Heap Assumptions on Demand

Andreas Podelski[1], Andrey Rybalchenko[2], and Thomas Wies[1]

[1] University of Freiburg
[2] MPI-SWS

Abstract. Termination of a heap-manipulating program generally depends on preconditions that express *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 prototype 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, 25]. 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, see [19], 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, 13,22]. 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, 25].

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 [22] 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 operating system remains responsive.

Figure 1 presents an 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. The handler needs to ensure consistency of the process scheduler's data structures, e.g. *ready list*. The ready list keeps track of all processes that are ready for being scheduled. When a process with identifier `process` is killed, the handler ensures that the process is removed from the ready list (if it is contained). Furthermore, the maximal priority of the remaining ready processes is 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.

```
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;
      } ...
      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].

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 HEAPINFER automatically infers the necessary preconditions for termination: `process_kill` expects the ready list to be acyclic. At the first inference step, the algorithm automatically introduces integer variables that measure the length of paths along pointer fields in the heap. Their value may be infinity, represented by ∞, 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 M_1, M_2, and M_3 respectively.

Then, HEAPINFER translates the heap program into a measure program over integers. For example, the first conjunct in the loop condition of the outer loop is translated to the disequality test $\mathrm{M}_2 \neq 0$, and the outer loop decrements the measure M_2 if its value is different from ∞. Next, the precondition inference process iteratively applies a termination analysis to the measure 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 non-terminating computations in the measure program. Whenever such a computation cannot be ruled out, the precondition is strengthened. Both the precondition and the fact derived from the heap program are assertions over measures.

In our example, the first termination check on the measure program fails. As a counterexample, it reports an infinite computation in which the 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 $\mathrm{M}_1 < \infty$. This rules out any infinite iteration of the outer loop in the measure program, and, hence, of the heap program.

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 $\mathrm{M}_1 < \infty$, is preserved by the heap updates in the body of the outer loop. Thus, the heap program maintains $\mathrm{M}_3 < \infty$ at entry to the inner loop. However, due to the loss of precision by the measure abstraction, this fact cannot be derived for the measure program. Now, the inference algorithm applies the shape analysis to check the validity of the assertion $\mathrm{M}_3 < \infty$ at the entry to the inner loop. This assertion is expressible using a reachability predicate supported by the shape analysis. The shape analysis verifies that $\mathrm{M}_3 < \infty$ holds. This fact is propagated to the measure program by assuming $\mathrm{M}_3 < \infty$ at the inner loop entry that, in turn, makes the subsequent termination check succeed. The inference process stops and reports the precondition $\mathrm{M}_1 < \infty$. It states that `process_kill` expects an acyclic ready list.

3 Preliminaries

We now provide necessary definitions of heap manipulating programs, their computations, and properties. To simplify presentation, we restrict ourselves to heap programs that manipulate singly-linked lists. An extension to multi-linked lists is discussed in the technical report [19].

Heap programs. We represent a *heap program* P_{H} by a tuple $(V, \mathcal{L}, \ell_0, \mathcal{T})$. Here, V is a finite set of program variables. Each variable $v \in V$ ranges over a set of memory addresses. $\mathcal{L}$ is a finite set of control locations of the program that includes the initial location ℓ_0. We assume a distinguished program variable pc that ranges over the control locations $\mathcal{L}$, and is includes in V. $\mathcal{T}$ is a finite set of program transitions. Each transition $\tau = (\ell, grd, op, \ell')$ consists of an entry and exit locations ℓ and ℓ', respectively, 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.

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

A *state* $s = (stack, h)$ of a heap program is a valuation of the program variables $stack$ 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 $s'(u) = s(u)$.

Each transition $\tau = (\ell, grd, op, \ell')$ represents a transition relation ρ_τ that contains pairs of states (s, s') such that $s(pc) = \ell$, $s \models grd$ and the following conditions apply to

s and s'. If op is an operation $\mathsf{assert}(grd)$, we have either $s \models grd$ and $s' = s[pc \mapsto \ell']$, or $s \not\models grd$ and $s' = s[pc \mapsto \ell_E]$. For dealing with update operations, we define an *evaluation* function $eval$ that computes the value of an expression in a given state.

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

For an operation that updates a program variable $v := exp$, we have $s' = s[pc \mapsto \ell', v \mapsto eval(s, exp)]$. In case of heap update operation $v.n := exp$, we have $s' = s[pc \mapsto \ell']$ and the heap function h is modified at the address $eval(s, v.n)$ to map to the value $eval(s, exp)$. Finally, if the update operation is an allocation operation $\mathsf{new}(v)$ then $s' = s[pc \mapsto \ell', v \mapsto a]$ and h is updated to $h \cup \{a \mapsto a'\}$ where $a \notin \mathsf{dom}(h)$ is a fresh address and $a' \in \mathsf{dom}(h) \cup \{a\}$. 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 $\mathsf{post}(\tau, s)$ the set of all τ-successors of s.

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

Measure Programs. A measure is a term $\mathrm{M}(e_1, e_2)$ 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:

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

Measure assertions are defined by the following grammar:

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

A measure program $P_{\mathrm{M}} = (\mathcal{M}, \mathcal{L}, \ell_0, \mathcal{T})$ 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, and can be easily avoided in practice by using measures for proving memory safety.

```
     input
       P_H: heap program
       M: set of tracked measures
     vars
       P_M : measure program
       st_i : measure statement at location ℓ_i and with guard guard_i
       PRE: measure assertion
     begin
1      P_M := 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 ... st_{m-1}.(st_m ... st_n)^ω := choose infinite trace in P_M
8          i := choose position in {m, ..., n}
9          if under precondition PRE, P_H ∪ ℓ_i : assert(¬guard_i) is safe then
10           P_M := P_M ∪ ℓ_i : assume(¬guard_i)
11         else
12           PRE:= PRE ∧ wlp(P_H, at_ℓ_i → ¬guard_i)
       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.

4 Algorithm

We present our algorithm HEAPINFER for the automatic inference of heap assumptions for termination in Figure 2. It takes as input a heap program P_{H} and a set of measures to be tracked for proving termination of P_{H}. The output of the algorithm is a set of preconditions that guarantee termination of the input heap program.

HEAPINFER executes in two phases: the translation of the heap program into a measure program that simulates the heap program, and a counterexample-guided refinement. The refinement phase iteratively derives two kinds of new facts. First, it computes invariants of the heap program that eliminate spurious non-terminating 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 this description with illustrative examples.

Translation. Figure 3 presents the function Translate that is used in line 1 of HEAPINFER to translate a heap program P_{H} into a measure program under a given set of tracked measures M. The translation can be seen as a source-to-source transformation. 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 update of all measures in the measure program (tracked or untracked). Tracked measures $\text{M}(e_1, e_2)$ are updated according to the update function $\text{M}_{upd}(e_1, e_2)$, as de-

$$
\begin{aligned}
P_{\text{H}} &= (V, \mathcal{L}, \ell_0, \ell_E, \mathcal{T}) \\
\mathsf{Translate}(M, P_{\text{H}}) &= (\mathcal{M}, \mathcal{L}, \ell_0, \ell_E, \bigcup_{\tau \in \mathcal{T}} \mathsf{trlT}(M, \tau)) \\
\mathsf{trlT}(M, (\ell, g, op, \ell')) &= \mathsf{bifurcate}(\ell, \mathsf{trlG}(g), \mathsf{trlO}(M, op), \ell') \\
\mathsf{trlG}(e_1 = e_2) &= \text{M}(e_1, e_2) = 0 \\
\mathsf{trlG}(\mathsf{true}) &= \mathsf{true} \\
\mathsf{trlG}(\mathsf{false}) &= \mathsf{false} \\
\mathsf{trlG}(\neg grd) &= \neg(\mathsf{trlG}(grd)) \\
\mathsf{trlG}(grd_1 \wedge grd_2) &= \mathsf{trlG}(grd_1) \wedge \mathsf{trlG}(grd_2) \\
\mathsf{trlO}(M, \mathsf{assert}(grd)) &= \mathsf{assert}(\mathsf{trlG}(grd)) \\
\mathsf{trlO}(M, upd) &= [ms := \mathsf{trlU}(M, upd, ms) \mid ms \in \mathcal{M}] \\
\mathsf{trlU}(M, upd, \text{M}(t_1, t_2)) &= \begin{cases} \text{M}_{upd}(t_1, t_2) & \text{if } \text{M}(t_1, t_2) \in M \\ * & \text{otherwise} \end{cases}
\end{aligned}
$$

If op is $x := y$ then
$\text{M}_{op}(e_1, e_2) \stackrel{\text{def}}{=} \text{M}(e_1[y/x], e_2[y/x])$

If op is $x := y.n$ then
$\text{M}_{op}(x, x) \stackrel{\text{def}}{=} 0$
$\text{M}_{op}(x.n^i, x.n^j) \stackrel{\text{def}}{=} \text{M}_{op}(y.n^{i+1}, y.n^{j+1})$
$\text{M}_{op}(e, x) \stackrel{\text{def}}{=}$
$\quad \text{M}(e, y) = \infty \Rightarrow \text{M}(e, y.n)$
$\quad \text{M}(e, y) < \infty$
$\quad\quad \text{M}(y, y.n) = 1$
$\quad\quad\quad \text{M}(y.n, e) \neq 0 \Rightarrow \text{M}(e, y) + 1$
$\quad\quad\quad \text{M}(y.n, e) = 0 \Rightarrow 0$
$\quad\quad \text{M}(y, y.n) = 0 \Rightarrow \text{M}(e, y)$
$\text{M}_{op}(x, e) \stackrel{\text{def}}{=}$
$\quad \text{M}(y, e) = \infty \Rightarrow \infty$
$\quad \text{M}(y, e) < \infty$
$\quad\quad \text{M}(y, e) > 0 \Rightarrow \text{M}(y, e) - 1$
$\quad\quad \text{M}(y, e) = 0$
$\quad\quad\quad \text{M}(y, y.n) = 1 \Rightarrow \text{M}(y.n, e)$
$\quad\quad\quad \text{M}(y, y.n) = 0 \Rightarrow 0$
$\text{M}_{op}(x.n^i, e) \stackrel{\text{def}}{=} \text{M}_{op}(y.n^{i+1}, e)$
$\text{M}_{op}(e, x.n^i) \stackrel{\text{def}}{=} \text{M}_{op}(e, y.n^{i+1})$
$\text{M}_{op}(e_1, e_2) \stackrel{\text{def}}{=} \text{M}(e_1, e_2)$

If op is $x.n := y$ then
let $e_1 = z.n^i$ and $e_2 = w.n^j$
$\text{M}_{op}(e_1, e_2) \stackrel{\text{def}}{=}$
$\quad i > 0 \wedge \text{M}(z, x) = k \wedge k < i \Rightarrow$
$\quad\quad \text{M}_{op}(y.n^{i-k-1}, e_2)$
$\quad j > 0 \wedge \text{M}(w, x) = k \wedge k < j \Rightarrow$
$\quad\quad \text{M}_{op}(e_1, y.n^{j-k-1})$
$\quad (i > 0 \rightarrow \text{M}(z, x) \geq \text{M}(z, e_1)) \ \wedge$
$\quad (j > 0 \rightarrow \text{M}(w, x) \geq \text{M}(w, e_2))$
$\quad\quad \text{M}(e_1, e_2) \leq \text{M}(e_1, x) \Rightarrow \text{M}(e_1, e_2)$
$\quad\quad \text{M}(e_1, e_2) > \text{M}(e_1, x)$
$\quad\quad\quad \text{M}(y, e_2) < \infty \wedge \text{M}(y, e_2) \leq \text{M}(y, x) \Rightarrow$
$\quad\quad\quad\quad \text{M}(e_1, x) + 1 + \text{M}(y, e_2)$
$\quad\quad\quad \text{M}(y, e_2) = \infty \vee \text{M}(y, e_2) > \text{M}(y, x) \Rightarrow \infty$

If op is $\mathsf{new}(x)$ then
$\text{M}_{op}(x, x) \stackrel{\text{def}}{=} 0$
$\text{M}_{op}(e, x) \stackrel{\text{def}}{=} \infty$
$\text{M}_{op}(x, e) \stackrel{\text{def}}{=} k, \quad k \in \mathbb{N}^+ \cup \{\infty\}$
$\text{M}_{op}(e, x.n^i) \stackrel{\text{def}}{=} *$
$\text{M}_{op}(x.n^i, e) \stackrel{\text{def}}{=} *$
$\text{M}_{op}(e_1, e_2) \stackrel{\text{def}}{=} \text{M}(e_1, e_2)$

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, trlT, trlG, trlO, and trlU translate transitions, guards, operations and updates, respectively.

fined in Figure 3, while untracked measures are non-deterministically assigned a value from $\mathbb{N} \cup \{\infty\}$.

The rules in Figure 3 defining the update functions should be read in the top-down way. The rule that matches first is applied. We only provide a detailed description for the translation of heap updates $x.n := y$ and omit other cases for brevity. Such heap updates are translated into updates of measures of the form $\mathrm{M}(z.n^i, w.n^j)$. 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. It is divided into three subcases. The first subcase handles the situation when x does not occur on the path from $z.n^i$ to $w.n^j$. Here, the measure remains unchanged. The second subcase deals with the situation when 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 third subcase accounts for the update eliminating any existing paths between $z.n^i$ and $w.n^j$. We present a soundness proof of the translation in the extended version of the paper [19].

Note that the provided updates of measures are precise with the exception of update expressions for new statements. Here, precision means that the evaluation of an update expression $\mathrm{M}_{op}(e_1, e_2)$ in a given state s determines the value of $\mathrm{M}(e_1, e_2)$ in the post state of s under operation op. Update expressions of new statements are not precise in this sense, because new statements translate into nondeterministic updates.

Each of the update functions $\mathrm{M}_{upd}(e_1, e_2)$ defines a set of guarded update expressions of the form $grd \Rightarrow exp$ with the following meaning. If grd is satisfied in the current state of the measure program then the next value of measure $\mathrm{M}(e_1, e_2)$ is determined by exp.

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.

Choosing measures to track. We determine the set of tracked measures M using a simple heuristic. Initially, we consider measures that are required for the precise translation of loop conditions. During the translation, additional measures are lazily taken into consideration if they occur in updates of existing tracked measures according to Figure 3. To ensure that the set M remains finite we only track measures of the form $\mathrm{M}(x, y)$ where x and y are program variables. Note that the precision of the inference algorithm is monotonic 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-guided refinement loop. In each iteration of the algorithm a termination checker is applied to check whether the measure program terminates under current precondition PRE. If the termination check succeeds then HEAPINFER stops and guarantees that the heap program is guaranteed to terminate under PRE. Otherwise, there exists a non-terminating computation in the measure program. The algorithm non-deterministically chooses one

of these computations: $st_1 \dots st_{m-1}.(st_m \dots st_n)^\omega$. Now there are two possible cases. First, the selected computation is spurious, *i.e.*, there is no corresponding computation in the heap program. Second, the computation is feasible in the heap program. To determine whether the counterexample is feasible, the algorithm chooses a guard grd_i from the loop segment $(st_m \dots st_n)$. Then, a safety checker is called to verify whether the negation of grd_i is an invariant of the heap program at location ℓ_i under the current precondition PRE.

If this safety check succeeds then we conclude that the found counterexample is spurious. In this case, we strengthen the guards of all transitions that start at ℓ_i in the measure program using the measure assertion $\neg grd_i$, and hence eliminate the counterexample from the measure program.

If 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). 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 a 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, then the inference algorithm will produce a precondition which is too strong. In this case the safety check in line 9 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 on program loops typically found in low-level system code.

Weakest preconditions of measure assertions. Algorithm HEAPINFER relies on an oracle wlp 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 3 to compute weakest preconditions for finite sequences of transitions. Assume that the current counterexample path π in the refinement loop is of the form $st_1 \dots st_{m-1}.(st_m \dots st_n)^\omega$. If the algorithm attempts to strengthen the precondition using a guard grd_i from a transition of the loop segment $(st_m \dots st_n)$, then we update precondition PRE as follows:

$$\text{PRE} := \mathsf{widen}(\text{PRE} \wedge \mathsf{wlp}(st_1, \dots, st_{i-1}, \neg grd_i)) \ .$$

The operator widen is a widening operator on measure assertions. $\mathsf{widen}(F)$ identifies a series of conjuncts $C(x.n^i), C(x.n^{i+1}), \dots$ in F and replaces them by the unbounded conjunction $\forall j \geq 0 : C(x.n^{i+j})$.

If one uses update expressions of measures to compute weakest preconditions then the only source for nondeterministic updates are new statements. We use a simple quantifier elimination procedure to eliminate the resulting universal quantifiers in weakest preconditions.

The algorithm HEAPINFER has a solid theoretical foundation. We briefly sketched soundness in the discussion above. Under assumption that the oracles for the termination check, safety check, and wlp computation always terminate, there exists a backtracking strategy on the nondeterministic choices (lines 7 and 8) such that the refinement loop in algorithm HEAPINFER always terminates. Finally, we identify a class of *regular* programs for which the algorithm HEAPINFER is complete. That is, it computes the *weakest* precondition for termination of the input heap program. The details are presented in the extended version [19].

5 Example

We illustrate the algorithm HEAPINFER on a simple, yet instructive example. The left-hand side of Figure 4 shows program TRAVERSE which traverses a singly-linked list. We apply algorithm HEAPINFER to program TRAVERSE with the singleton set of tracked measures containing only $\mathrm{M}(p, q)$. Executing line 1 in the algorithm yields the measure program P_{M} shown on the right-hand side of Figure 4. For legibility, we omit the non-deterministic updates of untracked measures. 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 computation $\epsilon.(\ell[1])^{\omega}$ that repeatedly executes the loop body according to case 1. There is only one position to choose in line 8 of the algorithm, namely, the one associated with location ℓ and guard $\mathrm{M}(p, q) = \infty$. As an assertion on states of program TRAVERSE, this guard means that q is not reachable from p. Obviously, the negated guard $\mathrm{M}(p, q) < \infty$ is not an invariant of program TRAVERSE at location ℓ. Hence, the condition in line 9 does not hold. In this case, the weakest precondition of the stem $\mathsf{wlp}_{\epsilon}(\mathrm{M}(p, q) < \infty)$ is again the assertion $\mathrm{M}(p, q) < \infty$. Thus, line 12 assigns PRE to $\mathrm{M}(p, q) < \infty$.

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 $\ell[2.2.1].(\ell[1])^{\omega}$. The loop part of this infinite trace is the same as for the previous counterexample. Thus, we again choose guard $\mathrm{M}(p, q) = \infty$. The condition in line 9 is again false. The weakest precondition of the negated guard $\mathsf{wlp}_{\ell[2.2.1]}(\mathrm{M}(p, q) < \infty)$ simplifies to the assertion

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

Line 12 updates the precondition PRE to:

$$\mathrm{PRE} \equiv \mathrm{M}(p, q) < \infty \wedge (\mathrm{M}(p, q) > 0 \ \vee \ \mathrm{M}(p, p.n) = 0 \ \vee \ \mathrm{M}(p.n, q) < \infty) \ .$$

The new precondition PRE means that q is reachable from p and either (1) p is different from q or (2) they are aliased and either (2.1) p has a self-loop or (2.2) 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.

In [19], we discuss additional example programs that manipulate singly- and doubly-linked lists. These examples are inspired by code fragments found in low-level system code, such as the example in Section 2.

ℓ : **do** $p := p.n;$ **while** $p \neq q$	ℓ : **do** $\text{M}(p,q) :=$ 1 $\text{M}(p,q) = \infty \Rightarrow \infty$ 2 $\text{M}(p,q) < \infty$ 2.1 $\text{M}(p,q) > 0 \Rightarrow \text{M}(p,q) - 1$ 2.2 $\text{M}(p,q) = 0$ 2.2.1 $\text{M}(p,p.n) = 1 \Rightarrow \text{M}(p.n,q)$ 2.2.2 $\text{M}(p,p.n) = 0 \Rightarrow 0;$ **while** $\text{M}(p,q) > 0$

Fig. 4. Program TRAVERSE and its associated measure program P_{M}.

6 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 [19] and a scheduling routine from the VAMOS kernel [17].

BOUNCER applies the BOHNE tool for symbolic shape analysis [24] to implement the oracle that checks assertion validity of heap programs [20, 23]. For proving termination of measure programs, BOUNCER applies the ARMC tool for proving termination of transition relations in linear arithmetic [18, 21]. The oracle for wlp 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:

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

The rewriting step allows one to 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 local use of measures, similarly 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.

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, A. Rybalchenko, and T. Wies. Heap assumptions on demand. Technical report, University of Freiburg, 2008. Available at http://www.informatik.uni-freiburg.de/∼wies/papers/HeapAssumptionsExtended.pdf.
20. A. Podelski and T. Wies. Boolean heaps. In *SAS*, 2005.
21. A. Rybalchenko. ARMC. http://www.mpi-sws.org/∼rybal/armc, 2008.
22. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 2002.
23. T. Wies. Symbolic Shape Analysis. Diploma thesis, Universität des Saarlandes, Germany, 2004.
24. T. Wies. The Bohne Tool. http://swt.informatik.uni-freiburg.de/wies/bohne, 2008.
25. T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. Verifying complex properties using symbolic shape analysis. In *HAV Workshop*, 2007.