Towards Modularly Comparing Programs using Automated Theorem Provers

Chris Hawblitzel[1], Ming Kawaguchi[2], Shuvendu K. Lahiri[1], and Henrique Rebêlo[3]

[1] Microsoft Research, Redmond, WA, USA
[2] University of California, San Diego
[3] Federal University of Pernambuco, Brazil

Abstract. In this paper, we present a general framework for modularly comparing two (imperative) programs that can leverage single-program verifiers based on automated theorem provers. We formalize (i) *mutual summaries* for comparing the summaries of two programs, and (ii) *relative termination* to describe conditions under which two programs relatively terminate. The two rules together allow for checking correctness of interprocedural transformations. We also provide a general framework for dealing with unstructured control flow (including loops) in this framework. We demonstrate the usefulness and limitations of the framework for verifying equivalence, compiler optimizations, and interprocedural transformations.

1 Introduction

The ability to compare two programs statically has applications in various domains. Comparing successive versions of a program for behavioral equivalence across various refactorings and ensuring that bug fixes and feature additions do not introduce compatibility issues, is crucial to ensure smooth upgrades [3]. Comparing different versions of a program obtained after various compiler transformations (*translation validation*) is useful to ensure that the compiler does not change the semantics of the source program [9, 8]. There are two enablers for program comparison compared to the more general problem of (single) program verification. First, one of the two programs serves as an implicit specification for the other program. Second, exploiting simple and automated abstractions for similar parts of the program can lead to greater automation and scalability.

Although several systems have been developed in recent years for equivalence checking of imperative programs, there has been a lack of general framework for comparing programs. Current systems provide solutions to specific instances of the problem — translation validators focus on intraprocedural loop optimizations [8], regression verification focuses on simple interprocedural refactorings [3].

In this paper, we describe a framework for comparing programs modularly. We develop two contracts for comparing two programs: (i) First, we formalize *mutual summaries* to relate the summaries of two (possibly recursive) procedures. Mutual summaries naturally generalize postconditions used for single

$$
\begin{array}{lll}
c & \in \{\ldots,-1,0,1,\ldots\} & \\
\mathtt{x} & \in \mathit{Vars} & \\
R & \in \mathit{Relations} & \\
U & \in \mathit{Functions} & \\
e & \in \mathit{Expr} & ::= \mathtt{x} \mid c \mid U(e,\ldots,e) \mid \mathsf{old}(e) \mid \langle e,e\rangle \mid e.1 \mid e.2 \\
\phi & \in \mathit{Formula} & ::= \mathsf{true} \mid \mathsf{false} \mid e \ \mathsf{relop}\ e \mid \phi \wedge \phi \mid \neg\phi \mid R(e,\ldots,e) \mid \forall u.\phi \\
s & \in \mathit{Stmt} & ::= \mathsf{skip} \mid \mathsf{assert}\ \phi \mid \mathsf{assume}\ \phi \mid \mathtt{x} := e \mid \mathsf{havoc}\ \mathtt{x} \mid \\
 & & \quad s;s \mid \langle s,s\rangle \mid s \diamond s \mid s \bowtie s \mid \mathtt{x} := \mathsf{call}\ f(e,\ldots,e) \\
p & \in \mathit{Proc} & ::= \mathsf{int}\ f(\mathtt{x} : \mathsf{int},\ldots) : \mathtt{r}\ \{\ s\ \}
\end{array}
$$

Fig. 1. A simple programming language.

program verification. (ii) Second, we formalize a *relative termination* specification that describes a condition $RT(f,h)$ on inputs of two procedures f and h under which the procedure h terminates whenever f terminates. Such contracts are useful to ensure that transformations do not change the terminating executions, and are important for ensuring that two transformations compose. We then provide a proof rule for checking mutual summaries and relative termination modularly. We show that these checks can be encoded using modular (single) program verifiers, and can be discharged efficiently using modern satisfiability modulo theories (SMT) solvers [2]. Finally, we provide a general framework for dealing with unstructured control flow (including loops) in this framework.

We demonstrate the usefulness of our approach on illustrative examples from equivalence checking, including conditional equivalence checking and translation validation. We encode proofs of various compiler loop optimizations such as software pipelining and loop unrolling. Our framework currently lacks the automation provided for specific forms of equivalence checking (e.g. automatically synthesizing a class of simulation relations for compiler transformations [8]). On the other hand, we show examples of comparing two programs with interprocedural changes for eliminating non-tail recursion (§4.3), monotonic behavior (§3.1), conditional equivalence (§4.3) and refactorings (§4.3) that were not amenable to automated theorem provers. We are currently incorporating the ideas in this paper into SymDiff [5], a language agnostic semantic diff framework that uses the modular program verifier Boogie [1], and the Z3 SMT solver [2].

2 Background

Figure 1 describes a programming language with recursive procedures and an assertion language. Loops and unstructured jumps can be translated into this language (§4.1). The language supports variables (*Vars*) and various operations on them. Expressions (*Expr*) can be variables, constants, or the result of applying a function U to a list of expressions. The expression $\mathsf{old}(e)$ refers to the value of e at the entry to a procedure. The expressions $e.1$ and $e.2$ extract the first and second components of a pair $\langle e_1, e_2\rangle$. *Formula* represents Boolean valued

$$\langle \mathsf{skip}, \sigma\rangle \Downarrow \sigma \quad \langle \mathsf{assert}\ \phi, \sigma\rangle \Downarrow \sigma \quad \langle g := e, \sigma\rangle \Downarrow \mathit{eval}(e[\sigma/g]) \quad \langle \mathsf{havoc}\ g, \sigma\rangle \Downarrow \sigma'$$

$$\frac{\phi[\sigma/g]}{\langle \mathsf{assume}\ \phi, \sigma\rangle \Downarrow \sigma} \qquad \frac{\begin{array}{c}\langle s_f, \sigma\rangle \Downarrow \sigma' \\ \text{where } s_f \text{ is the body of } f\end{array}}{\langle \mathsf{call}\ f(), \sigma\rangle \Downarrow \sigma'} \qquad \frac{\begin{array}{c}\langle s_1, \sigma_1\rangle \Downarrow \sigma_1' \\ \langle s_2, \sigma_2\rangle \Downarrow \sigma_2'\end{array}}{\langle \langle s_1, s_2\rangle, \langle \sigma_1, \sigma_2\rangle\rangle \Downarrow \langle \sigma_1', \sigma_2'\rangle}$$

$$\frac{\begin{array}{c}\langle s_1, \sigma\rangle \Downarrow \sigma' \\ \langle s_2, \sigma'\rangle \Downarrow \sigma''\end{array}}{\langle s_1; s_2, \sigma\rangle \Downarrow \sigma''} \qquad \frac{(\langle s_1, \sigma\rangle \Downarrow \sigma') \ \vee \ (\langle s_2, \sigma\rangle \Downarrow \sigma')}{\langle s_1 \ \diamond \ s_2, \sigma\rangle \Downarrow \sigma'} \qquad \frac{\begin{array}{c}\langle s_1, \sigma\rangle \Downarrow \sigma' \\ \langle s_2, \sigma\rangle \Downarrow \sigma'\end{array}}{\langle s_1 \ \bowtie \ s_2, \sigma\rangle \Downarrow \sigma'}$$

Fig. 2. Dynamic semantics.

expressions and can be the result of relational operations on *Expr*, Boolean operations ($\{\wedge, \neg\}$), or quantified expressions ($\forall u.\phi$).

A state σ of a program at a given program location is a valuation of the variables in scope, which may include procedure parameters, locals and global variables. Figure 2 presents big-step dynamic semantics $\langle s, \sigma\rangle \Downarrow \sigma'$, which says that statement s executes to completion, transforming the initial state σ into a new state σ'. For simplicity, the formalizations in the paper (e.g. Figure 2) assume that the program contains only one variable, a global variable named g. The value of the global implicitly defines the state in such cases. (Note that g can hold tuples and arrays, which can be used to encode additional variables, procedure parameters, and procedure return values.) Most statements in *Stmt* are standard, we only describe the non-standard ones here: The assignment statement is standard (we assume an evaluation function $\mathit{eval}(e)$ that evaluates closed expressions to values). havoc x scrambles the value of a variable x to an arbitrary value. $s \ \diamond \ t$ denotes a *demonic* non-deterministic choice to either execute statements in s or t, and can be used to model conditional statements [1]. The statement $s \ \bowtie \ t$ denotes *angelic* non-deterministic choice, where the choice of executing s or t may be made in whichever way is most beneficial to the verification process. Finally, the statement $\langle s_1, s_2\rangle$ requires that the current state be a pair value ($\sigma = \langle \sigma_1, \sigma_2\rangle$); if this is satisfied, then $\langle s_1, s_2\rangle$ evaluates s_1 in state σ_1 to produce a new state σ_1', separately evaluates s_2 in state σ_2 to produce a new state σ_2', and then combines the two new states into a single new state that is a pair value: $\langle \sigma_1', \sigma_2'\rangle$. We do not expect programmers to use the statements $s \ \bowtie \ t$ and $\langle s_1, s_2\rangle$ directly; these statements are used for instrumenting programs when checking relative termination and mutual summaries respectively.

Figure 3 presents axiomatic static semantics for statements s, expressed as a weakest (liberal) precondition $\phi = wp(s, \phi')$. The definition of $wp(s, \phi')$ is standard except for the $\langle s_1, s_2\rangle$ statement and call statement. The definition of $wp(\langle s_1, s_2\rangle, \phi)$

$$(wp(s_1, wp(s_2, \phi[\langle g_1, g_2\rangle/g][g/g_2])[g_2/g][g/g_1])[g_1/g])[g.1/g_1, g.2/g_2]$$

$wp(\mathsf{skip}, \phi) = \phi$ $\quad$ $wp(\mathsf{assert}\ \phi', \phi) = \phi' \wedge \phi$

$wp(\mathsf{assume}\ \phi', \phi) = \phi' \implies \phi$ $\quad$ $wp(g := e, \phi) = \phi[e/g]$

$wp(\mathsf{havoc}\ g, \phi) = \forall g.\ \phi$ $\quad$ $wp(s_1; s_2, \phi) = wp(s_1, wp(s_2, \phi))$

$wp(s_1 \ \diamond\ s_2, \phi) = wp(s_1, \phi) \wedge wp(s_2, \phi)$ $\quad$ $wp(s_1 \ \bowtie\ s_2, \phi) = wp(s_1, \phi) \vee wp(s_2, \phi)$

$wp(\mathsf{call}\ f(), \phi) = \forall g'.\ R_f(g, g') \implies \phi[g'/g]$

$$wp(\langle s_1, s_2\rangle, \phi) = (wp(s_1, wp(s_2, \phi[\langle g_1, g_2\rangle/g][g/g_2])[g_2/g][g/g_1])[g_1/g])[g.1/g_1, g.2/g_2]$$

Fig. 3. Static semantics.

is long but not particularly deep; intuitively, it just extracts the two components of $g.1$ and $g.2$ the input state g into temporary variables g_1 and g_2, and then shuffles these values in and out of g to capture the effects of evaluating s_1 on g_1 and s_2 on g_2. Similarly, the effect of a call to a procedure f is simply replaced by an uninterpreted relation $R_f(g, g')$,

$$wp(\mathsf{call}\ f(), \phi) = \forall g'.\ R_f(g, g') \implies \phi[g'/g]$$

where g is the state before the call and g' is the state after the call completes. We often use $R = \bigcup_f \{R_f\}$ to refer to set of relation symbols over all the procedures. The following proposition connects the dynamic and static semantics:

Proposition 1. (Basic Soundness) *If $\langle s, \sigma\rangle \Downarrow \sigma'$ and $wp(s, \phi)[\mathsf{old}(g)/g]$ is valid and no symbol in R appears free in ϕ, then $\phi[\sigma/\mathsf{old}(g), \sigma'/g]$ is valid.*

In the next section, we will illustrate how the mutual summaries and the relative termination contracts constrain the relation R_f for a procedure.

3 Mutual summaries and relative termination

A *program* P consists of a set of procedures $\{f_1, \ldots, f_k\}$, identified by their names. We let f, h, f_i, h_i range over procedure names. The set P contains a union of procedures from two versions of a program. We use the notation $a \doteq \lambda f, h.\ \phi(f, h)$ to be an indexed (by a pair of procedures) set of formulas such that $\phi(f, h)$ denotes the formula for the pair (f, h). We extend this notation to refer to an indexed set of expressions, constants, sets of states, *etc.*

3.1 Mutual summaries

For any pair of procedures $f \in P$ and $h \in P$, a mutual summary $MS(f, h)$ is a relation over the input and output states of f and h. It is expressed as a formula over two copies of the input and the output variables. In general, f and h may have different sets of parameters and return values. In the case where f and h both take a single parameter x, return a single return value r, and access a single global variable g, the relation looks like:

$$\lambda x_1, x_2, g_1, g_2, r_1, r_2, g'_1, g'_2.\ \phi(x_1, x_2, g_1, g_2, r_1, r_2, g'_1, g'_2)$$

```
int g;
void Foo1(int x){                              void Foo2(int x){
     if (x < 100){                                  if (x < 100){
        g := g + x;                                    g := g + 2*x;
        Foo1(x + 1);                                   Foo2(x + 1);
     }                                              }
}                                              }
```

Fig. 4. Running example.

where x_i, g_i, r_i and g'_i refer to the state of the parameters, input globals, return and the output globals for the i-th procedure. For brevity, we will identify a mutual summary directly with $\phi(x_1, x_2, g_1, g_2, r_1, r_2, g'_1, g'_2)$, instead of the relation (that is, avoid the λ). The semantics from Figures 2 and 3 contain just one variable g, so we write (using a curried function that accepts a pair of pre-states and a pair of post-states):

$$\lambda g_1, g_2.\lambda g'_1, g'_2.\ \phi(g_1, g_2, g'_1, g'_2)$$

Definition 1 (mutual summaries). *For procedures f and h with bodies s_f and s_h, $MS(f,h)$ holds if for every tuple of states $(\sigma_f, \sigma'_f, \sigma_h, \sigma'_h)$ such that $\langle s_f, \sigma_f\rangle \Downarrow \sigma'_f$ and $\langle s_h, \sigma_h\rangle \Downarrow \sigma'_h$, $MS(f,h)\langle \sigma_f, \sigma_h\rangle\langle \sigma'_f, \sigma'_h\rangle$ evaluates to true.*

Example 1. Consider the two programs in Figure 4. Consider the following mutual summary $MS(\texttt{Foo1}, \texttt{Foo2})$ for this pair of procedures:

$$(x_1 = x_2 \wedge x_1 \geq 0 \wedge g_1 \leq g_2) \implies (g'_1 \leq g'_2)$$

The summary states that if the procedures `Foo1` and `Foo2` are executed in two states where the respective parameters are equal and greater than 0, and if the value of the global at entry to `Foo1` is less than or equal to the value of the global at entry to `Foo2`, and if both procedures terminate, then the value of the global at exit from `Foo1` will be less than or equal to the value of the global at exist from `Foo2`.

3.2 Relative Termination

One difficulty with using mutual summaries is that they do not *compose*, since they only talk about partial correctness. Consider three procedures `A1`, `A2` and `A3` and mutual summaries $MS(\texttt{A1}, \texttt{A2})$ and $MS(\texttt{A2}, \texttt{A3})$ that express that `A1` and `A2` (respectively `A2` and `A3`) are equivalent when both procedures terminate on an input. We cannot conclude that `A1` and `A3` are equivalent when both procedures terminate on an input, since `A2` may not terminate on any input.

The relative termination specification expresses the circumstances in which f's termination implies h's termination. For any pair of procedures $f \in P$ and $h \in P$, a relative termination specification $RT(f,h)$ is a relation over the input

$$
\begin{array}{l}
\mathit{AXIOMS} = \mathit{AXIOMS}_{MS} \wedge \mathit{AXIOMS}_{RT} \\
\mathit{AXIOMS}_{MS} = \forall f_1, f_2, \sigma_1, \sigma_1', \sigma_2, \sigma_2'. \\
\quad R_{f_1}(\sigma_1, \sigma_1') \wedge R_{f_2}(\sigma_2, \sigma_2') \implies \mathit{MS}(f_1, f_2)\langle \sigma_1, \sigma_2 \rangle \langle \sigma_1', \sigma_2' \rangle \\
\mathit{AXIOMS}_{RT} = \forall f_1, f_2, \sigma_1, \sigma_1', \sigma_2. \\
\quad R_{f_1}(\sigma_1, \sigma_1') \wedge \mathit{RT}(f_1, f_2)\langle \sigma_1, \sigma_2 \rangle \implies \exists \sigma_2'.\ R_{f_2}(\sigma_2, \sigma_2')
\end{array}
$$

$$
\begin{array}{l}
\mathit{CONDITIONS} = \\
R \text{ does not occur free in } \mathit{MS}, \text{ and} \\
R \text{ does not occur free in } \mathit{RT}, \text{ and} \\
\forall f_1, f_2. \\
\ (\mathit{AXIOMS} \implies wp(\langle s_{f_1}, s_{f_2} \rangle, \mathit{MS}(f_1, f_2)\ \mathsf{old}(g)\ g)[\mathsf{old}(g)/g]) \wedge \\
\ (\mathit{AXIOMS} \wedge \mathit{RT}(f_1, f_2)\ g \implies wp(\langle s_{f_1}, at(s_{f_2}) \rangle, \mathsf{true}))
\end{array}
$$

Fig. 5. Conditions and axioms.

states of f and h. It is expressed as a formula over two copies of the input variables. The relative termination specification $RT(f, h)$ is a relation

$$\lambda x_1, x_2, g_1, g_2.\ \phi(x_1, x_2, g_1, g_2)$$

where x_i and g_i refer to the state of the parameters and input globals for the i-th procedure. In general, f and h may have different parameters. For the semantics from Figures 2 and 3, where there are no parameters, we write:

$$\lambda g_1, g_2.\ \phi(g_1, g_2)$$

Definition 2 (relative termination). *For procedures f and h with bodies s_f and s_h, $RT(f, h)$ holds if for every tuple of states (σ_f, σ_h) such that there exists a σ_f' such that $\langle s_f, \sigma_f \rangle \Downarrow \sigma_f'$ and $RT(f, h)\langle \sigma_f, \sigma_h \rangle$ is true, there is some σ_h' such that $\langle s_h, \sigma_h \rangle \Downarrow \sigma_h'$.*

Note that we do not insist that every execution from a pre-state eventually terminates, but rather at least one. For the example in Figure 4, although $RT(\texttt{Foo1}, \texttt{Foo2}) = \texttt{true}$ is the weakest condition for relative termination (since both `Foo1` and `Foo2` always terminate), proving such a relative termination specification requires reasoning about the two programs separately using ranking functions. We later show that a stronger condition $RT(\texttt{Foo1}, \texttt{Foo2}) = x_1 \leq x_2$ can be proved modularly without any other proof rules.

3.3 Modular checking

We now describe a method to decompose the checking that a program P satisfies a set of mutual summaries MS and relative termination specifications RT.

Figure 5 expresses the assumptions in AXIOMS and the checks for guaranteeing the mutual summaries and relative termination as a condition in $\mathit{CONDITIONS}$ that must be satisfied. The AXIOMS consists of assumptions for MS and RT specifications respectively. The AXIOMS_{MS} assumes $MS(f_1, f_2)$ on the pre-post states of f_1 and f_2. The AXIOMS_{RT} assumes f_2 terminates whenever it starts from a state σ_2 that is related (by $RT(f_1, f_2)$) to a terminating input state σ_1 of f_1 (we defer discussion of $at()$ until we explain AXIOMS_{RT}).

The check for mutual summaries is given by:

$$\mathit{AXIOMS} \Longrightarrow wp(\langle s_f, s_h \rangle, MS(f,h)\ \mathsf{old}(g)\ g)[\mathsf{old}(g)/g]$$

where s_f and s_h are the bodies of the procedures f and h. Intuitively, this formula prescribes a sequence of steps for checking that $MS(f,h)$ holds. First, we assume AXIOMS holds. Second, we assign the global state variable g an initial value of $\mathsf{old}(g)$. Third, we symbolically execute the statement $\langle s_f, s_h \rangle$, which assigns a new state to the variable g. (This has the effect of executing s_f on $g.1$ and separately executing s_g on $g.2$.) Finally, we assert that in this new state g, relative to the old state $\mathsf{old}(g)$, the mutual summary $MS(f,h)$ holds. Observe that this is analogous to modular (single) program verification, where we symbolically execute a single procedure body s_f and than assert that f's postcondition holds. The key novelty in mutual summaries is that the checking process executes *two* procedure bodies, and asserts a summary that can mention the state of both procedures.

The checking procedure ensures that if s_f and s_g execute on concrete states σ_f and σ_g, then the mutual summary $MS(f,h)$ relates the new states σ'_f and σ'_g to the old states σ_f and σ_g:

Theorem 1. *For procedures f and h with bodies s_f and s_h, if CONDITIONS is satisfied then $MS(f,h)$ holds; that is, if $\langle s_f, \sigma_f \rangle \Downarrow \sigma'_f$ and $\langle s_h, \sigma_h \rangle \Downarrow \sigma'_h$ hold for any $\sigma_f, \sigma'_f, \sigma_h, \sigma'_h$, then $MS(f,h)\langle \sigma_f, \sigma_h \rangle \langle \sigma'_f, \sigma'_h \rangle$ is true.*

The procedure bodies s_f and s_h may contain call statements. For example, the procedure bodies in Figure 4 contain recursive calls to the procedures. Suppose that procedure body s_f contains a call statement $\mathsf{call}\ f'()$ and procedure body s_h contains a call statement $\mathsf{call}\ h'()$. The weakest precondition (Figure 3) inserts an assumption $R_{f'}(g_f, g'_f)$, where g_f and g'_f are the states before and after the $\mathsf{call}\ f'()$ statement. Similarly, the weakest precondition inserts an assumption $R_{h'}(g_h, g'_h)$ for $\mathsf{call}\ h'()$. These assumption may trigger AXIOMS_{MS} (Figure 5), which then produces an assumption about $MS(f,h)\langle g_f, g_h \rangle \langle g'_f, g'_h \rangle$. This assumption may be used to help prove the weakest precondition for $\langle s_f, s_h \rangle$, so that mutual summaries for recursive procedures are established inductively, assuming mutual summaries for callees while checking summaries for callers. Observe that this is analogous to modular (single) program verification, where we assume the postconditions of callees while checking the contracts in the caller.

We now show how RT are checked modularly. Figure 5 imposes the following condition for guaranteeing properties of relative termination:

$$\mathit{AXIOMS} \wedge RT(f,h)\ g \Longrightarrow wp(\langle s_f, at(s_h) \rangle, \mathsf{true})$$

Essentially, the formula requires that the weakest precondition of $\langle s_f, at(s_h) \rangle$ be implied by the axioms and the termination condition $RT(f,h)$. Figure 6 defines $at(s_h)$ as a transformation on s_h that include inserting an assert statement before

$$\begin{array}{ll}
at(\mathsf{skip}) = \mathsf{skip} & at(s_1 ; s_2) = at(s_1); at(s_2) \\
at(\mathsf{assert}\ \phi) = \mathsf{assert}\ \phi & at(\langle s_1 , s_2 \rangle) = \mathsf{assert\ false} \\
at(\mathsf{assume}\ \phi) = \mathsf{assert}\ \phi & at(s_1\ \diamond\ s_2) = at(s_1)\ \bowtie\ at(s_2) \\
at(g := e) = g := e & at(s_1\ \bowtie\ s_2) = \mathsf{assert\ false} \\
at(\mathsf{havoc}\ g) = \mathsf{havoc}\ g & at(\mathsf{call}\ f()) = \mathsf{assert}\ (\exists g'.\ R_f(g, g')); \mathsf{call}\ f()
\end{array}$$

Fig. 6. Assertions for checking relative termination. $at(s)$ replaces a statement with a new statement.

each call in s_h, and converting each assume statement in s_h into an assertion. The purpose of checking this is to verify that all the termination assertions in $at(s_h)$ hold, where each termination assertion verifies that a potentially non-terminating statement actually terminates. In particular, call statements may fail to terminate and assume statements may block. If these inserted assertions are satisfied, then s_h is guaranteed to terminate:

Theorem 2. *For procedures f and h with bodies s_f and s_h, if CONDITIONS is satisfied then $RT(f, h)$ holds; that is, if $\langle s_f, \sigma_f \rangle \Downarrow \sigma'_f$ and $RT(f, h)\langle \sigma_f, \sigma_h \rangle$ is valid for any $\sigma_f, \sigma'_f, \sigma_h$, then there is some σ'_h such that $\langle s_h, \sigma_h \rangle \Downarrow \sigma'_h$.*

As with mutual summaries, relative termination is assumed for callees when checking termination of the callers, so that relative termination for recursive procedures can be established inductively.

Given these rules, one can prove that the $MS(\texttt{Foo1}, \texttt{Foo2}) = (x_1 = x_2 \wedge x_1 \geq 0 \wedge g_1 \leq g_2) \implies (g'_1 \leq g'_2)$ holds for Figure 4. One can similarly prove that $RT(\texttt{Foo1}, \texttt{Foo2}) = x_1 \leq x_2$ holds, assuming it holds for nested pairs of calls. In both cases, we use the fact that whenever $x_1 \leq x_2$, `Foo2` cannot execute the nested recursive call to itself without `Foo1` calling itself. Although the condition $RT(\texttt{Foo1}, \texttt{Foo2}) = x_1 \geq x_2$ holds, it cannot be proved modularly using only these proof rules. This is expected, as these rules are only sound, but not complete.

3.4 Proof sketch

We have proven the main theorems (Theorem 1 and Theorem 2).[1] The key lemma is a proof that the axioms in Figure 5 are valid.

Lemma 1. (Full Soundness) *If CONDITIONS is satisfied and $\langle s, \sigma \rangle \Downarrow \sigma'$ and $(AXIOMS \implies wp(s, \phi)[\mathsf{old}(g)/g])$ is valid and no symbol in R appears free in ϕ, then $\phi[\sigma/\mathsf{old}(g), \sigma'/g]$ is valid.*

The main challenge in the proof of this lemma is that the validity of the axioms depend on the conditions in Figure 5, which in turn mention the axioms. To break this circularity, we build up the axiom validity inductively on the call depth (maximum number of nested calls) in an execution $\langle s, \sigma \rangle \Downarrow \sigma'$. The base

[1] Detailed proofs are available off the extended technical report page at `http://research.microsoft.com/apps/pubs/?id=154989`.

```
void MutualCheck⟨f, h⟩
    (x_f : int, x_h : int){
    chkTerm := false;
    g_f := g;
    inline r_f := call f(x_f);
    g'_f := g;
    havoc g;
    g_h := g;
    inline r_h := call h(x_h);
    g'_h := g;
    assert
        MS(f, h)(x_f, x_h, g_f, g_h,
            r_f, r_h, g'_f, g'_h);
}
```

```
void RelTermCheck⟨f, h⟩
    (x_f : int, x_h : int){
    g_f := g;
    chkTerm := false;
    inline r_f := call f(x_f);
    g'_f := g;
    havoc g;
    g_h := g;
    assume
        RT(f, h)(x_f, x_h,
            g_f, g_h);
    chkTerm := true;
    inline r_h := call h(x_h);
    g'_h := g;
}
```

```
axiom(
$\forall x_1, x_2, g_1, g_2, r_1, r_2, g'_1, g'_2.$
$\{R_f(x_1, g_1, r_1, g'_1),$
$R_h(x_2, g_2, r_2, g'_2)\}$
$(R_f(x_1, g_1, r_1, g'_1) \wedge$
$R_h(x_2, g_2, r_2, g'_2))$
 $\Longrightarrow$
$MS(f, h)(x_1, x_2, g_1, g_2,$
$r_1, r_2, g'_1, g'_2))$

axiom(
$\forall x_1, x_2, g_1, g_2.$
$\{RT(f, h)(x_1, x_2, g_1, g_2)\}$
$(RT(f, h)(x_1, x_2, g_1, g_2)$
$\wedge R_f(x_1, g_1, r_1, g'_1))$
 $\Longrightarrow$
$(\exists r_2, \exists g'_2.\ R_h(x_2, g_2, r_2, g'_2)))$


free post $R_f$(x, old(g), r, g)
pre  chkTerm $\Longrightarrow$
        $\exists r,\ \exists g'.R_f$(x, g, $r$, $g'$)
modifies g
int f(x : int) : r;
```

Fig. 7. Encoding the rules into a modular program verifier Boogie.

case uses empty relations $R_{f_1} = \emptyset, \ldots, R_{f_k} = \emptyset$, meaning that there are no calls (call depth 0). The inductive case assumes relations $R_{f_1}, \ldots, R_{f_k}$ for call depth n, and increases the membership of these relations to include executions with call depth $n + 1$.

3.5 Encoding in Boogie

By exploiting the close analogy between mutual summaries, relative termination, and traditional modular (single) program verification, we can use automated single-program verification tools like Boogie to check mutual summaries and relative termination for a subset of programs described in Figure 1. We restrict ourselves to the case of programs that do not contain any angelic choice statements ($s \bowtie t$), and the only use of a demonic choice ($s \diamond t$) or an assume statement in the program syntax comes from the modeling of conditional statements. These restrictions along with absence of loops ensure that the only source of non-termination comes from nested procedure calls. Figure 7 shows an encoding of the axioms and conditions from Figure 5.

First, we define the predicate R_f for each procedure f over the input and output symbols, and add it as a "free" postcondition for f. The "free" postconditions of a procedure are unchecked postconditions that are only assumed at call sites, but never asserted. The precondition (guarded by a ghost global variable `chkTerm`) captures the assertion for checking relative termination.

Second, we define a procedure MutualCheck$\langle f, h\rangle$ that checks a mutual summary $MS(f, h)$. Note that the global variable `chkTerm` is set to false — this has the effect of turning off the relative termination assertions at call-sites. We write "inline $\mathtt{r} :=$ call $f(\mathtt{x})$" to inline the body of f (upto calls). The assert checks the mutual summary $MS(f,h)$ after executing f and h on their copies of globals.

Third, we define a procedure $\textsc{RelTermCheck}\langle f,h\rangle$ that defines how to check the relative termination of h with respect to f under $RT(f,h)$. The setting `chkTerm` = true enables assertions of termination before potentially nonterminating statements while checking the body of h.

Finally, the axiom(.) encode the axioms in $AXIOMS$. Each axiom has a set of *triggers* that control when the axioms are instantiated [2]. The triggers represent a list of expressions inside $\{.\}$, containing all the bound variables in a quantified axiom.

4 Applications

In this section, we show the application of our approach towards various examples of intraprocedural and interprocedural transformations. [2]

4.1 Loops and unstructured control

In this section, we provide a general framework for translating arbitrary unstructured control flow graphs (including loops) into recursive procedures. Unstructured control flow is fairly common when dealing with low-level programs such as binaries. The general scheme requires that certain locations in a program be decorated as special *function labels* (`FLABEL`). Given a program where every cycle passes through at least one function label, the following simple algorithm transforms this program into a set of mutually recursive procedures. First, each function label becomes a procedure, whose parameters are all local variables and procedure parameters in scope. Second, the body for each procedure is the collection of statements reachable from that procedure's function label via paths that do not pass through a function label. Finally, each goto statement to a function label (or implicit fall-through to a function label) becomes a tail-recursive call to the procedure for that function label. Notice that in the second step, the same statements might be included separately in different procedures, if those statements are reachable from different function labels. In the worst case, each statement could be included in each generated procedure, so the worst-case size of the resulting program is the product of the original program size and the number of function labels. Figure 8 shows an example of such loop extraction.

Although the general scheme allows the user flexibility in the choice of `FLABELS` to eliminate loops, one can automate the extraction for structured programs. For such programs, it suffices to identify the set of loop heads as `FLABEL`. We have implemented a variant of this scheme in Boogie to automate the translation of structured loops into tail-recursive procedures. [3] For the examples in this paper, we however explicitly mention the set of `FLABEL` locations.

[2] Detailed Boogie examples used in this paper are available off the extended technical report page at `http://research.microsoft.com/apps/pubs/?id=154989`.

[3] The exact Boogie options to be specified are `''/printInstrumented /extractLoops /deterministicExtractLoops ''`.

4.2 Intraprocedural translation validation

This section describes the use of mutual summaries to perform (intraprocedural) translation validation [9], focusing on the validation of compiler loop optimizations. The validation consists of three steps: (1) eliminating unstructured control (including loops), (2) providing mutual summaries, (3) user-specified *inlining* of calls to recursive procedures zero or more times to express the effect of loop optimizations such as loop unrolling.

In describing the examples in this section, we follow the approach by Kundu et al. [4] to express *parameterized* versions of programs, where the effect of a loop-free and call-free block of statements is modeled as an application of an uninterpreted function. The type of the globals is an uninterpreted type `T`, and there is a single global `g` of this type representing the global state unless otherwise noted.

```
void A(){
  i := 0;
While1:
  if(i < E(n)){
     g := S1(g,i);
     g := S2(g,i);
     i := i + 1;
L1: //FLABEL
     goto While1;
  }
}

void B(){
  i := 0;
  g := S1(g,i);
While2:
  if(i < E(n)-1){
     g := S2(g,i);
     i := i + 1;
L2: //FLABEL
     g := S1(g,i);
     goto While2;
  }
  g := S2(g,i);
  i := i + 1;
}
```

(a)

```
void A'(){
  i := 0;
  if(i < E(n)){
     g := S1(g,i);
     g := S2(g,i);
     i := i + 1;
     r := call L1(i);
  }
}

int L1(int i){
  i' := i;
  if(i' < E(n)){
     g := S1(g,i');
     g := S2(g,i');
     i' := i' + 1;
     r := call L1(i');
     return r;
  }
}
```

(b)

```
void B'(){
  i := 0;
  g := S1(g,i);
  if(i < E(n)-1){
     g := S2(g,i);
     i := i + 1;
     r := call L2(i);
     return ;
  }
  g := S2(g,i);
  i := i + 1;
}

int L2(int i){
  i' := i;
  g := S1(g,i');
  if(i' < E(n)-1){
     g := S2(g,i');
     i' := i' + 1;
     i' := call L2(i');
     return i';
  }
  g := S2(g,i');
  i' := i' + 1;
  return i';
}
```

(c)

Fig. 8. Example of software pipelining. (a) Input programs and (b,c) programs after loop extraction for `A` and `B` respectively.

Software pipelining Figure 8 describes the encoding of software pipelining, where `E`, `S1` and `S2` represent uninterpreted predicates or functions. The optimization can be expressed as a composition of two transformations [4] (a) transformation from `A` to `B`, and (b) replacing the sequence of statements

$$\mathtt{g} := \mathtt{S2}(\mathtt{g},\mathtt{i}); \mathtt{i} := \mathtt{i} + 1; \mathtt{g} := \mathtt{S1}(\mathtt{g},\mathtt{i});$$

in B with

$$\texttt{g} := \texttt{S1}(\texttt{g}, \texttt{i} + \texttt{1}); \texttt{g} := \texttt{S2}(\texttt{g}, \texttt{i}); \texttt{i} := \texttt{i} + \texttt{1}$$

We only describe the proof of the transformation from `A` to `B`. The latter follows under the assumption that the call-free statements `g := S1(g,i+1)` and `g := S2(g,i)` commute. Although it is easy to see that the second transformation cannot affect termination, a rigorous proof of the composed transformation would need the use of relative termination (omitted for brevity).

Apart from the use of `FLABEL` to extract loops into recursive procedures, the interesting part of the proof is in the following mutual summaries used to express the relationships between the two versions:

- $MS(\texttt{A}', \texttt{B}') \doteq (E(n) > 0 \wedge g_1 = g_2) \implies g'_1 = g'_2$
- $MS(\texttt{L1}, \texttt{L2}) \doteq (E(n) > 0 \wedge i_1 = i_2 \wedge g_1 = g_2 \wedge i_2 < E(n)) \implies (g'_1 = g'_2 \wedge r_1 = r_2)$
- $MS(\texttt{L1}, \texttt{L1}) \doteq i_1 \geq E(n) \implies (g'_1 = g_1 \wedge r_1 = i_1)$

The constraint $E(n) > 0$ present in the summaries is the condition under which the transformation is sound. The constraint $i_2 < E(n)$ is a precondition for `L2` that is expressed as an antecedent in the mutual summary for $MS(\texttt{L1}, \texttt{L2})$. Finally, the $MS(\texttt{L1}, \texttt{L1})$ is a postcondition for `L1` that is required to reason about the last iteration of the loop in `L1` — it expresses that when input $i \geq E(n)$, then `L1` does not transform the state and returns the input i. We believe that only the last postcondition is the additional price paid for using mutual summaries instead of traditional simulation relations in earlier works [4].

Loop unrolling Figure 9 describes the example of loop unrolling, where `B` performs two iterations of the loop whenever `i + 1 < E(n)`. The interesting part for the proof is that the body of extracted procedure `L1` has to be inlined *once* inside itself to match it up with `L2`. We omit the resulting mutual summaries that express $(\texttt{A}', \texttt{B}')$ and $(\texttt{L1}, \texttt{L2})$ are equivalent (modulo termination).

```
void A(){
  i := 0;
L1: //FLABEL
  if(i < E(n)){
     g := S1(g,i);
     i := i + 1;
     goto L1;
  }
}
```

```
void B(){
  i := 0;
L2; //FLABEL
  if(i + 1 < E(n)){
     g := S1(g,i); i := i + 1;
     g := S1(g,i); i := i + 1;
     goto L2;
  }
  if(i  < E(n)){
     g := S1(g,i); i := i + 1;
  }
}
```

Fig. 9. Example of loop unrolling. Input programs `A` and `B`.

```
T a, b; //globals
void A(){
  i := 0;
L1: //FLABEL
  if(i < E(n)){
     a := S1(a,i);
     if (F(b)){
       a := S2(a,i);
     }
     i := i + 1;
     goto L1;
  }
}
```

```
void B(){
  i := 0;
  if (F(b)){
L2: //FLABEL
    if(i < E(n)){
       a := S1(a,i); a := S2(a,i); i := i + 1;
       goto L2;
    }
  } else {
L3: //FLABEL
    if(i < E(n)){
       a := S1(a,i); i := i + 1;
       goto L3;
    }
  }
}
```

Fig. 10. Example of loop unswitching.

Loop unswitching Figure 10 describes the example of loop unswitching. Here the loop in `A` (at `L1`) is split into two loops in `B` since the condition `F(b)` does not change in the loop. The mutual summaries for this proof are:

- $MS(\mathtt{A}', \mathtt{B}') \doteq (a_1 = a_2 \wedge b_1 = b_2) \implies a_1' = a_2'$
- $MS(\mathtt{L1}, \mathtt{L2}) \doteq (F(b_1) \wedge i_1 = i_2 \wedge a_1 = a_2 \wedge b_1 = b_2) \implies (a_1' = a_2' \wedge r_1 = r_2)$
- $MS(\mathtt{L1}, \mathtt{L3}) \doteq (\neg F(b_1) \wedge i_1 = i_2 \wedge a_1 = a_2 \wedge b_1 = b_2) \implies (a_1' = a_2' \wedge r_1 = r_2)$

The interesting part of the mutual summaries is the presence of the conditions under which the loop `L1` matches with `L2` or `L3`. This is also one of the instances where the mutual summaries relate one procedure (`L1`) to multiple procedures (`L2` and `L3`).

Other compiler optimizations. In addition to the optimizations shown in this section, we have been able to prove many other examples of loop optimizations handled by previous works [13, 4]. The notable exceptions are transformations such as loop reversal and loop interchange that may change the order of updates to an array. Previous works have used a special PERMUTE rule [13], that tries to permute the order of updates in a loop. We are currently investigating encoding this rule using mutual summaries and relative termination. Nevertheless, our approach already handles many examples of interprocedural transformations that are beyond the ability the PERMUTE rule (§ 4.3).

4.3 Interprocedural transformations

In this section, we show the applications of our approach towards instances of interprocedural transformation.

Compiler optimizations Our approach can be used to prove various compiler optimizations that require global (or interprocedural) analysis. The proof of *tail recursion elimination* can be done easily after the loop is extracted into

```
T a, b; //globals
void A(){
  call B(0);
}
void B(int i){
  if(i < E(n)){
     call B(i+1);
     a := S1(a,i);
     b := S2(b,i);
  }
}
```

```
void C(){
  call D(0); call E(0);
}
void D(int i){
  if(i < E(n)){
     call D(i+1); a := S1(a,i);
  }
}
void E(int i){
  if(i < E(n)){
     call E(i+1); b := S2(b,i);
  }
}
```

Fig. 11. Example of restricted interprocedural loop fission.

a tail-recursive procedure using `FLABEL`. The proof for *inlining* will be similar to the proof of *loop unrolling* discussed in earlier section. Similarly, *global constant propagation* can be encoded using mutual summaries that express that a particular global or return variable has a constant value.

In addition to common compiler optimizations, Figure 11 demonstrates a transformation of a single non-tail recursive procedure into two non-tail recursive procedures (corresponds to a restricted interprocedural version of *loop fission* optimization). This can be handled using mutual summaries (omitted).

```
f1(n) {
  if (n == 0) {
    return 1;
  } else {
    return
     n * f1(n - 1);
  }
}
```

```
f2(n, a) {
  if (n == 0) {
    return a;
  } else {
    return
      f2(n - 1, a * n);
  }
}
```

Fig. 12. Example for tail vs. non-tail recursive factorial.

Figure 12 shows two implementations of factorial, one tail recursive and one not tail recursive. We can prove that these compute the same result ($\texttt{f1}(n) = \texttt{f2}(n, 1)$) using the following mutual summary: $\mathit{MS}(\texttt{f1}, \texttt{f2}) \doteq (n_1 = n_2) \implies (r_1 * a_2 = r_2)$.

Conditional equivalence Bug fixes and feature additions result in two versions of a program that are behaviorally equivalent only under a subset of inputs. We show that mutual summaries can be used for showing *conditional equivalence* even for recursive procedures. Figure 13 contains two versions of a procedure `f` (denoted as `f1` and `f2` respectively) that recursively evaluates an expression rooted at the argument x. The new version differs in functionality when an additional argument `u` is provided that indicates "unsigned" arithmetic instead of the signed arithmetic represented by {+,-}. The following mutual summary $\mathit{MS}(\texttt{f1}, \texttt{f2})$ validates that the two procedures agree when u is off: $(x_1 = x_2 \wedge u = 0) \implies r_1 = r_2$.

Most examples in this Section have different set of inputs for the two versions, and thus not amenable to be abstracted with a common uninterpreted function [3]. Let us briefly comment on the relationship with previous works

```
int f1(int x){
   if (Op[x] = 0)
     return Val[x];
   a := f1(A[x]);
   b := f1(B[x]);
   if (Op[x] = 1)
     return a + b;
   else if (Op[x] = 2)
     return a - b;
   else
     return 0;
}
```

```
int f2(int x, int u){
   if (Op[x] = 0)
     return Val[x];
   a := f2(A[x], u);
   b := f2(B[x], u);
   if (Op[x] = 1){
     if (u) return uAdd(a,b);
     else   return a + b;
   } else if (Op[x] = 2){
     if (u) return uSub(a,b);
     else   return a - b;
   } else   return 0;
}
```

Fig. 13. Example for feature addition and conditional equivalence.

```
void D(ref x){
  d[x] := U(d[x]);
}
void A(ref x){
  if (x != null){
    call A(next[x]);
    call D(x);
  }
}
void B(ref x){
  if (x != null){
    call D(x);
    call B(next[x]);
  }
}
```

(a)

```
void AD(x,y){
  inline call A(x);
  call D(y);
}
void DA(x,y){
  call D(y);
  inline call A(x);
}
void DD(x,y){
  inline call D(x);
  inline call D(y);
}
```

(b)

Mutual summaries (A vs. B)
$MS(\texttt{A},\texttt{B}) : (x_1 = x_2 \wedge d_1 = d_2) \implies d'_1 = d'_2$
$MS(\texttt{A},\texttt{A}) : (x_1 = x_2 \wedge d_1 = d_2) \implies d'_1 = d'_2$
$MS(\texttt{D},\texttt{D}) : (x_1 = x_2 \wedge d_1 = d_2) \implies d'_1 = d'_2$
$MS(\texttt{AD},\texttt{DA}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2) \implies d'_1 = d'_2$
$MS(\texttt{DA},\texttt{AD}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2) \implies d'_1 = d'_2$
$MS(\texttt{DD},\texttt{DD}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2) \implies d'_1 = d'_2$

Relative termination conditions (A vs. B)
$RT(\texttt{AD},\texttt{DA}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2)$
$RT(\texttt{DA},\texttt{AD}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2)$
$RT(\texttt{DD},\texttt{DD}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2)$
(c)

Fig. 14. Example of list traversal transformation. (a) Two different implementations A, and B of list traversal, (b) auxiliary procedures introduced during the proof, (c) *MS* and *RT* used to prove A and B equivalent.

that use identical uninterpreted functions to abstract equivalent procedures [8, 3]. Using an uninterpreted function (instead of mutual summaries) to represent equivalent procedures is an optimization for the purpose of equivalence checking; it avoids introducing the implication $((x_1 = x_2 \wedge g_1 = g_2) \implies r_1 = r_2 \wedge g'_1 = g'_2)$ explicitly in the formula to the theorem prover. However, the use of an uninterpreted function is restricted to modeling deterministic procedures, and only works when compared procedures have identical sets of arguments and globals.

List traversal Finally, we describe an example that requires careful interplay between mutual summary and relative termination specifications and well beyond the realm of present approaches using automated provers. Consider the two versions A and B of a program in Figure 14. Each version traverses elements in a list following the `next` field and updates the `data` field by an uninterpreted function U in the procedure D. The procedure B is a tail-recursive version of A. The transformation can be applied (either manually or by a compiler) to optimize the performance of the implementation. Preservation of semantics includes showing that the two versions diverge on the same inputs; it is easy to see that neither program terminates when the input is a cyclic list.

Although the change from A to B just swaps the order of calls to D and the recursive call, it has a global impact. Figure 15 demonstrates that the order of invoking the procedure D differs when A and B are invoked on the same input $u_0 \doteq \mathtt{x}, u_1 \doteq \mathtt{next}[u_0], \ldots, u_{k+1} \doteq \mathtt{next}[u_k]$. This makes proving the semantic equivalence of such transformations non-trivial. An intuition to understand the transformation from A to B is to think of creating intermediate programs that progressively transform an execution of A to an execution of B. Figure 15 shows the execution of such an intermediate program T that follows B's execution and then follows A's executions.

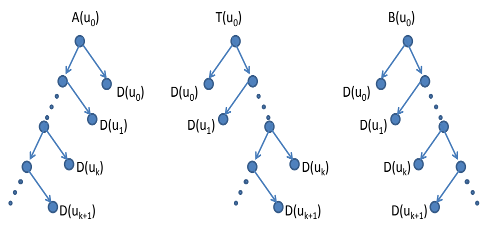

Fig. 15. Executions of different procedures over time.

To handle such transformations, we need to provide a specification that allows *commuting* the calls to A and D. Such a specification can be obtained by introducing *composed procedures* AD, DA (Figure 14(b)) and writing mutual summary specifications on them. The procedure AD invokes A followed by D, and inlines the body (not nested callees) of A. The mutual summaries $MS(\mathtt{AD}, \mathtt{DA})$ and $MS(\mathtt{DA}, \mathtt{AD})$ (Figure 14(c)) express the fact that the summaries of AD and DA are equal on any input — in other words, A and D commute.

To leverage these auxiliary composed procedures in the proof, we have to relate the summary relations of these procedures (e.g. $R_{\mathtt{AD}}$) with that of the underlying procedures ($R_{\mathtt{A}}$ and $R_{\mathtt{D}}$). For a procedure fh composed of f and h, we automatically introduce the following axiom, which says that fh has a terminating execution if and only if f and h have a terminating execution through some intermediate global value g_2:

$$\forall x_1, x_2, g_1, g_1'.\ R_{fh}(x_1, x_2, g_1, g_1') \iff (\exists g_2.\ R_f(x_1, g_1, g_2)\ \wedge\ R_h(x_2, g_2, g_1'))$$

For this axiom to be sound, we require that at least one of f and h be inlined in fh. Although we do not yet have a formal proof of soundness for this axiom, we require the inlining for the axiom to fit within the inductive framework of Section 3.4. Intuitively, before the inductive step adds (x_1, x_2, g_1, g_1') to R_{fh}, the proof considers $R_{fh}(x_1, x_2, g_1, g_1')$ to be false, and thus requires that at least one of $R_f(x_1, g_1, g_2)$ and $R_h(x_2, g_2, g_1')$ be false for the axiom to hold, meaning that there cannot be calls in fh that introduce both the assumptions $R_f(x_1, g_1, g_2)$ and $R_h(x_2, g_2, g_1')$.

We need similar specifications for showing that D commutes with itself. Figure 14(c) contains all the mutual summary specification for this proof. The mutual summaries such as $MS(\texttt{A}, \texttt{A})$ are needed to express that `A` is *deterministic*, a requirement to be able to prove the commute mutual summaries described above. Figure 14(c) also lists the relative termination conditions that were specified for this proof. Given the above mutual summaries and relative termination conditions (all of which express equality of inputs and outputs), we can show that all these specification are true to establish that if `A` and `B` start out with the same inputs and `A` terminates, then so does `B` with equal outputs.

5 Related work

Our work is most closely related to work on proving equivalence in the context of compiler validation. Translation validation [9] is an approach for validating compilers by ensuring that each pair of source and target programs produced by the compiler are semantically equivalent. Necula [8] provided techniques to infer simulation relations by performing a lock-step analysis of the two programs, that generates simulation relations for simple compiler optimizations. Mutual summaries can capture such proofs that are based on establishing simulation relations. Zuck et al. [13] provide a rule PERMUTE that allows proving more complex optimizations that permute order of execution of loops (e.g. in loop reversal optimization). Tate et al. [11] provide an approach called *equality saturation* where an equality saturated program expression graph (PEG) can be used to capture equivalent programs. Tristan et al [12] instead provide rules for normalizing PEGs to perform translation validation. These approaches are automated and have been applied on various production compilers. Various domain specific languages (Cobalt [6], PEC [4]) have been devised to express compiler transformations as rewrite rules in a language. However, these approaches cannot validate interprocedural transformations (§ 4.3). Finally, the CompCert project [7] uses interactive theorem provers to provide an end-to-end correctness guarantee of semantic preservation by a compiler; this results in greater flexibility but less automation than approaches based on automated theorem provers. Pnueli and Zaks [10] generalize simulation-relation based translation validation to check simple interprocedural optimizations such as tail-recursion elimination, global constant propagation and inlining. However, program transformations such as translating a non tail-recursive procedure to its tail-recursive counterparts (Figure 14, Figure 12) will not be possible in this approach. Godlin and Strichman [3] describe automated methods for checking equivalence and mutual termination (under equal inputs) of mutually recursive procedures using uninterpreted functions as summaries. Our approach is not limited to proving equivalence but can be used to compare arbitrary mutual summaries. Mutual summaries provide more extensibility (at the cost of automation) by relating the summaries of two procedures with an arbitrary relation. This allows us to not only prove intraprocedural optimizations (that are not possible in [3]), but also new examples of interprocedural transformations (§4.3), including those that cannot be proved earlier (§6 in [3]). Relative termination allows reasoning about termination under specific conditions and generalizes the earlier work of checking mutual termination [3].

6 Conclusion

In this paper, we provided a general framework for comparing programs using program verifiers and automated theorem provers. We are currently working on extending the framework to handle more complex program transformations (e.g. the PERMUTE rule [13]), and automating the generation of mutual summary and the relative termination specifications. For most of the simple equality specifications used in this paper, we expect to leverage existing invariant synthesis techniques (e.g. predicate abstraction) to infer a majority of these specifications.

References

1. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87, 2005.
2. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, pages 337–340, 2008.
3. B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Inf.*, 45(6):403–439, 2008.
4. S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Programming Language Design and Implementation (PLDI '09)*, pages 327–337. ACM, 2009.
5. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification (CAV '12)*, LNCS 7358, pages 712–717, 2012.
6. S. Lerner, T. D. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Programming Language Design and Implementation (PLDI '03)*, pages 220–231, 2003.
7. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL '06)*, pages 42–54, 2006.
8. G. C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation (PLDI '00)*, pages 83–94, 2000.
9. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*, pages 151–166, 1998.
10. A. Pnueli and A. Zaks. Validation of interprocedural optimizations. In *Compiler Optimization Meets Compiler Verification (COCV '08)*, 2008.
11. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *Principles of Programming Languages (POPL '09)*, pages 264–276, 2009.
12. J. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for llvm. In *Programming Language Design and Implementation (PLDI '11)*, pages 295–305, 2011.
13. L. D. Zuck, A. Pnueli, B. Goldberg, C. W. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Formal Methods in System Design*, 27(3):335–360, 2005.