Proving Mutual Termination of Programs

Dima Elenbogen

Proving Mutual Termination of Programs

Research Thesis

Submitted in partial fulfillment of the requirements

for the degree of Master of Science in Computer Science

Dima Elenbogen

Submitted to the Senate of
the Technion — Israel Institute of Technology
Iyar 5774 Haifa May 2014

The research thesis was done under the supervision of Assoc. Prof. Ofer Strichman and under the joint supervision of Prof. Shmuel Katz in the Computer Science Department.

The generous financial support of the Technion is gratefully acknowledged. This material is based on research sponsored by the Air Force Research Laboratory, under agreement number FA8655-11-1-3006. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Two programs are said to be *mutually terminating* if they terminate on exactly the same inputs. We suggest inference rules and a proof system for proving mutual termination of a given pair of functions $\langle f,\ f' \rangle$ and the respective subprograms that they call under a free context. Given a (possibly partial) mapping between the functions of the two programs, the premise of the rule requires proving that given the same arbitrary input in, $f(in)$ and $f'(in)$ call functions mapped in the mapping with the same arguments. A variant of this proof rule with a weaker premise allows to prove termination of one of the programs if the other is *known* to terminate for all inputs. In addition, we suggest various techniques for battling the inherent incompleteness of our solution, including a case in which the interface of the two functions is not identical, and a case in which there is partial information about the partial equivalence (the equivalence of their input/output behavior) of the two given functions.

We present an algorithm for decomposing the verification problem of whole programs to that of proving mutual termination of individual functions, based on our suggested inference rules. The reported prototype implementation of this algorithm is the first to deal with the mutual termination problem.

Abbreviations and Notations

Notation	Page(-s)	Notation	Page(-s)
$\equiv_{\langle o,o' \rangle}$	67	$map_{\mathcal{M}}$	13
$[\![f(\boldsymbol{in})]\!]$	9	$map_{\mathcal{F}}$	8
$\|s\|$	21	$map_{\mathcal{F}}(m)$	13
$\sqsupseteq_c$	69	*mapped*	21
$C(m)$	19	MSCC	13
call-equiv	10	$out(f)$	67
callees	11	*p-equiv*	12
calls	10	$p\text{-}equiv_{\langle o,o' \rangle}$	67
covered	28	*part_eq*	28
DAG	13	$part\_eq_{\langle o,o' \rangle}$	67
depth	75	S	30
$depth_m$	79	$\mathcal{S}(\pi)$	21
$f_{\restriction B}$	61	$\mathcal{S}_m(\pi)$	21
f^{UF}	11	*term*	9, 69
f_S^{UF}	45	*trivial* MSCC	13
input	9	UF_f	10
isolated version	11	Π	61
left side	44	π^1	9
m-term	9, 61	π^{UF}	20
m_term	28		

Chapter 1

Introduction

Whereas termination of a single program has been widely studied (e.g., [7, 10, 11, 18]) for several decades by now, with the focus being, especially in the last few years, on automating such proofs, little attention has been paid to the related problem of proving that two similar programs (e.g., two consecutive versions of the same program) terminate on exactly the same inputs. Ideally one should focus on the former problem, but this is not always possible either because the automatic techniques are inherently incomplete, or because the program does not terminate on all inputs *by design*, e.g., a reactive program. In such cases there is value in solving the latter problem, because developers may wish to know that none of their changes affect the termination behavior of their program. Moreover, the problem and solution thereof can be defined in the granularity of functions rather than whole programs; in this case the developer may benefit even more from a detailed list of pairs of functions that terminate on exactly the same set of inputs. Those pairs that are not on the list can help detecting termination errors.

Our focus is on successive, closely related versions of a program because it both reflects a realistic problem of developers, and offers opportunities for decomposition and abstraction that are not possible with the single-program termination problem. This problem, which was initially proposed in [21] and coined *mutual termination*, is proven undecidable in Appendix A.1 via a simple reduction from the halting problem. We argue, however, that in many cases it is easier to solve automatically, because unlike termination

proofs for a single program, it does not rely on proving that the sequence of states in the programs' computations can be mapped into well-founded sets. Rather, it can be proven by showing that the loops and recursive functions have the same set of function calls given the same inputs, which is relatively easier to prove automatically. In Sect. 3.2, for example, we show how to prove mutual termination of two versions of Collatz's famous $3x+1$ problem [19]; whereas proving termination of this program is open for many decades, proving mutual termination with respect to another version is simple.

Our suggested method for decomposing the proof is most valuable when the two input programs P and P' are relatively similar in structure. In fact, its complexity is dominated by the *difference* between the programs, rather than by their absolute size. It begins by heuristically building a (possibly partial) map between the functions of P and P'. It then progresses bottom-up on the two call graphs [4], and each time proves the mutual termination of a pair of functions in the map, while abstracting their callees. The generated verification conditions are in the form of assertions about 'flat' programs (i.e., without loops and recursive calls), which are proportional in size to the two compared functions. It then discharges these verification conditions with a bounded model-checker (CBMC [9] in our case). Each such program has the same structure: it calls the two compared functions sequentially with the same non-deterministic input, records all subsequent function calls and their arguments, and asserts in the end that they have an equivalent set of function calls. According to our proof rule, the validity of this assertion is sufficient for establishing their mutual termination.

The algorithm is rather involved because it has to deal with cases in which the call graphs of P and P' are not isomorphic (this leads to unmapped functions), with mutually recursive functions, and with cases in which the proof of mutual termination for the callees has failed. It also improves completeness by utilizing extra knowledge that we may give to it on the *partial equivalence* of the callees, where two functions are said to be partially equivalent if given the same inputs they terminate with the same outputs, or at least one of them does not terminate. If we know that two mapped callees are partially equivalent, we abstract them with the same uninterpreted function, which

increases our chance to prove mutual termination. Partial equivalence was studied in [21, 24]. It is implemented in RVT [24] and Microsoft's SymDiff [27]. We also implemented our algorithm in RVT, which enables us to gain this information in a preprocessing step.

To summarize our contributions in this thesis, it presents:

- a proof rule for inferring mutual termination of recursive (and mutually-recursive) functions at the leaves of their respective call graphs,
- an extension of the first rule that applies also to internal nodes in the call graphs, and
- a proof rule for inferring *termination* of one function (not mutual termination) in case the other function is known to be terminating.

More importantly,

- it shows how these rules can be applied to whole programs via a bottom-up decomposition algorithm, and
- reports on a prototype implementation of this algorithm – the first to deal with the mutual termination problem.

Some of the results of the research work to be reported in this thesis have been recently published in [14].

1.1 Structure of this thesis

This thesis is structured as follows. The next chapter gives a formal definition of our problem and describes the preprocessing steps which are applied to programs so that we were able to use our proof rules and decomposition algorithm. Chapter 3 proposes a proof rule for proving mutual termination for functions in mutual recursion and a generalization thereof to cases in which functions outside the mutual recursion component are called as well. Chapter 4 suggests a method for applying the generalized rule to whole programs, based on a bottom-up traversal of the two call graphs. The completeness of that method is the subject of Chapter 5, which proposes several methods for

improving it. Chapter 6 considers a problem related to mutual termination: *assuming* that P terminates, prove that P' terminates. Experiments and conclusions are summarized in Chapter 7. The appendix contains a proof that the mutual termination problem is undecidable.

Chapter 2

Preliminaries

2.1 Preprocessing and mapping

Let P and P' be two programs whose mutual termination is to be checked. The following three preprocessing steps are applied to them:

1. All loops are extracted to new recursive functions. By *function* we refer to a programming language entity, rather than a mathematical entity. The description of this step is thoroughly detailed in Appendix C of [20]. An interested reader can see an example in this thesis how extracting the loops of the programs listed in Fig. 5.1 results in programs as listed in Fig. 5.2 (pages 55–56). Implicit loops caused by *goto* statements directed backward and long-jumps (*goto* outside the function scope) are not supported.

   When this step is passed, no loops remain in functions. Hence, non-termination can only arise from recursion.

2. All global variables that are read by a function are appended to its formal parameter list, and the calling sites are changed accordingly. This is not essential for the proof, but simplifies the presentation. It should be noted that this step in itself is impossible in general programs that access the heap, because it is undecidable whether there exists an input to a function that causes the function to read a particular variable. Our only way out of this problem is to point out that it

is easy to overapproximate this information (in the worst case just take the whole list of global variables) and to state that, based on our experience with a multitude of real programs, it is rather easy to compute this information precisely or slightly overapproximate it with static analysis techniques such as alias analysis. Indeed, the same exact problem exists in RVT and SymDiff for the case of partial equivalence, and there, as in our case, overapproximation can only hinder completeness, not soundness. In general, we will not elaborate on issues arising from aliasing because these are not unique to mutual termination, and are dealt with in [24, 27].

3. A *bijective* map $map_{\mathcal{F}}$ between the functions of P and P' is derived. We apply the same heuristics for deriving this map as those that were used in [23]. For functions $f \in P$ and $f' \in P'$, it is possible that $\langle f, f' \rangle \in map_{\mathcal{F}}$ only if f and f' have the same prototype, i.e., the same list of formal input parameter types. We emphasize that the output of the two functions need not be compatible (e.g., f can update more global variables than f'). The restriction to bijective maps seems detrimental for completeness, because the two compared programs are not likely to have such a map. In practice with inlining such a mapping is usually possible, as we will describe later in Sect. 3.2. Our goal is to prove mutual termination of pairs of functions in $map_{\mathcal{F}}$.

2.2 Definitions and notations

2.2.1 Mutual termination

Although we assume that the compared functions are originally deterministic (i.e., no internal non-determinism resulting from uninitialized variables, etc.), since our methodology introduces non-determinism as part of the solution process, the definitions and proofs in this article will refer to non-deterministic functions. By *non-deterministic* function, we mean a modeling tool for the purpose of verification, i.e., the verification engine checks the verification condition under every possible value in the range defined by the type of the function's return-value.

Given a function f and actual values $\boldsymbol{in}$ for its inputs, let $[\![f(\boldsymbol{in})]\!]$ denote the set of all possible computations of the call of $f(\boldsymbol{in})$, i.e., sequences of states that begin right after the call $f(\boldsymbol{in})$, and are either infinite (in case $f(\boldsymbol{in})$ does not return) or end at the exit from the call. By convention, $[\![f(\boldsymbol{in})]\!] = \emptyset$ if $\boldsymbol{in}$ does not match the input signature of f. Let $term(\pi)$ denote that a given computation π is finite, i.e., it represents a terminating computation. We now define:

Definition 2.2.1 (Mutual termination of functions)**.** Two functions $\langle f, f' \rangle \in map_{\mathcal{F}}$ are *mutually terminating* if and only if

$$\forall \boldsymbol{in}, \boldsymbol{in'}.\ \boldsymbol{in} = \boldsymbol{in'} \rightarrow \forall \pi \in [\![f(\boldsymbol{in})]\!], \pi' \in [\![f'(\boldsymbol{in'})]\!].\ term(\pi) \leftrightarrow term(\pi')\ . \tag{2.1}$$

Note that a function that can either terminate or not terminate on the same input cannot be mutually terminating with any other function according to this definition. Let $m\text{-}term(f, f')$ denote the fact that f and f' are mutually terminating functions. We emphasize that the inputs $\boldsymbol{in}$ and $\boldsymbol{in'}$ may include the heap.

The definition of mutual termination between programs is quite similar to Def. 2.2.1:

Definition 2.2.2 (Mutual termination of programs)**.** Two programs $\langle P, P' \rangle$ are *mutually terminating* if and only if

$$\forall \boldsymbol{in}, \boldsymbol{in'}.\ \boldsymbol{in} = \boldsymbol{in'} \rightarrow \forall \pi \in [\![P(\boldsymbol{in})]\!], \pi' \in [\![P'(\boldsymbol{in'})]\!].\ term(\pi) \leftrightarrow term(\pi')\ , \tag{2.2}$$

where we override $[\![P(\boldsymbol{in})]\!]$ to denote the set of all possible computations of program P with input $\boldsymbol{in}$.

2.2.2 Call equivalence

Given a computation $\pi \in [\![f(\boldsymbol{in})]\!]$, we denote by π^1 the projection of π to states in the top frame of the stack. Note that we follow the convention by which the stack grows downwards. The top frame therefore contains states reachable in the initial call of f. This implies that π^1 includes states in f

```
function f(int n)                          function g(int n)
    if n > 100 then return n−10;               int ret := f(n);
    int temp := g(n + 11);                     return ret;
    return g(temp);
```

Figure 2.1: An implementation of the *McCarthy-91* program [31].

itself, but does not include states in recursive calls to f or in other functions that f calls. An example of a computation π and its corresponding π^1 is given in Fig. 2.2. Let $calls(\pi^1)$ denote the *set* (not a *multiset*) of function-call statements found in π^1, or, formally:

$$calls(\pi^1) \doteq \{\langle g, \boldsymbol{in_g}\rangle \mid g(\boldsymbol{in_g}) \text{ is called in } \pi^1\} \ . \tag{2.3}$$

For example, for π given in Fig. 2.2, $calls(\pi^1) = \{\langle g, 110\rangle, \langle g, 100\rangle\}$. We use the set *calls* to define:

Definition 2.2.3 (Call-equivalence of functions)**.** Functions f and f' are *call-equivalent* if and only if

$$\forall \boldsymbol{in}, \boldsymbol{in'}.\ \boldsymbol{in} = \boldsymbol{in'} \rightarrow \forall \pi \in [\![f(\boldsymbol{in})]\!], \pi' \in [\![f'(\boldsymbol{in'})]\!].\ calls(\pi^1) = calls(\pi'^1) \ . \tag{2.4}$$

Denote by *call-equiv*(f, f') the fact that f and f' are call-equivalent. It is undecidable to determine *call-equiv*(f, f'). We, therefore, abstract the callees as explained next.

2.2.3 Function isolation.

With each function g, we associate an uninterpreted function UF_g such that g and UF_g have the same prototype and return type. This definition is generalized naturally to cases in which g has multiple outputs owing to global data and arguments passed by reference. An uninterpreted function returns a non-deterministic value, but is constrained to return the *same* value if called

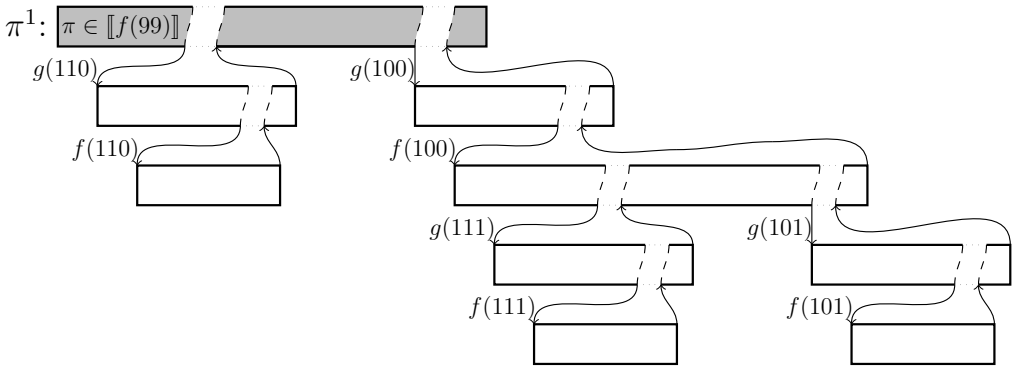

Figure 2.2: An illustration of a computation $\pi \in [\![f(99)]\!]$, where f is defined in Fig. 2.1. π^1 is highlighted with gray shading.

multiple times with the same inputs. In Sec. 3 we will describe how we model this function in a standard programming language. For now it is only important to know that it does not contain function calls or unbounded loops. Let $callees(f)$ denote the set of the functions called in f. We emphasize that $callees(f)$ is defined syntactically, i.e., it contains the functions that appear in the code of f, regardless of whether they are actually called with any particular input. We now define:

$$f^{UF} \doteq f[g(expr_{in}) \leftarrow UF_g(expr_{in}) \mid g \in callees(f)] \,, \tag{2.5}$$

where $expr_{in}$ is the expression(s) denoting actual argument(s) with which g is called. f^{UF} is called the *isolated* version of f. By construction it has no loops or function calls, except for calls to uninterpreted functions, which, recall, in themselves do not call functions or have unbounded loops. For this reason $call\text{-}equiv(f^{UF}, f'^{UF})$ is decidable.

The definition of f^{UF} requires all function calls to be replaced with uninterpreted functions. A useful relaxation of this requirement, which we will later use, is that it can inline non-recursive functions. Clearly, the result is still nonrecursive. Therefore, we still refer to this as an isolated version of f.

2.2.4 Partial equivalence.

The following definition will be used for specifying which functions are associated with the *same* uninterpreted function, when isolating their callers:

Definition 2.2.4 (Partial equivalence of functions)**.** Two functions f and f' are *partially equivalent* if and only if any two terminating computations of f and f' starting from the same inputs, return the same value.

Denote by $p\text{-}equiv(f, f')$ the fact that f and f' are partially equivalent. We enforce that

$$UF_g \equiv UF_{g'} \rightarrow (\langle g, g' \rangle \in map_{\mathcal{F}} \wedge p\text{-}equiv(g, g')) \quad \text{(enforce-1)} , \qquad (2.6)$$

i.e., we associate g and g' with the same uninterpreted function only if $\langle g, g' \rangle \in map_{\mathcal{F}}$, and g, g' were proven to be partially equivalent. The list of pairs of functions that are proven to be partially equivalent is assumed to be an input to the mutual termination algorithm.

Chapter 3

Proof rules

A rule for proving mutual termination of individual 'leaf' functions (i.e., that do not call functions other than themselves) appears in [21]. Here we strengthen that rule by making its premise weaker, and consider the more general problem of proving mutual termination of any pair of functions (including mutually recursive ones), which enables us to consider whole programs.

Given a call graph of a general program, a corresponding directed acyclic graph (DAG) may be built by collapsing each maximal strongly connected component (MSCC) into a single node. Nodes that are not part of any cycle in the call graph (corresponding to non-recursive functions) are called *trivial* MSCCs in the DAG. Other MSCCs correspond to either simple or mutually recursive function(s). Given the MSCC DAGs of the two input programs, denote by $map_{\mathcal{M}}$ a map between their nodes, which is consistent with $map_{\mathcal{F}}$. Namely, if $\langle m, m' \rangle \in map_{\mathcal{M}}$, f is a function in m, and $\langle f, f' \rangle \in map_{\mathcal{F}}$, then f' is a function in m' (and vice-versa).

Consider, then, two nontrivial MSCCs m, m' such that $\langle m, m' \rangle \in map_{\mathcal{M}}$, that are *leaves* in the MSCC DAGs. A projection of $map_{\mathcal{F}}$ to an MSCC m (regardless of whether m is a leaf or not) is defined in the natural way:

$$map_{\mathcal{F}}(m) \doteq \{\langle f, f' \rangle \mid \langle f, f' \rangle \in map_{\mathcal{F}}, f \in m\} \ . \tag{3.1}$$

Our goal is to prove mutual termination of each of the pairs in $map_{\mathcal{F}}(m)$.

The following proof rule gives us a way to do it by proving call-equivalence of each of these pairs:

$$\frac{\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ \textit{call-equiv}(f^{UF}, f'^{UF})}{\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ \textit{m-term}(f, f')} \quad \text{(M-TERM)} \qquad (3.2)$$

Its soundness will be proven in Sect. 3.4.

The premise of (3.2) is weaker than (hence, the rule itself is stronger than) the one suggested in [21], because the latter required the compared functions to be partially equivalent. Furthermore, whereas [21] refers to leaf MSCCs only, later on in this chapter we will generalize (3.2) so it also applies to non-leaf MSCCs, and, hence, tackles the general case.

3.1 Incompleteness

The abstraction of calls using uninterpreted functions is a major source of incompleteness. Two examples of incompleteness are:

- *call-equiv*(f^{UF}, f'^{UF}) may be false, but the counterexample may rely on values returned by an uninterpreted function that are different than what the corresponding concrete function would have returned if called with the same arguments.
- The concrete versions of a function do not terminate, but their abstractions terminate and are followed by different function calls on the two sides, which leads to call equivalence not being true.

We continue this chapter by discussing how the premise of (M-TERM) can be checked (Sec. 3.2) and how this rule can be generalized to whole programs (Sec. 3.3). Finally, in Sec. 3.4, we prove soundness.

3.2 Checking the premise

We check the premise of (3.2) by building a loop- and recursion-free program for each pair of functions that we want to prove call equivalent, which includes

```
function f(int a)
    int even := 0, ret := 0;
    if a > 1 then
        if ¬(a % 2) then                ▷ even
            a := a/2;
            even := 1;
        else a := 3a + 1;
        ret := even + f(a);
    return ret;
```

```
function f′(int a')
    int t′, odd′ := 0, ret′ := 0;
    if a′ ≤ 1 then return ret′;
    t′ := a′/2;
    if a′%2 then                    ▷ odd
        a′ := 6t′ + 4;
        odd′ := 1;
    else a′ := t′;
    ret′ := odd′ + f′(a′);
    return ret′;
```

Figure 3.1: Two variations on the Collatz ("$3x + 1$") function that are mutually terminating. f (f') returns the total number of times the function was called with an even (odd) number. We use the convention that % is the modulo operator, ':=' is an assignment and '=' is equality. Note that when a' is odd, $a'/2 = (a' - 1)/2$, and, hence, $6(a'/2) + 4 = 3a' + 1$.

an assertion whose validity proves the premise. Checking the validity of assertions in such programs is decidable for programming languages with finite types such as C, and indeed our implementation uses the software model checker CBMC [9] for this purpose. Here we describe the construction informally, and only for the case of simple recursion at the leaf functions. We will consider the general case in a more formal way in Chapter 4.

Let f, f' be simple recursive functions that only call themselves. We associate a set of call instructions with each called function. For example, in f only f itself is called, and, hence, we maintain a set of call instructions to f. We then build a program with the following structure: `main` assigns equal non-deterministic values to the inputs of f and f'. It then calls an implementation of f^{UF} and f'^{UF}, and finally asserts that the sets of call instructions are equal. The example below (hopefully) clarifies this construction. From now on, we use the convention in pseudo-codes by which % is the modulo operator, ':=' is an assignment and '=' is equality.

Example 3.2.1. Consider the two variants of the Collatz ("$3x + 1$") pro-

```
function f^UF(int a)
    int even := 0, ret := 0;
    if a > 1 then
        if ¬(a % 2) then                ▷ even
            a := a/2;
            even = 1;
        else a := 3a + 1;
        ret := even + UF(f, a);
    return ret;
```

```
function f'^UF(int a')
    int t', odd' := 0, ret' := 0;
    if a' ≤ 1 then return ret';
    t' := a'/2;
    if a'%2 then                        ▷ odd
        a' := 6t' + 4;
        odd' := 1;
    else a' := t';
    ret' := odd' + UF(f', a');
    return ret';
```

Figure 3.2: The isolated versions of the two given functions of Fig. 3.1.

gram [19] in Fig. 3.1, which return different values (see explanation in the caption of the figure). The Collatz program is a famous open problem in termination: no one knows whether it terminates for all (unbounded) integers. That's why the question whether call equivalence can be proven is particularly interesting. Indeed, proving mutual termination of the two variants given here is easy.

The definitions of f^{UF}, f'^{UF} appear in Fig. 3.2. Note that in this case f, f' are not partially equivalent, and, therefore, according to (2.6) we replace the recursive calls with different uninterpreted functions. Indeed, we call UF above with two different function indices (f and f'), which means that on equal values of a and a' they do not necessarily return the same nondeterministic value.

A *proof-theoretic* method for establishing *call-equiv*(f^{UF}, f'^{UF}) formulates a verification condition which is valid only if the two functions are call-equivalent. For this purpose we need to represent the transition relation of the two functions $T_{f^{UF}}, T_{f'^{UF}}$, which can be easily done with the help of the static single assignment form [5, 36], e.g., we use $a_0, a_1, ...$ for the different versions of a. Fig. 3.3 presents $T_{f^{UF}}$ and $T_{f'^{UF}}$. In this case the verification

$even_0 = 0 \wedge$ $ret_0 = 0 \wedge$ $a_1 = a_0/2 \wedge$ $even_1 = 1 \wedge$ $a_2 = 3a_0 + 1 \wedge$ $a_3 = \neg(a_0 \% 2)\ ?\ a_1 : a_2 \wedge$ $even_2 = \neg(a_0 \% 2)\ ?\ even_1 : even_0 \wedge$ $ret_1 = a_0 > 1\ ?\ even_2 + UF(f, a_3)$ $: ret_0$	$t'_0 = 0 \wedge$ $odd'_0 = 0 \wedge$ $ret'_0 = 0 \wedge$ $t'_1 = a'_0/2 \wedge$ $a'_1 = 6t'_1 + 4 \wedge$ $odd'_1 = 1 \wedge$ $a'_2 = t'_1 \wedge$ $a'_3 = a'_0 \% 2\ ?\ a'_1 : a'_2 \wedge$ $odd'_2 = a'_0 \% 2\ ?\ odd'_1 : odd'_0 \wedge$ $ret'_1 = \neg(a'_0 \leq 1)\ ?\ odd'_2 + UF(f', a'_3)$ $: ret'_0$
$T_{f^{UF}}$	$T_{f'^{UF}}$

Figure 3.3: Transition relations $T_{f^{UF}}$ and $T_{f'^{UF}}$ derived from f^{UF} and f'^{UF}, respectively. The definitions of f^{UF} and f'^{UF} appear in Fig. 3.2. The *static single-assignment* form [36] is used, e.g., $a_0, a_1, ...$, stand for the different versions of a.

condition is:

$$\begin{array}{ll} (T_{f^{UF}} \ \wedge \ T_{f'^{UF}} \ \wedge \ a_0 = a'_0) \ \rightarrow & \triangleright \text{ given the same inputs} \\ \big(((a_0 > 1) \ \leftrightarrow \ \neg(a'_0 \leq 1)) \ \wedge & \triangleright \text{ equal guards} \\ ((a_0 > 1) \ \rightarrow & \triangleright \text{ if called, then} \\ \quad (a_3 = a'_3))\big) \ . & \triangleright \text{ equal arguments} \end{array} \tag{3.3}$$

This is easy to validate using a decision procedure for linear arithmetic and uninterpreted functions [28].

In this case we are able to prove termination without partial equivalence, because the return values of UF_f and $UF'_{f'}$ affect neither the guarding conditions nor the input arguments of other function calls. We defer the

function UF(function index g, input arguments $\boldsymbol{in}$)
  if $\boldsymbol{in} \in args[g]$ **then return** the output of the earlier call UF(g, $\boldsymbol{in}$);
  $args[g] := args[g] \bigcup \boldsymbol{in}$;
  return a non-deterministic value;

function MAIN
  for each $\langle g, g' \rangle \in map_{\mathcal{F}}$ **do** $args[g] := args[g'] := \emptyset$;
  $\boldsymbol{in} := nondet()$; $f^{UF}(\boldsymbol{in})$; $f'^{UF}(\boldsymbol{in})$;
  assert($args[f] = args[f']$); ▷ checks call equivalence

Figure 3.4: The flat program that we generate and then verify its assertion, given the two functions of Fig. 3.1. Note that in the pseudo-code above, `UF(g,`$\boldsymbol{in}$`)` represents $UF_g(\boldsymbol{in})$ in the main text of the thesis. The definitions of f^{UF} and f'^{UF} appear in Fig. 3.2.

presentation of the case in which the functions are known to be partially equivalent to Chapter 4.

□

In stark contrast to the corresponding termination problem (recall that termination of the Collatz program is not known), the demonstrated proof-theoretic method proved call-equivalence (and thus mutual termination by (M-TERM)) in Ex. 3.2.1 even when the variable types were infinite. Its major disadvantage is that this method requires a program analysis in order to derive the conditions. So we choose instead a *model-theoretic* method because it is supported by the tool we use. The model-theoretic method delegates most of the analysis to an off-the-shelf model checker. Instead of analyzing the code to derive a general formula expressing the conditions of calls and the actual arguments, derived programs are generated that would record all arguments to the relevant functions. These programs are never actually executed. Instead, an assertion of equivalence between the sets of arguments with which the functions are called is model-checked, showing it true in every possible computation, and thus automatically detecting call-equivalence.

Fig. 3.4 demonstrates an example of such a generated program for the two variations on the Collatz functions of Ex. 3.2.1. The top of Fig. 3.4 shows

an implementation `UF` of the uninterpreted functions. It receives a function index (abusing notation for simplicity, we assume here that a function name represents also a unique index) and the actual arguments. It records the set of call instructions in the array `args`.

The assertion in `MAIN`, shown in the bottom of Fig. 3.4, is verified by a model-checker. The model checker we use is CBMC [9]. We will elaborate on this method in Chapter 4. Although the comparison between the proof-theoretic and model-theoretic methods is not fair, because the latter assumes finite types, our choice of the model-theoretic approach is sufficiently sound as we target C programs. CBMC is able to reason about C programs, in which variables are of finite types.

What if there is no bijective map $map_{\mathcal{F}}$, or if some of the pairs of functions cannot be proven to be mutually terminating? It is not hard to see that it is sufficient to prove mutual termination of pairs of functions that together intersect all cycles in m, m', whereas the other functions are inlined. The same observation was made with regard to proving *partial equivalence* in a technical report [22]. This observation can be used to improve completeness: even when there is no bijective mapping or when it is impossible to prove mutual termination for all pairs in m, m', it is still sometimes possible to prove it for some of the pairs. The algorithm that we describe in Chapter 4 uses this observation.

We continue in the next section by generalizing (M-TERM) to the case in which there are calls to functions that are defined outside the MSCCs.

3.3 Generalization

We now generalize (M-TERM) to the case that m, m' are not leaf MSCCs. This means that there is a set of functions $C(m)$ outside of m that are called by functions in m. $C(m')$ is defined similarly with respect to m'. The premise now requires that these functions are mutually-terminating:

$$\frac{\begin{array}{l}\forall\langle f, f'\rangle \in map_{\mathcal{F}}(m).\ call\text{-}equiv(f^{UF}, f'^{UF}) \wedge \\ (\forall\langle g, g'\rangle \in map_{\mathcal{F}}.\ ((g \in C(m) \wedge g' \in C(m')) \rightarrow m\text{-}term(g, g'))\end{array}}{\forall\langle f, f'\rangle \in map_{\mathcal{F}}(m).\ m\text{-}term(f, f')}\ (\text{M-TERM}^{+})\ . \tag{3.4}$$

Recall that (2.5) prescribes that calls to functions in $C(m)$ and $C(m')$ are replaced with calls to uninterpreted functions in f^{UF}, f'^{UF}.

We continue in the next section with soundness proofs.

3.4 Soundness proofs for (M-TERM) and (M-TERM^{+})

We begin by defining, for a given a computation π:

$$\pi^{UF} \doteq \pi^{1}[g(\boldsymbol{in_g}) \leftarrow UF_g(\boldsymbol{in_g}) \mid \langle g, \boldsymbol{in_g}\rangle \in calls(\pi^{1})]\ , \tag{3.5}$$

namely, we replace the function calls with calls in π^1 to their respective uninterpreted functions, with the same arguments. It is not hard to see that

$$\frac{\pi \in [\![f(\boldsymbol{in})]\!] \wedge term(\pi)}{\pi^{UF} \in [\![f^{UF}(\boldsymbol{in})]\!]}\ . \tag{3.6}$$

When π is infinite, on the other hand, there may be statements in f that would be executed if the non-terminating call *would* have returned. Since the call is replaced by an uninterpreted function that *does* return, those statements will be executed in f^{UF}. In such a case there must exist a computation $\hat{\pi}$ in $[\![f^{UF}(\boldsymbol{in})]\!]$ that extends π^{UF}. In other words, π^{UF} is a prefix of $\hat{\pi}$. More formally, letting $prefix(\pi^{UF}, \hat{\pi})$ denote that π^{UF} is a prefix of $\hat{\pi}$, we have

$$\frac{\pi \in [\![f(\boldsymbol{in})]\!]}{\exists\hat{\pi} \in [\![f^{UF}(\boldsymbol{in})]\!].\ prefix(\pi^{UF}, \hat{\pi})}\ . \tag{3.7}$$

Lemma 3.4.1. For any given pair of functions $\langle f, f'\rangle \in map_{\mathcal{F}}$, function g',

and inputs $\boldsymbol{in}, \boldsymbol{in_g}$, the following inference is sound for any $\pi \in [\![f(\boldsymbol{in})]\!]$:

$$\frac{term(\pi) \wedge call\text{-}equiv(f^{UF}, f'^{UF}) \wedge \exists \pi' \in [\![f'(\boldsymbol{in})]\!].\ \langle g', \boldsymbol{in_g} \rangle \in calls(\pi'^1)}{\exists g.\ (\langle g, g' \rangle \in map_{\mathcal{F}} \wedge \langle g, \boldsymbol{in_g} \rangle \in calls(\pi^1))} \tag{3.8}$$

Proof. Let $\langle f, f' \rangle \in map_{\mathcal{F}}$, input $\boldsymbol{in}$, function g' and input $\boldsymbol{in_g}$ satisfy the premise. The bijectivity of $map_{\mathcal{F}}$ ensures existence of a function g such that $\langle g, g' \rangle \in map_{\mathcal{F}}$.

By (3.7) π'^{UF} is a prefix of some $\hat{\pi}' \in [\![f'^{UF}(\boldsymbol{in})]\!]$. Note that $\langle UF_{g'}, \boldsymbol{in_g} \rangle \in calls(\pi'^{UF})$, which implies $\langle UF_{g'}, \boldsymbol{in_g} \rangle \in calls(\hat{\pi}')$. Hence, $call\text{-}equiv(f^{UF}, f'^{UF})$ implies:

$$\forall \hat{\pi} \in [\![f^{UF}(\boldsymbol{in})]\!].\ \langle UF_g, \boldsymbol{in_g} \rangle \in calls(\hat{\pi})\ . \tag{3.9}$$

The premise of (3.6) holds, which implies $\pi^{UF} \in [\![f^{UF}(\boldsymbol{in})]\!]$. Thus (3.9) implies $\langle UF_g, \boldsymbol{in_g} \rangle \in calls(\pi^{UF})$. The construction of π^{UF} implies $\langle g, \boldsymbol{in_g} \rangle \in calls(\pi^1)$. □

Given a computation $\pi \in [\![f(\boldsymbol{in})]\!]$, let $\mathcal{S}(\pi)$ denote the set of call-stacks appearing during π, and for $s \in \mathcal{S}(\pi)$ let $|s|$ be the number of frames in s (possibly infinite). Let $\mathcal{S}_m(\pi)$ denote the subset of stacks in $\mathcal{S}(\pi)$ that consist solely of functions in a given MSCC m. Given a call-stack $s' \in \mathcal{S}(\pi')$, let $mapped(s')$ denote a call-stack which holds the following: for each $i \in \mathbb{N}$, $f'_i(\boldsymbol{in_i})$ is the i-th call in s' if and only if the i-th call in $mapped(s')$ is $f_i(\boldsymbol{in_i})$ such that $\langle f_i, f'_i \rangle \in map_{\mathcal{F}}(m)$. Obviously, $s' \in \mathcal{S}_{m'}(\pi')$ implies that $mapped(s')$ consists solely of calls of functions from m such that $\langle m, m' \rangle \in map_{\mathcal{M}}$. Further, given a function call $f(\boldsymbol{in})$, let $[f(\boldsymbol{in})]$ denote the stack-frame of this call; for brevity, we also let $[f(\boldsymbol{in})]$ denote a call-stack which consists of this only frame. Given a non-empty finite call-stack s, let $s \cdot [g(\boldsymbol{in_g})]$ denote the call-stack resulted right after calling $g(\boldsymbol{in_g})$ from the bottom frame of s.

The next lemma addresses call-stacks consisting of calls of mutually recursive functions belonging to the same MSCC, i.e., call-stacks without outer calls. It briefly states that given call-equivalence between all the abstractions, if some finite call-stack appears in a computation of one side, then its cor-

responding mapped call-stack appears in all finite computations of the other side.

Lemma 3.4.2. For any given $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$, input $\boldsymbol{in}$, finite call-stack s', and computation $\pi \in [\![f(\boldsymbol{in})]\!]$:

$$\frac{\begin{array}{l} term(\pi) \wedge \forall \langle h, h' \rangle \in map_{\mathcal{F}}(m).\ call\text{-}equiv(h^{UF}, {h'}^{UF}) \wedge \\ \quad \exists \pi' \in [\![f'(\boldsymbol{in})]\!].\ s' \in \mathcal{S}_{m'}(\pi') \end{array}}{mapped(s') \in \mathcal{S}_m(\pi)} \ . \tag{3.10}$$

Proof. Consider $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$, input $\boldsymbol{in}$, finite call-stack s', and computation $\pi \in [\![f(\boldsymbol{in})]\!]$ which satisfy the premise. Since s' is finite, there is some value d such that $|s'| \leq d$. The proof is by induction on the bound d.

Base: For $d = 1$, the only call-stack of size 1 in any $\pi' \in [\![f'(\boldsymbol{in})]\!]$ is $[f'(\boldsymbol{in})]$. Analogously, the only call-stack of size 1 in any $\pi \in [\![f(\boldsymbol{in})]\!]$ is $[f(\boldsymbol{in})]$. Note that $mapped([f'(\boldsymbol{in})]) = [f(\boldsymbol{in})]$. Further note that $f \in m$, which implies $mapped([f'(\boldsymbol{in})]) \in \mathcal{S}_m(\pi)$.

Step: Assume that the rule holds up to a given d and the premise holds at $d+1$ for a call $f'(\boldsymbol{in})$. Consider a call-stack s' such that $1 < |s'| \leq d+1$. Let $\pi' \in [\![f(\boldsymbol{in})]\!]$ be a computation which satisfies $s' \in \mathcal{S}_{m'}(\pi')$. We now prove that the consequent of the rule is true for $d+1$, i.e.,

$$mapped(s') \in \mathcal{S}_m(\pi) \ . \tag{3.11}$$

Assume that the bottom frame of s' is $[g'(\boldsymbol{in_g})]$. Consider a call-stack s'_p such that $s' = s'_p \cdot [g'(\boldsymbol{in_g})]$. Assume that the bottom frame of s'_p is $[h'(\boldsymbol{in_h})]$. This implies that $h'(\boldsymbol{in_h})$ directly calls $g'(\boldsymbol{in_g})$ in π', i.e.,

$$\exists \pi'_h \in [\![h'(\boldsymbol{in})]\!].\ \langle g', \boldsymbol{in_g} \rangle \in calls({\pi'_h}^1) \ . \tag{3.12}$$

Thus the premise of (3.8) holds. Hence, Lemma 3.4.1 implies:

$$\forall \pi_h \in [\![h(\boldsymbol{in_h})]\!].\ (term(\pi_h) \rightarrow \exists g.\ (\langle g, g' \rangle \in map_{\mathcal{F}} \wedge \langle g, \boldsymbol{in_g} \rangle \in calls({\pi_h}^1))) \ . \tag{3.13}$$

Since $|s'_p| \leq d$, the induction hypothesis, (3.10) holds up to d, and therefore, $mapped(s'_p) \in \mathcal{S}_{m'}(\pi')$. Hence, the bottom frame of $mapped(s'_p)$ is

$[h(\boldsymbol{in_h})]$. The subcomputation of π which starts from this call $h(\boldsymbol{in_h})$ is finite owing to $term(\pi)$, also implied by the induction hypothesis. (3.13) implies that in this subcomputation, $h(\boldsymbol{in_h})$ must call $g(\boldsymbol{in_g})$. Hence, $mapped(s_p) \cdot [g(\boldsymbol{in_g})] \in \mathcal{S}(\pi)$. Now note that the call $g'(\boldsymbol{in_g})$ is found in $s' \in \mathcal{S}_{m'}(\pi')$, which implies $g' \in m'$. In combination with $\langle g, g' \rangle \in map_{\mathcal{F}}$, implied by (3.13), the latter means $g \in m$. Consequently, $mapped(s_p) \cdot [g(\boldsymbol{in_g})] \in \mathcal{S}_m(\pi)$ by (3.11).

It is just left to note that $mapped(s_p) \cdot [g(\boldsymbol{in_g})] = mapped(s')$. Thereby, $mapped(s') \in \mathcal{S}_m(\pi)$ holds. □

The rest of our proofs in this chapter rely on the following observations:

$$term(\pi) \leftrightarrow \exists d \in \mathbb{Z}^+. \, \forall s \in \mathcal{S}(\pi). \, |s| \leq d \, , \tag{3.14}$$

and:

$$\forall \pi', s' \in \mathcal{S}(\pi'). \, |s'| = |mapped(s')| \, . \tag{3.15}$$

An immediate consequence of (3.14) is:

$$\neg term(\pi) \leftrightarrow \forall d \in \mathbb{Z}^+. \, \exists s \in \mathcal{S}(\pi). \, |s| > d \, . \tag{3.16}$$

Also observe that for a leaf MSCC m, $\mathcal{S}_m(\pi) \equiv \mathcal{S}(\pi)$.

Lemma 3.4.3. For any given $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$, input $\boldsymbol{in}$, and computations $\pi \in [\![f(\boldsymbol{in})]\!]$, $\pi' \in [\![f'(\boldsymbol{in})]\!]$:

$$\frac{\begin{array}{l}\forall \langle h, h' \rangle \in map_{\mathcal{F}}(m). \; call\text{-}equiv(h^{UF}, {h'}^{UF}) \wedge \\ \forall d \in \mathbb{Z}^+. \, \exists s' \in \mathcal{S}_{m'}(\pi'). \, |s'| > d\end{array}}{\neg term(\pi)} \, . \tag{3.17}$$

Proof. Consider $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$ called with the same argument $\boldsymbol{in}$. Assume that the premise of (3.17) holds for given computations $\pi \in [\![f(\boldsymbol{in})]\!]$ and $\pi' \in [\![f'(\boldsymbol{in})]\!]$. Falsely assume $term(\pi)$. By (3.14) $term(\pi)$ implies that there is some finite value d for which

$$\forall s \in \mathcal{S}_m(\pi). \, |s| \leq d \tag{3.18}$$

holds. On the other hand, the premise of (3.17) guarantees:

$$\exists s' \in \mathcal{S}_{m'}(\pi').\ |s'| > d\ . \tag{3.19}$$

The premise of (3.10) holds. This implies $mapped(s') \in \mathcal{S}_m(\pi)$ by Lemma 3.4.2. (3.15) and (3.19) imply $|mapped(s')| = |s'| > d$, which contradicts (3.18). Hence, the assumption $term(\pi)$ was wrong, i.e., π must be infinite. □

Theorem 3.4.1. (M-TERM) is sound.

Proof. Assume that the premise of rule (M-TERM) holds. Consider $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$ called with the same argument $\boldsymbol{in}$. Falsely assume $\neg m\text{-}term(f, f')$. Without loss of generality, consider a finite computation $\pi \in [\![f(\boldsymbol{in})]\!]$ and an infinite computation $\pi' \in [\![f'(\boldsymbol{in})]\!]$. (3.16) implies:

$$\forall d \in \mathbb{Z}^+.\ \exists s' \in \mathcal{S}_{m'}(\pi').\ |s'| > d\ .$$

The premise of (3.17) now holds. Hence, by Lemma 3.4.3, $\neg term(\pi)$ holds, in contradiction to the assumption that π is finite.

The same argument would hold if we reversed the roles of f and f'. Hence, $m\text{-}term(f, f')$ must hold. □

3.4.1 Proof of (M-TERM$^+$)

We continue towards proving the soundness of (M-TERM$^+$). The following lemma extends (3.16) to cases in which there are mutually-terminating calls outside the MSCC.

Lemma 3.4.4. The following inference rule holds for any computations π and π':

$$\frac{\begin{array}{l}\forall \langle h, h' \rangle \in map_{\mathcal{F}}(m).\ call\text{-}equiv(h^{UF}, h'^{UF}) \wedge \\ \forall \langle g, g' \rangle \in map_{\mathcal{F}}.\ ((g \in C(m) \wedge g' \in C(m')) \rightarrow m\text{-}term(g, g')) \wedge \\ \exists \langle f, f' \rangle \in map_{\mathcal{F}}(m), \boldsymbol{in}.\ (\pi \in [\![f(\boldsymbol{in})]\!] \wedge term(\pi) \wedge \pi' \in [\![f'(\boldsymbol{in})]\!] \wedge \neg term(\pi'))\end{array}}{\exists s' \in \mathcal{S}_{m'}(\pi').\ \forall d \in \mathbb{Z}^+.\ |s'| > d}\ . \tag{3.20}$$

Note that the premise simply strengthens the premise of rule (M-TERM$^+$) with the third line.

Proof. Assume that given computations π and π' satisfy the premise of the rule. (3.16) implies:

$$\exists s' \in \mathcal{S}(\pi').\ \forall d \in \mathbb{Z}^+.\ |s'| > d\ . \tag{3.21}$$

Proving that s' in (3.21) *must* belong to $\mathcal{S}_{m'}(\pi')$, i.e., $s' \in \mathcal{S}_{m'}(\pi')$, but $s' \notin \mathcal{S}(\pi') \setminus \mathcal{S}_{m'}(\pi')$, amounts to validating the following:

$$\neg\exists \tilde{s}' \in \mathcal{S}(\pi') \setminus \mathcal{S}_{m'}(\pi').\ \forall d \in \mathbb{Z}^+.\ |\tilde{s}'| > d\ . \tag{3.22}$$

Falsely assume that such unbounded call-stack $\tilde{s}'$ exists, i.e., it satisfies

$$\tilde{s}' \in \mathcal{S}(\pi') \setminus \mathcal{S}_{m'}(\pi') \wedge \forall d \in \mathbb{Z}^+.\ |\tilde{s}'| > d\ . \tag{3.23}$$

The call-stack $\tilde{s}'$ consists of two non-intersecting parts: the finite prefix $\tilde{s}'_1$ which consists solely of functions in m' and the unbounded suffix $\tilde{s}'_2$ which consists solely of functions ouside of m'. Assume that the bottom frame of $\tilde{s}'_1$ is $[h'(\boldsymbol{in_h})]$ and that the top frame of $\tilde{s}'_2$ is $[g'(\boldsymbol{in_g})]$. Note that $g' \in C(m')$ and $\tilde{s}'_1 \in \mathcal{S}_{m'}(\pi')$ hold. The latter implies $\mathit{mapped}(\tilde{s}'_1) \in \mathcal{S}_m(\pi)$. The premise of (3.8) holds. Therefore, by Lemma 3.4.1, in π the bottom frame of $\mathit{mapped}(\tilde{s}'_1)$, which is $[h(\boldsymbol{in_h})]$, directly calls $g(\boldsymbol{in_g})$, where $\langle g, g' \rangle \in \mathit{map}_{\mathcal{F}}$ and thus $g \in C(m)$. This call must return because of $\mathit{term}(\pi)$. Hence, $m\text{-}\mathit{term}(g, g')$ implies that the call $g'(\boldsymbol{in_g})$ in π' must return. It is a contradiction to the assumption that $\tilde{s}'$ is an unbounded call-stack (see (3.23)), in which no call returns. Hence, such $\tilde{s}'$ cannot exist. □

Now we can prove the soundness of (M-TERM$^+$).

Theorem 3.4.2. (M-TERM$^+$) is sound.

Proof. Assume that the premise of rule (M-TERM$^+$) holds. Consider $\langle f, f' \rangle \in \mathit{map}_{\mathcal{F}}(m)$ called with the same argument $\boldsymbol{in}$. Falsely assume $\neg m\text{-}\mathit{term}(f, f')$. Without loss of generality, consider a finite computation $\pi \in [\![f(\boldsymbol{in})]\!]$ and an

infinite computation $\pi' \in [\![f'(\boldsymbol{in})]\!]$. Lemma 3.4.4 implies:

$$\forall d \in \mathbb{Z}^+.\ \exists s' \in \mathcal{S}_{m'}(\pi').\ |s'| > d\ .$$

The premise of (3.17) now holds. Hence, by Lemma 3.4.3, $\neg term(\pi)$ holds, in contradiction to the assumption that π is finite.

The same argument would hold if we reversed the roles of f and f'. Hence, $m\text{-}term(f, f')$ must hold. □

Chapter 4

A decomposition algorithm

In this chapter we present an algorithm for proving mutual termination of full programs. As mentioned in Chapter 3, the call graph of a program can be viewed as a DAG where the nodes correspond to MSCCs. After building a mapping between the MSCCs of the two call graphs, the algorithm traverses the DAG bottom-up. For each mapped pair of MSCCs m, m', it attempts to prove the mutual termination of their mapped functions, based on (M-TERM$^+$).

The algorithm is inspired by a similar algorithm for verification of *partial equivalence*, which is described in a technical report [22]. The algorithm here is more involved, however, because it handles differently cases in which the checked functions are also partially equivalent (recall that this information, i.e., which functions are known to be partially equivalent, is part of the input to the algorithm). Furthermore, the algorithm in [22] is described with a non-deterministic step, and here we suggest a method for determinizing it.

The preprocessing and mapping are as detailed in Sect. 2.1. Hence the program is loop-free, globals accessed by a function are sent instead as additional inputs, and there is a (possibly partial) mapping $map_{\mathcal{F}}$ between the functions of P and P'.

4.1 The algorithm

Table 4.1 contains the labels we use in the decomposition algorithm (Alg. 1).

Predicate label	What is labeled	Meaning
m_term	$\langle f, f' \rangle \in map_{\mathcal{F}}$	The mapped functions f and f' are mutually terminating
part_eq	$\langle f, f' \rangle \in map_{\mathcal{F}}$	The mapped functions f and f' are partially equivalent
covered	$\langle m, m' \rangle \in map_{\mathcal{M}}$	The MSCC pair has been processed.

Table 4.1: Predicate labels used in the decomposition algorithm.

The input to Alg. 1 consists of P, P', a (possibly partial) mapping $map_{\mathcal{F}}$ between their functions, and (implicitly) those paired functions that are known to be partially equivalent. Its output is a set of function pairs that are marked as *m_term*, indicating that it succeeded to prove their mutual termination based on (M-TERM$^+$). We now describe the three functions used by this algorithm.

ProveMT.

This entrance function traverses the call graphs of P, P' bottom-up, each time focusing on a pair of MSCCs. In line 2 it inlines all non-recursive functions that are not mapped in $map_{\mathcal{F}}$. In line 3 it uses renaming to resolve possible name collisions between the globals of the two input programs. The next line builds the MSCC DAGs MD and MD' from the call graphs, as explained in Sect 3. Line 5 attempts to build $map_{\mathcal{M}}$ (defined in page 13), only that it must be *bijective*. If such a bijective map does not exist, the algorithm aborts. In practice one may run the algorithm bottom-up until reaching non-mapped MSCCs, but we omit this option here for brevity.

The bottom-up traversal starts in line 6. Initially all MSCCs are unmarked. The algorithm searches for a next unmarked pair $\langle m, m' \rangle$ of MSCCs all of whose children pairs are marked. If m, m' are trivial (see page 13 for a definition), then line 10 simply checks the call-equivalence of the function pair $\langle f, f' \rangle$ that constitutes $\langle m, m' \rangle$, and marks them accordingly in line 10. Note that even if the descendants of m, m' are mutually-terminating, m, m' are not necessarily so, because they may

Algorithm 1 Pseudo-code for a bottom-up decomposition algorithm for proving that pairs of functions mutually terminate.

1: **function** ProveMT(Programs P, P', map between functions $map_{\mathcal{F}}$)
2: Inline non-recursive non-mapped functions;
3: Solve name collisions in global identifiers of P, P' by renaming;
4: Generate MSCC DAGs MD, MD' from the call graphs of P, P';
5: If possible, generate a bijective map $map_{\mathcal{M}}$ between the nodes of MD and MD' that is consistent with $map_{\mathcal{F}}$[a]; Otherwise **abort**;
6: **while** $\exists \langle m, m' \rangle \in map_{\mathcal{M}}$ not marked *covered* but its children are, **do**
7: Choose such a pair $\langle m, m' \rangle \in map_{\mathcal{M}}$ and mark it *covered*;
8: **if** m, m' are trivial **then**
9: Let f, f' be the functions in m, m', respectively;
10: **if** CallEquiv (isolate$(f, f', \emptyset)$) **then** mark f, f' as m_term;
11: **else**
12: Select non-deterministically $S \subseteq \{\langle f, f' \rangle \mid \langle f, f' \rangle \in map_{\mathcal{F}}(m)\}$ that intersects every cycle in m and m';
13: **if** $\forall \langle f, f' \rangle \in S$. CallEquiv (isolate$(f, f', S)$) **then**
14: **for each** $\langle f, f' \rangle \in S$ **do** mark $\langle f, f' \rangle$ as m_term;
15: **else** mark the ancestors of m, m' as *covered*;

16: **function** isolate(functions f, f', function pairs S) ▷ Builds f^{UF}, f'^{UF}
17: **for each** $\{\langle g, g' \rangle \in map_{\mathcal{F}} \mid g, g'$ are reachable from $f, f'\}$ **do**
18: **if** $\langle g, g' \rangle \in S$ or $\langle g, g' \rangle$ is marked m_term **then**
19: Replace calls to $g(expr_{in})$ with calls to uf(g, $expr_{in}$);
20: Replace calls to $g'(expr_{in'})$ with calls to uf'(g', $expr_{in'}$);
21: **else** inline g, g' in their callers;
22: **return** $\langle f, f' \rangle$;

23: **function** CallEquiv(A pair of isolated functions $\langle f^{UF}, f'^{UF} \rangle$)
24: Let δ denote the program:

▷ here add the definitions of uf() and uf'() (see Fig. 4.1).
$\boldsymbol{in} := nondet(); f^{UF}(\boldsymbol{in}); f'^{UF}(\boldsymbol{in});$
for each $\{\langle g, g' \rangle \in map_{\mathcal{F}} \mid g \in callees(f) \vee g' \in callees(f')\}$ **do**
assert$(args[g] \subseteq args[g'])$;

25: **return** CBMC(δ);

[a] It is desirable but not necessary to add pairs of trivial nodes to $map_{\mathcal{M}}$.

call their descendants with different parameters. Also note that if this check fails, we continue to check their ancestors (in contrast to the case of non-trivial MSCCs, listed next). The reason is that even if $\langle f,\ f'\rangle$ are not mutually terminating for every input, their callers may still be, because they can be mutually terminating in the context of their callers. We can check this by inlining them, which is only possible because they are not recursive.

Next, consider the case that the selected m, m' in line 7 are not trivial. In line 12 the algorithm chooses non-deterministically a subset S of pairs from $map_{\mathcal{F}}(m)$ that intersects all the cycles in m and m'. In graph-theoretic terms, the functions in S constitute a *feedback vertex set* [15] of both m and m'. This guarantees that we can always inline the functions in m, m' that are not in S. Determinization of this step will be considered in Sect. 4.2. If CallEquiv returns TRUE for all the function pairs in S, then all those pairs are labeled as m_term in line 14. Otherwise it abandons the attempt to prove their ancestors in line 15, by marking them as $mscc\_covered$ in line 15: it cannot prove that mapped functions in $\langle m, m'\rangle$ are mutually terminating, nor can it inline these functions in their callers, so we cannot check all its ancestors.

Regardless of whether $\langle m,\ m'\rangle$ are trivial, they get marked as $mscc\_covered$ in line 7, and the loop in ProveMT continues to another pair.

ISOLATE.

The function ISOLATE receives as input a pair $\langle f, f'\rangle \in map_{\mathcal{F}}$ and a set S of paired functions which, by construction (see line 12), contains only pairs from the same MSCCs as f, f', i.e., if $f \in m$ and $f' \in m'$, then $\langle g, g'\rangle \in S$ implies that $g \in m$ and $g' \in m'$. As output, it generates f^{UF} and f'^{UF}, or rather a relaxation thereof as explained at the bottom of page 11. We will occasionally refer to them as *side 0* and *side 1*. These functions do not have function calls (other than to uninterpreted

```
1: function UF(function index g, input parameters in)          ▷ Called in side 0
2:     if in ∈ params[g] then return the output of the earlier call UF(g, in);
3:     params[g] := params[g] ⋃ in;
4:     return a non-deterministic output;

5: function UF'(function index g', input parameters in')       ▷ Called in side 1
6:     if in' ∈ params[g'] then return the output of the earlier call UF'(g', in');
7:     params[g'] := params[g'] ⋃ in';
8:     if in' ∈ params[g] then                                  ▷ ⟨g, g'⟩ ∈ map_F
9:         if ⟨g, g'⟩ is marked part_eq then
10:            return the output of the earlier call UF(g, in');
11:        return a non-deterministic output;
12:    assert(0);                  ▷ Not call-equivalent: params[g'] ⊈ params[g]
```

Figure 4.1: Functions UF and UF' emulate uninterpreted functions if instantiated with functions that are mapped to one another. They are part of the generated program δ, as shown in CallEquiv of Alg. 1. These functions also contain code for recording the parameters with which they are called.

functions, see line 19), but may include inlined (non-recursive) callees that were not proven to be mutually terminating (see line 21). ISOLATE should be thought of as working on a new copy of the original programs in each invocation[1].

The implementations of UF and UF' appear in Fig. 4.1, and are rather self-explanatory. Their main role is to check call-equivalence. This is done by checking that they are called with the same set of inputs. When $\langle g, g' \rangle$ is marked *part_eq*, UF and UF' emulate the *same* uninterpreted function (i.e., to return the same output given the same input),

[1]There is some redundancy in the listing of this algorithm, as demonstrated by the following case: f calls g, and g calls h (assume that g is the only function to call h), where h is inlined in line 21, but then g is replaced with an uninterpreted function in line 19, which makes the former inlining redundant. Our implementation avoids such cases by reaching the same result by only considering functions for inlining if their callers have already been dealt-with.

formally:

$$\forall \boldsymbol{in}.\ \text{UF}(g, \boldsymbol{in}) = \text{UF'}(g', \boldsymbol{in})\ .$$

When $\langle g, g' \rangle$ is not marked *part_eq*, UF and UF' emulate two *different* uninterpreted functions.

CallEquiv.

Our implementation is based on the C model checker CBMC [9], which enables us to fully automate the check for call-equivalence. CBMC is complete for bounded programs (i.e., loops and recursions are bounded), and, indeed, the program δ we build in CallEquiv is of that nature. It simply calls f^{UF}, f'^{UF} (which, recall, have neither loops nor function calls by construction), with the same non-deterministic value, and asserts in the end that the set of calls in f is included in the set of calls in f' (the other direction is checked in lines 8, 12 of UF'). Examples of such generated programs that we checked with CBMC are available online in [1].

4.1.1 Examples

The following example demonstrates how Alg. 1 works.

Example 4.1.1. Consider the call graphs in Fig. 4.2.

Assume that $\langle f_i, f'_i \rangle \in map_{\mathcal{F}}$ for $i = 1, \ldots, 5$, and that the functions represented by gray nodes are known to be partially equivalent to their counterparts. Line 4 generates the following nodes of the MSCC DAGs (listed bottom-up, left-to-right):

- $MD = \{\{f_5\}, \{f_3\}, \{f_2, f_4\}, \{f_1\}\}$;
- $MD' = \{\{f'_5\}, \{f'_3\}, \{f'_2, f'_4, f'_6\}, \{f'_1\}\}$.

The MSCC mapping $map_{\mathcal{M}}$ in line 5 is naturally derived from $map_{\mathcal{F}}$:

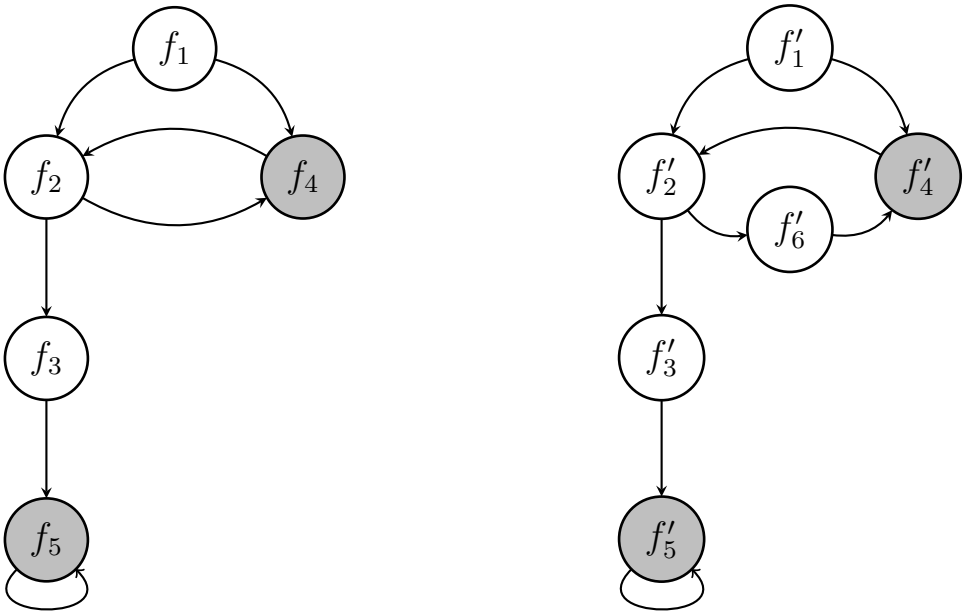

Figure 4.2: Call graphs of the programs discussed in Ex. 4.1.1. Partially equivalent functions are marked gray.

$$map_{\mathcal{M}} = \left\{ \begin{array}{l} \langle \{f_5\}, \{f'_5\} \rangle, \\ \langle \{f_3\}, \{f'_3\} \rangle, \\ \langle \{f_2, f_4\}, \{f'_2, f'_4, f'_6\} \rangle, \\ \langle \{f_1\}, \{f'_1\} \rangle \end{array} \right\}$$

The only leaf MSCC pair $\langle \{f_5\}, \{f'_5\} \rangle$ is chosen in line 7. It is shown in Fig. 4.3. This is a case of simple recursion. In line 12 the only possible

Figure 4.3: A mapped pair of MSCCs. Each one consists of a simple recursive function.

Figure 4.4: Call graphs of the isolated versions of f_5 and f'_5. UF_{f_5} and $UF_{f'_5}$, which have replaced the calls to f_5 and f'_5, respectively, emulate the same uninterpreted functions and are marked gray.

S is $\langle f_5, f'_5 \rangle$. ISOLATE replaces all recursive calls to $f_5(\boldsymbol{in})$, $f'_5(\boldsymbol{in'})$ with UF(f_5, $\boldsymbol{in}$), UF'(f'_5, $\boldsymbol{in'}$), respectively, which emulate the same uninterpreted functions. The constructed pair of isolated functions $\langle f_5{}^{UF}, f'_5{}^{UF} \rangle$ is shown in Fig. 4.4. Assume that CALLEQUIV returns TRUE. Line 14 marks $\langle f_5, f'_5 \rangle$ m_term.

The next available MSCC pair chosen in line 7 is $\langle \{f_3\}, \{f'_3\} \rangle$. This is a case of a pair of trivial MSCCs, which is handled in lines 8–10. ISOLATE replaces all calls to $f_5(\boldsymbol{in})$ and $f'_5(\boldsymbol{in'})$ with calls to UF(f_5, $\boldsymbol{in}$) and UF'(f'_5, $\boldsymbol{in'}$), respectively, which emulate the same uninterpreted functions. The constructed pair of isolated functions $\langle f_3{}^{UF}, f'_3{}^{UF} \rangle$ is shown in Fig. 4.5. Assume that CALLEQUIV returns FALSE. The algorithm cannot assign label m_term to $\langle f_5, f'_5 \rangle$ in line 10, i.e., it cannot prove their mutual termination in a general context. However, since this is a pair of trivial MSCCs, the algorithm can try to establish their mutual termination under the context of their calling functions.

So the algorithm proceeds to the upper MSCC pair $\langle \{f_2, f_4\}, \{f'_2, f'_4, f'_6\} \rangle$. This is a case of mutually recursive functions. Assume that line 12 chooses $S = \{\langle f_2, f'_2 \rangle, \langle f_4, f'_4 \rangle\}$. The algorithm is to check CALLEQUIV (ISOLATE(f_2, f'_2, S)) and CALLEQUIV (ISOLATE(f_4, f'_4, S)) in line 13. Assume that first it checks call-equivalence for $\langle f_2, f'_2 \rangle$.

ISOLATE(f_2, f'_2, S) inlines calls to f_3 in f_2, and then it replaces calls to f_4 and f_5 with calls to UF (distinguished as calls to $UF_{f'_4}$ and $UF_{f'_5}$, respectively,

Figure 4.5: Call graphs of the isolated versions of f_5 and f_5'. Partially equivalent UF_{f_5} and $UF_{f_5'}$, which have replaced the calls to f_5 and f_5', respectively, are marked gray.

Figure 4.6: Call graphs of the isolated versions of f_2 and f_2'. Partially equivalent UF, UF', which respectively replace calls to f_5, f_5' and f_4, f_4', are distinguished as UF_{f_5},$UF_{f_5'}$ and UF_{f_4}, $UF_{f_4'}$, respectively, for better understanding.

in Fig. 4.6 for better understanding). In f_2' calls to f_3' and f_6' are inlined, and then calls to f_4' and f_5' are replaced with calls to UF'. Each replacing uninterpreted function pair emulates a pair of the same functions. These replacements are shown in Fig. 4.6. Assume that CallEquiv returns TRUE. However, the algorithm cannot yet mark $\langle f_2, f_2' \rangle$ with label m_term. It needs yet to check call-equivalence for $\langle f_4, f_4' \rangle$.

Fig. 4.7 shows the function-call replacements made by ISOLATE(f_4, f_4', S). It has no calls to inline, but there are calls $f_2(\boldsymbol{in})$ in f_4 which are replaced with calls to UF(f_2, $\boldsymbol{in}$). Similarly, calls to $f_2'(\boldsymbol{in'})$ in f_4' are replaced with

Figure 4.7: Call graphs of the isolated versions of f_4 and f'_4. UF and UF', which replace calls to f_2 and f'_2, respectively, are distinguished with UF_{f_2} and $UF_{f'_2}$, respectively, for better understanding. They emulate different uninterpreted functions.

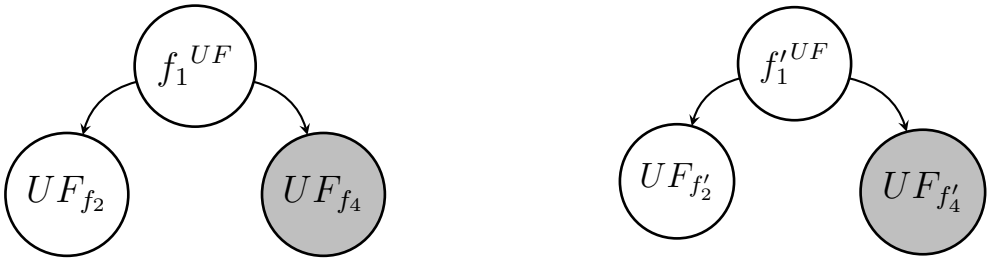

Figure 4.8: Call graphs of the isolated versions of f_1 and f'_1. Two different uninterpreted functions UF and UF', which replace calls to f_2 and f'_2, respectively, are distinguished as UF_{f_2} and $UF_{f'_2}$, respectively, for better understanding. The same uninterpreted functions UF and UF', which replace calls to f_4 and f'_4, respectively, are distinguished as UF_{f_4} and $UF_{f'_4}$, respectively.

calls to UF'(f'_2, $\boldsymbol{in'}$). Note that in this case UF and UF' emulate different uninterpreted functions. Assume CallEquiv returns TRUE. Line 14 of the algorithm marks both $\langle f_2, f'_2\rangle$ and $\langle f_4, f'_4\rangle$ with label m_term.

The last MSCCs pair to check is $\langle\{f_1\}, \{f'_1\}\rangle$. This is again a case of a pair of trivial MSCCs. ISOLATE replaces all calls to $f_2(\boldsymbol{in})$ and $f'_2(\boldsymbol{in'})$ with calls to UF(f_2, $\boldsymbol{in}$) and UF'(f'_2, $\boldsymbol{in'}$), respectively, which emulate two different uninterpreted functions. Calls to $f_4(\boldsymbol{in})$ and $f'_4(\boldsymbol{in'})$ are replaced with calls to UF(f_4, $\boldsymbol{in}$) and UF'(f'_4, $\boldsymbol{in'}$), respectively, which emulate the same uninterpreted functions. The constructed pair of isolated functions $\langle f_3{}^{UF}, f'_3{}^{UF}\rangle$ is shown in Fig. 4.8. Assume that CallEquiv returns TRUE. The algorithm marks $\langle f_5, f'_5\rangle$ m_term in line 10, i.e., it has proven their

MSCCs	Checked functions	Description	Res.
$\{f_5\}$, $\{f_5'\}$	$\langle f_5, f_5'\rangle$	In line 12 the only possible S is $\langle f_5, f_5'\rangle$. ISOLATE replaces the recursive call to f_5, f_5' with UF, UF', respectively ($^=$). Assume CALLEQUIV returns TRUE. $\langle f_5, f_5'\rangle$ is marked m_term in line 14.	✓
$\{f_3\}$, $\{f_3'\}$	$\langle f_3, f_3'\rangle$	This is a case of trivial MSCCs, which is handled in lines 8–10. ISOLATE replaces the calls to f_5, f_5' with UF, UF', respectively ($^=$). Assume CALLEQUIV returns FALSE.	✗
$\{f_2, f_4\}$, $\{f_2', f_4', f_6'\}$		In line 12 let $S = \{\langle f_2, f_2'\rangle, \langle f_4, f_4'\rangle\}$.	
	$\langle f_2, f_2'\rangle$	In f_2 calls to f_3 are inlined, and calls to f_4, f_5 are replaced with calls to UF. In f_2' calls to f_3', f_6' are inlined, and calls to f_4', f_5' are replaced with calls to UF' ($^=$). Assume CALLEQUIV returns TRUE.	$✓^c$
	$\langle f_4, f_4'\rangle$	In f_4, f_4' calls to f_2, f_2' are respectively replaced with calls to UF, UF' ($^\neq$). Assume CALLEQUIV returns TRUE. Now $\langle f_2, f_2'\rangle$ and $\langle f_4, f_4'\rangle$ are marked m_term in line 14.	✓
$\{f_1\}$, $\{f_1'\}$	$\langle f_1, f_1'\rangle$	Again, a case of a trivial MSCC. Calls to f_2, f_2' are respectively replaced with UF, UF' ($^\neq$). Calls to f_4, f_4' are replaced with UF, UF' ($^=$), respectively. Assume CALLEQUIV returns TRUE. $\langle \{f_1\}, \{f_1'\}\rangle$ is marked m_term.	✓

Table 4.2: Applying Alg. 1 to the call graphs in Fig. 4.2 under the assumptions made in Ex. 4.1.1 about the results of CALLEQUIV. The following notations are used in the table:
'✓' means that the pair is marked m_term,
'$✓^c$' means that it is marked conditionally (it becomes unconditional once all other pairs in S are marked as well), and
'✗' means that it is not marked m_term;
'($^=$)' denotes that UF and UF' emulate the same uninterpreted functions, while
'($^\neq$)' denotes that they emulate different uninterpreted functions.

Figure 4.9: Call graphs of the isolated versions of f_4 and f_4' when $\langle f_2,\ f_2'\rangle \notin S$. The same UF, UF', which respectively replace calls to f_5, f_5' and f_4, f_4', are distinguished as UF_{f_5}, $UF_{f_5'}$ and UF_{f_4}, $UF_{f_4'}$, respectively, for better understanding.

mutual termination in a general context.

The output of the algorithm, based on the aforementioned assumptions about the results of the checks for call equivalence, is that the following pairs of functions are marked as *m_term*: $\langle f_1, f_1'\rangle$, $\langle f_2, f_2'\rangle$, $\langle f_4, f_4'\rangle$, and $\langle f_5, f_5'\rangle$. Since functions f_1 and f_1' are main in the compared programs, those whole programs are mutually terminating. The overall progress of the algorithm in this example is summarized in Table 4.2.

□

In the previous example only maximal feedback vertex sets were chosen. The following example demonstrates how the algorithm proceeds when a chosen feedback vertex set S is not maximal.

Example 4.1.2. Consider again the call graphs in Fig. 4.2. Suppose that the algorithm has processed MSCC pairs $\langle\{f_5\},\ \{f_5'\}\rangle$ and $\langle\{f_3\},\ \{f_3'\}\rangle$ as described in the previous example. Now the algorithm is proceeding to MSCC pair $\langle\{f_2, f_4\},\ \{f_2', f_4', f_6'\}\rangle$. Assume that line 12 chooses $S = \{\langle f_4,\ f_4'\rangle\}$ having left $\langle f_2,\ f_2'\rangle$ outside of S. The algorithm is to check CallEquiv (isolate(f_4, f_4', S)) in line 13.

isolate(f_4, f_4', S) inlines calls to f_3 in f_2, inlines calls to f_2 in f_4, and then it replaces calls to f_4, f_5 with calls to UF (distinguished as calls to $UF_{f_4'}$ and $UF_{f_5'}$, respectively, in Fig. 4.9 for better understanding). On side 1, calls

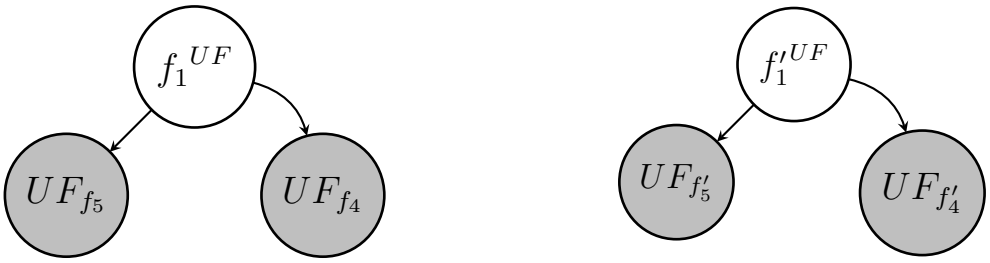

Figure 4.10: Call graphs of the isolated versions of f_1 and f_1'. The same uninterpreted functions UF and UF', which replace calls to f_4 and f_4', respectively, are distinguished as UF_{f_4} and $UF_{f_4'}$, respectively, for better understanding. The same uninterpreted functions UF and UF', which replace calls to f_5 and f_5', respectively, are distinguished as UF_{f_5} and $UF_{f_5'}$, respectively.

to f_3' and f_6' are inlined in f_2', calls to f_2' are inlined in f_4', and then calls to f_4' and f_5' are replaced with calls to UF'. Each replacing uninterpreted function pair emulates a pair of the same functions. These replacements are shown in Fig. 4.9. Assume CallEquiv returns TRUE. Line 14 of the algorithm marks only $\langle f_4,\ f_4' \rangle$ with label *m_term*, while $\langle f_2,\ f_2' \rangle$ remains unmarked. As a result, in the last iteration of the algorithm (for the uppermost MSCC pair), ISOLATE($f_1,\ f_1',\ ...$) will not replace the calls to f_2 (f_2') with calls to UF (UF') because $\langle f_2,\ f_2' \rangle$ is not marked *m_term*. Instead, in line 21, those calls will be inlined. The call graphs of the program generated by ISOLATE are shown in Fig 4.10.

Assume that the check of call equivalence for $\langle f_1,\ f_1' \rangle$ succeeds. Then the output of the algorithm, based on the aforementioned assumptions about the results of the checks for call equivalence, is that the following pairs of functions are marked as *m_term*: $\langle f_1, f_1' \rangle$, $\langle f_4, f_4' \rangle$, and $\langle f_5, f_5' \rangle$. Consequently, those whole programs are mutually terminating. The overall progress of the algorithm in this example is summarized in Table 4.3.

□

MSCCs	Checked functions	Description	Res.
$\{f_5\}$, $\{f'_5\}$	$\langle f_5, f'_5\rangle$	In line 12 the only possible S is $\langle f_5, f'_5\rangle$. ISOLATE replaces the recursive call to f_5, f'_5 with UF, UF', respectively ($^{=}$). Assume CALLEQUIV returns TRUE. $\langle f_5, f'_5\rangle$ is marked *m_term* in line 14.	✓
$\{f_3\}$, $\{f'_3\}$	$\langle f_3, f'_3\rangle$	This is a case of trivial MSCCs, which is handled in lines 8–10. ISOLATE replaces the calls to f_5, f'_5 with UF, UF', respectively ($^{=}$). Assume CALLEQUIV returns FALSE.	✗
$\{f_2, f_4\}$, $\{f'_2, f'_4, f'_6\}$		In line 12 let $S = \{\langle f_4, f'_4\rangle\}$.	
	$\langle f_4, f'_4\rangle$	Calls to f_3 are inlined in f_2, calls to which are inlined in f_4, and calls to f_4, f_5 are replaced with calls to UF. On the other side, calls to f'_3, f'_6 are inlined in f'_2, calls to which are inlined in f'_4, and calls to f'_4, f'_5 are replaced with calls to UF' ($^{=}$). Assume CALLEQUIV returns TRUE.	✓
$\{f_1\}$, $\{f'_1\}$	$\langle f_1, f'_1\rangle$	Again, a case of a trivial MSCC. Calls to f_2, f'_2 are respectively replaced with UF, UF' ($^{\neq}$). Calls to f_4, f'_4 are replaced with UF, UF' ($^{=}$), respectively. Assume CALLEQUIV returns TRUE. $\langle\{f_1\}, \{f'_1\}\rangle$ is marked *m_term*.	✓

Table 4.3: Applying Alg. 1 to the call graphs in Fig. 4.2 under the assumptions made in Ex. 4.1.2 about the results of CALLEQUIV. The following notations are used in the table:
'✓' means that the pair is marked *m_term*,
'✗' means that it is not marked *m_term*;
'($^{=}$)' denotes that UF and UF' emulate the same uninterpreted functions, while
'($^{\neq}$)' denotes that they emulate different uninterpreted functions.

4.2 Choosing a vertex feedback set deterministically

The choice of the set S in line 12 of Alg. 1 is non-deterministic. In RVT, however, we determinized the choice. The deterministic version of the algorithm tries various choices for such sets until it detects a successful one. In the worst case this amounts to trying all sets, which is exponential in the size of the MSCC. Observe, however, that large MSCCs are rare in real programs and, indeed, this has never posed a computational problem in our experiments.

The determinized version of the decomposition algorithm is presented in Alg. 2. The differences versus the non-deterministic version are:

- lines 12–15 of ProveMT of Alg. 1 are replaced in Alg. 2 with a call to function ProveMT_NonTrivial, defined in lines 12–22, and
- line 25 of CallEquiv of Alg. 1 is replaced in Alg. 2 with lines 32–35.

The changes in CallEquiv will be explained in Sect. 4.2.1. Here we describe the function ProveMT_NonTrivial.

ProveMT_NonTrivial.

The loop in lines 14–21 tries various subsets of pairs from $map_{\mathcal{F}}(m)$ that intersect every cycle in m and m' until one is discovered all of whose function pairs are found call-equivalent and thereafter marked as m_term in line 20. If no such subset is discovered, the ancestors of the current MSCC pair $\langle m, m' \rangle$ are doomed (line 22).

The objective is that a *maximal* set S of function pairs would be tried each time, because the larger the set is, the more functions are declared to be mutually terminating in case of success. Further, larger sets imply fewer functions to inline, and, hence, the burden on CallEquiv is expected to be smaller. Line 14 of Alg. 2 delegates the optimization problem of finding such a maximal set to function ChooseS, detailed later in this section. Since ChooseS needs to know for which function pairs the recent call-equivalence checks have failed, every such function

Algorithm 2 Determinization of Alg. 1.

1: **function** ProveMT(Programs P, P'; map between functions $map_{\mathcal{F}}$)
2: Inline non-recursive non-mapped functions;
3: Solve name collisions in global identifiers of P, P' by renaming.
4: Generate MSCC DAGs MD, MD' from the call graphs of P, P';
5: If possible, generate a bijective map $map_{\mathcal{M}}$ between the nodes of MD and MD' that is consistent with $map_{\mathcal{F}}$; Otherwise **abort**.
6: **while** $\exists \langle m, m' \rangle \in map_{\mathcal{M}}$ not marked *covered* but its children are, **do**
7: Choose such a pair $\langle m, m' \rangle \in map_{\mathcal{M}}$ and mark it *covered*;
8: **if** m, m' are trivial **then**
9: Let f, f' be the functions in m, m', respectively;
10: **if** CallEquiv (isolate$(f, f', \emptyset)$) **then** mark $\langle f, f' \rangle$ as m_term;
11: **else** ProveMT_NonTrivial($\langle m, m' \rangle$);

12: **function** ProveMT_NonTrivial(A pair of MSCCs $\langle m, m' \rangle$)
13: $failed\_pairs := S := \emptyset$;
14: **while** $(S :=$ ChooseS$(\langle m, m' \rangle, failed_pairs, S)) \neq \emptyset$ **do**
15: $failed\_pairs := \emptyset$;
16: **for each** $\langle f, f' \rangle \in S$ **do**
17: **if** $\neg$CallEquiv (isolate(f, f', S)) **then**
18: $failed\_pairs := failed\_pairs \cup \{\langle f, f' \rangle\}$;
19: **if** $failed\_pairs = \emptyset$ **then** ▷ Every check has succeeded
20: **for each** $\langle f, f' \rangle \in S$ **do** mark $\langle f, f' \rangle$ as m_term;
21: **return** ;
22: mark the ancestors of m, m' as *covered*;

23: **function** isolate(Functions f, f'; function pairs S) ▷ Builds f^{UF}, f'^{UF}
24: **for each** $\{\langle g, g' \rangle \in map_{\mathcal{F}} \mid g, g'$ are reachable from $f, f'\}$ **do**
25: **if** $\langle g, g' \rangle \in S$ or $\langle g, g' \rangle$ is marked m_term **then**
26: Replace calls to $g(expr_{in})$ with calls to uf(g, $expr_{in}$);
27: Replace calls to $g'(expr_{in'})$ with calls to uf'(g', $expr_{in'}$);
28: **else** inline g, g' in their callers;
29: **return** $\langle f, f' \rangle$;

30: **function** CallEquiv(A pair of isolated functions $\langle f^{UF}, f'^{UF} \rangle$)
31: Let δ denote the program:
▷ here add the definitions of uf() and uf'() (see Fig. 4.1).
$\boldsymbol{in} := nondet()$;
$f^{UF}(\boldsymbol{in})$;
$f'^{UF}(\boldsymbol{in})$;
for each $\{\langle g, g' \rangle \in map_{\mathcal{F}} \mid g \in callees(f) \vee g' \in callees(f')\}$ **do**
assert$(params[g] \subseteq params[g'])$;
32: **if** WasProven($proven$, $\langle f^{UF}, f'^{UF} \rangle$) **or**[a] CBMC($\delta$) **then**
33: $proven := proven \cup \{\langle f^{UF}, f'^{UF} \rangle\}$;
34: **return** true;
35: **return** false

[a] The second condition is evaluated only if the first condition has been evaluated as false.

Algorithm 3 Pseudo-code for function ChooseS, which finds a feedback vertex set over a given pair of MSSCs while blocking previously failed solutions.

1: **function** ChooseS(A pair of MSSCs: $\langle m, m' \rangle$,
sets of function pairs: $failed\_pairs$, S_{failed})
▷ $failed\_pairs \subseteq S_{failed}$

2: **if** $failed\_pairs = \emptyset$ **then**

3: $\sigma := \emptyset$;

4: **else**

5: $\sigma := \sigma \cup \{\{f \mid \langle f, f' \rangle \in S_{failed}\}\}$;

6: Let β denote the pseudo-Boolean formula:

$$\text{maximize } S\text{:} \quad \max \sum_{g \in mapped(\mathrm{m})} v_g \ ,$$

where $mapped(m)$ denotes the set of functions in m that are mapped in $map_{\mathcal{F}}$, v_g is a Boolean variable associated with a function node g in $M = m \cup m'$, subject to the following constraints for M:

i.	Banning unmapped nodes from S:	$\forall g \in (M \setminus mapped(M)).\ \neg v_g$
ii.	Defining the edges:	$\forall \{f, g \mid (f, g)$ is an edge in $M\}.$ $\neg v_f \wedge \neg v_g \rightarrow\ e_{fg}$
iii.	Transitive closure of edges:	$\forall f, g, h \in M.\ e_{fg} \wedge e_{gh} \rightarrow e_{fh}$
iv.	Forbidding self loops:	$\forall g \in M.\ \neg e_{gg}$
v.	Enforcing mapping $map_{\mathcal{F}}$:	$\forall \langle g, g' \rangle \in map_{\mathcal{F}}(m).\ v_g \leftrightarrow v_{g'}$
vi.	Blocking the failed solutions:	$\bigwedge_{s_i \in \sigma} (\bigvee_{g \in s_i} \neg v_g)$;

7: $s :=$ MiniSat$^{+}(\beta)$; ▷ MiniSat returns $\emptyset$ if β is unsatisfiable

8: **return** $\{\langle f, f' \rangle \mid v_f \in s \wedge \langle f, f' \rangle \in map_{\mathcal{F}}\}$

pair (for which the call-equivalence check has failed) is recorded into a set $failed\_pairs$ in line 18 of Alg. 2.

ChooseS.

Function ChooseS, presented in Alg. 3, solves the above optimization problem for a given pair of mapped MSSCs $\langle m, m' \rangle$ via a reduction to a 0-1 ILP problem (equivalently, a pseudo-Boolean formula β). Each function node g (g') in m (m') is associated with a Boolean variable v_g ($v_{g'}$) in that formula, indicating whether it is a part of S. The objective is thus to maximize the sum of those variables that are mapped (those that are unmapped cannot be in S anyway). In addition, there is a variable for each edge in $m \cup m'$ (e.g., e_{gh} for edge (g, h)), which is set to true if and only if neither of the vertices of the edge is a function in S. By enforcing a transitive closure, we guarantee that if there is a cycle of edges set to true (i.e., a cycle in which none of the nodes is in S), then the self edges (e.g., e_{gg}) are set to TRUE as well. We then prevent such cycles by setting them to FALSE. The problem formulation appears in line 6 of Alg. 3, and is rather self-explanatory. The generated pseudo-Boolean formula β is then solved by MINISAT+ [13] (see line 7). Once a solution s to this problem is found, the last line returns the set $S = \{\langle f, f' \rangle \mid v_f \in s, \langle f, f' \rangle \in map_{\mathcal{F}}\}$.

If the returned set fails (i.e., one of the pairs in it cannot be proven to be call-equivalent), ChooseS will be called again for the current MSCC pair $\langle m, m' \rangle$. From now on, by *left side* of a set of pairs we mean the set of the elements found in the left side of the pairs. In that repeated call, in line 5 the left side of the failed set S_{failed} will be added to σ. Thus S_{failed} will be removed from the solution space owing to constraint #vi in line 6 of Alg. 3. When there are no more solutions available, the repeated call will return an empty set.

Note that argument $failed\_pairs$ is unused in the current version of ChooseS, but its presence in the function interface enables compatibility with the final version of ChooseS, which will be presented in Sect. 4.2.2.

We have implemented three optimizations in RVT. The rest of this chap-

ter will elaborate on them.

4.2.1 Recycling proofs

The first optimization that we have implemented is the following. Searching for a vertex feedback set as explained above amounts to solving an optimization problem for various subsets of vertices. We may save some effort by analyzing the cause of failure. Specifically, when failing with a set S_0, we save the strict subset $s_0 \subset S_0$ for which we were able to prove call-equivalence. Let S_1 be a new set under consideration and let $\langle g, g' \rangle \in S_1 \cap s_0$. Denote by f_S^{UF} the function f^{UF} as constructed given the set S. Then the positive result of g can be reused if $g_{S_0}^{UF}$ is equally or more abstract than $g_{S_1}^{UF}$. This condition holds if none of the functions inlined in $g_{S_0}^{UF}$ are abstracted in $g_{S_1}^{UF}$.

The positive results are accumulated in the set *proven* in line 33 of Alg. 2. Searching *proven* for a previous proof reusable for two given abstracted functions f^{UF}, f'^{UF} is executed in a Boolean function WasProven, whose pseudo-code is not explicitly listed. WasProven is invoked in line 32 and returns TRUE if it could find isolated versions of functions f, f' which are equally or more abstract than f^{UF} and f'^{UF}, respectively. In this case a heavy operation of checking the generated program δ with CBMC is skipped. Only when nothing can be recycled, CBMC is involved (line 32). An example of a recycled proof will be given in Ex. 4.2.1 (at the end of the next section).

4.2.2 Optimizing function ChooseS

The two additional optimizations we have implemented concern the process of function ChooseS. Alg. 4 presents an optimized version of this function. Note that its lines 1–4, 6, 8, and 9 are syntactically equal to lines 1–4, 5, 7, and 8, respectively, in Alg. 3. Here is the description of the two optimizations:

- *Generalizing counterexamples*:
  Recall that constraint #vi (see line 6 of Alg. 3) blocks failed solutions. In some cases it is possible to generalize the failed solution, which expedites convergence. Let S_{failed} denote a set with which the mutual-termination check has failed. It is frequently possible to find a strict

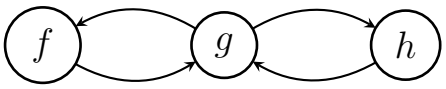

Figure 4.11: An example of MSCC where a counterexample may be generalized.

subset of S_{failed} which is sufficient to make the proof fail. Specifically, such a subset can consist of the set $failed\_pairs$ of the failing pair nodes in S_{failed} plus nodes in S_{failed} that can be reached from $failed\_pairs$ not through any other node in S_{failed}. As an example, consider the MSCC $\{f, g, h\}$ as shown in Fig. 4.11, and $S_{failed} = \{\langle f, f' \rangle, \langle g, g' \rangle, \langle h, h' \rangle\}$ such that f', g', h' are mapped with f, g, h, respectively. If $\langle f, f' \rangle$ fails, then we must remove from S_{failed} either $\langle f, f' \rangle$ or $\langle g, g' \rangle$, regardless of what we do with $\langle h, h' \rangle$. Hence, here we can regard the subset $\{\langle f, f' \rangle, \langle g, g' \rangle\}$ as the failing set, rather than S_{failed} itself, which strengthens constraint #vi. In other words, adding the negation of this subset removes from the solution space sets that are bound to fail.

Counterexamples are generalized inside function GeneralizeCounterexample, which is not explicitly listed in the pseudo-code of Alg. 4. The arguments of ChooseS needed for this purpose are:

- S_{failed}, and
- $failed\_pairs$, which is the set of function pairs whose call-equivalence checks have failed upon S_{failed} (it receives the value of the local variable $failed\_pairs$ of the function ProveMT_NonTrivial of Alg. 2).

GeneralizeCounterexample is called in line 5 of Alg. 4. The left side of its output, i.e., the left side of either the generalized counterexample or the originally failing S_{failed} if no counterexample could be generalized, is added to σ (line 6); then the output is removed from the solution space owing to constraint #vi.

- *Higher priority for partially-equivalent pairs*:
  Among equal-sized sets, it is better to give priority to sets that include

Algorithm 4 An optimized version of function ChooseS presented in Alg. 3.

1: **function** ChooseS(A pair of MSSCs: $\langle m, m' \rangle$,
sets of function pairs: $failed\_pairs$, S_{failed})
▷ $failed\_pairs \subseteq S_{failed}$

2: **if** $failed\_pairs = \emptyset$ **then**

3: $\sigma := \emptyset$;

4: **else**

5: $S_{failed} :=$ GeneralizeCounterexample($failed\_pairs$, S_{failed});

6: $\sigma := \sigma \cup \{\{f \mid \langle f, f' \rangle \in S_{failed}\}\}$;

7: Let β denote the pseudo-Boolean formula:

maximize S: max $\sum_{g \in mapped(\mathrm{m})} v_g \cdot w_g$,

where $mapped(m)$ denotes the set of functions in m that are mapped in $map_{\mathcal{F}}$, v_g is a Boolean variable associated with a function node g in $M = m \cup m'$, and $w_g = \begin{cases} |m| + 1 & \text{if } \langle g, g' \rangle \text{ is marked as } part\_eq \text{ s.t. } \langle g, g' \rangle \in map_{\mathcal{F}}, \\ |m| & \text{otherwise,} \end{cases}$
subject to the following constraints for M:

i.	Banning unmapped nodes from S:	$\forall g \in (M \setminus mapped(M)).\ \neg v_g$
ii.	Defining the edges:	$\forall \{f, g \mid (f, g) \text{ is an edge in } M\}.$ $\neg v_f \wedge \neg v_g \rightarrow\ e_{fg}$
iii.	Transitive closure of edges:	$\forall f, g, h \in M.\ e_{fg} \wedge e_{gh} \rightarrow e_{fh}$
iv.	Forbidding self loops:	$\forall g \in M.\ \neg e_{gg}$
v.	Enforcing mapping $map_{\mathcal{F}}$:	$\forall \langle g, g' \rangle \in map_{\mathcal{F}}(m).\ v_g \leftrightarrow v_{g'}$
vi.	Blocking the failed solutions:	$\bigwedge_{s_i \in \sigma} (\bigvee_{g \in s_i} \neg v_g)$;

8: $s :=$ MiniSat$^+(\beta)$; ▷ MiniSat$^+$ returns $\emptyset$ if β is unsatisfiable

9: **return** $\{\langle f, f' \rangle \mid v_f \in s \wedge \langle f, f' \rangle \in map_{\mathcal{F}}\}$

a maximal number of pairs that are marked as *part_eq*, because there is a better chance of success with partially-equivalent functions (recall that these are replaced with the same uninterpreted functions). This strategy is implemented by assigning weights in the objective which on one hand give higher priority to partially-equivalent function pairs, and on the other hand still ensure finding the largest set S possible. We will prove in Proposition 4.2.1 that the following weights guarantee these two properties:

- $\star$ $|m|+1$ for functions that are maked as *part_eq*;
- $\star$ $|m|$ for other functions,

where m is the currently considered MSCC in side 0.

Proposition 4.2.1. Function ChooseS in Alg. 4 returns a set with a maximal number of function pairs subject to constraints #i-vi.

Proof. Proving this proposition amounts to validating the following formula for any MSCC m:

$$\forall s, \tilde{s} \subseteq m. \quad \forall_{\substack{1\leq i\leq |s| \\ 1\leq j\leq |\tilde{s}|}} w_i, \tilde{w}_j \in \{|m|, |m|+1\}. \\ |s| > |\tilde{s}| \;\Rightarrow\; \sum_{i=1}^{|s|} w_i > \sum_{j=1}^{|\tilde{s}|} \tilde{w}_j \;. \tag{4.1}$$

Consider any $s, \tilde{s} \subseteq m$ such that s is larger than $\tilde{s}$. For $\tilde{s}$ we have:

$$\forall_{1\leq j\leq |\tilde{s}|}\; \tilde{w}_j \in \{|m|, |m|+1\}.\; \sum_{j=1}^{|\tilde{s}|} \tilde{w}_j \leq |\tilde{s}| \cdot (|m|+1) \;.$$

Since s is larger than $\tilde{s}$, i.e., $|\tilde{s}| \leq |s| - 1$, we deduce:

$$|\tilde{s}| \cdot (|m|+1) \leq (|s|-1) \cdot (|m|+1) \;.$$

But the fact that $|s| \leq |m|$ implies:

$$(|s|-1) \cdot (|m|+1) < |s| \cdot |m| \;,$$

which means:

$$\forall_{1\leq j\leq|\tilde{s}|}\ \tilde{w}_j \in \{|m|, |m|+1\}.\ \sum_{j=1}^{|\tilde{s}|} \tilde{w}_j < |s|\cdot|m|\ .$$

On the other hand, for s we have:

$$\forall_{1\leq i\leq|s|}\ w_i \in \{|m|, |m|+1\}.\ \sum_{i=1}^{|s|} w_i \geq |s|\cdot|m|\ .$$

Hence, the formula in (4.1) is valid. □

The following example will demonstrate all the three described optimizations.

Example 4.2.1. Consider the call graphs in Fig. 4.12. Assume that $\langle f_i, f'_i\rangle \in map_{\mathcal{F}}$ for $i = 0, \ldots, 5, 7, \ldots, 9$, and that the functions represented by gray nodes are known to be partially equivalent to their counterparts. The MSCC mapping $map_{\mathcal{M}}$ in line 5 of Alg. 2 is naturally derived from $map_{\mathcal{F}}$:

$$map_{\mathcal{M}} = \left\{ \begin{array}{l} \langle\{f_5\},\ \{f'_5\}\rangle, \\ \langle\{f_3\},\ \{f'_3\}\rangle, \\ \langle\{f_2, f_4\},\ \{f'_2, f'_4, f'_6\}\rangle, \\ \langle\{f_1\},\ \{f'_1\}\rangle, \\ \langle\{f_7, f_8, f_9\},\ \{f'_7, f'_8, f'_9, f'_{10}\}\rangle, \\ \langle\{f_0\},\ \{f'_0\}\rangle \end{array} \right\}$$

Suppose that Alg. 2 has processed MSCC pairs $\langle\{f_5\},\ \{f'_5\}\rangle$, $\langle\{f_3\},\ \{f'_3\}\rangle$, and $\langle\{f_2, f_4\},\ \{f'_2, f'_4, f'_6\}\rangle$, and, in the same manner as in Ex. 4.1.1, found the following mapped function pairs as mutually terminating: $\langle f_5,\ f'_5\rangle$, $\langle f_2,\ f'_2\rangle$, $\langle f_4,\ f'_4\rangle$, and $\langle f_1,\ f'_1\rangle$. Now the algorithm proceeds with the MSCC pair $\langle\{f_7, f_8, f_9\},\ \{f'_7, f'_8, f'_9, f'_{10}\}\rangle$. When ChooseS listed in Alg. 4 is called for the first time for this pair, its mission amounts to solving the optimization problem given in Fig. 4.13. Since this is the first attempt ($failed\_pairs = \emptyset$), constraint #vi is irrelevant. The solution yields the largest possible set[2]

[2]For simplicity, we will index the feedback vertex sets returned by ChooseS, e.g.,

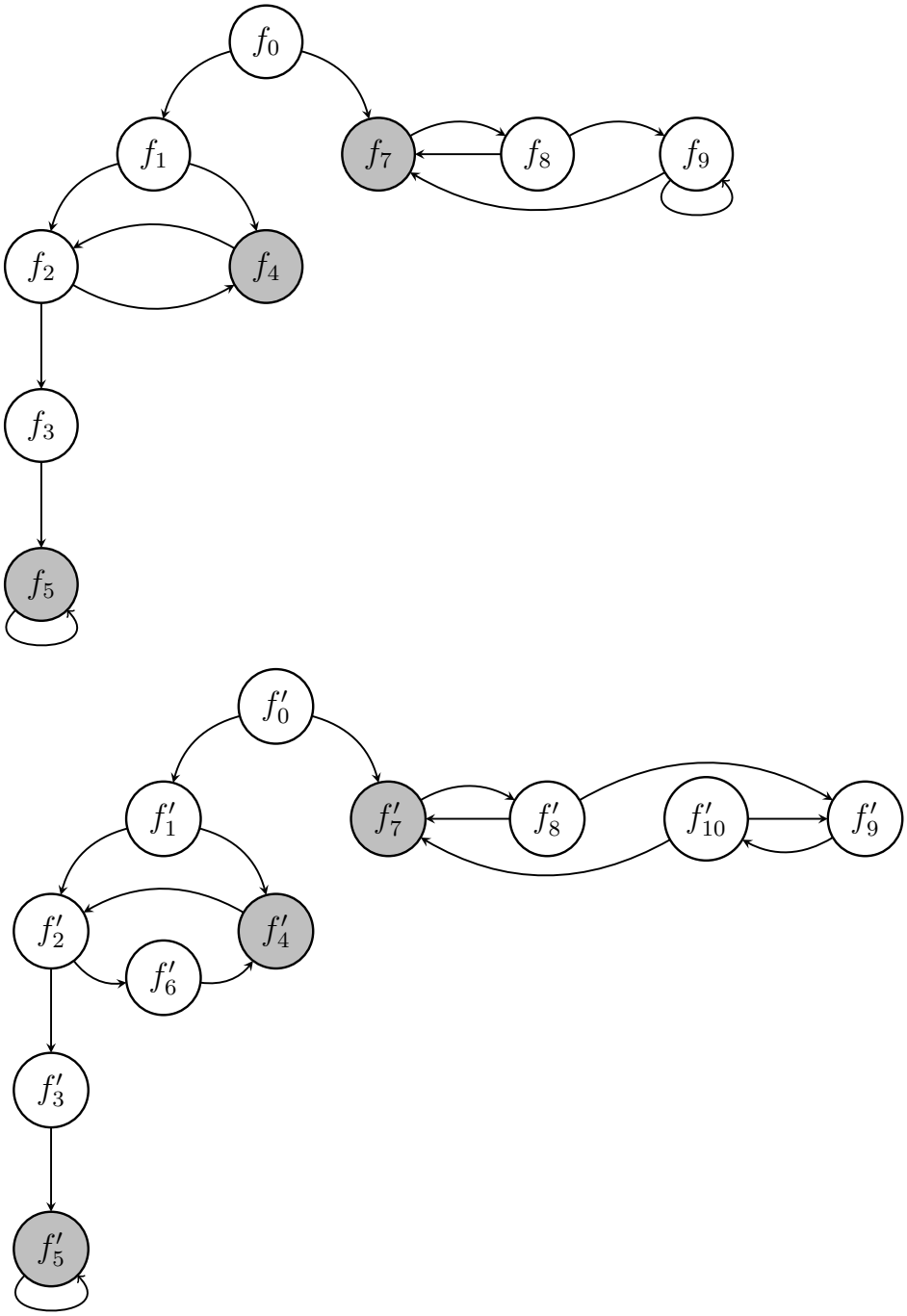

Figure 4.12: Call graphs of the programs discussed in Ex. 4.2.1. Partially equivalent functions are gray.

$S_1 = \{\langle f_7, f'_7\rangle, \langle f_8, f'_8\rangle, \langle f_9, f'_9\rangle\}$ exactly as Alg. 3 would choose if it were running. Assume that CallEquiv (isolate(f_7, f'_7, S_1)) and CallEquiv (isolate(f_8, f'_8, S_1)) return false, while CallEquiv (isolate(f_9, f'_9, S_1)) returns true. This result invalidates S_1. Another feedback vertex set should be sought by ChooseS. Now note that there is a strict subset of S_1 which consists of a failing node pair $\langle f_7, f'_7\rangle$ plus $\langle f_8, f'_8\rangle$, which is the only pair in S_1 reachable from $\langle f_7, f'_7\rangle$ not through any other nodes in S_1. GeneralizeCounterexample reduces S_1 and returns $S_{failed} = \{\langle f_7, f'_7\rangle, \langle f_8, f'_8\rangle\}$. Seeking a new feedback vertex set amounts to solving the same optimization problem given in Fig. 4.13 with an additional constraint for blocking S_{failed}:

vi. Blocking the failed solutions: $(\neg v_{f_7} \vee \neg v_{f_8})$.

If we applied the version of ChooseS defined in Alg. 3, Minisat+ would yield either $\{f_7, f_9\}$ or $\{f_8, f_9\}$. However, its only solution in the optimized version of ChooseS (defined in Alg. 4) is $\{f_7, f_9\}$, because f_7 and f'_7 are partially-equivalent (unlike f_8, f'_8), and, consequently, $\{f_7, f_9\}$ weighs more than $\{f_8, f_9\}$.

For proving the mutual termination of the pairs in $S_2 = \{\langle f_7, f'_7\rangle, \langle f_9, f'_9\rangle\}$, Alg. 2 is to check the call equivalence of those pairs. In CallEquiv (f_9^{UF}, f_9^{UF}), the previous proof of the call equivalence of $\langle f_9^{UF}, f_9'^{UF}\rangle$ is recycled by RecycleProof because $f_{9_{S_1}}^{UF} \equiv f_{9_{S_2}}^{UF}$ and $f'^{UF}_{9_{S_1}} \equiv f'^{UF}_{9_{S_2}}$. Thus CBMC is involved only for verifying the call equivalence of $\langle {f_7}^{UF}, {f'_7}^{UF}\rangle$. Assume that it fails. ChooseS is called again. GeneralizeCounterexample cannot reduce the failing set S_2, which is blocked by updating constraint #vi:

vi. Blocking the failed solutions: $(\neg v_{f_7} \vee \neg v_{f_8}) \wedge (\neg v_{f_7} \vee \neg v_{f_9})$.

Note that any feedback vertex set must cover both the self-loop of f_9 and the circle between f_7 and f_8. Therefore, the only option left is $S_3 = \{\langle f_8, f'_8\rangle, \langle f_9, f'_9\rangle\}$. This time CallEquiv ($f_9^{UF}$, f_9^{UF}) cannot recycle the previous proofs of the call-equivalence of $\langle f_9^{UF}, f_9'^{UF}\rangle$ because $f_{9_{S_3}}^{UF}$ ($f'^{UF}_{9_{S_3}}$) contains abstracted calls to UF_{f_8} ($UF'_{f'_8}$), which is missing in $f_{9_{S_1}}^{UF}$ ($f'^{UF}_{9_{S_1}}$).

S_1, S_2, etc.

Assume that CallEquiv (isolate(f_8, f'_8, S_3)) returns false. In this case, ChooseS is called one more time and returns an empty set indicating that there no more solutions available. Alg. 2 finally gives up by marking the main pair of MSCCs $\langle\{f_0\}, \{f'_0\}\rangle$ as *covered* (but not as *m_term*) in line 22 and ends. This is an example where we are unable to prove the mutual termination of the main functions, but we are able to prove the mutual termination of some of their descendants.

□

$$\max \sum_{g \in \{f_7, f_8, f_9\}} v_g \cdot w_g \ ,$$

where $w_{f_7} = 4$ and $w_{f_8} = w_{f_9} = 3$,
subject to the following constraints for $\{f_7, f_8, f_9, f'_7, f'_8, f'_9, f'_{10}\}$:

i.	Banning unmapped nodes from S:		$\neg v_{f'_{10}}$
ii.	Defining the edges:	$\neg v_{f_7} \wedge \neg v_{f_8} \rightarrow e_{f_7 f_8}$ $\neg v_{f_8} \wedge \neg v_{f_9} \rightarrow e_{f_8 f_9}$ $\neg v_{f_8} \wedge \neg v_{f_7} \rightarrow e_{f_8 f_7}$ $\neg v_{f_9} \wedge \neg v_{f_7} \rightarrow e_{f_9 f_7}$ $\neg v_{f_9} \wedge \neg v_{f_9} \rightarrow e_{f_9 f_9}$	$\neg v_{f'_7} \wedge \neg v_{f'_8} \rightarrow e_{f'_7 f'_8}$ $\neg v_{f'_8} \wedge \neg v_{f'_9} \rightarrow e_{f'_8 f'_9}$ $\neg v_{f'_8} \wedge \neg v_{f'_{10}} \rightarrow e_{f'_8 f'_{10}}$ $\neg v_{f'_9} \wedge \neg v_{f'_8} \rightarrow e_{f'_9 f'_8}$ $\neg v_{f'_{10}} \wedge \neg v_{f'_7} \rightarrow e_{f'_{10} f'_7}$
iii.	Transitive closure of edges:	$e_{f_7 f_8} \wedge e_{f_8 f_9} \rightarrow e_{f_7 f_9}$ $e_{f_7 f_9} \wedge e_{f_9 f_7} \rightarrow e_{f_7 f_7}$ $e_{f_8 f_9} \wedge e_{f_9 f_7} \rightarrow e_{f_8 f_7}$ $e_{f_8 f_7} \wedge e_{f_7 f_8} \rightarrow e_{f_8 f_8}$ $e_{f_9 f_7} \wedge e_{f_7 f_8} \rightarrow e_{f_9 f_8}$ $e_{f_9 f_7} \wedge e_{f_7 f_9} \rightarrow e_{f_9 f_9}$ ...	$e_{f'_7 f'_8} \wedge e_{f'_8 f'_9} \rightarrow e_{f'_7 f'_9}$ $e_{f'_7 f'_8} \wedge e_{f'_8 f'_{10}} \rightarrow e_{f'_7 f'_{10}}$ $e_{f'_7 f'_{10}} \wedge e_{f'_{10} f'_7} \rightarrow e_{f'_7 f'_7}$ $e_{f'_8 f'_{10}} \wedge e_{f'_{10} f'_7} \rightarrow e_{f'_8 f'_7}$ $e_{f'_8 f'_7} \wedge e_{f'_7 f'_8} \rightarrow e_{f'_8 f'_8}$ $e_{f'_9 f'_8} \wedge e_{f'_8 f'_7} \rightarrow e_{f'_9 f'_7}$ $e_{f'_9 f'_8} \wedge e_{f'_8 f'_9} \rightarrow e_{f'_9 f'_9}$ $e_{f'_9 f'_8} \wedge e_{f'_8 f'_{10}} \rightarrow e_{f'_9 f'_{10}}$ $e_{f'_{10} f'_7} \wedge e_{f'_7 f'_8} \rightarrow e_{f'_{10} f'_8}$ $e_{f'_{10} f'_8} \wedge e_{f'_8 f'_9} \rightarrow e_{f'_{10} f'_9}$ $e_{f'_{10} f'_8} \wedge e_{f'_8 f'_{10}} \rightarrow e_{f'_{10} f'_{10}}$ ...
iv.	Forbidding self loops:	$\neg e_{f_7 f_7}$ $\neg e_{f_8 f_8}$ $\neg e_{f_9 f_9}$	$\neg e_{f'_7 f'_7}$ $\neg e_{f'_8 f'_8}$ $\neg e_{f'_9 f'_9}$ $\neg e_{f'_{10} f'_{10}}$
v.	Enforcing mapping $map_{\mathcal{F}}$:	$v_{f_7} \leftrightarrow v_{f'_7}$ $v_{f_8} \leftrightarrow v_{f'_8}$ $v_{f_9} \leftrightarrow v_{f'_9}$	
vi.	Blocking the failed solutions:		

Figure 4.13: A pseudo-Boolean formulation of the optimization problem of finding the largest set of function pairs intersecting all cycles in both $\{f_7, f_8, f_9\}$ and $\{f'_7, f'_8, f'_9, f'_{10}\}$. The list of the transitive closure constraints (iii) is not full as floccinaucinihilipilificated constraints are omitted here.

Chapter 5

Improving completeness

No sound method of proving mutual termination can be complete because this problem is undecidable, but we should strive to improve the completeness of our approach. The two major reasons of its incompleteness were mentioned in Sect. 3.1. They are related to the overapproximation of the real behavior caused by replacing recursive calls with uninterpreted functions. Refining our uninterpreted functions can solve a few of overapproximation-related issues. For example, enforcement (enforce-1) (see (2.6)) is found effective. Recall, it enforces that uninterpreted functions replacing a pair of partially equivalent functions must be the same. Additional ideas for refinements of uninterpreted function which have not yet been implemented will be proposed in Sect. 7.1.

However, there exist other reasons for the incompleteness in our approach. In this chapter we will address a few of them we have coped with. Some of them are applicable to or refine the output of the decomposition algorithm presented in a technical report [22] for verification of partial equivalence. Such improvements are valuable for proving mutual termination too because, as we mentioned above, knowing that some functions are partially-equivalent can be beneficial for establishing their mutual termination.

```
int main() {
    int y, x = 1;
    while (x < 10) {
        y = 2 + x;
        x = y + y;
    }
    return x*2;
}
```

```
int main'() {
    int x' = 1, y';
    while (x' <= 9) {
        y' = x' + 2;
        x' = 2 * y';
    }
    return x' << 1;
}
```

Figure 5.1: Two versions of programs each of which contains a loop with an uninitialized variable y (y') which is written-to before ever being read.

5.1 Reducing prototypes of loop-replacing functions

Appendix C of [20] gives a detailed description of how loops are replaced with functions. Local variables that are used inside loops are part of the interface of the replacing function, even if they are written-to before being read. The problem is that these variables are local and, hence, receive a non-deterministic value and thus make the uninterpreted functions representing the loop return different values. The following example demonstrates the issue.

Example 5.1.1. Consider the pair of C programs listed in Fig. 5.1. Hereafter, the syntax of C is slightly violated, for instance, by ending identifiers of side 1 with **'**, in order to adhere to the notations we have used until now. Extracting the loops into separate recursive functions results in the two programs listed in Fig. 5.2. When partial equivalence of $\langle main, main' \rangle$ is verified, $\langle main^{UF}, main'^{UF} \rangle$ are generated as listed in Fig. 5.3. The values of y and y' in $\langle main^{UF}, main'^{UF} \rangle$, respectively, are non-deterministic. Consequently, not all the arguments passed into calls UF (Loop_main_while1, &x, &y) and UF' (Loop_main_while1', &x', &y') are considered equal, because direct pointers are considered equal if they point to equal values. Thus those calls are considered different. As a result, $\langle main^{UF}, main'^{UF} \rangle$ are not consid-

```
int Loop_main_while1(int *px,
                     int *py)
{
   if (!(*px < 10)) return 0;
   *py = 2 + *px;
   *px = *py + *py;
   return Loop_main_while1(px, py);
}


int main() {
   int y, x = 1;
   Loop_main_while1(&x, &y);
   return x*2;
}
```

```
int Loop_main_while1'(int *px',
                      int *py')
{
   if (!(*px' <= 9)) return 0;
   *py' = *px' + 2;
   *px' = 2 * *py';
   return Loop_main_while1'(px', py');
}


int main'() {
   int x' = 1, y';
   Loop_main_while1'(&x', &y');
   return x' << 1;
}
```

Figure 5.2: Two versions of programs from Fig. 5.1 after elimination of their loops.

```
...


int main^UF() {
   int y, x = 1;
   UF (Loop_main_while1, &x, &y);
   return x*2;
}
```

```
...


int main'^UF() {
   int x' = 1, y';
   UF' (Loop_main_while1', &x', &y');
   return x' << 1;
}
```

Figure 5.3: Parts of the program generated for proving the mutual termination of functions $main$, $main'$, defined in Fig. 5.2.

```
int Loop_main_while1(int *px) {
   int y;
   if (!(*px < 10)) return 0;
   y = 2 + *px;
   *px = y + y;
   return Loop_main_while1(px);
}

int main() {
   int y, x = 1;
   Loop_main_while1(&x);
   return x*2;
}
```

```
int Loop_main_while1'(int *px') {
   int y';
   if (!(*px' <= 9)) return 0;
   y' = *px' + 2;
   *px' = 2 * y';
   return Loop_main_while1'(px');
}

int main'() {
   int x' = 1, y';
   Loop_main_while1'(&x');
   return x' << 1;
}
```

Figure 5.4: Two versions of programs from Fig. 5.1 after replacement of their loops with functions and reduction of variables y and y' from the argument lists of those replacing functions. See Fig. 5.2 for a comparison.

ered call-equivalent. Hence, RVT will fail to prove $m\text{-}term(main^{UF}, main'^{UF})$. □

There is no good reason to include the variables of loop bodies that satisfy the two following conditions, into the argument list of the functions that replace the loop:

C1. before their values are ever read, some value is assigned into them, and

C2. they are no longer used after the body of the loop.

They may become mere local variables in the replacing functions.

Example 5.1.2. Reconsider the programs given in Fig. 5.1 and note that variable y (y') in function *main* (*main*′) is initialized every time before being read in the loop-body. In fact, there is a single execution path in that loop-body. Thus, y (y') satisfies condition C1. Moreover, note that it is not used after the end of the loop body, i.e., it satisfies C2 too. Hence,

py (py') may be reduced from the argument list of the loop-replacing function $Loop\_main\_while1$ ($Loop\_main\_while1'$), and, furthermore, y (y') may become a local variable inside it as listed in Fig. 5.4. Now $m\text{-}term(main, main')$ can be proven.

□

Here is a description of the procedure we apply for detecting variables that satisfy conditions C1 and C2. Validating C1 amounts to checking that a variable is initialized before being read in every computation path in the loop-body block. If it passes the check, C2 shall be validated. The latter validation is done using *live-variables* analysis [32]. If it establishes that the variable has stopped being a live variable by the end of the loop body, then C2 holds.

Two simple *intraprocedural static analyses* [25] aid to validate C1 in a checked loop-body block. The first analysis, which we call *Write-To (WT)*, for each node of the control flow graph of that block, finds variables that something is written to them in *all* execution paths leading to the node, including writings in this node itself. The nodes of control flow graphs on which we run our analyses are expressions in the C-language. WT is a *flow-sensitive forward* [17, 38] *must* [33] analysis. Based on its results, the second one, called *Read-Uninitialized (RU)*, finds those variables that may be read before something is written to them, i.e., detects potential reads of unitialized variables. Those variables of the checked loop which are not listed in the results of RU are written-to before being read in this loop-body, i.e., satisfy C1. RU is a flow-sensitive forward *may* [33] analysis. Both analyses are formally defined in Tables 5.1 and 5.2.

The described reduction of variables from the argument lists of loop-replacing functions can be useful for proving partial equivalence too.

5.2 Mapping functions with different numbers of input parameters

Recall that in Sect. 2.1 we imposed a bijective map $map_{\mathcal{F}}$ between the functions of the two compared programs P, P' as a precondition to apply our

kill **function**
$kill(B^\ell) \quad = \quad \emptyset$
gen **function**
$gen(B^\ell) \quad = \quad def(B)$ in all other cases: $gen(B^\ell) \quad = \quad \emptyset$
Data flow equations $\mathsf{WT}^=$
$\mathsf{WT}_{entry}(\ell) \quad = \quad \begin{cases} \emptyset & \text{if } \ell = init(S_\star) \\ \bigcap\{\mathsf{WT}_{exit}(\ell') \mid (\ell', \ell) \in flow(S_\star)\} & \text{otherwise} \end{cases}$ $\mathsf{WT}_{exit}(\ell) \quad = \quad \big(\big(\mathsf{WT}_{entry}(\ell) \setminus kill(B^\ell)\big) \cup gen(B^\ell)\big)$, where $B^\ell \in blocks(S_\star)$

Table 5.1: Definition of WT analysis. This is an *intraprocedural flow-sensitive forward* ($F = flow(S_\star)$) *must* ($\bigsqcup = \bigcap$) analysis. Let $def(n)$ denote the set of the variables updated in the control flow graph node n. See Chapter 2 of [33] for understanding the rest of the notations used here.

kill **function**
$kill(B^\ell) \quad = \quad \emptyset$
gen **function**
$gen(B^\ell) \quad = \quad \{v \mid v \in use(B) \wedge v \notin \mathsf{WT}_{entry}(\ell)\}$
Data flow equations $\mathsf{RU}^=$
$\mathsf{RU}_{entry}(\ell) \quad = \quad \begin{cases} \emptyset & \text{if } \ell = init(S_\star) \\ \bigcup\{\mathsf{RU}_{exit}(\ell') \mid (\ell', \ell) \in flow(S_\star)\} & \text{otherwise} \end{cases}$ $\mathsf{RU}_{exit}(\ell) \quad = \quad \big(\big(\mathsf{RU}_{entry}(\ell) \setminus kill(B^\ell)\big) \cup gen(B^\ell)\big)$, where $B^\ell \in blocks(S_\star)$

Table 5.2: Definition of RU analysis. This is an *intraprocedural flow-sensitive forward* ($F = flow(S_\star)$) *may* ($\bigsqcup = \bigcup$) analysis. Let *use(n)* denote the set of the variables which are read in the control flow graph node n. See Chapter 2 of [33] for understanding the rest of the notations used here.

```
int h(int x) {

   if (x <= 0)
      return h(1 - x);
   return x;
}
```

```
int h'(int x', int b') {
   if (b' != 0)
      report'("...");
   if (x' <= 0)
      return h'(1 - x', b');
   return x';
}

void report'(const char *s') {
   ...
}
```

Figure 5.5: Two versions of a program where functions h and h' have different prototypes. Nevertheless, we would like to prove $m\text{-}term(h, h')$.

algorithm. Furthermore, for functions $f \in P$ and $f' \in P'$, $\langle f, f' \rangle \in map_{\mathcal{F}}$ only if f and f' have the same prototypes. Our method requires this in order to be able to check $call\text{-}equiv(f^{UF}, f'^{UF})$. However, we would like to extend the definition of mutual termination so that it captures cases in which although the two functions have different numbers of input parameters, they terminate with respect to the common elements of their prototypes. The following example demonstrates such a case.

Example 5.2.1. Consider two versions of a program listed in Fig. 5.5. Functions h and h' have different numbers of input arguments. However, argument b' does not affect the guarding conditions of recursive calls in h'. Thus we would like to still be able to prove that h and h' mutually terminate regardless of the value of b'.

□

In Sect. 5.2.1 we will formally present a method for detecting such input parameters as b' in function h' from Ex. 5.2.1. We coin input parameters which have no influence on the termination of their function *termination-inert*. But now we will describe what we do with them assuming we have detected them.

Let Π denote the indices of the common elements in the prototypes of f and f'. Let $\boldsymbol{in}$ and $\boldsymbol{in'}$ denote the actual input arguments of f and f' respectively, and let $\boldsymbol{in}|_\Pi$, $\boldsymbol{in'}|_\Pi$ denote their respective projections to the elements defined by Π. The following definition generalizes Def. 2.2.1 to functions with different prototypes.

Definition 5.2.1 (Generalized mutual termination of functions)**.** Two functions f and f' are *mutually terminating*, if and only if

$$\forall \boldsymbol{in}, \boldsymbol{in'}.\ \boldsymbol{in}|_\Pi = \boldsymbol{in'}|_\Pi \rightarrow \forall \pi \in [\![f(\boldsymbol{in})]\!], \pi' \in [\![f'(\boldsymbol{in'})]\!].\ term(\pi) \leftrightarrow term(\pi')\ . \tag{5.1}$$

We continue to use the predicate $m\text{-}term(f, f')$ to denote mutual termination of f, f', only that now it also applies to the case that f, f' have different prototypes, according to the definition above. Note that (5.1) generalizes (2.1), and is more difficult to prove because (5.1) has a universal quantifier over variables that are not constrained on the left-hand-side of the implication. For example, considering h and h' of Example 5.2.1, we need to prove:

$$\forall x, x', b'.\ x = x' \rightarrow \forall \pi \in [\![h(x)]\!], \pi' \in [\![h'(x', b')]\!].\ term(\pi) \leftrightarrow term(\pi')\ ,$$

that is, b' is unconstrained.

We suggest to reduce the problem of generalized mutual termination (Def. 5.2.1) to that of mutual termination (Def. 2.2.1) by deriving new non-deterministic functions from f, f' so that their inputs are restricted to the common part Π. We call this construction *hiding*.

Definition 5.2.2 (Hiding function parameters)**.** Given a function f in a program P and a subset B of the formal input parameters of f, the *hiding* of B is given by the following transformation of P:

- Let $f_{\downarrow B}$ be f after the following transformation:
  - remove B from the prototype of f;
  - declare the B variables as local variables of f, and initialize them with non-deterministic values;

```
int h'↓{b'}(int x') {
    int b' = nondet();
    if (b' != 0)
        report'("...");
    if (x' <= 0)
        return h'↓{b'}(1 - x');
    return x';
}
```

Figure 5.6: Function $h'_{\downarrow\{b'\}}$, derived from function h' (see Fig 5.5) 'hiding' b' from the parameter list (see Def. 5.2.2).

- In P, replace f with $f_{\downarrow B}$ and all calls to f with calls to $f_{\downarrow B}$.

As an example, hiding of b' in Fig. 5.5 of Ex. 5.2.1 results in the function appearing in Fig. 5.6. Functions h and h' have different numbers of input parameters. However, no value of parameter b' can affect the guarding conditions of recursive calls in h'. Thus we would like to still be able to prove that h and h' mutually terminate regardless of the value of b'.

Let B, B' be the set of parameters of f, f', respectively, outside of the common part Π, i.e., these are parameters that are not mapped to parameters in the other function. Hiding these parameters results in $f_{\downarrow B}$ and ${f'}_{\downarrow B'}$, which have the same prototype and can therefore be checked for mutual termination with the inference rules of Chapter 3. It is left to prove that mutual termination of these transformed functions imply the mutual termination of the original ones. In other words, we need to prove the soundness of the following proof rule:

$$\frac{m\text{-}term(f_{\downarrow B}, {f'}_{\downarrow B'})}{m\text{-}term(f, f')} \quad (\text{M-TERM-}\Pi) \quad . \tag{5.2}$$

Proof. It is sufficient to show that $f_{\downarrow B}$ overapproximates f and ${f'}_{\downarrow B'}$ overapproximates f', because mutual termination of overapproximating functions clearly implies mutual termination of the original ones. Indeed $f_{\downarrow B}$ overapproximates f because by construction, for any actual values passed to the

B parameters in f, the executions of f can be mimicked by $f_{\downarrow B}$ by choosing the same values for the corresponding local variables (recall that they are initialized with nondeterministic values). The same argument proves that ${f'}_{\downarrow B'}$ overapproximates f'. □

Note that the modifications that create $f_{\downarrow B}$ (${f'}_{\downarrow B'}$) do not necessarily preserve the semantics of f (f'). Consequently, checking $m\text{-}term(f_{\downarrow B}, {f'}_{\downarrow B'})$ usually involves *different* uninterpreted functions.

Example 5.2.2. Reconsider functions h and h' listed in Fig 5.5. Parameter b' of the prototype of h' is a termination-inert input argument. It can be excluded from the parameter list of h'. Function $h'_{\downarrow\{b'\}}$, listed in Fig 5.6, and h, defined in Fig 5.5, have the same prototype. Now RVT can prove $m\text{-}term(h, h'_{\downarrow\{b'\}})$ and infer $m\text{-}term(h, h')$.

□

5.2.1 Detecting termination-inert input parameters

An algorithm for checking whether a given argument v of function f is a termination-inert input argument of f consists of two stages. First, it builds a *System Definition Graph* [26] (SDG) for program P where f is defined.

Briefly, an SDG is an extension of a *Program Dependence Graph* [16, 29] (PDG) for multi-function programs. The original nodes of some function's PDG represent the statements of the function. The edges of the PDG represent data and control dependencies between the statements of the function and thus define their partial order: the semantics of the function is preserved if its statements are executed in this order.

Algorithm 5 Algorithm for checking whether an input argument is termination-inert.

```
1: function IsCallEquivInert(Program P, function f, argument v)
2:     Build an SDG for P;
3:     for each call to f in this SDG do
4:         if this call is reachable from node v = v_{in_f} then return FALSE;
5:     return TRUE;
```

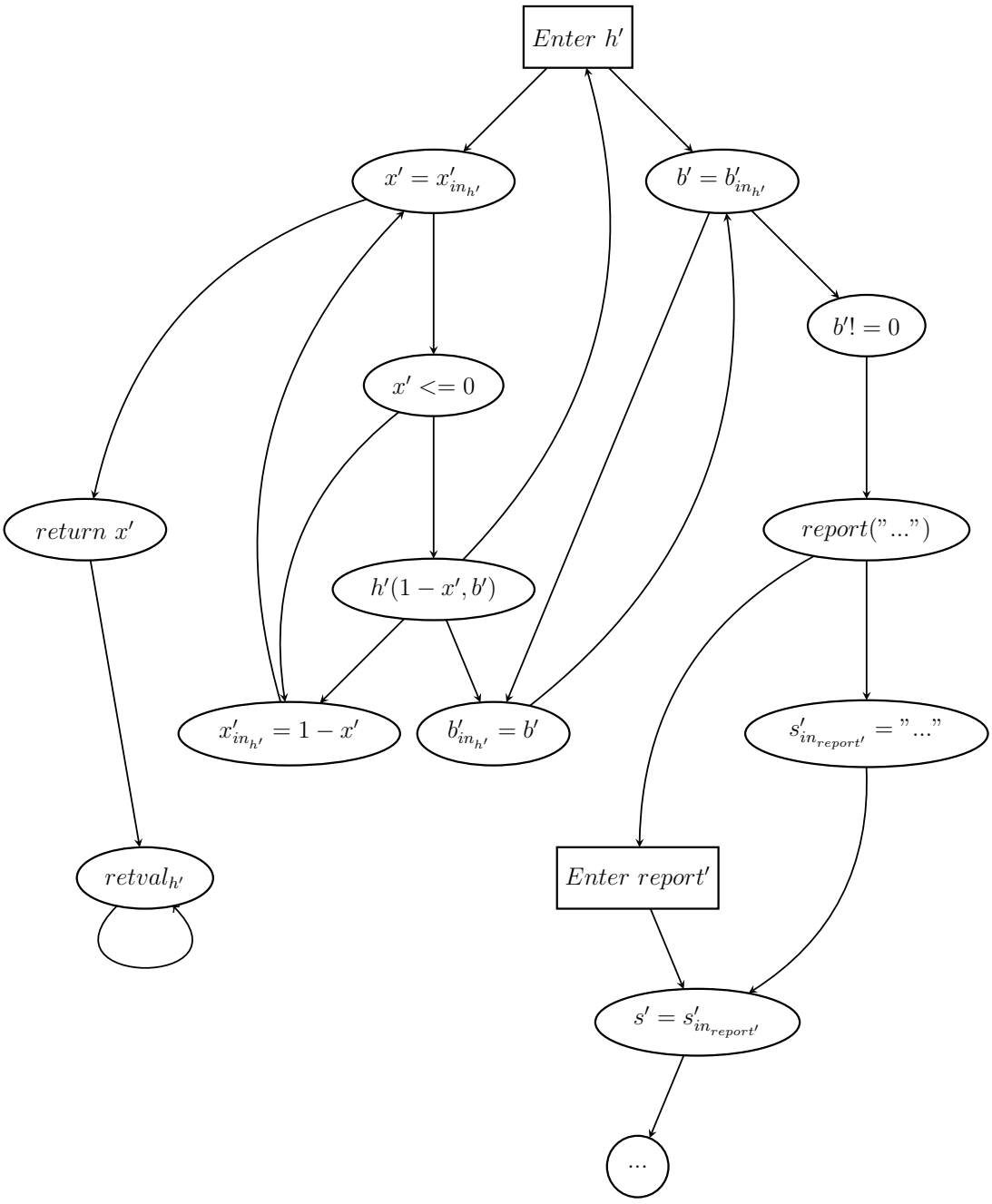

Figure 5.7: The System Definition Graph [26] of the sub-program starting in function h', defined in Fig 5.5.

An SDG consists of PDGs for each function of the program plus the following additions. Each function g of the program is associated with an entrance node "*Enter g*". For each input argument u of this function, the SDG contains a node of type $u = u_{in_g}$ and an edge entering this node and leaving node "*Enter g*". Each node representing a call to function g has a leaving edge entering the entrance node of g, i.e., "*Enter g*". In addition, for an expression *expr* passed as parameter u in that call, there are a node of type $u_{in_g} = expr$ with the two following edges:

- an entering edge which leaves that function-call node, and
- a leaving edge which enters the recently mentioned node of type $u = u_{in_g}$.

The return value of g has its own dedicated node $retval_g$. Its entering edges leave nodes whose statements affect the return value. Its leaving edges enter nodes whose statements depend on the return value. Fig. 5.7 demonstrates an example of an SDG built for the sub-program starting in function h' from Fig 5.5.

At the second stage the algorithm checks whether any of the calls to function f is reachable from node $v = v_{in_f}$, where, recall, v is the name of the given input argument. If none is reachable, then argument v is a termination-inert input argument of f. The algorithm is presented in Alg. 5.

Example 5.2.3. Regard the SDG in Fig. 5.7, built for the sub-program starting in function h' from Fig 5.5. It has no node of a function call to h' which is reachable from node $b' = b'_{in_{h'}}$. Hence, b' does not affect any guarding condition over any recursive call to h'.

5.3 Partial equivalence with respect to a subset of outputs

The improvement reported in this section refines the output of the decomposition algorithm for checking partial equivalence [22].

Recall that functions in a language such as C may have multiple outputs, and that so far we defined partial equivalence with respect to all of them, i.e.,

```
int g(int x, int *p) {
  if (x > 5 || p == NULL)
    return 0;
  *p = 0;
  g(g(x + 1, p), NULL);
  return x;
}
```

```
int g'(int x', int *p') {
  if (x' > 5 || p' == NULL)
    return 0;
  *p' = 1;
  g'(g'(x' + 1, p'), NULL);
  return x';
}
```

```
int g^UF(int x, int *p) {
  if (x > 5 || p == NULL)
    return 0;
  *p = 0;
  UF_g(UF_g(x + 1, p), NULL);
  return x;
}
```

```
int g'^UF(int x', int *p') {
  if (x' > 5 || p' == NULL)
    return 0;
  *p' = 1;
  UF'_g'(UF'_g'(x' + 1, p'), NULL);
  return x';
}
```

Figure 5.8: (top) Functions g and g' are partially equivalent with respect to their return value, but not with respect to the other output $*p, *p'$. We show that this 'restricted' partial equivalence is sufficient for proving mutual termination; (bottom) the isolated versions of g, g'.

given the same inputs, the two functions are equivalent in all output elements pair-wise. However, sometimes the equivalence of *some* of the outputs is sufficient for proving mutual termination, as demonstrated in the following example.

Example 5.3.1. Consider the functions listed at the top of Fig. 5.8. Formally, g and g' are not partially equivalent because different values are assigned to p and p', which are among the outputs of g and g', respectively. But the return values of g and g' are equivalent. This fact could be useful for establishing $m\text{-}term(g, g')$.

Consider g^{UF} and g'^{UF} listed at the bottom of Fig. 5.8. The obstacle for proving $call\text{-}equiv(g^{UF}, g'^{UF})$ is the fact that UF_g and $UF'_{g'}$ are different uninterpreted functions (because $\neg p\text{-}equiv(g, g')$). We may solve this problem by enforcing the equivalence of the *return* values of UF_g and $UF'_{g'}$ only (but

not those of `*p,*p'`). We can do this if we are able to prove that g and g' are partially equivalent with respect to their return values.

□

Let *out(f)* denote the list of outputs of f.

Definition 5.3.1 (Partial equivalence with respect to individual outputs)**.** Two functions f and f' are *partially equivalent with respect to individual outputs* $\langle o, o' \rangle$ such that $o \in out(f) \wedge\ o' \in out(f')$ if and only if for all input $\boldsymbol{in}$ and for any $\pi \in [\![f(\boldsymbol{in})]\!], \pi' \in [\![f'(\boldsymbol{in})]\!]$ which satisfy $term(\pi) \wedge term(\pi')$, π and π' end with the same value for o and o'.

Let $p\text{-}equiv_{\langle o,o' \rangle}(f, f')$ denote the fact that f and f' are partially equivalent with respect to $\langle o, o' \rangle$. We can now refine enforcement (enforce-1) (see (2.6)):

$$UF_f \equiv_{\langle o,o' \rangle} UF'_{f'} \rightarrow \big(\langle f, f' \rangle \in map_{\mathcal{F}} \wedge p\text{-}equiv_{\langle o,o' \rangle}(f, f') \big) \qquad \text{(enforce-2)}\ , \tag{5.3}$$

where $\equiv_{\langle o,o' \rangle}$ is the natural restriction of $\equiv$ to $\langle o, o' \rangle$. We refine the implementation of UF' (see Fig. 4.1) in a manner compatible with this condition, i.e., given the same inputs, the values of o and o' are still non-deterministic but forced to be the same when we are able to prove $p\text{-}equiv_{\langle o,o' \rangle}(f, f')$. Otherwise, no such enforcement is made. The correspondingly refined implementation of UF' is shown at the bottom of Fig. 5.9.

Our tool RVT is capable of proving partial equivalence of functions with respect to individual outputs. When it is activated for checking partial equivalence, it first attempts to establish $p\text{-}equiv(f, f')$ for each $\langle f,\ f' \rangle$ that it is checks. Only if it fails to have proven this, it checks the equivalence of output elements one by one. For each pair of output elements $\langle o,\ o' \rangle$ with respect to which partial equivalence could be proven, it assigns label $part\_eq_{\langle o,o' \rangle}$ to $\langle f,\ f' \rangle$. Thereby it finds a maximal mapping $\{ \langle o, o' \rangle \mid o \in out(f) \wedge o' \in out(f') \wedge p\text{-}equiv_{\langle o,o' \rangle}(f, f') \}$. This mapping can be also useful when the outputs of f cannot be bijectively mapped with the outputs of f'.

```
1: function UF(function index g, input parameters in)          ▷ Called in side 0
2:     if in ∈ params[g] then return the output of the earlier call UF(g, in);
3:     params[g] := params[g] ⋃ in;
4:     return a non-deterministic output;

5: function UF'(function index g', input parameters in')       ▷ Called in side 1
6:     if in' ∈ params[g'] then return the output of the earlier call UF'(g', in');
7:     params[g'] := params[g'] ⋃ in';
8:     if in' ∈ params[g] then                                  ▷ ⟨g, g'⟩ ∈ map_F
9:         result := [];
10:        for each o_i ∈ out(g) do                             ▷ o'_i ∈ out(g')
11:            if ⟨g, g'⟩ is marked as part_eq or as part_eq_⟨o_i,o'_i⟩ then
12:                append the result for o_i of the earlier call UF(g, in') to result;
13:            else append a non-deterministic value to result;
14:        return result;
15:    assert(0);                  ▷ Not call-equivalent: params[g'] ⊈ params[g]
```

Figure 5.9: Implementations for functions UF and UF', where the latter takes into consideration partial information about partial equivalence. UF and UF' emulate uninterpreted functions if instantiated with functions that are mapped to one another, and form a part of the generated program δ, as shown in CallEquiv of Alg. 1 or in the determinization thereof Alg. 2 (see pages 29, 42). These functions also contain code for recording the parameters with which they are called.

Chapter 6

Inference rules for proving termination

6.1 Proof rule (TERM)

We now consider a different variant of the mutual termination problem: *Given that a program P terminates, does P' terminate as well?* Clearly this problem can be reduced to that of mutual termination, but in fact it can also be solved with a weaker premise. We first define:

Definition 6.1.1 (Call-containment)**.** Function f *call-contains* a function f' if and only if

$$\forall \boldsymbol{in}, \boldsymbol{in'}.\ \boldsymbol{in} = \boldsymbol{in'} \rightarrow \forall \pi \in [\![f(\boldsymbol{in})]\!], \pi' \in [\![f'(\boldsymbol{in'})]\!].\ calls(\pi^1) \supseteq calls(\pi'^1)\ . \tag{6.1}$$

Denote by $f \sqsupseteq_c f'$ the fact that f call-contains f'. We now overload $term$ to refer to the set of computations possible in a function f with any input:

$$term(f) \doteq \forall \boldsymbol{in}.\ \forall \pi \in [\![f(\boldsymbol{in})]\!].\ term(\pi)\ . \tag{6.2}$$

We can now define the rule for leaf MSCCs m, m':

$$\varphi(m,n,p) = \begin{cases} m+n & if\ p = 0 \\ 0 & if\ n = 0 \wedge p = 1 \\ 1 & if\ n = 0 \wedge p = 2 \\ m & if\ n = 0 \wedge p > 2 \\ \varphi(m, \varphi(m, n-1, p), p-1) & if\ n > 0 \wedge p > 0 \end{cases}$$

$$A(a,b) = \begin{cases} b+1 & if\ a = 0 \\ A(a-1,1) & if\ a > 0 \wedge b = 0 \\ A(a-1, A(a, b-1)) & if\ a > 0 \wedge b > 0 \end{cases}$$

Figure 6.1: The original Ackermann [3] function φ and its two-variable variation A, developed by Péter and Robinson [34].

$$\frac{\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ \left(term(f) \wedge f^{UF} \sqsupseteq_c f'^{UF}\right)}{\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ term(f')}\ (\textsc{term})\ . \tag{6.3}$$

One can note that call equivalence (Def. 2.2.3) is simply bi-directional call containment, which makes the premise of the new rule weaker than that of (M-TERM) (see (3.2)) for proving mutual termination. The soundness of (TERM) will be proven in Sect. 6.3. A generalization to non-leaf MSCCs will be given in Sect. 6.2.

Example 6.1.1. Consider the original definition of Ackermann function [3] and its more famous two-argument variation, developed by Péter and Robinson [34]. Fig. 6.1 displays the functions. The arguments of each variation are non-negative integers. Observe that once φ is called with some m_0 passed as the first argument, all the recursive calls of φ pass m_0 as the first argument.

$$\varphi_m^{UF}(n,p) = \begin{cases} m+n & if\ p=0 \\ 0 & if\ n=0 \wedge p=1 \\ 1 & if\ n=0 \wedge p=2 \\ m & if\ n=0 \wedge p>2 \\ UF_{\varphi_m}(UF_{\varphi_m}(n-1,p),p-1) & if\ n>0 \wedge p>0 \end{cases}$$

$$A^{UF}(a,b) = \begin{cases} b+1 & if\ a=0 \\ UF_A(a-1,1) & if\ a>0 \wedge b=0 \\ UF_A(a-1,UF_A(a,b-1)) & if\ a>0 \wedge b>0 \end{cases}$$

Figure 6.2: The isolated versions of the original Ackermann function φ and its more famous two-variable variation A, developed by Péter and Robinson.

This observation allows to define function φ_m as following:

$$\varphi_m(n,p) = \begin{cases} m+n & if\ p=0 \\ 0 & if\ n=0 \wedge p=1 \\ 1 & if\ n=0 \wedge p=2 \\ m & if\ n=0 \wedge p>2 \\ \varphi_m(\varphi_m(n-1,p),p-1) & if\ n>0 \wedge p>0 \end{cases}$$

Note that $\varphi_m(n,p) = \varphi(m,n,p)$. Now the prototypes of φ_m and A can be matched:

- argument n in φ_m matches argument b in A, and
- argument p in in φ_m matches argument a in A.

Their corresponding isolated versions are shown in Fig. 6.2.

UF_{φ_1} can be called in ${\varphi_1}^{UF}(y,x)$ only when both x and y are positive. In this case the following calls take place:

- $UF_{\varphi_1}(y-1,x)$, and

- $UF_{\varphi_1}(UF_{\varphi_1}(y-1,x),x-1)$.

But given those positive x and y, the calls of $UF_A(y,x)$ in A^{UF} are:

- $UF_A(x,y-1)$, and
- $UF_A(x-1,UF_A(x,y-1))$.

If we are given that φ_1 and A are partially equivalent, we can enforce:

$$UF_{\varphi_1}(y,x) = UF_A(x,y) \ ,$$

and then:

- the first parameter of $UF_{\varphi_1}(UF_{\varphi_1}(y-1,x),x-1)$ will be equal to the second parameter of $UF_A(x-1,UF_A(x,y-1))$, and
- the second parameter of $UF_{\varphi_1}(UF_{\varphi_1}(y-1,x),x-1)$ will be equal to the first parameter of $UF_A(x-1,UF_A(x,y-1))$.

Consequently,

$$\begin{array}{l} \forall x>0, y>0, w, z. \\ \quad \varphi_1(z,w) \in calls\left({\varphi_1}^{UF}(y,x)\right) \leftrightarrow A(w,z) \in calls\left(A^{UF}(x,y)\right) \ . \end{array} \tag{6.4}$$

It was previously mentioned that ${\varphi_1}^{UF}(y,x)$ contains no function calls when either x or y (or both) is not positive. Hence,

$$\forall x, y \in \mathbb{N}.(x=0 \vee y=0) \rightarrow calls\left({\varphi_1}^{UF}(y,x)\right) = \emptyset \tag{6.5}$$

The combination of (6.4) and (6.5) will imply $A^{UF} \sqsupseteq_c {\varphi_1}^{UF}$. Having taken in consideration that $term(A)$ is a known fact [6], we can conclude $term(\varphi_1)$ according to rule (TERM), i.e., that the original version of the Ackermann function $\varphi(m,n,p)$ terminates for $m=1$.

□

The weakness of our method appears for $m \neq 1$, when $\varphi(m,y,x)$ and $A(x,y)$ are not partially equivalent. In this case we cannot prove $A^{UF} \sqsupseteq_c {\varphi_m}^{UF}$ because we may not enforce $UF_{\varphi_m}(y-1,x) = UF_A(x,y-1)$.

6.2 Generalized rule (TERM$^+$)

A generalization to non-leaf MSCCs can be done in a similar way to (3.4):

$$\frac{\begin{array}{l}\forall\, \langle f, f' \rangle \in map_{\mathcal{F}}(m).(term(f) \wedge f^{UF} \sqsupseteq_c f'^{UF}) \wedge \\ \forall \langle g, g' \rangle \in map_{\mathcal{F}}.\ ((g \in C(m) \wedge g' \in C(m')) \rightarrow m\text{-}term(g, g'))\end{array}}{\forall\, \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ term(f')}\ (\text{TERM}^+)\ , \tag{6.6}$$

where, recall, $C(m)$ denotes the functions that are outside of m and are called by functions in m. A proof appears in Sect. 6.3.

The decomposition algorithm applies with the following change: in function CallEquiv , the statement asserting $params[g] \subseteq params[g']$ (line 31 in Alg. 2) should be removed. Namely, the adapted version of function CallEquiv is presented in Alg. 6. The only assertion that should be verified is thus inside UF' (see line 12 in Fig. 4.1 or line 15 of the refined implementation thereof in Fig. 5.9). It checks that every call on side 0 is matched by a call on side 1.

6.3 Soundness proofs for (TERM) and (TERM$^+$)

The outline of the proof of (TERM) is the following:

- In Theorem 6.3.1 we will falsely assume that the premise $f^{UF} \sqsupseteq_c f'^{UF}$ for all pairs $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$ holds, whereas there exists computations $\pi \in [\![f(\boldsymbol{in})]\!]$ and $\pi' \in [\![f'(\boldsymbol{in})]\!]$ such that $term(\pi)$ and $\neg term(\pi')$.
- In Lemma 6.3.2 we show that $term(\pi)$ together with the premise above implies that $\forall \pi' \in [\![f'(\boldsymbol{in})]\!].\ term(\pi')$, which is a contradiction.
- To prove Lemma 6.3.2 we first prove an auxiliary lemma—Lemma 6.3.1—that if there exists a function call $g'(\boldsymbol{in_g})$ in π'^1 and π is finite, then π^1 must contain a matching call to $g(\boldsymbol{in_g})$ where $\langle g, g' \rangle \in map_{\mathcal{F}}$.

Algorithm 6 CallEquiv from Alg. 2 updated for proving termination of functions.

30: **function** CallEquiv(A pair of isolated functions $\langle f^{UF}, f'^{UF} \rangle$)
31: Let δ denote the program:
$\triangleright$ here add the definitions of UF() and UF'()
$\triangleright$ (see Fig. 4.1 or Fig. 5.9).
$\boldsymbol{in} := nondet()$;
$f^{UF}(\boldsymbol{in})$;
$f'^{UF}(\boldsymbol{in})$;
32: **if** WasProven($proven$, $\langle f^{UF}, f'^{UF} \rangle$) **or**[a] CBMC($\delta$) **then**
33: $proven := proven \cup \{\langle f^{UF}, f'^{UF} \rangle\}$;
34: **return** TRUE;
35: **return** FALSE

[a]The second condition is evaluated only if the first condition has been evaluated as false.

We begin by defining, for a given a computation π:

$$\pi^{UF} \doteq \pi^1[g(\boldsymbol{in_g}) \leftarrow UF_g(\boldsymbol{in_g}) \mid \langle g, \boldsymbol{in_g} \rangle \in calls(\pi^1)] \ , \tag{6.7}$$

namely we replace the function calls in π^1 with calls to their respective uninterpreted functions, with the same arguments. It is not hard to see that

$$\frac{\pi \in [\![f(\boldsymbol{in})]\!] \wedge term(\pi)}{\pi^{UF} \in [\![f^{UF}(\boldsymbol{in})]\!]} \ . \tag{6.8}$$

When π is infinite, on the other hand, there may be statements in f that would be executed if the non-terminating call *would* have returned. Since the call is replaced by an uninterpreted function that *does* return, those statements will be executed in f^{UF}. In such a case there must exist a computation $\hat{\pi}$ in $[\![f^{UF}(\boldsymbol{in})]\!]$ that extends π^{UF}. In other words, π^{UF} is a prefix of $\hat{\pi}$. More formally, letting $prefix(\pi^{UF}, \hat{\pi})$ denote that π^{UF} is a prefix of $\hat{\pi}$, we have

$$\frac{\pi \in [\![f(\boldsymbol{in})]\!]}{\exists \hat{\pi} \in [\![f^{UF}(\boldsymbol{in})]\!].\ prefix(\pi^{UF}, \hat{\pi})} \ . \tag{6.9}$$

Lemma 6.3.1. For any given pair of functions $\langle f, f' \rangle \in map_{\mathcal{F}}$, function g', and inputs $\boldsymbol{in}, \boldsymbol{in_g}$, the following inference is sound for any $\pi \in [\![f(\boldsymbol{in})]\!]$:

$$\frac{term(\pi) \wedge f^{UF} \sqsupseteq_c f'^{UF} \wedge \exists \pi' \in [\![f'(\boldsymbol{in})]\!].\ \langle g', \boldsymbol{in_g} \rangle \in calls(\pi'^1)}{\exists g.\ (\langle g, g' \rangle \in map_{\mathcal{F}} \wedge \langle g, \boldsymbol{in_g} \rangle \in calls(\pi^1))} \tag{6.10}$$

Its proof reminds very much of that given for Lemma 3.4.1 in Sect. 3.4.

Proof. Let $\langle f, f' \rangle \in map_{\mathcal{F}}$, input $\boldsymbol{in}$, function g' and input $\boldsymbol{in_g}$ satisfy the premise. The bijectivity of $map_{\mathcal{F}}$ ensures existence of a function g such that $\langle g, g' \rangle \in map_{\mathcal{F}}$.

By (6.9) π'^{UF} is a prefix of some $\hat{\pi}' \in [\![f'^{UF}(\boldsymbol{in})]\!]$. Note that $\langle UF_{g'}, \boldsymbol{in_g} \rangle \in calls(\pi'^{UF})$, which implies $\langle UF_{g'}, \boldsymbol{in_g} \rangle \in calls(\hat{\pi}')$. Hence, $f^{UF} \sqsupseteq_c f'^{UF}$ implies:

$$\forall \hat{\pi} \in [\![f^{UF}(\boldsymbol{in})]\!].\ \langle UF_g, \boldsymbol{in_g} \rangle \in calls(\hat{\pi})\ . \tag{6.11}$$

The premise of (6.8) holds, which implies $\pi^{UF} \in [\![f^{UF}(\boldsymbol{in})]\!]$. Thus (6.11) implies $\langle UF_g, \boldsymbol{in_g} \rangle \in calls(\pi^{UF})$. The construction of π^{UF} implies $\langle g, \boldsymbol{in_g} \rangle \in calls(\pi^1)$. □

We now define:

$$depth(\pi) \doteq max\{|s| \mid s \in \mathcal{S}(\pi)\}\ , \tag{6.12}$$

where, recall, $\mathcal{S}(\pi)$ denotes the set of call-stacks appearing during a given computation π, and for $s \in \mathcal{S}(\pi)$, $|s|$ is the number of frames in s (possibly infinite). Fig. 6.3 illustrates $depth(\pi)$, for the program listed in Fig. 2.1.

Our proofs of the lemmas in the rest of this chapter rely on the following observations:

O1. $term(\pi) \leftrightarrow \exists d \in \mathbb{Z}^+.\ depth(\pi) \leq d$;

O2. $\neg term(\pi)$ implies that among all function calls made in π^1, there is exactly one which does not return, and its call-statement is the last statement in π^1.

The next lemma addresses mutually recursive functions without outer calls, i.e., calls to functions outside the MSCCs.

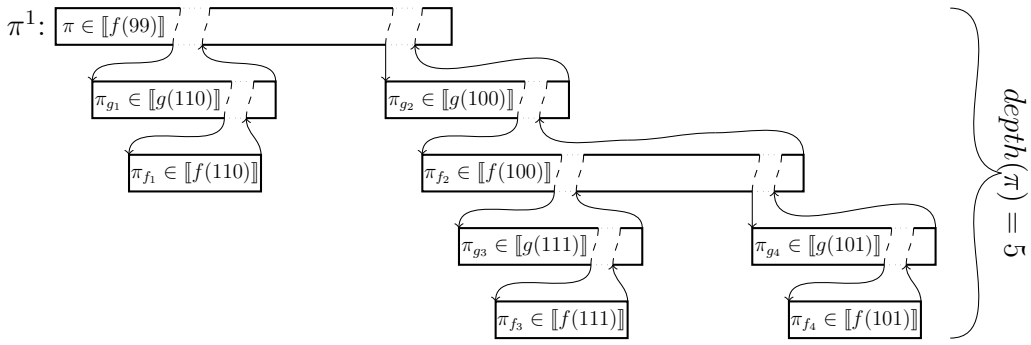

Figure 6.3: An illustration of $depth(\pi)$, defined in (6.12), for a computation $\pi \in [\![f(99)]\!]$, where f is defined in Fig. 2.1. For a subcomputation π_{g_i} of π beginning at g (a callee of f), $depth(\pi_{g_i}) < depth(\pi)$.

Lemma 6.3.2. For any given $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$ called with the same input $\boldsymbol{in}$,

$$\frac{\forall \langle h, h' \rangle \in map_{\mathcal{F}}(m).\ h^{UF} \sqsupseteq_c h'^{UF} \wedge \exists \pi \in [\![f(\boldsymbol{in})]\!].\ term(\pi)}{\forall \pi' \in [\![f'(\boldsymbol{in})]\!].\ term(\pi')}\ . \tag{6.13}$$

Proof. Consider the finite computation π provided by the premise. By O1, $depth(\pi)$ is bounded by a finite value. Let $d = depth(\pi)$. The proof is by induction on d.

Base: For $d = 1$, π does not contain any call statements. Falsely assume $\exists \pi' \in [\![f'(\boldsymbol{in})]\!].\ \neg term(\pi')$, which by O2 implies that π'^1 contains some call statement $g'(\boldsymbol{in_g})$. Since the premise of Lemma 6.3.1 holds ($f^{UF} \sqsupseteq_c f'^{UF}$ holds owing to the premise of (6.13)), π must contain a matching call statement $g(\boldsymbol{in_g})$, where $\langle g, g' \rangle \in map_{\mathcal{F}}$, in contradiction to the assumption that there are no call statements in π. Hence, $term(\pi')$ holds.

Step: Assume that the rule holds up to a given d and the premise holds at $d+1$ for a call $f(\boldsymbol{in})$. Let $\pi \in [\![f(\boldsymbol{in})]\!]$ be a computation which satisfies $depth(\pi) \leq d+1$, and hence, $term(\pi)$ holds (by O1). We now prove that the consequent of the rule is true for $d+1$.

Falsely assume that there is a computation $\pi' \in [\![f'(\boldsymbol{in})]\!]$, where $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$, such that $\neg term(\pi')$ holds. This implies that $f'(\boldsymbol{in})$ must make some call $g'(\boldsymbol{in_g})$ which does not return. Since the premise of Lemma 6.3.1

holds, π contains a matching call statement $g(\boldsymbol{in_g})$, where $\langle g, g' \rangle \in map_{\mathcal{F}}$. Now note that $depth(\pi) \leq d+1$ implies that for any subcomputation π_g of π such that $\pi_g \in [\![g(\boldsymbol{in_g})]\!]$, $depth(\pi_g) \leq d$. By the induction hypothesis, (6.13) holds up to d and, therefore, the following is true:

$$\forall \pi'_g \in [\![g'(\boldsymbol{in_g})]\!].\, term(\pi'_g)\ .$$

Consequently, the supposedly non-terminating call of $g'(\boldsymbol{in_g})$ in π' must have returned, which is a contradiction. Hence, $term(\pi')$ holds. □

Theorem 6.3.1. (TERM) is sound.

Proof. Falsely assume that the premise of rule (TERM) holds

$$\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\, \left(term(f) \wedge f^{UF} \sqsupseteq_c f'^{UF}\right)\ ,$$

but the consequent

$$\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\, term(f')$$

does not. This means that there exists $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$, an input $\boldsymbol{in}$, and computations $\pi \in [\![f(\boldsymbol{in})]\!]$ and $\pi' \in [\![f'(\boldsymbol{in})]\!]$ such that $term(\pi)$ and $\neg term(\pi')$. The former implies by O1 that $depth(\pi)$ is bounded by some finite value d. The premise of (6.13) now holds, and hence by Lemma 6.3.2, all the computations of $f'(\boldsymbol{in})$, including π', must be finite, which contradicts our assumption that π' is infinite. Hence, $term(f')$ must hold. □

6.3.1 Proof of (TERM$^+$).

The outline of the proof is the following:

- In Theorem 6.3.2 we will falsely assume that the premise of (TERM$^+$)

  $$\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).(term(f) \wedge f^{UF} \sqsupseteq_c f'^{UF}) \wedge$$
  $$\forall \langle g, g' \rangle \in map_{\mathcal{F}}.\ ((g \in C(m) \wedge g' \in C(m')) \rightarrow m\text{-}term(g, g'))$$

  holds, but there exists computations $\pi \in [\![f(\boldsymbol{in})]\!]$ and $\pi' \in [\![f'(\boldsymbol{in})]\!]$

such that $term(\pi)$ and $\neg term(\pi')$, which contradicts the consequent of that rule.

- In Lemma 6.3.3 we show that $\neg term(\pi')$ can only be caused by an inner call statement in ${\pi'}^1$ (i.e., a call to a function in m'), which does not return.
- In Lemma 6.3.4 we prove, in contrast, that all inner calls of π' must terminate if $term(\pi)$ holds.

The proof is based on the following inference rule, which holds for any given $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$ called with the same input $\boldsymbol{in}$:

$$\frac{\begin{array}{l}\forall \langle h, h' \rangle \in map_{\mathcal{F}}(m).\ h^{UF} \sqsupseteq_c {h'}^{UF} \wedge \\ \forall \langle g, g' \rangle \in map_{\mathcal{F}}.\ (g \in C(m) \wedge g' \in C(m')) \rightarrow m\text{-}term(g, g') \wedge \\ \exists \pi \in [\![f(\boldsymbol{in})]\!].\ term(\pi)\end{array}}{\forall \pi' \in [\![f'(\boldsymbol{in})]\!].\ term(\pi')}\ . \tag{6.14}$$

Note that the premise simply strengthens the premise of rule (TERM$^+$) with the third line, requiring that there exists a finite computation in $f(\boldsymbol{in})$. In order to show the soundness of (6.14), we need first to prove the following lemma:

Lemma 6.3.3. Consider a pair of MSCCs $\langle m, m' \rangle \in map_{\mathcal{M}}$ and a pair of functions $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$ called with an input $\boldsymbol{in}$ which satisfy the premise of (6.14). Then

$$\exists \pi' \in [\![f'(\boldsymbol{in})]\!].\ \langle f, f' \rangle \in map_{\mathcal{F}}(m) \wedge \neg term(\pi')$$

implies the following:

1. ${\pi'}^1$ must contain exactly one call statement which does not return, and
2. the called function must belong to m'.

Proof. The first item is an immediate consequence of observation O2. It is left to prove that the non-returning function must belong to m'. Let $g'(\boldsymbol{in_g})$ be this non-returning call. Falsely assume $g' \in C(m')$. This call is made in

$f'(\boldsymbol{in})$. $term(\pi)$ and $f^{UF} \sqsupseteq_c f'^{UF}$ imply by Lemma 6.3.1 that $g(\boldsymbol{in_g})$ is called in π, where $\langle g, g' \rangle \in map_{\mathcal{F}} \wedge g \in C(m)$. This call must terminate because of $term(\pi)$. The latter fact and $m\text{-}term(g, g')$ imply that all computations of $g'(\boldsymbol{in_g})$ are finite, in contradiction to the assumption that the call of $g'(\boldsymbol{in_g})$ does not return. Hence, the assumption $g' \in C(m')$ was wrong, which implies $g' \in m'$. □

Now we can prove the soundness of (6.14). The proof follows similar lines to those used in the proof above of (TERM). Specifically, we will extend Lemma 6.3.2 to cases in which there are mutually-terminating calls outside the MSCC. Correspondingly, we define

$$depth_m(\pi) \doteq max\{|s| \mid s \in \mathcal{S}_m(\pi)\} , \tag{6.15}$$

where, recall, $\mathcal{S}_m(\pi)$ denotes the subset of stacks in $\mathcal{S}(\pi)$ that consist solely of functions in a given MSCC m.

Lemma 6.3.4. Rule (6.14) is sound.

Proof. For the finite computation π guaranteed by the third line of the premise, let $d = depth_m(\pi)$. The proof is by induction on d.

Base: For $d = 1$, π does not contain any inner calls statements, i.e., any calls of functions of m. Falsely assume $\neg term(\pi')$, which implies by Lemma 6.3.3 that π'^1 contains some call statement $g'(\boldsymbol{in_g})$ such that $g' \in m'$. Since the premise of Lemma 6.3.1 holds ($f^{UF} \sqsupseteq_c f'^{UF}$ holds owing to the premise of (6.14)), π must contain a matching call statement $g(\boldsymbol{in_g})$, where $\langle g, g' \rangle \in map_{\mathcal{F}}(m)$, in contradiction to the assumption that there are no inner call statements in π. Hence, $term(\pi')$ holds.

Step: Assume that the rule holds up to a given d and the premise holds at $d + 1$ for a call $f(\boldsymbol{in})$. Let $\pi \in [\![f(\boldsymbol{in})]\!]$ be a computation which satisfies $depth_m(\pi) \leq d+1 \wedge term(\pi)$. We now prove that the consequent of the rule is true for $d + 1$. Falsely assume:

$$\exists \pi' \in [\![f'(\boldsymbol{in})]\!].\ \langle f, f' \rangle \in map_{\mathcal{F}}(m) \wedge \neg term(\pi') .$$

Lemma 6.3.3 implies that $f'(\boldsymbol{in})$ must make some call $g'(\boldsymbol{in_g})$, where $g' \in m'$, which does not return. Since the premise of Lemma 6.3.1 holds, π contains

a matching call statement $g(\boldsymbol{in_g})$, where $\langle g, g' \rangle \in map_{\mathcal{F}}(m)$. Now note that $depth_m(\pi) \leq d + 1$ and $g \in m$ imply that for any subcomputation π_g of π such that $\pi_g \in [\![g(\boldsymbol{in_g})]\!]$, $depth_m(\pi_g) \leq d$. By the induction hypothesis, (6.14) holds up to d and, therefore,

$$\forall \pi'_g \in [\![g'(\boldsymbol{in_g})]\!].\, term(\pi'_g)$$

holds. Consequently, the supposedly non-terminating call of $g'(\boldsymbol{in_g})$ in π' must have returned, which is a contradiction. Hence, $term(\pi')$ holds. □

Theorem 6.3.2. (TERM$^+$) is sound.

Proof. Assume that the premise of rule (TERM$^+$) holds. Consider $\langle f, f' \rangle \in map_{\mathcal{F}}(m)$ called with the same argument $\boldsymbol{in}$. $term(f)$ implies that every computation $\pi \in [\![f(\boldsymbol{in})]\!]$ is finite. Falsely assume $\neg term(f')$, which implies that there is an infinite computation $\pi' \in [\![f'(\boldsymbol{in})]\!]$. The premise of (6.14) now holds, and, hence, by Lemma 6.3.4, all the computations of $f'(\boldsymbol{in})$, including π', must be finite, which contradicts our assumption that π' is infinite. Hence, $term(f')$ must hold. □

Chapter 7

Experience and conclusions

We implemented Alg. 2 in RVT [1, 24], and tested it with many small programs and one real software project. Here we describe the latter.

We tested our tool on the open source project BETIK [2], which is an interpreter for a scripting language. The code has 2000 – 2500 lines (depending on the version). It has many loops and recursive functions, including mutual recursion forming an MSCC of size 14. We compared eight consecutive versions of this program from the code repository, i.e., seven comparisons. The amount of changes between the versions varied with an average of 3–4 (related) functions. Somewhat to our surprise, many of the changes do *not* preserve termination behavior in a free context, mostly because these functions traverse global data structures on the heap.

In five out of the seven comparisons, RVT discovered correctly, in less than 2 minutes each, that the programs contained mapped functions that do not mutually terminate. Fig. 7.1 displays two versions of a function called INT_VALUE, which receives a pointer to a node in a syntax tree. The old version compared the type of the node to several values, and if none of them matched it simply returned the input node. In the new code, a 'default' branch was added, that called INT_VALUE' with the node's subtype. In an arbitrary context, it is possible that the syntax 'tree' is not actually a tree, rather a cyclic graph, e.g., owing to data aliasing. Hence, there is a context in which the old function terminates whereas the new one is trapped in infinite recursion.

```
value_t *INT_VALUE(value_t *v) {
  switch(v→type) {
    case 0:
      v = ...;
      break;
    ⋮
    case N:
      v = ...;
      break;
  }
  return v;
}
```

```
value_t *INT_VALUE'(value_t *v') {
  switch(v'→type) {
    case 0:
      v' = ...;
      break;
    ⋮
    case N:
      v' = ...;
      break;
    default:
      v' = INT_VALUE'(v'→subvalue);
  }
  return v';
}
```

Figure 7.1: Two possibly non-mutually terminating versions of INT_VALUE.

An additional example in which mutual termination is not preserved is the code presented in Fig. 7.2. It contains a function called PARSE_FUNCCALL that receives a pointer to a function node and processes it according to the function name. The newer version handles an additional option for the function name and calls a new function LIST_SET_ITEM. The latter receives a pointer to a list, traverses it from the list head, and modifies data of some of its items. The traversal ends upon reaching a 'NULL' node, but may not terminate in an arbitrary context, e.g., when the list is cyclic. The new function is not mapped to any function in the old code. Indeed, Alg. 1 and Alg. 2 abort in line 5 when encountering this function.

In the remaining two comparisons RVT marked correctly, in less than a minute each, that all mapped functions are mutually terminating.

7.1 Conclusion and future research.

Checking mutual termination for two whole programs is a crucial sub-task in proving their *full equivalence*, which means that they are both partially

```
function PARSE_FUNCCALL(fcall_t *f)
{
    if (!strcmp(f→func_name, "env")) {
        ...
    } else if (!strcmp(..., "len")) {
        ...
    }
}
```

```
function PARSE_FUNCCALL'(fcall_t *f')
{
    if (!strcmp(f'→func_name, "env")) {
        ...
    } else if (!strcmp(..., "len")) {
        ...
    } else if (!strcmp(..., "set")) {
        list_t *list' = ...;
        LIST_SET_ITEM'(list', ...);
    }
}

function LIST_SET_ITEM'(list_t *list', ...)
{
    listitem_t *item' = list'→head;
    while(item' != NULL) {
        ...
        item' = item'→next;
    }
}
```

Figure 7.2: Two possibly non-mutually terminating versions of PARSE_FUNCCALL and a newly introduced non-mapped function LIST_SET_ITEM'.

equivalent and mutually terminating [30, 35]. Particularly, listing the functions that changed their termination behavior owing to code updates may be valuable to programmers. In this research several steps have been made towards achieving these goals. We showed a proof rule for mutual termination. We presented a bottom-up decomposition algorithm for handling entire programs. This algorithm calls a model-checker for discharging the premise of the rule. Our prototype implementation of this algorithm in RVT is the first to give an automated (inherently incomplete) tool to the mutual termination problem.

The limitations of the tool are inherited from RVT itself, namely the fact that it did not cover various features of C, for instance, unions, abnormal castings, etc. It is important to note for empirical evaluation that the work for this thesis has included a large engineering effort for fulfilling the aim of proving the mutual termination of real programs. We have supported many of those uncovered features, including as important ones as pointers, some cases of casting to $void*$ and back. Despite that at the current stage, RVT is still not utterly robust, its quality has significantly risen.

An urgent conclusion from our experiments is that checking mutual termination under free context is possibly insufficient, especially when it comes to programs that manipulate a global structure on the heap. Developers would also want to know whether their programs mutually terminate under the context of their specific program. This is not an easy modification to our algorithm, because the decomposition is based on a bottom-up traversal, hence ignoring the context. Perhaps, a method can be found that propagates information down the call graphs that can restrict the context in which pairs of functions are checked for mutual termination.

Another direction for future research is to improve the information that is propagated *upwards*. Specifically, it would be nice to refine the abstraction imposed by the use of uninterpreted functions. Adding *function summaries* [12] that provide more information about what these functions do (thus making them more interpreted) is bound to make the method more complete.

A third direction is to interface RVT with an external tool that checks termination: in those cases that they can prove termination of one side but

not of the other, the inference rules of Chapter 6 can be useful for proving termination in the other side. The solution suggested there can be fully automated unlike many existing approaches for proving termination that rely on searching for well-founded sets, which can sometimes be tricky. Knowing that a pair of functions terminate (not just mutually terminate) can also be beneficial because in such a situation they should be excluded from call-equivalence checks of their callers. Also it seems plausible to develop methods for proving termination by using the rule (M-TERM$^+$). One needs to find a variant of the input program that on the one hand is easier to prove terminating, and on the other hand is still call-equivalent to the original program.

An orthogonal direction for improving RVT is related to performance. The current implementation checks the call-equivalence of MSSC pairs iteratively one after another. However, decomposition usually creates numerous pairs whose call-equivalence checks do not require establishing mutual termination for a majority of the other pairs. RVT could execute such independent checks as parallel tasks and thus boost its performance on modern multicore hardware.

Finally, it seems essential to develop algorithms for proving mutual termination for multithreaded programs. This can be a tough challenge owing to non-determinism in scheduling of threads, possibility for deadlocks, data races, and other problems that do not occur in a single-threaded program. A theoretical research into the related problem of partial equivalence between multithreaded programs has recently appeared in [8].

Appendix

A.1 A proof of undecidability of the mutual termination problem

Theorem A.1.1. The mutual termination problem is undecidable.

Proof. Falsely assume that the mutual termination problem is decidable, i.e., it is possible to write a Boolean function CheckMutualTerm, which using only finite amount of time determines whether two given functions are mutually terminating. Let $\mathcal{E}_f$ denote a function having the same prototype as a function f and immediately terminating without computing anything. Consider the following function:

function CheckTermination(A function: f)
  return CheckMutualTerm(f, $\mathcal{E}_f$);

Since $\mathcal{E}_f$ is a terminating function, CheckTermination determines during finite amount of time whether a given function f is terminating, i.e., it decides the halting problem. But the latter problem was proven undecidable in [37]. Hence, the assumption that the mutual termination problem is decidable was wrong. □

Bibliography

[1] http://ie.technion.ac.il/∼ofers/rvt.html.

[2] Available from `http://code.google.com/p/betik`.

[3] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99:118–133, 1928.

[4] Frances E. Allen. Control flow analysis. SIGPLAN Notices 5(7), pages 1–19, 1970.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. In *In Conference Record of the 15th ACM Symposium on Principles of Programming Languages*, pages 1–11. ACM, New York, January 1988.

[6] Clara Bertolissi. *The graph rewriting calculus: properties and expressive capabilities*. PhD thesis, L'Institut National Polytechnique de Lorraine, 2005.

[7] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *CAV*, pages 491–504, 2005.

[8] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Regression verification for multi-threaded programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'12)*, pages 119–135. Springer-Verlag, 2012.

[9] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.

[10] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *SAS*, pages 87–101, 2005.

[11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, 2011.

[12] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577, 2011.

[13] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *JSAT*, 2(1-4):1–26, 2006.

[14] Dima Elenbogen, Shmuel Katz, and Ofer Strichman. Proving mutual termination of programs. In *Hardware and Software: Verification and Testing (HVC'12)*, pages 24–39. Springer-Verlag, May 2013.

[15] S. Even. *Graph Algorithms*. Computer Science Press, 1979.

[16] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Computer Systems*, 9(3):319–349, 1987.

[17] C.N. Fisher and R.J.L. Blanc. *Crafting a Compiler*. The Benjamin-Cummings Series in Computer Science. Benjamin/Cummings, 1988.

[18] R.W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[19] Lynn E. Garner. On the Collatz 3n + 1 algorithm. *Proceedings of the American Mathematical Society*, 82(1):19–22, 1981.

[20] Benny Godlin. Regression verification: Theoretical and implementation aspects. Master's thesis, Technion, Israel Institute of Technology, 2008.

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

[22] Benny Godlin and Ofer Strichman. Regression verification. Technical Report IE/IS-2011-02, Technion, 2011. http://ie.technion.ac.il/tech_reports/1306207119_j.pdf.

[23] Benny Godlin and Ofer Strichman. Regression verification – proving equivalance of similar programs. *Journal of Software Testing, Verification & Reliability*, 23(3):241 – 258, 2013.

[24] Benny Godlin and Ofer Strichman. Regression verification. In 46^{th} *Design Automation Conference (DAC)*, 2009.

[25] M.S. Hecht. *Flow Analysis of Computer Programs*. North Holland, 1977.

[26] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Computer Systems*, 12(1):26–61, 1990.

[27] Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebelo. Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research, 2010.

[28] Daniel Kroening and Ofer Strichman. *Decision procedures – an algorithmic point of view*. Theoretical computer science. Springer-Verlag, May 2008.

[29] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[30] D.C. Luckham, D.M.R. Park, and M.S. Paterson. On formalized computer programs. *J. Comp. Systems Sci.*, 4(3):220–249, 1970.

[31] Zohar Manna and John McCarthy. Properties of programs and partial function logic. *Machine Intelligence*, 5:27–37, 1969.

[32] F. Nielson. Program transformation in a denotational setting. *ACM Trans. Prog. Lang. Sys.*, 7:359–379, 1985.

[33] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis.* Springer-Verlag, Berlin, 2005.

[34] R. Péter. Konstruktion nichtrekursiver funktionen. In *Math Annalen*, volume 111, pages 42–60, 1935.

[35] Terrence W. Pratt. Kernel equivalence of programs and proving kernel equivalence and correctness by test cases. *International Joint Conference on Artificial Intelligence*, 1971.

[36] B.K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conference Record of the 15th ACM Symposium on Principles of Programming Languages*, pages 12–27. ACM, New York, January 1988.

[37] C. Strachey. An impossible program. *Computer Journal*, 1965.

[38] R.A. Wilhelm, D.A. Maurer, and D. Maurer. *Compiler Design.* International Computer Science Series. Addison-Wesley, 1995.

עשויים לדעת האם תוכניותיהם עוצרות בהקשר הספציפי שתחתיו הן אמורות לרוץ. כנראה, יש לפתח שיטה להפצת מידע למטה בגרף הקריאות לשם הגבלת ההקשר שתחתיו נבדקת עצירה הדדית לזוג פונקציות.

אנחנו מציעים כיוונים אפשריים נוספים להמשך מחקר עתידי בנושא. ראשית, מומלץ לעדן יותר את הקירוב המוטל על ידי השימוש בפונקציות בלתי מפורשות. שנית, כדאי ש-RVT יתממשק לכלים חיצוניים לבדיקת עצירה של פונקציות. אם ידוע שאחת הגירסאות תמיד עוצרת, אז ניתן להפעיל את הכלל $TERM^+$ בעל הנחה חלשה יותר לשם בדיקה האם גם הגירסה השנייה עוצרת. כמו כן, ניתן להסיר זוג פונקציות שידועות כעוצרות תמיד מהבדיקות הנדרשות להוכחת העצירה ההדית של תוכניותיהן. ולבסוף, תחום מאתגר להמשך המחקר הוא בעית העצירה ההדדית של תוכניות רבות תהליכים (multithreaded).

במקרה שבו המנשקים של שתי הפונקציות נבדלים, קרי, הן מקבלות מספר פרמטרים שונה. כמו כן, ניתן לדייק את ההתנהגויות של f^{UF} ו-f'^{UF} במידה וידוע כי f ו-f' קוראות לפונקציות h ו-h' (בהתאמה) אשר ממופות ב-map_F והינן *שקולות חלקית* (partially equivalent), כלומר כאלה שבהינתן אותם קלטים כל זוג חישובים סופיים בשתיהן מחזיר אותם פלטים. במקרה זה ניתן לעדן את הקירוב המוטל על ידי החלפת הקריאות הללו בקריאות לפונקציות בלתי מפורשות UF_h ו-$UF_{h'}$ זהות. גם מידע חלקי אודות השקילות החלקית של h ו-h' מסייע לעדן יותר את UF_h ו-$UF_{h'}$.

בהסתמך על כללי ההיסק המתוארים לעיל, אנחנו מציגים אלגוריתם לפירוק בעית אימות של תוכניות שלמות לבעית הוכחת העצירה ההדדית של זוגות של פונקציות בודדות. בהתחלה האלגוריתם בונה גרף קריאות בין הפונקציות של כל אחת מהתוכניות (נקרא להם G_0 ו- G_1). מכל אחד מהם הוא בונה אחרי כן גרף מכוון חסר מעגלים של רכיבים קשירים היטב מקסימליים ומנסה למפות בין הרכיבים הנ"ל של התוכנית הראשונה לבין רכיבי התוכנית השנייה. אם הוא הצליח, הוא מטפל בזוגות ממופים של הרכיבים הנ"ל באופן סידרתי החל בעלים וכלה בזוג השורשים כאשר התנאי לבחירת זוג רכיבים לטיפול הוא שכל בניהם כבר טופלו. לכל זוג רכיבים ממופים m ו-m' האלגוריתם בוחר קבוצה לא ריקה S של זוגות של פונקציות ששייכות ל-m ו-m' וממופות ב-map_F. על S לכסות את כל המעגלים ב-G_0 וב-G_1 הנוצרים בין הפונקציות השייכות ל-m ו-m' (בהתאמה). עדיף שהקבוצה S תהיה מקסימלית בגודלה. אנחנו מתארים שיטה לבחירה דטרמיניסטית של הקבוצה. לאחר מכן, האלגוריתם מייצר עותק לכל אחת משתי התוכניות המקוריות. קריאות לפונקציות h ו-h' שממופות ב-map_F מתעדכנות בעותקים כדלהלן:

- במידה ואחד התנאים הבאים מתקיים:
  - זוג h, h' שייך ל-S, או
  - העצירה ההדדית של h ו-h' כבר נקבעה,

  הן מוחלפות בקריאות לפונקציות בלתי מפורשות UF_h ו- $UF_{h'}$ (בהתאמה);
- אחרת, הן מוּסָרוֹת והקוד של h ו-h' נשזר במקומן (בהתאמה).

באופן זה מתקבלות f^{UF} ו-f'^{UF} עבור כל זוג פונקציות ממופות f ו-f' (בהתאמה) ששייך ל-S. אם עבור כל זוג f ו-f' ששייך ל-S נקבע כי הוא מקיים את הנחת כלל ההיסק $M\text{-}TERM^+$ כאשר נבדקת עצירה הדדית (או, לחילופין, את הנחת כלל ההיסק $TERM^+$ כאשר נבדקת עצירה), אזיי כל הזוגות האלה הם זוגות של פונקציות עוצרות הדדית (או, לחילופין, עוצרת). זה מסביר את העדיפות לבחירת קבוצה S גדולה ככל שניתן, הרי שהאלגוריתם משתדל להוכיח עצירה הדדית עבור מספר רב ככל שניתן של זוגות פונקציות. קיום ההנחה של כלל ההיסק המתאים נבדק בעזרת כלי ממוחשב לבדיקת תכונות של תוכניות חסומות בשפת תיכנות C בשם CBMC.

מימשנו אב טיפוס של האלגוריתם הנ"ל לבדיקת סיום הדדי עבור תוכניות כתובות בשפת C. המימוש הוא ראשון מסוגו שעוסק בבעית העצירה ההדדית. הוא מושתת על כלי לאימות נסיגתי (regression verification) בשם RVT. לטובת הערכה אימפירית של עבודתנו יצוין כי תוך כדי המימוש הושקע מאמץ הנדסי גדול למען התמודדות עם תוכנות אמיתיות. כתוצאת לוואי, שיפרנו משמעותית את האיכות של RVT ותמכנו במספר תכונות מתקדמות של שפת C שלא נתמכו בעבר.

הרצנו את אב הטיפוס הן על כמה תוכניות פשוטות והן על מספר גירסאות פיתוח עוקבות של פרויקט תוכנה אמיתי. אנחנו מדווחים על המסקנות שהתקבלו בעקבות הריצות הללו. המסקנה המיידית מהבדיקות היא שככל הנראה בדיקת עצירה הדדית תחת הקשר חופשי אינה מספיקה. מפתחים

תקציר

שתי תוכניות נקראות *מסתיימות (עוצרות) הדדית* (mutually terminating) כאשר התוכנית הראשונה עוצרת אם ורק אם השנייה עוצרת בהינתן אותם קלטים. בעית העצירה ההדדית של תוכניות לא זכתה במחקר רחב לעומת בעית העצירה של תוכנית בודדת. המטרה העיקרית של המחקר היא פיתוח ויישום שיטה להוכחת עצירה הדדית של זוג נתון של פונקציות בהיקראן תחת הקשר חופשי. בהינתן שתי תוכניות *קרובות*, כגון שתי גירסאות עוקבות של פרויקט תוכנה, הוכחת העצירה ההדדית של זוג הפונקציות הראשיות שלהן, בהתאמה, מהווה הוכחה לעצירה ההדדית של התוכניות עצמן. קביעה האם שתי תוכנית עוצרות הדדית אינה מושתתת על הוכחה שסידרת חישובים שמבצעת כל תוכנית ניתנת למיפוי ליחס מבוסס היטב (well-founded set). לכן אנחנו משערים כי היא משימה קלה יותר לחישוב אוטומטי מאשר קביעה האם כל אחת מהתוכניות המושוות עוצרת. יתר על כן, קיימות תוכניות שאינן אמורות לעצור עם כל הקלטים על פי התיכון שלהן, כגון תוכניות תגובתיות (reactive programs).

אנחנו מתמקדים בהוכחת סיום הדדי של שתי תוכניות דטרמיניסטיות (קרי, ללא תתי-תהליכים (single-threaded) וללא אי-דטרמיניזם מובנה) P_0 ו- P_1. לשם כך נדרשות שלוש פעולות מקדימות:

א) המרת כל לולאה לרקורסית זנב. לאחר שלב זה, הסיבה הבלעדית שנותרה לריצה אינסופית של תוכנית היא קריאות רקורסיביות.

ב) צירוף של משתנים גלובלים שפונקציה ניגשת אליהם לרשימת הפרמטרים הפורמלית שלה. קריאות לפונקציה זאת מתעדכנות בהתאם.

ג) בנית מיפוי map_F בין הפונקציות של P_0 ו- P_1. המיפוי יכול להיות חלקי, אם כי חשוב שיכסה את כל המעגלים בגרף הקריאות בין הפונקציות של כל אחת מהתוכניות. הקוד של פונקציה שלא ממופית נשזר (inlined) במקומות שבהם הפונקציה נקראת.

לכל פונקציה ממופית g אנחנו מגדירים פונקצית קירוב-מעל (overapproximation) g^{UF} תוך החלפת כל קריאה לפונקציה h בקריאה לפונקציה בלתי מפורשת (uninterpreted function) מתאימה (נקרא לה בשם UF_h). בעזרת הגדרה זאת אנחנו מציעים כלל היסק בשם $M\text{-}TERM^+$ להוכחת עצירה הדדית של זוג נתון של פונקציות ממופות f ו-f'. הנחת הכלל דורשת כי בהינתן קלט שרירותי ***in***, $f^{UF}(\textbf{\textit{in}})$ ו-$f'^{UF}(\textbf{\textit{in}})$ קוראות לפונקציות בלתי מפורשות שמחליפות זוג פונקציות שממופות ב-map_F עם אותם ארגומנטים. קיום ההנחה ניתן לבדיקה ממוחשבת בעזרת כלים אוטומטיים לבדיקת תכונות (model checkers). אם ההנחה מתקיימת עבור כל זוג של הפונקציות הממופות, אזי הם כולם זוגות של פונקציות עוצרות הדדית.

גירסה נוספת בשם $TERM^+$ של כלל ההיסק הנ"ל בעלת הנחה חלשה יותר מאפשרת הוכחת עצירה (לא רק סיום הדדי) של פונקציה f' של אחת הגירסאות של תוכנית נתונה כאשר *ידוע* כי הפונקציה f הממופת ל-f' בגירסה השנייה עוצרת בהינתן כל קלט. הנחת הכלל דורשת כי f עוצרת לכל קלט וכי בהינתן קלט שרירותי ***in***, לכל קריאה ב-$f'^{UF}(\textbf{\textit{in}})$ לפונקציה בלתי מפורשת $UF_{h'}$ קיימת קריאה עם אותם ארגומנטים לפונקציה בלתי מפורשת UF_h ב-$f^{UF}(\textbf{\textit{in}})$, כאשר f ו-f' וכן h ו-h' הן (בהתאמה) זוגות של פונקציות שממופות ב-map_F. אם ההנחה מתקיימת עבור כל זוג של הפונקציות הממופות, אזיי כל הפונקציות הממופות של התוכנית השניה עוצרות.

בעית העצירה ההדדית אינה כריעה באופן כללי. לכן הפתרון שלנו בהיותו נאות לא יכול להיות שלם. אף על פי כן אנחנו מציעים מספר שיטות בכדי להילחם באי השלמות הטבועה בגישתנו, כולל בין היתר

המחקר נעשה בהנחיית פרופ"ח עופר שטריכמן ובהנחייה משותפת של פרופ' שמואל כ"ץ בפקולטה למדעי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי. חומר זה מבוסס על מחקר שמומן על־ידי מעבדת מחקר של חיל האוויר האמריקאי בכפוף להסכם מספר FA6003־1־11־5558. ממשלת ארצות הברית רשאית להעתיק ולהפיץ העתק של חומר זה לשימוש ממשלתי תוך התעלמות מכל הערה אודות זכויות יוצרים.

הוכחת סיום הדדי של תוכניות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

דימה אלנבוגן

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
אייר ה'תשע"ד חיפה מאי 2014

הוכחת סיום הדדי של תוכניות

דימה אלנבוגן