**Software Tools for Technology Transfer manuscript No.**
(will be inserted by the editor)

# Automatic Equivalence Checking of Programs With Uninterpreted Functions and Integer Arithmetic

**Nuno P. Lopes, José Monteiro**

INESC-ID / Instituto Superior Técnico – Universidade de Lisboa

The date of receipt and acceptance will be inserted by the editor

**Abstract.** Proving equivalence of programs has several important applications, including algorithm recognition, regression checking, compiler optimization verification and validation, 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 semi-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 precise 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 equivalence of programs has several important applications, including, but not limited to, algorithm recognition [3], regression checking [19, 27, 42], compiler optimization verification [23, 34, 41] and validation [55, 58, 66, 70, 73, 75], and information flow proofs [6, 69].

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 semantics 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 associated non-determinism (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 just described. 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 Blast [37, 38], CPAchecker [11], Duality [51], HSF [30], Slam [4], and UFO [2], 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 when limited to 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 semi-algorithm for automatically proving partial equivalence of programs consisting of integer arithmetic operations and applications of UFs. The proposed algorithm is applicable, in particular, to programs containing nested loops.

The algorithm works as follows. 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 loop-free programs. Loops are summarized using 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. The algorithm then composes the two programs sequentially and checks the resulting program for safety.

The proposed algorithm is sound, but necessarily incomplete since it is parameterized by a method for computing closed forms of recurrences, which is known to be an undecidable problem.

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.

This paper extends the work presented at the International SPIN Symposium on Model Checking of Software 2013 [47].

The rest of the paper is organized as follows. Section 2 gives an intuition of how our algorithm proves the equivalence of programs through 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 [41], a state-of-the-art tool for compiler optimization verification. Section 6 presents the related work, and Section 7 concludes.

## 2 Illustrative Example

We illustrate our algorithm for program equivalence checking on a simple example. Figure 1 shows two equivalent example programs, where $f$ is a UF symbol. [1] Our objective is to prove that these two programs are indeed equivalent.

```
$i := 0$
while $i < N$ do
    $k := f(k,\ i)$
    $i := i + 1$
```

```
$i := N$
while $i \geq 1$ do
    $k := f(k,\ N - i)$
    $i := i - 1$

if $N \leq 0$ then
    $i := 0$
else
    $i := N$
```

**Fig. 1.** Example of two equivalent programs.

The first step of the algorithm is to do sequential composition of the two programs, where the second program is renamed to operate over a distinct set of variables from the first. 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. Similarly, we assume that the corresponding variables of the two programs have the same value at the beginning of the composed program. The resulting composed program can be seen in Figure 2.

Now 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. However, state-of-the-art software verification tools are not able to verify the correctness of the program of Figure 2, since it requires complex invariants to be synthesized.

We now show how our algorithm proceeds.

The second step of the algorithm is to replace 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 on $k$ and $i$):

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

where $a$, $b$, and $c$ are fresh 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 fresh variables associated with each input parameter. Similarly, for the UF application of the right program we obtain:

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

Intuitively, these expressions (polynomials) have a unique value for each set of UF symbol and input parameters (since variables $a$, $b$, and $c$ are fresh). Therefore, no other sequence of commands can always produce

[1] As an anecdote, when developing this example, we forgot the **if** command in the program on the right. Fortunately, our prototype quickly pointed out our mistake (of different values of $i$ at the end of the programs when the loops do not execute).

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

$i := 0$
if $i < N$ then
while $i < N$ do
    $k := f(k,\ i)$
    $i := i + 1$

$\bar{i} := \bar{N}$
while $\bar{i} \geq 1$ do
    $\bar{k} := f(\bar{k},\ \bar{N} - \bar{i})$
    $\bar{i} := \bar{i} - 1$

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. 2.** Sequential composition of the programs of Figure 1. The program on the right was renamed, so that each variable $v$ becomes $\bar{v}$.

```
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 $R_{\bar{i}}(\bar{n}-1) \geq 1 \wedge R_{\bar{i}}(\bar{n}) < 1$
    $\bar{k} := R_{\bar{k}}(\bar{n})$
    $\bar{i} := R_{\bar{i}}(\bar{n})$

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.** Program of Figure 2 after removing the loops and the UF applications.

the same value without doing the same UF application with the same inputs, meaning that with this abstraction we do not lose information necessary for the safety proof. This is because there always exists an assignment to fresh variables $a$, $b$, and $c$ that leads to different results for different UF applications, which would therefore violate the assertion.

This transformation is closely related to *polynomial interpolation*, which consists in determining a polynomial of a certain degree that passes through a given set of points.

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 4.2.2.

The third step that the algorithm performs is summarizing and subsequently 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 precisely.

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

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

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:

$$\begin{aligned} R_{\bar{i}}(n) &= R_{\bar{i}}(n-1) - 1 \\ R_{\bar{i}}(0) &= \bar{N} \\ R_{\bar{k}}(n) &= a \times R_{\bar{k}}(n-1) + b \times (\bar{N} - R_{\bar{i}}(n-1)) + c \\ R_{\bar{k}}(0) &= k_0 \end{aligned}$$

Figure 3 shows the programs of Figure 2 after both transformations (elimination of loops and UF applications) have been applied. The references to recurrences were not replaced with their closed-form solutions to avoid cluttering the example.

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.

The expression used in this example to compute the trip count is correct only for linear loop conditions, since in that case there is only one solution for the specified expression. For non-linear loop conditions, we need to compute the minimum positive $n$ that satisfies the expression, which can be accomplished for instance using optimizing solvers or with multiple calls to regular constraint solvers.

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):

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

For the right program, the solution for $R_{\bar{k}}(n)$ is equal to $R_k(n)$ of the left program, and for $R_{\bar{i}}$ is:

$$R_{\bar{i}}(n) = \bar{N} - n$$

The fourth and final step of the algorithm is to prove that the composed program after the described transformations (which is now only over integer arithmetic and loop-free) is correct.

To prove program safety, we can use standard software verification techniques (e.g., software model checking [40]). Since the number of control-flow 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.

## 3 Program Model

We assume that programs are specified in the WHILE language, whose syntax is given in Figure 4. Expressions are side-effect free and 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 this 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. Additionally, we use two additional commands, **assume** and **assert**, as syntactic sugar, which are defined as follows:

$$\textbf{assume } b \equiv \textbf{if } b \textbf{ then skip else } (\textbf{while } 0 \leq 0 \textbf{ do skip})$$

$$\textbf{assert } b \equiv \textbf{if } b \textbf{ then skip else abort}$$

Let $\sigma$ be a program state, which is a map from program variables to integers and from UF symbols to maps

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

**Fig. 4.** WHILE language syntax. $n$ is an integer number, $v$ is a variable name, $f$ 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., $\wedge$, $\vee$).

from tuples of integers (of the same arity as the function) to integers (an interpretation of the UFs). Let $\sigma(v)$ be the value of variable $v$ in program state $\sigma$. Let $\sigma(f)(v_1, \ldots, v_n)$ be the value of the interpretation of the UF symbol $f$ in $\sigma$ applied to $v_1, \ldots, v_n$. This notation is extended for expressions, such that $\sigma(e)$ is the value of expression $e$ with each variable evaluated in $\sigma$. Let $\sigma[v \mapsto n]$ be a program state that is identical to state $\sigma$, except for the value of variable $v$, which is $n$. Let $\sigma_0$ be the initial state of an execution of a program. We have that $\sigma_0(v) = v_0$ and $\sigma_0(f) = f_0$ for each variable $v$ and UF symbol $f$ used in the program, with fresh variables $v_0$ and arbitrary maps $f_0$.

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

The operational semantics of the commands of the WHILE language is shown in Figure 5. The command **abort** is irreducible, and therefore there is no corresponding reduction rule for it.

A program $P$ (a command) is said to be safe iff there is no initial state $\sigma_0$ such that $P$ terminates in an irreducible command, i.e., $P$ is safe iff

$$\neg\exists\sigma_0 : \langle P,\ \sigma_0\rangle \to^* \langle \textbf{abort},\ \sigma'\rangle$$

Let $\mathsf{Vars}(P)$ be the set of variables of program $P$. 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 program $P$ (defined by the user). Let $\sigma \downarrow V$ be the projection of state $\sigma$ over the set of variables $V$ and let $\sigma \downarrow \mathsf{Out}(P)$ be the observable state of $\sigma$ 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 (with $V = \mathsf{Out}(P_1) = \mathsf{Out}(P_2)$):

$$\langle P_1,\ \sigma_0\rangle \to^* \sigma_1 \wedge \langle P_2,\ \sigma_0\rangle \to^* \sigma_2 \implies \sigma_1 \downarrow V = \sigma_2 \downarrow V$$

$$\frac{}{\langle \textbf{skip},\ \sigma\rangle \to \sigma} \qquad \frac{}{\langle v\ :=\ e,\ \sigma\rangle \to \sigma[v \mapsto \sigma(e)]}$$

$$\frac{\langle c_1,\ \sigma\rangle \to \langle c_1',\ \sigma'\rangle}{\langle c_1\ ;\ c_2,\ \sigma\rangle \to \langle c_1'\ ;\ c_2,\ \sigma'\rangle} \qquad \frac{\langle c_1,\ \sigma\rangle \to \sigma'}{\langle c_1\ ;\ c_2,\ \sigma\rangle \to \langle c_2,\ \sigma'\rangle}$$

$$\frac{}{\langle \textbf{abort}\ ;\ c,\ \sigma\rangle \to \langle \textbf{abort},\ \sigma\rangle}$$

$$\frac{\sigma(b) = \mathsf{true}}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2,\ \sigma\rangle \to \langle c_1,\ \sigma\rangle}$$

$$\frac{\sigma(b) = \mathsf{false}}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2,\ \sigma\rangle \to \langle c_2,\ \sigma\rangle}$$

$$\frac{\sigma(b) = \mathsf{true}}{\langle \textbf{while}\ b\ \textbf{do}\ c,\ \sigma\rangle \to \langle c\ ;\ \textbf{while}\ b\ \textbf{do}\ c,\ \sigma\rangle}$$

$$\frac{\sigma(b) = \mathsf{false}}{\langle \textbf{while}\ b\ \textbf{do}\ c,\ \sigma\rangle \to \sigma}$$

**Fig. 5.** Operational semantics of the WHILE language.

## 4 Program Equivalence Checking

In this section, we present the new algorithm to check if two programs over the combined theory of uninterpreted function symbols and integer arithmetic (UF+IA) 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 include UF applications nor depend on variables whose value may depend directly or indirectly on the evaluation of an UF application.

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 (transformation commonly known as loop unswitching [1]). Similarly, phase-change loops can be rewritten as multiple loops, using, e.g., splitter predicates [64].

We speculate that Restriction 4 could be lifted, and give a brief discussion in Appendix A.

### *4.2 Algorithm*

The algorithm has four steps:

1. Sequential composition of the two programs.
2. Eliminate UF applications.
3. Replace loops with recurrences.
4. Check the correctness of the resulting program.

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 relatively complete (under the stated restrictions and for certain variable domains), computing the closed-form solution of recurrences is undecidable, and therefore the overall method is incomplete. A thorough discussion on the completeness of the algorithm is given in Appendix A.

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

#### 4.2.1 Sequential Composition

The first step of the algorithm is to do the sequential composition of the two input programs that we would like to check for equivalence. The second program is renamed so that it operates over a different set of variables from the first.

Let $P_1$ and $P_2$ be the two input programs. 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, we assume that $\mathsf{Out}(P_1) = \mathsf{Out}(P_2)$.

#### 4.2.2 Eliminate UF applications

The second step of the algorithm is to eliminate UF applications. This is accomplished by replacing each UF application with a polynomial over its inputs. This rewriting must only preserve program safety (i.e., the program is safe iff the rewritten program is safe). The program transformation $\mathsf{T}$ shown in Figure 6 implements such a replacement.

The polynomial $\mathsf{p}(f, e_1, \ldots, e_n)$ can be defined in multiple ways in order to accomplish our goal of preserving safety. For the domain of rationals, reals or complex numbers, we can use a standard polynomial usually used to interpolate functions with multiple inputs [25, 56], which is as follows:

$$\sum_{\alpha \cdot \mathbf{1} \leq d} C_\alpha\, X^\alpha$$

$$\mathsf{T}(e) = \begin{cases} n & \text{if } e = n \\ v & \text{if } e = v \\ \mathsf{T}(e_1) \oplus \mathsf{T}(e_2) & \text{if } e = e_1 \oplus e_2 \\ \mathsf{p}\left(f, \mathsf{T}(e_1), \ldots, \mathsf{T}(e_n)\right) & \text{if } e = f(e_1, \ldots, e_n) \end{cases}$$

$$\mathsf{T}(b) = \begin{cases} \mathsf{T}(e) \leq 0 & \text{if } b = e \leq 0 \\ \mathsf{T}(b_1) \otimes \mathsf{T}(b_2) & \text{if } b = b_1 \otimes b_2 \end{cases}$$

$$\mathsf{T}(c) = \begin{cases} \textbf{skip} & \text{if } c = \textbf{skip} \\ v \ := \ \mathsf{T}(e) & \text{if } c = v \ := \ e \\ \mathsf{T}(c_1) \ ; \ \mathsf{T}(c_2) & \text{if } c = c_1 \ ; \ c_2 \\ \textbf{if } \mathsf{T}(b) \textbf{ then } \mathsf{T}(c_1) \textbf{ else } \mathsf{T}(c_2) & \text{if } c = \textbf{if } b \textbf{ then } c_1 \\ & \qquad \textbf{else } c_2 \\ \textbf{while } \mathsf{T}(b) \textbf{ do } \mathsf{T}(c_1) & \text{if } c = \textbf{while } b \textbf{ do } c_1 \\ \textbf{abort} & \text{if } c = \textbf{abort} \end{cases}$$

**Fig. 6.** Definition of the program transformation $\mathsf{T}$.

where $C = \begin{pmatrix} f_1 \cdots f_m \end{pmatrix}$ is an $m$-tuple containing variables $f_i$ associated with the given UF symbol $f$, $X = \begin{pmatrix} e_1 \cdots e_n \end{pmatrix}$ is an $n$-tuple with the input values of the given UF application, the exponent vector $\alpha = \begin{pmatrix} \alpha_1 \cdots \alpha_n \end{pmatrix}$ is an ordered partition with nonnegative entries, and $\alpha \cdot \mathbf{1} = \sum_{i=0}^{n} \alpha_i$ is the usual vector dot product. $X^\alpha = \prod_{i=1}^{n} X_i^{\alpha_i}$ is a monomial of degree $\sum_{i=0}^{n} \alpha_i$, with $X_i = e_i$, and $C_\alpha$ being the element of $C$ corresponding to $\alpha$.

This summation produces a polynomial where each term (a monomial) has a degree up to $d$. The degree of a monomial is the sum of the exponents of its variables. For example, $x^3$ and $x\,y^2$ both have degree three. Polynomial $\mathsf{p}$ is, therefore, a summation of all combinations of monomials of degree up to $d$ with $n$ variables (the number of inputs to the UF application). For example, if we have $d = 3$, $\mathsf{p}\,(f, x, y)$ would be equal to:

$$f_1\,x^3 + f_2\,y^3 + f_3\,x^2\,y + f_4\,x\,y^2 + f_5\,x^2 + f_6\,y^2 + \\ f_7\,x\,y + f_8\,x + f_9\,y + f_{10}$$

The maximum degree $d$ of the monomials is the smallest nonnegative integer that satisfies the following constraint:

$$\mathsf{u}(f) \leq \binom{n+d}{n}$$

where $\binom{n}{k} = \dfrac{n!}{k!\,(n-k)!}$ is the binomial coefficient. We use $m = \binom{n+d}{n}$ to denote the number of monomials of $\mathsf{p}$.

A discussion of the presented polynomial and of alternatives for $\mathsf{p}\,(f, e_1, \ldots, e_n)$ for other domains is given in Appendix B.

The value of $\mathsf{u}(f)$ is the maximum number of times that the given uninterpreted function $f$ is possibly applied with a set of distinct values in each and every *static* program path. Only function applications whose value is possibly used in a boolean expression need to be considered. Function applications appearing in a loop body are only counted once.

Two UF applications are equivalent iff they are of the same UF symbol and have the same input values, for all possible program input. Transformation $\mathsf{T}$ 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 $f_i$ that is used only by applications of that symbol. Therefore, and together with results from the domain of polynomial interpolation (shown in Appendix A), we can guarantee that the value of an UF application cannot be reproduced by any sequence of commands for all inputs.

For example, the following boolean expression

$$f(x,y) = 0 \wedge f(x,z) = 3 \wedge f(w,z) = 1 \wedge f(2,3) = 0 \wedge \\ g(x) \leq 0$$

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

$$\begin{aligned} & f_1\,x^2 + f_2\,y^2 + f_3\,x\,y + f_4\,x + f_5\,y + f_6 = 0 \wedge \\ & f_1\,x^2 + f_2\,z^2 + f_3\,x\,z + f_4\,x + f_5\,z + f_6 = 3 \wedge \\ & f_1\,w^2 + f_2\,z^2 + f_3\,w\,z + f_4\,w + f_5\,z + f_6 = 1 \wedge \\ & 2^2\,f_1 + 3^2\,f_2 + 6\,f_3 + 2\,f_4 + 3\,f_5 + f_6 = 0 \wedge \\ & g_1 \leq 0 \wedge \end{aligned}$$

where all $f_i$ and $g_1$ 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 $g$).

In this expression we have four applications of $f$ with (possibly) different input parameters. Therefore, we have $\mathsf{u}(f) = 4$, and also $n = 2$ (since $f$ has two input parameters). The smallest $d$ such that $4 \leq \binom{2+d}{2}$ is $d = 2$, and so each polynomial has six terms ($m = \binom{4}{2} = 6$). The applications of the uninterpreted function $f$ were, consequently, transformed into summations of all the six monomials of two variables of degree up to two.

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

For example, the optimal value for $\mathsf{u}$ in the following program excerpt is $\mathsf{u}(f) = 3$ (assuming no other UF applications in the rest of the program). Although there are four applications of $f$ with possibly distinct input values, only up to three applications are ever statically encountered and used in a boolean expression in a single path. Moreover, the application of $f$ in the loop body is counted only once in $\mathsf{u}(f)$, despite that that application may appear multiple times in a path in which the loop is traversed more than once.

```
while $i < n$ do
    $k := 2 \times k$
    $j := 0$
    while $j < m$ do
        $k := k + j$
        $j := j + 1$
    $i := i + 1$
```

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

**Fig. 7.** 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.

```
if ... then
    $j := f(x)$
else
    $k := f(y)$

while ... do
    $l := f(l)$

if $f(z) \leq 0 \wedge j \leq 0 \wedge k \leq 0 \wedge l \leq 0$ then
    ...
```

The proof of soundness and completeness of transformation T, i.e., that program $P$ is safe iff program $\mathsf{T}(P)$ is safe is given in Appendix A.

#### 4.2.3 Replace loops with recurrences

The third 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 recurrence for the previous iteration of the outer loop.

An example program and its system of recurrences is shown in Figure 7. The recurrences $R_v(n)$ and $V_v(n)$ represent the value of variable $v$ at iteration $n$ of the inner loop and the outer loop, respectively. 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:

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

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

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 $\sigma_{n-1}(b) \wedge \sigma_n(\neg b)$
    $v := \sigma_n(v)$
else
    assume $n = 0$
```

The fresh variable $n$ represents the number of iterations performed by the loop. State $\sigma_n$ maps each variable to the closed-form solution of its corresponding recurrence at iteration $n$, or to itself if the variable is not modified in the loop body $c$. Variable $v$ ranges over all variables that are possibly modified in the loop body. For the previous example, we have for the inner loop that, 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 of the true branch 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 (e.g., [45]) or by doing multiple calls to an SMT solver. The condition for non-linear guards can also be expressed directly using a quantified formula (with a single quantifier alternation).

For the example in Figure 7, the program after removing the loops is shown in Figure 8. The command "**assume** $y = 0$" at the end can be removed as an optimization, since there are no further uses of $y$ afterward.

```
if $i < n$ then
    assume $V_i(y-1) < n \land V_i(y) \geq n$
    $j := 0$
    if $j < m$ then
        assume $R_j(x-1) < m \land R_j(x) \geq m$
        $j := R_j(x)$
    else
        assume $x = 0$
    $k := V_k(y)$
    $i := V_i(y)$
else
    assume $y = 0$
```

**Fig. 8.** Program of Figure 7 after replacing the loops with a set of assignments over the system of recurrences including $V_i(n)$, $V_k(n)$, and $R_j(n)$.

### 4.2.4 Safety Checking

The fourth and final step of the algorithm is to prove the resulting composed program correct. 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, we can use standard software verification techniques (e.g., model checking [40]). Since the number of paths is finite, we can also use an algorithm that enumerates all paths and tests if any of those makes the condition of the **assert** command falsifiable.

## 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 [43,72]. 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.

### *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, expressions, or conditions that should be matched within a program under optimization.

We show an example optimization (loop unrolling) in Figure 9. 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.

```
                          while $(I+1) < N$ do
                              S
while $I < N$ do              $I := I + 1$
    S                  ⇒      S
    $I := I + 1$              $I := I + 1$

                          if $I < N$ then
                              S
                              $I := I + 1$
```

**Fig. 9.** 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 $N$.

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 the loop unrolling optimization 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 $\mathsf{S}$ 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 $\mathsf{S}$ cannot modify variables $I$ and $N$.

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

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

where $v \in \mathsf{W}(\mathsf{S})$ and $\mathsf{R}(\mathsf{S}) = \{r_1, \ldots, r_n\}$. Template expressions are replaced with a single UF application over their read set.

In the loop unrolling example, $\mathsf{S}$ 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 $\mathsf{S}$ on $x$. While variable $x$ does not appear explicitly in the transformation function, $\mathsf{S}$ does indeed modify $x$ in the example instantiation.

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 $\mathsf{S}$ 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 $\mathsf{S}$ cannot read a variable that it writes to, nor vice versa.

Similarly, to state that template statements $\mathsf{S}$ and $\mathsf{T}$ 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[2], which stands for Compiler Optimization coRrectness checKer. CORK is implemented in OCaml (with approximately 1,100 lines of code), 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 9. 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, and then using Mathematica to check validity of the equivalence assertion. If the equivalence check fails, CORK prints a counterexample path.

CORK performs three optimizations to improve the performance. First, CORK reduces the number of satisfiability queries issued to Mathematica by discharging 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 [41]. 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., [1,52]). The first part is a set of optimizations that do not manipulate loops explicitly. These optimizations 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 [28,75], 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 could not finish the satisfiability check of a constraint within the timeout of 15 minutes.

The execution times of PEC and CORK are within the same order of magnitude, but CORK advances the

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

| Optimization | PEC | Queries | Rec. | 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 elim. | ✓ | 2 | 0 | 0.34s |
| Loop inv. 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 | $\checkmark_p$ | 10 | 12 | 23.45s |
| Loop fusion | $\checkmark_p$ | 10 | 12 | 23.34s |
| Loop interchange | $\checkmark_p$ | 15 | 24 | 29.30s |
| Loop reversal | $\checkmark_p$ | 7 | 5 | 8.41s |
| Loop skewing | $\checkmark_p$ | 16 | 24 | 8.50s |
| Loop flattening | × | — | — | T/O |
| Loop strength reduction | × | 6 | 4 | 5.63s |
| Loop tiling | × | 7 | 9 | 10.94s |

**Table 1.** List of compiler optimizations [1,52], how PEC performs ($\checkmark_p$ 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.

state-of-the-art by being able to prove correct more optimizations than PEC.

## 6 Related Work

Proving the equivalence of programs is known to be undecidable in general. However, there has been some 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 (including invariant and interpolant generation, loop trip counting, and so on).

**Manual and Semi-Automated Proofs** Relational Hoare logic [8] is a proof system that enables the verification of equivalence between two programs. However, the system only supports the verification of structurally equivalent programs (while many optimizations do not obey this constraint). Barthe et al. [5] lift some of the restrictions of this work through the usage of product programs. The set of possible verifiable transformations is still dependent on the set of built-in proof rules.

Liang et al. [46] adapted relational Hoare logic to the setting of concurrent programs. Proofs were mechanized in the interactive theorem prover Coq [9].

**Bisimulation** Parameterized equivalence checking (PEC [41,68]) is a technique to verify the correctness of compiler optimizations automatically. It works by automatically finding a bisimulation relation [62] between the original and the optimized template programs. For structurally different loops, PEC relies on a set of heuristics inspired in [28,75].

**Recurrence Equivalence** Barthou et al. [7] and Shashidhar et al. [65] present different algorithms to prove the equivalence of systems of affine recurrence equations that are structurally similar.

Verdoolaege et al. [71] 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. [49] and Person et al. [57] present different techniques to detect differences between two programs that are mostly equal.

Ramos and Engler [59] present an algorithm to check for program equivalence automatically. The implemented tool is based on KLEE [15] and can only prove equivalence up to a bounded number of loop unrollings.

**Software Verification and Invariant Synthesis** State-of-the-art software verification tools, such as Blast [37, 38], CPAchecker [11], Duality [51], HSF [30], Slam [4], and UFO [2], 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 when limited to the theory of integer arithmetic, let alone the combined theory of uninterpreted function symbols and linear integer arithmetic (UF+LIA). Similar problems have been faced by techniques doing information flow proofs through program self-composition [69].

Beyer et al. [10] present an algorithm to synthesize loop invariants over the UF+LIA theory, and Rybalchenko and Sofronie-Stokkermans [61] present an algorithm to synthesize interpolants over the same theory. McMillan [50] introduced an algorithm to generate interpolants from the unsatisfiability proofs of the Z3 SMT solver [22]. Gulwani et al. [32] present a technique to synthesize invariants based on constraint solving. 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., [14,21,53,60,63]) can only generate non-linear invariants with bounded exponents. However, this is not sufficient for the verification of the non-linear integer programs generated by the algorithm proposed in this paper (after removing the UF applications), since these programs often require loop invariants with unbounded exponents (arising from, e.g., UF applications with self-feedback). Other invariant synthesis techniques, such as the ones based on abstract interpretation (e.g., [20]), usually only support linear arithmetic.

Acceleration (e.g., [13, 18, 29, 39]) is a set of techniques to summarize periodic relations (arising from, e.g., loops) in a precise way. The resulting relation has usually to be expressible either in Presburger arithmetic or in an appropriate abstract domain. However, transitive closures of loops arising from the verification of compiler optimizations are usually not expressible in Presburger arithmetic.

Gupta et al. [35] present an algorithm to solve recursion-free Horn clauses in the theory of UF+LIA. Grebenshchikov et al. [30] extend this work to recursive Horn clauses in order to support the verification of programs with loops and recursive functions. The interpolation algorithm used by the corresponding tool suffers from the same limitations as the others.

Gulwani and Tiwari [33] 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. [12] and Gulwani et al. [31] present algorithms to compute symbolic bounds of loop trip counts. However, the computed trip counts may not be sufficiently precise for equivalence checking proofs.

Godlin and Strichman [26] propose a set of proof rules to prove equivalence of programs and to prove mutual termination using UFs to abstract recursive function calls. Loops are encoded as recursive functions.

The technique of Godlin and Strichman is later extended with the introduction of mutual summaries [36], which consists in logical relations between the input/output relation of the two implementations of each function present in both programs under equivalence checking.

**Compiler Correctness** Many approaches have been proposed to improve the correctness of compilers. We briefly present the ones we have not described yet.

CompCert [44] is a compiler that aims to provide end-to-end correctness guarantees (from a program's source code down to the resulting binary). CompCert was written from scratch with verification in mind, and its correctness proofs are done in Coq. Vellvm [74] is a Coq-based framework that enables the development and verification of compiler optimizations for LLVM.

Translation validation (e.g., [28, 55, 58, 66, 70, 73, 75]) is a technique for establishing the correctness of compiler optimizations *after* the optimization was run by checking the original and optimized programs for equivalence. Namjoshi and Zuck [54] propose augmenting transformation functions so that they generate auxiliary invariants to help the translation validation process, which otherwise could fail to derive those invariants automatically.

Guo and Palsberg [34] present a bisimulation-based technique to reason about the correctness of trace optimizations. Dissegna et al. [23] present a more general framework to reason about the correctness of trace optimizations based on abstract interpretation.

PSyCO [48] is a tool that can automatically synthesize weakest preconditions of compiler optimizations in the form of read and write sets as used in this paper.

## 7 Conclusion and Future Work

In this paper we presented, as far as we know, the first semi-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.

In the future, we plan to study the precise characterization of how the limitations of recurrence solving algorithms reflect on the equivalence verification tasks that the proposed algorithm can effectively handle, as well as how to lift some of the restrictions of the algorithm.

*Acknowledgements.* The authors thank João Pedro Afonso, Ruslán Ledesma-Garza, and the anonymous reviewers for their comments and suggestions on earlier drafts of this paper, as well as João Pimentel Nunes for insightful discussions.

This work was partially supported by the FCT grants SFRH/BD/63609/2009 and INESC-ID multiannual funding PEst-OE/EEI/LA0021/2013.

## A Proof of Soundness and Completeness

Let $\mathsf{S}^P(\sigma)$ be a copy of state $\sigma$ where the interpretation of every uninterpreted function (UF) symbol is replaced with values for variables $f_i$ used in Section 4.2.2. Moreover, the values for variables $f_i$ and the initial variables values $v_0$ are chosen such that for every boolean expression $b$ appearing in program $P$, it is guaranteed that $\sigma(b) = \mathsf{S}^P(\sigma)(\mathsf{T}(b))$. The justification of the existence of such an assignment is given in Theorem 1.

In this section, we use the term free variable to denote logic or program variables (depending on the context) that are not constrained and therefore can take any value. In particular, a free program variable is never assigned to and cannot be constrained in any program path.

Let $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ be, respectively, the set of rational, real, and complex numbers.

**Lemma 1 (Solution for nested $f(x)$).** *For a function $f(x) = a_n x^n + \ldots + a_1 x + a_0$, an arbitrary number of nested applications of $f$ to $x$ can take any value in the codomain ($\mathbb{R}$ or $\mathbb{C}$) if $x$ and $a_n$ are free, i.e., $f(f(\ldots f(x)\ldots)) = b$ always has a solution for fixed $a_{n-1}, \ldots, a_0, b$ and free $a_n$ and $x$.*

*Proof.* The maximum degree of the polynomial given by $p = f(f(\ldots f(x)\ldots)) - b$ in $x$ is $n^k$, where $k > 0$ is the number of applications of $f$. If $n$ (the degree of $x$ in $f(x)$) is odd, then $n^k$ will be odd as well. Therefore, if $n$ is odd, it follows from the intermediate value theorem [67] that there always exists a value for $x$ for arbitrary $a_n, \ldots, a_0, b$ such that $p = 0$.

The maximum degree of $a_n$ in $p$ is given by the following recurrence: $d(k) = n\,d(k-1) + 1$ and $d(0) = 0$. The closed-form solution for this recurrence is $d(k) = \dfrac{n^k - 1}{n - 1}$. Now assume that $n$ is even, since we already proved the lemma for $n$ odd. We can then conclude that $d(k)$ is odd for any nonnegative $k$, and therefore there exists $a_n$ for arbitrary $a_{n-1}, \ldots, a_0, b, x$ such that $p = 0$. □

**Lemma 2 (Solution for conjunction of nested $f(x)$).** *For a function $f(x) = a_n x^n + \ldots + a_1 x + a_0$, a conjunction of nested applications of $f$ of the form $f(f(\ldots f(x_1)\ldots)) = b_1 \wedge \ldots \wedge f(f(\ldots f(x_q)\ldots)) = b_q$ is satisfiable if any of the following statements holds:*

1. *Coefficients $a_i$ range over $\mathbb{C}$ and at least $q \le n+1$ of those are free.*
2. *Variables $x_i$ range over $\mathbb{C}$ and are free.*
3. *$n$ is odd and variables $x_i$ range over $\mathbb{R}$ and are free.*
4. *$q = 1$ and $a_n$ and $x_1$ range over $\mathbb{R}$ and are free.*

*Proof.* For condition 1, we note that there is at least one free coefficient $a_i$ for each polynomial in the conjunction. Then it follows from the fundamental theorem of algebra [67] that it is always possible to find values for the free $a_i$ that satisfy the equalities. Similar reasoning apply for condition 2, where each equality can be solved in order of its respective $x_i$.

Conditions 3 and 4 follow directly from Lemma 1. □

We now state under which conditions the transformation $\mathsf{T}$ as given in Section 4.2.2 is sound and complete, which we will use later to prove Theorems 1 and 2.

**Definition 1.** $\mathsf{T}$ is sound and complete if one of the following statements holds:

1. There are no nested applications of UFs in loops and program variables range over $\mathbb{Q}$, $\mathbb{R}$, or $\mathbb{C}$.
2. Variables $f_i$ range over $\mathbb{C}$.
3. There is only one nested UF application produced by a loop, say $f(f(\ldots f(x)\ldots))$, with $x$ being free, and variables $f_i$ and $x$ ranging over $\mathbb{R}$.
4. $\mathsf{u}(f)$ is odd for all $f$ appearing in nested applications in loops, and the input to these applications are variables that are free and range over $\mathbb{R}$.

We note that $\mathsf{u}(f)$ can always be arbitrarily increased (to, e.g., become odd) if need be. Also, in order to guarantee soundness, program variables and polynomial coefficients can be changed to take values in larger domains (say, convert from $\mathbb{Z}$ to $\mathbb{R}$), by giving up on completeness. With such a change, the algorithm remains sound, i.e., if it proves that two programs are equivalent then they are. However, losing completeness means that the algorithm may fail to prove equivalence of two equivalent programs because a larger variable domain may increase the set of possible behaviors/outcomes of a program, which can lead to the loss of equivalence.

**Definition 2.** We define *statically implied equalities* of UF symbols as the set of all equalities involving applications of UFs that are implied by any *static* path in a given program (e.g., $f(x) = 3$). Nested applications of UFs arising from loops are not unfolded. For example, for a program "**while** … **do** $x := f(x)$" and a path that traverses the loop three times, we only consider the equality $x = f(f(f(x_0)))$.

**Theorem 1 (Existence of $\mathsf{S}^P(\sigma_0)$).** *For every program $P$ respecting Definition 1, $\sigma_0$ is a possible initial state of $P$ iff $\mathsf{S}^P(\sigma_0)$ also is.*

*Proof.* If $P$ does not contain any application of UF symbols, then the statement is trivially correct, since $P = \mathsf{T}(P)$ and therefore $\sigma_0 = \mathsf{S}^P(\sigma_0)$.

Otherwise, we consider the set of statically implied equalities of UF symbols. Let $c$ be the conjunction of the elements of said set that refer only to non-nested UF applications, and $r$ the conjunction of the remaining elements (arbitrarily nested applications from loops). Moreover, we trivially have that $\sigma_0(b) = \sigma_0(c \wedge r)$.

We now assume that all UFs have only one input parameter and that there is only one UF symbol $f$. Therefore, $\mathsf{T}(c)$ can be seen as a linear system $Ax = b$, where $A$ is a square matrix of size $n \times n$ with the powers 0 to $(n-1)$ of the input parameters of the UF applications, and $x$ is a vector with fresh variables $f_i$. Moreover, $A$ is a Vandermonde matrix [67].

For example, for $c = f(x_1) = b_1 \wedge \cdots \wedge f(x_n) = b_n$, $\mathsf{T}(c)$ results in the following linear system:

$$\begin{bmatrix} 1 & x_1^1 & \cdots & x_1^{n-1} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_n^1 & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

If $x_i \ne x_j$ for all $i \ne j$, then $c$ is satisfiable. Moreover, the lines and the columns of the coefficient matrix $A$ are linearly independent, which guarantees that the system has a solution (by the unisolvence theorem [67]). Therefore, $\mathsf{T}(c)$ is also satisfiable.

If there are $i, j$ with $i \ne j$ such that $x_i = x_j$, and $c$ is satisfiable, then $b_i = b_j$. In this case, the corresponding system of $\mathsf{T}(c)$ has infinitely many solutions, and therefore $\mathsf{T}(c)$ is satisfiable as well.

Finally, if $c$ is unsatisfiable, then there are $i, j$ with $i \neq j$ such that $x_i = x_j$ and $b_i \neq b_j$. The linear system of $\mathsf{T}(c)$ has no solution, and therefore $\mathsf{T}(c)$ is unsatisfiable as well.

If $c$ is unsatisfiable, then there is no interpretation for the UF symbols that makes $c$ be $\mathsf{true}$, and therefore we have $\sigma_0(c) = \mathsf{S}^P(\sigma_0)(\mathsf{T}(c)) = \mathsf{false}$. If $c$ is satisfiable, then $\sigma(c)$ may or may not be $\mathsf{true}$ depending on the interpretation of the UFs in $\sigma_0$, but we are always guaranteed to be able to find coefficients for $\mathsf{S}^P(\sigma_0)$ either way such that $\sigma_0(c) = \mathsf{S}^P(\sigma_0)(\mathsf{T}(c))$.

If $c$ contains UFs symbols with more than one parameter, then the resulting polynomials in $\mathsf{T}(c)$ are more complex. Similar reasoning can be done by using generalized versions of the unisolvence theorem for multivariate polynomial interpolation (c.f., [25, 56]).

If $c$ contains multiple UF symbols, the evaluation of $c$ can be split in multiple linear systems, one per symbol.

If $r$ is empty, then the proof is completed.

Otherwise, let $\#c$ and $\#r$ be, respectively, the number of equalities in $c$ and $r$. By definition of $\mathsf{u}$, we have for any $f$ that $\#c + \#r \leq \mathsf{u}(f)$. Moreover, only the first $f_1, \ldots, f_{\#c}$ coefficients are defined by $c$, and the remaining $f_{\#c+1}, \ldots, f_{\mathsf{u}(f)}$ remain free. Therefore, the proof follows immediately from Lemma 2.

For UFs with more than one input parameter, Lemma 2 also applies by observing that the degree of the polynomial obtained by nested applications is dominated by the nested input variable and the coefficient of that parameter. $\square$

From Theorem 1, it follows that if $\mathsf{u}(f)$ is odd, then $\mathsf{u}(f)$ does not need to count with applications with free variables as input. This fact can be used as an optimization to reduce the degree of polynomials to the smallest odd number that is greater than or equal to the number of applications with non-free inputs.

**Theorem 2 (Soundness and completeness of $\mathsf{T}$).** *Transformation $\mathsf{T}$ preserves safety of programs, i.e., for any state $\sigma_0$ and program $P$ respecting Definition 1, the following holds:*

$$\langle P,\ \sigma_0\rangle \to^* \sigma \iff \langle \mathsf{T}(P),\ \mathsf{S}^P(\sigma_0)\rangle \to^* \sigma'$$

*Proof.* The proof goes by structural induction on the syntax of $P$.

The base cases are: $P = \textbf{skip}$, $P = v := e$, and $P = \textbf{abort}$, which are all trivially correct.

For the induction step, we need to consider three cases. As the induction hypothesis, assume that the theorem holds for commands $c_1$ and $c_2$.

For $P = \textbf{if}\ b\ \textbf{do}\ c_1\ \textbf{else}\ c_2$, we have $\mathsf{T}(P) = \textbf{if}\ \mathsf{T}(b)\ \textbf{do}\ \mathsf{T}(c_1)\ \textbf{else}\ \mathsf{T}(c_2)$. By definition of $\mathsf{S}^P(.)$, we know that $\sigma(b) = \mathsf{S}^P(\sigma)(\mathsf{T}(b))$ and therefore $P$ reduces to $c_1$ (resp. $c_2$) iff $\mathsf{T}(P)$ reduces to $\mathsf{T}(c_1)$ (resp. $\mathsf{T}(c_2)$).

For $P = \textbf{while}\ b\ \textbf{do}\ c_1$, we note that since $b$ cannot include nor depend on UF applications (per restriction 4 in Section 4.1), then $\mathsf{T}(b) = b$, and therefore $\mathsf{T}(P) = \textbf{while}\ b\ \textbf{do}\ \mathsf{T}(c_1)$. Moreover, we have that $\sigma_1(b) = \sigma_1'(b)$ for every states $\sigma_1$ and $\sigma_1'$ resulting from the reduction of $c_1$ and $\mathsf{T}(c_1)$, respectively, since $b$ cannot depend on the result of any UF symbol. Therefore we are left to prove that $\langle c_1\ ;\ \ldots\ ;\ c_1,\ \sigma_0\rangle \to^* \sigma$ iff $\langle \mathsf{T}(c_1)\ ;\ \ldots\ ;\ \mathsf{T}(c_1),\ \mathsf{S}^P(\sigma_0)\rangle \to^* \sigma'$, which is covered in the following case.

For $P = c_1\ ;\ c_2$, assume that $\langle c_1,\ \sigma_0\rangle \to^* \sigma_1$ and $\langle \mathsf{T}(c_1),\ \mathsf{S}^P(\sigma_0)\rangle \to^* \sigma_1'$. If $\mathsf{S}^P(\sigma_1) = \sigma_1'$, then the theorem is trivially correct. Otherwise, and without loss of generality, consider that $\mathsf{S}^P(\sigma_1)$ and $\sigma_1'$ differ only in the value of variable $v$ because $c_1$ contained an assignment of the form $v := f(x)$. Let $c_2'$ be a copy of $c_2$ where references to $v$ were replaced with $f(x)$. Therefore, our proof goal of $\langle c_2,\ \sigma_1\rangle \to^* \sigma_2 \iff \langle \mathsf{T}(c_2),\ \sigma_1'\rangle \to^* \sigma_2'$ is equivalent to $\langle c_2',\ \sigma_1\rangle \to^* \sigma_2'' \iff \langle \mathsf{T}(c_2'),\ \mathsf{S}^{c_2'}(\sigma_1)\rangle \to^* \sigma_2'''$, which holds per the induction hypothesis. $\square$

We speculate, but leave the proof for future work, that restriction 4 in Section 4.1 could be lifted altogether, i.e., it may be possible to allow UFs in loop guards. We believe this could be done by counting UF symbols in loop guards twice when computing $\mathsf{u}(f)$ for any symbol $f$. Intuitively, we may only need to interpolate the values of an UF symbol when the loop guard flips (i.e., when in one iteration it was true and in the following it became false).

## B Discussion on Polynomial Interpolation

The polynomial for $\mathsf{p}\,(f, e_1, \ldots, e_n)$ given in Section 4.2.2 requires coefficients to range over the set of rational ($\mathbb{Q}$), real ($\mathbb{R}$), or complex numbers ($\mathbb{C}$). Therefore, for the domain of integers ($\mathbb{Z}$), it is unsound to use the given polynomial, since in general there may not exist integer values for variables $f_i$ such that Theorem 1 holds.

Integer-valued polynomials are polynomials with coefficients in some domain, whose value for every point (or for every interpolating point) is an integer [16]. In particular, it is possible to interpolate a set of points using integer-valued polynomials with rational coefficients [17, 24]. However, these polynomials can only be used if the verification tool used in the algorithm supports rational numbers and their combined operation with integer variables from the remainder of the program.

There is still ongoing research on interpolation by integer-valued polynomials that may yield interesting results that could be of use for our algorithm. We leave as a conjecture that the following polynomial can interpolate any set of $n + 1$ integer points:

$$f(x) = \sum_{i=0}^{n} \left\lfloor \frac{a_i\, x^i}{b_i} \right\rfloor$$

where $a_i$ and $b_i$ are integer coefficients, and $\left\lfloor \frac{x}{y} \right\rfloor$ is the integer division. A drawback of this polynomial is that solving recurrences with integer division is harder than with, say, division in $\mathbb{Q}$, because the function may become discontinuous. Moreover, it is unclear whether it would be possible to amend Theorem 1 for such a polynomial.

## 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. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *Proc. of the 24th International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2012.
3. C. Alias and D. Barthou. On the recognition of algorithm templates. In *Proc. of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification*. Elsevier, Amsterdam, 2003.
4. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 2002.
5. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *Proc. of the 17th International Conference on Formal Methods*. Springer Berlin Heidelberg, 2011.
6. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proc. of the 17th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, Washington, 2004.
7. D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. In *Proc. of the 8th International Euro-Par Conference on Parallel Processing*. Springer Berlin Heidelberg, 2002.
8. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 2004.
9. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004.
10. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *Proc. of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2007.
11. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. of the 23rd International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2011.
12. R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: algebraic bound computation for loops. In *Proc. of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. Springer Berlin Heidelberg, 2010.
13. M. Bozga, R. Iosif, and F. Konečný. Fast acceleration of ultimately periodic relations. In *Proc. of the 22nd International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2010.
14. D. Cachera, T. Jensen, A. Jobin, and F. Kirchner. Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. In *Proc. of the 19th International Conference on Static Analysis*. Springer Berlin Heidelberg, 2012.
15. C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, 2008.
16. P.-J. Cahen and J.-L. Chabert. *Integer-Valued Polynomials*, volume 48 of *Mathematical Surveys and Monographs*. American Mathematical Society, 1997.
17. P.-J. Cahen, J.-L. Chabert, and S. Frisch. Interpolation domains. *Journal of Algebra*, 225(2):794–803, 2000.
18. N. Caniart, E. Fleury, J. Leroux, and M. Zeitoun. Accelerating interpolation-based model-checking. In *Proc. of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008.
19. S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs. In *Proc. of the 13rd International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2012.
20. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, 1978.
21. L. Dai, B. Xia, and N. Zhan. Generating non-linear interpolants by semidefinite programming. In *Proc. of the 25th International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2013.
22. L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proc. of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008.
23. S. Dissegna, F. Logozzo, and F. Ranzato. Tracing compilation by abstract interpretation. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 2014.
24. S. Frisch. Interpolation by integer-valued polynomials. *Journal of Algebra*, 211(2):562 – 577, 1999.
25. M. Gasca and T. Sauer. On the history of multivariate polynomial interpolation. *J. Comput. Appl. Math.*, 122(1-2):23–35, Oct. 2000.
26. B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Inf.*, 45(6):403–439, July 2008.
27. B. Godlin and O. Strichman. Regression verification. In *Proc. of the 46th Annual Design Automation Conference*. IEEE Computer Society, Washington, 2009.
28. B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electron. Notes Theor. Comput. Sci.*, 132:53–71, May 2005.

29. L. Gonnord and P. Schrammel. Abstract acceleration in linear relation analysis. *Sci. Comput. Program.*, 93:125–153, Nov. 2014.
30. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *Proc. of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 2012.
31. S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Proc. of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York, 2009.
32. S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Proc. of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation.* Springer Berlin Heidelberg, 2009.
33. S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In *Proc. of the 15th European Conference on Programming Languages and Systems.* Springer Berlin Heidelberg, 2006.
34. S.-y. Guo and J. Palsberg. The essence of compiling with traces. In *Proc. of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York, 2011.
35. A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *Proc. of the 9th Asian Conference on Programming Languages and Systems.* Springer Berlin Heidelberg, 2011.
36. C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Towards modularly comparing programs using automated theorem provers. In *Proc. of the 24th International Conference on Automated Deduction.* Springer Berlin Heidelberg, 2013.
37. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York, 2004.
38. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York, 2002.
39. H. Hojjat, R. Iosif, F. Konečný, V. Kuncak, and P. Rümmer. Accelerating interpolants. In *Proc. of the 10th International Conference on Automated Technology for Verification and Analysis.* Springer Berlin Heidelberg, 2012.
40. R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, Oct. 2009.
41. S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 2009.
42. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Proc. of the 24th International Conference on Computer Aided Verification.* Springer Berlin Heidelberg, 2012.
43. V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 2014.
44. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.
45. Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik. Symbolic optimization with SMT solvers. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York, 2014.
46. H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York, 2012.
47. N. P. Lopes and J. Monteiro. Automatic equivalence checking of UF+IA programs. In *Proc. of the 20th International SPIN Symposium on Model Checking of Software.* Springer Berlin Heidelberg, 2013.
48. N. P. Lopes and J. Monteiro. Weakest precondition synthesis for compiler optimizations. In *Proc. of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation.* Springer Berlin Heidelberg, 2014.
49. T. Matsumoto, H. Saito, and M. Fujita. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In *Proc. of the 7th International Symposium on Quality Electronic Design.* IEEE Computer Society, Washington, 2006.
50. K. L. McMillan. Interpolants from Z3 proofs. In *Proc. of the International Conference on Formal Methods in Computer-Aided Design.* Springer Berlin Heidelberg, 2011.
51. K. L. McMillan and A. Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, Jan. 2013.
52. S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
53. M. Müller-Olm and H. Seidl. Computing polynomial program invariants. *Inf. Process. Lett.*, 91:233–244, Sept. 2004.
54. K. S. Namjoshi and L. D. Zuck. Witnessing program transformations. In *Proc. of the 20th International Conference on Static Analysis.* Springer Berlin Heidelberg, 2013.
55. G. C. Necula. Translation validation for an optimizing compiler. In *Proc. of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation.* ACM, New York, 2000.
56. P. J. Olver. On multivariate interpolation. *Studies in Applied Mathematics*, 116(2):201–240, 2006.
57. S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, New York, 2008.
58. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems.* Springer Berlin Heidelberg, 1998.
59. D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *Proc. of the 23rd International Conference on Computer Aided Verification.* Springer Berlin Heidelberg, 2011.

60. E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42:443–476, Apr. 2007.
61. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *Proc. of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2007.
62. D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, May 2009.
63. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 2004.
64. R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying loop invariant generation using splitter predicates. In *Proc. of the 23rd International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2011.
65. K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In *Proc. of the 14th International Conference on Compiler Construction*. Springer Berlin Heidelberg, 2005.
66. M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *Proc. of the 23rd International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2011.
67. G. Strang. *Linear Algebra and Its Applications (2nd Ed.)*. Academic Press, 1980.
68. Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *Proc. of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 2010.
69. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proc. of the 12th International Conference on Static Analysis*. Springer Berlin Heidelberg, 2005.
70. J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 2011.
71. S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *Proc. of the 21st International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2009.
72. X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 2011.
73. A. Zaks and A. Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *Proc. of the 15th International Symposium on Formal Methods*. Springer Berlin Heidelberg, 2008.
74. J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 2013.
75. 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.*, 27:335–360, Nov. 2005.