Proving Mutual Termination of Programs

Dima Elenbogen[1] Shmuel Katz[1] Ofer Strichman[2]

[1] CS, Technion, Haifa, Israel. {katz,edima}@cs.technion.ac.il
[2] Information Systems Engineering, IE, Technion, Haifa, Israel
ofers@ie.technion.ac.il

Abstract. Two programs are said to be *mutually terminating* if they terminate on exactly the same inputs. We suggest a proof rule that uses a mapping between the functions of the two programs for proving mutual termination of functions f, f'. The rule's premise requires proving that given the same arbitrary input $\boldsymbol{in}$, $f(\boldsymbol{in})$ and $f'(\boldsymbol{in})$ call mapped functions with the same arguments. A variant of this rule with a weaker premise allows to prove termination of one of the programs if the other is known to terminate for all inputs. 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 rules.

1 Introduction

Whereas termination of a single program has been widely studied (e.g., [9, 6, 4, 7]) 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 *by design* the program does not terminate on all inputs. 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 [12] and coined *mutual termination*, can easily be proven undecidable as can be seen 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 a well-founded set. 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 Sec. 3, for example, we show how to prove mutual termination of two versions of the famous Collatz's $3x+1$ problem [10]; 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, 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 [5] in our case). Each such program has the same structure: it calls the two compared functions sequentially with the same nondeterministic 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. Partial equivalence was studied in [12, 14] and is implemented in RVT [14] and Microsoft's SymDiff [15]. We also implemented our algorithm in RVT, which enables us to gain this information in a preprocessing step.

To summarize our contributions in this paper, we present a) a proof rule for inferring mutual termination of recursive (and mutually-recursive) functions at the leaves of their respective call graphs, b) an extension of the first rule that applies also to internal nodes in the call graphs, and c) a proof rule for inferring *termination* (not mutual termination) in case the other function is known to be terminating. More importantly, we show how these rules can be applied to whole programs via a bottom-up decomposition algorithm, and report on a prototype implementation of this algorithm – the first to deal with the mutual termination problem.

2 Preliminaries

Our goal is to prove mutual termination of pairs of functions in programs that are assumed to be deterministic (i.e., single threaded and no internal nondeterminism). Formally:

Definition 1 (Mutual termination of functions). *Two functions f and f' are* mutually terminating *if and only if they terminate upon exactly the same inputs.*

By *input* we mean both the function parameters and the global data it accesses, including the heap. Denote by $m\text{-}term(f, f')$ the fact that f and f' mutually terminate.

Preprocessing and mapping As a preprocessing step, all loops are extracted to external recursive functions, as shown in [11]. After this step nontermination can only arise from recursion. In addition, all global variables that are read by a function are added to the end of 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 [14, 15].

As a second step, a *bijective* map $map_{\mathcal{F}}$ between the functions of P and P' is derived. 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 describe later in Sect. 3.

Definitions and notations

- *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[1].

[1] This definition generalized naturally to cases in which g has multiple outputs owing to global data and arguments passed by reference.

The following definition will be used for specifying which functions are associated with the *same* uninterpreted function:

Definition 2 (Partial equivalence of functions). *Two functions f and f' are* partially equivalent *if any two terminating executions 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

$$UF_g = UF_{g'} \Rightarrow (\langle g, g' \rangle \in map_{\mathcal{F}} \wedge p\text{-}equiv(g, g')) \text{ (enforce)} \tag{1}$$

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. We now define:

$$f^{UF} \doteq f[g(expr_{in}) \leftarrow UF_g(expr_{in}) \mid g \text{ is called in } f] \;, \tag{2}$$

where $expr_{in}$ is the expression(s) denoting actual parameter(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.
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.

- *Call equivalence.*
  Let $calls(f(\boldsymbol{in}))$, where $\boldsymbol{in}$ is a vector of actual values, denote the set of function call instructions (i.e., a function name and the actual parameter values) invoked in the body of $f()$ during the execution of $f(\boldsymbol{in})$. Note that $calls(f(\boldsymbol{in}))$ does not include calls from descendant functions and hence also not from recursive calls.
  We can now define:

  Definition 3 (Call-equivalence of functions). *f and f' are* call-equivalent *if and only if*

  $$\forall \langle g,\, g' \rangle \in map_{\mathcal{F}}, in_f, in_g.\; g'(in_g) \in calls(f'(in_f)) \Leftrightarrow g(in_g) \in calls(f(in_f)) \;.$$

  Denote by $call\text{-}equiv(f, f')$ the fact that f and f' are call-equivalent. Note that it is decidable whether f^{UF} and f'^{UF} are call-equivalent, because these are functions without loops and function calls, as explained above.

3 A proof rule

In an earlier publication by the 3rd author [12], there appears a rule for proving mutual termination of individual 'leaf' functions (i.e., that do not call functions other than themselves). 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, which enable us to consider whole programs.

Given a call graph of a general program, a corresponding 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 two compared programs P, P', let map_m be a map between the nodes of their respective MSCC DAGs, which is consistent with map_f. Namely, if $\langle m, m' \rangle \in map_m$, 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, $m' \in map_m$, respectively, that are *leaves* in the MSCC DAGs. Denote by

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

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:

$$\frac{\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ \textit{call-equiv}(f^{UF}, f'^{UF})}{\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ \textit{m-term}(f, f')} \quad (\text{M-TERM}) \tag{3}$$

The premise of (3) is weaker than (hence the rule itself is stronger than) the one suggested in [12], because the latter required the compared functions to be partially equivalent. Furthermore, whereas [12] refers to leaf MSCCs only, later on in this section we generalize (3) so it also applies to non-leaf MSCCs, and hence tackles the general case.

Note that the abstraction of calls with uninterpreted functions is the source of incompleteness. $\textit{call-equiv}(f^{UF}, f'^{UF})$ may fail, 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 parameters. Furthermore, it is possible that the concrete function and its counterpart on the other side 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.

Checking the premise We check the premise of (3) by building a loop- and recursion-free program for each pair of functions that we want to prove call equivalent. 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 Sec. 4.

Let f, f' be simple recursive functions that only call themselves. We associate a set of call instructions with each called function (this set represents $calls(f(\boldsymbol{in}))$. 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 nondeterministic 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.

```
function f(int a)                            function f'(int a')
  int even := 0, ret := 0;                     int t', odd' := 0, ret' := 0;
  if a > 1 then                                if a' ≤ 1 then return ret';
    if ¬(a % 2) then           ▷ even          t' := a'/2;
      a := a/2;                                if a'%2 then                  ▷ odd
      even = 1;                                  a' := 6t' + 4;
    else a := 3a + 1;                            odd' := 1;
    ret := even + f(a);                        else a' := t';
  return ret;                                  ret' := odd' + f'(a');
                                               return ret';
```

Fig. 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. Note than when a' is odd, $a'/2 = (a' - 1)/2$, and hence $6(a'/2) + 4 = 3a' + 1$.

Example 1. Consider the two variants of the Collatz ("$3x + 1$") program [10] in Fig. 1[2], 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. On the other hand proving mutual termination of the two variants given here is easy. The comparison is not fair, however, because our decision procedure assumes finite types: we target C programs. But as we show in the full version of this article [1], it is solvable even when the input parameter is an unbounded integer, using a decision procedure for linear arithmetic.

The definitions of f^{UF}, f'^{UF} appear at the top part of Fig. 2. The middle part of the same figure 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 parameters. Note that it records the set of call instructions in the array `params`.

Note that in this case f, f' are not partially equivalent, and therefore according to (1) 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 inputs they do not necessarily return the same nondeterministic value. We defer the presentation of the case in which the functions are known to be partially equivalent to Sec. 4. □

What if the mapping is not bijective, or if some of the pairs 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

[2] In the pseudocode we use the convention by which % is the modulo operator.

```
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';
```

```
function UF(function index g, input parameters in)
    if in ∈ params[g] then return the output of the earlier call UF(g, in);
    params[g] := params[g] ⋃ in;
    return a nondeterministic value;

function MAIN
    in = nondet(); f^UF(in); f'^UF(in);
    assert(params[f] = params[f']);           ▷ checks call equivalence
```

Fig. 2. The flat program that we generate and then verify its assertion, given the two functions of Fig. 1 .

regard to proving partial equivalence in a technical report [13]. 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 Sec. 4 uses this observation.

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:

$$\frac{(\forall \langle g, g' \rangle \in map_{\mathcal{F}}.\ (g \in C(m) \wedge g' \in C(m')) \rightarrow m\text{-}term(g, g')) \wedge \\ (\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ call\text{-}equiv(f^{UF}, f'^{UF}))}{\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ m\text{-}term(f, f')}\ (\text{M-TERM}^{+})\ . \tag{4}$$

Recall that (2) prescribes that calls to functions in $C(m)$ and $C(m')$ are replaced with uninterpreted functions in f^{UF}, f'^{UF}.

A full soundness proof of the generalized rule appears in Appendix A, whereas here we only sketch its steps. The proof begins by showing that the premise

implies $\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m)$. *call-equiv*$(f, f')$. Now, falsely assume that there is a pair $\langle f,\, f' \rangle \in map_{\mathcal{F}}(m)$ that is not mutually terminating whereas the premise holds. For some value in, suppose that it is $f(in)$ that terminates, while $f'(in)$ does not. The infinite call stack of $f'(in)$ must contain a call, say from $h'(in_1)$ to $g'(in_2)$, whereas $h(in_1)$ does not call $g(in_2)$ in the call stack of $f(in)$ (assuming $\{\langle g,\, g' \rangle,\, \langle h,\, h' \rangle\} \subseteq map_{\mathcal{F}}$). This contradicts our premise that $\langle h,\, h' \rangle$ are call-equivalent. The argument is a little more involved when there are multiple calls to the same function, and when there are calls to functions in $C(m), C(m')$, but we leave such subtleties to Appendix A.

4 A decomposition algorithm

In this section we present an algorithm for proving mutual termination of full programs. As mentioned in Sec. 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 [13] . 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 [13] is described with a non-deterministic step, and here we suggest a method for determinizing it.

The preprocessing and mapping is as in Sec. 2. 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

The input to Alg. 1 is 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 it succeeded to prove their mutual termination based on (M-TERM$^+$). We now describe the three functions used by this algorithm.

ProveMT. This entry function traverses the call graphs of P, P' bottom-up, each time focusing on a pair of MSCCs. In line 2 it inlines all nonrecursive functions that are not mapped. 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 Sec.3. Line 5 attempts to build map_m (as defined at the top of Sect. 3), 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 nonmapped 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 such that all its children pairs are marked. If m, m' are trivial (see Sec. 3 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 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), because even if $\langle f, f' \rangle$ are not mutually terminating for every input, their callers may still be (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 11 the algorithm chooses non-deterministically a subset S of pairs from $map_{\mathcal{F}}(m)$ that intersects all the cycles in 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 subsection 4.3. 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 13: 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 11) contains only pairs from the same MSCCs as f, f', i.e., if $f \in m$ and $f' \in m'$, then $(g, g') \in S$ 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 after Eq. (2). We will occasionally refer to them as *side 0* and *side 1*. These functions do not have function calls (other than to uninterpreted functions), but may include inlined (nonrecursive) callees that were not proven to be mutually terminating.

The implementations of uf and uf' appear in Fig. 3, 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 *partially_equiv*, uf and uf' emulate the *same* uninterpreted function, i.e.,

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

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

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_m between the nodes of MD and MD' that is consistent with $map_{\mathcal{F}}$; Otherwise **abort**.
6: **while** $\exists \langle m, m' \rangle \in map_m$ not marked *covered* but its children are, **do**
7: Choose such a pair $\langle m, m' \rangle \in map_m$ 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** Select non-deterministically $S \subseteq \{\langle f, f' \rangle \mid \langle f, f' \rangle \in map_{\mathcal{F}}(m)\}$ that intersect every cycle in m and m';
12: **for** each $\langle f, f' \rangle \in S$ **do**
13: **if** $\neg$CallEquiv (isolate(f, f', S)) **then abort**;
14: **for** each $\langle f, f' \rangle \in S$ **do** mark f, f' as *m_term*;

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

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

▷ here add the definitions of uf() and uf'() (see Fig. 3).
$\boldsymbol{in} := nondet(); f^{UF}(\boldsymbol{in}); f'^{UF}(\boldsymbol{in});$
$\forall \langle g, g' \rangle \in map_{\mathcal{F}}$. if g (or g') is called[a] in f (orf') assert($params[g] \subseteq params[g']$);

23: **return** CBMC(δ);

[a] By 'called' we mean that a call appears in the function. It does not mean that there is necessarily an input that invokes this call.

1: **function** UF(function index g, input parameters $\boldsymbol{in}$) ▷ Called in side 0
2: **if** $\boldsymbol{in} \in params[g]$ **then return** the output of the earlier call UF(g, $\boldsymbol{in}$);
3: $params[g] := params[g] \bigcup \boldsymbol{in}$;
4: **return** a non-deterministic output;

5: **function** UF'(function index g', input parameters $\boldsymbol{in'}$) ▷ Called in side 1
6: **if** $\boldsymbol{in'} \in params[g']$ **then return** the output of the earlier call UF'(g', $\boldsymbol{in'}$);
7: $params[g'] := params[g'] \bigcup \boldsymbol{in'}$;
8: **if** $\boldsymbol{in'} \in params[g]$ **then** ▷ $\langle g, g' \rangle \in map_{\mathcal{F}}$
9: **if** $\langle g, g' \rangle$ is marked *partially_equiv* **then**
10: **return** the output of the earlier call UF(g, $\boldsymbol{in'}$);
11: **return** a non-deterministic output;
12: **assert(0)**; ▷ Not call-equivalent: $params[g'] \not\subseteq params[g]$

Fig. 3. 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.

CallEquiv. Our implementation is based on the C model checker CBMC [5], 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 no loops or function calls by construction), with the same nondeterministic 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 [2].

4.2 An example

The following example demonstrates Alg. 1. Consider the call graphs in Fig. 4.

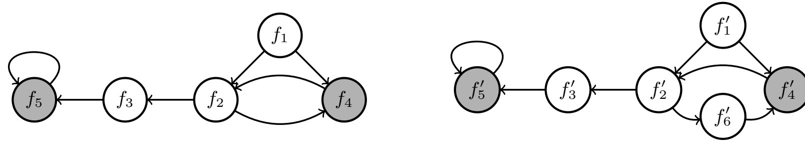

Fig. 4. Call graphs of the input programs P, P'. Partially equivalent functions are gray.

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: $MD = \{\{f_5\}, \{f_3\}, \{f_2, f_4\}, \{f_1\}\}$; $MD' = \{\{f'_5\}, \{f'_3\}, \{f'_2, f'_4, f'_6\}, \{f'_1\}\}$. The MSCC mapping map_m in line 5 is naturally derived from $map_{\mathcal{F}}$.

MSCCs	Pair	Description	Res.
$\{f_5\}$, $\{f_5'\}$	$\langle f_5, f_5'\rangle$	In line 11 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 11 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' ($^{\neq}$), while 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 1. Applying Alg. 1 to the call graphs in Fig. 4. '✓' means that the pair is marked *m_term*, '$✓^c$' that it is marked conditionally (it becomes unconditional once all other pairs in S are marked as well), and '✗' that it is not marked. ($^=$) and ($^{\neq}$) denote that UF, UF' emulate the same, or, respectively, different, uninterpreted functions.

The progress of the algorithm is listed in Table 1. The output in this case, based on assumptions about the results of the checks for call-equivalence that are mentioned in the table, is that the following pairs of functions are marked as *m_term*: $\langle f_5, f_5'\rangle$, $\langle f_2, f_2'\rangle$, $\langle f_4, f_4'\rangle$, and $\langle f_1, f_1'\rangle$.

4.3 Choosing a vertex feedback set deterministically

In line 7 the choice of the set S is nondeterministic. Our implementation determinizes it by solving a series of optimization problems. 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.

Our objective is to find a maximal set S of function pairs, 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. Our implementation solves this optimization problem via a reduction to a pseudo-Boolean formula, which is then solved by MINISAT+ [8]. Each function node g in m (and m') is associated with a Boolean variable v_g, indicating whether it is part of S. The objective is thus to maximize the sum of these variables that are mapped (those that are unmapped cannot be in S anyway). In addition, there is a variable e_{ij} for each

edge (i, j), which is set to true iff neither i nor j is 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., $e_{i,i}$) are set to TRUE as well. We then prevent such cycles by setting them to FALSE. Let *mapped(m)* denote the set of functions in m that are mapped. The problem formulation appears in Fig. 5, and is rather self-explanatory. In case the chosen set S fails (i.e., one of the pairs in S cannot be proven to be mutually terminating), we add its negation (see constraint #6) and repeat.

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

subject to the following constraints, for $M \in \{m, m'\}$:

1. Unmapped nodes are not in S: $\forall g \in (M \setminus \mathit{mapped}(M)).\ \neg v_g$
2. Defining the edges: $\forall \{i, j \mid (i, j) \text{ is an edge in } M\}.\ \neg v_i \wedge \neg v_j \rightarrow\ e_{ij}$
3. Transitive closure: $\forall 0 < i, j, k \leq |M|.\ e_{ij} \wedge e_{jk} \rightarrow e_{ik}$
4. Self loops are not allowed: $\forall 0 < i \leq |M|.\ \neg e_{ii}$
5. Enforce mapping: $\forall \langle g, g' \rangle \in map_{\mathcal{F}}, g \in m.\ v_g \leftrightarrow v'_g$
6. For each failed solution Sl: $\bigvee_{\langle g, g' \rangle \in Sl} \neg v_g$

Fig. 5. A pseudo-Boolean formulation of the optimization problem of finding the largest set of function pairs from m, m' that intersect all cycles in both m and m'.

5 An inference rule for proving termination

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 $term(f)$ to denote that f terminates and

$$\begin{aligned}&\textit{call-contains}(f, f') \doteq\\ &\quad \forall \langle g,\ g' \rangle \in map_{\mathcal{F}}, in_f, in_g.\ g'(in_g) \in calls(f'(in_f)) \Rightarrow g(in_g) \in calls(f(in_f))\ .\end{aligned}$$

Using these predicates, we can now define the rule for leaf MSCCs m, m':

$$\frac{\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ \Big(term(f) \wedge \textit{call-contains}(f^{UF}, {f'}^{UF}\Big)}{\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ term(f')}\ (\text{TERM})\ . \quad (5)$$

Theorem 1. (TERM) *is sound.*

Proof. The proof follows similar lines to that of (M-TERM$^+$). We give a proof sketch. Falsely assume that there is a function f' in m' that does not terminate, whereas for all $\langle g, g' \rangle \in map_{\mathcal{F}}(m)$, *call-contains*$(g, g')$. There exists a value in such that $f'(in)$ does not terminate. The infinite call stack of $f'(in)$ must contain

a call, say from $h'(in_1)$ to $g'(in_2)$, whereas $h(in_1)$ does not call $g(in_2)$ in the call stack of $f(in)$ (assuming $\{\langle g, g'\rangle, \langle h, h'\rangle\} \subseteq map_{\mathcal{F}}$). This contradicts our premise that $call\text{-}contains(h, h')$ is true. $\square$

Note that call-equivalence (Def. 3) is simply bi-directional call-containment. A generalization to non-leaf MSCCs can be done as in (4).

The decomposition algorithm of the previous section (Alg. 1) applies with the following change: the last statement of line 22 (asserting $params[g] \subseteq params[g']$) should be removed. The only assertion that should be verified is thus inside UF (line 12 in Fig. 3), which checks that every call on side 0 is matched by a call on side 1.

6 Experience and conclusions

We implemented Alg. 1 in RVT [14, 2], 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 [3], which is an interpreter for a scripting language. The code has 2 – 2.5 KLOC (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. An example is 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. The full version of this article [1] includes the code of this function as well as an additional example in which mutual termination is not preserved.

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

Conclusion and future research. We showed a proof rule for mutual termination, and a bottom-up decomposition algorithm for handling whole 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) solution to the mutual termination problem.

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. Another 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, we can use the results of Sec. 5 to prove termination in the other side[3]. We can also benefit from knowing that a pair of functions terminate (not just mutually terminate) because in such a case they should be excluded from the call-equivalence check of their callers. Finally, 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.

References

1. Full version available from `http://ie.technion.ac.il/~ofers/atva-full.pdf`.
2. http://ie.technion.ac.il/~ofers/rvt.html.
3. Available from `http://code.google.com/p/betik`.
4. A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, pages 491–504, 2005.
5. E. Clarke and D. 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.
6. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *SAS*, pages 87–101, 2005.
7. B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, 2011.
8. N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. *JSAT*, 2(1-4):1–26, 2006.
9. R. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.
10. L. E. Garner. On the Collatz 3n + 1 algorithm. *Proceedings of the American Mathematical Society*, 82(1):19–22, 1981.
11. B. Godlin. Regression verification: Theoretical and implementation aspects. Master's thesis, Technion, Israel Institute of Technology, 2008.
12. B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.
13. B. Godlin and O. Strichman. Regression verification. Technical Report IE/IS-2011-02, Technion, 2011. http://ie.technion.ac.il/tech_reports/1306207119_j.pdf.
14. B. Godlin and O. Strichman. Regression verification. In 46th *Design Automation Conference (DAC)*, 2009.
15. M. Kawaguchi, S. K. Lahiri, and H. Rebelo. Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research, 2010.

[3] However, as one of our anonymous reviewers pointed out, it is unlikely that it is easy to prove termination for one side and not the other, yet the calls of the first contain the calls of the second.

A Soundness proof for (M-TERM$^+$)

For simplicity, assume that $map_{\mathcal{F}}$ provides a bijective mapping between the functions in m, m' (otherwise we can possibly satisfy this assumption via inlining). In the following by *context* of a function call $f(\boldsymbol{in})$ we mean the execution from the time $f(\boldsymbol{in})$ entered the stack until it left it. We begin with two simple observations:

O1. If there is a call $f(\boldsymbol{in})$ in the context of $f(\boldsymbol{in})$, then $f(\boldsymbol{in})$ does not terminate.
O2. If $f(\boldsymbol{in})$ terminates, then there can only be a finite number of parameter vectors in the context of $f(\boldsymbol{in})$ with which f is called.

Denote by $PV_f[g]$ the *s*et (not a multiset) of parameter vectors with which function g is directly called from function f.

Lemma 1. *If call-equiv*(f^{UF}, f'^{UF})*, then call-equiv*(f, f')*.*

Proof. : We prove the contrapositive of the lemma:

$$\neg call\text{-}equiv(f, f') \rightarrow \neg call\text{-}equiv(f^{UF}, f'^{UF}) \ .$$

Suppose f and f' are called with the same value but for some $\langle g, g' \rangle \in map_{\mathcal{F}}$ and input $\boldsymbol{in}$, $g(\boldsymbol{in})$ is called from f and $g'(\boldsymbol{in})$ is not called from f'. This implies $\neg call\text{-}equiv(f, f')$. In f^{UF}, each call of g is replaced with a call of UF_g with the same parameters. If the parameters values $\boldsymbol{in}$ do not depend on the result of any of the recursive functions, then the value of the parameters passed into the corresponding call to UF_g in f^{UF} will not depend on the result of any uninterpreted function and, therefore, will be equal to the value of $\boldsymbol{in}$. Otherwise, namely, $\boldsymbol{in}$ depends on the result of some (or several) recursive function(s), then the non-deterministic values returned by the corresponding uninterpreted function(s) will affect the parameter passed into the corresponding call to UF_g in f^{UF}. But among all the possibilities for those non-deterministic values, there are values that were actually returned by the recursive functions replaced by those uninterpreted functions. Those actual values caused that $\boldsymbol{in}$ was passed into the mentioned call of g. Those actual values (returned by the uninterpreted functions) cause UF_g to be called with $\boldsymbol{in}$ in f^{UF}. Hence, $\boldsymbol{in} \in PV_f[UF_g]$.

If there are no calls of g' in f' at all, then f'^{UF} contains no calls of $UF_{g'}$ either. Hence, $PV_{f'}[UF_{g'}] = \emptyset$, i.e., $PV_f[UF_g] \neq PV_{f'}[UF_{g'}]$. Otherwise (there is a call of g' in f') by our assumption $\boldsymbol{in} \notin PV_{f'}[g']$. Analogously, we can show that in this case when all uninterpreted functions return actual values that their corresponding original functions would, some value different from $\boldsymbol{in}$ is passed into $UF_{g'}$ in f'^{UF}. Hence, there exists a computation with a set of non-deterministic values for which $\boldsymbol{in} \notin PV_{f'}[UF_{g'}]$, i.e., $PV_f[UF_g] \neq PV_{f'}[UF_{g'}]$.

So in both cases we found a computation where $PV_f[UF_g] \neq PV_{f'}[UF_{g'}]$, which implies $\neg call\text{-}equiv(f^{UF}, f'^{UF})$. □

The next lemma proves the validity of the rule (M-TERM) for basic programs consisting of a single function. The function can be either simply recursive or not recursive at all.

Lemma 2. *If all the functions called in $\langle f,\ f'\rangle \in map_{\mathcal{F}}$ are f and f' themselves, respectively, then $call\text{-}equiv(f, f') \rightarrow m\text{-}term(f, f')$.*

Proof. Consider $\langle f,\ f'\rangle \in map_{\mathcal{F}}$ called with the same parameter $\boldsymbol{in}$ such that each one of f and f' can call only itself. Falsely assume $\neg m\text{-}term(f, f')$. W.l.o.g., assume $term(f(\boldsymbol{in}))$ and $\neg term(f'(\boldsymbol{in}))$, where, recall, $term(f(\boldsymbol{in}))$ denotes that $f(\boldsymbol{in})$ terminates.

Consider $\boldsymbol{in_1}$, the parameter passed into the non-returning recursive call to f'. $call\text{-}equiv(f, f')$ implies that f is called with $\boldsymbol{in_1}$ too. $\boldsymbol{in_1} \neq \boldsymbol{in}$ because otherwise we would get $\neg term(f(\boldsymbol{in}))$ according to O1.

Now consider terminating $f(\boldsymbol{in_1})$ and non-terminating $f'(\boldsymbol{in_1})$. The situation is quite similar to $f(\boldsymbol{in})$ and $f'(\boldsymbol{in})$. Thus we conclude that $f(\boldsymbol{in_2})$ must call terminating $f(\boldsymbol{in_2})$, while $f(\boldsymbol{in_1})$ must call non-terminating $f'(\boldsymbol{in_2})$, such that $\boldsymbol{in_2} \notin \{\boldsymbol{in}, \boldsymbol{in_1}\}$. We can go on descending the call stacks discovering a new value $\boldsymbol{in_n} \notin \{\boldsymbol{in}, \boldsymbol{in_1}, ..., \boldsymbol{in_{n-1}}\}$ with which both f and f' are called. But the number of such unique values passed into f is finite according to O2. Hence, we must eventually reach a situation where either:

- $f(\boldsymbol{in_n})$ calls f with some $\boldsymbol{in_i}$ that is already found in the call stack, which contradicts the assumption $term(f(\boldsymbol{in}))$ (by O1);
- $f(\boldsymbol{in_n})$ calls f with some value $\boldsymbol{in_{extra}}$ such that $f'(\boldsymbol{in_{extra}})$ is never called in $f'(\boldsymbol{in_n})$. It contradicts $call\text{-}equiv(f, f')$;
- $f(\boldsymbol{in_n})$ has no more calls to f. However, $f'(\boldsymbol{in_n})$ must call f' because of $\neg term(f'(\boldsymbol{in_n}))$. Again, it contradicts $call\text{-}equiv(f, f')$.

Consequently, the assumption $\neg m\text{-}term(f, f')$ cannot be true. ☐

The next lemma addresses mutual recursion with terminating outer calls.

Lemma 3. *The following rule is sound:*

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

Proof. (Proof sketch) Consider $\langle f,\ f'\rangle \in \langle m,\ m'\rangle$ that are called with the same parameter $\boldsymbol{in}$. Assume by negation $\neg m\text{-}term(f, f')$. W.l.o.g., assume $term(f(\boldsymbol{in}))$ and $\neg term(f'(\boldsymbol{in}))$.

The premise of the lemma implies that all the outer calls (beyond m, m') terminate. Hence, only inner calls inside m, m' could cause $\neg term(f'(\boldsymbol{in}))$.

From now on, the proof is very similar to that of Lemma 2. We start traversing the infinite call stack of $f'(\boldsymbol{in})$. The only difference is that instead of descending with the same function pair in every call stack level (i.e., $\langle f,\ f'\rangle$ in the proof of Lemma 2), we now descend to some pair $\langle h_i,\ h'_i\rangle \in \langle m,\ m'\rangle$ in each level $\#i$. According to O2, the number of unique values for every function from m found in the call stack of $f(\boldsymbol{in})$ is finite because $term(f(\boldsymbol{in}))$ is assumed. On the other side, the number of functions in m' is finite. Hence, the infinite call stack of $f'(\boldsymbol{in})$ must include calls to some function g' s.t. $\langle g,\ g'\rangle \in \langle m,\ m'\rangle$ which repeats

an infinite number of times. The latter will contradict either $term(f(\boldsymbol{in}))$ or $call\text{-}equiv(g, g')$ (similarly to how the infinitely called f' led to the final contradiction in the proof of Lemma 2). $\square$

Lemma 4. *The following rule is sound:*

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

Proof. Consider $\langle f,\ f' \rangle \in map_{\mathcal{F}}$ called with the same parameter $\boldsymbol{in}$. Assume by negation $\neg m\text{-}term(f, f')$. W.l.o.g., assume $term(f(\boldsymbol{in}))$ and $\neg term(f'(\boldsymbol{in}))$.

Consider any function call $g(\boldsymbol{in_1})$ in $f(\boldsymbol{in})$ s.t. $g \notin m \wedge \exists g'.\langle g,\ g' \rangle \in map_{\mathcal{F}}$. $call\text{-}equiv(f, f')$ implies that $f'(\boldsymbol{in_1})$ also calls $g'(\boldsymbol{in_1})$. $\langle g,\ g' \rangle \notin \langle m,\ m' \rangle$ implies $m\text{-}term(g, g')$, which implies that $g(\boldsymbol{in_1})$ and $g'(\boldsymbol{in_1})$ mutually terminate.Hence, $g'(\boldsymbol{in_1})$ must terminate because otherwise this would contradict $term(f(\boldsymbol{in}))$.

Hence, all the outer function calls (referring beyond m, m') must terminate. Thus all the conditions satisfy the premise of Lemma 3, which implies $m\text{-}term(f, f')$. $\square$

Theorem 2. *The inference rule* (M-TERM$^+$) *is sound.*

Proof. According to Lemma 1, $\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ call\text{-}equiv(f^{UF}, {f'}^{UF})$ implies $\forall \langle f, f' \rangle \in map_{\mathcal{F}}(m).\ call\text{-}equiv(f, f')$. Thus, the second line of the premise of the rule of Lemma 4 is satisfied. The upper line of the premise in (M-TERM$^+$) matches the upper line of the premise of the rule of Lemma 4. Having all its premises satisfied, Lemma 4 implies:

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

$\square$