On Template-Based Inference of Rich Invariants in Leon

EPFL Technical Report EPFL-REPORT-190578, November 2013

Ravichandhran Madhavan
EPFL
ravi.kandhadai@epfl.ch

Viktor Kuncak
EPFL
viktor.kuncak@epfl.ch

Abstract

We present an approach for inferring rich invariants involving user-defined recursive functions over numerical and algebraic data types. In our approach, the developer provides the desired shape of the invariant using a set of templates. The templates are quantifier-free affine predicates with unknown coefficients. We also provide an enumeration based strategy for automatically inferring some of the templates. We present a scalable counter-example driven algorithm that finds the unknown coefficients in templates and thus computes expressive inductive invariants. Our algorithm incrementally solves a set of quantified constraints involving recursive functions and data structures. We discuss several optimizations that make the approach scale to complex programs and present an empirical evaluation. Our implementation proves correctness properties as well as symbolic bounds on running times of recursive programs. For example, the implementation establishes that the time taken to insert into a red-black tree is bounded by the logarithm of its size.

1. Introduction

Automated reasoning about functional programs is a challenging task because the expectations on the complexity of proving such programs are high. Functional languages eliminate many sources of low-level errors. The properties that need to be proved then have the nature of problem-specific inductive mathematical reasoning over discrete structures. Automating this process is important both for reliability of functional programming infrastructure, today used in many companies[1], as well as for general rigorous reasoning about systems described using recursive functions as shown by many case studies using interactive theorem provers (Kaufmann et al. [21], Leroy [23]).

Many automated program analysis and verification tasks can be formulated as checking whether there *exists* an invariant that is preserved in *all* steps of the execution. The $\exists\forall$ form of the problem makes it among the most difficult types of problems being addressed with automated analysis tools today. The analysis we present in this paper aims to discover invariants that can prove program correctness as well as bounds on program execution time. We consider this problem in the context of first-order functional programs to isolate the intrinsic difficulty of reasoning about correctness from the questions of low-level representations of data in terms of pointers and mutable structures. Detailed correctness invariants often need to contain invocations of user-defined recursive functions that are specific to the program being verified. For example, to describe data structure invariants, it is often most convenient to write recursive functions that check these invariants (see, e.g., Blanc et al. [5], Suter et al. [36]). This paper makes a step towards automatically generating invariants containing such user-defined functions.

Our approach takes as inputs the desired shapes of invariants, which we call a *template*. The goal of the analysis becomes finding coefficients in the template. The resulting problem remains of $\exists\forall$ form, but now the existential quantifiers range over numerical coefficients, instead of over invariants. One of the useful techniques to solve such invariants over numeric domains and array-like structures is Farkas lemma, which converts a $\exists\forall$ problem over linear constraints into a $\forall$ problem over non-linear constraints. While explored before, this basic approach is becoming more interesting in the light of advances in non-linear reasoning inside solvers such as Z3. A notable aspect of our problem is that the program and the templates being solved contain not only numerical operations and array-like structures, but also contain algebraic data types, recursive functions, and a large number of disjunctions that encode the conditionals. We present techniques that handle all these language features through an incremental and counterexample-driven algorithm that fully leverages the ability of SMT solver to handle disjunctions. We show that our technique is effective for the problem of discovering highly application-specific invariants in functional programs. To our knowledge, no other tool has been demonstrated the ability to infer the expressive kinds of invariants that we present. We therefore consider, as our main contribution, the system implementing our algorithm with several optimizations and extensions that achieved such expressive power and automation on a set of benchmarks in this challenging domain.

[1] `http://www.haskell.org/haskellwiki/Haskell_in_industry`, `http://typesafe.com/company/casestudies`

```
sealed abstract class Color
case class Red() extends Color
case class Black() extends Color

sealed abstract class Tree
case class Empty() extends Tree
case class Node(color: Color, left: Tree, value: Int, right: Tree)
      extends Tree
```

Figure 1. Red-black tree data structure in Scala

```
def twopower(x: Int) : Int = {
  require(0 ≤ x)
  if(x < 1) 1
  else 2∗twopower(x−1) }

def size(t: Tree): Int = {
  require(blackBalanced(t))
  t match {
    case Empty() ⇒ 0
    case Node(_, l, v, r) ⇒ size(l) + 1 + size(r)
    }
} ensuring(res ⇒ true
  template((a,b) ⇒ twopower(blackHeight(t)) ≤ a∗res + b))
```

Figure 2. Computing the size of a red-black tree

```
def ins(x: Int, t: Tree): Tree = {
  require(redBlack(t) && blackBalanced(t))
  t match {
    case Empty ⇒ Node(Red,Empty,x,Empty)
    case Node(c,a,y,b) ⇒
      if (x < y) balance(c, ins(x, a), y, b)
      else if (x == y) Node(c,a,y,b)
      else balance(c,a,y,ins(x, b))
}}ensuring (res ⇒ true
      template((a,b) ⇒ time ≤ a∗blackHeight(t) + b))

def blackHeight(t : Tree) : Int = { t match {
  case Empty() ⇒ 0
  case Node(Black(), l, _, _) ⇒ blackHeight(l) + 1
  case Node(Red(), l, _, _) ⇒ blackHeight(l)
}}

def redBlack(t: Tree) : Boolean = t match {
  case Empty() ⇒ true
  case Node(Black(), l, _, r) ⇒ redBlack(l) && redBlack(r)
  case Node(Red(), l, _, r) ⇒ isBlack(l) && isBlack(r)
                              && redBlack(l) && redBlack(r)
  case _ ⇒ false }

def blackBalanced(t : Tree) : Boolean = t match {
  case Node(_,l,_,r) ⇒ blackBalanced(l) && blackBalanced(r)
                       && blackHeight(l) == blackHeight(r)
  case _ ⇒ true }
```

Figure 3. Inserting an element into a red-black tree.

1.1 Examples

To illustrate the scope of our approach, consider an implementation of the red-black tree data structure in Scala shown in Figures 1, 3, and 2. The pre and postconditions are specified using the constructs **require** and **ensuring**. The postcondition must hold whenever the execution reaches the exit point of the procedure. Consider function ins in Figure 3, which is the main step of insertion of a node into a tree. The function uses the fact that the tree is sorted to examine only the relevant path in the tree. As it returns from insertion, it performs balancing of the tree through rotations, ensuring that the height of the tree is in the order of the logarithm of trees size. To rigorously establish the performance of this code, we would like to verify that the running time of the function is linear in the number of black nodes in the tree, which in turn is bounded by the logarithm of the size of the tree. For this purpose, we introduce a function blackHeight that counts the number of black nodes on an arbitrary path. This computation is sensible because red-black trees have the blackBalanced invariant, ensuring that the number of black nodes is the same on all paths of the tree. Furthermore, we write the function size in Figure 2 that computes the total number of nodes in a tree. All these functions are executable, so we can use them for computation. In our system we can also use these functions in preconditions and postconditions, which gives the language both great expressive power and ease of use.

To establish the desired performance property, we would like to show the relationships between tree size, tree black height, and the execution time of ins. While we have an intuition of these properties from our knowledge of the data structure, showing that it holds for this particular implementation would require us to determine a number of constants that depend on the compilation and time measurement model. Our system aids the developer in such tasks, where the structure of the invariants is known, but the exact details are known. Analogous, to work in sketching of Solar-Lezama et al. [34], we allow the user to specify a "sketch" of the postcondition, which, following verification terminology, we call a *template*. The system can discover simple templates automatically. For more complex cases the user writes down in the **ensuring** clause a **template** construct that lists parameters and then a condition referring to these parameters. In our example, the postcondition of size function in Figure 2 contains in its **ensuring** clause the template

$$2^{\mathsf{blackHeight}(t)} \leq a \cdot \mathsf{res} + b$$

which indicates that the power of the black height is bounded by some linear function of the result (**res**) of the size function. The user function 2^n itself is also specified as an executable recursive function twopower. Note that the user does not need to specify the coefficients a, b of the linear bound. Our system automatically infers the values $a = 4$ and $b = 2$ and proves the inductive invariant $2^{\mathsf{blackHeight}(t)} \leq 4 \cdot \mathsf{res} + 2$. In

```
def merge(h1: Heap, h2: Heap) : Heap = {
  require(hasLeftistProperty(h1) && hasLeftistProperty(h2))
  h1 match {
    case Leaf() ⇒ h2
    case Node(_, v1, l1, r1) ⇒ h2 match {
      case Leaf() ⇒ h1
      case Node(_, v2, l2, r2) ⇒
        if(v1 > v2) makeT(v1, l1, merge(r1, h2))
        else makeT(v2, l2, merge(h1, r2)) }}}
ensuring(res ⇒ true template((a,b,c) ⇒
  time ≤ a*rightHeight(h1) + b*rightHeight(h2) + c))
```

Figure 4. Merge of two leftist heaps

general, our verification algorithm solves for the unknown coefficients so that the postcondition in conjunction with the template expression becomes an inductive invariant of the procedure.

Our system also allows postconditions to refer to the execution time spent in function invocation, denoted by **time**. To specify that the execution time of ins is $O(\mathsf{blackHeight}(t))$, we write in **ensuring** clause of ins a template

$$\mathbf{time} \leq a \cdot \mathsf{blackHeight}(t) + b$$

This provides the user a very expressive way to describe bounds on execution time, without knowing the specific constants. Our analysis automatically finds the values of parameters in the bound and proves a bound

$$\mathbf{time} \leq 136 \cdot \mathsf{blackHeight}(t) + 72$$

Note that this bound involves computing the running time of recursive functions over trees, and comparing it to the invocation of another recursively defined function. In this process, our analysis also infers auxiliary inductive properties of other functions that are used, such as the property that blackHeight always returns a positive number. Through inference of unknown parameters, the system substantially simplifies the verification process for user-defined properties of data structures. The technique applies to other data structures as well, with a range of different bounds definable using recursive functions. As another example, Figure 4 shows a merge operation on leftist heaps, for which our system succeeds in showing a running time that is a linear function of the heights of the two heaps given as arguments. Together with an analogous specification of size in terms of rightHeight as in red-black tree, this establishes a logarithmic time bound for merge. For other examples analyzer succeeds in proving polynomial bounds with different exponents, differentiating between more efficient and less efficient versions. We discuss these and further examples in our evaluation in Section 4, including results showing fully automated inference of postconditions by inferring the templates themselves (Figure 13).

```
1  def constructVC(body, pre, inv) {
2   let res be the variable denoting the result in inv
3   let φ_pre, φ_body, φ_inv be formulas
4      representing pre, (res == body), inv
5   let φ be (φ_pre ∧ φ_body ∧ ¬(φ_inv))
6   return flattenTerms(φ) }
```

Figure 5. Constructing a verification condition for a function.

```
1  def refineVC(φ_vc, F) {
2   for each atom r = f(V) in φ_vc that is not inlined {
3    let (pre, body, inv) be F(f)
4    Replace formal parameters by V in pre, body and inv
5    let φ_pre, φ_body and φ_inv be formulas
6      representing pre, (r == body) and inv[r/res]
7    Conjoin r = f(V) with φ_body ∧ (φ_pre ⇒ φ_inv) in vc
8    if f is recursive and f was not encountered before {
9      let φ_f be constructVC(body, pre, inv)
10     conjoin φ_f with vc }}}
```

Figure 6. Refining a verification condition by unrolling functions. The parameter F is a mapping from functions to their definitions and candidate invariants.

2. The Invariant Inference Algorithm

We next present the core of our invariant inference algorithm. The algorithm is implemented on top of Leon's verification algorithm (see Suter et al. [36], Blanc et al. [5]), and benefits from its fair unrolling of recursive functions and efficient communication with the Z3 solver.

Main aspects of our algorithm are in figures 8, 9 10, 11, which describe a counter-example driven approach for incrementally finding inductive invariants in the presence of templates. Incremental approach is important to avoid the expensive non-linear constraint solving to be applied to large constraints, or invoked too many times. The reason for using non-linear constraint solving is that solving these constraints gives much stronger progress guarantees (Beyer et al. [4]) than removing one concrete counterexample as in counterexample-guided inductive synthesis with concrete instances.

2.1 Refining Verification Conditions

Our algorithm operates on verification conditions (VCs), which are quantifier-free formulas that refer to invocations of recursive functions. In addition to propositional logic operations, the verification conditions contain operations of integer linear arithmetic, algebraic data type operations, and function symbol applications. The VCs we generate for a function and an expression is a sufficient (but not necessary) condition for the expression to be an inductive invariant of the function. The process of converting a function with the invariant to be verified to a verification condition is straight forward and is sketched in Figure 5. In our approach, we

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

def append1(l1 : List, l2 : List) : (List, Int) = {
  val t1 = 1
  l1 match {
   case Nil() ⇒ (l2, t1)
   case Cons(x,xs) ⇒ { val (e2, t3) = {
         val (e4, t5) = {
            val (e7, t8) = append1(xs, l2)
            (e7, 2 + t8)
         }
        (Cons(x, e4), (t5 + 2)) }
    (e2, (t1 + t3) )}}
} ensuring(res ⇒ true
  template((a,c) ⇒ res._2 ≤ a∗size(l1) + c))
```

Figure 7. An example illustrating the instrumentation of time. append1 is the instrumented version of the function append

incrementally make the verification conditions more precise by unrolling the functions in it as outlined in Figure 6.

2.2 Instrumenting Programs with Time

To allow the users to refer to the time spent in an invocation of a function, we instrument the function body with a time variable that counts the number of operations performed by the function. We convert every expression and sub-expression e in the program that is not a variable or constant to a pair (e,t), where t is the time taken by the expression and is as the sum of the execution times of the sub-expressions of e (including function calls) plus the cost of the operation performed by e. The cost of every operation is by default set to 1 in our implementation, however this is configurable by the user. The instrumentation process is performed transitively on all functions invoked inside the body. The functions instrumented with time are modified to return the time spent in their bodies in addition to their original return values, for tracking time in their callers. Figure 7 illustrates the instrumentation with a function that concatenates two lists. The function append1 is the instrumented version of the function append. Notice that the instrumentation significantly increases the program sizes and also necessitates efficient handling of tuples.

2.3 Finding Template Parameters

The algorithm shown in Figure 8 outlines the steps involved in solving the templates in the postconditions. For each function f in the given program, we construct a verification condition ϕ_{vc}. The condition ϕ_{vc} may have as free variables, the template variables $tvars$ (which are the unknown coefficients in the template) and program variables X. The goal of

```
1  input :
2   A program P with functions f_1, ··· , f_n
3  output :
4   A mapping Invs from functions to inductive invariants
5
6   def solveTemplates(P) {
7     let (pre_i, body_i, post_i, tmpl_i) be the definition of f_i
8     let F be a mapping from f_i to (pre_i, body_i, (post_i ∧ tmpl_i))
9     var Invs = ∅
10    for each function f_i in P {
11      let (pre, body, tmpl) be F(f)
12      let params be the set of unknown coefficients in tmpl
13      let φ_vc be constructVC(body, pre, tmpl)
14      var sol = ∅
15      while (sol == ∅ and iteration limit not reached) {
16        sol = solveUNSAT(params, φ_vc)
17        if (sol == ∅)
18          refineVC(φ_vc, F) }
19      let inv be obtained by replacing every t ∈ params
20        by sol(t) in tmpl
21      Add (f ↦ inv) to Invs }
22    return Invs }
```

Figure 8. An algorithm that solves the unknown coefficients of the templates in function postconditions

the remaining steps is to find the values of template parameters that makes ϕ_{vc} unsatisfiable.

The non-linearity in the verification condition ϕ_{vc} is introduced only due to the templates since our program expressions are linear modulo recursive functions. Hence, every non-linear term is of the form ax where a is a free template variable and x is a bound universally quantified program variable. We refer to such formulas as *parametric linear formulas*.

The procedure solveUNSAT(params,ϕ_{vc}) invoked at line 16 of Figure 8 tries to find values for parameters params such that the formula resulting after fixing these parameters becomes unsatisfiable (has no satisfying interpretation for program variables and uninterpreted function symbols). If solveUNSAT does not find values for parameters, we refine the verification condition and retry the above steps.

2.4 Solving Parametric Formulas

Figure 9 presents our algorithm for solving an alternating satisfiability problem: finding the values ι of params such that replacing params according to ι results in unsatisfiable formula. At a high level, our algorithm iteratively explores the satisfiable disjuncts in the formula ϕ. In each iteration, it comes up with an candidate model that invalidates all the disjuncts explored thus far. The algorithm halts when every disjunct of ϕ is unsatisfiable or when it is unable to find an invalidating assignment for a disjunct during some iteration. In every iteration, our algorithm invalidates atleast one disjunct that was not previously encountered and therefore, terminates. In the sequel, we explain our approach in detail.

```
input :
 A parametric linear formula φ with parameters 'params'
output :
 Assignments for params such that φ(params) is unsatisfiable,
 or ∅ if no such assignment exists

def solveUNSAT(params, φ) {
 construct an arbitrary initial mapping ι : params ↦ ℝ
 var C = true
 while(true) {
  let φ_inst be obtained from φ by
    replacing every t ∈ params by ι(t)
  if (φ_inst is unsatisfiable) return ι
  else {
    choose σ such that σ ⊨ φ_inst
    let σ' be ι ⊎ σ
    choose a disjunct d of φ such that σ' ⊨ d
    let C_d be unsatConstraints(d, σ)
    C = C ∧ C_d
    if (C is unsatisfiable) return ∅
    else {
      choose m such that m ⊨ C
      let ι = (m projected onto params) }}}}

def unsatConstraints(d, σ) {
  Partition d into d_uif, d_adt and d_simp
  let d_1 be convertUIF(d_uif, σ)
  let d_2 be convertADT(d_adt, σ)
  let d' be d_simp ∧ d_1 ∧ d_2
  let C_d be applyFarkasLemma(d')
  return C_d }
```

Figure 9. A procedure for finding parameters for formula to make it unsatisfiable

We keep track of the following information in every iteration. (a) A candidate substitution ι for the template variables params, and (b) A non-linear real constraint C containing template variables whose models, when projected onto the template variables, invalidate all the disjuncts explored until the present iteration.

We initially choose an arbitrary assignment of the template variables params as a candidate model ι. We also initialize the constraint C to $true$. We iteratively perform the following steps.

As a first step, we instantiate params using the candidate model ι to obtain a formula ϕ_{inst}. Note that, in our case, ϕ_{inst} is a formula belonging to the theory of *linear integer arithmetic*, uninterpreted functions and algebraic data types. If ϕ_{inst} is unsatisfiable then the mapping ι is the desired instantiation and is returned. Otherwise, there exists a counter-example σ that satisfies ϕ. Note that ϕ_{inst} can be solved using off-the-shelf theory solvers such as Z3.

Given a σ, we construct a model σ' for ϕ by combining the models σ and ι. Note that σ and ι have non-overlapping domains as ι is an assignment for template variables and σ is an assignment for program variables. We pick a disjunct d of ϕ that is satisfied by σ'. This operation can be performed efficiently in time linear in the size of ϕ without explicitly constructing a disjunctive normal form.

Intuitively, the disjunct d corresponds to a *static path* in a finite unrolled version of the function body for which the candidate invariant, obtained by instantiating the template and the postcondition with the mapping ι, is *not inductive*. The disjunct d is analogous to a spurious counter-example path discovered by the counter-example driven abstraction refinement (CEGAR) approaches for program verification. However, counter-example paths discovered by the CEGAR approaches are abstract paths that violate a candidate invariant. The abstract paths are typically used to find concrete paths from which new predicates are discovered. In contrast, the counter-example path that our approach finds is a *static path* (in an unrolled version of the function) for which the candidate invariant is not inductive.

In the subsequent steps, we generate a non-linear real arithmetic constraint C_d that when solved yields an assignment for template variables that invalidates d. The procedure unsatConstraints, discussed in the next section, generates such a constraint. We conjoin the constraints C_d generated in this step with those generated in the earlier iterations. We then look for a satisfying assignment for C. If a model (say m) exists then it invalidates all the disjuncts (including d) that have been explored in this and previous iterations. We make the projection of m onto the template variables the new candidate model and repeat the above process. Note that the candidate invariant corresponding to m will be inductive for the path corresponding to d.

2.5 Generating constraints for invalidating a disjunct

Given a parametric linear disjunct d with uninterpreted functions and algebraic data types, we next present a technique that generates a plain *non-linear* real arithmetic constraint over the parameters of d (which are the template variables) whose solution invalidates d, in other words, the solution models $\neg d$. We first reduce the UIFs and ADTs in the disjunct d to equisatisfiable parametric linear arithmetic formulas. We then apply the existing techniques based on *Farkas' Lemma* to generate constraints that ensure unsatisfiability of d.

Reducing UIFs and ADTs to arithmetic formulas with equality. Given a disjunct d we partition it into three parts: d_{simp}, d_{uif} and d_{adt} having the following properties. d_{simp} is a disjunct that contains only inequalities, equalities and dis-equalities between variables and numerical constants. d_{uif} is a conjunction of atoms of the form $r = f(V)$ and d_{adt} is a conjunction of atoms of the form $r = cons(V)$.

Consider d_{uif}. It can be reduced to an *equisatisfiable* formula ψ_1 belonging to the theory of equalities $\mathcal{T}_{\mathcal{E}}$ using the hierarchical approach described in Sofronie-Stokkermans

[33], where $\psi_1 : \bigwedge\{(V = V') \Rightarrow (r = r') \mid r = f(v), r' = f(V')\}$

Similarly, the ADT part d_{adt} can also be reduced to an equivsatisfiable $\mathcal{T}_{\mathcal{E}}$-formula $\psi_2 : \bigwedge\{(V = V') \iff (r = r') \mid r = c(v), r' = c(V')\}$. The formula $\psi' : d_{simp} \wedge \psi_1 \wedge \psi_2$ belongs to the theory of mixed real/integer arithmetic with equality and is equivsatisfiable to d. Moreover, ψ' is also *model preserving* i.e, every model for ψ' can be extended to a model for d and vice-versa. Hence, it suffices to find an assignment of values for the template variables that would make ψ' unsatisfiable since such an assignment will also make d unsatisfiable.

```
1  input :
2   A disjunct d_uif with predicates of the form r = f(V)
3   A satisfying assignment σ of d_uif.
4  output : A linear arithmetic disjunct d_L.
5
6  def convertUIF(d_uif, σ) {
7   var d_L = ⊤
8   for each pair of atoms r = f(V), r' = f(V') in d_uif {
9    if( σ ⊨ (V = V') )
10    d_L = d_L ∧ (r = r')
11   else {
12    Let ⋁_i c_i = ¬(V = V')
13    d_L = d_L ∧ ⋀{c_i | σ ⊨ c_i} }}
14  return d_L }
```

Figure 10. The algorithm that converts uninterpreted function calls to linear arithmetic formulas

Selecting a disjunct of ψ'. Note that, by construction, the formula ψ' defined above is a complex, deeply nested formula with alternating conjunctions and disjunctions. Instead of considering the entire formula at once for invalidation, we choose a single disjunct of ψ' that is satisfied by the counter-example σ (that is a model for ϕ_{inst}) for invalidation. We pick disjuncts $d_1 \in \psi_1$ and $d_2 \in \psi_2$ such that $\sigma \models d_1$ and $\sigma \models d_2$, and construct a disjunct d' of ψ' as $d_{simp} \wedge d_1 \wedge d_2$. The remaining disjuncts in ψ' may be chosen during a subsequent iteration if they are found to violate the inductiveness of the candidate invariant of that iteration.

The algorithms that perform these steps is shown in Figures 10 and 11. Our algorithm does not explicitly construct the formulas ψ_1, ψ_2 and ψ'; instead, it directly finds the disjuncts d_1 and d_2 of ψ_1 and ψ_2 using the counter-example σ.

Invalidating a parametric linear disjunct. Let d' be a parametric linear disjunct with mixed integers and reals of the form $\bigwedge_{i=1}^{N} L_i(a, x) \;\; op \;\; 0$ where, a is a vector of parameters, x is a vector of variables, op is either $\leq$ or $<$ and each $L_i(a, x)$ is a parametric linear term of the form $c_1 x_1 + \cdots + c_m x_m + c_0$ where each coefficients c_i is either a parameter variable or a constant. We generate a sufficient condition C_d for d' to be unsatisfiable by applying Farkas' Lemma and Motzkin's transposition theorem as explained in Rybalchenko and Sofronie-Stokkermans [31] and Colón et al. [8]. The constraint C_d thus generated is a non-linear constraint over the theory of reals, but remains sound for integers.

```
1  input :
2   A disjunct d_adt with predicates of the form r = cons(V)
3   A satisfying assignment σ of d_adt.
4  output : A linear arithmetic disjunct d_L.
5
6  def convertADT(d_adt, σ) {
7   var d_L = ⊤
8   for each pair of atoms r = cons(V), r' = cons(V') in d_adt {
9    if(σ ⊨ (r = r')) d_L = d_L ∧ (V = V') ∧ (r = r')
10   else { Let ⋁_i c_i = ¬(V = V' ∧ r = r')
11          d_L = d_L ∧ ⋀{c_i | σ ⊨ c_i} } }
12  return d_L }
```

Figure 11. The algorithm that converts calls to constructors of algebraic data types to linear arithmetic formulas

The procedure applyFarkasLemma(d') invoked at line 30 generates the non-linear constraints C_d for the disjunct d'. A model for C_d projected onto the template variables will invalidate the disjunct d.

3. Making the Technique Practical

We next present aspects of our algorithm that we implemented as a result of our experience in using it to verify functional programs. They improve the automation of the approach by automatically synthesizing certain classes of templates and algebraic properties, as well as improve the performance of the constraint solving process.

3.1 More Automation through Synthesis of Templates

We implemented a simple type-based strategy for inferring the invariant templates automatically. For each function f, we maintain a set of terms T_f initialized to the arguments and return variables of f. The template for f, at any given instance, is a linear combination of integer valued variables in T_f. If our invariant inference algorithm failed to prove the desired property, we expand the templates by adding additional terms as explained below, and rerun the algorithm. For every function g in the program of arity n, we apply g to every combination of n terms in T_f that are type compatible with the arguments of g. We also ensure that the template generated for a function does not invoke the function itself recursively. In our empirical evaluation, we find that this approach is useful for inferring invariants that do not have nested function applications (see Section 4)

3.2 Utilizing Common Axiomatic Specifications

We enhance our approach to allow users to specify axiom schemas for functions that are local theory extensions such as those discussed in Jacobs and Kuncak [20] . Congruence,

injectivity and monotonicity are some examples of such axioms. Note that injectivity is implicity assumed for ADT constructors. Our implementation allows specifying functions with integer valued arguments and return values as monotonic.

We instantiate the axioms (such as monotonicity) while constructing the verification condition. For every pair of calls $r = f(V)$, $r' = f(V')$ in a verification condition, to a function f that is specified as monotonic, we create new formulas $(V < V' \Rightarrow r < r') \wedge (V' < V \Rightarrow r' < r)$ and conjoin it with the verification condition. The counter-example driven algorithm presented earlier incrementally explores the parts of the axioms that are required for solving the templates.

This property enhances the ability of the tool to prove non-linear properties. Two of our benchmarks, namely, *Leftist heaps* and *Concat variations* benefited from the monotonicity axioms.

3.3 Simplifying Non-linear Constraints using Quantifier Elimination

As described in section 2.5, our approach, iteratively, applies Farkas' lemma to a parametric linear disjunct d' and generates a set of non-linear constraints for invalidating d'. We present an optimization that *simplifies* the disjunct d' to reduce the complexity of the generated non-linear constraint. We split d into two parts: a linear part d_{li} in which every atom is a linear relation over the program variables, and a non-linear part d_{nl} which is a non-linear formula over template and program variables. Typically, d_{li} corresponds to a long unrolled path in the program containing several fresh variables introduced during the verification condition generation. For instance, conversion of programs to formulas, flattening of functions, conversion of *field selectors* to match constructs, introduce several fresh variables. Most of these variables do not appear in d_{nl}. Hence, we eliminate the variables in d_{li} that do not appear in d_{nl} by applying basic quantifier elimination rules of Presburger arithmetic (Oppen [25]). In particular, we apply the one-point rule that uses equalities to eliminate variables and the rule that eliminates atoms with variables for which only upper or lower bounds exist. Our evaluation shows that this optimization drastically reduces the sizes of formulas (see Section 4).

3.4 Prioritizing Disjunct Exploration using Timeouts

The non-linear Farkas' constraints that we generate for invalidating disjuncts in the verification condition consume varying amounts of time for solving, even for formulas of comparable sizes. To side-step this problem, in our implementation, we abort the non-linear constraint solving process after a predefined time-out and force the algorithm to explore a different disjunct by blocking the counter-example σ in the VC that led to the discovery of the disjunct.

4. Empirical Evaluation

We have implemented our invariant inference on top of the Leon verifier for Scala (Blanc et al. [5]), building on the release from the GitHub repository. We expect that our techniques would be a useful addition to other verification systems as well. We evaluate our implementation on a set of benchmarks written in a purely functional subset of Scala programming language. We consider two classes of benchmarks. In the first class, we use our tool to infer symbolic bounds on the execution time of the programs when most, but not all, correctness invariants were manually provided and the templates for symbolic bounds were specified. In the second class of evaluation benchmarks, we use our tool to prove the correctness properties completely automatically by inferring the necessary invariants for the intermediate recursive functions. The invariants were inferred using templates automatically enumerated by the tool as described in section 3. All experiments were performed on a machine with 8 core, 3.5 GHz, intel i7 processor, having 16GB RAM, running Ubuntu operating system. For solving, linear/non-linear constraints generated by tool we use the Z3 solver of de Moura and Bjørner [13], version 4.3.

Figure 12 summarizes the results of applying the tool for symbolic running time bound inference. The figure shows the benchmarks chosen for evaluation. Together the benchmarks had 718 lines of Scala code. For brevity, we present the templates and its solution only for a few functions in the benchmarks though our tool solved templates of several other functions in the program. *ListOps* and *TreeOps* are a collection of standard list and tree operations that includes operations like *distinct* that removes duplicates, *add all* and *remove all*. PropLogic is a set of propositional logic operations. *DEqueue* is a two list, efficient, implementation of a doubly ended queue with operations like *enqueue*, *dequeue* and *pop*. *Leftist Heap* is an leftist heap data structure implementation with *merge*, *insert* and *remove-max* operations. *Binary Trie* is binary prefix tree with operations like *insert* that inserts a sequence of input bits into the tree, *find* and *create* that creates a new tree form an input sequence. *Concat Vars.* compares two different strategies for sequence concatenation.

The column *iter.* shows the number of calls made to the procedure solveUNSAT (summed up over the analysis of all functions). This corresponds to the total number of disjuncts explored by the tool. The columns *UIF+ADT* and *Ctrs.* show the per iteration averages of the number of UIF/ADT-Constructor calls in the verification condition, and the size of the non-linear constraint solved in each iteration (i.e, the size of C in solveUNSAT algorithm), respectively. The number of compatible pairs of UIF/ADT calls, that need to be considered for reduction, were in the order of several thousands in each iteration.

The column *elim.%* is the average percentage of atoms in the parametric linear disjunct d' of algorithm solveUNSAT

benchmark	*template*	*inferred invariant*	*time(s)*	*iter*	*Averages per iteration*		
					UIF + ADT	*Ctrs*	*elim.%*
Sorting			9.2	37	78	110	49
sortedIns	**time**≤a∗size(l)+b	**time**≤8∗size(l)+5					
Ins. sort	**time**≤a∗mul(size(l),size(l))+b	**time**≤9∗mul(size(l),size(l))+3					
partition	**time**≤a∗size(l)+b	**time**≤20∗size(l)+8					
qsort	**time**≤a∗mul(size(l),size(l))+b	*failed*					
ListOps			14.6	64	72	120	47
reverse	**time**≤a∗size(l)+b	**time**≤6∗size(l)+4					
reverse2	**time**≤a∗mul(size(l),size(l))+b	**time**≤6∗mul(size(l),size(l))+4					
distinct	**time**≤a∗mul(size(l),size(l))+b	**time**≤10∗mul(size(l),size(l))+7					
TreeOps			3.8	46	17	130	67
remove	**time**≤a∗height(t)+b	**time**≤30∗height(t)+15					
PropLogic			14.5	19	263	204	47
simplify	**time**≤a∗size(f)+b	**time**≤16∗size(f)+1					
DEqueue			31.4	71	109	92	68
dequeue	**time**≤a∗qsize(q)+b	**time**≤11∗qsize(q)+63					
Leftist Heap			41	19	367	186	73
merge	**time**≤a∗rh(h1)+b∗rh(h2)+c	**time**≤31∗rh(h1)+31∗rh(h2)+5					
size	twopow(rh(t))≤a∗**res** + b	twopow(rh(t))≤4∗**res** + 2					
Binary Trie			8.9	19	215	30	89
insert	**time**≤a∗lsize(inp) + c	**time**≤24∗lsize(inp) + 3					
find	**time**≤a∗lsize(inp) + c	**time**≤19∗lsize(inp) + 1					
Concat.Vars.			13.2	34	78	115	39
f_worst	**time**≤a∗mul(mul(n,m),m+1) +b∗n+c∗m+d	**time**≤9∗mul(mul(n,m),m+1) +3∗n+80∗m+4					
f_good	**time**≤a∗mul(n,m)+b∗n+c∗m+d	**time**≤37∗mul(n,m)+1∗n+9∗m+3					
Red-black Tree			210.4	44	904	192	85
ins	**time**≤a∗bh(t)+b	**time**≤136∗bh(t)+67					
add	**time**≤a∗bh(t)+b	**time**≤161∗bh(t)+79					
size	twopow(bh(t))≤a∗**res** + b	twopow(bh(t))≤4∗**res** + 2					

Figure 12. Results of applying our approach for inferring symbolic running time bounds. In the figure, mul stands for the multiplication operation expressed as a recursive user-defined function

eliminated by our quantifier elimination optimization (discussed in section 3). These atoms if not eliminated contribute to the size of the final non-linear constraint C approximately by the same amount (and will also introduce an equal number of unknown variables).The result shows that the optimization reduces the sizes of the disjuncts, on average, by more than 50% (sometimes upto 80%) which is very significant considering the fact that even the state-of-the-art non-linear constraint solvers do not scale well to large formulas.

The tool was able to solve logarithmic bounds for two benchmarks: Leftist heap and Red-black tree, and quadratic bounds for several benchmarks, defined over recursive functions manipulating ADTs. In fact, the tool solved a cubic bound for *Concat variantions* that compares two implementations of list concatenations, one that performs concatenation at the end and other at the beginning of the lists. The one that appends to the beginning was found to be more efficient by an order of magnitude. As another example, the ListOps benchmark has two implementations of *reverse*, one more efficient than other, a difference that the analysis successfully establishes. For *Binary Trie* the tool proved that the time taken for insert and find operations is in order of size of the input and independent of the dimensions of the tree.

Figure 13 summarizes the results of applying the tool for proving correctness properties automatically by enumerating templates. In the figure, the invariants inferred by the tool are marked with †. We present the unsimplified form of the inferred invariants to given an idea of the template enumerated by the tool to infer the invariants. We only show the template for some important functions though the tool inferred appropriate templates for *all* transitively called functions. Note that all of our benchmarks have several deeply nested functions. For instance, for the Leftist heap implementation, the property required to be proven is that the size of the heap

Benchmark	*Property, Inferred Invariants and Analysis time*
insertion sort	(7.4s)
sort	size(l)==size(**res**)
sortedIns	0∗e+1∗size(l)+(−1∗size(**res**))+1)==0) †
ListOps	(5.4s)
reverse	size(l)==size(**res**)
append	1∗size(l2)+1∗size(l1)+(−1∗size(**res**))+0)==0) †
TreeOps	(74s)
addAll	size(**res**)≥size(t)
insert	(0∗elem+1∗size1(t)+(−1∗size(**res**))+0)≤0) †
removeAll	size(**res**)≤size(t)
remove	(0∗elem+(−1∗size1(t))+1∗size(**res**)+0)≤0) †
DEqueue	(6.34s)
pop	qsize(**res**)==qsize(q) − 1
removeLast	(1∗size(l))+(−1∗size(**res**))+−1)==0) †
Binary Trie	(24.4s)
create	ht(**res**)≥lsize(inp)
insert	((−2 ∗ ht(t))+(−2∗ht(**res**)) †
	+2∗lsize(inp) + −1)≤0) †
Leftist Heap	(587.2s)
removeMax	size(**res**)≥size(h)−1
merge	(−1∗size(h1))+0∗rh(h2)+(−1∗size(h2)) †
	+1∗size(**res**)+0∗rh(h1)+0∗rh(**res**)+0)==0) †

Figure 13. Fully automatically proving correctness properties by inferring and solving templates. Invariants marked with † are inferred fully automatically by generating template first and then finding its parameters. The remaining invariants are the desired properties requested by the user.

decreases after *removeMax*. *removeMax* transitively invokes *merge* and other functions for which no template or invariants were specified. The tool inferred an invariant for *merge* for establishing this property (see Figure 4 for the source code of *merge*). For the *Binary Trie* example, the tool was able to completely automatically prove that the height of a new tree created from a sequence is at least as long as the sequence.

5. Related Work

We highlight the most direct influences among a large body of related work. The work of Cook et al. [9] is the closest to ours because it also performs template-based analysis of resource bounds and handles program paths efficiently, for which it uses the idea of path invariants from Beyer et al. [4]. Further improvements of performance of constraint solving can be done through combination with other auxiliary analyses as in Gupta et al. [19]; our technique would also benefit from such orthogonal integration in some cases. Analyses for imperative code mostly rely on a separate tool for shape analysis of low-level code. Our programs use algebraic data types, so the type system provides a reliable starting point for precise abstraction. The technique that our algorithm uses to eliminate uninterpreted functions and algebraic data types to obtain purely numerical constraints is related to techniques of Beyer et al. [3] and Sofronie-Stokkermans [33], but they use an explicit algorithm instead of an SMT solver to handle the disjunctions arising from axiomatizing function symbols. Our approach uses counterexamples to drive the elimination of UIFs and ADTs as described in Figures 10 and 11. This extension allows us to scale to formulas with thousands of UIFs and ADTs by exploring a small fraction of the formulas (see section 4). The difficulty of verifying higher-order programs is reflected in very small size of the benchmarks that most of the existing approaches can handle. A notable advance are liquid types and abstract refinement types of Vazou et al. [37], which have been applied to realistic functional programming libraries. The refinement types approach can prove correctness of functional programs, but uses syntactic templates and does not perform constraint-based analysis to infer template parameters. Our implementation is built on top of a Leon verifier. However, using Leon directly to verify these programs requires manually coming up with k-inductive invariants. Recent work of Beyene et al. [2] provides a very general framework and system for infering 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 functional programs whereas Beyene et al. [2] targets temporal logic formulas applied to operating systems code. The underlying HSF tool of Grebenshchikov et al. [16] has been shown applicable to a wide range of analysis problems; it may also be promising in our domain in the future, given appropriate translator from Scala to Horn clauses. Counterexample-guided refinement ideas are ubiquitous in verification, as well as in software synthesis, where they are used in counterexample-guided inductive synthesis algorithms by Solar-Lezama et al. [34], Gulwani et al. [18], and Kneuss et al. [22]. One important difference in approaches such as ours, as well as in path invariants Beyer et al. [4], is that an infinite family of counterexamples is eliminated at once. The idea of using Farkas lemma to infer coefficients of linear templates goes back at least to the work of Colón et al. [8] and has been generalized in different directions by Sankaranarayanan et al. [32] and Cousot [10]. Work that verifies correctness properties for imperative programs in less automated way includes that of Qiu et al. [27] as well as system VCC (Cohen et al. [7]). Inference of loop invariants for such programs is a very difficult topic and has been explored by, e.g., Podelski and Wies [26] and Lev-Ami et al. [24]. Other lines of work directly aim to address resource bounds. Among the successful systems is SPEED of Gulwani et al. [17], as well as COSTA system of Albert et al. [1] that can solve recurrence equations and infer non-linear bounds. For domain-specific languages the cost estimation problem has also been addressed in the context of database algorithm synthesis (Spielmann et al.

[35]). Techniques to derive quantitative invariants on data structures have a long tradition and mostly followed an abstract interpretation methodology (Cousot and Cousot [11]) in which target properties are not used to guide the analysis. Notable examples include work of Deutsch [14] that derives symbolic bounds on heap paths and Rugina and Rinard [29, 30] that also deals with array structures. More recent works in this line include those of Rugina [28], Cousot et al. [12], Gopan et al. [15], Bouajjani et al. [6].

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] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, 2011.

[7] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, 2009.

[8] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.

[9] B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis. Finding heap-bounds for hardware synthesis. In *FMCAD*, 2009.

[10] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, 2005.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[12] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, 2011.

[13] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[14] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In *PLDI*, 1994.

[15] D. Gopan, F. DiMaio, N. Dor, T. W. Reps, and S. Sagiv. Numeric domains with summarized dimensions. In *TACAS*, 2004.

[16] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[17] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[18] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[19] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. *STTT*, 15(4):291–303, 2013.

[20] S. Jacobs and V. Kuncak. Towards complete reasoning about axiomatic specifications. In *VMCAI*, 2011

[21] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000

[22] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.

[23] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[24] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. Software Testing and Analysis*, 2000

[25] D. C. Oppen. Elementary bounds for presburger arithmetic. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, 1973

[26] A. Podelski and T. Wies. Counterexample-guided focus. In *POPL*, 2010.

[27] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, 2013.

[28] R. Rugina. Shape analysis quantitative shape analysis. In *SAS*, 2004.

[29] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI*, 2000.

[30] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235, 2005.

[31] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, 2007

[32] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *POPL*, 2004. ISBN 1-58113-729-X.

[33] V. Sofronie-Stokkermans. Hierarchical and modular reasoning in complex theories: The case of local theory extensions. In *FroCoS*, 2007.

[34] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[35] A. Spielmann, A. Nötzli, C. Koch, V. Kuncak, and Y. Klonatos. Automatic synthesis of out-of-core algorithms. In *SIGMOD*, 2013

[36] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, 2011.

[37] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.