Termination Analysis of Heap-Manipulating Programs by Abstract Interpretation

Master's Thesis Project Description

Lukas Neukom
Supervised by *Dr. Caterina Urban*

1 Introduction

Turing proved in 1936 that the Halting Problem is undecidable. A common approach to relax the problem is to allow "unknown" as a result. This means that the problem can now be solved but the challenge is to return "unkown" as seldom as possible.

Since termination guarantees are very important for software reliability much research has been done to develop practical termination provers. Over the last decades powerful termination provers such as AProVE [2], Ultimate Automizer [1] and FuncTion [3] have been developed. Since 2003 there exists an annual termination competition[1] which features categories such as termination of term rewriting, termination of java bytecode programs, termination of C programs and more.

FuncTion is a static analyzer which is able to infer piecewise-defined ranking functions for programs written in a C-like language. It is based on abstract interpretation, a mathematical theory of sound approximation of the semantics of a computer program.

2 Core Goals

FuncTion currently provides only limited support for proving termination of heap-manipulating programs. The core goal of this project is the development of a new static analysis based on abstract interpretation for proving termination of heap-manipulating programs.

A major difficulty is that the termination often depends on the shape of the heap which might be an unbounded structure. For example, the program shown in Listing 1 creates an infinite number of possible heaps.

```
List createList(int n) {
  List l = null;
  while (n > 0) {
    l = new List(l);
```

[1] http://termination-portal.org/wiki/Termination_Competition

```
    --n;
  }
  return l;
}
```

Listing 1: Creating a acyclic singly linked list

This means that an analysis has to summarize the heap while still retaining information needed for termination. Another problem is that languages with pointer arithmetic have undefined behavior for memory unsafe programs (e.g. out-of-bound writes on arrays). Because of this some simplifications will be made and in a first step, we will only consider programs where termination depends on the size of an array or a singly linked list. The core goals of the project include the following steps:

Literature Get an idea of the state-of-the-art termination analysis methods and heap abstractions.

Examples Collect interesting examples and find arguments for their termination (or non termination) to see what information needed as part of the abstraction. One such example is shown in Listing 2.

```
void traverseList(List l) {
  while (l != null) {
    l = l->next;
  }
}
```

Listing 2: Traversing a singly linked list

This program terminates on acyclic singly linked lists since the length of the list pointed to by `l` decreases in each iteration. Typical programs that follow this pattern are min, max, contains, fold, reduce, etc. A more complicated example, which finds a solution to a specific instance of Josephus problem[2], is shown in Listing 3.

```
List josephus(List l) {
  while (l != l->next) {
    l->next = l->next->next;
    l = l->next;
  }
  return l;
}
```

Listing 3: Josephus problem

The original assumption for the input of this counting-out game is that the input is a cycle. But the algorithm terminates for every possible input. Since either the length of the list decreases if `l` points to a node not contained in a cycle or the size of the cycle decreases. This example poses the additional problem that the structure of the heap is changed during execution.

[2] `https://en.wikipedia.org/wiki/Josephus_problem`

Design and formalization of the termination analysis Based on the examples we will develop an abstraction which can be used to prove termination for programs with the mentioned simplifications.

We are investigating an approach similar to the termination analysis currently implemented in FuncTion. Using a backwards analysis sufficient preconditions are searched for a program to terminate. We are working on a heap abstraction that allows us to reason about cyclic and acyclic portions of singly linked lists. The designed termination analysis then has to be formalized.

Implementation The termination analysis is then implemented in FuncTion. This includes extending the parser with needed constructs for heap manipulation.

Experimental evaluation As last step the implemented termination analysis will be compared to state-of-the-art termination analyzer such as APoVE and Ultimate Atomizer.

3 Possible Extensions

Termination depending on values A possible extension is the set of program where termination depends on values stored in the heap. Examples are shown in Listing 4 and 5.

```
int A[3] = {0, 0, 0};
while (A[2] < 10) {
  A[2]++;
}
```

Listing 4: Program where termination depends on a value of the array

```
void traverseList(List l) {
  while (l->value != 5) {
    l = l->next;
  }
}
```

Listing 5: Program where termination depends on a value of the list

This means that we have to extend our abstraction to be able to reason about the possible values stored in the heap.

Other data structures Another extension is to generalize the analysis to other data structures such as doubly linked lists, trees or graphs. Listing 6 shows an example program that finds the minimum in a binary search tree. Example graph algorithms are breath-first search, shortest path, etc.

```
int min(Node n) {
  while (n->left != null) {
    n = n->left;
```

```
  }
  return n->value;
}
```

Listing 6: Minimum in a binary search tree

Pointers Another extensions is to allow pointers together with pointer arithmetic as in C or C++. Since memory unsafe programs can have undefined behavior those programs would have to be checked for memory safety first. Listing 7 shows an example where accessing a location outside of an arrays boundary results in undefined behavior[3].

```
int A[4] = {0, 1, 2, 3};
int *p = arr + 5;
```

Listing 7: Memory unsafe program in C++

Other liveness properties Another extensions is to generalize the analysis to other liveness properties, in particular guarantee ("something good occurs at least once"), recurrence ("something good occurs infinitely often") or persistence ("something good eventually happens continuously") properties [4].

References

[1] Matthias Heizmann et. al. Ultimate automizer with smtinterpol. In *TACAS*, pages 641–643, 2015.

[2] Thomas Ströder, Cornelius Aschermann, Florian Frohn, Jera Hensel, and Jürgen Giesl. Aprove: Termination and memory safety of c programs. In *TACAS*, pages 417–419, 2015.

[3] Caterina Urban. FuncTion: An abstract domain functor for termination (competition contribution). In *TACAS*, pages 464–466, 2015.

[4] Caterina Urban and Antoine Miné. Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation. In *VMCAI*, pages 190–208, 2015.

[3] `http://en.cppreference.com/w/cpp/language/ub`