiplication axioms a few times will produce the following formula (simplified for brevity): $wz < xy \wedge xy = (x-1)(y-1) + x + y - 1 \wedge ((x \geq 0 \wedge y \geq 0 \wedge x \leq w \wedge y \leq z) \rightarrow xy \leq wz) \wedge x < w - 1 \wedge y < z - 1 \wedge ax + b \leq 0 \wedge ay + b \leq 0$. This formula can be solved without interpreting multiplication. $a = -1, b = 0$ is a solution for the parameters.

3.4 Finding Strongest Bounds

For computing strongest bounds, we assume that every parameter in the template appears as a coefficient of some expression. We approximate the rate of growth of an expression in the template by counting the number of function invocations (including nonlinear operations) performed by the expression. We order the parameters in the descending order of the estimated rate of growth of the associated expression, breaking ties arbitrarily. Let this order be $\sqsubseteq$. For instance, given a template **res**≤a∗f(g(x,f(y)))+c∗g(x)+a∗x +b, we order the parameters as $a \sqsubseteq c \sqsubseteq b$. We define an order $\leq^*$ on $Params \mapsto \mathbb{R}$ by extending $\leq$ lexicographically with respect to the ordering $\sqsubseteq$. We find a *locally* minimum solution ι_{min} for the parameters with respect to $\leq^*$ as explained below.

Let ι be the solution found by the solveUNSAT procedure. ι is obtained by solving a set of nonlinear constraints C. We compute a minimum satisfying assignment ι_{min} for C with respect to the total order $\leq^*$ by performing a *binary search* on the solution space of C starting with the initial upper bound given by ι. We stop the binary search when, for each parameter p, the difference between the values of p in the upper and lower bounds we found is ≤ 1. We need to bound the difference between the upper and lower bounds since the parameters in our case are reals. ι_{min} may not falsify ϕ although ι does. This is because C only encodes the constraints for falsifying the disjuncts of ϕ explored until some iteration. We use ι_{min} as the next candidate model and continue the iterations of the solveUNSAT algorithm.

In general, the inferred bounds are not guaranteed to be the strongest as the verification conditions we generate are sufficient but not necessary conditions. However, it would be the strongest solution if the functions in the program are *sufficiently surjective* [27, 28], if there are no nonlinear operations and there is no loss of completeness due to applying Farkas' Lemma on integer formulas. Our system also supports finding a concrete counter-example, if one exists, for the values smaller than those that are inferred.

3.5 Inference of Auxiliary Templates

We implemented a simple strategy for inferring invariant templates automatically for some functions. For every function f for which a template has not been provided, we assume a default template that is a linear combination of integer valued arguments and return values of f. For instance, for a function size(l) we assume a template a***res**+b≤0 (where, **res** is the return value of size). This enables us to infer and use correctness invariants like size(l)≥0 automatically.

3.6 Analysis Strategies

Inter-procedural analysis. We solve the resource bound templates for the functions modularly in a bottom-up fashion. We solve the resource bound templates of the callees independent of the callers, minimize the solution to find strong bounds and use the bounds while analysing the callers. The auxiliary templates that we infer automatically are solved in the context of the callers in order to find context-specific invariants.

Targeted unrolling. Recall that we unroll the functions in a VC if the VC is not solvable by solveUNSAT (i.e, when the condition at line 19 is true). As an optimization we make the unrolling process more demand-driven by unrolling only those functions encountered in the disjuncts explored by the solveUNSAT procedure. This avoids unrolling of functions along disjuncts that are already unsatisfiable in the VC.

Prioritizing Disjunct Exploration. Typically, the VCs we generate have a large number of disjuncts some of which are easier to reduce to false compared to others. We bias the implementation to pick the easier disjuncts by using timeouts on the nonlinear constraints solving process. Whenever we timeout while solving a nonlinear constraint, we block the disjunct that produced the nonlinear constraint in the VC so that it is not chosen again. In our experiments, we used a timeout of 20s. This strategy, though conceptually simple, made the analysis converge faster on many benchmarks.

4 Empirical Evaluation

We have implemented our algorithm on top of the Leon verifier for Scala [5], building on the release from the GitHub repository. We evaluate our tool on a set of benchmarks shown in Fig. 6 written in a purely functional subset of Scala programming language. The experiments were performed on a machine with 8 core, 3.5 GHz, intel i7 processor, having 16GB RAM, running Ubuntu operating system. For solving the SMT constraints generated by tool we use the Z3 solver of [10], version 4.3. The Benchmarks used in the evaluation comprises of approximately 1.5K lines of functional Scala code with 130 functions and 80 templates. All templates for execution bounds specified in the benchmarks were precise bounds. Fig. 6 shows the lines of codes *loc*, number of procedures *P* and a sample template for running time bound that was specified, for the benchmarks.

Benchmark	loc	P	Sample template used in benchmark
List Operations (*list*)	60	8	a*(size(l)*size(l))+b
Binary search tree (*bst*)	91	8	
addAll			a*(lsize(l)*(height(t)+lsize(l)))+b*lsize(l)+c
removeAll			a*(lsize(l)*height(t))+b*lsize(l)+c
Doubly ended queue (*deq*)	86	14	a*qsize(q)+b
Prop. logic transforms (*prop*)	63	5	a*size(formula)+b
Binary Trie (*trie*)	119	6	a*inpsize(inp)+c
qsort, isort, mergesort (*sort*)	123	12	a*(size(l)*size(l))+b
Loop transformations (*loop*)	102	10	a*size(program)+b
Concatenate variations (*cvar*)	40	5	
strategy 1			a*((n*m)*m)+c*(n*m)+d*n+e*m+f
strategy 2			a*(n*m)+b*n+c*m+d
Leftist heap (*lheap*)	81	10	
merge			a*rheight(h1)+b*rheight(h2)+c
removeMax			a*leftRightheight(h) + b
Redblack tree (*rbt*)	109	11	a*blackheight(t)+b
AVL tree (*avl*)	190	15	a*height(t)+b
Binomial heap (*bheap*)	204	12	
merge			a*treenum(h1)+b*treenum(h2)+c
deleteMin			a*treenum(h1)+b*minchildren(h2)+c
Speed benchmarks(*speed*)	107	8	a*((k+1)*(len(sb1)+len(sb2)))+b*size(str1)+c
Fold operations (*fold*)	88	7	
listfold, *treefold*			a*(k*k)+b, a*size(t)+b

Fig. 6. Benchmarks used in the evaluation comprising of approx. 1.5K lines of scala code, 130 functions and 80 templates. P denotes the number of procedures.

The benchmark *list* implements a set of list manipulation operations like *append*, *reverse*, *remove*, *find* and *distinct*–that removes duplicates. *bst* implements a binary search tree with operations like *insert*, *remove*, *find*, *addall* and *removeall*. The function lsize(l) (used in the templates) is the size of the list of elements to be inserted/removed from the tree. *deq* is an amortized, doubly-ended queue with *enqueue*, *dequeue*, *pop* and *concat* operations. *prop* is a set of propositional logic transformations like converting a formula to negation normal form and simplifying a formula. *lheap* is a leftist heap data-structure implementation with *merge*, *insert* and *removemax* operations. This benchmark also specified a logarithmic bound on the *right height*: $2^{\mathsf{rheight(h)}} \leq \mathsf{a*heapSize(h)} + \mathsf{b}$ which was solved by the tool. The function leftRightheight (used in the template) computes the right height of the left child of a heap.

trie is a binary prefix tree with operations: *insert*–that inserts a sequence of input bits into the tree, *find*, *create* –that creates a new tree from an input sequence and *delete*–that deletes a sequence of input bits from the tree. The benchmark *cvars* compares two different strategies for sequence concatenation. One strategy exhibits cubic behavior on a sequence of concatenation operations (templates shown in Fig. 6) and the other exhibits a quadratic behavior. *rbt* is an implementation of red-black tree with *insert* and *find* operations. This benchmark also specified a logarithmic bound on the black height: $2^{\mathsf{blackheight(h)}} \leq \mathsf{a*treeSize(h)+b}$ which was solved by the tool.

avl is an implementation of AVL tree with *insert*, *delete* and *find* operations. *bheap* implements a binomial heap with *merge*, *insert* and *deletemin* operations. The functions treenum and minchildren (used in templates), compute the number of trees in a binomial heap and the number of children of the tree containing the minimum element, respectively. *speed* is a functional translation of the code snippets presented in figures 1,2, 9 of [14], and the code snippets on which it was mentioned that the tool failed (Page 138 in [14]). The benchmark *fold* is a collection of fold operations over trees and lists. These were mainly included for evaluation of *depth* bounds.

Fig. 7 shows the results of running our tool on the benchmarks. The column *bound* shows the time bound inferred by the tool for the sample template shown in Fig. 6. This may provide some insights into the constants that were inferred. The bounds inferred are inductive. Though the constants inferred could potentially be rationals, in many cases, the SMT solver returned integer values. In case a value returned by the solver for a parameter is rational, we heuristically check if the ceil of the value also yields an inductive bound. This heuristic allowed us to compute integer values for almost all templates.

The column *time* shows the total time taken for analysing a benchmark. In parentheses we show the time the tool spent in minimizing the bounds after finding a valid initial bound. The subsequent columns provide more insights into the algorithm. The column *VC size* shows the average size of the VCs generated by the benchmarks averaged over all refinements. The tool performed 11 to 42 VC refinements on the benchmarks. The column *disj.* shows the total number of disjuncts falsified by the tool and the column *NL size* shows the average size of the nonlinear constraints solved in each iteration of the solveUNSAT procedure.

Our tool was able to solve 78 out of 80 templates. Two templates were not solvable because of the incompleteness in the handling of nonlinearity. The results also show that our tool was able to keep the average size of the generated nonlinear constraints small in each iteration in spite of the large VC sizes, which

	Sample bound inferred	*time*	*avg.VC*	*disj.*	*NL*
	time≤	*(min.time)*	*size*		*size*
list	9*(size(l)*size(l))+2	17.7s (8.7s)	1539.7	108	59.9
bst	8*(lsize(l)*(height(t)+lsize(l))) +2*lsize(l)+1 29*(lsize(l)*height(t))+7*lsize(l)+1	31s (14.2s)	637.4	79	84
deq	9*qsize(q)+26	17.3s (8.6s)	405.7	80	27.9
prop	52*size(formula)−20	19.5s (1.2s)	1398.5	59	38.1
trie	42*inpsize(inp)+3	3.3s (0.5s)	356.8	54	23.5
sort †	8*(size(l)*size(l))+2	6.8s (1.6s)	274.9	85	29.6
loop	16*size(program)−10	10.6s (4.9s)	1133.8	44	52.4
cvar	5*((n*m)*m)−(n*m)+0*n+8*m+2 9*(n*m)+8*m+0*n+2	25.2s (14.7s)	1423.2	61	49.4
lheap	22*rheight(h1)+22*rheight(h2)+1 44*leftRightheight(h)+5	166.7s (144s)	1970.5s	152	106.4
rbt	178*blackheight(t)+96	124.5s (18.8s)	3881.2	149	132.6
avl	145*height(t)+19	412.1s (259.1s)	1731.8	216	114
bheap	31*treenum(h1)+38*treenum(h2)+1 70*treenum(h1)+31*minchildren(h2)+22	469.1s (427.1s)	2835.5	136	157.2
speed	39*((k+1)*(len(sb1)+len(sb2))) +18*size(str1)+34	28.6s (6.4s)	1084.9	111	85.8
fold	12*(k*k)+2 12*size(t)+1	8.5s (0.8s)	331.8	44	23

Fig. 7. Results of running our tool on the benchmarks. † the tool failed on 2 templates in the *sort* benchmark

is very important since even the state-of-the-art nonlinear constraint solvers do not scale well to large nonlinear constraints.

Fig. 8 shows the results of applying our tool to solve templates for *depth* bounds for our benchmarks. All the templates used were precise. The tool was able to solve all 80 templates provided in the benchmarks. In Fig. 8, the benchmarks which have asymptotically smaller *depth* compared to their execution time (*work*) are starred. Notice that the constants involved in the depth bounds are much smaller for every benchmark compared to its work, even if the depth is not asymptotically smaller than work. Notice that the tool is able to establish that the depth of *mergesort* is linear in the size of its input; the depth of negation normal form transformation is proportional to the nesting depth of its input formula and also that the depth of fold operations on trees is linear in the height of the tree.

Comparison with CEGIS. We compared our tool with *Counter Example Guided Inductive Synthesis*(CEGIS) [26] which, to our knowledge, is the only existing approach that can be used to find values for parameters that would falsify a parametric formula containing ADTs, uninterpreted functions and nonlinear operations. CEGIS is an iterative algorithm that, given a parametric formula ϕ with parameters param and variables X, makes progress by finding a solution

	Inferred depth bound: depth≤	*time*
list	5∗(size(l)∗size(l))+1	9.7s
bst	4∗(lsize(l)∗(height(t)+lsize(l)))+2∗lsize(l)+1 4∗(lsize(l)∗height(t))+4∗lsize(l)+1	335.8s
deq	3∗qsize(q)+13	106.4s
*prop**	5∗nestingDepth(formula)−2	31.4s
trie	8∗inpsize(inp)+1	4.1s
*msort**	45∗size(l)+1	20.2s
qsort	7∗(size(l)∗size(l))+5∗size(l)+1	164.5s
isort	5∗(size(l)∗size(l))+1	3s
loop	7∗size(program)−3	404s
cvar	3∗((n∗m)∗m)−$\frac{1}{8}$∗(n∗m)+n+5∗m+1 3∗(n∗m)+3∗n+4∗m+1	270.8s
lheap	7∗rheight(h1)+7∗rheight(h1)+1 14∗leftRightheight(h)+3	42s
rbt	22∗height(t)+19	115.3s
avl	51∗height(t)+4	185.3s
bheap	7∗treenum(h1)+7∗treenum(h2)+2 22∗treenum(h1)+7∗minchildren(h2)+16	232.5s
speed	6∗((k+1)∗(len(sb1)+len(sb2)))+5∗size(str1)+6	41.8s
*fold**	6∗k+1 5∗height(tree)+1	3.1s

Fig. 8. Results of inferring bounds on *depths* of benchmarks.

for param that rules out at least one assignment for X that was feasible in the earlier iterations. In contrast to our approach which is guaranteed to terminate, CEGIS may diverge if the possible values for X is infinite. We actually implemented CEGIS and evaluated it on our benchmarks. CEGIS diverges even on the simplest of our benchmarks. It follows an infinite ascending chain along which the parameter corresponding to the constant term of the template increases indefinitely. We also evaluated CEGIS by bounding the values of the parameters to be ≤ 200. In this case, CEGIS worked on 5 small benchmarks (viz. *list*, *bst*, *deq*, *trie* and *fold*) but timed out on the rest after 30min. For the benchmarks on which it worked, it was 2.5 times to 64 times slower than our approach.

5 Related Work

We are not aware of any existing approach that can handle the class of templates and programs that our approach handled in the experimental evaluation.

Template-based Invariant Inference. The work of [8] is possibly closest to ours because it performs template-based analysis of imperative programs for finding heap bounds and handles program paths incrementally using the idea of path invariants from [4]. [8] infers only linear bounds. It handles data-structures using a separate shape analysis that tracks the heap sizes. Our approach is for functional programs. We handle a wide range of recursive functions over ADTs and are not restricted to size. We integrate the handling of ADTs into the template solving process, which allows us to solve precise templates. We support nonlinearity and are capable of computing strongest bounds. We are able to handle complex data-structure implementations such as Binomial Heap. [3] presents an approach for handling uninterpreted functions in templates. We handle disjunctions that arise because of axiomatizing uninterpreted functions efficiently through our incremental algorithm that is driven by counter-examples and are able to scale to VCs with hundreds of uninterpreted functions. Our approach also supports algebraic data types and handles sophisticated templates that involve user-defined functions. The idea of using Farkas' lemma to solve linear templates of numerical programs goes back at least to the work of [7] and has been generalized in different directions by [25], [9], [15]. [9] and [25] present systematic approaches for solving nonlinear templates for numerical programs. Our approach is currently based on light-weight axiomatization of nonlinear operations which is targeted towards practical efficiency. It remains to be seen if we can integrate more complete non-linear reasoning into our approach without sacrificing scalability.

Symbolic Resource Bounds Analyses. [14] (SPEED) presents a technique for inferring symbolic bounds on loops of C programs that is based on instrumenting programs with counters, inferring linear invariants on counters and combining the linear invariants to establish a loop bound. This approach is orthogonal to ours where we attempt to find solutions to user-defined templates. In our benchmarks, we included a few code snippets on which it was mentioned that their tool did not work. Our approach was able to handle them when the templates were provided manually. Our approach is also extensible to other resource bounds such as *depth*. The COSTA system of [1] can solve recurrence equations and infer nonlinear time bounds, however, it does not appear to support algebraic data types nor user-defined functions within resource bounds.

Other Related works. Counterexample-guided refinement ideas are ubiquitous in verification, as well as in software synthesis, where they are used in counterexample-guided inductive synthesis (CEGIS) algorithms by [26], [13], and [18]. One important difference in approaches such as ours is that an infinite family of counterexamples is eliminated at once. Our experimental results of comparison with CEGIS in section 4 indicates that these approaches may suffer from similar divergence issues particularly for the resource bound inference problem. Recent work [2] provides a general framework and system for inferring invariants, which can also handle $\exists\forall$ problems of the form we are considering. The comparison of two approaches requires further work because our target are contracts with function invocations whereas [2] targets temporal logic formulas. The underlying HSF tool [11] has been shown applicable to a wide range of analysis problems. HSF could simplify the building of a resource analyzer such as ours, though it does not support algebraic data types and resource bound computation out of the box.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.
2. T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *CAV*, 2013.
3. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, 2007.
4. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, 2007.
5. R. W. Blanc, E. Kneuss, V. Kuncak, and P. Suter. An overview of the Leon verification system. In *Scala Workshop*, 2013.
6. G. E. Blelloch and B. M. Maggs. Parallel algorithms. *Communications of the ACM*, 39:85–97, 1996.
7. M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.
8. B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis. Finding heap-bounds for hardware synthesis. In *FMCAD*, 2009.
9. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, 2005.
10. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.
11. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.
12. R. Guerraoui, V. Kuncak, and G. Losa. Speculative linearizability. In *PLDI*, 2012.
13. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.
14. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, 2009.
15. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.
16. S. Jacobs and V. Kuncak. Towards complete reasoning about axiomatic specifications. In *VMCAI*, 2011.
17. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.
18. E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.
19. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
20. R. Madhavan and V. Kuncak. Symbolic resource bound inference. Technical Report EPFL-REPORT-190578, EPFL, 2014. `http://infoscience.epfl.ch/record/190578`.
21. T. J. M. Makarios. The independence of Tarski's Euclidean axiom. *Archive of Formal Proofs*, October 2012. `http://afp.sf.net/entries/Tarskis_Geometry.shtml`, Formal proof development.
22. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.
23. D. C. Oppen. Elementary bounds for presburger arithmetic. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, 1973.
24. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, 2007.

25. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, 2004.
26. A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.
27. P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, 2010.
28. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, 2011.
29. L. Yu. A formal model of IEEE floating point arithmetic. *Archive of Formal Proofs*, July 2013. `http://afp.sf.net/entries/IEEE_Floating_Point.shtml`, Formal proof development.

A Proof of Correctness, Completeness and Termination

In this section, we prove the correctness and the completeness of the solveUNSAT algorithm presented in section 3.2. Let $\mathcal{T} = (\Sigma, \mathcal{A})$ be the first-order theory of real arithmetic, uninterpreted functions and ADTs. Σ is a signature consisting of real numbers, arithmetic operations/relations, equality relation $=$, function symbols *FuncSyms* and constructor symbols *ConSyms*. $\mathcal{A}$ is a set of closed axioms of the theory defined over Σ.

We consider *linear parametric formulas* $\mathcal{F}$ belonging to $\mathcal{T}$ whose syntax is given below.

$$\begin{aligned}
\mathcal{F} \rightarrow & (\mathcal{F} \vee \mathcal{F}) \mid (\mathcal{F} \wedge \mathcal{F}) \mid (\neg \mathcal{F}) \mid A \\
\textit{Atom: } A \rightarrow & R_a \mid R_t \\
\textit{Term: } T \rightarrow & T_a \mid T_t \mid T_f \\
\textit{Func. Term: } T_f \rightarrow & f(T^*), f \in \textit{FuncSyms} \\
\text{Arith. relation: } R_a \rightarrow & T_a \; op \; T_a, \quad op \in \{=, \leq, <\} \\
\text{Arith. term: } T_a \rightarrow & V_a \mid \mathbb{R} \mid K \cdot V_a \mid T_a \; op \; T_a \mid T_f \quad op \in \{+, -\} \\
\text{Coefficient: } K \rightarrow & \mathbb{R} \mid \textit{Params} \\
\text{ADT relation: } R_t \rightarrow & T_t = T_t \\
\text{ADT term: } T_t \rightarrow & V_t \mid T_f \mid \textit{cons}(T^*), \textit{cons} \in \textit{ConSyms},
\end{aligned}$$

where V_a is a set real valued variables (also referred to as numerical variables), *Params* is a set of parameter variables and V_t is a set of ADT variables. We establish that the procedure solveUNSAT is correct and complete for $\mathcal{F}$.

Notations and Terminology. Let identifiers $\textit{Ids} = V_a \cup V_t \cup \textit{Params} \cup \textit{FuncSyms}$. Let $id(\phi)$ denote the set of identifiers in a formula ϕ. Let $\mathcal{T}_\mathcal{I}$ denote the set of ADT terms that do not have any identifiers. We refer to the elements of $\mathcal{T}_\mathcal{I}$ as instances of ADTs. We associate a *size* with every ADT instance which is the number of constructors used by the instance. Let $\textit{Values} = \mathbb{R} \cup \mathcal{T}_\mathcal{I} \cup (\textit{Values} \mapsto \textit{Values})$ denote the domain of values which includes real numbers, instances of ADTs and functions from values to values. Let substitutions $\textit{Subst} = \textit{Ids} \hookrightarrow \textit{Values}$ be the set of partial mappings from identifiers to values.

Given a substitution σ, let $\phi\,\sigma$ denote the formula obtained by replacing every variable $x \in \mathit{dom}(\sigma)$ by $\sigma(x)$. We say that, σ is model of ϕ (or σ satisfies ϕ) denoted by $\sigma \models \phi$ iff $\mathit{dom}(\sigma) = \mathit{id}(\phi)$ and $\phi\,\sigma$ belongs to the axioms $\mathcal{A}$ of the theory $\mathcal{T}$. Given a formula $\phi \in \mathcal{F}$, we say $\models \phi$ iff, for every $\sigma \in \mathit{Subst}$, $\sigma \models \phi$. Given a formula $\phi \in \mathcal{F}$, we say $\not\models \phi$ iff ϕ is unsatisfiable.

Given a tuple of variables $X : (x_1, \cdots, x_n)$ and $\sigma \in \mathit{Subst}$, $\hat{\sigma}(X)$ denotes the pointwise application of σ, i.e, $(\sigma(x_1), \cdots, \sigma(x_n))$.

A formula ϕ is in conjunctive normal form (CNF) if it is of the form $\phi = \bigwedge_{j=1}^{m} C_j$ and $C_j : \bigvee_{i=1}^{n_j} a_{ij}$, where a_{ij} is an atom. We use the term *disjunct* to refer to a conjunction of atoms and the term *clause* to refer to a disjunction of atoms. Given a formula ϕ in CNF, we say that d is a *disjunct of* ϕ if d is a disjunct and for every clause in the formula ϕ, d contains one atom belonging the clause. That is, d is of the form $a_{i_1 1} \wedge a_{i_2 2} \wedge \cdots \wedge a_{i_m m}$ for some $i_1, i_2, \cdots, i_m$.

For convenience, we interpret *disjuncts* as sets of atoms. Given disjuncts R and S, we use $x \in R$ to denote that x is an atom of R, $R \subseteq S$ to imply that every atom of R is present in S, $R \cup S$ to denote the *conjunction* of atoms in R and S, and $(R \setminus S)$ to denote the disjunct obtained by dropping the atoms in S from R. We use $d \in \mathit{disjunct}(\phi)$ to denote that d is a disjunct of ϕ. In the sequel, we assume that all formulas are in CNF form. We use $[m..n]$ to denote a closed integer interval from m to n.

A.1 Correctness and Completeness of uninterpreted functions and ADT constructors elimination

Choosing a disjunct of ***elimFuncs***(d) **using a model of** d**.**

Let $d \in \mathcal{F}$ and $\rho \models d$. We now define an operation $\mathit{chooseEF}(d, \rho)$ that chooses a disjunct d' of $\mathit{elimFuncs}(d)$, defined in section 3.1, such that $\rho \models d'$. By definition, $\mathit{elimFuncs}(d)$ has three parts: $(d \setminus (F \cup T)) \wedge \delta_1 \wedge \delta_2$. Let d_{num} denote $(d \setminus (F \cup T))$. We add all atoms in d_{num} to d', i.e, $d_{num} \subseteq d'$.

Let $r = f(v_1, \cdots, v_n)$, $r' = f(u_1, \cdots, u_n)$ be two atoms in d and let ψ denote $(\bigwedge\limits_{i=1}^{n} v_i = u_i) \Rightarrow (r = r')$. ψ is a clause i.e, a disjunction of atoms. We choose an atom of ψ that is satisfied by ρ as follows. (a) If $\exists i.\ \rho(v_i) \neq \rho(u_i)$, we choose $v_i = u_i$. (b) Otherwise, $\forall i.\ \rho(v_i) = \rho(u_i)$. Since $\rho \models d$ the following must hold, $\rho(r) = \rho(f)(\hat{\rho}(u_1, \cdots, u_n)) = \rho(f)(\hat{\rho}(v_1, \cdots, v_n)) = \rho(r')$. Therefore, we choose $r = r'$.

For every clause ψ in δ_1, we choose an atom of ψ as described above and add it to d'. Similarly, for every clause C in δ_2 we choose an atom of C that is satisfied by ρ and add it to d'. By construction, $\rho \models d'$. d' is a disjunct of $\mathit{elimFuncs}(d)$ as we have chosen an atom from every clause of $\mathit{elimFuncs}(d)$.

Lemma 1. ***(Model Preservation)*** *If $d \in \mathcal{F}$ is a disjunct and $\rho \models d$ then $\rho \models \mathit{chooseEF}(d, \rho)$ and $\rho \models \mathit{elimFuncs}(d)$* □

Lemma 2. ***(Soundness of elimFuncs)*** *Let $d \in \mathcal{F}$ be a satisfiable disjunct. Let $\iota \in \mathit{Subst}$ be a substitution for parameters, $\mathit{dom}(\iota) = \mathit{Params}$. If $\not\models (\mathit{elimFuncs}(d)\ \iota)$ then $\not\models d\ \iota$*

Proof. We prove the contrapositive form of the lemma: if $d\ \iota$ is satisfiable then $\mathit{elimFuncs}(d)\ \iota$ is satisfiable. Let $\sigma \models d\ \iota$. We now show that $\sigma \models \mathit{elimFuncs}(d)\ \iota$.

Consider the assignment $\sigma' : \sigma \uplus \iota$ defined as follows:

$$\sigma' = \begin{cases} \sigma(x) & x \in \mathit{dom}(\sigma) \\ \iota(x) & x \in \mathit{dom}(\iota) \end{cases}$$

Clearly, σ' is a model of d. By Lemma 1, $\sigma' \models \mathit{elimFuncs}(d)$, which implies that $\sigma \models \mathit{elimFuncs}(d)\ \iota$ □

Lemma 3. ***(Completeness of elimFuncs)*** *Let $d \in \mathcal{F}$ and $\rho \models d$. Let $d' = \mathit{chooseEF}(d, \rho)$. Let $\iota \in \mathit{Subst}$ be a substitution for parameters i.e, $\mathit{dom}(\iota) = \mathit{Params}$. If $\not\models d\ \iota$ then $\not\models d'\ \iota$.*

Proof. We prove the contrapositive form of the lemma: if $d'\ \iota$ is satisfiable then $d\ \iota$ is satisfiable. Let $\sigma' \models d'\ \iota$. Let $d\ \iota = d_{num} \wedge d_{uf} \wedge d_{adt}$ where d_{num} is a conjunction of atoms containing only numerical variables V_a, d_{uf} is a conjunction of atoms of the form $r = f(x_1, \cdots, x_n)$ for some function $f \in \mathit{FuncSyms}$, and d_{adt} is a conjunction of atoms of the form $r = \mathit{cons}(x_1, \cdots, x_n)$ or $u \neq v$ or $u = v$, for some constructor $\mathit{cons} \in \mathit{ConSyms}$ and ADT variables $\{u, v\} \subseteq V_t$.

We construct a model σ of $d\ \iota$ as follows. We start with an initial assignment σ_{init} that binds the numerical variables V_a in $d\ \iota$ to concrete values. We incrementally consider each atom a of $d\ \iota$ and extend the initial assignment by introducing bindings for previously unseen identifiers in a in such a way that a is satisfied. The assignment obtained after processing every atom of $d\ \iota$ models $d\ \iota$.

Let the initial assignment σ_{init} be the projection of σ' on to variables in V_a. That is, $\sigma_{init}(x) = \sigma'(x)$ if x is a numerical variable.

Part-I: $\sigma_{init} \models d_{num}$

d_{num} is a part of $d'\ \iota$ i.e, $d_{num} \subseteq d'\ \iota$. This follows from the definition of $\mathit{elimFuncs}(d)$. Therefore, $\sigma' \models d_{num}$. By definition, d_{num} has only numerical variables in V_a. Therefore, $\sigma_{init} \models d_{num}$.

Part-II: Extending σ_{init} to variables in d_{adt}

We introduce a few definitions and operations that will be useful in the rest of the proof. We say that $\sigma \in \mathit{subst}$ *embeds* ($\preccurlyeq$) in $\rho \in \mathit{subst}$, iff the following conditions holds:

1. $\mathit{dom}(\sigma) \subseteq \mathit{dom}(\rho)$.
2. σ preserves all equalities and dis-equalities between ADT variables V_t implied by ρ.
$$\forall \{x, y\} \subseteq \mathit{dom}(\sigma) \cap V_t. \quad \rho(x) = \rho(y) \Leftrightarrow \sigma(x) = \sigma(y)$$
3. For every variable $x \in \mathit{dom}(\sigma) \cap V_t$, $\rho(x)$ and $\sigma(x)$ are *shape isomorphic.* That is, $\rho(x)$ and $\sigma(x)$ may differ only in the numerical values.

We say that an assignment σ is a S-model for some set S of atoms iff σ satisfies every atom in S. Given a S-model σ, an assignment ρ such that $\sigma \preccurlyeq \rho$ and a variable $x \in \mathit{dom}(\rho) \setminus \mathit{dom}(\sigma)$, we define $\sigma \triangleright_\rho x$ as an assignment that extends σ by introducing a mapping for x such that $(\sigma \triangleright_\rho x) \preccurlyeq \rho$ and $\sigma \triangleright_\rho x$ is a S-model.

$$(\sigma \triangleright_\rho x)(u) = \begin{cases} \sigma(v) & \exists v \in \mathit{dom}(\sigma).\ \rho(v) = \rho(u) \\ \mathit{freshen}_\sigma(\rho(u)) & \mathit{otherwise} \end{cases}$$

where $\mathit{freshen}_\sigma(v)$ return a fresh numeral that does not occur in σ if v is a numeral. If v is an ADT instance, it replaces every numeral in v by a fresh numeral that does not occur in σ.

It is easy to see that $\sigma \triangleright_\rho x \preccurlyeq \rho$ as all the three properties of embedding are preserved by the above definition. It is also obvious that $\sigma \triangleright_\rho x$ is a S-model as x does not appear in S since $x \notin \mathit{dom}(\sigma)$ and σ is a S-model. With these definitions, we now proceed to the proof.

Let $d_{adt} = d_{cons} \wedge d_{comp}$, where d_{cons} is a conjunction of atoms of the form $r = \mathit{cons}(x_1, \cdots, x_n)$ and d_{comp} is a conjunction of atoms of the form $u \neq v$ or $u = v$.

Let the atoms in d_{cons} be ordered as $a_1 : r_1 = \mathit{cons}_1(X_1), \cdots, a_n : r_n = \mathit{cons}_n(X_n)$ such that for each r_i, the *size* of the ADT instance $\rho(r_i)$ is larger than or equal to the size of every $\rho(r_j)$, $j \in [\mathit{1..i}]$. This ordering ensures that $\rho(r_i)$ cannot have as sub-structures $\{\rho(r_j) \mid j > i\}$. Given a set of ADT instances, such an ordering is always possible as the ADTs are *acyclic*.

We construct, as explained below, a sequence of assignments $\sigma_0, \cdots, \sigma_n$, where $\sigma_0 = \sigma_{init}$ such that each σ_i satisfies the following properties: (a) $\sigma_i \models \bigwedge_{j=1}^{i} a_i$ and (b) $\sigma_i \preccurlyeq \rho$. Note that σ_0, which equals σ_{init}, satisfies the above properties. Given an assignment σ_{i-1} that satisfies the above properties, we now show how to construct an assignment σ_i satisfying the above properties.

Let the atom a_i be $r = \mathit{cons}(x_1, \cdots, x_m)$. Let $S = \{a_j \mid j \in [\mathit{1..i}]\}$. Given, σ_{i-1} is a S-model. Let $W \subseteq \{x_1, \cdots, x_m\}$ be the set of variables that do not belong to $\mathit{dom}(\sigma_{i-1})$. Let $W = \{w_1, \cdots, w_k\}$. Define σ_{ext} as

$$\sigma_{ext} = (\cdots((\sigma_{i-1} \triangleright_\rho w_1) \triangleright_\rho w_2) \cdots) \triangleright_\rho w_k$$

By the definition of $\triangleright$, σ_{ext} is a S-model and $\sigma_{ext} \preccurlyeq \rho$. σ_{ext} has a mapping for all $x_1, \cdots, x_m$.

Let t denote the ADT instance $\mathit{cons}(\widehat{\sigma_{ext}}(x_1, \cdots, x_m))$. Define σ_i as $\sigma_{ext}[r \mapsto t]$.

Claim. If $r' \in \mathit{dom}(\sigma_{ext})$ and $\rho(r') = \rho(r)$ then, $\sigma_{ext}(r') = t$

Proof. Since $r' \in \mathit{dom}(\sigma_{ext})$ it should either be an argument passed to the constructor *cons* (say x_i) or should belong to $\mathit{dom}(\sigma_{i-1})$. If $r' = x_i$ then, $\rho(r)$ must contain itself, as $\rho \models a_i : r = \mathit{cons}(\cdots, x_i, \cdots)$ and $\rho(r') = \rho(r)$. This is not possible as $\rho(r)$ is acyclic. Therefore, r' cannot be an argument x_i. Therefore, say $r' \in \mathit{dom}(\sigma_{i-1})$.

There are two scenarios in which r' could have been assigned a mapping in σ_{i-1}: (a) there exists $a_j : z = \mathit{cons}'(y_1, \cdots, y_m)$, $j < i$ and $r' = y_i$ for some $i \in [\mathit{1}..m]$ or (b) there exists $a_j : r' = \mathit{cons}'(y_1, \cdots, y_m)$, $j < i$.

Consider case (a). Say r' is an argument y_i of cons'. Clearly, y_i is a substructure of z. Since, ρ is a model of a_j, $\rho(y_i)$ is a sub-structure of $\rho(z)$. By our choice of ordering of the atoms, $\rho(r)$ is at least as large as $\rho(z)$ as a_j precedes a_i in the ordering. Therefore, $\rho(y_i)$ is strictly smaller than $\rho(r)$. Since $\sigma_{ext} \preccurlyeq \rho$, by property 3, $\rho(r)$ and $\sigma_{ext}(r)$ have the same size, and $\rho(y_i)$ and $\sigma_{ext}(y_i)$ have the same size. Therefore, $\sigma_{ext}(y_i)$ is strictly smaller than $\sigma_{ext}(r)$. Since, $r' = y_i$, $\sigma_{ext}(r')$ is strictly smaller than $\sigma_{ext}(r)$. Which contradicts our premise that $\sigma_{ext}(r') = \sigma_{ext}(r)$. Therefore, this case is not possible.

Consider case (b). If cons' and cons are different constructors then $\rho(r) = \rho(r')$ is not possible. Therefore, assume that cons and cons' are equal. In the rest of the proof, we use cons instead of cons'.

Recall that $d' = \mathit{chooseEF}(d, \rho)$. Since $\rho(r) = \rho(r')$ the atom $r = r'$ would belong to d'. Similarly, $\bigwedge_{i=1}^{m}(x_i = y_i) \subseteq d'$ as for all i, $\rho(x_i) = \rho(y_i)$. If, for some i, $\{x_i, y_i\} \subseteq V_a$ then $\sigma_{ext} \models (x_i = y_i)$. This is because $(x_i = y_i) \in d'$, $\sigma' \models d'$, σ_{init} is a projection of σ' on to V_a and $\sigma_{init} \subseteq \sigma_{ext}$.

Therefore, say $\{x_i, y_i\} \subseteq V_t$. Since $\rho(x_i) = \rho(y_i)$ and $\sigma_{ext} \preccurlyeq \rho$, $\sigma_{ext}(x_i) = \sigma_{ext}(y_i)$. Hence, $\sigma_{ext}(x_i) = \sigma_{ext}(y_i)$, for all $i \in [\mathit{1}..m]$. Hence, $\sigma_{ext}(r') = t$. □

Claim. If $r' \in \mathit{dom}(\sigma_{ext})$ and $\rho(r') \neq \rho(r)$ then, $\sigma_{ext}(r') \neq t$.

Proof. $r' \in \mathit{dom}(\sigma_{ext})$ implies that $r' \in W$ or $r' \in \mathit{dom}(\sigma_{i-1})$, where W is the set of constructor arguments that do not belong to $\mathit{dom}(\sigma_{i-1})$. Clearly, if $r' \in W$ then, $\sigma_{ext}(r') \neq t$ by the definition of t. Hence, the claim holds.

Say $r' \in \mathit{dom}(\sigma_{i-1})$. There are two scenarios in which r' could have been assigned a mapping in σ_{i-1}: (a) there exists $a_j : z = \mathit{cons}'(y_1, \cdots, y_m)$, $j < i$ and $r' = y_i$ for some $i \in [\mathit{1}..m]$ or (b) there exists $a_j : r' = \mathit{cons}'(y_1, \cdots, y_m)$, $j < i$.

Consider case (a). As explained in the proof of the above claim, $\sigma_{ext}(r')$ is strictly smaller than $\sigma_{ext}(r)$ in this case. Therefore, $\sigma_{ext}(r') \neq t$ holds. Hence, the claim holds.

Consider case (b). If cons and cons' are different constructors then $\sigma_{ext}(r') \neq t$ and hence the claim holds. Hence, consider the case where cons and cons' are the same constructor.

Given $\rho(r) \neq \rho(r')$. Therefore, the values of atleast one of the constructor arguments should be different in ρ. Recall that $d' = \mathit{chooseEF}(d, \rho)$. By definition of *chooseEF*, an atom of the form $x_i \neq y_i$, for some i, wherein $\rho(x_i) \neq \rho(y_i)$ will belong to d'. If $\{x_i, y_i\} \subseteq V_a$ (numerical variables) then $\sigma_{ext} \models x_i \neq y_i$. This is because $(x_i \neq y_i) \in d'$, $\sigma' \models d'$, σ_{init} is a projection of σ' on to V_a and $\sigma_{init} \subseteq \sigma_{ext}$.

Consider the case where $\{x_i, y_i\} \subseteq V_t$. $x_i \neq y_i$ belongs to d' only because $\rho(x_i) \neq \rho(y_i)$. Since $\sigma_{ext} \preccurlyeq \rho$, $\sigma_{ext}(x_i) \neq \sigma_{ext}(y_i)$ (property 2 of $\preccurlyeq$ definition). Hence, $\sigma_{ext}(r') \neq t$. □

Claim. t is shape isomorphic to $\rho(r)$. That is, t and $\rho(r)$ differ only in numerical values.

Proof. By definition, $t = cons(\widehat{\sigma_{ext}}(x_1, \cdots, x_m))$. For each $x_i \in V_t$, $\sigma_{ext}(x_i)$ is either $\sigma_{i-1}(x_i)$ or is obtained by applying an extension operation $\sigma \triangleright_\rho x_i$ for some $\sigma \preccurlyeq \rho$. In the former case, $\sigma_{ext}(x_i)$ and $\rho(x_i)$ are shape isomorphic by hypothesis. In the latter case, they are shape isomorphic by the definition of $\triangleright$. Since, ρ satisfies $a_i : r = cons(x_1, \cdots, x_m)$, $\rho(r) = cons(\hat{\rho}(x_1, \cdots, x_n))$. The fact that, for all $i \in [1..m]$, $\rho(x_i)$ is shape isomorphic to $\sigma_{ext}(x_i)$ implies the claim. □

By the above three claims, $\sigma_i \preccurlyeq \rho$. By definition of t, $(r \mapsto t) \models a_i$. Therefore, σ_i models $(S \cup \{a_i\})$.

The claims hold for all $\sigma_i, 0 \leq i \leq n$ by induction. Therefore, $\sigma_n \models d_{cons}$ and $\sigma_n \preccurlyeq \rho$. Let $x_1, \cdots, x_m$ be the set of variables in $dom(\rho) \cap V_t$ that do not have a mapping in σ_n. Define an assignment $\sigma_{adt} = (\cdots((\sigma_n \triangleright_\rho v_1) \triangleright_\rho v_2) \cdots) \triangleright_\rho v_m)$. By definition of $\triangleright$, $\sigma_{adt} \models d_{cons}$ and $\sigma_{adt} \preccurlyeq \rho$. Furthermore, σ_{adt} has a mapping for all ADT variables in d.

Now it is almost trivial to prove that d_{comp} is satisfied by σ_{adt}. Since, σ_{adt} implies all equalities and disequalities implied by ρ (property 2 of $\preccurlyeq$ relation), and since ρ satisfies d_{comp}, σ_{adt} also satisfies d_{comp}.

Part-III: Extending σ_{adt} to uninterpreted functions in d_{uf}

We now extend the assignment σ_{adt} to uninterpreted functions in the disjunct d. Let $d_{uf} = \bigwedge_{i=1}^{n} a_i$. Similar to Part-II, we create a sequence of assignments $\sigma_0, \sigma_1, \cdots, \sigma_n$, where $\sigma_0 = \sigma_{adt}$, such that each σ_i satisfies all atoms a_1 to a_i. For convenience, we allow the assignments to have only partially definitions for the functions.

σ_0 trivially satisfies the claim. Assume that σ_{i-1} satisfies all atoms up to a_{i-1}. We construct a σ_i as described below that satisfies all atoms up to a_i. Let $a_i : r = f(u_1, \cdots, u_n)$. Let $E = \widehat{\sigma_{i-1}}(u_1, \cdots, u_n)$.

Define σ_i as σ_{i-1} if f is defined for E in σ_{i-1}. Otherwise, let σ_i be $(\sigma_{i-1} \cup \{f \mapsto (E \mapsto \sigma_{i-1}(r))\}$, where the $\cup$ operation combines the partial definitions of f given by the left and the right operands.

Claim. If f is defined for E in σ_{i-1} then, $(\sigma_{i-1}(f))(E) = \sigma_{i-1}(r)$

Proof. Since f is defined for E in σ, it should have been added to some σ_j, $j < i$. Which implies that $a_j : r' = f(v_1, \cdots, v_m)$, $\widehat{\sigma_{j-1}}(v_1, \cdots, v_m) = E$ and $\sigma_j(f)(E) = \sigma_{j-1}(r')$. We now show that $\sigma_{j-1}(r')$ (which equals $\sigma_{i-1}(r')$) and $\sigma_{i-1}(r)$ are equal.

There are two cases to consider: (a) $\rho(r) = \rho(r')$ and (b) $\rho(r) \neq \rho(r')$.

Consider case (a). If $\{r, r'\} \subseteq V_t$ then $\sigma_{i-1}(r) = \sigma_{i-1}(r')$ as $\sigma_{i-1} \preccurlyeq \rho$. Hence, the claim holds. Say $\{r, r'\} \subseteq V_a$. Since $\rho(r) = \rho(r')$, by the choice of the disjunct d', $r = r'$ belongs to d'. This implies that $\sigma_{init}(r)$ (which is $\subseteq \sigma_{i-1}(r)$) models $r = r'$ as $\sigma' \models (r = r')$ and σ_{init} is projection of σ' on to V_a. Therefore, the claim holds.

Consider case (b): $\rho(r) \neq \rho(r')$. In this case, by the choice of d', a predicate $u_i \neq v_i$, for some $i \in [1..m]$, wherein $\rho(u_i) \neq \rho(v_i)$ will belong to d'. By an argument similar to the previous cases, it can be shown that in this case, $\sigma_{i-1}(u_i) \neq \sigma_{i-1}(v_i)$ irrespective of whether u_i, v_i belong to V_a or V_t. Therefore, $\widehat{\sigma_{i-1}}(v_1, \cdots, v_n) \neq E$ and hence, $\widehat{\sigma_{j-1}}(v_1, \cdots, v_n) \neq E$ (since $\sigma_{j-1} \subseteq \sigma_{i-1}$). This implies that the entry for E is not added by the atom a_j contrary to our assumption. Hence, this case is not possible. □

By the above claim, $\sigma_i \models a_i$. Therefore, by induction, σ_n satisfies d_{uf}.

Theorem 2. *Let $\phi \in \mathcal{F}$ be a parametric formula with parameters* params*. Let $\iota =$ solveUNSAT$(\phi,$ params$)$.*

1. *The procedure* solveUNSAT *is correct. That is, if $\iota \neq \emptyset$ then $\phi\ \iota$ is unsatisfiable.*
2. *The procedure* solveUNSAT *is complete. That is, if $\iota = \emptyset$ then there does not exist an assignment for* params *that will falsify ϕ.*
3. *The procedure* solveUNSAT *terminates.*

Proof. The solveUNSAT algorithm shown in Figure 5 returns a non-empty model ι only at line 9. This line will be reached only when $\phi_{inst} = \phi\ \iota$ is unsatisfiable. Hence, solveUNSAT is correct.

The solveUNSAT algorithm returns $\emptyset$ only at line 19. When line 19 is reached the following properties hold.

(a) $\sigma' \models d$, where $d \in \mathit{disjunct}(\phi)$,
(b) $d' = \mathit{chooseEF}(d, \sigma')$, and
(c) there exists no assignment ι for params such that $d'\ \iota$ is unsatisfiable.

The property (c) follows from the completeness of Farkas' Lemma for linear real arithmetic. By Lemma 3, (a),(b) and (c) imply that there does not exist an assignment for params that will falsify d and hence, ϕ. Therefore, the procedure is complete.

Let d be a disjunct of ϕ. Let $\{d_1, \cdots, d_m\}$ be the disjuncts of $\mathit{elimFuncs}(d)$. d can be chosen at line 11 of the algorithm at most m times. This is because, every time d is chosen, at least one $d_i, i \in [\mathit{1}..m]$ that was satisfiable in the earlier iterations will be falsified by the new assignment computed at line 22. Once all disjuncts $d_i, i \in [\mathit{1}..m]$ are falsified by an assignment ι for params, $\mathit{elimFuncs}(d)\ \iota$ will be unsatisfiable. By Lemma 2, $\not\models \mathit{elimFuncs}(d)\ \iota$ implies $\not\models d\ \iota$. Hence, d will be falsified when all the disjuncts of $\mathit{elimFuncs}(d)$ are falsified. Therefore, d can be chosen at line 11 of the algorithm at most m times. Since there are only a finite number of disjuncts in ϕ, the number of iterations of the algorithm is bounded.

In each iteration, the algorithm solves for the non-linear constraints C. Since C is a quantifier-free real arithmetic formula it has a decision procedure. Therefore, the algorithm terminates. □