Integrating Static Analysis and General-Purpose Theorem Proving for Termination Analysis

Panagiotis Manolios and Daron Vroon
Georgia Institute of Technology
College of Computing
801 Atlantic Drive
{manolios,vroon}@cc.gatech.edu

ABSTRACT

We present emerging results from our work on termination analysis of software systems. We have designed a static analysis algorithm which attains increased precision and flexibility by issuing queries to a theorem prover. We have implemented our algorithm and initial results show that we obtain a significant improvement over the current state-of-the-art in termination analyses. We also outline how our approach, by integrating theorem proving queries into static analyses, can significantly impact the design of general-purpose static analyses.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification, F.4.1 [Mathematical Logic and Formal Languages]Mechanical Theorem Proving

General Terms: Algorithms, Verification.

Keywords: Static Analysis, Theorem Proving, Termination, Liveness, ACL2.

1. OVERVIEW AND MOTIVATION

Termination is of critical importance in the realm of software analysis, especially in software verification. Establishing termination allows us to strengthen a partial correctness result to a total correctness result. Code reachability and other path analyses sometimes assume the termination of loops and recursive functions, often producing results that are not correct unless termination holds. Termination analysis also plays an important role in the analysis of reactive systems, non-terminating systems that engage in ongoing interaction with their environments (*e.g.*, operating systems or networking protocols). In this case, termination arguments are used to prove liveness properties such as the absence of deadlock or livelock, by establishing that some desired behavior is not postponed forever.

In this paper, we present a termination analysis algorithm that issues queries to a general-purpose theorem prover to obtain more precise analyses than is possible with traditional static analysis techniques. The theorem prover is called in a controlled, time-limited way; this guarantees that it can be used in an automatic fashion as a black-box for static analysis. In addition, the theorem prover can be given general queries, even ones that are over undecidable fragments of logic. If the query can be resolved, it improves the precision of our analysis. If not, then our static analysis algorithm proceeds as it normally would without the theorem prover.

We believe that general-purpose theorem proving will play a vital role in termination analysis of software written in real, feature-rich programming languages. We tested several state-of-the-art termination analyses based on current static analysis methods on software written in the ACL2 programming language, a first-order functional language, and found that existing methods establish termination in only a very limited number of cases. We then implemented a preliminary version of our termination analysis algorithm using the ACL2 theorem prover as a black-box, and found a substantial increase in accuracy over previous methods. In addition, our analysis gives abstract counter-examples when it fails, allowing the user to refine the analysis to find a termination proof, if one exists.

In addition to improving termination analysis, we believe our method of querying properly configured general-purpose theorem provers can be useful for static analysis in other settings. For example, in path-sensitive analyses, the theorem prover can help eliminate impossible paths by proving that incompatibilities exist between various branching conditions. Overall, we think that such tasks as compiler optimization, bug isolation, automatic test generation, and formal software verification can all benefit from our technique.

After discussing related work, we present an overview of our preliminary results in Section 2. We conclude with a discussion of impact and future work in Section 3.

1.1 Related Work

Most of the current state-of-the-art tools in termination analysis do not make use of general-purpose theorem proving. Instead, they employ static analysis and decision procedures to reason about a subset of the termination problem. These include tools for analyzing the termination of programs with linear and limited polynomial integral behavior [4, 5, 17]. All of these approaches fail for programs that include looping behaviors that do not fit their limited scope. For example, they cannot handle recursive function definitions or loops whose termination depends on data structure invariants.

ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

```
define f(x) =
  if not(integerp(x))
     or (x = 0)
    then 0
    else if x < 0
           then f(x+1)
           else f(x-1)
```

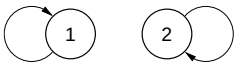

Figure 1: Definition and CCG for `f`

Currently, the most advanced procedures for proving termination in pure functional languages and term rewriting systems are the size-change principle [12] and the dependency pair method [2].

The size-change principle involves giving objects a well-founded "size," and showing that for all possible infinite execution paths, there exists a value that decreases infinitely often. The strength of this method lies in its backend which can comprehensively analyze all of the infinite paths of a function. We use the ideas in the backend in our own termination analysis. However, the major limitation of the size-change principle is its frontend, *e.g.*, this analysis does not account for the conditional branches that govern when a callsite is reached, which is almost always crucial for termination analysis. Consider, for example, the function, `f`, given in Figure 1. Since the size-change principle does not consider the tests of `if`-statements, it must consider infinite state sequences that cannot occur, including the sequence that alternates between the two recursive calls. In this case, `x` is alternately increased by 1 and decreased by 1. Thus every two steps, the value of `x` returns to its original value, leading to an infinite loop. The size-change principle therefore cannot prove the termination of this function. However, this path is not actually possible, and the function clearly terminates.

Recently, the size-change principle and dependency pair method were combined into a single termination analysis, implemented in a tool called AProVE, that is more powerful than either technique by itself [18, 8]. However, this analysis is aimed at logic programming languages and term rewriting systems; when we applied it to functional languages we found its effectiveness in determining termination to be quite limited.

One example of the use of general-purpose theorem proving for termination analysis is the ACL2 theorem proving system. ACL2 consists of a programming language, logic, and theorem prover [10, 9]. The programming language can roughly be thought of as an applicative superset of a subset of Common Lisp. The logic axiomatizes the semantics of the programming language, thereby allowing the user to reason formally about her programs. The theorem prover makes it possible for the user to provide only an outline of a proof, with ACL2 filling in the details. ACL2 is used by a large community, and has been used for some impressive industrial problems (for more about ACL2 and its applications, see [11]).

Termination plays a key role in ACL2, as every defined function (using the definitional principle) must be shown to terminate before ACL2 will admit it. To do this, ACL2 attempts to guess a well-founded measure for the function and to prove that it decreases with each recursive call. It does this by using a simple static analysis to guess a measure, followed by a call to the theorem prover to attempt to prove that the measure always decreases. Our recent work has involved improving ACL2's ability to successfully complete such proofs [13, 14, 15, 16]. If ACL2 cannot guess a measure or if it cannot prove that the measure it guesses decreases, the user must provide her own well-founded measure and must use the theorem prover to show that it decreases. This is often a non-trivial task, and due to ACL2's limited static analysis, it is required for all but the most basic of recursive schemes.

The combination of theorem proving and static analysis is used in ESC/Java [7] and the SLAM project [3]. However, neither SLAM nor ESC/Java can reason about the termination of the code they analyze.

2. PRELIMINARY RESULTS

Our termination analysis is designed for first-order, purely functional languages. It includes support for a wide variety of data-types, including strings, symbols, integers, rationals, characters, lists, trees, and recursive data-types. It also supports many types of control flow used in such languages, including recursion and mutual recursion. In other words, our analysis is designed for real, complete, feature-rich programming languages used in practice. Our implementation operates on software written in the ACL2 programming language, which is used in numerous industrial and academic settings. In this section we give an overview of our analysis and discuss the implementation and experimental results.

2.1 Termination Analysis

Given a set of definitions, our analysis starts by determining the relevant branching conditions that lead to the execution of a callsite, a function call that could potentially be involved in a loop (*e.g.*, a recursive or mutually-recursive call). These predicates, along with the name of the function containing the callsite and the call itself comprise the *calling context*.

Next, we construct a reachability graph of the calling contexts, which we call the *calling context graph (CCG)*. This is similar to a static callgraph, but the use of theorem proving allows us to get a far more detailed picture of the system behavior. Recall that a callgraph has function names for nodes, and an edge from f to g if f contains a call to g. In the CCG, the nodes are calling contexts, and there is an edge from c_1 to c_2 if there are values for the parameters of the function containing c_1 such that the conditions under which c_1 is called are met, and the call c_1 results in the conditions under which c_2 is called being met (in which case the call of c_2 is executed next). We use the theorem prover to attempt to prove that a c_1 call cannot lead to a c_2 call, in which case we do not add an edge in the CCG. If the theorem prover fails to prove this conjecture or times out, then we add the edge to the graph. This gives us an over-approximation of system behavior, *i.e.*, no realizable system behavior is ignored.

Consider the CCG in Figure 1 for the function `f`. The callgraph for this function has one node, `f`, with a self-loop. All this tells us is that function `f` can call itself. Note that the CCG has 2 nodes, one for each calling context, and each has a self loop. There are no edges between the two contexts. This is because if `x` is a negative integer, `x+1` is either a negative integer or 0, and if `x` is a positive integer, `x-1` is either a positive integer or 0. This gives us the added

information that the execution of one context only leads to further executions of the same context. This is a key insight for the termination analysis, since it rules out the possibility that the value of `x` infinitely fluctuates between a finite number of values.

What remains is to determine that there is no execution of the program that corresponds to an infinite path through the CCG. Since every infinite path through the CCG must have an infinite suffix with nodes in exactly one non-trivial strongly-connected component (SCC), we separate the CCG into its non-trivial SCCs, and continue with our analysis on each SCC. For the CCG in Figure 1, this means that we can analyze each of the two self-loops separately.

At this point, a set of *calling context measures (CCMs)* is created for each calling context. These are expressions over the parameters of the function containing the context that return a value in a well-founded domain. This is done through a combination of general heuristics and reusable system-defined patterns. By default, a CCM is created for each parameter of the parent function of the context by applying to the parameter a fixed mapping from ACL2 objects into the natural numbers. For each edge in the CCG, a *CCM function* is created that takes in a CCM from each adjacent context, and returns either `>`, `>=`, or `none`. The function must follow the rules that if `>` is returned, then if the conditions of both contexts are met, the CCM from the source of the edge must always be greater than the CCM from the sink of the edge. Likewise, if `>=` is returned, it must be the case that under the conditions of both contexts, the CCM from the source of the edge must never be less than that of the sink of the edge. Once again, the theorem prover is used to determine the values returned by each CCM function.

The annotated CCG is now analyzed to determine if for every infinite path through the CCG, there is at least one CCM that never increases and which also decreases infinitely often. This analysis can be accomplished using an approach that is similar to the analysis of "size change graphs," which is handled by the backend of the size-change approach. If our analysis terminates, we are done; otherwise, we report a failure to prove termination and provide an abstract counterexample that shows a repeating sequence of calls that our analysis cannot prove terminating.

2.2 Preliminary Experimental Results

We have implemented a preliminary version of our algorithm using the ACL2 theorem proving system. Our implementation includes the analyses described in the previous section, as well as a few other analyses that are still under development. We ran the implementation on the library of books distributed with the ACL2 theorem prover, which is a collection of published projects by ACL2 users. We focused on the 84 functions for which the users were forced to provide their own measures or proof hints. Of these, our algorithm automatically proved 60 of them terminating with no user interaction (and without the original user provided hints and measures). The following examples are taken from these experiments.

The example in Figure 2 follows a common induction scheme that was used in several of the functions in our experiments. The `upto` function, given integers `i` and `max` such that `i` $\leq$ `max`, counts from `i` to `max`, returning the difference between `max`+1 and `i`. It does this by incrementing `i` until it is greater than `max`. Neither the current version of ACL2 nor the other methods cited in Section 1.1 can automatically prove this function terminating. Our algorithm, on the other hand, does prove `upto` terminating, choosing `nfix((max + 1) - i)` as a CCM, where `nfix(x)` returns `x` if `x` is a natural number and `0` otherwise.

```
define upto (i, max) =
  if integerp(i) and
     integerp(max) and
     i <= max
    then upto(i+1, max) + 1
    else 0
```

Figure 2: `upto` function definition

```
define h(x, y, i) =
  if zp(x) or zp(i)
    then
      list(x, y, i)
    else
      list(h(x-1, y-1, g(y-1)),
           h(x, f(y), i-1))
```

Figure 3: `h` function definition

The code in Figure 3 calls several functions, `f` and `g`, whose definitions are not known during termination analysis. These functions are so-called *encapsulated*, meaning that only certain properties of the functions are known, but not their definitions. Encapsulation is a powerful and useful mechanism for information hiding, allowing one to prove theorems that apply to a wide range of functions (*e.g.*, to functions that satisfy commutativity and associativity). What we know about the encapsulated functions `f` and `g` is that `g` always returns an integer no less than 2. In addition, the `zp` function returns false when given a positive integer, and true otherwise. Thus, there are two recursive calls made when both `x` and `i` are positive integers. In the first, `x` and `y` are decreased by one, but we do not know what happens to `i`. In the other, `x` remains the same, `i` is decreased by one, and we do not know what happens to `y`.

As before, none of the current state-of-the-art techniques presented in Section 1.1 can automatically prove termination. For example, in ACL2's current analysis, the user must provide a measure that decreases with every recursive call. A valid measure is the tuple $\langle$`nfix(x)`,`nfix(i)`$\rangle$, where the well-founded relation used is the lexicographic ordering on tuples of natural numbers. In the first recursive call, `nfix(x)` is decreased, so the overall measure decreases even though we do not know what happens to `nfix(i)`. In the second case, `nfix(x)` remains the same and `nfix(i)` decreases, so the overall measure decreases.

Our method, by examining the conditions under which the recursive calls are made, can determine that it is dealing with positive integers, and therefore it determines when the CCMs are decreasing. This leads to the construction of an annotated CCG for which our algorithm can determine that any infinite path reaching the first context infinitely often leads to `x` decreasing infinitely often, and that any path reaching the second context infinitely often causes `i` to decrease infinitely often. Thus, since `x` and `i` range over a well-founded structure, the function must terminate.

3. IMPACT AND FUTURE DIRECTIONS

We have presented an overview of a technique for proving termination of programs written in a first-order functional language. We use a static analysis algorithm that includes queries to a general-purpose theorem prover. By combining static analysis with theorem proving, we attain increased precision. Our initial experimental results show that such an analysis significantly improves upon the current state-of-the-art in termination analysis, whether based on standard static analysis or theorem proving techniques.

Based on these initial results, we are currently developing a more extensive suite of techniques for utilizing theorem proving in the context of termination analysis. These techniques allow us to further refine CCGs, thereby attaining increased precision. We are also considering methods for more aggressive abstractions that prune irrelevant and redundant parts of the CCG.

Our technique currently presents an abstract counterexample when it cannot prove termination. This may or may not be an actual counterexample to the termination of the system. To handle the case where it is not a true counterexample, we are developing techniques to assist the user in proving termination. We are also investigating methods for automatically determining the feasibility of abstract counterexamples. If the abstract counterexample generated is spurious, we would like to refine our termination analysis in a way that rules out this counterexample in the future. That is, we are developing a termination analysis algorithm that uses the counterexample-guided abstraction-refinement framework.

Another (more long-term) goal of our work is to extend our analysis to deal with imperative languages such as C. We plan on doing this by taking advantage of various static analyses, including data-flow, control-flow, alias analysis, etc., and taking advantage of the fact that Static Single Assignment (SSA), the popular intermediate language used for the analysis and optimization of imperative programs, is conceptually a kind of pure functional language [1]. This will allow us to generate CCGs for imperative languages that we can then analyze using the framework described in this paper.

From our preliminary experimental results, we expect that our work will significantly extend what can be done with termination analysis. By incorporating queries to a general-purpose theorem prover, our analyses can be used to reason about the many possible sources of looping behavior in feature-rich programming languages, including arbitrary arithmetic operations and data structures. The result is increased accuracy when automatically reasoning about termination. Such reasoning is useful in numerous software analyses and verification techniques, ranging from proving total functional correctness, to determining reachability, to various path analyses, to reasoning about reactive systems.

Finally, we expect that techniques we have presented, in which general-purpose theorem provers are queried in the context of static analysis algorithms and methods, can be used to construct improved compiler optimizations, bug isolation techniques, automatic test generation methods, and software verification techniques, because the queries to the theorem prover can only improve the precision of the underlying static analyses.

4. REFERENCES

[1] Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, 1998.

[2] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[3] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Spin 2001, Workshop on Model Checking Software*, volume 2057 of *LNCS*, pages 103 – 122. Springer–Verlag, May 2001.

[4] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In Cousot [6], pages 113–129.

[5] Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In Cousot [6], pages 1–24.

[6] Radhia Cousot, editor. *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005*, volume 3385 of *Lecture Notes in Computer Science*. Springer, 2005.

[7] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[8] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA-04)*, volume 3091 of *LNCS*, pages 210–220. Springer–Verlag, 2004.

[9] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[10] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[11] Matt Kaufmann and J Strother Moore. ACL2 homepage. See URL `http://www.cs.utexas.edu/users/moore/acl2`.

[12] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM Press, 2001.

[13] Panagiotis Manolios and Daron Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal Of Automated Reasoning*. To Appear.

[14] Panagiotis Manolios and Daron Vroon. Algorithms for ordinal arithmetic. In Franz Baader, editor, *19th International Conference on Automated Deduction – CADE-19*, volume 2741 of *LNAI*, pages 243–257. Springer–Verlag, July/August 2003.

[15] Panagiotis Manolios and Daron Vroon. Ordinal arithmetic in ACL2. In Matt Kaufmann and J Strother Moore, editors, *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, July 2003. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/`.

[16] Panagiotis Manolios and Daron Vroon. Integrating reasoning about ordinal arithmetic into ACL2. In *Formal Methods in Computer-Aided Design: 5th International Conference – FMCAD-2004*, LNCS. Springer–Verlag, November 2004.

[17] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.

[18] Reneé Thiemann and Jürgen Giesl. Size-change termination for term rewriting. Technical Report AIB-2003-02, RWTH Aachen, January 2003.