Automatic Equivalence Checking of UF+IA Programs

Nuno P. Lopes and José Monteiro

INESC-ID / IST - TU Lisbon

Abstract. Proving the equivalence of programs has several important applications, including algorithm recognition, regression checking, compiler optimization verification, and information flow checking.
Despite being a topic with so many important applications, program equivalence checking has seen little advances over the past decades due to its inherent (high) complexity.
In this paper, we propose, to the best of our knowledge, the first algorithm for the automatic verification of partial equivalence of two programs over the combined theory of uninterpreted function symbols and integer arithmetic (UF+IA). The proposed algorithm supports, in particular, programs with nested loops.
The crux of the technique is a transformation of uninterpreted functions (UFs) applications into integer polynomials, which enables the summarization of loops with UF applications using recurrences. The equivalence checking algorithm then proceeds on loop-free, integer only programs.
We implemented the proposed technique in CORK, a tool that automatically verifies the correctness of compiler optimizations, and we show that it can prove more optimizations correct than state-of-the-art techniques.

1 Introduction

Proving the equivalence of programs has several important applications, including, but not limited to, algorithm recognition [2], regression checking [11, 13, 24], compiler optimization verification [18, 23] and validation [30, 32, 40, 43, 46, 47], and information flow proofs [5, 42].

The objective of algorithm recognition is to identify known algorithms (such as a sorting algorithm, or even a specific algorithm like quicksort) out of large and complex programs. This can be useful, for example, to improve code comprehension and for automatic documentation generation. Algorithm recognition can be accomplished by searching for an equivalent algorithm in a database.

Regression verification aims at tracking the functional differences in a program in each code change. The idea is that a tool that performs regression verification can pinpoint the parts of the program where the semantics were changed since the previous code revision, so that the developer can manually confirm if those were the intended changes. Additionally, these tools can help the developer confirm if some code refactoring or manual optimization preserved the semantics or not.

Compiler optimization verification consists in verifying that a given optimization is semantic preserving for all allowed code inputs, i.e., that the original and optimized code templates are equivalent. Optimization validation verifies that an optimization ran correctly by checking the original and optimized pieces of code for equivalence (after the optimization was run).

In the domain of information flow, proofs for the non-existence of information leaks can be accomplished by establishing the equivalence of the program with itself (self-composition). Since the programs have some non-determinism associated (the private information), a program will not be equivalent to itself if some of the non-determinism may be observable (meaning that it may leak secure information).

Uninterpreted function symbols (UFs) are frequently used in software verification tasks, including in the applications mentioned above. UFs are quite appealing because they allow certain details of the programs to be abstracted out by replacing with UFs the parts whose specifics are irrelevant to the proof being done.

Despite being an important area with several applications, state-of-the-art software verification tools, such as ARMC [33], Blast [20,21], CPAchecker [9], FSoft [22], HSF [15], Impact [27], and Slam [3], are unable to prove equivalence of most programs containing loops. These tools are usually not able to automatically derive sufficiently strong loop invariants to complete equivalence proofs of looping programs, even if just considering the theory of integer arithmetic, let alone the combined theory of uninterpreted function symbols and integer arithmetic (UF+IA).

In this paper, we present, to the best of our knowledge, the first algorithm to automatically prove the equivalence of programs consisting of integer arithmetic operations and applications of UFs. The proposed algorithm is applicable, in particular, to programs containing zero or more (nested) loops.

Applications of UFs are first rewritten to integer arithmetic expressions (polynomials over the inputs of the applications), and then our equivalence checking algorithm works on purely integer manipulating programs. Loops are summarized as recurrences, for which we compute the closed-form solution. The provably correct conversion of UF applications to integer expressions makes possible the representation of loops with UF applications using recurrences.

We have implemented the proposed algorithm in CORK, a tool that verifies the correctness of compiler optimizations, and we show that CORK can prove more optimizations correct than state-of-the-art techniques.

The rest of the paper is organized as follows. Section 2 gives an intuition of how our algorithm proves the equivalence of programs with a simple example. Section 3 presents the program model that we consider and gives preliminary definitions. Section 4 describes our algorithm for automatic partial equivalence checking of programs over the UF+IA theory. Section 5 presents CORK, a tool that verifies the correctness of compiler optimizations automatically, and provides an evaluation on how CORK compares with PEC [23], a state-of-the-art tool for compiler optimization verification. Section 6 presents the related work.

```
i := 0
while i < N do
    k := UF(k, i)
    i := i + 1
```

```
i := N
while i ≥ 1 do
    k := UF(k, N − i)
    i := i − 1

if N ≤ 0 then
    i := 0
else
    i := N
```

Fig. 1. Example of two equivalent programs.

2 Illustrative Example

We illustrate our algorithm for program equivalence checking on a simple example. Figure 1 shows two equivalent example programs. Our objective is to prove that these two programs are indeed equivalent.

The first step of the algorithm is to replace the applications of uninterpreted functions (UFs) with expressions over integers. In the left program, we replace the UF application with the following expression (a polynomial of degree one):

$$a \times k + b \times i + c$$

where a, b, and c are free variables not occurring in the input programs, and are associated with this specific UF symbol. Other UF symbols occurring in the program would have different sets of free variables associated with each input parameter. Similarly, for the UF application of the right program we obtain:

$$a \times k + b \times (N - i) + c$$

These expressions (polynomials) have a unique value for each set of UF symbol and input parameters, which is not reproducible through any other sequence of operations. This is because free variables are universally quantified, and therefore there always exists an assignment to the variables a, b, and c that leads to different results for different UF applications.

As we shall see later, the degree of the polynomials that replace UF applications is not always one. We give a lower bound for this degree in Section 4.2.

The second step that the algorithm performs is removing the loops. This is accomplished by replacing each loop with a set of assignments to the variables modified in the loop. The expressions assigned to each variable are expressed over the closed-form solution of a system of recurrences that summarizes the loop.

For the left program, we obtain the following system of recurrences:

$$\begin{aligned} R_i(n) &= R_i(n-1) + 1 \\ R_i(0) &= 0 \\ R_k(n) &= a \times R_k(n-1) + b \times R_i(n-1) + c \\ R_k(0) &= k_0 \end{aligned}$$

```
i := 0
if i < N then
    assume R_i(n-1) < N ∧ R_i(n) ≥ N
    k := R_k(n)
    i := R_i(n)
```

```
i := N
if i ≥ 1 then
    assume V_i(n-1) ≥ 1 ∧ V_i(n) < 1
    k := V_k(n)
    i := V_i(n)

if N ≤ 0 then
    i := 0
else
    i := N
```

Fig. 2. Programs of Figure 1 with loops and UF applications removed.

where n represents the loop iteration number, and k_0 is the (arbitrary) value of k when the program starts (required since k is not initialized before its first usage). A recurrence for N is not needed, since it is not modified in the loop.

The recurrence $R_x(y)$ represents the value of variable x at iteration number y. For example, the recurrence $R_i(n)$ defined previously means that the value of i in any given iteration is equal to the value of i in the previous iteration plus one. Moreover, before the loop starts, i has the value zero.

Similarly, for the right program we obtain the following system of recurrences:

$$\begin{aligned} V_i(n) &= V_i(n-1) - 1 \\ V_i(0) &= N \\ V_k(n) &= a \times V_k(n-1) + b \times (N - V_i(n-1)) + c \\ V_k(0) &= k_0 \end{aligned}$$

Figure 2 shows the programs of Figure 1 after both transformations (elimination of loops and UF applications) have been applied.

The **assume** command ensures that its input boolean expression is satisfiable, or the program execution is blocked otherwise. We use this command to implicitly compute the trip count of loops.

Intuitively, if m is the number of iterations performed by a loop, in the iterations numbered $0 \ldots (m-1)$ the loop guard is true, and it is false in the following iteration (m). Therefore, m is the first iteration when the loop guard becomes false.

After the **assume** command in the example is evaluated, the value of n is the number of times that the corresponding loop would have been executed and therefore $R_x(n)$ represents the value of the variable x after the loop terminates.

We can now compute the closed-form solution of the previously given systems of recurrences. For the left program we obtain the following solution (computed by Wolfram Mathematica 8):

$$\begin{aligned} R_i(n) &= n \\ R_k(n) &= \frac{b\,(a^n - an + n - 1) + (a-1)\,(a^n((a-1)k_0 + c) - c)}{(a-1)^2} \end{aligned}$$

assume $i = \bar{i} \wedge k = \bar{k} \wedge N = \bar{N}$

$i := 0$
if $i < N$ **then**
  assume $R_i(n-1) < N \wedge R_i(n) \geq N$
  $k := R_k(n)$
  $i := R_i(n)$

$\bar{i} := \bar{N}$
if $\bar{i} \geq 1$ **then**
  assume $V_i(\bar{n}-1) \geq 1 \wedge V_i(\bar{n}) < 1$
  $\bar{k} := V_k(\bar{n})$
  $\bar{i} := V_i(\bar{n})$

if $\bar{N} \leq 0$ **then**
  $\bar{i} := 0$
else
  $\bar{i} := \bar{N}$

assert $i = \bar{i} \wedge k = \bar{k} \wedge N = \bar{N}$

Fig. 3. Sequential composition of the programs of Figure 2. The right program is renamed, so that each variable v becomes $\bar{v}$.

For the right program, the solution for $V_k(n)$ is equal to $R_k(n)$ of the left program, and for V_i is:

$$V_i(n) = N - n$$

The final step of the algorithm is to prove equivalent the transformed programs (that are now only over integer arithmetic and loop-free). To accomplish this, we first do the sequential composition of the two programs, where the second is renamed to operate over a distinct set of variables from the first program. We then add an assertion at the end of the composed program to verify that the value of the corresponding variables of the two programs are equal when the programs terminate.

The sequential composition of the programs of Figure 2 is shown in Figure 3. The references to recurrences were not replaced by their closed-form solutions to avoid cluttering the example.

If we prove that the composed program is safe, i.e., that the condition of the **assert** command is true for all inputs, then we have proved that the two input programs are equivalent.

To prove program safety, and since the number of symbolic paths of the composed programs is always finite (as we remove the loops), we can use a simple algorithm that enumerates all paths and checks if the assertion is violated in any of them.

$$
\begin{aligned}
e &::= n \mid\ v \mid\ e_1 \oplus e_2 \mid\ \mathsf{UF}(e_1, \ldots, e_n) \\
b &::= e \leq 0 \mid b_1 \otimes b_2 \\
c &::= \textbf{skip} \mid\ v\ :=\ e \mid\ c_1\ ;\ c_2 \mid\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid\ \textbf{while}\ b\ \textbf{do}\ c_1 \mid\ \textbf{assume}\ b \\
&\quad \mid \textbf{assert}\ b
\end{aligned}
$$

Fig. 4. WHILE language syntax. n is an integer number, v is a variable name, UF is an uninterpreted function symbol, $\oplus$ is a binary operator over integer expressions (e.g., $+$, $-$), and $\otimes$ is a binary operator over boolean expressions (e.g., $\wedge$, $\vee$).

3 Program Model

We assume that programs are specified in the WHILE language, whose syntax is given in Figure 4, and with customary semantics. The expressions are over the combined theory of uninterpreted function symbols and integer arithmetic (UF+IA). The evaluation of expressions is parameterized on an interpretation for each UF symbol.

For the sake of ease of reading, in the examples given throughout the paper, we relax the syntax of expressions (e.g., to accept more operators than $\leq$), but those examples can be trivially converted to the WHILE language we present.

Let σ be a program state, which is a valuation of the program variables. Let $\sigma(v)$ be the value of the variable v in the program state σ. This notation is extended for expressions, such that $\sigma(e)$ is the expression e with each variable replaced with its value in state σ. Let $\sigma[v \mapsto n]$ be a program state that is identical to state σ, except for the value of variable v, which is n. Let σ_0 be the initial state of an execution of a program. We have that $\sigma_0(v) = v_0$ for each variable v used in the program, with variable v_0 being fresh.

A configuration $\langle c,\ \sigma\rangle$ is a pair where c is a command and σ is a state. Let $\langle c,\ \sigma\rangle \rightarrow \langle c',\ \sigma'\rangle$ be the reduction of the configuration $\langle c,\ \sigma\rangle$ to the configuration $\langle c',\ \sigma'\rangle$ in one step. Let $\langle c,\ \sigma\rangle \rightarrow \sigma'$ be the reduction in one step of the configuration $\langle c,\ \sigma\rangle$ to the state σ' when there are no further commands left to execute. Finally, let $\langle c,\ \sigma\rangle \rightarrow^* \sigma'$ be the reduction in one or more steps of the configuration $\langle c,\ \sigma\rangle$ to the state σ'.

Let $\mathsf{Vars}(P)$ be the set of variables of program P (a command). A variable v is fresh in program P if $v \notin \mathsf{Vars}(P)$. Let $\mathsf{Out}(P) \subseteq \mathsf{Vars}(P)$ be the set of output observable variables of a program P. Let $\sigma \downarrow V$ be the projection of the state σ over the set of variables V and let $\sigma \downarrow \mathsf{Out}(P)$ be the observable state of σ of program P.

Two programs are considered partially equivalent iff starting in the same arbitrary state, they terminate in the same observable state for all possible UF interpretations, i.e., P_1 and P_2 are partially equivalent iff the following holds:

$$\langle P_1,\ \sigma_0\rangle \rightarrow^* \sigma_1 \wedge \langle P_2,\ \sigma_0\rangle \rightarrow^* \sigma_2 \implies \sigma_1 \downarrow \mathsf{Out}(P_1) = \sigma_2 \downarrow \mathsf{Out}(P_2)$$

with $\mathsf{Out}(P_1) = \mathsf{Out}(P_2)$.

4 Program Equivalence Checking

In this section, we present the new algorithm to check if two programs over the UF+IA theory are partially equivalent.

4.1 Restrictions

We impose the following restrictions on the programs that our equivalence checking algorithm can handle:

1. UFs must have exactly one output parameter.
2. There can be no branching (i.e., **if** statements) inside loops. Nested loops, however, are allowed.
3. The trip count of inner loops may not depend on the outer loops, i.e., the number of times that inner loops iterate is constant relative to outer loops.
4. Loop conditions cannot involve UF applications.

Restriction 1 can be lifted by splitting UFs with more than one output into newly created UFs (one per output). Restriction 2 can be relaxed by allowing branching conditions that always evaluate to the same value in all loop iterations. In that case, the program can be rewritten to move the branches out of the loop.

4.2 Algorithm

The algorithm runs in three steps:

1. Eliminate UF applications.
2. Replace loops with recurrences.
3. Check resulting programs for equivalence.

Applications of UFs are abstracted using polynomials, in order to obtain programs with integer operations only. This allows us to compute the closed-form of loops using recurrences.

Although our algorithm is sound and complete (under the stated restrictions), computing the closed-form solution of recurrences is undecidable, and therefore the overall method is incomplete.

In the following sections, we describe each step of the algorithm separately.

Eliminate UF applications The first step of the algorithm is to eliminate UF applications. This is accomplished by replacing each UF application with a polynomial over its inputs, as defined by the transformation T:

$$T(e) = \sum_{i=1}^{n} \sum_{j=0}^{\mathsf{u}(\mathsf{UF},i)} \mathtt{UF}_{i,j} \times (\mathsf{T}(e_i))^j, \quad \text{if } e = \mathsf{UF}(e_1, \ldots, e_n)$$

The other trivial (do nothing) cases are omitted for brevity.

The function $\mathsf{u}(f, i)$ used by transformation T defines the degree of the polynomial that replaces an UF application. The value of $\mathsf{u}(f, i)$ is the maximum number of times that the given UF f is applied with distinct values in the ith parameter in each and every program path minus one. Only function applications whose value is possibly used in a boolean expression need to be considered.

Intuitively, two programs with UF applications are equivalent iff, for each possible input and for each observable output, the number of times the UFs are applied is equal in both programs, and the values of the input parameters of each application are equal as well.

Transformation T captures this information precisely by replacing each UF application with a polynomial over the inputs of the application. Each UF symbol is assigned a set of fresh variables $\mathtt{UF}_{i,j}$ that is used only by applications of that symbol. Therefore, the value of an UF application cannot be reproduced by any sequence of commands that does not include exactly the same UF application.

For example, the following boolean expression

$$f(x) = 0 \wedge f(y) = 1 \wedge f(z) = 2 \wedge g(x) \leq 0 \wedge y < z \wedge z < x$$

is translated to (assuming no more applications of f nor g in the rest of the program):

$$f_{1,2} \times x^2 + f_{1,1} \times x + f_{1,0} = 0 \wedge f_{1,2} \times y^2 + f_{1,1} \times y + f_{1,0} = 1 \wedge$$
$$f_{1,2} \times z^2 + f_{1,1} \times z + f_{1,0} = 2 \wedge g_{1,0} \leq 0 \wedge y < z \wedge z < x$$

where $f_{1,2}$, $f_{1,1}$, $f_{1,0}$, and $g_{1,0}$ are fresh variables. These variables are never written by the program, and are only read by transformed expressions that originally contained the same UF symbols (f and/or g).

The applications of the uninterpreted function f were transformed into polynomials of degree two, since we have three applications of f with (possibly) different input parameters.

A polynomial with a lower degree would not be sufficient to represent this boolean expression without imposing constraints on the input parameters that did not exist in the original expression with UFs. For example, if we use a polynomial of degree one for the applications of f, we obtain (excluding the constraint with g):

$$f_{1,1} \times x + f_{1,0} = 0 \wedge f_{1,1} \times y + f_{1,0} = 1 \wedge f_{1,1} \times z + f_{1,0} = 2 \wedge y < z \wedge z < x$$

This formula is not satisfiable, while its original UF form is. A polynomial of degree two (as shown above) or of higher degree, however, is guaranteed to yield a satisfiable formula for all distinct x, y, and z (by the Unisolvence Theorem [41]).

Computing the value of $\mathsf{u}(f, i)$ as defined is hard, and may require prior static analysis. This value can, however, be safely over-approximated by the number of applications of f in the whole program, at the expense of generating more complex expressions.

For example, the optimal values for u in the following program excerpt are $\mathsf{u}(f, 1) = 1$ and $\mathsf{u}(f, 2) = 0$ (assuming no other UF applications in the rest of

```
while i < n do
    k := 2 × k
    j := 0
    while j < m do
        k := k + j
        j := j + 1
    i := i + 1
```

$$
\begin{aligned}
R_j(x) &= R_j(x-1) + 1 \\
R_j(0) &= 0 \\
R_k(x) &= R_k(x-1) + R_j(x-1) \\
R_k(0) &= 2 \cdot V_k(y-1) \\
V_i(y) &= V_i(y-1) + 1 \\
V_i(0) &= i_0 \\
V_k(y) &= R_k(x) \\
V_k(0) &= k_0
\end{aligned}
$$

Fig. 5. An example program and the corresponding system of recurrences that summarizes the two loops, where R_j and R_k represent the behavior of the inner loop on the variables j and k, respectively, and V_i and V_k represent the outer loop.

the program). Although there are three applications of f with a different first parameter, only two applications are ever encountered and used in a boolean expression in a single path.

```
if ... then
    j := f(y, 3)
else
    k := f(z, 3)

if f(x, 3) ≤ 0 ∧ j ≤ 0 ∧ k ≤ 0 then
    ...
```

The value of $\mathsf{u}(f, i)$ must be computed over the composed program (and not over each of the two input programs independently), including the **assert** command that is added at the end of it (Section 4.2).

Replace loops with recurrences The second step of the algorithm is to eliminate loops, by replacing each loop with a system of recurrences. The transformation is carried out as follows. Each variable that is assigned in the loop gets a recurrence over a newly introduced variable that represents the loop trip count. For nested loops, the initial value of a recurrence in an inner loop is the value of the previous iteration of the outer loop.

An example program and its system of recurrences is shown in Figure 5. The recurrence $R_v(n)$ represents the value of the variable v at the inner loop iteration n, and $V_v(n)$ in the outer loop. For example, the value of variable k in the iteration x of the inner loop, $R_k(x)$, is equal to the sum of the values of variables k and j of the previous (inner loop) iteration. The value of k in the beginning of the first inner loop iteration, $R_k(0)$, is equal to twice the value of k in the previous outer loop iteration.

The closed-form solution for the system of recurrences is the following:

$$R_j(x) = x \qquad R_k(x) = \frac{4 \cdot V_k(y-1) + x^2 - x}{2}$$

$$V_i(y) = i_0 + y \qquad V_k(y) = \frac{k_0 \cdot 2^{y+1} + (x-1) \cdot x \cdot (2^y - 1)}{2}$$

We note that while the solution of $R_k(x)$ still includes a reference to a recurrence — $V_k(y-1)$ — it is only used to compute the solution of $V_k(y)$ and it is never used directly by the next steps of the algorithm. We only need the value of k after the outer loop terminates, which is represented by $V_k(y)$.

After computing the closed-form solution for the system of recurrences, each loop of the form "**while** b **do** c" is replaced with the following code:

if b **then**
  assume $\sigma_{n-1}(b) \wedge \sigma_n(\neg b)$
  $v_i := \sigma_n(v_i)$
else
  assume $n = 0$

The fresh variable n represents the number of iterations performed by the loop. σ_n is a state where each variable maps to the closed-form solution of its corresponding recurrence at point n, or to itself if the variable is not modified in the loop body c. Variable v_i ranges over all variables that are possibly modified in the loop body. For the previous example, we have for the inner loop, e.g., $\sigma_x(j) = R_j(x) = x$ and $\sigma_x(n) = n$.

Intuitively, a loop executes n times if the loop guard is true for the first n iterations (iterations $0 \ldots (n-1)$) and false in the following iteration (iteration n). The number of iterations is implicitly computed when the **assume** command is evaluated. Its expression states that the loop guard of iteration $n-1$ should be true, and that at iteration n the guard should be false instead.

We note that there can be multiple solutions for the expression given to the **assume** command if the loop guard is non-linear. In this case, the number of loop iterations is the smallest positive n that makes the formula satisfiable. Computing the smallest n can be achieved, for example, by using an optimizing solver or by doing multiple calls to an SMT solver.

For the example in Figure 5, the program after removing the loops is shown in Figure 6. The command "**assume** $y = 0$" at the end can be removed as an optimization, since there are no further uses of y afterward.

Equivalence Checking The third and final step of the algorithm is to prove the equivalence between the two programs after they undergo the transformations previously described.

We do this by sequentially composing the first program with a renamed version of the second. The second program is renamed so that it operates over a different set of variables from the first.

```
if i < n then
   assume V_i(y − 1) < n ∧ V_i(y) ≥ n
   j := 0
   if j < m then
      assume R_j(x − 1) < m ∧ R_j(x) ≥ m
      j := R_j(x)
   else
      assume x = 0
   k := V_k(y)
   i := V_i(y)
else
   assume y = 0
```

Fig. 6. Program of Figure 5 after replacing the loops with a set of assignments over the system of recurrences including $V_i(n)$, $V_k(n)$, and $R_j(n)$.

Let P_1' and P_2' be, respectively, the programs P_1 and P_2 after removing the UF applications and the loops. The composed program is as follows.

assume $\forall v \in \mathsf{Vars}(P_1') \cap \mathsf{Vars}(P_2') : v = \bar{v}$
P_1'
$\bar{P_2'}$
assert $\forall v \in \mathsf{Out}(P_1') : v = \bar{v}$

Program $\bar{P_2}'$ is the same as the program P_2', but where each variable v was renamed to $\bar{v}$. Moreover, $\mathsf{Out}(P_1') = \mathsf{Out}(P_2')$.

If the composed program is safe, i.e., if the condition of the **assert** command is true for all inputs, then the two original programs are partially equivalent.

To prove program safety, and since the number of symbolic paths is finite, we can use an algorithm that enumerates all paths and tests if any of those makes the condition of the **assert** command false.

Note that the value of $\mathsf{u}(f, i)$ defined in Section 4.2 for the composed program above must take into account the paths that pass through programs P_1 *and* P_2, as well as the **assert** command (which takes a boolean expression by itself).

5 Verification of Compiler Optimizations

To evaluate the proposed algorithm, we implemented a prototype to prove the correctness of compiler optimizations. This is an important topic, since all mainstream compilers were shown recently to have several bugs in the optimization passes [45]. Moreover, if the compiler is not proved correct, properties verified on the source-code level of a program are not carried to the binary code, since the compiler may introduce bugs during the translation process.

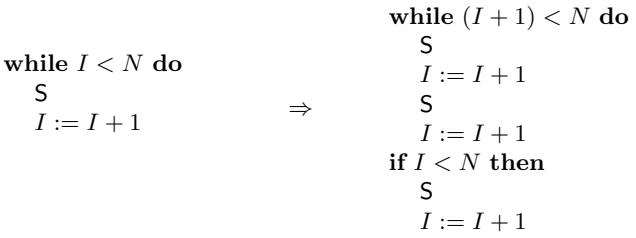

Fig. 7. Loop unrolling: the source template is on the left, and the transformed template on the right. Template statement S cannot modify template variables I and N.

5.1 From Compiler Optimizations to Program Equivalence

We specify a compiler optimization as a transformation function from a *source* template program to a *target* template program. These template programs can be modeled as UF+IA programs, where UFs represent arbitrary statements, or expressions that should be matched within a program under optimization.

We show an example optimization (loop unrolling) in Figure 7. This optimization transforms a loop into a new loop that performs only half of the iterations of the original loop, but where each iteration of the new loop performs twice the work of an iteration of the original loop.

The template statement S is a placeholder for an arbitrary statement (e.g., variable assignments, function calls, or other loops) that may be present in a loop under optimization. Template variables I and N are placeholders for arbitrary program variables. The transformation function states how each template statement/expression is transformed (e.g., moved, duplicated, eliminated) to produce the optimized program.

As an example, we apply loop unrolling to the following program.

while $i < n$ **do**
  $x := i + 2$
  $i := i + 1$

Running the optimization with S instantiated to "$x := i + 2$", I to "i", and N to "n" yields the following program:

while $i < n$ **do**
  $x := i + 2$
  $i := i + 1$
  $x := i + 2$
  $i := i + 1$
if $i < n$ **then**
  $x := i + 2$
  $i := i + 1$

To verify a compiler optimization correct, we split the transformation function into two programs (the source and target templates), and then we convert the template programs into UF+IA programs. Finally, we use the proposed equivalence checking algorithm to prove that the source and target templates are equivalent, which implies that the optimization is correct.

Preconditions of optimizations are specified as read and write sets of the template statements/expressions, which contain the variables that the template statements/expressions *may* read and write, respectively. For example, the read set of S in loop unrolling is $R(\mathsf{S}) = \{c_1, I, N\}$, and the write set is $W(\mathsf{S}) = \{c_1\}$, since the precondition is that S cannot modify variables I and N.

The conversion of a template program to an UF+IA program is done by replacing each template statement S with a set of assignments of the following form:

$$v_i \ := \ S_i(r_1, \ldots, r_n)$$

where $v_i \in W(\mathsf{S})$ and $R(\mathsf{S}) = \{r_1, \ldots, r_n\}$. The transformation of template expressions is done similarly.

In the loop unrolling example, S is replaced with a single assignment (with S_1 being a fresh UF symbol):

$$c_1 \ := \ S_1(c_1, I, N)$$

Variable c_1 is what we call a context variable. These fresh variables c_i represent the variables that are possibly in scope where a template may be instantiated (possibly none) and that do not appear in the template function.

In our example, c_1 represents the effects of S in x. While variable x does not appear explicitly in the transformation function, S does indeed modify x in the example instantiation.

The values computed for the function u are the following: $\mathsf{u}(S_1, 1) = 1$ and $\mathsf{u}(S_1, 2) = 1$, since there are two applications of S_1 with possibly different values that are used in a boolean expression (the **assert** command); and $\mathsf{u}(S_1, 3) = 0$, since N is constant.

At least one context variable is added to each program. Moreover, the read and write sets of each template statement must include at least one context variable, unless the precondition of the optimization states that, e.g., a given statement does not read any other variable than x. Similarly, template expressions may read a variable that is not present in the transformation function (again, unless stated otherwise in the precondition), and therefore their read set must include a context variable.

We may add more than one context variable to a program to express certain preconditions over template statements. For example, if a statement S is idempotent, we have that $R(\mathsf{S}) \cap W(\mathsf{S}) = \emptyset$. Therefore, we have to have at least two distinct context variables c_1 and c_2 to have, e.g., $R(\mathsf{S}) = \{c_1\}$ and $W(\mathsf{S}) = \{c_2\}$ to state that S cannot read a variable that it writes to, nor vice versa.

Similarly, to state that template statements S and T commute, we have $W(\mathsf{S}) \cap R(\mathsf{T}) = W(\mathsf{T}) \cap R(\mathsf{S}) = W(\mathsf{S}) \cap W(\mathsf{T}) = \emptyset$. In this case, we also need at least two distinct context variables.

5.2 Evaluation

We implemented a prototype named CORK[1], which stands for Compiler Optimization coRrectness checKer. CORK is implemented in OCaml ($\sim$ 1,100 LoC), and uses Wolfram Mathematica 8.0.4 for both constraint and recurrence solving.

CORK takes as input a transformation function in the format of the example in Figure 7. CORK then derives two programs over the UF+IA theory as described in the previous section, and subsequently checks if they are equivalent. The equivalence check is done by enumerating each path of the composed program, since the number of paths is finite and small. If the equivalence check fails, CORK prints a counterexample path.

CORK performs three optimizations to improve the performance. First, CORK discharges by itself equality tests of syntactically equal expressions. Second, CORK performs equality propagation on the satisfiability queries sent to Mathematica. Finally, CORK checks the equality of program variables (arising from the **assert** command at the end of the composed program) one-by-one, instead of just one satisfiability query per path. CORK then uses the established equalities in the following queries. Moreover, variable equality checks are ordered so that first are checked the induction variables, and the remaining variables are ordered by the length of their value expressions. Establishing first the equality of expressions involving induction variables improves the performance significantly.

We ran CORK over a set of optimizations (mostly loop-manipulating). The experiments were run on a machine running Linux 3.6.2 with an Intel Core 2 Duo 3.00 GHz CPU, and 4 GB of RAM. The results are shown in Table 1.

We first note that the number of recurrence solving queries is higher than expected (more than one per loop), since we compute the recurrences per path and we do not cache any information across paths. Optimizations that do not manipulate loops explicitly do not generate any recurrence.

We compare the results of CORK with the state-of-the-art tool PEC [23]. Since PEC is not publicly available, we compare only with the published results.

The table is divided in four sets of optimizations (described in, e.g., [1]). The first part is a set of optimizations that do not manipulate loops explicitly, which are trivially proven correct by both CORK and PEC. The second part is a set of optimizations that PEC can prove correct without the help of heuristics. The third part is a set of optimizations that PEC can only prove correct by using the permute heuristic [14, 47], since otherwise it could not find a bisimulation relation automatically. The fourth and last part of the table contains a set of optimizations that PEC cannot prove correct, since it cannot find a bisimulation automatically, even with the permute heuristic. CORK, on the other hand, is able to prove correct the loop strength reduction and loop tiling optimizations. CORK fails to prove correct the loop flattening optimization, since Mathematica is unable to compute the closed-form solution of recurrences with integer division.

The execution time of PEC and CORK is within the same order of magnitude, but CORK advances the state-of-the-art by being able to prove correct more optimizations than PEC.

[1] Prototype and benchmarks available from `http://web.ist.utl.pt/nuno.lopes/cork/`.

Optimization	PEC	# Sat. queries	# Recurrences	Time
Code hoisting	✓	2	0	0.32s
Constant propagation	✓	0	0	0.33s
Copy propagation	✓	0	0	0.33s
If-conversion	✓	2	0	0.34s
Partial redundancy elimin.	✓	2	0	0.34s
Loop invariant code motion	✓	7	5	3.48s
Loop peeling	✓	9	5	3.26s
Loop unrolling	✓	13	8	12.17s
Loop unswitching	✓	14	14	8.19s
Software pipelining	✓	9	5	8.02s
Loop fission	$✓_p$	10	12	23.45s
Loop fusion	$✓_p$	10	12	23.34s
Loop interchange	$✓_p$	15	24	29.30s
Loop reversal	$✓_p$	7	5	8.41s
Loop skewing	$✓_p$	16	24	8.50s
Loop flattening	×	—	—	FAIL
Loop strength reduction	×	6	4	5.63s
Loop tiling	×	7	9	10.94s

Table 1. List of compiler optimizations [1], how PEC performs ($✓_p$ means PEC needs the permute heuristic), the number of satisfiability and recurrence solving queries issued to Mathematica, and the time that CORK took to prove each optimization correct.

6 Related Work

Proving the equivalence of programs is undecidable. However, there has been advances over the last decades to solve the problem under certain assumptions.

Several alternative approaches exist to prove the equivalence of programs, namely manual or semi-automated (with the help of an iterative theorem prover) approaches, bisimulation relation synthesis, symbolic execution, recurrence equivalence, and software model checking based techniques.

Manual and Semi-Automated Proofs Relational Hoare logic [7] is a proof system that enables the verification of equivalence between two programs. The system only supports the verification of structurally equivalent programs (yet, for example, many compiler optimizations do not obey this constraint). Barthe et al. [4] lift some of the restrictions of this work through the usage of product programs. The set of structural differences that the programs under equivalence checking may exhibit is still dependent on the set of built-in proof rules. Liang et al. [25] adapted relational Hoare logic to the setting of concurrent programs.

Bisimulation Parameterized equivalence checking (PEC [23]) is a technique to verify the correctness of compiler optimizations automatically. It works by automatically finding a bisimulation relation [37] between the original and the optimized template programs. For structurally different loops, PEC relies on a set of heuristics inspired in [14, 47].

Recurrence Equivalence Barthou et al. [6] and Shashidhar et al. [39] present different algorithms to prove the equivalence of systems of affine recurrence equations that are structurally similar. Verdoolaege et al. [44] propose an algorithm to prove the equivalence of integer affine programs where loops are described as recurrences. The algorithm does not compute the closed-form solution for the recurrences, but instead uses widening to reach a fixed point. The algorithm handles commutative operators by trying all possible permutations.

Symbolic Execution Matsumoto et al. [26] and Person et al. [31] present different techniques to detect differences between two programs that are mostly equal. Ramos and Engler [34] present an algorithm to check for program equivalence automatically up to a bounded number of loop unrollings.

Software Verification and Invariant Synthesis State-of-the-art software verification tools are unable to prove equivalence of most programs containing loops, since they are usually unable to automatically derive sufficiently strong loop invariants to complete the proof, even if just considering the theory of integer arithmetic, let alone the UF+LIA theory.

Beyer et al. [8] present an algorithm to synthesize loop invariants over the UF+LIA theory, and Rybalchenko and Stokkermans [36] present an algorithm to synthesize interpolants over the same theory. McMillan [28] introduced an algorithm to generate interpolants from the unsatisfiability proofs of Z3 [12]. However, the language of interpolants/invariants supported by these algorithms is not able to express an unbounded number of UF applications, which is often required to prove equivalence of programs that have UF applications inside loops.

Polynomial loop invariant generation techniques (e.g., [29, 35, 38]) can only generate invariants with bounded exponents, which is not sufficient for the verification of the integer programs we generate (after removing the UF applications), since these programs often require loop invariants with unbounded exponents.

Gupta et al. [19] present an algorithm to solve recursion-free Horn clauses in the theory of UF+LIA. Grebenshchikov et al. [15] extend this work to recursive Horn clauses in order to support the verification of recursive programs. The interpolation algorithm used suffers from the same limitations as the others.

Gulwani and Tiwari [17] present an algorithm for the verification of programs over the UF+LIA theory. However, only equalities over UF applications are supported, and conditional branches are abstracted non-deterministically, which is too weak for the application of equivalence checking.

Blanc et al. [10] and Gulwani et al. [16] present algorithms to compute symbolic bounds of loop trip counts. However, the computed trip counts may not be sufficiently precise for equivalence checking proofs.

7 Conclusion

In this paper we presented, as far as we know, the first algorithm for the equivalence checking of looping programs over the combined theory of uninterpreted function symbols and integer arithmetic (UF+IA).

For evaluation purposes, we developed CORK, a tool that proves the correctness of compiler optimizations, which is based on the proposed equivalence checking algorithm. CORK proves correct more optimizations than other tools known as state-of-the-art.

Acknowledgments. The authors thank João Pedro Afonso, Ruslán Ledesma-Garza, and the anonymous reviewers for their comments and suggestions on earlier drafts of this paper.

This work was partially supported by the FCT grants SFRH/BD/63609/2009 and INESC-ID multiannual funding PEst-OE/EEI/LA0021/2011.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.
[2] C. Alias and D. Barthou. On the recognition of algorithm templates. In *COCV*, 2003.
[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.
[4] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, 2011.
[5] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, 2004.
[6] D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. In *Euro-Par*, 2002.
[7] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.
[8] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, 2007.
[9] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, 2011.
[10] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: algebraic bound computation for loops. In *LPAR*, 2010.
[11] S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs. In *VMCAI*, 2012.
[12] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.
[13] B. Godlin and O. Strichman. Regression verification. In *DAC*, 2009.
[14] B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electron. Notes Theor. Comp. Sci.*, 132, 2005.
[15] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.
[16] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, 2009.
[17] S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In *ESOP*, 2006.
[18] S.-y. Guo and J. Palsberg. The essence of compiling with traces. In *POPL*, 2011.
[19] A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *APLAS*, 2011.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.
[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
[22] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software verification platform. In *CAV*, 2005.
[23] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, 2009.
[24] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *CAV*, 2012.
[25] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.
[26] T. Matsumoto, H. Saito, and M. Fujita. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In *ISQED*, 2006.
[27] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
[28] K. L. McMillan. Interpolants from Z3 proofs. In *FMCAD*, 2011.
[29] M. Müller-Olm and H. Seidl. Computing polynomial program invariants. *Inf. Process. Lett.*, 91:233–244, Sept. 2004.
[30] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.
[31] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *SIGSOFT*, 2008.
[32] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.
[33] A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL*, 2007.
[34] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *CAV*, 2011.
[35] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42:443–476, Apr. 2007.
[36] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, 2007.
[37] D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, May 2009.
[38] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *POPL*, 2004.
[39] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In *CC*, 2005.
[40] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *CAV*, 2011.
[41] G. Strang. *Linear Algebra and Its Applications (2nd Ed.)*. Academic Press, 1980.
[42] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, 2005.
[43] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.
[44] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *CAV*, 2009.
[45] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.
[46] A. Zaks and A. Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *FM*, 2008.
[47] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27, 2005.