Methods to Improve Completeness of Regression Verification

Maor Veitsman

Methods to Improve Completeness of Regression Verification

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Information
Management Engineering

Maor Veitsman

Submitted to the Senate
of the Technion — Israel Institute of Technology
Elul 5776 Haifa September 2016

This research was carried out under the supervision of Prof. Ofer Strichman as part of the degree in Master of Science In Information Management Engineering in the Faculty of Industrial Engineering and Management.

The results of Chap 2 appeared in

> Ofer Strichman and Maor Veitsman. Regression verification for unbalanced recursive functions. In *Proceedings of FM 2016*. Springer, 2016.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract **1**

1 Introduction **3**
1.1 Regression Verification 3
1.2 Abstract Interpretation 8
1.3 Thesis Outline 8

2 Unbalanced recursion **11**
2.1 Four types of unrolling 12
2.2 A proof rule based on domain partitioning 14
2.2.1 Proving base-case equivalence 15
2.2.2 Proving step-case equivalence 15
2.3 A generalization to mutually recursive functions 16
2.4 Proving equivalence of functions not in lock-step 18
2.5 Related work and competing tools 20
2.6 Conclusions 22

3 Value anlysis **25**
3.1 Value Analysis for recursive functions 25
3.2 Using value analysis to strengthen uninterpreted functions 26
3.2.1 Strengthening uninterpreted functions 26
3.2.2 Intersection of return value ranges 28

4 Future work and Conclusion **31**
4.1 Future work 31
4.2 Conclusion 32

List of Figures

1.1 Two functions to calculate GCD of two nonnegative integers. . . . 5
1.2 After isolation of the functions, i.e., replacing their function calls with calls to the same uninterpreted function U. By definition of uninterpreted functions U enforces partial equivalence of the recursive calls. . . . 6
1.3 RVT generates such a main function for each pair of isolated functions $f, f' \in map_f$ that it attempts to prove partially-equivalent. If f, f' access global variables and the heap, then the construction is more involved. . . . 6
1.4 Call graphs of two programs. Grey nodes represent syntactically equivalent functions . . . 7
1.5 A simple mock recursive function to illustrate value analysis. . . . 9

2.1 The different base cases prevent (1.1) from proving the equivalence of these two functions. After isolation, when $n = 1$, `fact1` returns 1, whereas `fact2` returns $1 * U(0)$, namely a nondeterministic value. 11
2.2 These two functions are not in lock-step, which prevents (1.1) from proving their partial equivalence. After isolation, for e.g., $n = 3$, `sum1` returns $3 + 2 + U(1)$ whereas `sum2` returns $3 + U(2)$, which are not necessarily equal terms. . . . 12
2.3 Four types of unrollings. . . . 13
2.4 Pseudocode of the first and second step of the base-case proof. `i` is increased until there is no assertion failure in *unroll&check*. . . . 15
2.5 Pseudocode of the check program used to prove the step case. . . 16
2.6 To prove equivalence of mutually recursive functions (top) with (2.6), we check separately the equivalence of each pair of functions, while replacing the calls to other functions with calls to UFs (bottom). 17
2.7 The function `sum2` after being unrolled syntactically once. . . . 18
2.8 Pseudocode of the two phases of the base-case proof, for unbalanced recursive functions. . . . 20

2.9 Pseudocode of the check program used to prove the step case equivalence for unrolled functions. 20
2.10 Unrolled and optimized version of the factorial function. 22
2.11 `fact2` contains a special condition. 22

3.1 Two semantically equivalent functions with unreachable different return statements. 26
3.2 We limit the set of possible outputs of the uf. 27
3.3 Functions $f1$ and $f2$ are not semantically equivalent in a free context, but they are indeed equivalent under the context of their calling function `main`. 27
3.4 Functions $f1$ and $f2$ cannot be proven equal without using the intersection of possible return value of $r1$ and $r2$. 29
3.5 Functions $f1$ and $f2$ cannot be proven equal without using the intersection of possible return value of their recursive call. 30

4.1 Function summaries illustration. 31
4.2 Functions $f1$ and $f2$ may only be proven equal when using a function summary bound on the uninterpreted function calls to $r1$ and $r2$. 32

Abstract

In this thesis I address the problem of proving the equivalence of two recursive programs. Specifically we use abstract interpretation to strengthen the premise of our proof rules and tackle recursive functions which have different base cases and/or are not in lock-step.

We show a proof rule for the case of different base cases, based on separating the proof into two parts—inputs which result in the base case in at least one of the two compared functions, and all the rest. We also show how unbalanced unrolling of the functions can solve the case in which the functions are not in lock-step. In itself this type of unrolling may again introduce the problem of the different base cases, and we show a new proof rule for solving it. None of the existing software equivalence checkers (like RÊVE , RVT , SYMDIFF), or general unbounded software model-checkers (like SEAHORN, HSFC, AUTOMIZER) can prove such equivalences.

In addition we use abstract interpretation to help bound the possible outputs of the uninterpreted functions return values thus improving the completeness of the proof. We then study an option of using the intersection of the ranges produced by abstract interpretation for refining the abstraction associated with uninterpreted functions.

Chapter 1

Introduction

1.1 Regression Verification

Given two similar programs P_1, P_2, a mapping map_f between their functions, and a definition of equivalence, Regression Verification [GS08] is the problem of identifying the pairs in map_f that are equivalent to one another. This undecidable problem can be thought of as a special case of *program equivalence*. Program equivalence has been discussed in the literature for over half a century (see, e.g., [IGA64])—mostly in the ACL2 community—as a challenge and a use case for theorem proving (e.g., proving that quick-sort has the same output as merge-sort), but without exploiting the similarity between P_1 and P_2 that is assumed in the case of regression verification. This assumed similarity provides many opportunities for optimizations, and generally leads to a complexity which is dominated by the magnitude of change rather than by the magnitude of P_1 and P_2 themselves.

The classic use-case for regression verification is one in which P_1, P_2 are two consecutive versions of the same program, and the goal is to identify the impact of change. It can be used for checking that refactoring or a performance optimization has not changed the program in a nonintended way. It can also be used for verifying that a bug-fix or an added feature affects only the part intended by the programmer. In a somewhat different direction, it was recently used for proving that the target code of two consecutive versions of a compiler are semantically the same [HLP+13]. In all these applications the typical definition of equivalence that is used is called *partial equivalence*[GS08]. It means that given the same inputs, the two functions return equal outputs, unless at least one of them does not terminate. By 'inputs' we mean the function parameters, global variables that it reads, and the heap; by 'outputs' we mean global variables to which the function writes, the heap and the return value.

There are several methods and tools for regression verification that are available in the public domain. MS-SymDiff [LHKR12] is a tool that reads two BPL (Boogie programming language) [LLM11] files corresponding to P_1 and P_2, and generates a verification condition in BPL for each pair of mapped functions. It uses Boogies's built-in access to Z3 [dMB08] and the invariant generator Duality [McM14] to try and prove the equivalence of functions with loops and recursive calls. SymDiff supports user-defined specifications, which means that partial equivalence is just one possible equivalence criterion; the user can alternatively define any predicate over the inputs and outputs of the two compared functions as the proof obligation, e.g., that the output of f is always smaller or equal to the output of f', for $\langle f, f' \rangle \in map_f$.

The tool Rêve attempts to prove the equivalence of recursive functions (currently individual functions rather than whole programs), and is based on a direct translation to Z3's Horn-clause format. This gives the tool access to Z3's PDR engine [BGMR15], which attempts to prove the equivalence between the two functions by gradually detecting invariants.

The third tool is rvt (Regression Verification Tool)[GS08][RVL]. Improving the rvt proof method for partial equivalence is the focus of this thesis. rvt begins by turning all loops into separate recursive functions, and building a map map_f between the functions. This mapping does not have to be bijective.

Other than partial equivalence, rvt also implements methods to prove mutual termination[EKS15] of two functions. The techniques presented here may also improve completeness of these methods, however in this thesis we concentrate on partial equivalence.

We will now explain how rvt attempts to prove the partial equivalence of two recursive functions. We will then describe how this principle can be applied given two *programs* (rather than just a pair of functions), with what we call the *decomposition* algorithm. The goal of rvt is to prove equivalence of as many pairs of functions as possible.

A proof rule for partial equivalence

To prove the equivalence of two recursive functions, rvt uses a proof rule that essentially applies induction: assume that the two functions are partially equivalent in the recursive calls, and try to prove that they are partially equivalent also in the current call. This is summarized by the following proof rule, for two simple

```
gcd1(int a, int b)
{ int g;
  if (!b) g = a;
  else {
    a = a%b;
    g = gcd1(b, a); }
  return g;
}
```

```
gcd2(int x, int y)
{ int z;
  z = x;

  if (y > 0)
    z = gcd2(y, z%y);
  return z;
}
```

Figure 1.1: Two functions to calculate GCD of two nonnegative integers.

recursive functions f and f':

$$\frac{\textsc{partial-equiv}(\texttt{call}f, \texttt{call}f') \vdash \textsc{partial-equiv}(f\ \texttt{body}, f'\ \texttt{body})}{\textsc{partial-equiv}(f, f')}(\textsc{part-eq}) \tag{1.1}$$

The more general case of mutually recursive functions is discussed at length in [GS08].

To check the premise, we need to replace the recursive calls in f and in f' with an over-approximation of f and f', respectively, that satisfies the predicate '$\textsc{partial-equiv}(\texttt{call}f, \texttt{call}f')$'. This is easy to do (albeit not necessarily the best way in terms of the strength of the method) by replacing the recursive calls with the *same* uninterpreted function: by definition, two instances of the same uninterpreted function are partially equivalent. After the replacement we say that f and f' are *isolated*. The following example, taken from [GS08], demonstrates isolation.

Example 1.1.1. Consider the two functions in Fig. 1.1. Let U be the uninterpreted function such that calls to U replace the recursive calls to `gcd1` and `gcd2`. Figure 1.2 presents the isolated functions. These are now 'flat' functions, i.e., without loops and recursion, and hence their partial equivalence is decidable. If they are indeed partially equivalent, then (1.1) implies that the original functions are partially equivalent as well.

RVT proves the equivalence of a pair of isolated functions f, f' by generating a program of the form appearing in Fig. 1.3, and invoking CBMC, a bounded model checker for C programs, to attempt to formally verify it. If it is successful, then

```
gcd1(int a, int b)
{ int g;
  if (!b) g = a;
  else {
    a = a%b;
    g = U(b, a); }
  return g;
}
```

```
gcd2(int x, int y)
{ int z;
  z = x;

  if (y > 0)
    z = U(y, z%y);
  return z;
}
```

Figure 1.2: After isolation of the functions, i.e., replacing their function calls with calls to the same uninterpreted function U. By definition of uninterpreted functions U enforces partial equivalence of the recursive calls.

```
int main(){
  int n = non_det();
  int ret1,ret2;
  ret1 = f(n);
  ret2 = f'(n);
  assert(ret1 = ret2);
}
```

Figure 1.3: RVT generates such a main function for each pair of isolated functions $f, f' \in map_f$ that it attempts to prove partially-equivalent. If f, f' access global variables and the heap, then the construction is more involved.

f, f' are declared equivalent. The schema shown in the figure is for the simple case in which the two compared functions do not access the heap and global variables.

Program decomposition in RVT

We will now describe the program decomposition algorithm which is implemented in RVT . The full description can be found in [BG12].

RVT receives two C programs P_1, P_2 as input, builds a (possibly partial) map map_f between its functions and global variables, and attempts to prove iteratively the partial equivalence of each $\langle f_1, f_2 \rangle \in map_f$. We prove the partial equivalence of $\langle f_1, f_2 \rangle \in map_f$ by proving the premise of (PART-EQ) using CBMC as the underlying decision procedure.

Before creating map_f RVT converts all loops into recursive functions. Therefor

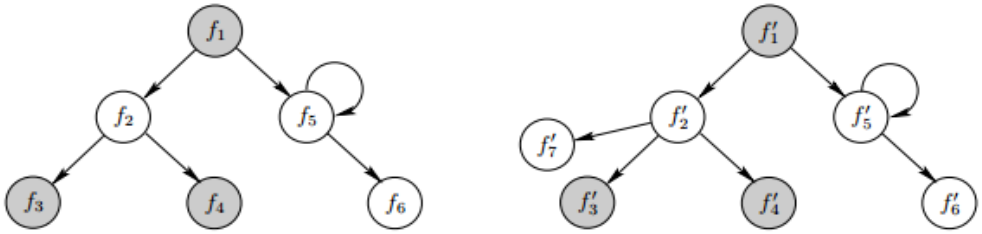

Figure 1.4: Call graphs of two programs. Grey nodes represent syntactically equivalent functions

the inputs of the main algorithm of RVT are two loop-free programs and map_f. Once the previous initial actions are complete RVT constructs two MSCC (Maximal Strongly Connected Component) DAGs (Directed Acyclic Graphs) based on the call graph. An MSCC DAG is a transformation performed on the call graph in which every MSCC in the call graph is collapsed into a single node. Given the two MSCC DAGs which represent the two programs under test and a map between the functions and global variables - map_f RVT attempts to map MSCCs on both sides. If the construction is successful RVT then traverses the DAGs bottom up and attempts to prove the partial equivalence of each pair of functions in every MSCC. For each MSCC RVT iterates over the functions comprising it and for each one it converts all function calls to uninterpreted functions and then proceeds to prove equivalence by proving the premise of (PART-EQ). In the following sections we will address function calls to functions which are outside the DAG (functions which were already proven to be equal) and function calls inside the recursive MSCC (functions which were not yet proven to be equal) differently.

In Fig. 1.4 we present two call graphs which represent two versions of a program. Each vertex in the graph represents a function in the program and an edge between to nodes represents a function call. When two functions are syntactically equivalent their matching nodes are colored grey. Also note that function f'_7 does not exist in the left program. RVT handles this by inlining the code of f'_7 to f'_2. For simplicity we assume that all partial equivalence checks are successful. In table 1.1 we illustrate the order of decomposition of the call graph.

It.	Pair	Description
1.	$\langle f_3, f'_3 \rangle$	These functions are chosen first and marked as equal due to their syntactic equivalence.
2.	$\langle f_4, f'_4 \rangle$	These functions are chosen second and marked as equal due to their syntactic equivalence.
3.	$\langle f_6, f'_6 \rangle$	We assume the partial equivalence proof of f_6 and f'_6 is successful
4.	$\langle f_2, f'_2 \rangle$	Function pairs $\langle f_3, f'_3 \rangle$ and $\langle f_4, f'_4 \rangle$ are proven to be equal and so are converted to uninterpreted functions. Function f'_7 is inlined. We assume the proof is successful.
5.	$\langle f_5, f'_5 \rangle$	Function pairs $\langle f_6, f'_6 \rangle$ are proven to be equal so both functions are converted to uninterpreted functions. Following along the premise of (PART-EQ) we convert the recursive call to an uninterpreted function call. We assume the proof is successful.
6.	$\langle f_1, f'_1 \rangle$	Finally assuming we proved all the predecessor pairs of functions we may convert all calls to uninterpreted functions and prove the equality of $\langle f_1, f'_1 \rangle$.

Table 1.1: This table illustrates the process of call graph decomposition.

1.2 Abstract Interpretation

Abstract interpretation is the theory of sound approximation of the semantics of a computer program. It was first formalized in [CC79]. It is defined as the theory of describing a program as a computation in another domain of abstract objects so that the results can be used to infer useful information on the original program. In this thesis we address specifically abstract interpretation with numerical interval domain as the abstract domain. We call this method value analysis. For example we may use value analysis to compute that the interval domain of the return value of function f in Fig. 1.5 is $[-1..5]$. We will use these intervals to refine the abstraction associated with the uninterpreted functions, and hence to strengthen our proof rules.

1.3 Thesis Outline

In this thesis I introduce and discuss causes of incompleteness of the proof rules used by RVT and suggest methods which tackle them and improve its overall completeness.

In chapter 2 we address two causes of incompleteness of (PART-EQ):

```
int f(int n){
  if (n < 0) return 5;
  if (n > 100) return 4;
  if (f(n-2) < 0) return -1;
  if (f(n+1) > 4) return 4;
  return 0;
}
```

Figure 1.5: A simple mock recursive function to illustrate value analysis.

1. Different base cases

2. Functions not in lock step

We will begin by describing and demonstrating the two problems in detail. We will then offer a solution to the different base-case problem. We will then show that a simple unbalanced unrolling for solving the second problem does not work since it inherently creates the different base-case problem, and hence needs a similar solution to the one we suggest for the first problem. Finally, we discuss the merits and shortcomings of this solution by comparing it to other approaches and existing tools.

In chapter 3 we discuss another contribution to the completeness of our proof rules which uses of *value analysis* as a mechanism to strengthen our proof rule (PART-EQ). We start by describing the method that we utilize to calculate over-approximated intervals of output values of recursive functions. We then show how we embed this method in (PART-EQ) and RVT . Finally, we show how the intersection of the over-approximated intervals which we compute using value analysis, can be used in certain cases to improve the completeness of RVT .

Chapter 2

Unbalanced recursion

The (PART-EQ) rule (1.1) is not, and cannot be, complete, owing to the undecidability of the problem. In this chapter we will focus on two specific reasons for the incompleteness of this rule: *different base-cases*, and *unbalanced recursion*. The former corresponds to a case in which for the same input, one of the functions returns on a base-case, and the other does not. The latter corresponds to a case in which the two recursive functions are not in *lock-step*. We will also consider the case in which both cases occur at the same time. The work presented in this chapter has been published in [SV16]. The examples below demonstrate the weakness of (1.1) when it comes to such cases.

Example 2.0.1. The two programs in Fig. 2.1 are partially equivalent, but (1.1) fails to prove it. The reason is the different base cases. After isolating these two functions, namely replacing their recursive calls with the same uninterpreted function, say U, they may return *different* values when $n = 1$: `fact1` returns 1, whereas `fact2` returns $1 * U(0)$.

Now consider the two partially-equivalent functions in Fig. 2.2. Their base

```
int fact1(int n){
  if (n <= 1) return 1;
  return n * fact1(n-1);
}
```

```
int fact2(int n){
  if (n <= 0) return 1;
  return n * fact2(n-1);
}
```

Figure 2.1: The different base cases prevent (1.1) from proving the equivalence of these two functions. After isolation, when $n = 1$, `fact1` returns 1, whereas `fact2` returns $1 * U(0)$, namely a nondeterministic value.

```
int sum1(int n){
  if (n <= 1) return n;
  return n + n-1 + sum1(n-2);
}
```

```
int sum2(int n){
  if (n <= 1) return n;
  return n + sum2(n-1);
}
```

Figure 2.2: These two functions are not in lock-step, which prevents (1.1) from proving their partial equivalence. After isolation, for e.g., $n = 3$, `sum1` returns $3 + 2 + U(1)$ whereas `sum2` returns $3 + U(2)$, which are not necessarily equal terms.

cases are in sync, but they are not in lock-step: `sum1` computes $\Sigma_{i=1..n} i$ in half the number of iterations compared to `sum2`. After isolation, for equal input n such that $n > 1$, the uninterpreted functions are called with different values, which may lead these two functions to return different values. For example, for $n = 3$ `sum1` returns $3 + 2 + U(1)$, whereas `sum2` returns $3 + U(2)$.

In the next section we will describe our solution strategy.

2.1 Four types of unrolling

Given a recursive function f and a natural *unrolling factor* $i > 0$, we define four types of unrolling of f i times. Fig. 2.3 illustrates the different unrolling types.

1. **Syntactic unrolling**: Create i copies of the original function: $f_1, f_2, .., f_i$, rename them, and rename accordingly their recursive calls. Replace the recursive call in the j-th copy, for $1 \leq j < i$ with a call to the $j+1$ copy. The recursive call in the i-th copy remains unchanged. Let $unroll(f, i)$ denote the syntactically unrolled program.

2. **Unroll and block**: The same as syntactic unrolling, but replace the body of the i-th copy f_i with a single call to `assume`$(false)$[1]. The `assume`(exp) statement restricts program traces to those that satisfy the boolean parameter exp. Hence adding `assume`$(false)$ to our program 'blocks' traces that reach the location of that assertion. Let $unroll\&block(f, i)$ denote this variant of unrolling.

[1]Most software model checkers support `assume` statements.

```
int fact1(int n){
  if (n <= 0) return 1;
  return n * fact1(n-1);
}

int fact(int n){
  if (n <= 0) return 1;
  return n * fact1(n-1);
}
```

***Unroll*(*fact*, 1)**

```
int fact1(int n){
  assume(false);
}

int fact(int n){
  if (n <= 0) return 1;
  return n * fact1(n-1);
}
```

***Unroll*&*Block*(*fact*, 1)**

```
int fact1(int n){
  assert(false);
}

int fact(int n){
  if (n <= 0) return 1;
  return n * fact1(n-1);
}
```

***Unroll*&*check*(*fact*, 1)**

```
int fact1(int n){
  return uf_fact1(n-1);
}

int fact(int n){
  if (n <= 0) return 1;
  return n * fact1(n-1);
}
```

***Unroll*&*UF*(*fact*, 1)**

Figure 2.3: Four types of unrollings.

3. **Unroll and check**: The same as syntactic unrolling, but replace the body of the i-th copy f_i with a single call to `assert`$(false)$. This causes the model checker to fail the proof if there exists a program trace that reaches depth i in the recursion. Let $unroll\&check(f, i)$ denote this variant of unrolling.

4. **Unroll and UF**: The same as syntactic unrolling, but replace the body of the i-th copy of f with a single call statement to an uninterpreted function U_f that is associated with f. We denote this action by $unroll\&uf(f, i)$.

Only the first of these four variants preserves the semantics of the original function f. $unroll\&block(f, i)$ underapproximates f, and $unroll\&uf(f, i)$ overapproximates it. $unroll\&check(f, i)$ is simply a way to check that i is high enough to capture all the traces of f.

We will use these unrolling variants in our proof rules below.

2.2 A proof rule based on domain partitioning

Recall that when the base cases in recursive functions are not in sync, the proof rule (PART-EQ) (1.1) is not strong enough to prove partial equivalence. We suggest a new proof rule for this purpose, in which we break the premise into two separate parts:

$$\frac{\textsc{base-equiv}(f, f') \quad \textsc{step-equiv}(f, f')}{\textsc{partial-equiv}(f, f')}(\textsc{sep-part-eq}) \tag{2.1}$$

Intuitively $\textsc{base-equiv}(f, f')$ is true if f, f' are partially equivalent for any input that invokes the base case in at least one of f, f', and $\textsc{step-equiv}(f, f')$ is true if f, f' are partially equivalent for all the other inputs.

More formally, let $in_B(f)$ denote the set of all inputs for which the resulting program traces do not reach a recursive call in function f, and $in_S(f)$ is the complement of $in_B(f)$. We note that for any f, f' with the same signature, $in_B(f) \cup in_B(f')$ and $in_S(f) \cap in_S(f')$ form a partition of the input domain.

We denote by $\textsc{partial-equiv}(f, f')|_s$ that f and f' are partially equivalent on the set of inputs s. Using this notation, we now define base-case equivalence:

$$\textsc{base-equiv}(f, f') \doteq \textsc{partial-equiv}(f, f')|_{in_B(f) \cup in_B(f')} \tag{2.2}$$

and similarly step-case equivalence:

$$\textsc{step-equiv}(f, f') \doteq \textsc{partial-equiv}(f, f')|_{in_S(f) \cap in_S(f')} \tag{2.3}$$

```
for i = 1... {
  in=non_det();
  ret1=unroll&block(f,1)(in);
  ret2=unroll&check(f',i)(in);
  assert(ret1 = ret2);
}
```

Phase 1

```
for i = 1... {
  in=non_det();
  ret1=unroll&block(f',1)(in);
  ret2=unroll&check(f,i)(in);
  assert(ret1 = ret2);
}
```

Phase 2

Figure 2.4: Pseudocode of the first and second step of the base-case proof. `i` is increased until there is no assertion failure in *unroll&check*.

In the next section we will show how we use the various types of unrolling from Sect. 2.1 to prove the premise of (2.1) based on (2.2) and (2.3).

2.2.1 Proving base-case equivalence

According to (2.2), to prove the premise BASE-EQUIV(f, f'), we can create a check program, similar to the one in Fig. 1.3, but while limiting the inputs to those that invoke the base case in either one of f or f'. To that end, we divide our proof into two phases:

1. Prove equivalence for inputs that result in a base case in f.
2. Prove equivalence for inputs that result in a base case in f'.

The pseudocode in Fig. 2.4 exhibits the programs that we generate for these two phases. By performing *unroll&block* on f, we limit any program trace in the proof that may lead to a recursive call. Because input which results in a base case in function f may result in an unknown number of recursive iterations in function f', we must create a bound for the amount of possible recursive iterations in function f'. We do this by applying *unroll&check* on f', where the unrolling bound is increased up to the point that f' does not make another recursive call, or a time-out is reached.

2.2.2 Proving step-case equivalence

To prove step-case equivalence we must limit our proof to program traces that result in a recursive call on **both** sides. To that end, we have to limit the inputs to $in_S(f) \cap in_S(f')$. Again, we use a program similar to the one in Fig. 1.3. However, we add a global variable *cnt* (initialized to 0), and increment it just

```
in = non_det();
ret1 = unroll&uf(f,1)(in);
ret2 = unroll&uf(f',1)(in);
assert(cnt < 2 || ret1 = ret2);
```

Figure 2.5: Pseudocode of the check program used to prove the step case.

before the call statement to the uninterpreted function (see `fact1` in *unroll&UF* in Fig. 2.3). It is worth to note that due to ease of implementation we chose to move the incrementation of *cnt* to the start of the uninterpreted function. This manipulation keeps the soundness of our technique. We then change our assertion to `assert`$(cnt < 2 \quad || \quad ret1 = ret2)$, where as before $ret1$ and $ret2$ are the return values of the two functions. This way, we check equivalence only for inputs that invoked a recursive call both in f and in f'. Fig. 2.5 illustrates the check program created for the step case.

2.3 A generalization to mutually recursive functions

We now generalize our proof rule to mutually recursive functions. Mutually recursive functions appear in the call graph as SCCs of a size larger than one, and our focus is on maximal SCCs—MSCCs. For simplicity, we consider the case in which the two non-trivial MCSS's m, m' do not have edges outside the MSCC (i.e., functions in m, m' do not call functions outside of m, m') and that there is a bijective mapping between the functions in m, m', which we denote here by map_f. A proof rule for this case was given in [GS08] and repeated here:

$$\frac{\begin{array}{l}\forall(f,f') \in map_f. \\ ((\forall(g,g') \in map_f.\text{p-equiv}(\texttt{call}g, \texttt{call}g')) \vdash \text{p-equiv}(f\ \texttt{body}, f'\ \texttt{body}))\end{array}}{\forall(f,f') \in map_f.\text{p-equiv}(f,f')}(\text{PROC-P-EQ}) \tag{2.4}$$

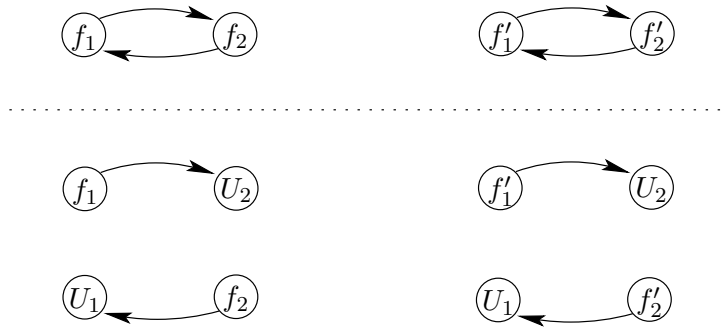

Figure 2.6: To prove equivalence of mutually recursive functions (top) with (2.6), we check separately the equivalence of each pair of functions, while replacing the calls to other functions with calls to UFs (bottom).

This rule is more intuitive after seeing how its premise can be checked. For this, [GS08] defines

$$f^{UF} \doteq f[g \leftarrow UF(g) \mid g \text{ is called in } f] \,, \tag{2.5}$$

or in words, f^{UF} replaces each function call to g in f, with a corresponding call to an uninterpreted function. Now (2.4) becomes

$$\frac{\forall (f, f') \in map_f.\ \textsc{partial-equiv}(f^{UF}, f'^{UF})}{\forall (f, f') \in map_f.\ \textsc{partial-equiv}(f, f')} \tag{2.6}$$

In words, the premise we need to prove is that every pair in map_f has to be proven equivalent, while replacing the calls to other functions in m, m' with uninterpreted functions. We emphasize that the calls to mapped functions in map_f are replaced with the *same* uninterpreted function. A sample pair of size-2-MSCCs and the proof obligations according to (2.6) appear in Fig. 2.6.

Our generalization of (2.1) to mutual recursion, can be thought of as splitting the input domain in (2.4) to the base-case and step-case, similarly to what we have shown in Sect. 2.2:

$$\frac{\forall (f, f') \in map_f.\textsc{base-equiv}(f, f') \quad \forall (f, f') \in map_f.\textsc{step-equiv}(f, f')}{\forall (f, f') \in map_f.\textsc{partial-equiv}(f, f')} \tag{2.7}$$

We now adjust the premise of (2.7) to support mutually recursive functions. First we generalize the definitions of $in_B(f)$ and $in_S(f)$. Let $in_B(f)$ denote the

```
int sum2_1(int n){
 if (n <= 1){
  return n;
 }
 return n + sum2_1(n-1);
}

int sum2(int n){
 if (n <= 1){
  return n;
 }
 return n + sum2_1(n-1);
}
```

```
int sum1(int n){
 if (n <= 1){
  return n;
 }
 return n + n - 1 + sum1(n-1);
}
```

Figure 2.7: The function `sum2` after being unrolled syntactically once.

set of all inputs for which the resulting program traces do not reach a call to another function in the MSCC, and let $in_S(f)$ denote the complement of $in_B(f)$. To prove the base we use the inference rule:

$$\frac{\forall (f, f') \in map_f.\text{PARTIAL-EQUIV}(f, f')|_{in_B(f) \cup in_B(f')}}{\forall (f, f') \in map_f.\text{BASE-EQUIV}(f, f')}\ , \tag{2.8}$$

and to prove the step, we use the rule:

$$\frac{\forall (f, f') \in map_f.\text{PARTIAL-EQUIV}(f, f')|_{in_S(f) \cap in_S(f')}}{\forall (f, f') \in map_f.\text{STEP-EQUIV}(f, f')}\ . \tag{2.9}$$

Since we are only partitioning the input domain in (2.6), whose correctness was already proven in [GS08], then correctness is implied.

2.4 Proving equivalence of functions not in lock-step

Recall the two versions of the `sum` function in Fig. 2.2 which are not in lock-step and therefore cannot be proven partially equivalent by the rule (PART-EQ). To solve this, we unwind `sum2`: the result is shown in Fig. 2.7. We can now see that for $n = 3$ both `sum1` and `sum2` return $3 + 2 + U(1)$.

Now let us look at another example. For $n = 2$, `sum1` returns $2+1+U(0)$ while

the unrolled function `sum2` returns $2 + 1$. By performing un-balanced syntactic unrolling we created base cases that are not in sync. We solve this similarly by separating the premise of our proof rule into two parts, the base-case proof and the step-case proof as before. Our proof rule for functions f and f' with the respective unrolling factors of n and m is:

$$\frac{\text{BASE-EQUIV}_{n,m}(f, f') \quad \text{STEP-EQUIV}_{n,m}(f, f')}{\text{PARTIAL-EQUIV}(f, f')}(\text{SEP-PART-EQ}) \tag{2.10}$$

The predicate $\text{BASE-EQUIV}_{n,m}(f, f')$ is true when f and f' are equivalent for each input that does not involve a recursive call in $unroll(f, n)$ or $unroll(f', m))$. More formally:

$$\begin{aligned}&\text{BASE-EQUIV}_{n,m}(f, f') \doteq \\ &\text{PARTIAL-EQUIV}(unroll(f, n), unroll(f', m))|_{in_B(unroll(f,n)) \cup in_B(unroll(f',m))}\end{aligned} \tag{2.11}$$

The predicate $\text{STEP-EQUIV}_{n,m}(f, f')$ is true when f and f' are partially equivalent for all other inputs: those that involve a recursive call on **both** $unroll(f, n)$, and $unroll(f', m)$ sides, or, more formally:

$$\begin{aligned}&\text{STEP-EQUIV}_{n,m}(f, f') \doteq \\ &\text{PARTIAL-EQUIV}(unroll(f, n), unroll(f', m))|_{in_S(unroll(f,n)) \cap in_S(unroll(f',m))}\end{aligned} \tag{2.12}$$

Next, we show how we verify that these predicates hold true, and thus prove the premise of 2.10.

Base case:

Since we limit our input to values in the union of $in_B(unroll(f, n))$ and $in_B(unroll(f', m))$, we prove the base case by separating the proof into two phases, similarly to Sect. 2.2.1. These phases are illustrated by Fig. 2.8. Note that now, in the first step, we unroll and block f with an unrolling factor of n, in order to capture inputs that result in a program trace that reaches one of the base cases in one of the first n recursive iterations of f. Similarly, in the second phase we apply unroll and block m times on f'.

```
for i = 1...{
  in=non_det();
  ret1=unroll&block(f,n)(in);
  ret2=unroll&check(f',i)(in);
  assert(ret1 = ret2);
}
```

Phase 1

```
for i = 1...{
  in=non_det();
  ret1=unroll&check(f',m)(in);
  ret2=unroll&block(f,i)(in);
  assert(ret1 = ret2);
}
```

Phase 2

Figure 2.8: Pseudocode of the two phases of the base-case proof, for unbalanced recursive functions.

```
cnt = 0;
in = non_det();
ret1 = unroll&uf(f,n)(in);
ret2 = unroll&uf(f',m)(in);
assert(cnt < 2 || ret1 = ret2);
```

Figure 2.9: Pseudocode of the check program used to prove the step case equivalence for unrolled functions.

Step case:

According to (2.12), we need to limit the proof to program traces that result in a recursive call on both sides after being unrolled n and m times, respectively. Similar to the program in Fig. 2.5, we do this by utilizing the counter *cnt*. The program is given in Fig. 2.9.

This entire process is now automated in RVT via a flag `-unroll n m`, where the user only has to replace n and m with constants.

2.5 Related work and competing tools

We have compared several leading unbounded model checkers that have scored high on recursive programs in the latest software verification competition. While they are not designed for program equivalence, Fig. 1.3 shows us that we can reduce this problem to a general verification problem over a single program. We have tested SEAHORN [GKKN15], HSF(C)[GGL+14] and AUTOMIZER [HHP13] using the `factorial` and `sum` examples that we presented in previous sections.

These model checkers were not able to prove equivalent the pair of functions in these examples.

RÊVE [FGK+14] is a program equivalence verifier. Specifically for the case of unbalanced base-cases, it uses a technique called *counterexample-based refinement*, by which counterexamples are checked individually by simulation and then blocked from the verification condition. For example, for our factorial example, the input triggering the base case not in sync is $n = 1$. After the proof failure RÊVE runs both programs with the given input to examine whether the counterexample in fact indicates an in-equivalence between the two programs. If it discovers that both outputs are equal for $n = 1$ then the program trace created by this input is blocked in the next iteration. Once this program input is disregarded then the proof succeeds. This process is unbounded, similarly to the iterative process in our base-case proof seen in Fig. 2.4.

Both tools may perform favourably in different scenarios. For example in programs where our inputs are from the integer domain and the expression applied on the parameter of the recursive call results in a series of inputs $\{n_1, n_2, ..\}$ which advances at a pace larger than one ($n_{i+1} - n_i > 1$) then our proof rule (SEP-PART-EQ) (2.1) may perform better. Generally speaking, (SEP-PART-EQ) may perform better when the expression applied on the parameters of the recursive call causes the series of input values of the recursive calls to be non-sequential over the input domain. Note how the recursion advances faster over the integer domain in the two programs in Fig. 2.10. These are two implementations of the factorial function, after applying loop unrolling. According to the proof method in Equation 2.1 we prove equivalence for two domains of inputs:

1. For inputs that result in a base-case for `fact1` (when $n \in [1..4]$), the proof will be successful in the first iteration.

2. For inputs that result in a base for `fact2` (when $n \in [1..8]$), the proof will be successful in the second iteration.

On the other hand when using counterexample-based refinement, three counterexamples are created and blocked before the proof succeeds.

Counterexample-based refinement may perform better in other scenarios. In Fig. 2.11 we used two identical implementations of the factorial function and added a base condition for $n = 10$. In the second iteration, the program trace created by the input $n = 10$ will be blocked after one iteration using counterexample-based refinement and the proof succeeds. However if we try to prove this using our rule in Equation 2.1, only after we apply *unroll&check* with a factor of 10 will the

```
int fact1(int n){
  if (n <= 1) return 1;
  if (n == 2) return 2;
  if (n == 3) return 6;
  if (n == 4) return 24;


  return n * (n-1) * (n-2)
      * (n-3) * fact1(n-4);
}
```

```
int fact2(int n){
  if (n <= 1) return 1;
  if (n == 2) return 2;
  if (n == 3) return 6;
  if (n == 4) return 24;
  ....
  if (n == 8) return 40320; //8!
  return n * (n-1) * (n-2)
      * (n-3) * fact1(n-4);
}
```

Figure 2.10: Unrolled and optimized version of the factorial function.

```
int fact1(int n){
  if (n <= 1) return 1;
  return n * fact1(n-1);
}
```

```
int fact2(int n){
  if (n <= 1) return 1;
  if (n == 10)
     return 3628800; //10!
  return n * fact1(n-1);
}
```

Figure 2.11: `fact2` contains a special condition.

proof succeed. In other words, the counterexample-refinement method requires fewer iterations with programs that have base-case conditions that are sporadic over the input domain.

2.6 Conclusions

We have presented techniques for proving the equivalence of two recursive functions that have different base-cases and/or are not in lock-step. As we have shown experimentally, none of the existing software equivalence checkers (like RÊVE , RVT , SYMDIFF), or general unbounded software model-checkers (like SEAHORN, HSFC, AUTOMIZER) can prove such equivalences. The proof rule that we presented for the case of different base cases is based on separating the proof into two parts—inputs which result in the base case in at least one of the two compared functions, and all the rest. To prove recursive functions that are not in lock-step, we showed that unbalanced unrolling does not solve in itself the problem, and

requires a more elaborate solution that involves a variation of the first rule for different base cases. We implemented these rules in our regression-verification tool RVT , which now has a web-interface in [RVL] and is open-source.

Chapter 3

Value anlysis

3.1 Value Analysis for recursive functions

Value analysis tools for C programs based on abstract interpretation have been available for many years now. However we were not able to find a tool which supports programs with recursive functions which are the main focus of this work. To integrate value analysis in RVT we have decided to use an existing value analysis tool and add support for recursive functions ourselves. In this work we used FramaC [PCY12]. FramaC is a suite of tools dedicated to the analysis of programs written in C. Specifically we will address the value analysis plugin which computes domains of the variables in the programs including the return value and any global variables.

To calculate value domains for recursive functions we offer Alg. 3.1. To simplify our presentation this algorithm will relate to functions with a single recursive call site and we will also address the return value only while the implementation itself relates to all output variables. The algorithm starts in line 3 by replacing the recursive function call site with a variable which represents the widest possible interval: $[-\infty..\infty]$ (in practice we use the built in minimum and maximum values of the related domain in C). This is implemented using built in constructs which FramaC's value analysis plugin supplies and allow us to bound the possible values of a variable. Now our function is flattened and contains no recursion. In line 4 we run FramaC's value analysis plugin and receive an interval int_res. If $int = int\_res$ we stop because we have reached a fixed point. If a fixed point has not been reached we continue by replacing the recursive call site with a variable bound to the previously computed interval. The algorithm ends when it reaches a fixed point or a timeout has occurred. We are able to use such variables by utilizing domain bounding.

Algorithm 3.1 Computing value analysis on recursive functions

```
INITIALIZE int = [−∞..∞]
while nottimeout do
  Replace recursive call with a variable bounded to int
  Run FramaC given f and store the resulted interval in int_res
  if int = int_res then
    BREAK
  else
    int = int_res
  end if
end while
```

```
int f1(int n){
  if (n <= 0) return 1;
  if (f1(n-1) < 0) return 2;
  else return 5;
}
```

```
int f2(int n){
  if (n <= 0) return 1;
  if (f2(n-1) < 0) return 4;
  else return 5;
}
```

Figure 3.1: Two semantically equivalent functions with unreachable different return statements.

3.2 Using value analysis to strengthen uninterpreted functions

3.2.1 Strengthening uninterpreted functions

Consider the two functions given in Fig. 3.1. Although the two functions return different values in line 3, it is clear that the guards of the `if` statements are never evaluated to true. However RVT is not able to prove the equality of the two functions, because when it creates the check program, both recursive calls are converted to calls to uninterpreted functions, which return arbitrary values.

By utilizing the method in Sect. 3.1 we can calculate that the outputs out_1, out_2 of functions $f1$ and $f2$ respectively hold $out_1, out_2 \in$[1,5] or more specifically $out_1 \in \{1, 2, 5\}$ and $out_2 \in \{1, 4, 5\}$. We can now limit the output of the uf functions to those calculated values. We limit the output of the uf using the `assume`() statement which is provided by CBMC. This is illustrated in Fig. 3.2, although in our implementation we add the `assume`() statement inside the implementation of the uninterpreted function f_{uf}, to limit its output to this range.

We can furthermore strengthen our proof by calculating the range of the

```
int f1(int n){
  if (n <= 0) return 1;
  int tmp = f_uf(n-1);
  assume(tmp == 1 || tmp == 5);
  if (tmp < 0) return 2;
  else return 5;
}
```

Figure 3.2: We limit the set of possible outputs of the uf.

```
int f1(int n){
  if (n <= 0) return -1;
  if (f1(n-1) < 0) return 2;
  else return 5;
}
int main(int x){
  if (x > 1){
        return f1(x);
  }
}
```

```
int f2(int n){
  if (n <= 0) return -1;
  if (f2(n-1) < 0) return 4;
  else return 5;
}
int main(int x){
  if (x > 1){
        return f2(x);
  }
}
```

Figure 3.3: Functions $f1$ and $f2$ are not semantically equivalent in a free context, but they are indeed equivalent under the context of their calling function `main`.

output of each function in a context-based manner. In Fig. 3.3 it is easy to see that functions $f1$ and $f2$ are not equal because the recursive calls may return a negative value. Specifically for the input $n = 1$ the recursive calls in functions $f1$ and $f2$ will return -1 and the final results of the two functions will be different. However under the context in which both functions may only be called when the input satisfies $n > 1$ the functions are equal. By calculating the possible range of outputs using value analysis under this specific context we can conclude that outputs out_1 and out_2 of functions $f1$ and $f2$ respectivelly hold $out_1 \in [2..5]$ and $out_2 \in [4..5]$. With these values we can continue to prove the equality of both functions.

3.2.2 Intersection of return value ranges

In this section we will consider a method to further strengthen uninterpreted functions by intersecting the value ranges calculated by value analysis. When addressing this subject we will consider calls to uninterpreted functions which replace calls to functions which were already proven to be equal and those which are assumed to be equal by the algorithm.

Intersecting output ranges of previously proven functions

RVT converts calls to functions which were already proven to be equal to calls to the same uninterpreted function. It then bounds the output of these functions to the over-approximated output range calculated by value analysis. The precision of this range can be improved, as we now show, which in turn improves the completeness of this proof technique.

Let $\overline{out_f}$ denote the over-approximated set of return values of a function f, as calculated by value analysis, and let out_f denote the precise set of possible return values of f. Note that

$$out_f \subseteq \overline{out_f} \ . \tag{3.1}$$

When two function $f1$ and $f2$ are partially equivalent, $out_{f1} = out_{f2}$ and moreover

$$out_{f1} \subseteq \overline{out_{f1}} \cap \overline{out_{f2}} \ . \tag{3.2}$$

To see why, assume the contrary: this implies that there is a value $x \in out_{f1}$ such that $x \notin \overline{out_{f1}}$ (or, equivalently, $x \notin \overline{out_{f2}}$), but this contradicts (3.1). Hence when we have already proved that $f1, f2$ are partially equivalent, we may limit the bound we place on the return values of the uninterpreted functions that replace them, to

$$\overline{out_{f1}} \cap \overline{out_{f2}} \ . \tag{3.3}$$

The example below demonstrates the value of this restriction.

Consider the two programs in Fig. 3.4. Functions $r1$ and $r2$ can be proven equivalent. Value analysis will result in the ranges $\overline{out_{r1}} = [1..7]$ and $\overline{out_{r2}} = [1..5]$. By replacing the calls to $r1$ and $r2$ in functions $f1$ and $f2$ with uninterpreted functions and bounding the output range to these intervals we are unable to prove the equality of $r1$ and $r2$ because the uf call in line 7 may return the value 7, which causes $f1$ and $f2$ to return a different value. However, by using the range specified in (3.3), we further minimize the bound of return values of $r1$ and $r2$ to the interval $\overline{out_{r1}} \cap \overline{out_{r2}} = [1..5]$. With this range the equivalence of functions $f1$

```
int r1(int n){
  if (n <= 0) return 1;
  if (r1(n-1) < 0) return 7;
  else return 5;
}
int f1(int x){
  if (r1(x) == 7) return -1;
  else return 5;
}
```

```
int r2(int n){
  if (n <= 0) return 1;
  if (r2(n-1) < 0) return 4;
  else return 5;
}
int f2(int x){
  if (r2(x) == 7) return -61;
  else return 5;
}
```

Figure 3.4: Functions $f1$ and $f2$ cannot be proven equal without using the intersection of possible return value of $r1$ and $r2$.

and $f2$ can be proven.

Intersecting ranges of recursive functions

So far we considered the intersection of possible return values of functions which were already proven to be equal. We will now discuss the case of recursive or mutually-recursive functions. Applying the intersection bound on the output of the uninterpreted functions means strengthening the premise of the proof. Soundness is preserved with this addition as we will now see.

The following is the premise of the (PART-EQ) proof rule:

$$\text{p-equiv}(\texttt{call}f1, \texttt{call}f2) \vdash \text{p-equiv}(f1\ \texttt{body}, f2\ \texttt{body}) \tag{3.4}$$

In previous paragraphs we concluded that for two functions $f1$ and $f2$ it holds that $out_{f1}, out_{f2} \in \overline{out_{f1}} \cap \overline{out_{f2}}$ when the two functions are partially equivalent. Therefore by proving the following premise we may soundly conclude that both functions are partially equivalent:

$$\begin{aligned}\text{p-equiv}(\texttt{call}f1, \texttt{call}f2) \wedge (out_{f1}, out_{f2} \in \overline{out_{f1}} \cap \overline{out_{f2}}) \vdash \\ \text{p-equiv}(f1\ \texttt{body}, f2\ \texttt{body})\end{aligned} \tag{3.5}$$

The two sample functions $f1$ and $f2$ in Fig. 3.5 illustrate the benefits of intersection when attempting to prove partial equivalence of recursive functions. $f1$ and $f2$ may only be proved after we bound the results of the uninterpreted functions to the intersection of the possible output values which is: $out_{f1}, out_{f2} \in \{1, 5\}$.

```
int f1(int n){
  if (n <= 0) return 1;
  if (f1(n-1) > 5) return 8;
  else return 5;
}
```

```
int f2(int n){
  if (n <= 0) return 1;
  if (f2(n-1) > 5) return 6;
  else return 5;
}
```

Figure 3.5: Functions $f1$ and $f2$ cannot be proven equal without using the intersection of possible return value of their recursive call.

Chapter 4

Future work and Conclusion

4.1 Future work

In chapter 3 we discussed strengthening the proof rule (PART-EQ) with over approximated bounds on output variables from uninterpreted function calls. The proof rule may be further improved by utilizing function summaries. Intuitively a summary of function f is an over approximated relation between the inputs and outputs of f. The function summary relation which describes function f In Fig. 4.1 is $\{(a, b) \mid a < 5 \text{ and } b = 4\} \cup \{(a, b) \mid a \geq 5 \text{ and } b = 3\}$ where a represents the input and b represents the output.

The two programs in Fig. 4.2 illustrate the possible benefits of using function summaries over abstract interpretation. Functions $r1$ and $r2$ are syntactically equivalent in this sample. Let us assume that they turn into uninterpreted function calls. The over approximated values set of the two functions is $\{1, 5\}$. Using this bound function $f1$ might return the value -1 and function $f2$ may return -2 and so (PART-EQ) will fail. Given a bound on the uninterpreted function calls based on the function summaries of $r1$ and $r2$ we can conclude that the proof will be

```
int f(int a){
  if (a < 5) return 4;
  else return 3;
}
```

Figure 4.1: Function summaries illustration.

```
int r1(int x){
  if (x > 8) return -5;
  return 1;
}

int f1(int x){
  if (x > 10 && r1(x) > 0)
    return -1;
  else return 1;
}
```

```
int r2(int x){
  if (x > 8) return -5;
  return 1;
}

int f2(int x){
  if (x > 10 && r2(x) > 0)
    return -2;
  else return 1;
}
```

Figure 4.2: Functions $f1$ and $f2$ may only be proven equal when using a function summary bound on the uninterpreted function calls to $r1$ and $r2$.

successful because for all inputs greater than 10 functions $r1$ and $r2$ return only negative numbers.

4.2 Conclusion

In this thesis I presented methods which increase the completeness of the proof rules used by RVT . The methods in chapter 2 were aimed at specific use cases: when base case conditions are not in sync and when the recursive steps are not in lock step. The method in chapter 3 may improve completeness on a wide variety of use cases. The methods suggested and implemented in this work improve the overall completeness of RVT and bring it closer to a state in which it can prove real world examples. These methods may also be implemented and integrated into other tools and frameworks.

Bibliography

[BG12] Ofer Strichman Benny Godlin. Regression verification: Proving the equivalence of similar programs. 2012.

[BGMR15] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis framework. In *Proceedings of the 6th Annual Symposium on Principles of Programming Languages*. ACM Press, 1979.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[EKS15] Dima Elenbogen, Shmuel Katz, and Ofer Strichman. Proving mutual termination. *Formal Methods in System Design*, 47(2):204–229, 2015.

[FGK+14] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rummer, and Mattias Ulbrich. Automating regression verification. *Intl. Conference on Automated Software Engineering*, 2014.

[GGL+14] Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Hsf(c): A software verifier based on horn clauses. *Tools and Algorithms for the Construction and Analysis of Systems*, 7214:549–551, 2014.

[GKKN15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015.

[GS08] Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403 – 439, 2008.

[HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. *25th International Conference on Computer Aided Verification*, 2013.

[HLP+13] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 191–201. ACM, 2013.

[IGA64] S. IGARASHI. *An axiomatic approach to equivalence problems of algorithms with applications.* PhD thesis, U. Tokyo, 1964. Rep. Compt. Centre, U. Tokyo 1968, pp 1 - 101.

[LHKR12] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 712–717. Springer, 2012.

[LLM11] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. The boogie verification debugger (tool paper). In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*, volume 7041 of *Lecture Notes in Computer Science*, pages 407–414. Springer, 2011.

[McM14] Kenneth L. McMillan. Lazy annotation revisited. Technical Report MSR-TR-2014-65, MSR, 2014.

[PCY12] Nikolai Kosmatov Virgile Prevosto Julien Signoles Pascal Cuoq, Florent Kirchner and Boris Yakobowski. Frama-c, a software analysis perspective. In *In proceedings of International Conference on Software Engineering and Formal Methods 2012 (SEFM'12), October 2012.*, 2012.

[RVL] RVT online tool. `http://ie.technion.ac.il/~ofers/frontend/`.

[SV16] Ofer Strichman and Maor Veitsman. Regression verification for unbalanced recursive functions. In *Proceedings of FM 2016*. Springer, 2016.