Solving Constrained Horn Clauses using Interpolation

MSR-TR-2013-6

Kenneth L. McMillan
Micrsoft Research

Andrey Rybalchenko
Technische Universität München

Abstract

We present an interpolation-based method for symbolically solving systems of constrained Horn clauses. The method can be used to solve for unknown predicates in the verification conditions of programs. Thus, it has a variety of applications, including including model checking of recursive and threaded programs. The method is implemented in tool called Duality, which we evaluate using device driver verification benchmarks.

1 Introduction

Many problems of inference in program verification can be reduced to solving a collection of logocal constraints, in the form of Horn clauses [12, 7]. In Hoare-style program verification, we apply the rules of a program logic to obtain purely logical proof subgoals whose validity implies correctness of a program. These subgoals are called the *verification conditions* or VC's. The VC's generally contain auxiliary predicates such as inductive invariants, that must be *inferred*. Leaving these auxiliary predicates undefined (that is, as symbolic constants) the problem of inference becomes a problem of *solving* for unknown predicates or relations satisfying a set of logical constraints. In the simplest case, this is an SMT (satisfiability modulo theories) problem.

For example, consider the following simple program fragment:

var i : int := 0;
while $i < N$ **invariant** $R(i)$ **do**
    $i = i + 2$;
done
assert $i \neq 41$;

Here, $R(i)$ is an unknown invariant of the loop. The VC's of this program are the following three logical formulas:

$$i = 0 \quad \Rightarrow \quad R(i) \tag{1}$$
$$R(i) \wedge i < N \quad \Rightarrow \quad R(i+2) \tag{2}$$
$$R(i) \wedge \neg(i < N) \quad \Rightarrow \quad i \neq 41 \tag{3}$$

These are just the three proof obligations of the Hoare logic rule for while loops, reduced to logic. They say, respectively, that invariant $R(i)$ is initially true, that it is preserved by an iteration of the loop, and that it implies the post-condition of the loop on exit. Note in particular that $R(i+2)$ is just the weakest precondition of $R(i)$ with respect to the loop body.

To prove the program, we must infer a value of the unknown predicate R that makes the VC's true. One such solution for $R(i)$ is $i \equiv 0 \bmod 2$. We observe that satisfiability of the VC's corresponds to proof of the program. Note that this is opposite to the typical case in model checking where satisfiability generally corresponds to a *counterexample*.

One might therefore suppose that solving for R could be accomplished by simply applying an SMT solver. However, because the variables in the VC's are implicitly universally quantified, traditional SMT solvers are not able to produce models for them. Instead, we require specialized methods to solve these formulas [12, 16]. In particular, since we want to handle non-finite state programs, we cannot express R in extension, as a set of tuples. Rather, we need a *symbolic* expression of R as s logical formula.

To solve these systems, we rely on the fact that the VC's for proof systems typically take the form of *constrained Horn clauses* (CHC's). That is, they have

form:

$$\phi \wedge P_1(\vec{x}_1) \wedge \ldots \wedge P_k(\vec{x}_k) \Rightarrow P(\vec{x})$$

where $\vec{x}, \vec{x}_i$ are vectors of (universally quantified) variables, and ϕ is a *constraint* expressed in some background theory.

Quite a variety of proof systems produce VC's in the form of CHC's. Thus, if we can solve CHC's, we can derive program proofs in a variety of proof systems. We have just seen that the VC's for program loops have this form. Now let us consider procedure calls, in the way they are handled by the VC generator for the Boogie programming language [5]. A parameter-less procedure π in Boogie has the following form:

procedure π:
  requires $P(\vec{x})$;
  ensures $Q(\vec{x}_{old}, \vec{x})$;
  σ

Here, formula P is a required precondition to enter the procedure, formula Q is the guaranteed post-condition and statement σ is the body of the procedure. The precondition is over the program variables $\vec{x}$, while the post-condition refers to the program variables $\vec{x}$ and their procedure entry values $\vec{x}_{old}$.

To generate verification conditions, Boogie replaces each call to π with the following code:

assert $P(\vec{x})$;
$\vec{x}_{old} := \vec{x}$;
havoc $\vec{x}$;
assume $Q(\vec{x}_{old}, \vec{x})$;

The havoc statement causes the variables $\vec{x}$ to take a non-deterministic value. Thus, the transformed call to π behaves as any code satisfying the specification of π. Now suppose π is called in the following context:

assert $R(\vec{x})$;
call π;
assert $S(\vec{x})$;

After replacing the call to π, the verification conditions we obtain from the weakest precondition calculus are:

$$\begin{aligned} R(\vec{x}) &\Rightarrow P(\vec{x}) \\ R(\vec{x}_{old}) \wedge Q(\vec{x}_{old}, \vec{x}) &\Rightarrow S(\vec{x}) \end{aligned}$$

Note that these formulas are in CHC form. They differ from the loop VC's, however, in that one VC has two unknown predicates on the left-hand side. We will see that this substantially complicates the problem. The number of unknowns on the left-hand side will be called the *degree* of the clause, a clause of degree one or less will be called *linear*. Generally, intraprocedural proofs

produce linear VC's, while modular proofs of procedures or concurrent threads produce non-linear VC's.

In particular, a very simple non-linear proof system for concurrent programs is given in [13]. We are given a collection of N parallel processes of the form:

process π_i:
  while * **do**
    ρ_i

Each process π_i has a set of local variables $\vec{x}_i$. The processes share a set $\vec{y}$ of global variables. We wish to prove in a modular way that, given some initial condition $\phi_{init}(\vec{x}_1, \ldots, \vec{x}_N, \vec{y})$, some error condition $\phi_{error}(\vec{x}_1, \ldots, \vec{x}_N, \vec{y})$ is never true. The relations to be inferred for each process i are $R_i(\vec{x}_i, \vec{y})$, which is the invariant of the loop, and $E_i(\vec{y}, \vec{y}')$, which is the environment constraint. The verification conditions for process π_i are:

$$\begin{array}{rcll} \phi_{init}(\vec{x}_1, \ldots, \vec{x}_N, \vec{y}) & \Rightarrow & R_i(\vec{x}_i, \vec{y}) & \\ R_i(\vec{x}_i, \vec{y}) \wedge [\rho_i](\vec{x}_i, \vec{y}, \vec{x}_i', \vec{y}') & \Rightarrow & R_i(\vec{x}_i', \vec{y}') & \\ R_i(\vec{x}_i, \vec{y}) \wedge E_i(\vec{y}, \vec{y}') & \Rightarrow & R_i(\vec{x}_i, \vec{y}') & \\ R_i(\vec{x}_i, \vec{y}) \wedge [\rho_i](\vec{x}_i, \vec{y}, \vec{x}_i', \vec{y}') & \Rightarrow & E_j(\vec{y}, \vec{y}') & \text{for } j \neq i \end{array}$$

where $[\rho_i]$ is the transition relation of statement ρ_i. These conditions state respectively that the initial condition implies the loop invariant, the loop body preserves the loop invariant, environment transitions preserve the loop invariant, and the loop body (executed atomically) is an allowed environment transition for other processes. The global correctness condition is:

$$R_1(\vec{x}_1, \vec{y}) \wedge \cdots \wedge R_N(\vec{x}_N, \vec{y}) \quad \Rightarrow \quad \neg\phi_{error}(\vec{x}_1, \ldots, \vec{x}_N, \vec{y})$$

Note again that all these proof sub-goals are CHC's.

A final case is type inference for dependently typed functional programs. For example, the type checking conditions in Liquid Types system [26] reduce to CHC form [7]. Thus, we can in principle use a generic solver for CHC's to infer refinement types in this system. In this way, we could avoid the complex and indirect translation of the problem to procedural program verification used in HMC [18].

In all these cases, given an appropriate VC generator, we can reduce the program proof problem to solving a system of CHC's for the value of unknown predicates. The advantage of this formulation is that it allows many program analysis or verification problems to be handled by a *generic* solver. This solver is independent of any programming language, proof system, or intermediate code representation. By using a logical formulation, we obtain a separation of concerns between encoding the program semantics and inference of proof constructs. Moreover, we obtain re-use at two levels. We can re-use existing VC generators to produce the problems, and we can re-use generic solvers for many inference applications (and thus amortize the effort involved in optimizing these solvers).

The questions remaining are how to actually perform the solving, and whether generic logical solvers can be as efficient as tools specialized to particular inference problems. In this paper, we will approach these questions using an generalization of the IMPACT algorithm [21]. We note that in case of loops and recursion, the VC's are generally cyclic. We solve this problem by finitely *unwinding* the cyclic system into an acyclic one. The acyclic system can be solved using an interpolating SMT solver such as [20, 22, 8]. Unwinding continues until the unwinding contains an *inductive subset*, meaning that a fragment of it can be folded into a solution of the original problem. This is conceptually very much as in IMPACT, which unwinds a cyclic CFG into an abstract reachability tree, or ART [15]. The difference is that, since the constraints are not linear, the unwinding may now be a DAG instead of a tree, which requires more general interpolation techniques. Correspondingly, a *counterexample* has the form of a tree rather than a path, as in IMPACT.

This generalization has several advantages. First, it allows us to apply the IMPACT algorithm to a wider range of problems (including, for example, interpocedural program analysis, as in Whale [2]). Second, it allows great flexibility in encoding the proof inference problem, depending on how we generate the verification conditions. By generating VC's at the level of procedures and loops, we can take advantage of a modern SMT solver's ability to search a large space of possible execution paths efficiently (as in bounded model checking). We also have the ability to easily change the proof rules of the system, for example, using alternative rules for loops to aid convergence, or expressing the unknown proof constructs in a restricted form to exploit domain knowledge we may have about the form of the proof. We will observe experimentally that this additional flexibility does not necessarily entail a cost in performance. That is, the approach is comparable in performance to well-established software model checking techniques based on explicit program control flow representations.

The primary contributions of this work are:

- A purely interpolant-based approach to solving systems of CHC's symbolically.
- Extending the range of such solvers to include integer arithmetic, arrays and quantified axioms.
- Generalizing the form of the problem to allow loop- and procedure-scale VC's.
- An implementation and experimental evaluation on real industrial problems.

1.1 Related work

There are two existing approaches to solving CHC's. Both are restricted to rational linear arithmetic constraints. The tool QARMC is based on predicate abstraction and uses Craig interpolation for abstraction refinement [12]. On

the other hand, Bjorner and Höder apply property-driven reachability analysis (PDR) to solve similar systems [16]. Both of these systems can be thought of as generalizing existing program analysis techniques at the purely logical level.

In this paper, we generalize a different program analysis technique, based on computing inductive invariants directly from interpolants. We obtain several potential advantages in this way. First, we can handle a broader class of VC's (where the left-hand sides are not conjunctions) allowing us to exploit the efficiency of modern SMT solvers to handle large blocks of code. Second, the methods allows us to exploit any logics and theories that an interpolating prover supports, including quantifiers, integer arithmetic and arrays. We will see these advantages in our experiments, as they allow the VC's generated by Boogie to be handled directly, without further translation or reduction. Finally, the method is purely interpolation-based, and does not rely on any method of quantifier elimination.

Constrained Horn clauses can also be solved by abstract interpretation, as in μZ [17]. However, this requires an abstract domain to be given, along with abstract relational transformers. Here, we do not assume a user-provided abstract domain. A related problem for finite universes is Datalog query answering. A tool using symbolic model checking methods for this purpose is BDDBDDB [28]. Here, we do not restrict to the finite state case. We use first order logic to express the problem, and compute a solution expressed symbolically in first-order logic.

There are also closely related algorithms that are specialized to particular program analysis or model checking problems. One is, of course, IMPACT. Algorithmically, the primary difference here is the extension to the non-linear case, covering interprocedural program verification and related problems. A generalization of IMPACT to the interprocedural case is WHALE. Algorithmically, this is closely related to the present method and also differs in significant ways. The similarities and differences will be discussed in detail in Section 5.1. In that section we will also consider the relation to various unwinding-based techniques, including bounded model checking techniques.

1.2 Notations

We use standard first-order logic over a signature Σ of function and predicate symbols of defined arities. Generally, we use ϕ, ψ for formulas, a, b, c for constants (nullary function symbols), f, g for functions, P, Q, R for predicates, x, y, z for individual variables, t, u for terms. We will use $\vec{x}$ for a vector of variables, $\vec{t}$ for a vector of terms, and so on. When we say a formula is *valid* or *satisfiable*, this is relative to a fixed background theory $\mathcal{T}$. A subset of the signature $\Sigma_I \subseteq \Sigma$ is considered to be *interpreted* by the theory. The *vocabulary* of a formula ϕ is the subset of $\Sigma \setminus \Sigma_I$ occurring in it, and is denoted $L(\phi)$. An *interpolant* for $A \wedge B$ is a formula I such that $A \Rightarrow I$ and $B \Rightarrow \neg I$ are valid, and $L(I) \subseteq L(A) \cap L(B)$. We will write $\phi[X]$ and $t[X]$ for a formula or term with free variables X. If P is a predicate symbol, we will say that a P-fact is a formula of the form $P(t_1, \ldots, t_n)$. A formula or term is *ground* if it contains no

variables. If $\mathcal{R}$ is a set of symbols, we say ϕ is $\mathcal{R}$-free when $L(\phi) \cap R = \emptyset$.

2 The relational post-fixed point problem

We now give a precise definition of the problem to be solved.

Definition 1 *A* constrained Horn clause *(*CHC*) over a vocabulary of predicate symbols $\mathcal{R}$ is a formula of the form $\forall X.B[X] \Rightarrow H[X]$ where*

- *The* head $H[X]$ *is a P-fact, for $P \in \mathcal{R}$, or is $\mathcal{R}$-free, and*
- *The* body $B[X]$ *is a formula of the form $\exists Y.\ \phi \wedge \psi_1 \wedge \cdots \psi_k$ where ϕ is $\mathcal{R}$-free and ψ_i is a P-fact for some $P \in \mathcal{R}$.*

The clause is called a query *if the head is $\mathcal{R}$-free, else a* rule*. A rule with body* TRUE *is a* fact*. The formula ϕ is called the* constraint *of the CHC.*

Observe that the existential quantifiers in the body $B[X]$ of a CHC can be converted to universal quantifiers in prenex form. Thus, $\forall X.(\exists Y.\ B[X,Y]) \Rightarrow H[X]$ can be rewritten as $\forall X, Y.\ B[X,Y] \Rightarrow H[X]$. To avoid clutter, we will generally leave the universal quantifiers implicit and write simply $B[X,Y] \Rightarrow H[X]$. If the head of clause C is a P-fact, we will write $\mathrm{hd}(C) = P$.

Also, we assume that our theory $\mathcal{T}$ includes the standard interpretation of equality on terms. This means that we can restrict our attention to P-facts of the form $P(\vec{x})$, where x is a vector of distinct variables. For example, a rule such as $B[X] \Rightarrow P(f(x))$ can be rewritten to $B[X] \wedge y = f(x) \Rightarrow P(y)$, where y is a fresh variable.

Definition 2 *A* relational post-fixed point problem *(RPFP) is a pair $(\mathcal{R}, \mathcal{C})$, where $\mathcal{R}$ is a set of predicate symbols and $\mathcal{C}$ is a set of CHC's over $\mathcal{R}$.*

We will refer to $\Sigma \setminus \mathcal{R}$ as the *background vocabulary* and assume that this coincides with the interpreted vocabulary Σ_I. A *symbolic relation* is a term of the form $\lambda\vec{x}.\ \phi[\vec{x}]$, such that $L(\phi) \subseteq \Sigma \setminus \mathcal{R}$ (that is, ϕ is over the background vocabulary). A *symbolic relational interpretation* σ over $\mathcal{R}$ is a map from symbols in $\mathcal{R}$ to symbolic relations of the appropriate arity. If ψ is a formula, we write $\psi\sigma$ to denote ψ with $\sigma(R)$ substituted for each $R \in \mathcal{R}$ and β-reduction applied. For example, if $\psi = R(a,b)$ and $\sigma(R) = \lambda x, y.\ x < y$, then $\psi\sigma$ is $a < b$.

Definition 3 *A* solution *of RPFP $(\mathcal{R}, \mathcal{C})$ is a symbolic relational interpretation σ over $\mathcal{R}$ such that, for all $C \in \mathcal{C}$, $C\sigma$ is valid.*

A subtle point worth noting here is that solution of an RPFP depends on the interpretation of the background symbols. If the background theory is complete (meaning it has a unique model up to isomorphism) then this gives a unique interpretation of $\mathcal{R}$. We can therefore think of an RPFP as a special case of SMT. If T is incomplete, however (say it includes uninterpreted function symbols) then the solution effectively gives one interpretation of $\mathcal{R}$ for each

theory model. This allows us to leave aspects of the program semantics (say, the heap model) incompletely defined, yet still prove the program correct.

We can also give a semantic notion of solution as follows:

Definition 4 *A* semantic solution *of RPFP* $(\mathcal{R}, \mathcal{C})$ *is a map* σ *from models of* $\mathcal{T}$ *to relational interpretations of* $\mathcal{R}$*, such that for every model* M *of* $\mathcal{T}$*,* $M, \sigma \models \mathcal{C}$*.*

The notions of semantic solution and symbolic solution do not necessarily coincide, as some semantic solutions may not be definable in the logic. If every RPFP with a semantic solution also has a symbolic solution, we could say that our theory is relatively complete for RPFP's (*i.e.*, we have a complete proof system relative to an oracle for the theory). In what follows, however, we do not require relative completeness. In cases where a symbolic solution does not exist, our solution algorithms will simply not terminate.

2.1 Solving RPFP's with predicate abstraction

As an introduction to the Duality algorithm, we will consider first the more familiar approach of predicate abstraction [11], applied to CHC solving. That is, given a set $\mathcal{P}$ of atomic predicates, we synthesize the strongest map from $\mathcal{R}$ to Boolean combinations of $\mathcal{P}$ that solves the problem. Alternatively, we can use a Cartesian predicate abstraction that allows only cubes over $\mathcal{P}$ (that is, conjunctions of literals). This is the approach taken by QARMC [12].

The difficulty in this approach is mainly to discover adequate predicates $\mathcal{P}$. QARMC does this by constructing a *derivation tree*. The derivation tree describes a potential *counterexample*, that is, a proof that the RPFP has no solution.

As an example, consider the following RPFP:

$$x = y \quad \Rightarrow \quad P(x, y) \tag{4}$$

$$P(x, y) \wedge z = y + 1 \quad \Rightarrow \quad P(x, z) \tag{5}$$

$$P(x, y) \wedge P(y, z) \quad \Rightarrow \quad Q(x, z) \tag{6}$$

$$Q(x, z) \quad \Rightarrow \quad x \leq z \tag{7}$$

We can think of these formulas as the VC's of a program with two procedures, P and Q. Procedure P is recursive and either returns its input (4) or calls itself and returns the result plus one (5). Procedure Q calls P twice (6). The query represents a specification that Q is non-decreasing (6).

To build a derivation tree that acts as a counterexample to our query, we start with the negation of the query, that is, $Q(x, z) \wedge \neg(x \leq z)$, as the root of the tree (see Figure 1(a)). Now we proceed to derive P-facts by unifying them with the heads of rules. For example, to derive $Q(x, z)$, we can use rule (6). Unifying the head of this rule with $Q(x, z)$, we obtain $P(x, y) \wedge P(y, z) \Rightarrow Q(x, z)$. Now, using rule (4), we can derive $P(x, y)$ by $x = y \Rightarrow P(x, y)$. Using rule (5), we derive $P(y, z)$ from $P(y, y') \wedge z = y' + 1 \Rightarrow P(y, z)$. Note we replaced y in (5) with a fresh variable y' to avoid a name clash. We continue with this process

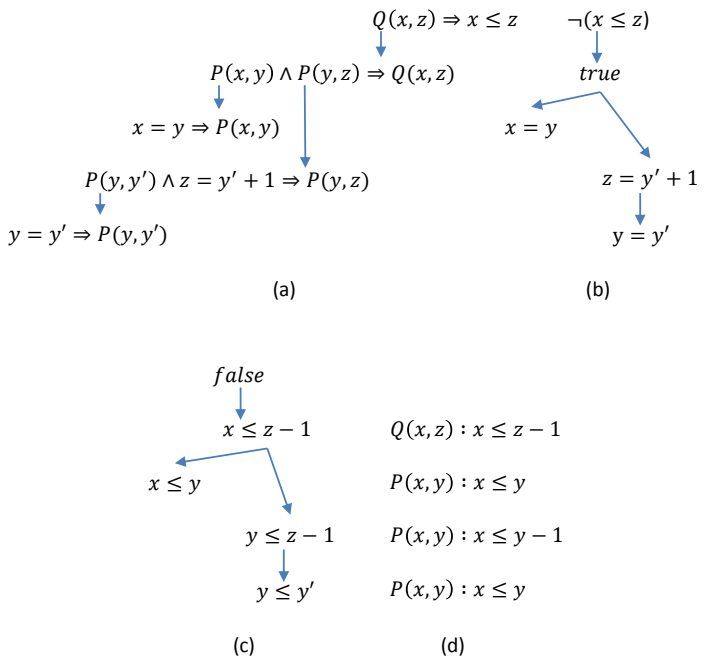

Figure 1: Predicate refinement by interpolation. (a) Derivation tree. Arrows represent unification steps. (b) Constraint tree. (c) Interpolant for constraint tree. (d) Extracted predicates.

until all P-facts in the tree have derivations (which means the leaves of the tree are facts). One possible completed derivation tree for our example is shown in Figure 1(a).

Now consider the non-root formulas in our derivation tree. Any satisfying assignment of the constraints in these formulas gives us a proof of certain ground facts. In our example, the non-root constraints are $x = y$, $z = y' + 1$ and $y = y'$ (see Figure 1(b)). One satisfying assignment is $x = y = y' = 1, z = 2$. Plugging this assignment in to the derivation tree, we obtain the following derivation:

$$\begin{aligned} & & P(1,1) \\ P(1,1) & \Rightarrow & P(1,2) \\ P(1,1) \wedge P(1,2) & \Rightarrow & Q(1,2) \end{aligned}$$

(note $P(1,1)$ is derived twice). We say that each of these facts is *derivable* by the corresponding sub-tree of the derivation.

Notice, however, that the fact $Q(1,2)$ is inconsistent with our root goal $Q(x,z) \wedge \neg(x \leq z)$. In fact, when we add the root constraint $\neg(x \leq z)$ our system of constraints becomes unsatisfiable. This means that our derivation tree cannot derive any facts that refute the desired property. We can obtain a certificate of this fact by computing a *Craig interpolant* for the constraint

tree. Figure 1(c) shows an interpolant for our example. The interpolant assigns a formula to each node of the constraint tree. This formula is only over the variables representing the fact being derived at that node (these are exactly the common ones between the sub-tree rooted at that node and the rest of the tree). Moreover, each node together with its children's interpolants implies its own interpolant, and the root node (the negation of the query) is contradicted. Thus, the interpolant constitutes a bottom-up proof that our derivation tree cannot provide a counterexample to the query.

In QARMC, atomic predicates used in the interpolants are then added as new abstraction predicates in $\mathcal{P}$. For our example, these predicates are shown in Figure 1(d). If $\mathcal{P}$ was $\{Q(x,z) : \ x \leq z\}$, we can now solve the problem, for example by:

$$\begin{aligned} Q(x,z) &\equiv x \leq z \\ P(x,y) &\equiv x \leq y \end{aligned}$$

When constructing a derivation tree, QARMC uses the results of the last predicate abstraction run to build a tree that is not already ruled out by the existing predicates. After adding the new predicates, we are guaranteed that at least the one derivation tree is ruled out in future runs.

2.2 Derivations and interpolation

We now make the above notions more formal. We have written a rule in the form

$$\phi \wedge P_1(\vec{x}_1) \wedge \cdots \wedge P_k(\vec{x}_k) \Rightarrow P(\vec{x}).$$

An alternative view, however, is that the CHC represents a set of ground implications satisfying the constraint ϕ. This view is equivalent for our purposes, since the Herbrand theorem tells us that a first-order formula has a model exactly when it has a model constructed from ground terms. We represent this set of ground implications as follows:

$$P_1(\vec{x}_1), \ldots, P_k(\vec{x}_k) \rightarrow P(\vec{x}) \mid \phi$$

This stands for the set of ground implications

$$P_1(\vec{x}_1)\sigma, \ldots, P_k(\vec{x}_k)\sigma \rightarrow P(\vec{x})\sigma$$

where σ is a ground substitution and $\phi\sigma$ is true.

We can then define a *derivation tree* for a set of CHC's $\mathcal{C}$ as a derivation using the following inference rule:

$$\frac{\begin{array}{l} P_1(\vec{x}_1) \mid \psi_1 \\ \ldots \\ P_k(\vec{x}_k) \mid \psi_k \\ P_1(\vec{x}_1), \ldots, P_k(\vec{x}_k) \rightarrow P(\vec{x}) \mid \phi \quad (*) \end{array}}{P(\vec{x}) \mid \phi \wedge \psi_1 \wedge \cdots \wedge \psi_k}$$

where $(*)$ is a rule in $\mathcal{C}$, possibly with variables renamed. The rule says simply that if we can derive the antecedents of an implication, we can derive the consequent (and in fact it corresponds to k instances of the standard resolution rule). Notice that at the leaves of a derivation tree we must have facts in $\mathcal{C}$ (rules with no antecedents).

In any instance of this inference rule, we will call ϕ the *local constraint* and $\gamma = \phi \wedge \psi_1 \wedge \cdots \wedge \psi_k$ the global constraint. The conclusion of a derivation tree is of the form $P(\vec{x}) \mid \gamma$. We can think of this as a set of ground facts $P(\vec{x})\sigma$ where $\gamma\sigma$ is true.

A derivation tree is *well-formed* if in every inference, for each ψ_i, $V(\psi_i) \cap V(\gamma) \subseteq V(\vec{x}_i)$. That is, the variables that each subtree has in common with the remainder of the tree are just the parameters of its conclusion. This can always be achieved by simply renaming variables in the tree (which corresponds to the "occurs check" in Prolog). From here on we assume all derivation trees are well-formed.

To create a derivation of a failure of a query, we introduce an imaginary predicate F corresponding to failure. We then treat a query $Q(\vec{x}) \Rightarrow \psi$ as a as rule $Q(\vec{x}) \rightarrow F \mid \neg\psi$. The query fails when we can derive F using this rule. Thus we say:

Definition 5 *A* failure derivation *for a query Q of the form $\phi \wedge P_1(\vec{x_1}) \wedge \cdots \wedge P_k(\vec{x}) \Rightarrow \psi$ in an RPFP is a derivation of F using an additional* failure clause*:*

$$P_1(\vec{x_1}), \ldots, P_k(\vec{x}) \rightarrow F \mid \phi \wedge \neg\psi$$

Now suppose that the global constraint γ of a derivation tree is unsatisfiable. In this case, the tree derives no facts, and we call it *vacuous*. For a vacuous derivation tree, we can compute an *interpolant*:

Definition 6 *An* interpolant *for a derivation tree T is a map I from inferences in T to formulas such that, for every inference rule application n in T with conclusion $P(\vec{x}) \mid \gamma$, premises $m_1, \ldots m_k$ and local constraint ϕ,*

1. *$V(I(n)) \subseteq V(\vec{x})$, and*
2. *$\models \phi \wedge I(m_1) \wedge \cdots \wedge I(m_k) \Rightarrow I(n)$, where $\models$ is the satisfaction relation, and*
3. *if n is the root of T then $I(n) =$ False.*

By induction on the tree we have $\gamma(n) \Rightarrow I(n)$, thus existence of an interpolant implies a derivation tree is vacuous. We can also show the following:

Theorem 1 *If theory $\mathcal{T}$ admits Craig interpolation, then every vacuous derivation tree has an interpolant.*

Proof By induction on the tree height, using the Craig property and well-formedness of the derivation. Consider the root inference of the tree. We know

ψ_1 is inconsistent with $\phi, \psi_2, \ldots \psi_k$, and since the derivation is well-formed, that the common vocabulary is $V(\vec{x}_i)$. Thus, there is a Craig interpolant for this pair over $\vec{x}$. Call it ι_1. Replacing ψ_1 with ι_1, we proceed to compute and interpolant ι_2 for ψ_2 and so on. For each premise m_i we set the interpolant $I(m_i) = \iota_i$. We compute interpolants for each subtree by setting ϕ_i to $\phi_i \wedge \neg\iota_i$. The result is an interpolant for the derivation. □

If a derivation tree is vacuous, we can compute an interpolant for it using an interpolating theorem prover. The proof of Theorem 1 shows one way to do this. A more efficient approach is to extend the interpolating prover to compute all the interpolant formulas in the tree at once. That is, we gather the local constraints into a tree as in Figure 1(c). From a single refutation proof of these constraints, the prover can derive interpolant formulas for every node of the tree. This is done in QARMC for the simple case of conjunctions of linear constraints over the reals. The interpolating version of the SMT solver Z3 [22] has also been extended to do this for the first-order theory of uninterpreted functions, arrays and linear arithmetic (AUFLIA).

3 DAG-like problems

Instead of applying predicate abstraction, the Duality algorithm constructs a solution to an RPFP (and thus a program proof) directly from the interpolants for its failure derivations. This is possible for a subclass of problems we will call *DAG-like*.

For a given a set of CHC's, we will say that predicate R *depends on* predicate Q when R occurs in the head of some CHC in which Q occurs in the body, and we write $D(Q, R)$. In our example above, the dependency relation contains the pairs (P, P) and (P, Q). As in this example, the dependency graph $(\mathcal{R}, D)$ is generally cyclic, as cycles in the VC's are induced by loops or recursion in the program.

Definition 7 *An RPFP* $(\mathcal{R}, \mathcal{C})$ *is* simple *if every relational symbol in* $\mathcal{R}$ *occurs* exactly once *in the head of some constraint in* $\mathcal{C}$*. It is* DAG-like *if in addition its dependency relation is acyclic.*

Our example problem is neither simple (since P occurs in the head of two rules) nor acyclic (since P occurs in a dependency cycle). However, we can make it DAG-like by *unwinding* it. For example, here is a possible unwinding:

$$x = y \quad \Rightarrow \quad P_0(x, y) \tag{8}$$

$$P_0(x, y) \wedge z = y + 1 \quad \Rightarrow \quad P_1(x, z) \tag{9}$$

$$P_0(x, y) \wedge P_1(y, z) \quad \Rightarrow \quad Q_1(x, z) \tag{10}$$

$$Q_1(x, z) \quad \Rightarrow \quad x \leq z \tag{11}$$

In the unwinding, we have created instances of unknown predicates by adding subscripts. We have created corresponding instances of the CHC's, in such a

way that the dependency relation is acyclic, and each unknown occurs in the head of exactly one rule. Such a problem has exactly one failure derivation tree for each query, since each P-fact can unify with exactly one head (if it were not acyclic, it might have no derivation, since a top-down derivation would continue infinitely).

At this point, a few additional notations will be helpful. A set of ground facts $P(\vec{x}) \mid \gamma$ can be thought of as an interpretation of P, that is, the assignment $P = \{\vec{x} \mid \gamma\}$. Conversely, we can think of a relational interpretation σ as a set of ground facts, that is, σ represents the set of facts $\{P(\vec{t}) \mid \vec{t} \in \sigma(P)\}$. Thus, it makes sense to write $P(\vec{t}) \in \sigma$ to mean $\vec{t} \in \sigma(P)$. We can also speak of the union and intersections of interpretation as sets of facts. Thus, if $\sigma_1(P) = \lambda\vec{x}.\gamma_1$ and $\sigma_2(P) = \lambda\vec{x}.\gamma_2$, then $(\sigma_1 \cup \sigma_2)(P) = \lambda\vec{x}.\gamma_1 \vee \gamma_2$ and $(\sigma_1 \cap \sigma_2)(P) = \lambda\vec{x}.\gamma_1 \wedge \gamma_2$. As usual in symbolic representations, the syntactic counterparts of union and intersection are disjunction and conjunction.

Now, suppose we construct the failure derivation tree T for a query Q. Finding it vacuous, we construct in interpolant I for T. Consider a node n in T whose head is $P(\vec{x})$ and whose interpolant is $I(n)$. As we noted above, $I(n)$ can be thought of as a set of facts $P(\vec{x}) \mid I(n)$ that gives an upper bound on the facts derivable at node n. We can also think of it as an *interpretation* of P. This gives us a way to derive a *solution* for a DAG-like problem from an *interpolant* for its derivation tree. To do this, we merge the interpretations corresponding to all the interpolant formulas into a single interpretation for $\mathcal{R}$:

Definition 8 *Given a derivation tree T over $\mathcal{R}$ and an interpolant I for T, the* conjunctive merge *of I is defined as*

$$\mathrm{CM}(I)(P) = \cap\{I(n) \mid \mathrm{hd}(n) = P\}$$

That is, the conjunctive merge takes the intersection of all the interpolants for each P, as sets of ground facts. As noted above, this corresponds to conjunction of the formulas. Now we want to show that this in fact yields a solution of the problem. We need the following lemma, that gives a logical characterization of solution of a CHC by a relational interpretation:

Lemma 1 *Let $P, P_1, \ldots, P_k$ be predicate symbols, ϕ a formula, $\vec{x}, \vec{x}_1, \ldots, \vec{x}_k$ vectors of terms and σ a symbolic relational interpretation. Then*

$$\sigma \models P_1(\vec{x}_1), \ldots P_k(\vec{x}_k) \rightarrow P(\vec{x}) \mid \phi \tag{12}$$

if and only if

$$\models \phi \wedge \sigma(P_1)(\vec{x}_1) \wedge \cdots \wedge \sigma(P_k)(\vec{x}_k) \Rightarrow \sigma(P)(\vec{x}) \tag{13}$$

Proof Forward direction. Let $P_1(\vec{t_1}), \ldots P_k(\vec{t_1}) \rightarrow P(\vec{t})$ be a ground instance of the constraint in (12) for some ground substitution ρ. Then (12) implies that if $P_1(\vec{t_1}), \ldots, P_k(\vec{t_k}) \in \sigma$, then $P(\vec{t}) \in \sigma$. If follows that (13) holds for ρ, and since this is true for all ρ satisfying ϕ that (13) is valid. Reverse direction. Suppose

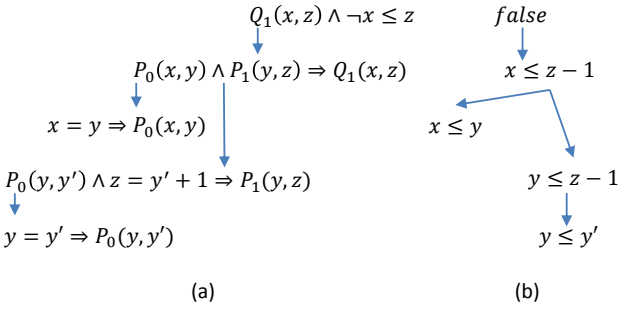

Figure 2: Derivation tree for unwinding.

that $P_1(\vec{t_1}), \ldots, P_k(\vec{t_k}) \in \sigma$. Then by (13) we know that $\sigma(P)(\vec{t})$ holds, thus $\vec{t} \in \sigma$, thus σ satisfies (12). □.

With this lemma, we can easily show that the conjunctive merge of the interpolant solves our problem:

Theorem 2 *Let $\Pi = (\mathcal{R}, \mathcal{C})$ be a DAG-like RPFP with unique query Q, let T be the unique failure derivation for Q in Π, and let I be an interpolant for T. Then $\mathrm{CM}(I)$ is a solution of Π.*

Proof Let $\sigma = \mathrm{CM}(I)$ and consider a rule C with $\mathrm{hd}(C) = P$. According to Lemma 1 we must prove (13). Now consider an arbitrary node n of T such that $\mathrm{hd}(n) = P$. By interpolant property 2, we have $\phi \wedge I(m_1) \wedge \cdots \wedge I(m_k) \Rightarrow I(n)$. Taken together for all n these properties imply (13), thus σ satisfies C. Further, by interpolant property 3, σ maps F to FALSE. This implies that the query Q is satisfied. □

The derivation tree for our unwound example is shown in Figure 2(a). This is the same as the previous tree, with the exception of subscripts on the predicates. The corresponding interpolant is shown in Figure 2(b). Its conjunctive merge is:

$$\begin{aligned} P_0(x,y) &\equiv x \leq y \\ P_1(x,y) &\equiv x \leq y - 1 \\ Q_1(x,z) &\equiv x \leq z - 1 \end{aligned}$$

According to the theorem, this interpretation should solve the problem. In fact, by plugging in these definitions, we can easily see that it does.

We observe here a useful *duality* between models and proofs. That is, suppose that T is a failure derivation for a query in Π. Then satisfiability of its global constraint γ gives us a logical proof that Π is unsatisfiable. On the other hand, *unsatisfiability* of γ, as witnessed by an interpolant I for T, gives us a *solution* for Π, by construing I as a relational interpretation.

We may ask at this point if it is not possible to solve the DAG-like problem directly using interpolation, without expanding the DAG into a tree. This

problem is solved in [1] for the linear case (only one unknown predicate in each body), without the assumption of simplicity. This case is special because each derivation is just a single path (without branching). For the non-linear case, the situation is significantly complicated, as a derivation for a DAG may require multiple facts to be derived at each node [13, 14]. It is not clear that any more efficient approach is possible than expansion into a tree, though this would be an interesting topic for future investigation.

Now we consider the question of DAG-like problems with multiple queries. It turns out this problem is straight-forward, since we can simply combine the solutions for the individual queries:

Theorem 3 *Let $\Pi = (\mathcal{R}, \mathcal{C} \cup \mathcal{Q})$ be an RPFP, where $\mathcal{C}$ is a set of rules, and $\mathcal{Q}$ a set of queries. Further, for every $Q \in \mathcal{Q}$, let $\Pi_Q = (\mathcal{R}, \mathcal{C} \cup \{Q\})$, and let σ_Q be a solution for Π_Q. Then $\cap_Q \sigma(Q)$ is a solution for Π.*

That is, to get a solution for an RPFP, we just take the intersection of the solutions for the individual queries. This allows us to solve a DAG-like problem incrementally, one query at a time.

4 The Duality algorithm

Based on our ability to solve DAG-like problems using interpolation, we now present an algorithm for solving general (cyclic) RPFP's. The algorithm is based on progressively unwinding the problem into a DAG-like problem. Each time a new instance of a query is added to the unwinding, we compute a solution for the query and intersect it with the existing solution. Thus, we always maintain the unwinding in a solved state. While constructing the unwinding, we search for a subset of the predicate instances whose solution constitutes a solution of the original problem. We will call such a subset an *inductive subset*. If we fail to find an inductive subset, our failure tells us where to add another instance of a rule or query to the unwinding. If at any point one of our failure derivations is non-vacuous, we report the derivation as a counterexample. Otherwise, we continue the unwinding until either a solution (a proof of the program) is obtained, or until resources are exhausted.

4.1 Duality example

Consider again our simple example (4–7). We begin the unwinding by instantiating the facts. In this case, we obtain simply

$$x = y \Rightarrow P_0(x, y) \tag{14}$$

Each time we add an instance, we use a fresh subscript for the head predicate. This guarantees that the unwinding is simple. We assign $\sigma(P_0) = \lambda x, y.$ TRUE. This guarantees that the unwinding is in a solved state. Now we try to construct an inductive subset. Our best guess is just $\{P_0\}$. From this set we construct an

assignment for the original cyclic problem. For each predicate $P \in \mathcal{R}$, we take the union (or disjunction) of its instances in the proposed inductive set. This gives:

$$\begin{aligned} P(x,y) &\equiv \textsc{True} \\ Q(x,z) &\equiv \textsc{False} \end{aligned}$$

That is, since we have no instances of Q, its assignment is empty. This assignment fails to solve the problem, since (6) is not valid. This tells us that we need to add an instance of (6). There is only one possible instance, and it is:

$$P_0(x,y) \wedge P_0(y,z) \Rightarrow Q_0(x,z) \tag{15}$$

That is, in the body of the instance, we must use existing predicates. In this way, we guarantee that the unwinding is DAG-like. As before, we set $\sigma(Q_0) = \lambda x, z.\textsc{True}$. Taking $\{P_0, Q_0\}$ as our proposed inductive set, we now see that the query (7) fails. Thus, we add an instance of the query:

$$Q_0(x,z) \Rightarrow x \leq z \tag{16}$$

To satisfy this query, we construct its failure derivation, compute an interpolant for it, and translate this interpolant back to a solution. The derivation tree and one possible interpolant are shown in Figure 3. This gives us the following solution:

$$\begin{aligned} \sigma(P_0) &= \lambda x, y.\ x = y \\ \sigma(Q_0) &= \lambda x, z.\ x \leq z \end{aligned}$$

Taking the intersection (conjunction) of this solution with our existing solution (all $\textsc{True}$) leaves it unchanged. Now, proposing $\{P_0, Q_0\}$ as an inductive subset, we find that (5) fails (in fact, we chose the predicate $x = y$ in the interpolant so the problem would not be too easily solved). This causes us to add in instance of (5):

$$P_0(x,y) \wedge z = y + 1 \Rightarrow P_1(x,z) \tag{17}$$

We set $\sigma(P_1) = \lambda x, z.\ \textsc{True}$. Proposing $\{P_0, P_1, Q_0\}$ as our inductive subset gives us

$$P(x,y) \equiv x = y \vee \textsc{True} \equiv \textsc{True} \tag{18}$$

since we assign each predicate the disjunction of its instances. This causes (6) to fail. Thus, we must add an instance (6). Now, however, we have a choice, since this rule depends on P and there are two instances of P in the unwinding. To make this choice, we apply the general principle that a failure of proof tells us where to search for a counterexample. In this case, we can examine the countermodel we obtain for (6) to determine which of the disjuncts of P is responsible for the failure. From this, we discover that we need to use P_1 at least once to obtain a counterexample. Arbitrarily, we choose the following instance:

$$P_0(x,y) \wedge P_1(y,z) \Rightarrow Q_1(x,z) \tag{19}$$

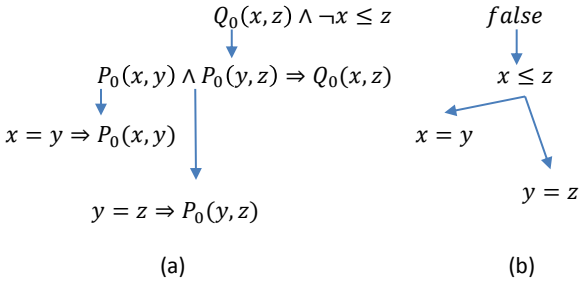

Figure 3: First derivation tree for unwinding.

This in turn causes us to add a new instance of the query:

$$Q_1(x,z) \Rightarrow x \leq z \tag{20}$$

Again, to satisfy this query, we construct its failure derivation. This is exactly the tree of Figure 2. Conjoining the interpolant-based solution from this tree to our existing solution, we now have:

$$\begin{aligned} P_0(x,y) &\equiv x = y \wedge x \leq y \\ P_1(x,y) &\equiv x \leq y - 1 \\ Q_0(x,z) &\equiv x \leq y \\ Q_1(x,z) &\equiv x \leq y - 1 \end{aligned}$$

Now, proposing $\{P_0, P_1, Q_0, Q_1\}$ as our inductive subset, we obtain the following solution (after a little simplification of formulas):

$$\begin{aligned} P(x,y) &\equiv x = y \vee x \leq y - 1 \\ Q(x,z) &\equiv x \leq y \end{aligned}$$

The reader can confirm that this is in fact a solution of our original problem. Thus, the algorithm terminates with a proof of the program.

We can summarize the algorithm as follows. We search for a counterexample by unwinding the problem into a DAG-like problem. When we fail to find a counterexample, we use interpolation to construct a solution for the unwinding, which corresponds to a proof that it contains no counterexamples to the original problem. We then try to generalize this solution to a solution of the original problem by proposing an inductive subset. If this in fact gives us a solution we are done. Otherwise the failure of inductiveness tells us where to expand the unwinding to continue the search to a counterexample.

4.2 Basic Duality algorithm

We now describe the basic duality algorithm a little more formally. The algorithm takes as input an RPFP Π, and returns either a symbolic solution, or a counterexample in the form of a failure derivation.

To formalize the notion of an unwinding, we will use the notion of an *embedding* of one RPFP into another:

Definition 9 *Given two RPFP's* $\Pi = (\mathcal{R}, \mathcal{C})$ *and* $\Pi' = (\mathcal{R}', \mathcal{C}')$*, an* embedding *h of* Π' *in* Π *is a pair of functions* (h_R, h_C)*, where* $h_R : \mathcal{R}' \to \mathcal{R}$ *and* $h_C : \mathcal{C}' \to \mathcal{C}$ *such that, for every* $C' \in \mathcal{C}'$*, we have* $h_C(C') = C'h_R$*.*

That is, Π' embeds in Π if there is a renaming of the predicates that takes every CHC of Π' to a CHC of Π. Our example unwinding above embeds into the original problem *via* the map $\{P_0 \to P, P_1 \to P, Q_0 \to Q, Q_1 \to Q\}$.

Definition 10 *An* unwinding $\mathcal{U}$ *of an RPFP* Π *is a pair* (Π', h) *such that*

- Π' *is DAG-like, and*
- *h is an embedding of* Π' *in* Π*.*

We say σ *is a solution of* $\mathcal{U}$ *if it is a solution of* Π'*.*

The basic Algorithm is shown in Figure 4. The state of the algorithm is a triple $(\Pi, \mathcal{U}, \sigma')$, where Π is the problem to solve, $\mathcal{U}$ is an unwinding of Π and σ is a symbolic solution of $\mathcal{U}$. The main loop at line 4 maintains the following invariants:

- $\mathcal{U}$ is an unwinding of Π, and
- σ' is a solution of $\mathcal{U}$.

The algorithm has just one basic step. We propose an inductive subset of $\mathcal{U}$ at line 5 (this choice is dealt with in Section 4.3). This subset induces a proposed solution σ for the original problem (line 6). We project the unwinding solution σ' onto the subset S, then substitute predicate symbols using the map h_R. In effect, each $P \in \mathcal{R}$ is assigned the disjunction of its instances in S. If this is indeed a solution, we return it (line 7). Otherwise, we choose a failed CHC (line 8). We then choose an instance of that clause to add to $\mathcal{U}$ (line 9). If the CHC is a rule, we assign its head predicate to TRUE (line 13). Else, it is a query. We solve it by constructing its failure derivation (line 16). If the derivation tree is satisfiable, we return a corresponding ground derivation as a counterexample (line 17). Otherwise, we compute an interpolant, and use this to strengthen σ' so that the new query instance is satisfied (lines 18,19).

In the following, we will consider various improvements to this basic algorithm.

4.3 Choosing an inductive subset by covering

The next question we must address is how to propose an inductive subset. Clearly, this operation must be fast, since we perform it in every iteration of the main loop. As in Lazy Abstraction [15], we do this by constructing a binary relation $\rhd$ on $\mathcal{R}$ called a *covering relation*. If $P \rhd R$, we say R *covers* P. This

Algorithm *Unwind*
Input: RPFP $\Pi = (\mathcal{R}, \mathcal{C})$
Output: A symbolic solution σ of Π,
or a counterexample derivation T.

1 Let $\mathcal{R}' = \emptyset$, $\mathcal{C}' = \emptyset$, $\Pi' = (\mathcal{R}', \mathcal{C}')$
2 Let σ', h_R, h_C be empty maps, and $h = (h_R, h_C)$
3 Let $\mathcal{U} = (\Pi', h)$
4 While TRUE do:
5 Let $S =$ *Propose-Inductive-Subset*$(\Pi, \mathcal{U}, \sigma')$.
6 Let $\sigma = (\sigma' \downarrow S)h_R$
7 If σ is a solution of Π, return σ.
8 Let $C \in \mathcal{C}$ such that $C\sigma$ not valid,
where $C = \phi \wedge P_1(t_1) \wedge \cdots P_k(t_k) \Rightarrow H$.
9 Choose $P'_1 \ldots P'_k$ such that
$\not\models \phi \wedge P'_1\sigma'(t_1) \wedge \cdots P'_k\sigma'(t_k) \Rightarrow H\sigma(t)$
10 If H of the form $P(t)$ (* C is a rule *) then
11 Let P' be a fresh predicate symbol, and add it to $\mathcal{R}'$.
12 Let $C' = \phi \wedge P'_1(t_1) \wedge \cdots P'_k(t_k) \Rightarrow H\langle P'/P\rangle$
13 Let $h_R(P') = P$ and $\sigma'(P) = \lambda\vec{x}.$TRUE
14 else (* C is query *)
15 Let $C' = \phi \wedge P'_1 \wedge \cdots P'_k \Rightarrow H$
16 Let T be the failure derivation for C' in Π'.
17 If T has a satisfying assignment ρ, return $T\rho$.
18 Let I be an interpolant for T.
19 Let $\sigma' = \sigma' \cap \mathrm{CM}(I)$.
20 Add C' to $\mathcal{C}'$ and let $H_C(C') = C$
21 Done.

Figure 4: Basic unwinding algorithm.

is allowed when $\sigma'(P) \subseteq \sigma'(R)$. The idea is that a node that is covered has no effect on the solution, thus we can drop it from the proposed inductive set. We will say that a predicate R is *covered* in the unwinding if it depends transitively on a covered predicate. Our proposed inductive subset is just the uncovered subset of the predicates $\mathcal{R}'$ of the unwinding.

As in [21], we avoid infinite cycles of coverings and uncoverings by requiring arcs in $\rhd$ to respect a suitable total order $\prec$ on nodes. A suitable ordering is order of creation. We apply the following rules to add and remove covering arcs:

- After each time $\sigma'(R)$ is strengthened at line 19 of the algorithm, if there is a $P \prec R$ such that $\sigma'(R) \subseteq \sigma'(P)$, we add an arc $R \rhd P$. This removes R and all its dependencies from the inductive set.
- After adding a covering arc $R \rhd P$, we remove any covering arcs $S \rhd P'$ where S depends transitively on R. In other words, we only allow predicates in the inductive set to cover other predicates.
- After each time $\sigma'(P)$ is strengthened at line 19, we recheck each existing arc $R \rhd P$. If $\sigma'(R) \not\subseteq \sigma'(P)$, we remove the arc.

This approach allows us to identify a potentially inductive subset of the unwinding with relatively low overhead. The covering checks can be performed with calls to an SMT solver, and checks can be memoized to amortize the effort.

4.4 Identifying candidate instantiations

Another aspect of the algorithm with potentially high overhead is the test for inductiveness at line 7. Clearly, we would not like to test every CHC at every iteration of the main loop. To avoid this, we can try to identify likely candidates for instantiation heuristically. An instantiation candidate consists of a CHC C and a vector of predicates $P'_1, \ldots, P'_k$ to to instantiate the body of C with. An acceptable candidate must meet the test of line 9 of the algorithm (it must cause inductiveness to fail). We maintain a queue of likely acceptable candidates, and a set E of *expanded* predicates. At line 7, if the queue is non-empty, we remove a candidate. If the candidate is in S, the uncovered subset, and meets the criterion of line 9, we can skip lines 7-9. If the queue becomes empty, we look for an unexpanded predicate P' in S. If one exists, we produce a set of candidates using P' and already expanded predicates in S. To avoid an explosion of candidates, we consider only the maximal expanded instance of any given predicate P according to the order $\prec$. This is quite important for rules C that have a large degree, since the number of candidates would be exponential in the size of the body of the rule. The obtained candidates are then placed in the queue, and predicate P' is added to the expanded set.

The effect of this strategy is to build an unwinding that is roughly "layered". That is, the predicate instances divide into a sequence of layers such that predicates in one layer depend only on predicates in the previous layer. At nearly every iteration, we obtain an acceptable candidate from the queue. Thus, we avoid frequent tests of inductiveness.

4.5 Improving convergence by forced covering

Duality has two ways of improving performance by re-using proofs in different contexts. The first is called *forced covering* and is inherited from IMPACT [21]. The idea of a forced covering is that a fact proved about one instance P_i of predicate P may be useful to prove of another instance P_j. In program proving terms, we may imagine that a fact proved about one execution of a procedure or loop may be true of another. By proving this fact about P_j, we allow P_i to cover P_j. Thus, we are helping to construct an inductive subset.

Forced covering works as follows. Suppose that P_i and P_j are instances of P in the unwinding (that is, $h_R(P_i) = h_R(P_j) = P$) and suppose that $P_i \prec P_j$. Thus, we might be able to add a covering arc $P_j \rhd P_i$. Now suppose that $\sigma'(P_i) = \lambda \vec{x}.\ \phi$. We want to prove the same fact of P_j, so we temporarily add the query $P_j(\vec{x}) \Rightarrow \phi$ to the unwinding. We then try to satisfy the query, as in lines 16-19 of the basic algorithm. If we obtain a counterexample, we simply remove the query and continue. Otherwise, strengthening the solution will result in adding a covering arc $P_j \rhd P_i$.

There is a heuristic question of when to try forced coverings, and what instances P_i to use as potential covers. In the implementation, we try forced covers for P_j just before it is expanded (used to generate instantiation candidates). We use just a few of the most recently generated instances P_i of P, provided they are not currently covered and $P_i \prec P_j$, so that a covering arc may be added.

Forced covering is an important optimization to the algorithm, especially for programs with loops and recursion. In practice, almost all of the query checks in Duality are the result of forced covering.

4.6 Partial derivation trees

Another potentially severe performance issue in the basic Duality algorithm is the size of derivation trees. In the worst case, a derivation tree for a DAG-like problem is exponential-size in the size of the DAG. Moreover, since each derivation tree represents a satisfiability problem that must be solved by an SMT solver, we would like the derivation trees to be as small as possible. We can greatly reduce the size of derivation trees by constructing *partial derivation trees*, relative to an existing assignment. The idea is that our existing solution represents an upper bound on the facts that can be derived at any point in the DAG. As we construct a derivation tree, we can substitute this upper bound for any given predicate, and thus obtain an upper bound on the possible derivable facts. This allows us to reuse interpolants we have previously derived to reduce the size of the derivation trees. In program verification terms, if we have derived a specification for a procedure that is sufficient in one calling context, we may be able to re-use it in others. This is the second form of re-use of proof effort in Duality.

In a *partial derivation tree* relative to interpretation σ', we allow the use of an additional inference rule, as follows:

$$\frac{}{P(\vec{x}) \mid \sigma'(P)(\vec{x})}$$

That is, rather than expand the derivation tree at P, we can simply re-use a previously computed upper bound on the derivable P facts. For example, in the derivation tree of Figure 2, we could have re-used the assignment $P_0(x, y) \equiv x = y$, deriving $P_0(x, y) \mid x = y$.

Using this inference rule, we can define a notion of interpolant, and a obtain result analogous to Theorem 2. That is, by intersecting interpolants obtained for the partial derivation tree with the existing solution σ', we obtain a solution for the new query. On the other hand, the partial derivation tree is an over-approximation. Therefore, we may find that it is not vacuous, when the full derivation tree is in fact vacuous. This is because we may have included facts at the root of the derivation tree that are not actually derivable. Thus, when the constraint tree is satisfiable, we must extend the partial derivation tree further at one of the approximated leaves. If the partial derivation tree becomes a complete derivation tree, we have a genuine counterexample.

The question then arises, if the partial derivation tree is not vacuous, and there are multiple approximated leaves, which do we choose to expand? In the current implementation, we choose the leaf whose predicate has been strengthened the fewest times, on the theory that this predicate is most likely to need strengthening. This is, however, a topic for future research.

5 Generalized Horn Clauses

In general, the formulas produced as verification conditions for a program do not always fall into the CHC form. For example, consider this program fragment:

assert $P(x)$;
if c **then**
    $x := \mathrm{A}(x)$
else
    $x := \mathrm{B}(x)$;
assert $Q(x)$;

The VC associate with this fragment would have a disjunction representing the if-then-else construct:

$$P(x) \wedge (c \wedge A(x, x') \vee \neg c \wedge B(x, x')) \Rightarrow Q(x')$$

While we could always reduce the problem to an equisatisfiable clausal form, there may be a significant performance disadvantage to this. Using large-grained VC's, our derivation trees will cover more behaviors of the code, allowing us to take advantage of the efficiency of a modern SMT solver to search for a counterexample within a large space of execution paths. Using large-grained VC's is the equivalent of using a large-block encoding in program analysis [6].

To do this, we expand our notion of Horn formula:

Definition 11 *A generalized Horn clause (GHC) over a predicate vocabulary $\mathcal{R}$ is a formula of the form $\forall X.\ B[X] \Rightarrow H[X]$, where*

- *The* head $H[X]$ *is either a* P*-fact, or is* $\mathcal{R}$*-free, and*
- *The* body $B[X]$ *is a formula of the form* $\exists Y.\ \phi[X,Y]$ *where* $\phi[X,Y]$ *is quantifier-free, and symbols in* $\mathcal{R}$ *occur only positively in* $\phi[X,Y]$.

Because the unknown predicates occur only positively in the body of the constraints, we could in principle reduce the body to disjunctive normal form and obtain an equivalent set of CHC's of exponential size. Thus we do not obtain an increase in expressiveness using GHC's, but rather hope to obtain efficiency by avoiding reduction to clausal form.

We now expand our notion of derivation tree to GHC's. Whereas in the CHC case, every P-fact in the body of a clause must be true to derive the head, in the the GHC case, we may have to derive only a subset of the body facts. We deal with this by introducing fresh Boolean variables. We assume the bodies of the GHC's have been rewritten into the form:

$$\phi \wedge (b_1 \Rightarrow P_1(t_1)) \wedge \cdots \wedge (b_k \Rightarrow P_k(t_k))$$

The constraint ϕ is obtained by replacing each P-fact in the body by a corresponding Boolean variable b_i. Our derivation rule is:

$$\frac{\begin{array}{l} b_1 \rightarrow P_1(\vec{x}_1) \mid \psi_1 \\ \cdots \\ b_k \rightarrow P_k(\vec{x}_k) \mid \psi_k \\ (b_1 \Rightarrow P_1(\vec{x}_1)), \ldots, (b_k \Rightarrow P_k(\vec{x}_k)) \rightarrow P(\vec{x}) \mid \phi \quad (*) \end{array}}{b \rightarrow P(\vec{x}) \mid (b \Rightarrow \phi) \wedge \psi_1 \wedge \cdots \wedge \psi_k}$$

The idea of this construction is that, if b_i is false, then $P_i(\vec{x}_i)$ is not needed to satisfy the body of the clause, thus we need not derive it. We now say a derivation with conclusion $b \rightarrow P(\vec{x}) \mid \gamma$ is *vacuous* when $\gamma\langle \textsc{True}/b\rangle$ is satisfiable (since a fact is only derived when b is true). Our interpolants for a derivation now contain the b variables. An interpolant $I(n)$, where $\mathrm{hd}(n) = P$ now corresponds to a set of facts $P(\vec{x}) \mid I(n)\langle \textsc{True}/b\rangle$.

With these definitions, we can prove analogs of Lemma 1 and Theorem 2 for GHC's, allowing us to use GHC's in the Duality algorithm.

5.1 Other unwinding approaches

The Duality algorithm can be viewed as an extension of the Impact algorithm [21] to the non-linear case, where derivations are trees rather than paths (thus, Impact is intraprocedural, while Duality can compute procedure summaries). A significant practical difference is that Impact operates on control-flow graph (CFG) representations of programs, at the granularity of basic blocks. Duality abstracts away from programs and operate only on the logical verification conditions. These may represent, for example, entire loops, procedures, or larger program fragments. Use of a CFG representation has certain advantages for the linear case, including the ability to easily slice program paths based on

data flow. On the other hand, by operating on large-grain VC's, Duality allows the interpolating decision procedure to handle the enumeration of program paths internally. Moreover, since the interpolator "sees" more of the program, it may produce more heuristically relevant interpolants by ignoring facts that are true only along specific execution paths.

Another approach to generalizing IMPACT to procedures is that of Whale [2]. For the specific application of procedural program verification, Duality is closely related to Whale, but differs in some important respects. Like IMPACT, Whale operates on CFG's. Like Duality, however, it is designed to operate at the granularity of procedures. In this application, Duality effectively unwinds the call graph upwards, from callees to callers, while Whale does the reverse, unfolding from the main procedure downward. This has several consequences. First, in Whale, node x can cover node y when the its predicate *stronger* than that of y, the reverse of Duality and IMPACT. This means that, in the final invariant, the annotation of a procedure is the *conjunction* of the annotations of its instances in the unfolding, rather than the disjunction. Extending downward rather than upward also means that annotations in the unfolding are not monotonically strengthened. Rather, extensions may require some predicates to be set back to TRUE. Whale does not implement forced covering, and it is unclear how this might be done. Whale also does not implement an analog of partial derivations. In principle it could be extended to do this, which might allow more significant re-use of speculated summaries. At this point it is unclear what the relative advantages of the two unwinding approaches are. Primarily, however, Duality differs from Whale in its generality, as it applies to any problem that can be expressed as an RPFP.

Duality is also related to various *bounded* verification techniques that also perform unwinding. For example, Corral [19] incrementally unwinds the call tree of a program in a manner very similar to the partial derivation tree construction of Duality (we can think of the global constraint of a derivation tree as a bounded model checking formula). FUNFROG also performs bounded verification in this way, using propositional logic [27]. It constructs interpolants and uses these as speculated (bounded) procedure summaries for successive program versions. Here, of course, we are concerned with *unbounded* verification. We use the interpolants to construct an inductive solution. In partial derivations, however, interpolants are re-used much as in FUNFROG.

6 Implementation and experiments

We now report on some experiments to evaluate the hypothesis that a purely logical solver for relational fixed-point problems can complete in performance with program-specific algorithms. If this is true, then we can separate the problem of developing core solving algorithms from the problem of interpreting programming languages and from particular intermediate program representations such as control/data flow graphs (CDFG's). This would in principle make it possible to develop verification engines that are more widely re-usable, and

also reduce the technical barrier to entry in developing verification algorithms.

To test this hypothesis, the Duality algorithm has been implemented in a prototype tool in the functional language F♯. Its input is an RPFP expressed symbolically as a set of GHC's in SMTLIB format. The background theory is AUFLIA (arrays, uninterpreted functions and linear integer arithmetic). Satisfiability checking and interpolation are performed by the interpolating prover of [22], based on the proof-generating capability of the SMT solver Z3 [10], and modified to compute tree interpolants.

There are a few optimizations in the implementation that are not described above. We modified the interpolating prover to bias it toward *weak* interpolants. Intuitively, since an instance of a predicate P in the unwinding is an *underapproximation* of P, we wish to generalize its specification by weakening it, and thus to compute weak interpolants. In addition, constructing a partial derivation tree involves a sequence of satisfiability problems, as we replace approximated leaves in the tree with further expansions of the derivation. Since many constraints are common between these instances, we use Z3 incrementally to conserve learned clauses.

To evaluate the approach, we selected a set of 20 randomly chosen benchmark problems from 1941 in the Static Driver Verifier [4] (SDV) benchmark suite. Random selection was used to avoid "cherry picking" or regression to the mean. Each problem consists of a Windows device driver instrumented with a property automaton. Since we are interested the application of Duality to proof construction, we choose from the set of benchmarks in which the property is believed to be true.

Translation path We used the tool Corral [19] to translate from the SDV intermediate representation (IR) into the Boogie language [5]. Boogie then converts the loops to tail recursion, and then generates procedure-level verification conditions (VC's) that are used as input to Duality. Thus each procedure in the program generates a single GHC, whose head relation is, effectively, a summary of the procedure (or loop, since loops are procedures). The translation to Boogie also introduces some uninterpreted functions, and a corresponding collection of background axioms (with quantifiers). These are passed to Duality and used as an extension of the background theory. This relies on interpolating Z3's ability to generate interpolants for quantified formulas.

A key motivation of Duality is to exploit the efficiency of Z3 to allow proof at the granularity of procedures rather than program statements. Enumeration of execution paths is handled by the efficient backtracking mechanism in Z3. To test this idea, we compared the performance of Duality to Yogi [25], a tool that incrementally computes a predicate abstraction of a program. Like Duality, Yogi constructs procedure summaries, however it does this by explicitly exploring execution paths. In many ways, Yogi and Duality are not directly comparable. However, since Yogi has be extensively optimized for the SDV application, it provides a good baseline to test whether purely symbolic methods can compete with specialized software verification tools.

In reporting performance, we exclude the time needed to translate the problem from the SDV intermediate representation into the Boogie language, as this

Yogi	Duality	BMC	Duality (localized)	Duality (core)
7.89	2.62	1.40	†0.81	†0.21
1.32	2.30	4.01	0.49	0.04
0.11	2.10	3.57	0.70	0.17
0.17	7.59	6.53	1.85	0.62
0.39	2.97	4.13	0.70	0.06
58.64	*2.91	19.96	*2.02	†0.81
61.59	*2.83	20.17	*2.05	†0.60
0.16	2.88	3.40	0.70	0.23
2.28	9.78	6.40	2.74	1.10
75.76	TO	394.16	238.68	144.47
11.92	227.19	212.69	12.71	2.32
730.46	*90.93	23.88	†3.57	†0.98
0.12	1.60	2.99	0.35	0.06
0.44	128.01	6.15	3.68	0.15
821.98	*44.08	35.88	†6.75	†2.38
248.75	203.70	40.07	*11.68	*4.49
0.13	4.94	9.00	0.45	0.07
0.09	5.07	7.54	0.56	0.06
0.35	21.10	5.97	1.26	0.31
0.10	1.77	3.14	0.39	0.07

Table 1: Performance on SDV benchmarks. Run times in seconds.

is implemented in a highly inefficient manner. All solutions were verified using Z3.

Table 1 shows the run-time results measured in seconds for the 20 benchmarks. The first column shows Yogi and the second shows Duality, on the VC's generated by Boogie, with a time-out set at 1000s. A scatter plot of these numbers is shown in Figure 5. While Yogi would be preferred overall, we note four benchmarks where Duality shows a rough order-of-magnitude improvement (marked with asterisks in the table).

Due the the manner of translation from IR to Boogie, the Boogie programs have a large number of global variables (in fact, one array variable for each field of each structure). We hypothesize that this may have a large effect on the SMT solver performance. To test this, we use a form of proof-based localization abstraction [24]. That is, we apply bounded model checking (BMC) using Corral, unwinding the loops twice. Corral identifies the global variables needed to prove these bounded instances using a counterexample-driven approach [19]. The unused globals are replaced by free variables in the program, which is then verified by Duality.

Column 3 of the table shows the BMC run time (the time to generate the localization abstraction) while column 4 shows the resulting Duality run time. In fact, we observe a large improvement. There are now six cases in which

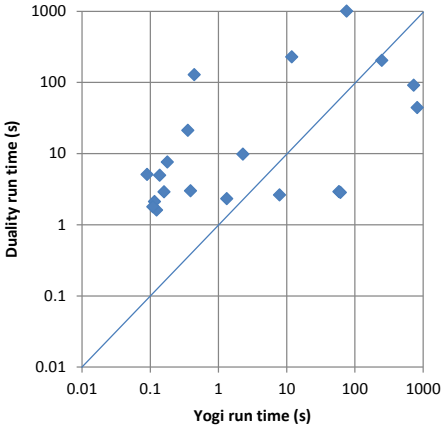

Figure 5: Run-time comparison, Duality *v.* Yogi.

Duality shows an order-of-magnitude improvement over Yogi, and three where it achieves two orders of magnitude improvement (indicated by daggers in the table). Duality's performance in proving the localized programs is generally far faster than the BMC step needed to generate the localization. Thus, this approach of localize-then-prove may be significantly sub-optimal. The Duality results suggest, however, what might be achieved if Duality were to implement an incremental localization technique as in [3].

Finally, column 5 of the table shows the core run time of Duality. This is the time spent actually solving SMT problems and computing interpolants. The remainder of the time is spent parsing files and copying formulas in F♯. We observe that most of the time in Duality is spent on overhead of this type that could be essentially eliminated in an efficient implementation. Figure 6 shows a scatter plot of Duality's core run time against Yogi. This gives a sense of the potential improvement that could be achieved using an RPFP solver based on an efficient interpolating SMT solver.

In cases where Duality does not perform well, it is also possible that the procedure-grain VC's are simply too large for Z3 to handle. An alternative would be to generate VC's at the granularity of procedure *calls* (that is, decorating each call site with a symbolic precondition). This is roughly the same granularity that is used in [6] and might produce better performance in some cases.

Of course this small set of experiments only shows that the approach has potential, and may serve to motivate exploration of purely symbolic algorithms. Addition experimentation is clearly needed to compare the approach to related techniques such as the IMPACT algorithm [21], and Whale [2]. It should be noted, though, that Duality applies to a wider class of problems than these tools. A comparison to QARMC [12] would also be useful. A direct comparison is difficult, however, since Duality handles a richer theory, and is intended to apply to large-grain VC's, while QARMC is restricted to CHC's.

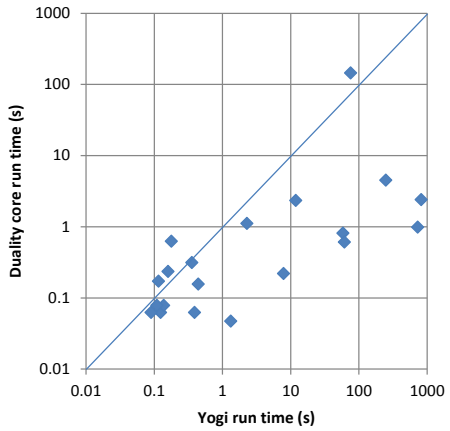

Figure 6: Run-time comparison, Duality (core) *v.* Yogi.

7 Conclusion

Relational fixed-point solving is a generic approach to program verification. It treats a program's verification conditions as a set of logical constraints to be solved for relational unknowns. A solution corresponds to a proof of the program. We have explored a purely interpolant-based approach to this problem, in the style of IMPACT. In doing so, we extended RPFP solving to a richer theory, and extended the problem domain to large-grain VC's, allowing the exploitation of highly efficient SMT solvers.

Experimentally, we explored one possible application: procedural program verification, specifically of control properties of device drivers. Though the results are very limited, we saw that a generic logical solver can compete in this domain with specialized and highly tuned software model checking techniques. Improvements in the efficiency of such a generic solver can potentially be exploited across a spectrum of applications, including refinement type inference [26], threaded program verification [13], protocol verification and so on. Termination verification, as in [9] also seems a possible application. Since the method produces explicit proofs, with auxiliary constructs such as inductive invariants and procedure summaries, it is also possible that its output can be re-usable across problems in a given domain (for example, procedure summaries might be re-usable in the verification of device drivers).

Building an efficient verification tool based on the approach remains for future work. This will require an effective solution for localization abstraction. It will be interesting to explore other applications, such as functional program verification, and perhaps strengthening of manual program annotations. Integration with other methods, such as [12, 23, 17] is also an interesting possibility. The result could be a powerful generic relational solver that is applicable to a wide variety of problem domains.

Acknowledgments We would like to thank Akash Lal, Shaz Qadeer, Atiya

Nori, and the SDV team for providing tools and data used in this paper.

References

[1] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From under-approximations to over-approximations and back. In *TACAS*, pages 157–172, 2012.

[2] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, pages 39–55, 2012.

[3] Nina Amla and Kenneth L. McMillan. Combining abstraction refinement and sat-based model checking. In *TACAS*, pages 405–419, 2007.

[4] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The static driver verifier research platform. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 119–122. Springer, 2010.

[5] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[6] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *FMCAD*, pages 189–197, 2010.

[7] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. In *SMT workshop*, 2012.

[8] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *CAV*, pages 299–303, 2008.

[9] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *CAV*, pages 415–418, 2006.

[10] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[11] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In *CAV*, pages 72–83. Springer-Verlag, 1997.

[12] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.

[13] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 331–344. ACM, 2011.

[14] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Solving recursion-free Horn clauses over LI+UIF. In *APLAS*, pages 188–203, 2011.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.

[16] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.

[17] Krystof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. μz - an efficient engine for fixed points with constraints. In *CAV*, pages 457–462, 2011.

[18] R. Jhala, R. Majumdar, and A. Rybalchenko. Hmc: Verifying functional programs using abstract interpreters. In *CAV*, pages 470–485, 2011.

[19] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A whole-program analyzer for Boogie. Technical Report MSR-TR-2011-60, Microsoft Research, May 2011.

[20] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[21] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[22] K. L. McMillan. Interpolants from Z3 proofs. In *FMCAD*, pages 19–27, 2011.

[23] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *CAV*, pages 104–118, 2010.

[24] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS*, pages 2–17, 2003.

[25] Aditya V. Nori, Sriram K. Rajamani, SaiDeep Tetali, and Aditya V. Thakur. The yogi project: Software property checking via static analysis and testing. In *TACAS*, pages 178–181, 2009.

[26] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.

[27] O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based function summaries in bounded model checking. In *HVC*, 2011.

[28] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In William Pugh and Craig Chambers, editors, *PLDI*, pages 131–144. ACM, 2004.