

Heap Assumptions on Demand

Andrey Rybalchenko¹, Andreas Podelski², and Thomas Wies²

¹ Max-Planck-Institute for Software Systems
rybal@mpi-sws.mpg.de

² University of Freiburg
{podelski,wies}@informatik.uni-freiburg.de

Abstract. Termination of a heap-manipulating program generally depends on preconditions that are *heap assumptions* (i.e., assertions describing reachability, aliasing, separation and sharing in the heap). We present an algorithm for the inference of such preconditions. The algorithm exploits a unique interplay between counterexample-producing abstract termination checker and shape analysis. The shape analysis produces heap assumptions on demand to eliminate counterexamples, i.e., non-terminating abstract computations. The experiments with our prototypical implementation indicate its practical potential.

1 Introduction

Heap-manipulating programs are prone to termination errors [2]. Manually inferring preconditions that exclude such errors is both tedious and hard, since the termination reasoning must involve the *shape* of the heap (we use the term shape in the broad sense to describe how heap locations and heap regions are aliased, inter-reachable, separated, and shared). In this paper, we present an algorithm HEAPINFER that automates this inference process. Given a heap-manipulating program, our algorithm computes a set of conditions on the shape of initial states, e.g., at the entry point of a given code fragment, that lead to terminating computations. We identify a class of *regular* programs for which the algorithm HEAPINFER is complete. An evaluation on characteristic examples practically demonstrates that the inferred preconditions are sufficiently weak.

Our algorithm iteratively applies a termination analysis to a ‘shape-free’ abstraction of the program. HEAPINFER avoids invocation of shape analysis until it finds a counterexample in the form of a non-terminating abstract computation, i.e., it applies shape analysis on demand. The shape analysis produces a *heap assumption*, which is an assertion describing the heap shape. This assumption refines either the abstraction or the precondition. As the result, the refinement step eliminates the counterexample. Thus, we obtain an iterative refinement scheme that applies counterexamples to guide the refinement of abstractions and preconditions.

The ‘shape-free’ abstraction and the demand-driven application of shape analysis rely on several specifics of termination proofs. A termination analysis synthesizes termination arguments in the form of ranking functions (whenever

possible). To define a ranking function directly on heaps does not seem appropriate. The notion of a rank is intimately related to numbers. Thus, an intermediate step of our algorithm is to translate the input program over pointer variables, a *heap program* P_H , into a program over integer variables, which we call a *measure program* P_M . This translation step from heap to measure programs represents a low-cost and coarse ‘shape-free’ abstraction.

The algorithm HEAPINFER applies a termination analysis to P_M at the next step. We obtain either a termination proof for P_M and, hence, also for P_H , or a counterexample, i.e., an infinite trace of P_M . In general, the attempt to find a termination proof for P_M fails. This is not surprising as we *expect* that a termination proof must involve some amount of information that only *shape analysis* can compute. Shape analysis is notoriously expensive, however. Hence, our algorithm calls a shape analysis on demand, i.e., for a specific, isolated task: to check the validity of an invariant assertion which is crafted for the counterexample. Recent shape analysis tools can exploit this kind of specificity by adapting the degree of precision, and thus keeping the practical cost of shape analysis at a minimum [3, 23]. Furthermore, these tools can efficiently handle series of analysis requests. They reuse results obtained for previously processed queries when proving a new assertion, and thus avoid re-computation from scratch.

If the shape analysis proves the validity of the invariant assertion by checking a corresponding *assert* statement in P_H , then HEAPINFER inserts a corresponding *assume* statement into the measure program P_M . Thus, it will refine the abstraction represented by P_M . The refined version of P_M still represents a sound abstraction of P_H , but the previously discovered counterexample is no longer feasible in the program P_M . The invariant assertion, which is crafted to exclude the counterexample of P_M , is an expression over integer variables. The expression can be evaluated in P_M as well as in P_H . Thus, it is meaningful in the *assert* statement of the program P_H over pointer variables as well as in the *assume* statement in the program P_M over integer variables.

In summary, the proposed algorithm HEAPINFER exploits a unique interplay between failed abstract termination proofs and shape analysis and applies an interleaving of abstraction and precondition refinement. Thus, we obtain the (to our knowledge first) algorithm for the inference of preconditions on the heap shape that guarantee termination of heap-manipulating programs. The experiments with our prototype implementation indicate its practical potential. We applied our implementation on characteristic fragments of heap manipulating programs including kernel code from an operating system [17]. The inferred preconditions match the intended calling environment, and were confirmed as such by the kernel developers.

Related Work. Our work fills a gap between two recent lines of research: termination proofs under given preconditions (for heap-manipulating programs), and precondition inference for correctness properties other than termination (memory safety of heap-manipulating programs and other safety properties). Our algorithm exploits the recent advances in the respected areas by utilizing the

corresponding analyses as subprocedures: shape analysis for heap-manipulating programs and termination analysis of integer-manipulating programs.

The recent termination analyses for heap manipulating programs, e.g., [2, 5], do not focus on precondition inference, but rather on proving termination under given preconditions. They do not take advantage of lazy reasoning about the heap. Unlike [2], the present version of our algorithm does not account for memory safety. It can be extended to track information related to memory safety by using measures, similarly to [5, 15].

The idea of extracting ranking functions from heap-manipulating programs by translating its statements into updates of integer variables is very natural and is classical by now. The existing transformations of heap-manipulating programs into programs over integer variables in [2, 5] are sophisticated. Each transformation uses a form of shape analysis as a preliminary step, i.e., before translating to a program over integer variables. The shape analysis is used to eagerly infer strongest invariants for the whole program, and is oblivious to the actual proof obligations required for termination reasoning. The cost of the translation and the size of the resulting program over integer variables depend on the number of shapes computed by the shape analysis. In contrast, our work aims at minimizing the cost of the shape analysis by using it only for checking specially crafted assertions. The complexity of the translation step into a measure program does not depend on the number of shapes. It is cubic in the number of pointer variables and linear in the number of statements of the heap program.

The recently proposed algorithm for deriving preconditions for memory safety of list-manipulating programs [8] employs quite different technical concepts. It neither applies shape analysis lazily, nor infers to preconditions for termination.

There is a large amount of related work on shape analysis (the synthesis of invariant assertions about the heap). A partial selection of various approaches contains [4, 6, 12, 13, 21]. Our algorithm uses shape analysis as a black box. While not requiring and being dependent on any particular implementation of shape analysis, HEAPINFER can benefit from shape analyses that are property-directed, e.g. [3, 23].

To the best of our knowledge, our work is the first that applies shape analysis on demand for inferring preconditions. A graph-based heap analysis [21] can be lazily combined with predicate abstraction [14] to improve its precision in proving safety properties [3].

Our algorithm relies on a termination prover for programs over numerical domains. There exist several practical methods and tools for proving termination of such programs, e.g. [7, 9, 10, 11, 16]. All these tools can be employed by our algorithm (after adding an extension to produce counterexamples, if necessary).

2 Preconditions for Kernel Code

A major application area of termination analyses for heap manipulating programs is low-level operating systems code [1, 2]. Often the operating system

kernel contains subroutines whose termination is an inevitable requirement for ensuring that the OS remains responsive.

See Figure 1 for a typical example of such a subroutine. It shows a fragment of the system call handler `process_kill` found in the process scheduler of the operating system VAMOS [17]. The handler kills the process with the given process ID. Among other tasks, the handler needs to ensure consistency of the process scheduler’s data structures, e.g. *ready list*. The ready list contains all processes which are in a state ready for being scheduled. When a process with identifier `process` is killed, the handler needs to ensure that it is removed from the ready list (if it is contained). Furthermore, the maximal priority of the remaining ready processes needs to be recomputed. The outer loop in the handler code traverses the ready list until either `process` is found or NULL is reached. If `process` is found it is removed from the list. Furthermore, if `process` has maximal priority, then the inner loop traverses the ready list once more to compute the new maximal priority of the remaining ready processes.

The execution of the handler `process_kill` may diverge if we call it from an arbitrary program state. The termination property of the code depends on the shape of the ready list. For example, if the ready list is cyclic and does not contain `process` then the outer loop does not terminate.

Our algorithm automatically infers the necessary preconditions for termination: `process_kill` expects an acyclic ready list. First, it automatically introduces integer variables that measure the length of paths along pointer fields in the heap. Their value may be ∞ , which indicates that the corresponding path does not exist in the heap. In our example, there are three measures that track the length of the paths following the `next` link from (1) `ready_list` to NULL, (2) `ready_list_elem` to NULL, and (3) `highest_search` to NULL. We refer to these measures as M_1 , M_2 , and M_3 .

Second, the algorithm translates the heap program into an integer program over measures. For example, the first conjunct in the loop condition of the outer loop is translated to the test ($M_2 \neq 0$), and the outer loop decrements the measure M_2 if M_2 has a value different from ∞ . Next, the precondition inference process iteratively applies a termination analysis to the integer program and a shape analysis to the heap program. The shape analysis is used to derive new facts from the heap program that rule out spurious infinite computations in the integer program. Whenever an infinite computation cannot be ruled out, the precondition is strengthened. Both the precondition and the facts derived from the heap program are assertions over measures.

In our example, the first termination check on the integer program fails. As a counterexample, it produces an infinite computation where measure M_2 is initially ∞ and is never decremented in the outer loop. This is because M_1 (and thus M_2) is initially unconstrained and might have value ∞ . This computation is feasible and corresponds to the infinite traversal of the ready list in case it is cyclic. Consequently, the inference algorithm strengthens the precondition by the assertion ($M_1 < \infty$). This rules out any infinite iteration of the outer loop in the integer program, and, hence, of the heap program.

```

int process_kill(unsigned int pid) {
    proc_id = pid & 127u;
    process = pid2pcb(proc_id); ...
    prev_elem = NULL;
    ready_list_elem = ready_list;
    while ((ready_list_elem != NULL) && (found == false)) {
        proc_id2 = ready_list_elem->pid;
        if (proc_id == proc_id2) {
            if (prev_elem != NULL)
                prev_elem->next = ready_list_elem->next;
            else
                ready_list = ready_list_elem->next;
            ready_list_elem->next = NULL;
            if (process->priority == max_prio) {
                highest_prio = 0u;
                highest_search = ready_list;
                while (highest_search != NULL) {
                    if (highest_search->priority > highest_prio)
                        highest_prio = highest_search->priority;
                    highest_search = highest_search->next;
                }
                max_prio = highest_prio;
            }
            memory_free(proc_id, process->page_table_length);
            process->pid = 0u;
            PT_SET_PREV_ELEM(process->page_table, 0u,
                (PT_PAGE_FIRST_VALUE), 0u, 0u, 1u);
            found = true;
        }
        prev_elem = ready_list_elem;
        ready_list_elem = ready_list_elem->next;
    } ...
}

```

Fig. 1. System call handler from the process scheduler of the VAMOS kernel [17].

Nevertheless, the next application of the termination analysis fails and produces a counterexample that infinitely often iterates through the inner loop with the value of measure M_3 being equal to ∞ . This might come as a surprise, because acyclicity of the ready list, expressed as $(M_1 < \infty)$, is preserved by the heap updates in the body of the outer loop. Thus, the heap program maintains $(M_3 < \infty)$ at entry to the inner loop. However, due to the information loss in the abstraction into measures, this fact cannot be derived for the integer program. Now, the inference algorithm applies the shape analysis to check the validity of the assertion $(m_3 < \infty)$ at the entry to the inner loop. This assertion is expressible in terms of a reachability predicate, which is supported by the shape analysis. The shape analysis verifies that $(M_3 < \infty)$ holds. This fact is propagated to the measure program by assuming $(M_3 < \infty)$ at the inner loop entry. In turn, the subsequent termination check succeeds. Thus, the inference process stops and reports the precondition $(M_1 < \infty)$, which means that `process_kill` expects an acyclic ready list.

In the following, we make the translation to measure programs precise and discuss the precondition inference algorithm in detail.

3 Preliminaries

In this section, we provide necessary definitions for heap manipulating programs, their computations, and properties. To simplify presentation we restrict ourselves to heap programs that manipulate singly-linked lists. In Section 7 we discuss an extension to multi-linked lists.

A heap program $P_H = (V, \mathcal{L}, \ell_0, \ell_E, \mathcal{T})$ consists of:

- V : a finite set of program variables. Each variable $v \in V$ ranges over a set of memory addresses.
- \mathcal{L} : a finite set of control locations of the program. We assume a distinguished program variable pc that ranges over the control locations, and add it to the set of program variables V .
- ℓ_0 : an initial control location.
- ℓ_E : an error control location.
- \mathcal{T} : a finite set of program transitions. Each transition $\tau = (\ell, \text{grd}, \text{op}, \ell')$ consists of an entry and exit locations ℓ and ℓ' , respectively, and a guard grd and operation op . Guards and operations are defined by the following grammar, where $v \in V \setminus \{pc\}$ and n is a data structure link name.

$$\begin{aligned} \text{exp} &::= v \mid \text{exp}.n \\ \text{grd} &::= \text{true} \mid \text{false} \mid \text{exp} = \text{exp} \mid \text{grd} \wedge \text{grd} \mid \neg \text{grd} \\ \text{op} &::= \text{assert}(\text{grd}) \mid v := v \mid v := v.n \mid v.n := v \mid \text{new}(v) \end{aligned}$$

States. A state $s = (\text{stack}, h)$ of a heap program is a valuation of the program variables stack , including pc , together with the heap function h . The heap function h is a *total* function from addresses to addresses. Function h models singly-linked data structures manipulated by the program.

Given a variable $v \in V$, we write $s(v)$ for the valuation of v in the state s . We write $s[v \mapsto e]$ to represent a state s' such that $s'(v) = e$ and for each $u \in V \setminus \{v\}$ we have $s'(u) = s(u)$.

Transition relations. Each transition $\tau = (\ell, \text{grd}, \text{op}, \ell')$ represents a transition relation ρ_τ that contains pairs of states (s, s') such that $s(\text{pc}) = \ell$, $s \models \text{grd}$, and, additionally, s and s' satisfy conditions given below for each operation kind. If op is an operation $\text{assert}(\text{grd})$, we have either $s \models \text{grd}$ and $s' = s[\text{pc} \mapsto \ell']$, or $s \not\models \text{grd}$ and $s' = s[\text{pc} \mapsto \ell_E]$. For dealing with update operations, we define an *evaluation* function eval that computes the value of an expression w.r.t. a given state.

$$\text{eval}(s, \text{exp}) \stackrel{\text{def}}{=} \begin{cases} s(v) & \text{if } \text{exp} = v, \\ h(\text{eval}(s, \text{exp}')) & \text{if } \text{exp} = \text{exp}'.n \end{cases}$$

For an operation that updates a program variable $v := \text{exp}$, we have $s' = s[\text{pc} \mapsto \ell', v \mapsto \text{eval}(s, \text{exp})]$. In case of heap update operation $v.n := \text{exp}$, we have $s' = s[\text{pc} \mapsto \ell']$ and the heap function h is modified at the address $\text{eval}(s, v.n)$ to return the value $\text{eval}(s, \text{exp})$. Finally, if the update operation is an allocation operation $\text{new}(v)$ then $s' = s[\text{pc} \mapsto \ell', v \mapsto a]$ and h is updated to $h \cup \{a \mapsto a'\}$ where $a \notin \text{dom}(h)$ is a fresh address and $a' \in \text{dom}(h) \cup \{a\}$. Thus, we assume a garbage-collected heap where we always allocate a fresh address, but we put no constraint on the value of the heap function for that fresh address.

For a state s and transition τ we denote by $\text{post}(\tau, s)$ the set of all τ -successors of s .

Computations. A program *computation* is a (possibly infinite) sequence $\sigma = s_0, \tau_0, s_1, \tau_1, \dots$ of states and transitions such that $s_0(\text{pc}) = \ell_0$, for each pair of consecutive states s_i and s_{i+1} we have $s_{i+1} \in \text{post}(\tau_i, s_i)$. If σ is finite then for its final state, say s , and for each transitions $\tau \in \mathcal{T}$ we have $\text{post}(\tau, s) = \emptyset$.

Measures. A measure is a term $M(e_1, e_2)$ where e_1 and e_2 are expressions. It denotes the length of the shortest (possibly empty) n -path in the heap from the address denoted by e_1 to the address denoted by e_2 , and ∞ if such a path does not exist.

We extend the evaluation function eval from expressions to measures as follows:

$$\text{eval}(s, M(e_1, e_2)) \stackrel{\text{def}}{=} \begin{cases} \infty & \text{if for all } i \in \mathbb{N} : s \models e_1.n^i \neq e_2 \\ \min\{i \in \mathbb{N} \mid s \models e_1.n^i = e_2\} & \text{otherwise.} \end{cases}$$

Measure Assertions. Measure assertions are defined by the following grammar:

$$\begin{aligned}
rel &::= < \mid > \mid \leq \mid \geq \mid = \\
const &::= 0 \mid 1 \mid 2 \mid \dots \mid \infty \\
mexp &::= const \mid M(exp, exp) \mid mexp + mexp \mid mexp - mexp \\
atom &::= true \mid false \mid mexp \, rel \, mexp \\
assn &::= atom \mid \neg assn \mid assn \wedge assn
\end{aligned}$$

Measure Programs. A measure program $P_M = (\mathcal{M}, \mathcal{L}, \ell_0, \mathcal{T})$ is a program whose program variables \mathcal{M} are the set of all measures. The set of locations \mathcal{L} , and initial location ℓ_0 are as for heap programs. A state of a measure program is a valuation of the pc together with valuations of all measures. Transitions of measure programs are guarded by measure assertions and perform simultaneous updates of all measures. Updates of measures are expressed in terms of measure expressions $mexp$.

Memory safety. The totality of heap function h implies that in a heap program P there exists no computation that can fail because of memory manipulation error, i.e., P is memory safe. This assumption simplifies the presentation of our ‘shape-free’ abstraction of heap programs, but does not impose any practical limitations.

4 Algorithm

This section presents our algorithm `HEAPINFER` for the automatic inference of heap assumptions for termination. Figure 2 shows the algorithm. It takes as input a heap program P_H and a (super)set of measures that need to be tracked in order to prove termination of P_H . The output of the algorithm is a set of preconditions that guarantee termination of the input heap program.

The algorithm consists of two phases: (1) the translation of the heap program into a measure program that simulates the heap program, and (2) a counterexample driven refinement loop. The refinement loop iteratively derives two kinds of new facts. First, it computes invariants of the heap program that eliminate spurious infinite computations in the measure program. Second, it infers preconditions that exclude feasible infinite computations in the heap program. In the following, we describe the two phases of `HEAPINFER` in more details. Section 5 supports the description with illustrative examples.

Translation Figure 3 defines the function `Translate` which is used in line 1 of the algorithm to translate a heap program P_H to a measure program under a given set of tracked measures M . The translation can be seen as a source-to-source transformation of the heap program into a measure program. Each transition of the heap program is translated to a set of transitions in the measure program. An update operation upd in the heap program is translated to a simultaneous

```

input
   $P_H$ : heap program
   $M$ : set of tracked measures
vars
   $P_M$ : measure program
   $st_i$ : measure statement at location  $\ell_i$  and with guard  $guard_i$ 
  PRE: measure assertion
begin
1   $P_M := \text{Translate}(M, P_H)$ 
2  PRE := true
3  repeat
4    if  $P_M$  terminates then
5      return “termination under precondition PRE”
6    else
7       $st_1 \dots st_{m-1}.(st_m \dots st_n)^\omega := \text{choose infinite trace in } P_M$ 
8       $i := \text{choose position in } \{m, \dots, n\}$ 
9      if under precondition PRE,
10      $P_H \cup \ell_i : \text{assert}(\neg guard_i)$  is safe
11     then
12        $P_M := P_M \cup \ell_i : \text{assume}(\neg guard_i)$ 
13     else
14       PRE := PRE  $\wedge \underline{\text{wp}}(P_H, at.\ell_i \rightarrow \neg guard_i)$ 
15     done
end.

```

Fig. 2. Algorithm HEAPINFER for demand-driven inference of heap assumptions. The algorithm uses three oracles: 1) the termination test on a measure program, 2) the safety check on the input heap program strengthened by a measure assertion, and 3) the weakest-precondition operator on measure assertions for the input heap program.

update of all measures in the measure program (tracked or untracked). The difference between tracked and untracked measures is that tracked measures $M(e_1, e_2)$ are updated according to the update function $M_{upd}(e_1, e_2)$ which is defined in Figure 4 while untracked measures are non-deterministically assigned a value from $\mathbb{N} \cup \{\infty\}$.

The rules in Figure 4 that define the update functions should be read top down. The rule that matches first is the one that applies. Most of the rules are straightforward with the exception of the rule for translation of heap updates $x.n := y$. This rule describes the translation of heap updates into updates of measures of the general form $M(z.n^i, w.n^j)$. Since the heap function n occurs in the subexpressions $z.n^i$ and $w.n^j$ of the measure, the translation needs to take into account the effect of the heap update to the denotation of these subexpressions. The first two cases apply the rule recursively until x does neither occur on the path from z to $z.n^i$ nor on the path from w to $w.n^j$. Thus, eventually the third case applies. The third case is again divided into three subcases. The first subcase handles the situation where x does not occur on the path from $z.n^i$ to $w.n^j$, meaning that the measure does not change its value. The second subcase

handles the situation where x is reachable from $z.n^i$ and the update introduces a new path from $z.n^i$ to $w.n^j$ via x and y . Finally, the last subcase handles the situation where the update destroys any existing paths between $z.n^i$ and $w.n^j$.

Each of the update functions $M_{upd}(e_1, e_2)$ defines a set of guarded update expressions of the form $grd \Rightarrow exp$ meaning that if grd is satisfied in the current state of the measure program then the new value of measure $M(e_1, e_2)$ is determined by exp . For example the update function for updates of the form $x := y.n$ and measures of the form $M(x, e)$, where e does not contain x , defines the set of guarded update expressions:

$$\left. \begin{aligned} & \{ M(y, e) = \infty \Rightarrow \infty, \\ & \quad M(y, e) < \infty \wedge M(y, e) > 0 \Rightarrow M(y, e) - 1, \\ & \quad M(y, e) < \infty \wedge M(y, e) = 0 \wedge M(y, y.n) = 1 \Rightarrow M(y.n, e), \\ & \quad M(y, e) < \infty \wedge M(y, e) = 0 \wedge M(y, y.n) = 0 \Rightarrow 0 \end{aligned} \right\}$$

Note that in a given state of the measure program and for any given measure, there is always exactly one guarded update expressions whose guard is satisfied, i.e., updates of tracked measures are deterministic with the exception of updates resulting from the translation of new statements which are itself nondeterministic.

Finally, the function `bifurcate` transforms a single transition with guarded update expressions for each tracked measure into a set of transitions. Each of the resulting transitions corresponds to one possible choice of picking one of the guarded update expressions per tracked measure. The guard of each resulting transition is the translated guard of the original transition in the heap program conjoined with the guards of the chosen guarded update expressions.

The correctness of the translation is discussed in Section 6.

Tracked Measures We determine the set of tracked measures M using a simple heuristic: Initially, we consider measures that are required for precise translation of loop conditions. During the translation of the program additional measures are tracked lazily if they occur in updates of existing tracked measures according to Figure 4. To ensure that the set M remains finite we only track measures of the form $M(x, y)$ where x and y are program variables. Note that the inference algorithm behaves monotonically with respect to M , i.e. adding more measures to the set will result in weaker preconditions.

Refinement loop The core of algorithm `HEAPINFER` is its counterexample driven refinement loop. In each iteration of the algorithm a termination checker is called to check whether the measure program terminates under current precondition `PRE`. If the termination check succeeds then the heap program, too, is guaranteed to terminate under `PRE` and the algorithm stops. Otherwise there exists some infinite computation in the measure program, i.e., a counterexample for termination. The algorithm non-deterministically chooses one of these computations:

$$st_1 \dots st_{m-1} \cdot (st_m \dots st_n)^\omega \ .$$

$$\begin{aligned}
P_{\text{H}} &= (V, \mathcal{L}, \ell_0, \ell_E, \mathcal{T}) \\
\text{Translate}(M, P_{\text{H}}) &= (\mathcal{M}, \mathcal{L}, \ell_0, \ell_E, \bigcup_{\tau \in \mathcal{T}} \text{trIT}(M, \tau)) \\
\text{trIT}(M, (\ell, g, op, \ell')) &= \text{bifurcate}(\ell, \text{trIG}(g), \text{trIO}(M, op), \ell') \\
\text{trIG}(e_1 = e_2) &= \text{M}(e_1, e_2) = 0 \\
\text{trIG}(\text{true}) &= \text{true} \\
\text{trIG}(\text{false}) &= \text{false} \\
\text{trIG}(\neg \text{grd}) &= \neg(\text{trIG}(\text{grd})) \\
\text{trIG}(\text{grd}_1 \wedge \text{grd}_2) &= \text{trIG}(\text{grd}_1) \wedge \text{trIG}(\text{grd}_2) \\
\text{trIO}(M, \text{assert}(\text{grd})) &= \text{assert}(\text{trIG}(\text{grd})) \\
\text{trIO}(M, \text{upd}) &= [\text{ms} := \text{trIU}(M, \text{upd}, \text{ms}) \mid \text{ms} \in \mathcal{M}] \\
\text{trIU}(M, \text{upd}, \text{M}(t_1, t_2)) &= \begin{cases} \text{M}_{\text{upd}}(t_1, t_2) & \text{if } \text{M}(t_1, t_2) \in M \\ * & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 3. Translation of a heap program to a measure program. We use $*$ to denote a non-deterministically chosen element from $\mathbb{N} \cup \{\infty\}$. Here, `bifurcate` creates a set of transitions for each choice of measure updates, `trIT`, `trIG`, `trIO`, and `trIU` translate transitions, guards, operations and updates, respectively.

Now there are two possible cases: either (1) this computation is spurious, *i.e.*, there is no corresponding computation in the heap program, or (2) the computation is feasible in the heap program. To determine whether the counterexample is feasible or not, the algorithm chooses some guard grd_i from the loop segment $(st_m \dots st_n)$. Then, a safety checker is called to verify whether, under the current precondition `PRE`, the negation of grd_i is an invariant of the heap program at location ℓ_i .

If the safety check succeeds then it proves that the counterexample is spurious. In this case we strengthen the guards of all outgoing transitions from ℓ_i in the measure program by the measure assertion $\neg \text{grd}_i$. This ensures that the counterexample is no longer a computation of the strengthened measure program.

If on the other hand the safety check fails, then the counterexample might correspond to a feasible computation in the heap program (or some other choice of grd_i will prove its spuriousness). In this case the algorithm invokes an oracle that computes the weakest precondition of the negated guard grd_i and adds it to the current precondition. If the same counterexample is produced in some later iteration of the refinement loop then the negation of guard grd_i is an invariant of the heap program at location ℓ_i under the new precondition. Thus, the counterexample is eliminated eventually.

If there is a counterexample in the measure program that is spurious but all guards in its loop are reachable by some finite computation in the heap program

If op is $x := y$ then

$$M_{op}(e_1, e_2) \stackrel{\text{def}}{=} M(e_1[y/x], e_2[y/x])$$

If op is $x := y.n$ then

$$M_{op}(x, x) \stackrel{\text{def}}{=} 0$$

$$M_{op}(x.n^i, x.n^j) \stackrel{\text{def}}{=} M_{op}(y.n^{i+1}, y.n^{j+1})$$

$$M_{op}(e, x) \stackrel{\text{def}}{=} M(e, y.n)$$

$$M(e, y) = \infty \Rightarrow M(e, y.n)$$

$$M(e, y) < \infty$$

$$M(y, y.n) = 1$$

$$M(y.n, e) \neq 0 \Rightarrow M(e, y) + 1$$

$$M(y.n, e) = 0 \Rightarrow 0$$

$$M(y, y.n) = 0 \Rightarrow M(e, y)$$

$$M_{op}(x, e) \stackrel{\text{def}}{=} M(y, e)$$

$$M(y, e) = \infty \Rightarrow \infty$$

$$M(y, e) < \infty$$

$$M(y, e) > 0 \Rightarrow M(y, e) - 1$$

$$M(y, e) = 0$$

$$M(y, y.n) = 1 \Rightarrow M(y.n, e)$$

$$M(y, y.n) = 0 \Rightarrow 0$$

$$M_{op}(x.n^i, e) \stackrel{\text{def}}{=} M_{op}(y.n^{i+1}, e)$$

$$M_{op}(e, x.n^i) \stackrel{\text{def}}{=} M_{op}(e, y.n^{i+1})$$

$$M_{op}(e_1, e_2) \stackrel{\text{def}}{=} M(e_1, e_2)$$

If op is $x.n := y$ then

let $e_1 = z.n^i$ and $e_2 = w.n^j$

$$M_{op}(e_1, e_2) \stackrel{\text{def}}{=} M(z, w)$$

$$i > 0 \wedge M(z, x) = k \wedge k < i \Rightarrow$$

$$M_{op}(y.n^{i-k-1}, e_2)$$

$$j > 0 \wedge M(w, x) = k \wedge k < j \Rightarrow$$

$$M_{op}(e_1, y.n^{j-k-1})$$

$$(i > 0 \rightarrow M(z, x) \geq M(z, e_1)) \wedge$$

$$(j > 0 \rightarrow M(w, x) \geq M(w, e_2))$$

$$M(e_1, e_2) \leq M(e_1, x) \Rightarrow M(e_1, e_2)$$

$$M(e_1, e_2) > M(e_1, x)$$

$$M(y, e_2) < \infty \wedge M(y, e_2) \leq M(y, x) \Rightarrow$$

$$M(e_1, x) + 1 + M(y, e_2)$$

$$M(y, e_2) = \infty \vee M(y, e_2) > M(y, x) \Rightarrow \infty$$

If op is $\text{new}(x)$ then

$$M_{op}(x, x) \stackrel{\text{def}}{=} 0$$

$$M_{op}(e, x) \stackrel{\text{def}}{=} \infty$$

$$M_{op}(x, e) \stackrel{\text{def}}{=} k, \quad k \in \mathbb{N}^+ \cup \{\infty\}$$

$$M_{op}(e, x.n^i) \stackrel{\text{def}}{=} *$$

$$M_{op}(x.n^i, e) \stackrel{\text{def}}{=} *$$

$$M_{op}(e_1, e_2) \stackrel{\text{def}}{=} M(e_1, e_2)$$

Fig. 4. Updates of measures for all update operations in heap programs.

then the inference algorithm will produce a precondition which is too strong. In this case the safety check in Line 10 will fail on all of the loop guards and the refinement will rule out the counterexample by strengthening the precondition. This incompleteness is deliberate. In such a case a ranking function based on measures simply does not exist. However, we do not expect to observe this incompleteness for the kind of loops typically found in low-level system code.

Weakest preconditions of measure assertions Algorithm HEAPINFER depends on an oracle `wp` that computes the weakest precondition for a measure assertion and a heap program. We propose a simple solution for implementing this oracle.

Note that measure assertions are closed under weakest preconditions for loop free heap programs. In fact, we can use the update functions from Figure 4 to compute weakest preconditions for finite sequences of transitions. Assume that the current counterexample path π in the refinement loop is of the form

$$st_1 \dots st_{m-1} \cdot (st_m \dots st_n)^\omega .$$

If the algorithm attempts to strengthen the precondition using some guard grd_i from a transition of the loop segment $(st_m \dots st_n)$, then we update precondition PRE as follows:

$$\text{PRE} := \text{widen}(\text{PRE} \wedge \text{wp}(st_1, \dots, st_{i-1}, \neg \text{grd}_i)) .$$

The operator `widen` is a widening operator on measure assertions. `widen(F)` looks for conjuncts of the form $C(x.n^i), C(x.n^{i+1}), \dots$ in F and replaces them by an “infinite” conjunction

$$\forall j \geq 0 : C(x.n^{i+j}) .$$

Note that this is not to be confused with narrowing which is over-approximation of a greatest fixed-point. Here we under-approximate a greatest fixed-point. Section 5 provides an example where widening is needed to ensure termination of algorithm HEAPINFER with the above implementation of oracle `wp`.

If one uses update expressions of measures to compute weakest preconditions then the only nondeterministic updates that affect weakest precondition computation are those that come from the translation of new statements. We use a very simple quantifier elimination procedure to eliminate the resulting universal quantifiers in weakest preconditions.

5 Examples

We will now execute the algorithm HEAPINFER on four instructive examples. These examples are inspired by typical code fragments found in low-level system code, such as the one discussed in Section 2. To clarify the presentation, we represent programs in structured pseudo code and use `assume` statements to represent guards of transitions and preconditions within programs. We further omit the non-deterministic updates of untracked measures in measure programs.

Example 1: TRAVERSE We first execute the algorithm on program TRAVERSE shown on the left hand side of Figure 5. We choose to only track measure $M(p, q)$. Executing line 3 in the algorithm yields the measure program P_M shown on the right hand side of Figure 5. Program P_M does not always terminate. Let us assume that the non-deterministic choice in line 7 of the algorithm HEAPINFER selects the infinite trace $stem.(loop)^\omega$ that repeatedly executes the loop body according to case 1. We use the notation $empty.(\ell[1])^\omega$ to refer to this infinite trace. The stem is empty, and the loop consists of the statement

$$loop \equiv \text{assume}(M(p, q) = \infty); \\ M(p, q) := \infty;$$

There is only one position to choose in line 8 of the algorithm, namely, the one associated with location ℓ and guard $M(p, q) = \infty$. As an assertion on states of program TRAVERSE, this guard means that q is not reachable from p . Obviously, the negated guard $M(p, q) < \infty$ is not an invariant of program TRAVERSE at location ℓ . This means that the condition in line 9/10 does not hold. In this case, the weakest precondition of the stem $\text{wp}_{stem}(M(p, q) < \infty)$ is again the assertion $M(p, q) < \infty$. Thus, line 15 sets PRE to $M(p, q) < \infty$.

One might expect that under the precondition that q is reachable from p the program TRAVERSE terminates. HEAPINFER finds that it is not sufficient. The next iteration of the algorithm produces the counterexample $\ell[2.2.1].(\ell[1])^\omega$. In this counterexample $stem$ consists of the statement:

$$\ell : \text{assume}(M(p, q) < \infty); \\ \text{assume}(M(p, q) = 0); \\ \text{assume}(M(p, p.n) = 1); \\ M(p, q) := M(p.n, q);$$

The sequence $loop$ is the same as before. We again choose guard $M(p, q) = \infty$. Thus, the condition in line 9/10 is again false. The weakest precondition of the negated guard $\text{wp}_{stem}(M(p, q) < \infty)$ is given by

$$\left(\begin{array}{l} M(p, q) < \infty \wedge M(p, q) = 0 \wedge \\ M(p, p.n) = 1 \end{array} \right) \rightarrow M(p.n, q) < \infty$$

which further simplifies to the assertion

$$M(p, q) > 0 \vee M(p, p.n) = 0 \vee M(p.n, q) < \infty .$$

Line 15 updates the precondition PRE to:

$$\text{PRE} \equiv M(p, q) < \infty \wedge \left(\begin{array}{l} M(p, q) > 0 \vee \\ M(p, p.n) = 0 \vee \\ M(p.n, q) < \infty \end{array} \right)$$

The new precondition PRE means that q is reachable from p and either

- p is different from q

<pre> ℓ : do p := p.n; while p ≠ q </pre>	<pre> ℓ : do M(p, q) := 1 M(p, q) = ∞ ⇒ ∞ 2 M(p, q) < ∞ 2.1 M(p, q) > 0 ⇒ M(p, q) - 1 2.2 M(p, q) = 0 2.2.1 M(p, p.n) = 1 ⇒ M(p.n, q) 2.2.2 M(p, p.n) = 0 ⇒ 0; while M(p, q) > 0 </pre>
---	--

Fig. 5. Program TRAVERSE and its associated measure program P_M .

- or they are aliased and either
 - p has a self-loop
 - or p is on a non-trivial cycle

We expect that the program TRAVERSE terminates under the current precondition. Indeed, the termination test of the measure program P_M under the precondition PRE succeeds and the algorithm returns that the program terminates under the precondition PRE.

Example 2: DESTRUCTTRAVERSE Next, we execute the algorithm on program DESTRUCTTRAVERSE shown in Figure 6. Program DESTRUCTTRAVERSE is a modified version of program TRAVERSE that first performs a destructive update $x.n := y$ before traversing the list p . We also inserted a precondition that ensures that p and q are not aliased. We expect that the destructive update affects the termination behavior of the program. Thus the algorithm should derive a precondition that differs from the one that was inferred in the previous example.

Again the only tracked measure in our run of the algorithm is $M(p, q)$. Figure 7 shows the measure program resulting from the translation of program DESTRUCTTRAVERSE. First of all, observe that from the assume statement $\text{assume}(M(p, q) > 0)$ it follows that even after execution of the measure update for $x.n = y$ the assertion $M(p, q) > 0$ always holds. Thus, in the following we can safely ignore all the 2.2 branches in the update of measure $M(p, q)$ in the loop body.

The first iteration of the algorithm produces the following counterexample for termination:

$$(\ell_0, \ell_1[1]).(\ell_2[1])^\omega .$$

This counterexample corresponds to the situation where the destructive update does not affect the path leaving p , but p does not reach q and thus the traversal of p diverges. This is a real counterexample and hence the safety check for the negated guard $M(p, q) < \infty$ of transition $\ell_2[1]$ fails. Therefore, the precondition PRE is set to the weakest precondition of the assertion $M(p, q) < \infty$ for the stem $(\ell_0, \ell_1[1])$:

$$\text{PRE} \equiv M(p, q) \leq M(p, x) \rightarrow M(p, q) < \infty .$$

The next iteration of the algorithm produces the counterexample:

$$(\ell_0, \ell_1[2.2]).(\ell_2[1])^\omega .$$

Here we have the situation that p might reach q before the destructive update, but even if this is the case, x is lying on that path and y (and thus p after the update) does not reach q . Hence traversing p after the update diverges. Again, this is a real counterexample. Computing the weakest precondition of stem $(\ell_0, \ell_1[2.2])$ for the negated guard of the loop transition $\ell_2[1]$ results in the assertion:

$$M(p, q) > M(p, x) \rightarrow M(y, q) < \infty .$$

Thus the precondition PRE is updated to:

$$\begin{aligned} \text{PRE} \equiv & (M(p, q) \leq M(p, x) \rightarrow M(p, q) < \infty) \wedge \\ & (M(p, q) > M(p, x) \rightarrow M(y, q) < \infty) . \end{aligned}$$

The next iteration of the algorithm produces yet another counterexample:

$$(\ell_0, \ell_1[2.3]).(\ell_2[1])^\omega .$$

In this situation p again reaches x before reaching q and, furthermore, y reaches x before reaching q . Thus the destructive update creates a pan-handle list p that does not contain q and once more the subsequent traversal of p diverges. The weakest precondition for the negated loop transition guard $M(p, q) < \infty$ results in the assertion:

$$M(p, q) > M(p, x) \rightarrow M(y, q) \leq M(y, x)$$

and the precondition PRE is set to:

$$\begin{aligned} \text{PRE} \equiv & (M(p, q) \leq M(p, x) \rightarrow M(p, q) < \infty) \wedge \\ & (M(p, q) > M(p, x) \rightarrow M(y, q) < \infty \wedge M(y, q) \leq M(y, x)) . \end{aligned}$$

The meaning of the precondition is:

- either p reaches q without reaching x first
- or p reaches x , but y reaches q before reaching x .

This precondition guarantees that the transition $\ell_2[1]$ is not enabled, which we propagate to the measure program by excluding the corresponding transition. The next iteration of the algorithm proves that the measure program terminates under precondition PRE.

Example 3: SEARCHDESTRUCTTRAVERSE The third example is program SEARCHDESTRUCTTRAVERSE given in Figure 8. It is a modification of program DESTRUCTTRAVERSE that first non-deterministically sets x to some address along the path from p to q before it executes the destructive update $x.n := y$ followed by a traversal of p to q .

The loop that non-deterministically assigns x always terminates, because p reaches q according to the assume statement at location ℓ_0 . However, we need

```

ℓ0 : assume( $p \neq q$ );
ℓ1 :  $x.n := y$ ;
ℓ2 : do
     $p := p.n$ ;
    while  $p \neq q$ 

```

Fig. 6. Program DESTRUCTTRAVERSE.

```

ℓ0 : assume( $M(p, q) > 0$ );
ℓ1 :  $M(p, q) :=$ 
    1  $M(p, q) \leq M(p, x) \Rightarrow M(p, q)$ 
    2  $M(p, q) > M(p, x)$ 
    2.1  $M(y, q) < \infty \wedge$ 
         $M(y, q) \leq M(y, x) \Rightarrow M(p, x) + 1 + M(y, q)$ 
    2.2  $M(y, q) = \infty \Rightarrow \infty$ 
    2.3  $M(y, q) > M(y, x) \Rightarrow \infty$ ;
ℓ2 : do
     $M(p, q) :=$ 
    1  $M(p, q) = \infty \Rightarrow \infty$ 
    2  $M(p, q) < \infty$ 
    2.1  $M(p, q) > 0 \Rightarrow M(p, q) - 1$ 
    2.2  $M(p, q) = 0$ 
    2.2.1  $M(p, p.n) = 1 \Rightarrow M(p.n, q)$ 
    2.2.2  $M(p, p.n) = 0 \Rightarrow 0$ ;
while  $M(p, q) > 0$ 

```

Fig. 7. Measure program P_M for DESTRUCTTRAVERSE.

to track the measure $M(x, q)$ in order to avoid that the algorithm infers false as the only precondition (computing the weakest precondition for the negated loop condition and the stem (ℓ_0, ℓ_1) and conjoining it with the assume statement gives false). For our convenience we will track $M(x, q)$ only until the end of the search loop. The measure program resulting from the translation is shown in Figure 9.

The search loop can be proved terminating immediately. However, the rest of the program behaves essentially in the same way as the previous example. If we only consider counterexamples that exit the search loop immediately using the break statement and then diverge in the traverse loop, then we compute the precondition:

$$(M(p, q) > 0 \rightarrow M(y, q) < \infty \wedge M(y, q) \leq M(y, p)) .$$

If we further consider counterexamples that iterate through the search loop at most twice, then we get the partial precondition:

$$(M(p, q) > M(p, p) \rightarrow M(y, q) < \infty \wedge M(y, q) \leq M(y, p)) \wedge \\ (M(p, q) > M(p, p.n) \rightarrow M(y, q) < \infty \wedge M(y, q) \leq M(y, p.n))$$

and counterexamples with up to three iterations gives us:

$$\begin{aligned} & (M(p, q) > M(p, p) \rightarrow M(y, q) < \infty \wedge M(y, q) \leq M(y, p)) \wedge \\ & (M(p, q) > M(p, p.n) \rightarrow M(y, q) < \infty \wedge M(y, q) \leq M(y, p.n)) \wedge \\ & (M(p, q) > M(p, p.n^2) \rightarrow M(y, q) < \infty \wedge M(y, q) \leq M(y, p.n^2)) . \end{aligned}$$

At this point the widening operator fires and produces the precondition PRE:

$$\forall i : M(p, q) > M(p, p.n^i) \rightarrow M(y, q) < \infty \wedge M(y, q) \leq M(y, p.n^i)$$

This assertion expresses the fact that the list starting from y reaches q , but is disjoint from the list that goes from p to q , *i.e.* the sharing point where the path from p to q and the path from y to q meet is q itself. Under this precondition the negated guard of $\ell_4[1]$ is an invariant. This allows us to prove termination of the measure program.

```

ℓ0 : assume( $p \neq q \wedge q \in p.n^*$ );
ℓ1 :  $x := p$ ;
ℓ2 : while  $x \neq q$  do
  ℓ2,0 : if * then break;
  ℓ2,1 :  $x := x.n$ ;
  od
ℓ3 :  $x.n := y$ ;
ℓ4 : do
   $p := p.n$ ;
  while  $p \neq q$ 

```

Fig. 8. Program SEARCHDESTRUCTTRAVERSE.

Example 4: CREATESEARCHDESTRUCTTRAVERSE Finally, we discuss a program CREATESEARCHDESTRUCTTRAVERSE, which is found in Figure 10. This program is as the previous example with the exception that the required precondition for termination: “the list from p to q is disjoint from the list that goes from y to q ” is ensured by explicitly creating a fresh list from p to q . Thus program CREATESEARCHDESTRUCTTRAVERSE always terminates.

If we execute the algorithm on this example (provided we choose the right measures for the create loop) then it returns precondition **true** after only two iterations. In the first iteration all counterexamples are of the form $stem.(\ell_4[1])^\omega$. The safety check proves that the negated guard of the transition $\ell_4[1]$ is an invariant. Inserting the corresponding assume statement into the measure program excludes all infinite computations. Thus, in the second iteration the algorithm proves termination of the measure program under precondition **true**.

6 Soundness

In the following we prove that our algorithm is sound.

```

ℓ0 : assume(0 < M(p, q) < ∞);
ℓ1 : M(x, q) := M(p, q);
ℓ2 : while M(x, q) > 0 do
  ℓ2,0 : if * then break;
  ℓ2,1 : M(x, q) := M(x, q) - 1;
  od
ℓ3 : M(p, q) :=
  1  M(p, q) ≤ M(p, x) ⇒ M(p, q)
  2  M(p, q) > M(p, x)
  2.1 M(y, q) < ∞ ∧
      M(y, q) ≤ M(y, x) ⇒ M(p, x) + 1 + M(y, q)
  2.2 M(y, q) = ∞ ⇒ ∞
  2.3 M(y, q) > M(y, x) ⇒ ∞;
ℓ4 : do
  M(p, q) :=
  1  M(p, q) = ∞ ⇒ ∞
  2  M(p, q) < ∞
  2.1 M(p, q) > 0 ⇒ M(p, q) - 1
  2.2 M(p, q) = 0
  2.2.1 M(p, p.n) = ⇒ M(p.n, q)
  2.2.2 M(p, p.n) = 0 ⇒ 0;
  while M(p, q) > 0

```

Fig. 9. Measure program P_M for SEARCHDESTRUCTTRAVERSE.

```

ℓ0 : assume(p = q ∧ q ∈ y.n*);
ℓ1 : do
  ℓ1,0 : new(x);
  ℓ1,1 : x.n := p;
  ℓ1,2 : p := x;
  ℓ1,3 : k := k - 1;
  while k > 0
ℓ2 : while x ≠ q do
  ℓ2,0 : if * then break;
  ℓ2,1 : x := x.n;
  od
ℓ3 : x.n := y;
ℓ4 : do
  p := p.n;
  while p ≠ q

```

Fig. 10. Program CREATESEARCHDESTRUCTTRAVERSE.

Proposition 1. *Let F be a measure assertion and let s_h and s_m be states of a heap, respectively a measure program. If s_h and s_m agree on the values of all measures occurring in F then*

$$s_h \models F \iff s_m \models F .$$

Proposition 2. *The translation of guards from heap to measure programs is correct: let F be a guard in a heap program and let s_h be a state of a heap then*

$$s_h \models F \iff s_h \models \text{trIG}(F) .$$

Proof Proof by structural induction on F . If F is of the form $e_1 = e_2$ then we have: $s_h \models e_1 = e_2$ iff the length of the maximal n -path between e_1 and e_2 is 0 iff $s_h \models \text{M}(e_1, e_2) = 0$ iff $s_h \models \text{trIG}(F)$. All other cases are trivial. ■

Proposition 3. *The translation of update operations from heap to measure programs is correct. Let τ be a transition in a heap program for some update operation op and let s, s' be states such that $s' \in \text{post}(\tau, s)$ then we have for all measures $\text{M}(e_1, e_2)$*

$$\text{either } \text{M}_{op}(e_1, e_2) = * \text{ or } \text{eval}(\text{M}(e_1, e_2), s') = \text{eval}(\text{M}_{op}(e_1, e_2), s) .$$

Proof Sketch We need to prove that for all update operations op and measures $\text{M}(z.n^i, w.n^j)$ all the guarded update expressions defined by $\text{M}_{op}(z.n^i, w.n^j)$ are correct. The proof is quite long. We restrict ourselves to the most interesting case, namely where op is a destructive update of the form $x.n := y$.

We prove by induction on $l \stackrel{\text{def}}{=} i + j$ that for all $i, j \in \mathbb{N}$

$$\text{eval}(\text{M}(z.n^i, w.n^j), s') = \text{eval}(\text{M}_{op}(z.n^i, w.n^j), s) .$$

Let $l = 0$ then the third case in the definition of $\text{M}_{op}(z.n^i, w.n^j)$ applies. Assume that s satisfies $\text{M}(z, w) \leq \text{M}(z, x)$. Then either (a) $\text{M}(z, w) = \text{M}(z, x) = \infty$ or (b) $\text{M}(z, w) < \infty \wedge \text{M}(z, w) \leq \text{M}(z, x)$. In case (a) z has no outgoing path to w and no outgoing path to x , thus no new path from z to w is created in s' and therefore

$$\text{eval}(\text{M}(z, w), s') = \text{eval}(\text{M}_{op}(z, w), s') = \infty .$$

In case (b) z has a path to w , but x is not lying on that path (though x and w might be aliased). Thus, the destructive update of $x.n$ does not affect the path from z to w and hence

$$\text{eval}(\text{M}(z, w), s') = \text{eval}(\text{M}_{op}(z, w), s) = \text{eval}(\text{M}(z, w), s) .$$

Now assume that s satisfies $\text{M}(z, w) > \text{M}(z, x)$ and furthermore

$$\text{M}(y, w) < \infty \wedge \text{M}(y, w) \leq \text{M}(y, x) .$$

In this case there is a path from z to w in s , but x lies on that path and x has no alias with w . Furthermore, there is a path from y to w and x is not lying on

that path. Thus in s' there is a new path created from z over x and y to w and the minimal length of that path is $M(z, x) + 1 + M(y, w)$ (the 1 comes from the fact that $x \neq w$), *i.e.*

$$\begin{aligned} eval(M(z, w), s') &= eval(M_{op}(z, w), s) \\ &= eval(M(z, x) + 1 + M(y, w), s) . \end{aligned}$$

Finally assume that s satisfies $M(z, w) > M(z, x)$ and

$$M(y, w) = \infty \vee M(y, w) > M(y, x) .$$

In this case again there is a path from z to w in s , x lies on that path, and x has no alias with w . Furthermore either there is no path from y to w in s or, if it exists, x is again lying on that path. Thus the path from z to w is destroyed in s' and therefore:

$$eval(M(z, w), s') = eval(M_{op}(z, w), s) = \infty .$$

Now let $l > 0$. First lets assume that $i > 0$ and s satisfies $M(z, x) = k$ for some $k < i$. Then $x = z.n^k$ and $x \neq z.n^l$ for all $l < k$. Then we know that state s' satisfies $z.n^{k+1} = y$ and thus we have by induction hypothesis

$$eval(M(z.n^i, w.n^j), s') = eval(M_{op}(y.n^{i-k-1}, w.n^j), s) .$$

The case for $j > 0$ where s satisfies $M(w, x) = k$ for some $k < j$ is analogous to the previous case and the third case is again analogous to the base case of the induction. ■

Proposition 4. *The measure program $P_M = \text{Translate}(P_H)$ simulates the heap program P_H , formally: let $\sigma_H = s_0, \tau_0, s_1, \tau_1, \dots$ be a computation of program P_H then there exists a computation $\sigma_M = s'_0, \tau'_0, s'_1, \tau'_1, \dots$ of program P_M such that for all i states s_i and s'_i agree on the values of all measures.*

Proof For all i define s'_i as the state of the measure program that agrees with s_i on the values of all measures. Now choose any two consecutive states s_i and s_{i+1} in σ_H . Let $\tau_i = (\ell, \text{grd}, \text{op}, \ell')$. From Prop. 2 and $s_i \models \text{grd}$ it follows that $s'_i \models \text{trIG}(\text{grd})$. If operation op is an assert statement $\text{assert}(g)$ then $s_i = s_{i+1}$ and hence $s'_i = s'_{i+1}$. Furthermore, by Prop. 2, $s_i \models g$ implies $s'_i \models \text{trIG}(g)$. Thus define $\tau'_i = (\ell, \text{trIG}(\text{grd}), \text{assert}(\text{trIG}(g)), \ell')$ and we have $s'_{i+1} \in \text{post}(\tau_i, s'_i)$.

If on the other hand op is an update operation then by Prop. 3 we know that for all measures $M(e_1, e_2)$ we have

$$\begin{aligned} \text{either } M_{op}(e_1, e_2) &= * \\ \text{or } eval(M(e_1, e_2), s_{i+1}) &= eval(M_{op}(e_1, e_2), s_i) . \end{aligned}$$

Thus for all tracked measures $M(e_1, e_2)$ with $M_{op}(e_1, e_2) \neq *$ we have

$$eval(M(e_1, e_2), s'_{i+1}) = eval(M_{op}(e_1, e_2), s'_i) .$$

Consequently there is some transition τ' in the bifurcation of $(\ell, \text{trIG}(\text{grad}), \text{trIO}(\text{op}), \ell')$ such that $s'_{i+1} \in \text{post}(\tau', s'_i)$. Choose such τ' and define $\tau'_i = \tau'$.

We can conclude that for all $i \in \mathbb{N}$ we have $s'_{i+1} \in \text{post}(\tau_i, s'_i)$, i.e. the sequence

$$\sigma_M \stackrel{\text{def}}{=} s'_0, \tau'_0, s'_1, \tau'_1, \dots$$

is a computation of the measure program. ■

Lemma 1. *At the beginning of every iteration of the algorithm the measure program P_M simulates the program P_H restricted to traces that satisfy the current precondition PRE.*

Proof Proof by induction on the number of iterations. In the first iteration the claim follows from Prop. 4. In any later iteration let P_M^0 and PRE^0 be the measure program and precondition at the beginning of the previous iteration of the loop. Now let $\sigma_H = s_0, \tau_0, s_1, \tau_1, \dots$ be a computation of P_H such that s_0 satisfies PRE. In the previous iteration either the precondition PRE_0 was strengthened or an assume statement was inserted into program P_M^0 . In any case s_0 satisfies PRE^0 and thus by induction hypothesis there exists a computation $\sigma_M = s'_0, \tau_0, s'_1, \tau_1, \dots$ of program P_M^0 such that s'_0 satisfies PRE^0 and for all i states s_i and s'_i agree on the value of all measures. If the precondition was strengthened, then $\text{PRE} = \text{PRE}^0 \wedge F$ for some measure assertion F and $P_H = P_H^0$. In this case σ_M is a computation of P_H and by Prop. 1 $s_0 \models F$ implies $s'_0 \models F$ and thus s'_0 satisfies the new precondition PRE.

If on the other hand program P_M^0 was modified then $\text{PRE} = \text{PRE}^0$. Let ℓ be the program location with the modification and let $\text{assume}(F)$ be the inserted assume statement. Now define a new sequence σ'_M by replacing all transitions $\tau'_i = (\ell, \text{grad}, \text{op}, \ell')$ in σ_M by transitions $(\ell, \text{grad} \wedge F, \text{op}, \ell')$. Since F is an invariant of P_H at location ℓ all states s_i with $s_i(\text{pc}) = \ell$ satisfy F . Thus by proposition Prop. 1 all states s'_i at location ℓ satisfy F and therefore σ'_M is a computation of program P_M . ■

Theorem 1 (Soundness). *Given a heap program P_H , if the algorithm returns a precondition PRE then program P_H terminates under precondition PRE.*

Proof From Lemma 1 it follows that in the final iteration of the algorithm every computation of program P_H that satisfies precondition PRE has a counterpart in the measure program P_M . Thus termination of P_M under precondition PRE implies termination of P_H under precondition PRE. ■

Completeness We will now give a characterization of a subclass of heap programs for which the algorithm is not only sound, but also complete. Intuitively, in the subclass of *regular* heap programs the termination of a computation depends solely on the *shape* of its states, i.e. on the truth value of measure assertions (“shape-compatible computations are termination-equivalent”).

Definition 1. A heap program P_H is regular if every computation σ of P_H that is shape-compatible with a non-terminating computation $(st_m, \dots, st_n)^\omega$ of the measure program P_M is non-terminating itself. Here shape-compatible formally means that for every statement st_i (with location ℓ_i and guard grd_i) on the loop (i.e., for $i = m, \dots, n$), the computation σ reaches a state at the location ℓ_i satisfying the measure assertion grd_i .

The next statement assumes perfect oracles for “terminates”, “is safe” and the weakest precondition operator wp .

Proposition 5 (Completeness). Applied to a regular program, the algorithm excludes no terminating computation, i.e., for every terminating computation σ there is one of the preconditions PRE produced by the algorithm such that σ satisfies PRE .

Proof Given a finite computation σ of the heap program P_H , we will show that in each iteration of the algorithm, when the algorithm has chosen an infinite computation of the measure program, say $st_1 \dots st_{m-1} \cdot (st_m \dots st_n)^\omega$, there exists a choice of a position i on the loop (i.e., between m and n), such that σ satisfies the conjunct that is added to the precondition PRE in that iteration. By the assumption that P_H is regular, there exists a location ℓ_i (for some i between m and n) such that σ never reaches ℓ_i in a state that satisfies grd_i . Let the algorithm choose the corresponding position i on the loop. Thus the conjunct added to PRE is of the form $\text{wp}(at_l_i \rightarrow \neg grd_i)$. The computation σ satisfies this conjunct since $(at_l_i \rightarrow \neg grd_i)$ is an invariant of σ . ■

Progress and Termination In Section 4 we gave a very abstract version of the algorithm that does not put any restriction on how non-deterministic choices of counterexamples and guards are made and how backtracking is done, once a valid precondition has been derived. As a consequence, algorithm HEAPINFER is not guaranteed to terminate. In the following we give sufficient restrictions on the possibilities for backtracking such that the algorithm always terminates (again under the assumption of perfect oracles for checking termination, safety, and computing wp).

Proposition 6 (Progress). If an infinite trace of the measure program is chosen as a counterexample to termination in an iteration of the refinement loop, then the counterexample is eliminated eventually (within a finite number of iterations of the repeat loop of the algorithm).

Proof The proof closely follows the discussion in Section 4. ■

In fact we know more. Namely: if a guard grd at some program location ℓ is chosen in an iteration of the refinement loop, then all infinite computations in the measure program that have a state satisfying grd at location ℓ are eventually eliminated.

Proposition 7. The iterative refinement of the precondition PRE terminates for each sequence of choices for the guards grd made in the refinement loop.

Proof In each iteration of the refinement loop one guard g_{rd} is chosen. The progress property guarantees that all infinite computations that have states satisfying g_{rd} are eventually eliminated in some later iteration of the refinement loop. If all guards are disabled then there are no infinite computations. Since the number of guards g_{rd} is finite, eventually all infinite computations are eliminated. Thus the iterative refinement of precondition PRE terminates for each sequence of choices for the guards. ■

Proposition 8 (Termination). *The algorithm HEAPINFER terminates producing a set of preconditions that ensure termination of the input program if the backtracking points for non-deterministic choices in the refinement loop are restricted in the following way: each time a valid precondition has been derived, the algorithm backtracks to the earliest iteration of the algorithm where a different guard can be chosen. If no such backtracking point exists then the algorithm stops.*

Proof In each iteration of the algorithm where all choices of guards in iterations of smaller depth have been made, there exists only a finite number of sequences of possible choices for the remaining guards. By Proposition 7 the iterative refinement of precondition PRE with any such sequence of choices for the guards terminates. Thus algorithm HEAPINFER terminates with this backtracking strategy producing a finite set of valid preconditions that ensure termination of the input heap program. ■

Note that the restriction on the backtracking points made in Proposition 8 has no effect on the completeness of the algorithm for regular programs that we obtained above.

7 Multi-linked data structures

We show how our algorithm presented in Section 4 can infer preconditions for termination of programs that manipulate multi-linked heaps. We use an illustrative example, and highlight the issues that arise from the multi-linked setup.

We consider the program BOTHENDS shown in Figure 11. We refine our notion of measure to account for different links. We extend measures to triples whose first component is the name of a linking field. Such extended measure, say $M(l, e_1, e_2)$, denotes the length of the l -path similarly to the definition of measure in Section 3. In our example, we consider measures $M(n, x, y)$ and $M(p, y, x)$ that yield the measure program shown on the right hand side of Figure 11. The translation into measure program is a straightforward extension of the singly-linked case. We make the link name explicit, and treat different links in isolation. That is, update expressions for the link n are oblivious to the existence of the link p , and vice versa.

<pre> ℓ₀ : while $x \neq y$ do if * then $x := x.n$; else $y := y.p$; od </pre>	<pre> ℓ₀ : while $M(n, x, y) \neq 0$ do if * then ℓ₁ : $M(n, x, y) :=$ 1 $M(n, x, y) = \infty \Rightarrow \infty$ 2 $M(n, x, y) < \infty$ 2.1 $M(n, x, y) > 0 \Rightarrow M(n, x, y) - 1$ 2.2 $M(n, x, y) = 0$ 2.2.1 $M(n, x, x.n) = 1 \Rightarrow M(n, x.n, y)$ 2.2.2 $M(n, x, x.n) = 0 \ 0$ ℓ₂ : $M(p, y, x) := M(p, y, x.n)$ else ℓ₃ : $M(n, x, y) := M(p, x, y.p)$ ℓ₄ : $M(p, y, x) :=$ 1 $M(p, y, x) = \infty \Rightarrow \infty$ 2 $M(p, y, x) < \infty$ 2.1 $M(p, y, x) > 0 \Rightarrow M(p, y, x) - 1$ 2.2 $M(p, y, x) = 0$ 2.2.1 $M(p, y, y.p) = 1 \Rightarrow M(p, y.p, y)$ 2.2.2 $M(p, y, y.p) = 0 \Rightarrow 0$ od </pre>
---	--

Fig. 11. Program BOTHENDS and the associated measure program.

Our algorithm proceeds as outlined in Section 4. After five iterations, it infers the precondition

$$\begin{aligned}
&M(n, x, y) > 0 \rightarrow M(n, x, y) < \infty \wedge \\
&M(n, x, y) > 1 \rightarrow M(n, x, y.p) < \infty \wedge \\
&\forall i : M(n, x, y) > i \rightarrow M(p, y, x.n^i) < \infty \wedge \\
&M(n, x, y) > 2 \rightarrow M(n, x, y.p.p) < \infty .
\end{aligned}$$

After applying widening, which is triggered by the chains 0, 1, 2 and $y, y.p, y.p.p$, we report

$$\begin{aligned}
\text{PRE} \equiv &\forall i : M(n, x, y) > i \rightarrow M(p, y, x.n^i) < \infty \wedge \\
&\forall i : M(n, x, y) > i \rightarrow m(n, x, y.p^i) < \infty .
\end{aligned}$$

This precondition requires that y is reachable from x following the link n , and all nodes on the n -path between x and y are reachable from y by traversing the link p . That is, we have a doubly-linked list between x and y .

8 Implementation and experiments

We developed a prototype implementation, called BOUNCER, of our algorithm for the demand-driven inference of heap assumptions. We applied BOUNCER to the example programs in Section 5 and a scheduling routine from the VAMOS kernel [17]. In the following, we present a brief overview of BOUNCER.

BOUNCER applies the BOHNE tool for symbolic shape analysis to implement the oracle that checks assertion validity of heap programs [19, 22]. For proving termination of measure programs, BOUNCER applies the ARMC tool for proving termination of transition relations in linear arithmetic [18, 20]. The oracle for wp uses widening, as described in Section 4.

We model the value ∞ in our translation to a measure program by a negative integer constant, say c . Our translation rewrites each measure expression according to the following rules:

$$\begin{aligned} mexp = \infty &\longrightarrow mexp = c, \\ mexp \leq \infty &\longrightarrow mexp = c \vee mexp \geq 0, \\ mexp < \infty &\longrightarrow mexp \geq 0. \end{aligned}$$

Thus, we can apply a termination checker for programs over numerical domains as black-box.

While our implementation is preliminary, we observe that the behavior of the algorithm with respect to the number of applied measures is similar to the behavior of algorithms for predicate abstraction with respect to the number of predicates. We believe that using measures locally, in a way similar to localized abstraction [14], can make our tool scale to larger programs.

Our experiments with process scheduling functions from the VAMOS kernel show that BOUNCER can successfully infer preconditions for termination for interesting practical programs. In the current implementation, we had to manually abstract all non-heap operations by non-deterministic choice. The inferred preconditions are in agreement with the preconditions provided manually by the VAMOS developers.

9 Conclusion

The contribution of this paper is to present the (to our knowledge first) algorithm to infer heap assumptions, i.e., preconditions on the shape of states for which the computation of the given heap-manipulating program terminates, and to investigate the theoretical basis of the algorithm and its practical potential. A number of ideas and concepts have flown into the algorithm; their fruitful combination may be of independent interest:

- Reasoning over a program with pointer variables can be delegated to reasoning over a ‘shape-free’ program whose first version is obtained by a syntax translation (i.e., without shape analysis or any other reasoning procedure over pointers) and whose subsequent versions are obtained by applying shape analysis selectively (i.e., in order to recover the originally incurred loss of precision where (and when) needed).
- The ‘computation’ of the new abstraction after each refinement step can be implemented by a source-to-source transformation, more specifically by inserting an *assume* statement (in contrast with the calls to a theorem prover for the re-computation of the abstract transformer, in the setting of predicate abstraction)

- Abstraction refinement can take place by plainly making explicit in the abstract program (by inserting an *assume* statement) what has been found to be implicit in the concrete program (by proving an invariant).
- By choosing the appropriate notion of ‘shape-free’ programs for the abstraction of program over pointer variables, one can apply the syntactically same assertion to states of either kind of programs and use it to formulate an invariant and an *assume* statement in the one and the other.
- A counterexample of the abstract program can be eliminated in the most direct way by the insertion of an *assume* statement blocking the one branch (in a non-deterministic choice) that was taken by the counterexample.
- The basic principle of eliminating counterexamples in counterexample-guided abstraction refinement can be extended from spurious counterexample to general counterexamples (spurious or not) in order to obtain an analogous notion of *counterexample-guided precondition refinement*.

Future work may be to infer combined, heap-*and*-integer assumptions (such as: “the input of a procedure must be a cyclic list of length strictly greater than parameter *sz*”), and to investigate the computation of alias assumptions from measure programs (based on the fact that aliasing is a special case of a measure assertion).

Acknowledgments. Andrey Rybalchenko is supported in part by Microsoft Research through the European Fellowship Programme. Thomas Wies is supported by a Microsoft Research European PhD Scholarship.

References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O’Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.
2. J. Berdine, B. Cook, D. Distefano, and P. O’Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, 2006.
3. D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *CAV*, 2006.
4. I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In *CAV*, 2007.
5. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, 2006.
6. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *SAS*, 2006.
7. A. Bradley, Z. Manna, and H. Sipma. The polyranking principle. In *ICALP*, 2005.
8. C. Calcagno, D. Distefano, P. O’Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, 2007.
9. M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV*, 2002.
10. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
11. P. Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In *VMCAI*, 2005.

12. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
13. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.
15. S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, 2007.
16. P. Manolios and D. Vroon. Termination analysis with calling context graphs. In *CAV*, 2006.
17. S. Maus. Developing an Operating System Kernel for the VAMP Processor. Diploma thesis, Universität des Saarlandes, 2005.
18. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, 2007.
19. A. Podelski and T. Wies. Boolean heaps. In *SAS*, 2005.
20. A. Rybalchenko. ARMC. <http://www.mpi-sws.org/~rybal/armc/>, 2008.
21. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 2002.
22. T. Wies. Symbolic Shape Analysis. Diploma thesis, Universität des Saarlandes, Germany, 2004.
23. T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. Verifying complex properties using symbolic shape analysis. In *HAV Workshop*, 2007.