Integrating CCG Analysis into ACL2[⋆]

Matt Kaufmann[1], Panagiotis Manolios[2], J Strother Moore[1], and Daron Vroon[2]

[1] Department of Computer Sciences
University of Texas at Austin
{kaufmann,moore}@cs.utexas.edu

[2] College of Computing
Georgia Institute of Technology
{manolios,vroon}@cc.gatech.edu

1 Introduction

ACL2 [6–8] is a powerful, industrial strength theorem proving system, which has been used on numerous verification projects. It is part of the Boyer-Moore family of provers, for which its authors received the 2005 ACM Software System Award. Termination plays a central role in ACL2's logic. It is used to demonstrate the logical consistency of function definitions and allows for the admission of an induction scheme that mirrors each function's recursive structure.

In previous work we introduced calling context graphs (CCGs) and showed how they can be combined with theorem proving queries to provide a powerful method for proving termination of programs written in first order, functional programming languages [15]. In this paper we describe work we are doing to integrate CCG-based termination analysis into ACL2.

We begin in the next section by providing relevant background on ACL2 and its current approach to termination analysis as well as the CCG approach to termination analysis. Section 3 then describes our work on extending ACL2 to include the CCG approach, with a focus on issues encountered. We conclude in Section 4.

2 Background

We begin with a brief overview of ACL2. We then contrast ACL2's existing termination analysis with the CCG-based approach.

2.1 ACL2 Overview

ACL2 ("A Computational Logic for Applicative Common Lisp") is an environment for theorem proving and functional programming that supports a first-order logic. ACL2 has been used on numerous applications, some of which are described in [6] and in the papers distributed as part of the periodic ACL2 workshops, the proceedings of which may be found via the Workshops link on the ACL2 home page [8]. Among these applications, several verification efforts are mentioned in [17], including register-transfer level and algorithmic descriptions of commercial floating point units [18–21], microcode-related verification for a Motorola digital signal processor [1,2], process separation for the Rockwell Collins AAMP7, a JVM bytecode [16] and Sun bytecode verifier [10], a verified model checker [11], and ordinal arithmetic algorithms [12–14].

Although ACL2 is an automated reasoning tool, its use is typically quite interactive, as the user breaks theorems into subsidiary lemmas. The goal is to produce ACL2 input files called *books*, which are collections of *events*, typically definitions and theorems for a common topic such as arithmetic, set theory, or processor verification. The book *certification* process attempts to *admit* each of the book's events, by proving its theorems and proving termination for its recursive definitions.

2.2 Termination Analysis in ACL2

Termination analysis in ACL2 is based on the notion of a *measure* that decreases on each recursive call. For example, the following simple recursive definition of integer exponentiation terminates because the absolute value of the exponent argument decreases on each recursive call.

[⋆] This research was funded in part by NSF grants CCF-0429924, IIS-0417413, and CCF-0438871, as well as the DARPA and NSF Grant CNS-0429591.

```
define f(x) =
   if !(intp(x)) or x < 2 then 0
   else if x mod 2 = 1 then f(x+1)
   else 1 + f(x/2)
```

```
if !(intp(x)) or x < 2 then 0
else if x mod 2 = 1 then 2 * x + 3
else 2 * x
```

1. $\langle$f, {intp(x), 2 $\leq$ x, x mod 2 = 1}, f(x+1)$\rangle$
2. $\langle$f, {intp(x), 2 $\leq$ x, x mod 2 $\neq$ 1}, f(x/2)$\rangle$

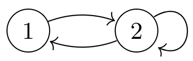

Fig. 1: Definition (top left), measure (top right), calling contexts (bottom left), and CCG (bottom right) for a function f.

```
define expt(base,exp) =
   if !(intp(exp)) or exp=0 then 1
   else if !(intp(base)) or base=0 then 0
   else if exp>0 then base*expt(base,exp-1)
   else expt(base,exp+1)/base
```

Note that at the first recursive call expt(base,exp-1), we know that the measure |exp| decreases (*i.e.*, |exp-1| < |exp|) because this is logically implied by two of the ruling if-tests, intp(exp) and exp>0 (the first is not negated because of the context of the call). Similarly, for the second recursive call, the conjunction of the ruling if-tests, intp(exp), ¬(exp=0) and ¬(exp > 0), implies |exp+1| < |exp|.

In general, the set of *rulers* of a subterm u in a given term is defined recursively. If R is the set of rulers of a subterm u in a term y, then the set of rulers of u in [if w then y else z] is the result of adding w to R, and the set of rulers of u in [if w then z else y] is the result of adding the negation of w to R. The set of rulers of u in other than an if-then-else term is the empty set.

ACL2 implements the following definitional principle, which we state for the recursive definition of a function f but which extends naturally to nests of mutually-recursive definitions. The principle requires the existence of a relation $\prec$ with domain, D, for which ACL2 can prove well-foundedness (induction), along with a *measure* term m,whose free variables are among the formal parameters of a given definition, for which ACL2 can prove $m \in D$. Moreover, there is the following proof obligation for each (recursive) call of f in the body, b, of the definition. Let $f(u_1, ..., u_k)$ be a subterm of b, let r be the conjunction of the set of rulers of this subterm in b, and let σ be the substitution mapping the i^{th} formal x_i of f to u_i. Then the corresponding proof obligation is that the implication $r \Rightarrow (m/\sigma \prec m)$ be provable in the current ACL2 environment.

As an aside, we note that under the conditions above, ACL2 soundly stores a corresponding induction scheme. Roughly, this scheme allows one to prove a formula φ by splitting into cases according to the set of branches through the top-level if structure of the body of the definition of f, where when proving φ under such a branch R we are allowed to assume φ/σ, for any substitution σ mapping x_i to u_i for a recursive call $f(u_1, ..., u_k)$ ruled by R.

The ACL2 user can provide the measure and well-founded relation for a given definition. If none is provided, then ACL2 uses heuristics to pick a formal parameter x for which the default measure is the size of x (for an appropriate notion of "size," for example so that the size of a pair is greater than the size of each component, the size of a natural number is itself, and so on).

2.3 CCG-Based Termination Analysis

In the preceding section we describe the current termination analysis used by ACL2, but it often fails to automatically establish termination. Consider the example on the left in Figure 1. Here, there are two recursive calls. If x is an odd integer greater than 1, we call f on x+1. If it is an even positive integer, we call f on x/2. ACL2 cannot automatically prove this function terminating, because it cannot find a measure that decreases both when x is odd and when it is even. Such a measure must be given by the user, and is shown on the right in Figure 1. Note that it is as complicated as the function itself. In [15], Manolios and Vroon describe a new termination analysis based on *calling* *context graphs* (CCGs), which is significantly more automated than the current ACL2 approach. Here we informally describe the core idea of the analysis.

Fix a set of mutually recursive functions, $\{f_1, f_2, \ldots f_n\}$. For each recursive call, e in a body of some function f_i, we form a *calling context*, $\langle f_i,\ R,\ e\rangle$, where R is the set of rulers of e in f_i. A CCG is a graph whose vertices are calling contexts, such that if there exists a call to function f_i that leads to the execution of a call, e, of a function f_j and ends up at another recursive call, e' in the body of f_j, then there is an edge from the context for e to the context for e' in the CCG. Intuitively, one can think of the CCG as being similar to a call graph, but at much finer granularity. That is, instead of telling us how execution leads from one function to another, a CCG tells us how the execution leads from each recursive callsite to another. The contexts and CCG for `f` are given in Figure 1.

Now recall the notions of *measure* and *well-founded relation* from the preceding section. Suppose we can assign a set of measures — called *calling context measures*, or CCMs — to each context such that every infinite path through the CCG has a corresponding sequence of CCMs that never increase and decrease infinitely often. It follows that every computation must then terminate. Theorem proving plays a key role in this analysis as it is used to prune edges from CCGs and to determine when CCMs are nonincreasing or decreasing as we traverse edges in CCGs.

In addition to this analysis, the CCG analysis provides a way to *absorb* a context into the CCG by merging it with each of its successors in the CCG. The resulting contexts represent two steps through the original CCG, giving us a more precise analysis. For example, our analysis would absorb context 1 in Figure 1. The new context then contains the call `f((x+1)/2)`. Since `(x+1)/2` $<$ `x` under the rulers of the call in the new context, our analysis can easily prove termination.

Our algorithm uses heuristics to pick CCMs, together with a sufficient condition for the above path-related criterion that is based on [9]. We ran our CCG implementation on the more than 10,000 functions of the ACL2 regression suite. It successfully proved termination for 98.7% of the functions with no user input. See [15] for details.

3 Integrating CCG Analysis and ACL2

As we saw in Section 2.1, termination plays an important role in ACL2's logic. A great deal of care must therefore be taken when altering ACL2's termination analysis. Otherwise, there is a danger of undermining the soundness of the system. In addition, the ACL2 theorem prover relies on information collected when functions are analyzed and admitted in order to guide and control future proof efforts. Changes to ACL2's termination analysis can interfere with this process and can easily render the system unusable. In this section, we discuss several challenges related to these issues that we have encountered while integrating CCG-based termination analysis into ACL2, as well as our solutions to those challenges.

In order to avoid infinite expansion of function definitions during proof attempts, ACL2 chooses parameters of each recursive function to be *controllers*, which are used heuristically when deciding whether or not to expand a function definition. In general, this strategy is very effective at gauging when definitions should be expanded during proof attempts. A key aspect of this is the choice of controllers. If too many controllers are chosen, the heuristics will be too restrictive, and the definition will not be opened when it needs to be. If there are too few, or the wrong ones are chosen, infinite expansion may result.

Currently, ACL2 chooses the controllers of a recursive function to be the parameters appearing in the measure used to prove termination. Since CCG-based termination analysis does not use a single measure, we needed to find a new way to choose controllers. The obvious first choice is the CCMs used in the analysis. However, while heuristics are used to choose CCMs, we also include the "size" of every parameter (as defined by the function `acl2-count`) as a CCM. Therefore, all parameters would be controllers if we were to use all the CCMs. However, even if all the parameters are used in CCMs, not all of them are necessarily relevant to the termination proof. By altering our CCM algorithm for finding the infinite decreasing sequences of CCMs, we were able to report which CCMs are used to prove termination. We then label the parameters used in these CCMs as controllers for the function containing the corresponding context.

Context absorption causes another challenge related to controller selection. When a context is absorbed, it is no longer represented in the CCG, having been replaced by the new merged contexts. Therefore no CCMs are reported for this context, which could lead to missing controllers for a function and infinite expansions. To address this challenge, we added *CCM propagation* to our controller analysis. This involves adding CCMs for an absorbed context that correspond to the CCMs of its successors in the original CCG. For example, if there is an absorbed context with call `f(g(x,y), h(z))`, and the first formal of `f` is a controller, then `g(x,y)` is added to the relevant CCM list of the absorbed context, and `x` and `y` are added to its list of controllers.

Using these techniques, we choose controllers that are comparable to the choices made using the measure-based approach. Rerunning old ACL2 books using the new termination method, we have yet to see theorems that went though before but did not go through under the new system due to controller-related issues.

The most challenging problem we have faced arises from how we use the theorem prover during CCG analysis. The typical usage of ACL2 involves calling the theorem prover to verify a conjecture that the user believes is valid. For CCG analysis, on the other hand, we use ACL2 as a black box for making queries, and often those queries are not valid. Therefore, we want to determine if ACL2 can prove a query easily, and if not, it is safe for our purposes to assume it is not valid. We do this when constructing the CCG to determine if we can prune an edge from the graph. We also use this technique to determine if a CCM is decreasing or non-increasing compared to some previous CCM on the path.

This becomes a problem in the presence of encapsulation or book certification. Encapsulation allows the user, among other things, to hide lemmas. For example, the user may need a particular lemma to get a proof to go through, but that lemma may not be generally useful, and may even cause problems later on. In this case, the user can employ encapsulation to use and then immediately hide the lemma. This can be done on a larger scale with books. Each book is certified once, then subsequently can be loaded without rerunning proofs. As with encapsulation, events such as theorems can be declared to be local, and will not be visible outside the book.

In order to deal with local events, the theorem prover typically makes two passes over the encapsulated events or books. In the case of encapsulated events, all the events are run in order to verify that they are valid. Then the theorem prover makes a second pass, this time skipping the proofs and loading only non-local events into the current logical world. Likewise, all the proofs and local events in a book are processed at certification time, but proofs are skipped and only non-local events are added to the logical world when the book is loaded.

This works well when the justification for termination can be determined statically using only the definition of the function. For example, with functions admitted using the measure-based termination analysis, the measure is determined statically from the function definition or is provided by the user; so the function can be loaded, and the controllers calculated from the measure, without reproving termination. In the case of CCG-based termination, on the other hand, theorem prover queries are used to find the relevant CCMs, which are necessary for the calculation of the controllers. In addition, this analysis depends on more than the function definition, e.g., it depends on what libraries are loaded. Therefore, we cannot simply skip CCG-based termination analysis when making the second pass over encapsulated events or books.

A first approximation at a solution to this problem is simply to run the CCG analysis during both passes of an encapsulation or book certification. However, local lemmas are not loaded during the second pass, so the CCG analysis may take place in a different logical environment during the second pass. This can lead to different CCG analyses, and even result in a successful termination analysis on the first pass, yet a failed analysis on the second (e.g., if the termination analysis depends on a local event).

The solution to this problem is provided by a new ACL2 feature called `make-event`. The idea behind this feature is that it allows users to compute events. That is, based on the current environment, a new event is made. An important feature of `make-event` is that the new event is computed once and then saved. Thus if ACL2 makes a second pass over the `make-event` in a different environment, the same new event is created. This alleviates the problem caused by the two pass system employed by ACL2 for encapsulation and books.

To use this feature to our advantage, we alter ACL2's built in definitional utility, `defun`, so that it can be told explicitly which CCMs are important to the termination analysis. Then, when given a definition without this hint, we run the CCG-based analysis, compute the relevant CCMs, and use `make-event` to create a new `defun` in which the these CCMs are explicitly given as a hint. Then, when loading a book or making a second pass over an encapsulation, ACL2 can forgo the CCG analysis and use the CCMs given to calculate the controllers for the function. More information on `make-event` may be found in the ACL2 documentation [8] starting with Version 3.0, and in [5].

4 Conclusions

We have explored the question of integrating CCGs, a new and powerful termination analysis method, into ACL2. Termination analysis plays a central role in ACL2 and is tightly integrated with numerous aspects of the theorem prover. This has made the integration of CCG-based termination analysis a challenging problem requiring novel analyses involving controllers and has also required the introduction of the `make-event` functionality. We have an initial implementation of this work in ACL2s, the ACL2 Sedan, which is a new version of the ACL2 system. ACL2s is based on Eclipse and is designed to improve the usability and interface of ACL2 in order to ease the learning curve for new users while providing useful new features for experts [3].

References

1. B. Brock and W. A. Hunt, Jr. Formal analysis of the motorola CAP DSP. In *Industrial-Strength Formal Methods*. Springer-Verlag, 1999.
2. B. Brock and J. S. Moore. A mechanically checked proof of a comparator sort algorithm. In *Engineering Theories of Software Intensive Systems*. IOS Press, Amsterdam, 2005 (to appear).
3. P. Dillinger, P. Manolios, J. S. Moore, and D. Vroon. ACL2s: The ACL2 sedan. `http://www.cc.gatech.edu/~manolios/acl2s`.
4. D. Greve and M. Wilding. A separation kernel formal security policy. In M. Kaufmann and J. S. 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/`.
5. M. Kaufmann. ACL2 support for practical theorem proving. In G. Sutcliffe, R. Schulz, and R. Schmidt, editors, *Proceedings of ESCoR 2006*, 2006. to appear.
6. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.
7. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.
8. M. Kaufmann and J. S. Moore. The ACL2 home page. `http://www.cs.utexas.edu/users/moore/acl2/`.
9. C. S. Lee, N. D. Jones, and A. 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.
10. H. Liu and J. S. Moore. Executable JVM model for analytical reasoning: A study. In *Workshop on Interpreters, Virtual Machines and Emulators 2003 (IVME '03)*, San Diego, CA, June 2003. ACM SIGPLAN.
11. P. Manolios. Mu-calculus model-checking. In Kaufmann et al. [6], pages 93–111.
12. P. Manolios and D. Vroon. Algorithms for ordinal arithmetic. In F. Baader, editor, *19th International Conference on Automated Deduction – CADE-19*, volume 2741 of *LNAI*, pages 243–257. Springer–Verlag, July/August 2003.
13. P. Manolios and D. 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.
14. P. Manolios and D. Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, pages 1–37, 2006. `http://dx.doi.org/10.1007/s10817-005-9023-9`.
15. P. Manolios and D. Vroon. Termination analysis with calling context graphs. In T. Ball and R. Jones, editors, *Computer-aided Verification (CAV) 2006*, volume to appear of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
16. J. S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003. http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-03.
17. J. S. Moore. How to prove theorems formally. In *http://www.cs.utexas.edu/users/moore/publications/how-to-prove-thms.ps*. Department of Computer Sciences, University of Texas at Austin, 2004.
18. J. S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.
19. D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998. http://www.onr.com/user/russ/david/k7-div-sqrt.html.
20. D. M. Russinoff and A. Flatau. Rtl verification: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–232, Boston, MA., 2000. Kluwer Academic Press.
21. J. Sawada. Formal verification of divide and square root algorithms using series calculation. In *Proceedings of the ACL2 Workshop, 2002*. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002`, Grenoble, April 2002.