Validating Library Usage Interactively

William R. Harris, Guoliang Jin, Shan Lu, and Somesh Jha

University of Wisconsin, Madison, WI, USA
{ wrharris, aliang, shanlu, jha }@cs.wisc.edu

Abstract. Programmers who develop large, mature applications often want to optimize the performance of their program without changing its semantics. They often do so by changing how their program invokes a library function or a function implemented in another module of the program. Unfortunately, once a programmer makes such an optimization, it is difficult for him to validate that the optimization does not change the semantics of the original program, because the original and optimized programs are equivalent only due to subtle, implicit assumptions about library functions called by the programs.

In this work, we present an interactive program analysis that a programmer can apply to validate that his optimization does not change his program's semantics. Our analysis casts the problem of validating an optimization as an abductive inference problem in the context of checking program equivalence. Our analysis solves the abductive equivalence problem by interacting with the programmer so that the programmer implements a solver for a logical theory that models library functions invoked by the program. We have used our analysis to validate optimizations of real-world, mature applications: the Apache software suite, the Mozilla Suite, and the MySQL database.

1 Introduction

Application developers often modify a program to produce a new program that executes faster than, but is semantically equivalent to, the original program. After a developer modifies his program, he can determine with high confidence whether the modified program executes faster than the original program by measuring the performance of the original and modified program on a set of performance benchmarks. Unfortunately, it is significantly harder for the developer to determine that the modified program is semantically equivalent to the original program.

Much previous work in developing correct compiler optimizations has focused on developing fully-automatic analyses that determine if two programs are equivalent [24, 26]. Unfortunately, such analyses usually require that the two programs call the same procedures with the same argument values. However, many practical optimizations modify a program to call a library function on different values, or call a different library function entirely. Such analyses cannot prove that such an optimization preserves the semantics of a program. Other analyses attempt

to determine if two programs are equivalent by analyzing the programs interprocedurally [12, 18]. Unfortunately, many practical optimizations modify calls to complex, heavily optimized library functions. Such functions may be difficult to analyze, or their source code may be unavailable.

In this work, we propose a new interactive analysis for determining under what conditions two programs are equivalent. Unlike previous work, the analysis that we propose is not fully automatic. Instead, the analysis takes as input from a programmer an original program and an optimized program, and suggests a candidate specification of the functions defined in libraries and other program modules (in this paper, we refer to all such function as "library" functions for simplicity) called by the program that implies that the original and optimized programs are equivalent. The programmer either validates or refutes the candidate specification, and the analysis uses this validation or refutation to iteratively suggest a new sufficient specification, until the analysis finds a sufficient specification that is validated by the programmer. If a programmer accepts an invalid specification of library functions, then the analysis may incorrectly determine that the programs are equivalent. However, even if a programmer accepts an invalid specification, the analysis still generates an explicit representation of the programmer's key assumptions in submitting the optimization. The assumptions can potentially be validated by other programmers or known techniques for verifying safety properties of programs [4, 7, 15].

There are two key challenges to developing an interactive equivalence checker. The first key challenge is to develop a checker that can construct candidate specifications about functions whose implementations may not be available, or that manipulate complex abstract datatypes, such as strings, that are difficult to reason about symbolically. The equivalence checker must find specifications describing such functions that it can soundly determine to be consistent and sufficient to prove that the original program is equivalent to the optimized program.

The second key challenge is to develop an interactive checker that queries its user with simple, non-redundant candidate specifications about the library functions that a program calls. To prove equivalence between an original and optimized program, the interactive checker must work with the user to construct a simulation relation from the state space of the original program to the state space of the optimized program. However, the complexity of constructing such a simulation relation should be largely hidden from the user, so that the user must only ever make simple Boolean decisions determining the validity of a candidate specification of library functions.

Our key insight to address the above challenges is to design the checker so that it treats the user as a solver for a theory that describes the library functions. The equivalence checker reduces the problem of checking equivalence to proving the validity of a set of formulas, using known techniques for checking equivalence [24]. To prove validity of the required formulas, the equivalence checker applies an *abductive theorem prover*, which generates a set of assumptions over the library functions, restricted to logical combinations of equalities, that are sufficient for each formula to be valid. To generate such assumptions, the theorem prover uses

```
string srch_strm(string s) {
L0: string sb := "";
L1: while(!find(sb, s)) {
      char c := get();
      sb := append(sb, c);
    }
L2: return sb;
}
```

```
string srch_strm'(string s) {
L0':  str sb := "";
      int pos := - len(s);
L1':  while(pos < 0
            || !find(sub(sb, pos), s)) {
       char c := get();
       sb := append(sb, c);
       pos := len(sb) - len(s);
     }
L2': return sb;
}
```

Fig. 1. An example original program `srch_strm` and its optimization `srch_strm'`, derived by simplifying the original and optimized programs submitted in Apache Bug #34464.

an *optimistic solver* for the theory of program libraries. If the optimistic solver finds consistent assumptions sufficient to prove validity, then the equivalence checker presents the assumptions for the user to validate or refute. In other words, the optimistic solver interacts with the user to implement a "guess-and-check" solver for the theory of the libraries.

The rest of this paper is organized as follows: §2 illustrates our equivalence problem and analysis on a function and its optimization submitted in a bug report for the Apache Ant build tool. §3 presents our abductive equivalence problem and analysis in detail. §4 presents an experimental evaluation of our analysis on a set of benchmarks taken from bug reports to fix performance issues in applications. §5 discusses related work, and §6 concludes.

2 Overview

In this section, we motivate the abductive equivalence problem and algorithm introduced in this paper using an optimization submitted in a bug report for the Apache Ant build tool [3]. Fig. 1 contains a program function `srch_strm` and an optimization of the function `srch_strm'` submitted in Apache Bug Report #34464 [2]. The actual original and optimized programs submitted in Bug Report #34464 use additional variables and control structure, and were written in Java. The actual programs have been simplified to `srch_strm` and `srch_strm'` in Fig. 1 in order to simplify our discussion of abductive equivalence problem. However, we have implemented our algorithm as a tool that checks the equivalence of the actual programs submitted in Apache Bug Report #34464 (we translated the programs to C++ by hand so that we could apply our checker, which uses the LLVM compiler framework [20]).

`srch_strm` and `srch_strm'` implement an equivalent search for a substring in an input stream. Both functions take a string `s`, and read characters from a stream until they have read a string that contains `s`. `srch_strm` implements

the search by constructing an empty string `sb` and iteratively checking if `sb` contains `s` as a subsequence. In each iteration, `srch_strm` calls `find(sb, s)`, which returns `True` if and only `sb` constrains `s` as a subsequence. If `find`$(\texttt{sb}, \texttt{s}) =$ `True`, then `srch_strm` returns `sb`. Otherwise, `srch_strm` gets a character `c` from the stream, appends `c` to `sb`, and iterates again.

An Apache developer observed that while `srch_strm` outputs the correct value for each input string, it is inefficient due to how it uses the string library functions `find` and `append`. In Apache Bug Report #34464, the developer submitted a patch to `srch_strm`, called `srch_strm'`, that is functionally equivalent to `srch_strm`, but which the developer measured to be more efficient than `srch_strm`. `srch_strm'` is structured similarly to `srch_strm`, but executes more efficiently by only searching for `s` in a sufficiently long suffix of `sb`. In particular, `srch_strm'` maintains an integer variable `pos` that stores the position in `sb` from which `srch_strm'` searches for `s`. In each iteration, `srch_strm'` constructs the substring of `sb` starting at `pos`, `sub(sb, pos)`, tries to find `s` in `sub(sb, pos)`, and if it fails, gets a new character `c` from the stream, appends `c` to `sb`, and iterates again.

An Apache developer submitted `srch_strm'` with an informal argument that it is semantically equivalent to `srch_strm`, but ideally, the developer would submit `srch_strm'` accompanied by an proof that that could be checked automatically to determine that `srch_strm` is equivalent to `srch_strm'`. Existing analyses for constructing automatically-checkable proofs of equivalence construct a *simulation relation* from P to P', which shows that every execution of P corresponds to an execution of P' that returns the same value [21, 24, 26]. A simulation relation $\sim$ from P to P' is a binary relation from the states of P to the states of P' such that: (1) $\sim$ relates each initial state of P to the state in P' with equal values in each variable; (2) $\sim$ relates each return state of P to a state of P' with the same return value; and (3) if $\sim$ relates a state q_0 of P to a state q_0' of P' and q_0 transitions to a state q_1 of P, then q_0' transitions, possibly over multiple steps, to some state q_1' of P' such that $\sim$ relates q_1 to q_1'.

One simulation relation from `srch_strm` to `srch_strm'`, under the informal description of the library functions `append`, `find`, `sub`, and `len`, is:

$$(L_0, L_0') : s = s' \tag{1}$$
$$(L_1, L_1') : s = s' \wedge sb = sb' \tag{2}$$
$$(L_2, L_2') : sb = sb' \tag{3}$$

The simulation relation of Eqns. (1)–(3) is represented as a map from a pair of program labels to a formula that describes pairs of program states. A state q of `srch_strm` at label `L` is related to a state q' of `srch_strm'` at label `L'` if the values of the variables in q and q' satisfy the formula mapped from (L, L') (where the variables of q' are primed). The simulation relation of Eqns. (1)–(3) relates all initial states of `srch_strm` to initial states of `srch_strm'` with equal values for `s` (Eqn. (1)), all return states of `srch_strm` to return states of `srch_strm'` that return the same value (Eqn. (3)), and states at the loop head of `srch_strm` to states at the loop head of `srch_strm'` with equal values for `s` and `sb` (Eqn. (2)).

The simulation relation of Eqns. (1)–(3) is a straightforward instance of the definition of a simulation relation for `srch_strm` and `srch_strm'`. However, the fact that the state relation in Fig. 1 satisfies condition (2) of a simulation relies on the semantics of the library functions `append`, `find`, and `sub` called by `srch_strm` and `srch_strm'`. Unfortunately, in practice it is difficult to automatically infer accurate specifications for library functions, as such functions may be unavailable or difficult to analyze.

Fortunately, while a programmer may not be able to give a complete formal specification of a library, they often understand a weaker, partial specification that implies the equivalence of a particular optimization. For example, `srch_strm` and `srch_strm'` are equivalent under the assumptions that (1) if the length of string sb is less than the length of string s, then $\texttt{find}(sb, s) = \mathsf{False}$ and (2) if $\texttt{find}(\texttt{sb}, \texttt{s}) = \mathsf{False}$, then `s` is a subsequence of the concatenation of `s` with a character `c` if and only if `s` is a subsequence of the the suffix of `s` and `c` of length equal to the length of `s`:

$$\forall sb, s.\ \texttt{len}(sb) - \texttt{len}(s) < 0 \implies \neg\texttt{find}(sb, s) \tag{4}$$

$$\forall sb, s, c.\ \neg\texttt{find}(sb, s) \implies (\texttt{find}(\texttt{append}(sb, c), s) \iff \texttt{find}(\texttt{sub}(sb, \texttt{len}(sb) - \texttt{len}(s)), s)) \tag{5}$$

Based on the insight that programmers can often reliably validate partial specifications of libraries, in this work we introduce the *abductive equivalence* problem AEQ (§3.2). An abductive equivalence problem is defined by an original program P, optimized program P', and an *oracle*, which models a programmer, that takes a formula φ describing the library functions called by P and P' and accepts φ if the oracle's model of the library functions satisfies φ. A solution to the problem is a simulation relation from P to P' the oracle's model of the library functions.

We present a sound algorithm for AEQ, called ChkAEQ (§3.3), that extends existing algorithms for checking program equivalence [24]. Like algorithms for checking equivalence, ChkAEQ first asserts that the return values of input programs P and P' are equal, and then reasons backwards over the executions of P and P' to construct a simulation relation from P to P' represented as a map from pairs of program control labels to formulas in a logic that describes the states of the program. The key feature of ChkAEQ is that as it constructs a relation $\sim$ from the states of P to P', it applies an *abductive theorem prover* to construct a condition on the library functions (i.e., a *library condition*) that implies that $\sim$ is a simulation relation. If ChkAEQ finds a simulation relation and sufficient library condition, it queries the input oracle on the library condition to determine if the oracle's model satisfies the library condition. If the library oracle validates the condition, then ChkAEQ returns the simulation relation. Otherwise, if the oracle refutes the condition, then ChkAEQ uses the refutation to continue to search for a simulation relation.

For `srch_strm` and `srch_strm'` (Fig. 1), ChkAEQ could infer the following library condition that is both consistent and sufficient for the state relation in

$$
\begin{array}{ll}
\mathsf{block} := \mathtt{L} : \mathsf{instr}; \mathsf{term} & \mathtt{L} \in \mathsf{Labels} \\
\mathsf{instr} := \mathtt{x_0} := \mathtt{f}(\mathtt{x_1}, \ldots, \mathtt{x_n}) \mid \mathtt{x_0} := \mathtt{g}(\mathtt{x_1}, \ldots, \mathtt{x_n}) & \{\mathtt{x_i}\}_i \subseteq \mathsf{Vars}; \mathtt{f} \in \mathsf{Ops}; \mathtt{g} \in \mathsf{LibOps} \\
\mathsf{term} := \mathtt{return\ x} \mid \mathtt{br\ x\ ?\ L_t\ :\ L_f} & \mathtt{L_t}, \mathtt{L_f} \in \mathsf{Labels}; \mathtt{x} \in \mathsf{Vars}
\end{array}
$$

Fig. 2. Syntax of the programming language Imp, described in §3.1. An Imp program is a set of blocks.

Fig. 1 to be a simulation relation:

$$\forall sb, s.\ \mathtt{find}(sb, s) \tag{6}$$

However, a programmer serving as a library oracle would refute Eqn. (6). ChkAEQ would then use the refutation to search for a condition consistent with the negation of Eqn. (6), and would eventually find the library conditions of Eqns. (4) and (5).

3 Abductive Equivalence

In this section, we formally define the abductive equivalence problem and algorithm. In §3.1, we define the syntax and semantics of a simple imperative language Imp. In §3.2, we define the abductive equivalence problem for Imp programs. In §3.3, we present a sound algorithm for solving the abductive equivalence problem. We discuss the abductive equivalence problem using Imp for simplicity. In practice, we have implemented an abductive equivalence checker for the LLVM intermediate language (see §4).

3.1 IMP: a Simple Imperative Language

IMP Syntax. An Imp program updates its state by executing a sequence of program and library operations. The Imp language (Fig. 2) is defined over a set of variable symbols Vars, a set of control labels Labels, a set of language function symbols Ops, and a set of library function symbols LibOps, where Vars, Labels, Ops, and LibOps are mutually disjoint. Labels contains a label `RET` that does not label any block of an Imp program (`RET` is used to define the semantics of a return instruction; see §3.1).

An Imp program is a set of basic blocks. Each Imp program P contains one initial block labeled with a $\iota(P) \in \mathsf{Labels}$. Each basic block block is a label followed by an instruction and block terminator. An instruction is an assignment of either a language operation or a library operation. A block terminator is either a return instruction or a conditional branch.

IMP Semantics. The operational semantics of Imp (Fig. 3) defines how a basic block transforms a given state under a given model of library operations.

$$\sigma_b^m[\![\mathtt{L} : \mathsf{instr}; \mathsf{term}]\!](L_0, V) \equiv \textbf{if } L_0 = L \textbf{ then } \sigma_t[\![\mathsf{term}]\!](\sigma_i^m[\![\mathsf{instr}]\!](V)) \textbf{ else } \bot \quad (7)$$

$$\sigma_i^m[\![\mathtt{x_0} := \mathtt{f}(\mathtt{x_1})]\!](V) \equiv V[x_0 \mapsto \mathsf{langmodel}(\mathtt{f})(V(x_1), \ldots, V(x_n)))] \quad (8)$$

$$\sigma_i^m[\![\mathtt{x_0} := \mathtt{g}(\mathtt{x_1})]\!](V) \equiv V[x_0 \mapsto m(\mathtt{g})(v(x_1), \ldots, V(x_n))] \quad (9)$$

$$\sigma_t[\![\mathtt{return\ x}]\!](V) \equiv (\mathtt{RET}, V[\mathsf{rv} \mapsto V(x)]) \quad (10)$$

$$\sigma_t[\![\mathtt{br\ x\ ?\ L_t\ :\ L_f}]\!](V) \equiv ((\textbf{if } v(x) \neq 0 \textbf{ then } L_t \textbf{ else } L_f), V) \quad (11)$$

Fig. 3. Operational semantics of Imp. σ_b^m is the operational semantics of a block and σ_i^m is the operational semantics of an instruction under library model m. σ_t is the operational semantics of a block terminator. In σ_b, $\bot$ denotes the undefined value. In σ_i and σ_t, for $a, b \in \mathbb{Z}$, $v[a \mapsto b]$ maps a to b, and maps $c \neq a$ to $v(c)$. In σ_t, $\mathsf{rv} \in \mathsf{Vars}$ is a variable of each program that stores the return value of the program.

An Imp *state* is a label paired with a *valuation*, which is a map from each program variable to an integer value: $\mathsf{Valuations} = \mathsf{Vars} \to \mathbb{Z}$ and $\mathsf{States} = \mathsf{Labels} \times \mathsf{Valuations}$. A *library model* $m : \mathsf{LibOps} \to (\mathbb{Z}^* \to \mathbb{Z})$ maps each library function to a function from a vector of integers to an integer.

The semantic function of a block σ_b^m (Fig. 3, Eqn. (7)) takes a basic block block and a state (L, v) and outputs the state to which block transforms (L, v) under m. The semantic function of an instruction σ_i^m (Fig. 3, Eqns. (8) and (9)) defines how an instruction updates a valuation. If an instruction assigns the result of a language operation, then the value of the operation is defined by a fixed *language model* $\mathsf{langmodel} : \mathsf{Ops} \to (\mathbb{Z}^* \to \mathbb{Z})$ (Fig. 3, Eqn. (8)). If an instruction assigns the result of a library operation, then the value of the operations is defined by m (Fig. 3, Eqn. (9)). The semantic function of a block terminator σ_t defines the effect of a block terminator on a given state (Fig. 3, Eqns. (10) and (11)).

3.2 The Abductive Equivalence Problem

To define the abductive equivalence problem, we adopt the definition of a simulation relation [21], which has been used to define the classical equivalence problem [24, 26]. A simulation relation from an Imp program P to an Imp program P' is a relation from states of P to states in P' that implies that if from inputs I, P has an execution that returns value v, then from I, P' also has an execution that returns v.

Defn. 1 *Let* Imp *program* P *be* compatible *with* Imp *program* P' *if every variable of* P *corresponds to a variable of* P'*:* $\mathsf{Vars}(P) \subseteq \mathsf{Vars}(P')$*. A* simulation relation *from* P *to a compatible program* P' *under library model* $m : \mathsf{LibOps} \to (\mathbb{Z}^* \to \mathbb{Z})$ *is a relation* $\sim \subseteq \mathsf{States} \times \mathsf{States}$ *from the states of* P *to the states of* P' *such that:*

1. Initial states*:* $\sim$ *relates each initial state of* P *to its analogous state in* P'*. For valuations* $V, V' \in \mathsf{Valuations}$*, if for each* $x \in \mathsf{Vars}(P)$*,* $V(x) = V'(x)$*, then* $\sim ((\iota(P), V), (\iota(P'), V'))$*.*

Algorithm 1 An abductive equivalence algorithm ChkAEQ. Takes as input a $\mathcal{T}_{lib}$ oracle $\mathcal{O}_m$ and two Imp programs P and P', and constructs a solution to $\mathsf{AEQ}(m, P, P')$ (Defn. 2). $\mathsf{Interact}(A)$ returns a simulation relation under a library condition that implies A and is validated by $\mathcal{O}_m$. ChkAEQ is discussed in §3.3.

```
1: function ChkAEQ(O_m, P, P')
2:     function Interact(A)
3:         (C, ~) := SimRel(P, P', A)
4:         if O_m(C) then return ~
5:         else return Interact(A ∧ ¬C)
6:         end if
7:     end function
8:     return Interact(True)
9: end function
```

2. Return states*: $\sim$ relates each return state of P to a return state of P' that returns an equal value. For each $V \in \mathsf{Valuations}$, there is some $V' \in \mathsf{Valuations}$ such that $V(\mathsf{rv}) = V'(\mathsf{rv})$ and $\sim ((\texttt{RET}, V), (\texttt{RET}, V'))$.*
3. Consecution*: if $\sim$ relates a state s_0 of P to a state s'_0 of P' and s_0 transitions to a state s_1 of P, then s'_0 eventually transitions to a state s'_1 of P' such that $\sim$ relates s_1 to s'_1. For program P, let the transition relation $\rightarrow_P \subseteq \mathsf{States} \times \mathsf{States}$ relate states connected by the transition function of P: for states $s_0, s_1 \in \mathsf{States}$, $\rightarrow_P (s_0, s_1)$ if and only if there is some block $B \in P$ such that $s_1 = \sigma_b^m[\![B]\!](s_0)$. Let $\rightarrow_P^* \subseteq \mathsf{States} \times \mathsf{States}$ be the reflexive transitive closure of $\rightarrow_P$. For $s_0, s'_0, s_1 \in \mathsf{States}$, if $\sim (s_0, s'_0)$ and $\rightarrow_P (s_0, s_1)$, then there is some $s'_1 \in \mathsf{States}$ such that $\rightarrow_{P'}^* (s'_0, s'_1)$ and $\sim (s_1, s'_1)$.*

The abductive equivalence problem for programs P and P' and library model m is to find a simulation relation from P to P' under m using P, P', and an *oracle* for m that answers Boolean queries about properties of m. Intuitively, the oracle formalizes the role of a programmer who can answer queries about the specification of a library. The oracle answers queries on properties expressed as formulas of a logical theory $\mathcal{T}_{lib}$ whose models describe Imp's library operations. For each library operation $\mathsf{g} \in \mathsf{LibOps}$, there is an uninterpreted function symbol g in $\mathcal{T}_{lib}$. The only predicate of $\mathcal{T}_{lib}$ is the equality relation $=$. The set of *library conditions* $\mathsf{Forms}(\mathcal{T}_{lib})$ are the first-order formulas of $\mathcal{T}_{lib}$, and for library condition $\varphi \in \mathsf{Forms}(\mathcal{T}_{lib})$, $m \models \varphi$ denotes that m is a model of φ. For library model m, the *oracle* $\mathcal{O}_m \subseteq \mathsf{Forms}(\mathcal{T}_{lib})$ *accepts* a $\mathcal{T}_{lib}$ formula if and only if m satisfies the formula: for $\varphi \in \mathsf{Forms}(\mathcal{T}_{lib})$, $\mathcal{O}_m(\varphi)$ if and only if $m \models \varphi$.

Defn. 2 *For library model m and* Imp *programs P and P', the* abductive equivalence problem $\mathsf{AEQ}(m, P, P')$ *is, given $\mathcal{O}_m$, P, and P', to find a simulation relation $\sim$ from P to P' under m.*

3.3 A Sound Algorithm for Abductive Equivalence

Interacting with an Oracle to Solve AEQ. In this section, we present an algorithm that soundly tries to solve the abductive equivalence problem (Defn. 2), called ChkAEQ (Alg. 1). ChkAEQ tries to solve an abductive equivalence problem $\mathsf{AEQ}(m, P, P')$ as follows. ChkAEQ first defines a function Interact (Alg. 1, lines [2]–[6]) that takes a library condition A, and constructs a simulation relation from P to P' under m if $m \models A$. When ChkAEQ is successful, it returns the simulation relation constructed by applying Interact to True (Alg. 1, line [8]). However, ChkAEQ may fail or not terminate.

Interact constructs a simulation relation by applying a function SimRel (Alg. 1, line [3]) that takes an original program P, optimized program P', and a library condition A, and constructs (1) a library condition C such that $C \implies A$ and (2) a simulation relation from P to P' under each library model that satisfies C (in which case, we say that $\sim$ is a simulation from P to P' *under* C). If $\mathcal{O}$ accepts C (Alg. 1, line [4]), then Interact returns $\sim$ (Alg. 1, line [4]). Otherwise, Interact calls itself with a stronger library condition that asserts that C is not valid (Alg. 1, line [5]).

Constructing a Simulation Relation. SimRel constructs a simulation relation represented as a symbolic state relation, which is a function from pairs of labels to a formula of a theory $\mathcal{T}$ for which each model defines a library model, a state of P, and a state of P'. The theory $\mathcal{T}_{\text{IMP}}$ describes of fixed programs P and P'. For each $\mathtt{f} \in \mathsf{Ops}$, let there be a unary function f in the logical theory $\mathcal{T}_{\text{IMP}}$. For each program variable $\mathtt{x} \in \mathsf{Vars}(P)$, let there be a $\mathcal{T}_{\text{IMP}}$ constant x, and for each $\mathtt{x} \in \mathsf{Vars}(P')$, let there be a $\mathcal{T}_{\text{IMP}}$ constant x' that does not correspond to any variable in $\mathsf{Vars}(P)$. Let the only predicate of $\mathcal{T}_{\text{IMP}}$ be the equality predicate, which $\mathcal{T}_{\text{IMP}}$ shares with $\mathcal{T}_{lib}$ (§3.2). Let the combination of $\mathcal{T}_{lib}$ and $\mathcal{T}_{\text{IMP}}$ be $\mathcal{T} = \mathcal{T}_{lib} + \mathcal{T}_{\text{IMP}}$ [25]. A symbolic state relation $\mathsf{sym} : (\mathsf{Labels} \times \mathsf{Labels}) \to \mathsf{Forms}(\mathcal{T})$ relates states $(L, v), (L', v') \in \mathsf{States}$ under library condition C if for each library model $m \models C$, $m \cup v \cup v' \models \mathsf{sym}(L, L')$.

To construct a symbolic simulation relation from P to P', SimRel (Alg. 2) first constructs a symbolic state relation $\mathsf{sym}_{\mathtt{ret}}$ that satisfies the return condition of a simulation relation (Alg. 2, lines [2]–[6]; the condition on return states is defined in Defn. 1, item 1).

From $\mathsf{sym}_{\mathtt{ret}}$, SimRel constructs a library condition A_{cs} such that $A_{cs} \implies A$, and a symbolic state relation sym_{cs} that satisfies the consecution condition of a simulation relation (Defn. 1, item 3) under A_{cs}. SimRel defines a *label transition relation* $\mathsf{lr} \subseteq (\mathsf{Labels} \times \mathsf{Labels}) \times (\mathsf{Labels} \times \mathsf{Labels})$, which defines the domain of sym_{cs}, by applying a function ConsLocRel (Alg. 2, line [7]). ConsLocRel can be implemented using known techniques and heuristics from classical equivalence checking [24]. SimRel then defines a function ConsecSimRel (Alg. 2, lines [8]–[21]) that takes as input (1) a library condition C, (2) a symbolic state relation sym, and (3) a workset W of label pairs on which sym may not satisfy Defn. 1, item 3 under C, and constructs a library condition A_{cs} and a state relation sym_{cs} that satisfy the consecution condition, and such that A_{cs} implies C. To construct

Algorithm 2 SimRel: takes an original program P, optimized program P', and library condition A, and constructs a library condition C such that $C \implies A$ and a simulation relation from P to P' under C. ConsecSimRel takes a library condition C, symbolic state relation sym, and set of pairs of labels W at which sym may not be a simulation relation under C and returns a library condition C' and symbolic simulation relation under C'.

```
 1: function SimRel(P, P', A)
 2:     function sym_ret(L, L')
 3:         if L = RET = L' then return rv = rv'
 4:         else return True
 5:         end if
 6:     end function
 7:     lr ← ConsLocRel(P, P')
 8:     function ConsecSimRel(C, sym, W)
 9:         if W = ∅ then return (C, sym)
10:         else
11:             (L, L') ← RemElt(W)
12:             cs := ⋁{wp_P^P' [[(L, L'), (L1, L1')]](sym(L1, L1')) | (L1, L1') ∈ succ_lr(L, L')}
13:             suff := ATP(sym(L, L') ⟹ cs)
14:             if IsT*Consistent(suff) then C := C ∧ suff
15:             else
16:                 sym(L, L') := cs
17:                 W := W ∪ preds_lr(L, L')
18:             end if
19:             return ConsecSimRel(C, sym, W)
20:         end if
21:     end function
22:     (A_cs, sym_cs) := ConsecSimRel(A, sym_ret, dom(lr) ∪ img(lr))
23:     if IsT*Valid(initeq ⟹ sym_cs(ι(P), ι(P'))) then return (A_cs, sym_cs)
24:     else abort
25:     end if
26: end function
```

sym_{cs} and A_{cs}, SimRel applies ConsecSimRel to A, $\mathsf{sym}_{\mathtt{ret}}$, and all pairs of labels in the domain and image of lr (Alg. 2, line [22]).

SimRel then checks that sym_{cs} satisfies the condition for a simulation relation on initial blocks (Defn. 1, item 1) by checking that $\mathsf{initeq} \equiv \bigwedge_{x \in \mathsf{Vars}(P)} x = x'$ implies the relation of states at the initial blocks of P and P' (Alg. 2, line [23]). If so, then SimRel returns $(A_{cs}, \mathsf{sym}_{cs})$ as a simulation relation (Alg. 2, line [23]). Otherwise, SimRel fails (Alg. 2, line [24]).

ConsecSimRel (Alg. 2, lines [8]–[21]) first checks if W is empty (Alg. 2, line [9]), and if so, returns C and sym (Alg. 2, line [9]). Otherwise, ConsecSimRel chooses a pair of a labels (L, L') from W (Alg. 2, line [11]) on which it will update sym. For $\varphi \in \mathsf{Forms}(\mathcal{T})$, let $\mathsf{wp}_P^{P'}[\![(L_0, L_0'), (L_1, L_1')]\!](\varphi)$ be the formula whose models define states that transition to states in φ over steps of execution in P from L_0 to L_1 and steps of execution in P' from L_0' to L_1' ($\mathsf{wp}_P^{P'}$ is defined from the semantics

of Imp (Fig. 3) using well-known techniques [13]). ConsecSimRel constructs cs, the disjunction of the weakest precondition of each formula to which sym maps each successor of L and L' under lr (Alg. 2, line [12]). ConsecSimRel then tries to construct a library condition suff $\in$ Forms($\mathcal{T}_{lib}$) that is consistent and implies that sym(L, L') implies cs. To construct suff, ConsecSimRel applies an abductive theorem prover ATP (Alg. 2, line [13]; ATP is discussed in App. A). If suff is consistent, then ConsecSimRel conjoins suff to C (Alg. 2, line [14]). Otherwise, ConsecSimRel updates sym to map L and L' to cs, and adds each predecessor of L and L' under lr to W (Alg. 2, lines [16]–[17]). ConsecSimRel then calls itself recursively on its updated library condition, symbolic state relation, and workset.

Correctness of ChkAEQ AEQ is at least as hard as determining if two Imp programs are equivalent. It is straightforward to show that Imp is Turing-complete, and thus that AEQ is undecidable. However, ChkAEQ is sound for AEQ:

Theorem 1. *For oracle $\mathcal{O}_m$ and* Imp *programs P and P', if $\sim$= ChkAEQ$(\mathcal{O}_m, P, P')$, then $\sim$ is a simulation relation from P to P' under m.*

Proof. See an extended version of this paper [14].

However, ChkAEQ is incomplete in that there are some AEQ problems that have a solution, but for which ChkAEQ either will abort or not terminate.

ChkAEQ also never queries its oracle on redundant inputs:

Theorem 2. *For library oracle $\mathcal{O}_m$, original* Imp *program P, and optimized* Imp *program P', let $\varphi_0, \ldots, \varphi_n \in$ Forms$(\mathcal{T}_{lib})$ be the sequence of $\mathcal{T}_{lib}$ formulas not accepted by $\mathcal{O}_m$ in an execution of ChkAEQ$(\mathcal{O}_m, P, P')$. Then for $0 < i \leq n$, $\bigwedge_{0 \leq j < i} \neg\varphi_j \not\Rightarrow \varphi_i$.*

Proof. See [14].

4 Experiments

We carried out a set of experiments to determine if programmers can apply ChkAEQ (§3) to validate practical optimizations. The experiments were designed to answer the following questions:

1. Given a function from a real-world program and its optimization, can ChkAEQ quickly find a library condition that is sufficient to prove that the programs are equivalent?
2. Can ChkAEQ find library conditions that are small and easy for a programmer to validate?

To answer these questions, we implemented ChkAEQ as a tool, `chklibs`, and applied `chklibs` to a set of program functions and their optimizations. Each function was taken from a mature, heavily-used program, namely the Apache

Benchmark Data					chklibs Performance			
Program Name	Bug ID	LoC			Analysis Time (s)	Num. Clauses	Avg. Query Size	
		Org.	Opt.	Diff.			Clause Size	Num. Preds
Apache	19101	27	28	5	0.325	2	9.5	6.0
	34464	23	20	34	18.188	5	9.0	6.4
	44408	51	52	6	0.050	1	8.0	6.0
	45464	569	570	6	0.165	1	8.0	6.0
	48778	30	28	16	0.534	7	11.4	6.0
Mozilla	103330	217	216	5	0.064	1	18.0	6.0
	124686	198	198	4	0.096	1	278.0	6.0
	267506	182	184	9	0.507	5	8.0	6.0
	409961	54	57	12	0.795	3	47.0	6.0
MySQL	38769	223	227	4	0.169	1	11.0	6.0
	38824	346	321	18	29.894	13	179.2	6.0

Table 1. Experimental data from using `chklibs`. The data given for each benchmarks program includes the name and of the source program, the bug report that presented the optimization, the number of lines of code of the original and optimized program functions, and the number of lines output by `diff` on the original and optimized programs. The data measuring `chklibs`'s performance includes the time taken by `chklibs` to construct a simulation relation (in seconds), the number of clauses on which `chklibs` queried the user, the average size of (i.e, number of logical symbols in) the clause, and the average number of predicates in each clause.

software suite, Mozilla Suite, or MySQL database. Each original program function was the subject a bug report reporting that the function's behavior was correct, but that its performance was inefficient. Each corresponding optimized function was the patched, optimized function provided in the bug report. We interacted with `chklibs` to find library conditions that were sufficient to prove that the optimization was correct, and were valid according to our understanding of the libraries.

The results of the experiments indicate that ChkAEQ can be applied to validate practical optimizations. In particular:

1. `chklibs` quickly inferred library conditions that were sufficient to prove equivalence. `chklibs` usually found validated sufficient library conditions in less than a second, and always found validated sufficient library conditions in less than 30 seconds (see Tab. 1), and usually found validated sufficient library conditions in less than a second.
2. `chklibs` often inferred syntactically compact sufficient library conditions. `chklibs` usually needed to suggest less than 10 disjunctive clauses until it suggested a sufficient set of clauses. The clauses always contained less than 10 predicates (and usually contained less than 5 predicates), and with some exceptions discussed below, were small enough that a programmer should be able to reason about their validity.

In §4.1, we describe in detail our procedure for evaluating ChkAEQ. In §4.2, we present and analyze the results of applying ChkAEQ.

4.1 Experimental Procedure

Implementation. In §3, we defined the abductive equivalence problem AEQ for a simple language IMP, and presented an algorithm ChkAEQ that solves AEQ. `chklibs` solves the abductive equivalence problem for the LLVM [20] intermediate language. To implement `chklibs`, we extended the operational semantics of IMP (§3.1) to an operational semantics for the LLVM intermediate language, which included describing various language features such as structs and pointers. Such an extension is standard, and we omit its details in this paper. `chklibs` is implemented in about 5,000 lines of OCaml [5] code, and uses the Z3 theorem prover [8] to implement the abductive theorem prover ATP (App. A). `chklibs` simplifies each query and presents the query to the user as a conjunction of disjunctive clauses. We discuss simplifications that `chklibs` applies to queries in App. B.

Evaluation. To evaluate `chklibs`, we used it to validate a set of optimizations submitted to improve the performance of real-world applications. In particular, we collected a set of bug reports from the public bug databases of Apache software suite [1], Mozilla Suite [22], and MySQL database [23] that each reported a performance issue and included a patch to fix the issue. We compiled each original program and its patch to the LLVM intermediate language. If a program and patch were originally implemented in a language supported by LLVM, such as C or C++, then we compiled the programs by applying the appropriate LLVM compiler front-end (`clang` or `clang++`, respectively). Otherwise, we rewrote the program functions by hand in C source code and compiled the source to the LLVM intermediate language by applying `clang`.

We applied `chklibs` to each original and optimized program, and interacted with `chklibs` to find library conditions that `chklibs` determined to be sufficient, and which were valid under our understanding of the libraries and program functions described informally in the bug report. In other words, we served as the library oracle introduced in §3.3. For each benchmark, we observed whether or not `chklibs` found library conditions that we believed to be valid, measured the total time spent by `chklibs` to infer sufficient specifications, measured the number of queries issued by `chklibs`, and measured the size of each query.

4.2 Results and Analysis

Results. Tab. 1 contains the results of applying `chklibs`. Each row in Tab. 1 contains data for a benchmark original and optimized program. In particular, Tab. 1 contains the name of the program from which the benchmark was taken, the ID of the bug report in which the optimization was submitted, the number of lines of code of the original and optimized program functions, the number of lines output by `diff` [11] on the original and optimized programs, the time spent by `chklibs` to construct a validated simulation relation (not including the time spent by us to validate or refute a query posed by `chklibs`), the number of clauses on which `chklibs` queried the user, and average size of (i.e., the number

of all logical symbols in) the clauses, and the average number of predicates in each clause. The size of the clause is the number of logical symbols in the clause.

Analysis. The data presented in Tab. 1 indicates that `chklibs` can be applied to suggest sufficient library conditions for equivalence that can often be easily validated by a programmer. In benchmarks where `chklibs` took an unusually long time to find validated sufficient conditions (Apache Bug 34464 and MySQL Bug 38824), `chklibs` posed a proportionally large number of queries that we refuted. In benchmarks where `chklibs` queried the user on an unusually large set of clauses (Apache Bug 48778 and MySQL Bug 38824), the original and optimized programs called different library functions at a proportionately large set of callsites. In benchmarks where `chklibs` queried the user on unusually large formulas (Mozilla Bug 124686 and MySQL Bug 38824), the formulas typically were constructed from equality conditions over addresses in the program that seem unlikely to alias. We believe that the size of these queries could be reduced drastically by combining `chklibs` with a more sophisticated alias analysis, or a programmer with a more detailed understanding of the calling conventions of the original and optimized function.

5 Related Work

Much existing work has focused on determining the equivalence of programs. Translation validation [24, 26] is the problem of determining if a source program is equivalent to an optimized program, and is often applied to validate the correctness of the phases of an optimizing compiler. Regression verification [12] addresses the related problem of determining that a program and a similar revision of the program are equivalent. Semantic differencing [16] summarizes the different behaviors of two programs. Symbolic execution has been applied to determine the equivalence of loop-free programs [6]. In this work, we address the problem of taking an original and optimized program and inferring conditions on the libraries invoked by the programs that are sufficient to prove that the programs are equivalent, and that are validated by an oracle who understands the libraries. To solve the problem, we have extended an existing analysis for checking the equivalence of imperative programs with loops [24] to use the results of an abductive theorem prover.

The SymDiff project [17–19] shares our goal of determining under what conditions two programs are correct. Existing work in SymDiff takes a concurrent program, constructs a sequential version of the program, and treats the sequential version as a reference implementation, searching the concurrent program only for bugs triggered by inputs that cause no error in the sequential program [17]. Existing work on conditional equivalence [18] takes an original and optimized program and infers sufficient conditions on the inputs of a program under which the original and optimized programs are equivalent, where the space of conditions forms a lattice. In contrast, our work interacts with a user to infer sufficient conditions on the libraries invoked by an original and optimized program, and

represents conditions as logical formulas. Given that the spaces of conditions described in techniques based on conditional equivalence must form a lattice, it is not immediately clear how to extend such a technique to interact with a user who may refute an initial condition suggested by the technique.

Recent work has extended the problem of deciding if a program always satisfies an assertion to an abductive setting, in which the problem is to find assumptions on the state of a program that imply that the program satisfies an assertion, and are validated by an oracle that answers queries about program states [9]. That work presents an algorithm that constructs sufficient assumptions by finding a minimum satisfying assignment of variables in a given formula [10], universally quantifying all unassigned variables, and eliminating the quantified variables using symbolic reasoning. Our work extends a different traditional problem in program analysis, that of checking program equivalence, to an abductive setting. While work on abductive assertion checking assumes that the theories for describing states of a program support quantifier elimination (e.g., linear arithmetic), we consider inferring assumptions for theories that may describe arbitrary library functions. Accordingly, our analysis applies an abductive theorem prover that does not assume that the theories modeling the semantics of a program support quantifier elimination, and instead generates assumptions as Boolean combinations of equality predicates.

Our approach to the abductive equivalence problem collects information from a programmer about the libraries that a program uses by querying the programmer for the validity of purely equational formulas over the library functions, and propagates the logical consequence of the equalities to the rest of the analysis. Our approach is inspired by equality propagation [25], which is a technique for combining solvers for theories whose only shared predicate is equality to solve a formula defined in the combination of the theories. Essentially, our approach uses the programmer as a theory solver for the theory modeling library functions.

6 Conclusion

We have presented an interactive algorithm that takes a program and an optimization of the program, and interacts with a programmer to validate that the original and optimized program are semantically equivalent. The algorithm helps a programmer find a condition on library functions called by the program that implies that the original program is semantically equivalent to the optimized program. Our algorithm interacts with the programmer as a solver for a theory describing the library functions. We implemented our algorithm as a tool, and applied it to a set of programs and optimizations of real-world applications. For most optimizations, the tool quickly inferred sufficient library conditions that we could validate.

References

1. Apache, Jan. 2013. `http://apache.org`.

2. Apache bug #34464, Jan. 2013. http://issues.apache.org/bugzilla/show_bug.cgi?id=34464.
3. Apache Ant, May 2012. http://ant.apache.org.
4. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.
5. The Caml language, Oct. 2012. http://caml.inria.fr.
6. E. M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, 2003.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
8. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
9. I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *PLDI*, 2012.
10. I. Dillig, T. Dillig, K. L. McMillan, and A. Aiken. Minimum satisfying assignments for SMT. In *CAV*, 2012.
11. GNU diffutils, May 2010. http://www.gnu.org/software/diffutils/.
12. B. Godlin and O. Strichman. Regression verification. In *DAC*, 2009.
13. D. Gries. *The Science of Programming*. Springer, 1981.
14. W. R. Harris, G. Jin, S. Lu, and S. Jha. Validating library usage interactively, Jan. 2013. http://pages.cs.wisc.edu/~wrharris/validating_library_usage.pdf.
15. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
16. D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM*, 1994.
17. S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *POPL*, 2012.
18. M. Kawaguchi, S. K. Lahiri, and H. Rebelo. Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research, Oct 2010.
19. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *CAV*, 2012.
20. C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.
21. R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
22. Mozilla – home of the Mozilla Project, 2011. http://www.mozilla.org/.
23. MySQL: The world's most popular open source database, 2012. http://www.mysql.com/.
24. G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.
25. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
26. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.

A An Abductive Theorem Prover for AEQ

To construct an abductive simulation relation, ConsecSimRel (§3.3) applies an abductive theorem prover ATP. ATP takes a $\mathcal{T}$ formula φ and tries to construct a $\mathcal{T}_{lib}$ formula that implies φ. We now present an implementation of ATP that uses a theorem prover for $\mathcal{T}$ that is parametrized on a *solver* for $\mathcal{T}_{lib}$. The $\mathcal{T}_{lib}$ solver

Algorithm 3 An Abductive Theorem Prover ATP. Takes as input a $\mathcal{T}$ formula φ and tries to construct a consistent $\mathcal{T}_{lib}$ formula that implies φ. ConsAssumptions takes a set of models cms and constructs a $\mathcal{T}_{lib}$ formula that is satisfied by no model in cms. LibSlvr is a theory solver for $\mathcal{T}_{lib}$: it takes a conjunction ctx of equalities over $\mathcal{T}_{lib}$ terms and constructs a Boolean combination of $\mathcal{T}_{lib}$ equalities implied by ctx.

```
 1: function ATP(φ)
 2:     function ConsAssumptions(cms)
 3:         BlockConds := LearnBlock(cms)
 4:         function LibSlvr(ctx) return ApplyCond(BlockConds, ctx)
 5:         end function
 6:         if IsT*Valid[T_lib : LibSlvr](φ) then return BlockConds
 7:         else
 8:             cms' := cms ∪ { T*Model[T_lib : LibSlvr](¬φ) }
 9:             return ConsAssumptions(cms')
10:         end if
11:     end function
12:     return ConsAssumptions(∅)
13: end function
```

takes as input a conjunction ctx of equalities over $\mathcal{T}_{lib}$ terms (i.e., $\mathcal{T}_{lib}$ *equalities*), and constructs a a Boolean combination of $\mathcal{T}_{lib}$ equalities implied by ctx. Such a $\mathcal{T}$ prover can be implemented using a satisfiability-modulo theories [25] solver.

ATP is defined in Alg. 3. ATP takes as input a formula $\varphi \in \mathsf{Forms}(\mathcal{T})$ and constructs a library condition $C \in \mathsf{Forms}(\mathcal{T}_{lib})$ that implies φ. To construct C, ATP defines and applies a recursive function ConsAssumptions (Alg. 3, lines [2]–[11]) that takes a set of $\mathcal{T}$ models cms, and constructs a library condition that implies φ and for which no model in cms is a model. ATP returns the condition constructed by applying ConsAssumptions to an empty set of $\mathcal{T}$ models (Alg. 3, line [12]).

ConsAssumptions applies a function LearnBlock that takes a set of $\mathcal{T}$ models cms and constructs a *blocking condition* BlockConds for cms, which is a consistent $\mathcal{T}_{lib}$ formula for which no model in cms is a model (Alg. 3, line [3]). We discuss one practical implementation of LearnBlock in §A.1.

After ConsAssumptions constructs a blocking condition BlockConds, it defines a $\mathcal{T}_{lib}$-solver LibSlvr (Alg. 3, lines [4]–[5]) that asserts BlockConds as axioms of $\mathcal{T}_{lib}$. In Alg. 3, a theory solver is specified as a function that takes as input a conjunction ctx of equalities over terms in the theory, and constructs a Boolean combination of equalities over terms in the theory implied by ctx. Such an interface corresponds to the interface for theory solvers provided by modern SMT solvers [8]. LibSlvr takes a conjunction of $\mathcal{T}_{lib}$ equalities ctx and constructs as its conclusion the result of applying the blocking condition BlockConds to ctx (Alg. 3, line [4]). LibSlvr constructs the conclusion by applying a function ApplyCond. We describe one practical implementation of ApplyCond in §A.1.

ConsAssumptions applies a $\mathcal{T}$ theorem prover with LibSlvr as the solver for $\mathcal{T}_{lib}$ to determine if BlockConds implies φ (Alg. 3, line [6]). If BlockConds implies φ, then ConsAssumptions returns BlockConds (Alg. 3, line [6]). Otherwise, ConsAssumptions applies the $\mathcal{T}$ theorem prover to obtain a model of the negation of φ under BlockConds, and adds the model to its set of countermodels (Alg. 3, line [8]). ATP then applies itself recursively to the extended set of countermodels.

A.1 Implementations of LearnBlock and ApplyCond

The abductive theorem prover ATP (App. A) applies a function LearnBlock to learn a blocking condition from a set of $\mathcal{T}$ models, and a function ApplyCond that applies a blocking condition to construct a conclusion as a logical combination of equalities over $\mathcal{T}_{\text{IMP}}$ terms. ATP, and thus ChkAEQ, can be implemented using any implementation of LearnBlock and ApplyCond. To develop an implementation of ChkAEQ that is useful in practice, one must design a space of conditions that can be learned efficiently from a set of models, and which can be applied to a logical context of equalities. Our implementation of ATP defines a space of conditions that are conjunctions of universally quantified implications over $\mathcal{T}_{lib}$ equalities. Our implementation of LearnBlock for this space of conditions takes a set of countermodels, applies a greedy algorithm to learn a set of minimal disjunctive clauses of which no countermodel is a model, universally quantifies all constants in the clauses, and defines a clause from each implication. Our implementation of ApplyCond takes a conjunction of equalities over $\mathcal{T}_{lib}$ terms, constructs the equivalence classes over $\mathcal{T}_{lib}$ terms implied by the context, and attempts to instantiate each universally quantified implication. While such a space of conditions does not allow ChkAEQ to solve all AEQ problems, the space does allow ChkAEQ to solve AEQ problems defined by practical optimizations (we discuss our practical experience applying our implementation of ChkAEQ in §4).

B Practical Query Simplifications

The algorithm ChkAEQ presented in §3.3 constructs a consistent condition on library functions and queries the library oracle by passing the formula directly to the oracle. `chklibs` aggressively simplifies each formula on which it queries its user. In particular, when ATP finds a sufficient condition under which a formula is valid, it greedily prunes the disjunction of predicates by dropping as many predicates as possible while still maintaining a consistent library condition. This pruning both causes ChkAEQ and ATP to converge faster in practice, and allows ChkAEQ to query its oracle on smaller formulas. Furthermore, `chklibs` constructs a library condition as a formula in conjunctive normal form (CNF). Rather than present the entire CNF formula to the library oracle, `chklibs` presents each disjunctive clause of the CNF formula to the oracle individually. When `chklibs` presents a disjunctive clause, it performs standard symbolic operations on the clause to transform the clause into an implication from a conjunction of non-negated equality predicates to a disjunction of non-negated equality

predicates. Previous work in applying abductive reasoning to prove the validity of program assertions [10] applied similar symbolic manipulations in order to present simple queries to a user.