Termination Analysis with Calling Context Graphs*

Panagiotis Manolios and Daron Vroon

College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332, USA
http://www.cc.gatech.edu/home/{manolios,vroon}

Abstract. We introduce *calling context graphs* and various static and theorem proving based analyses that together provide a powerful method for proving termination of programs written in feature-rich, first order, functional programming languages. In contrast to previous work, our method is highly automated and handles any source of looping behavior in such languages, including recursive definitions, mutual recursion, the use of recursive data structures, etc. We have implemented our method for the ACL2 programming language and evaluated the result using the ACL2 regression suite, which consists of numerous libraries with a total of over 10,000 function definitions. Our method was able to automatically detect termination of over 98% of these functions.

1 Introduction

Proofs of termination are a critical component of program correctness arguments. In the case of transformational systems, termination proofs allow us to extend partial correctness results to total correctness. In the case of reactive systems, they are used to prove liveness properties, *i.e.*, to show that some desirable behavior is not postponed forever. Unfortunately, besides being the quintessential undecidable problem [21], termination analysis is further exacerbated by modern programming language features such as recursion, mutual recursion, non-linear loop conditions, and loops that depend on recursive data structures.

Because of this, previous work has tended to focus on finding decidable fragments of the problem, or has been designed for simple languages that lack the complexity of actual programming languages. Within such restricted settings, much progress has been made, *e.g.*, there is work on analyzing the termination of semi-algebraic programs, toy functional languages, and term rewriting systems (see Section 6).

We present a new termination analysis based on calling context graphs (CCGs) for a fully featured class of modern functional programming languages. If a purely functional program is nonterminating, there exists a sequence of values $v_1, v_2, \ldots, v_n$ such that for some function f_1, $f_1(v_1, v_2, \ldots, v_n)$ leads to an infinite sequence of function calls, $f_2(\ldots), f_3(\ldots), \ldots$, where the call to f_i results in the call to f_{i+1}, for all i. CCGs are a data structure which can conservatively approximate all such possible sequences. In addition, we show that CCGs are amenable to various analyses, involving both static analysis and theorem proving, that enable us to construct surprisingly precise approximations of the actual function call sequences. The termination proof then involves

* This research was funded in part by NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

assigning sets of calling context measures (CCMs) over well-founded domains to the calls and showing that for every possible infinite sequence there is a corresponding sequence of CCMs that is infinitely decreasing. We present an algorithm based on CCGs and CCMs that can automatically reason about any source of looping behavior in first order purely functional programming languages and which can automatically handle a much larger class of programs than previous approaches.

We have implemented our algorithm in the ACL2 theorem proving system, which consists of a feature-rich first-order functional programming language, a logic for that language, and an automatic theorem prover [11, 10, 9]. It has a large, worldwide user base, and has been used in a wide variety of industrial verification projects ranging from reasoning about modern processor designs to modeling programs written in imperative languages such as Java. ACL2 is part of the Boyer-Moore family of provers, for which its authors received the 2005 ACM Software System Award. Termination plays a key role in ACL2, as it is used to justify induction schemes and also every defined function must be shown to terminate. Therefore, users spend a significant amount of time reasoning about termination, and stand to greatly benefit from the work presented here.

In order to evaluate our work, we ran our implementation on the ACL2 regression suite, a collection of numerous libraries by a variety of authors covering topics such as commercial floating point verification (at AMD and IBM), JVM bytecode verification, term rewriting algorithm verification, the verification of a model checker, the verification of graph algorithms, etc. Our algorithm was able to automatically prove termination for over 98% of the more than 10,000 functions in the regression suite. This was accomplished with no user interaction.

The rest of the paper is organized as follows. In Section 2 we introduce the core of first-order functional languages. In Section 3, we introduce and develop the theory of calling context graphs. Our termination algorithm appears in Section 4, and experimental results are given in Section 5. Some readers may want to read Section 5 first. We end with related work and conclusions.

2 Semantics

While our method works for feature-rich, first-order functional programming languages including ACL2, such languages are quite complicated and we cannot fully describe them here. Instead, in Figure 1, we present the semantics of FL, a language that only contains the core features of first-order functional languages. The semantics are similar to what can be found in standard programming language texts. Some readers may want to skim this section initially, returning as needed later.

We are concerned with proving the termination of well-formed function definitions (members of the set $Defs$), which are of the form `define` $f(x_1, \ldots, x_n) = e$, where $f \in FName$ is a function name, $x_1, \ldots, x_n \in Var$ are variables, and $e \in Expr$ is an expression whose free variables are a subset of $\{x_1, \ldots, x_n\}$.

The universe of values over which FL is defined is Val and it includes symbols, strings, integers, rationals, and lists, but is otherwise unspecified. However, since this is a first order language, functions are not first class data objects, and are not included in

$$
\begin{array}{rcl}
d & \in & Defs \\
f & \in & FName \\
x & \in & Var \\
e & \in & Expr \\
v & \in & Val \\
u & \in & Val_\perp = Val \cup \{\perp\} \\
\epsilon & \in & Env = Var \to Val \\
\phi & \in & Funct = Val^* \to Val_\perp \\
\psi & \in & TFunct = Val^* \to Val \subseteq Funct \\
h & \in & IHist = FName \to Val^* \to Val_\perp \\
H & \in & Hist = FName \to Val^* \to Val \subseteq IHist
\end{array}
$$

$$
\begin{array}{rcl}
[\![e]\!]^h\,\epsilon & : & Expr \times IHist \times Env \to Val_\perp \\
\mathcal{O} & : & Op \to TFunct \\
str & : & Funct \to Val_\perp^* \to Val_\perp \\
\mathcal{D}\,[\![d]\!]\,H & : & Defs \times Hist \to Funct^+ \\
fix & : & (Funct^* \to Funct^*) \to Funct^*
\end{array}
$$

$$
str\ (\phi)\ \langle u_i \rangle_{i=1}^n = \begin{cases} \perp & \text{if } \langle \exists i \in [1..n] :: u_i = \perp \rangle \\ \phi\,(u_i)_{i=1}^n & \text{otherwise} \end{cases}
$$

$$
fix\ \xi = \lim_{j \to \infty} \xi^j\ \langle \lambda \boldsymbol{v_i} \,.\, \perp \rangle_{i=1}^m
$$

$$
[\![x]\!]^h\,\epsilon = \epsilon.x \quad , \quad [\![v]\!]^h\,\epsilon = v
$$

$$
[\![e_1\ \ op\ \ e_2]\!]^h\,\epsilon = str\ (\mathcal{O}\,[\![\texttt{op}]\!])\ \left\langle [\![e_i]\!]^h\,\epsilon \right\rangle_{i=1}^2
$$

$$
[\![f(e_1, \ldots, e_n)]\!]^h\,\epsilon = str\ (h.f)\ \left\langle [\![e_i]\!]^h\,\epsilon \right\rangle_{i=1}^n
$$

$$
\left[\!\!\left[\begin{array}{l} \texttt{let } x_1 \texttt{ = } e_1 \\ \quad \ldots \\ \quad x_n \texttt{ = } e_n \\ \texttt{ in } e \end{array} \right]\!\!\right]^h \epsilon = str\ \left(\lambda(v_i)_{i=1}^n \,.\, [\![e]\!]^h\,\epsilon[v_i/x_i]_{i=1}^n\right)\ \left\langle [\![e_i]\!]^h\,\epsilon \right\rangle_{i=1}^n
$$

$$
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]^h\,\epsilon = str\ \left(\lambda(v) \,.\, \begin{cases} [\![e_2]\!]^h\,\epsilon & \text{if } v \neq \texttt{nil}, \\ [\![e_3]\!]^h\,\epsilon & \text{otherwise.} \end{cases}\right)\ \left\langle [\![e_1]\!]^h\,\epsilon \right\rangle
$$

$$
\mathcal{D}\left[\!\!\left[\begin{array}{l} \texttt{define } f_1(x_1^1, \ldots, x_{n_1}^1) \texttt{ = } e_1 \\ \ldots \\ \texttt{define } f_k(x_1^k, \ldots, x_{n_k}^k) \texttt{ = } e_k \end{array} \right]\!\!\right] H = fix\left(\lambda(\phi_i)_{i=1}^k \,.\, \left\langle \lambda(v_j^i)_{j=1}^{n_j} \,.\, [\![e_i]\!]^{H[\phi_i/f_i]}\,[v_j^i/x_j^i]_{j=1}^{n_i} \right\rangle_{i=1}^k\right)
$$

Fig. 1. Language Semantics of FL.

Val. We use $\perp$ (which is not in Val) to denote nontermination, and $Val_\perp = Val \cup \{\perp\}$. An environment maps variables to values.

Function definitions in FL denote mathematical functions, which can either be members of the set $Funct$ or $TFunct$. $Funct$ consists of a set of partial functions, which means that for some inputs, functions in $Funct$ may return $\perp$, denoting nontermination. $TFunct$ is the subset of $Funct$ consisting of all the total (*i.e.*, terminating) functions. A *history* maps function names to total functions (of the appropriate arity) and an *intermediate history* maps function names to partial functions (of the appropriate arity).

The termination problem we consider is: given a history, H, and a set of mutually recursive definitions, d, show that the functions corresponding to the definitions in d are terminating. To do this, we need to refer not only to H, but also to the (possibly partial) functions corresponding to the definitions in d. This is accomplished by using an intermediate history, h, which is just H extended so that it includes the function names appearing in d and their corresponding functions, as given by the semantics of FL (which are given in Figure 1 and described in more detail in the next paragraph). We then attempt to prove that the functions defined in d terminate, which implies that the intermediate history, h, is actually a history. If so, we have a new history. Otherwise, we reject d, revert to H, and report the problem to the user. This allows the user to incrementally define programs, as is common in programming environments for functional languages, such as Lisp.

We use five functions to define the semantics of FL. The function $[\![e]\!]^h\,\epsilon$ defines how to evaluate an expression, e, given an intermediate history, h, and an environment, ϵ. The function $\mathcal{O}$ maps FL's unspecified set of built-in operators (Op) to their corresponding functions. The set of built-in operators includes the usual Boolean and arithmetic operators, such as `and`, `or`, `not`, `iff`, `implies`, `+`, `-`, `/`, `*`, etc. The function *str* corresponds to strict application. As input, it takes a function and a vector of values (possibly including $\perp$, which indicates nontermination). It returns $\perp$ if any of the input values is $\perp$; otherwise, it returns the result of applying the function to the values (which could also be $\perp$). The definitions of the semantics functions for variables, values, built-in operators, function application, `let`s, and `if`s are now straightforward.

Function definitions are handled with $\mathcal{D}\,[\![d]\!]\,H$, which defines what mathematical functions (elements of $Functs$) correspond to a set of function definitions, d, given history H. Its definition depends on the *fix* function, which is used to define the semantics of recursive function definitions using the standard fixpoint approach. The *fix* function takes as input ξ, a function from a vector of functions to a vector of functions, and returns the vector of functions obtained by taking the limit as j approaches infinity of applying ξ to the vector of functions returning $\perp$. The definition of $\mathcal{D}\,[\![d]\!]\,H$ uses *fix* to "unroll" the bodies of the definitions an unbounded number of times, which results in a vector of partial functions that corresponds to the semantics of the definitions.

Throughout the rest of this paper, unless otherwise specified, we assume a fixed history, H and a set of syntactically correct, mutually-recursive function definitions, d, such that none of the function names in d are the same as those in the domain of H. The intermediate history h is obtained by extending H with the semantics of the function definitions in d. To simplify the notation, we assume the uniqueness of subexpressions. That is, if expression e has two identical subexpressions, then we have some way of determining which is which. This can be accomplished by pairing each subexpression with its unique position within the base expression. We use "$e_1 \trianglelefteq e_2$" to denote that e_1 is a sub-expression of e_2.

We now give several definitions related to the semantics of FL that we will use throughout the paper. We begin by defining the set of *governors* under which a subexpression e' of e is reached, ignoring nontermination (for now). Our definition is synonymous with that in [12]. If e is an FL `let` statement and $e' \trianglelefteq e$, then we use $\sigma^e_{e'}$ to denote the substitution (a mapping from variables to expressions) corresponding to the `let` bindings of e that are visible in e'. For example, if $e =$ `let x =` e_1 `in` e_2, then $\sigma^e_{e_2} = \{\langle \mathtt{x}, e_1\rangle\}$ and $\sigma^e_{e_1} = \{\}$. We use $e\sigma$ to denote the expression obtained by applying substitution σ to e.

Definition 1. *Given expressions e', e such that $e' \trianglelefteq e$, the* set of governors of e' in e *is the set* $\{e_1\sigma^e_{e_1} \mid$ `if` e_1 `then` e_2 `else` $e_3 \trianglelefteq e \wedge e' \trianglelefteq e_2\} \cup \{$`not(`$e_1\sigma^e_{e_1}$`)` $\mid$ `if` e_1 `then` e_2 `else` $e_3 \trianglelefteq e \wedge e' \trianglelefteq e_3\}$.

The idea of the governors of e' in e is that the execution of e reaches e' exactly when the governors are true. We therefore define the more general notion of when expressions "hold":

Definition 2. *We say a set of expressions, E,* holds *for environment ϵ, denoted $\mathcal{H}^h\,[\![E]\!]\,\epsilon$, if* $\bigwedge_{e\in E}([\![e]\!]^h\,\epsilon \notin \{\mathtt{nil}, \perp\})$.

```
define f(x) =
   if not(intp(x)) or x = 0
      then 0
      else if x < 0 then f(x+1)
                    else f(x-1)
define dec(n) =
   if not(intp(x)) or x ≤ 0
      then 255
      else n - 1

define foo(i, j) =
   if i = 1 then
      if j = 1 then 0
      else foo(dec(j), dec(j))
   else foo(dec(i), j)
```

1. ⟨f, {intp(x), x ≠ 0, x < 0}, f(x+1)⟩
2. ⟨f, {intp(x), x ≠ 0, x ≥ 0}, f(x-1)⟩

1. ⟨foo, {i = 1, j ≠ 1}, foo(dec(j),dec(j))⟩
2. ⟨foo, {i ≠ 1}, foo(dec(i),j)⟩

Fig. 2. Definitions, contexts, and minimal CCGs for `f` and `foo`.

3 Calling Context Graphs

In this section, we introduce calling context graphs (CCGs) and related notions. We also show how CCGs can be used reason about program termination.

Definition 3. *A* calling context *is a triple, $\langle f, G, e\rangle$, where f is the name of a function defined in d, G is a set of expressions whose free variables are all parameters of f, and e is a call of a function in d whose free variables are all parameters of f. This is a* precise calling context *if e is a subexpression in the body of f and G is the set of governors of e in the body of f.*

We sometimes refer to a calling context simply as a context. The definitions and contexts for two examples are given in Figure 2. We now introduce the notion of a well-formed sequence of contexts, a notion that is strongly related to termination in FL.

Definition 4. *Let $c = \left(\langle f_i, G_i, f_{i+1}(e_{i,1}, \ldots, e_{i,n_{i+1}})\rangle\right)_i$ be a sequence of calling contexts, where n_i is the arity of f_i and $(x_{i,k})_{k=1}^{n_i}$ are the formals of f_i. For a given vector of values $\boldsymbol{v}$, we define a sequence of environments where $\epsilon_1^{\boldsymbol{v}}$ maps $x_{1,k}$ to v_k and $\epsilon_{i+1}^{\boldsymbol{v}}$ maps $x_{i+1,k}$ to $[\![e_{i,k}]\!]^h\ \epsilon_i^{\boldsymbol{v}}$. We say c is* well-formed *if there exists a* witness *for c: a vector of values, $\boldsymbol{v}$, such that for every $i > 0$, $\mathcal{H}^h\ [\![G_i]\!]\ \epsilon_i^{\boldsymbol{v}}$ and $\langle \forall j \leq n_i :: [\![e_{i,j}]\!]^h\ \epsilon_i^{\boldsymbol{v}} \neq \bot\rangle$.*

We use the notation $\epsilon_i^{\boldsymbol{v}}$ introduced in the above definition throughout the paper. Termination in FL can be expressed in terms of well-formed sequences, as we see in the next theorem. (Due to space considerations all proofs have been elided.)

Theorem 1. *The functions of d terminate on all inputs iff every well-formed sequence of precise contexts is finite.*

We now define the notion of a *calling context graph* and show that it is a conservative approximation of the well-formed sequence of contexts.

Definition 5. *A* calling context graph (CCG)*, is a directed graph, $\mathcal{G} = (C, E)$, where C is a set of calling contexts, and for any pair of contexts $c_1, c_2 \in C$, if the sequence $\langle c_1, c_2\rangle$ is well-formed, then $\langle c_1, c_2\rangle \in E$. If C is the set of precise contexts of d, then $\mathcal{G}$ is called a* precise CCG of d.

```
define size(x) = if pairp(x) then size(first(x)) + size(rest(x)) + 1
                              else if intp(x) then abs(x) else 0
```

Fig. 3. Definition of `size`

The minimal precise CCG for function `f` in Figure 2 is shown in the same figure. Note that there is no edge between the two contexts. This is because if `x` is a positive integer, then decrementing `x` by 1 will not lead to a negative integer. Likewise, adding 1 to `x` if it is a negative integer cannot produce a positive integer. Notice that this mirrors the looping behaviors of the function. Figure 2 also contains the minimal precise CCG for function `foo`. Notice that if the first context of `foo` is reached, `foo` calls itself, passing in `(dec j)` for both arguments. Since `(dec j)` cannot simultaneously be both equal to 1 and not equal to 1, it is impossible to immediately reach context 1 again. However both contexts can reach context 2, and context 2 can reach context 1.

Lemma 1. *Given a CCG, $\mathcal{G} = (C, E)$, every well-formed sequence of calling contexts of C is a path in $\mathcal{G}$.*

Note that the converse of the above lemma does not hold. This is because the definition of a CCG only requires local reachability whereas a well-formed sequence of contexts requires that the entire sequence correspond to a single computation. As a result, a CCG is an abstraction of the actual system. We use CCGs to perform a local analysis which if successful can determine that the definitions terminate. To do this, we start by assigning calling context measures to contexts in the CCG.

Definition 6. *Given a calling context, $c = \langle f, G, e\rangle$, and a set $S \subseteq Val$, a* calling context measure (CCM) for c over S, *s, is an expression whose free variables are parameters of f and for any environment, ϵ, $\mathcal{H}^h [\![G]\!] \epsilon \Rightarrow [\![s]\!]^h \epsilon \in S$.*

CCMs simply map the parameters of a function into some set. For our purposes, this set will have a well-founded ordering on it. Now we create a mechanism for comparing the CCM of two adjacent contexts in a CCG.

Definition 7. *Let $\mathcal{G} = (C, E)$ be a CCG with $e = \langle c_1, c_2\rangle \in E$. Let $\langle S, \prec\rangle$ be a well-founded structure where S_{c_1} and S_{c_2} are sets of CCMs over S for c_1 and c_2, respectively. Then, the* CCM function for *e over $\prec, S_{c_1}$, and S_{c_2} is the function $\phi : S_{c_1} \times S_{c_2} \to \{>, \geq, \times\}$ such that: (1) $\phi(s_1, s_2) = >$ only if for all witnesses $\boldsymbol{v}$ for $\langle c_1, c_2\rangle$, we have $[\![s_1]\!]^h \epsilon_1^{\boldsymbol{v}} \succ [\![s_2]\!]^h \epsilon_2^{\boldsymbol{v}}$; (2) $\phi(s_1, s_2) = \geq$ only if for all witnesses $\boldsymbol{v}$ for $\langle c_1, c_2\rangle$, we have $[\![s_1]\!]^h \epsilon_1^{\boldsymbol{v}} \succeq [\![s_2]\!]^h \epsilon_2^{\boldsymbol{v}}$; (3) $\phi(s_1, s_2) = \times$, otherwise.*

We represent CCM functions for $\langle c_1, c_2\rangle$ graphically with a box containing the CCMs for c_1, c_2 on the left and right, respectively. An edge is drawn from s_1, a left CCM, to s_2, a right CCM, with the label $\phi(s_1, s_2)$ iff it is $>$ or $\geq$. If $\phi(s_1, s_2)$ is $\times$, no edge is drawn.

We now consider some examples. For the function `f` in Figure 2, we use the `size` function in Figure 3 applied to `f`'s parameter, `x`, as the only CCM for both contexts. The range of `size` is the set of natural numbers, and the function is designed to mirror

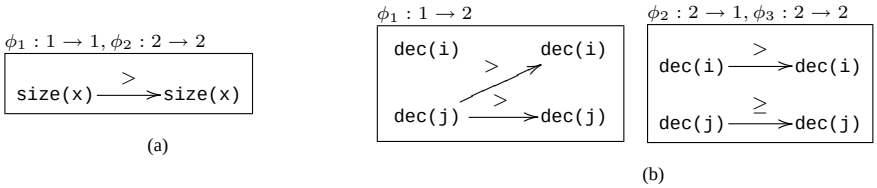


Fig. 4. (a) CCM function for `f`. (b) CCM functions for `foo`

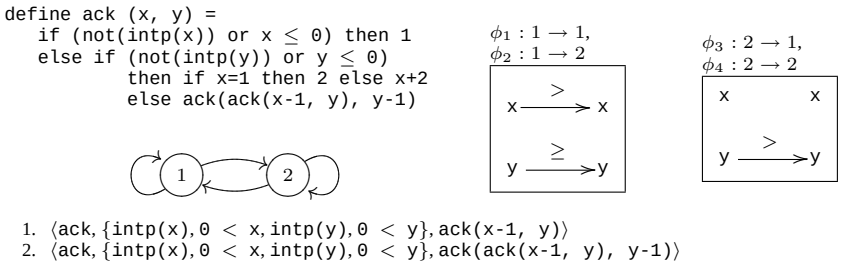


Fig. 5. Ackermann's function.

common induction schemes, *e.g.*, induction on the size of a list. Notice that for each context in our example, the CCM decreases for all values of `x` that satisfy the governors of the context. The resulting CCM functions are shown in Figure 4a. For the function `foo` in Figure 2, we use different CCMs. Namely, we apply `dec` to the arguments; note that `dec` always returns a natural number, which is a well-founded domain under the $<$ relation. The result is shown in Figure 4b. The question of how to choose CCMs is addressed in Section 4.

We use CCM functions to show that certain infinite paths are not feasible and also to show that CCGs correspond to terminating functions.

Definition 8. *We say that a CCG, $\mathcal{G} = (C, E)$ is* well-founded *if there exists a well-founded structure, $\langle S, \prec \rangle$ and a mapping, m, from C into sets of CCMs over S such that $\mathcal{M}_{C,\prec,m}(c)$ for all infinite paths, $c = c_1, c_2, \ldots$, through $\mathcal{G}$. $\mathcal{M}_{C,\prec,m}$ is a* CCM predicate *and holds for an infinite sequence of contexts, c, iff there exists $i_0 \geq 1$ and a sequence $s_{i_0}, s_{i_0+1}, \ldots$ such that for all $i \geq i_0$, $s_i \in m(c_i)$ and $\phi_i(s_i, s_{i+1}) \in \{>, \geq\}$, and for infinitely many such i, $\phi_i(s_i, s_{i+1}) = >$, where ϕ_i denotes the CCM function for $\langle c_i, c_{i+1} \rangle$ with CCMs $m(c_i)$ and $m(c_{i+1})$.*

It is important to note here that we do not need to fix a CCM for each context in order to satisfy the CCM predicate. Rather, we can select from any of the CCMs for a given context each time it appears in a sequence. For example, consider Ackermann's function, given in Figure 5. Here, if a sequence contains context 2 infinitely often, then `y` decreases infinitely, and if it does not, then there is an infinite suffix of the sequence that is just context 1, which means that `x` decreases infinitely often. It is possible to create one measure that decreases in both cases, but this measure requires a well-founded structure more powerful and complex than the natural numbers.

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

1. $\langle$g, {}, f(x+1)$\rangle$
2. $\langle$h, {}, f(x-1)$\rangle$
3. $\langle$f, {intp(x), x $\neq$ 0, x $<$ 0}, g(x)$\rangle$
4. $\langle$f, {intp(x), x $\neq$ 0, 0 $\leq$ x}, h(x)$\rangle$

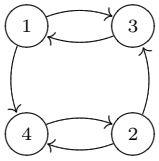
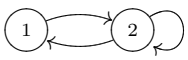


Fig. 6. Altered version of function defined in Figure 2

```
define f(x) =
   if not(intp(x))
      or x ≤ 1
      then 0
   else if x mod 2 = 1
      then f(x+1)
   else 1 + f(x/2)
```

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



$\phi_1 : 1 \rightarrow 2$

size(x)	size(x)

$\phi_2 : 2 \rightarrow 1, \phi_3 : 2 \rightarrow 2$

size(x) $\xrightarrow{>}$ size(x)

Fig. 7. Example of the abstraction inherent in the infinite CCM relation.

It turns out that we only need to consider maximal SCCs (strongly connected components) to establish termination.

Theorem 2. *Let $\mathcal{G} = (C, E)$ be a CCG, s.t. C is the set of precise contexts of d. If every maximal SCC of $\mathcal{G}$ is well-founded, then all functions of d terminate on all inputs.*

Notice that the converse of Theorem 2 does not hold because the paths of a CCG are a superset of the well-formed sequences of contexts. For example, notice that when we split function `f` from Figure 2 into several functions, as in Figure 6, all the contexts now appear in the same SCC. Why? Consider the function, `g`. Note that `g(2)` results in the call `f(3)`, which leads to context 4. A similar situation arises for `h`. Thus $1, 4, 2, 3, 1, 4, 2, 3, \ldots$ is a valid path through any CCG, even though it is not a well-formed sequence of contexts. Each time through the loop $1, 4, 2, 3$, the value of `x` stays the same, hence, the termination analysis presented so far fails.

Another source of imprecision is due to the local analysis used in determining if a CCG is well-founded. If a value decreases over several steps, but increases for one of those steps, the termination analysis presented so far will fail. Consider the example in Figure 7. When `x` is odd, 1 is added to `x` and when it is even, `x` is divided by 2. This continues until `x` is 1 (or not a positive integer). This results in an overall decrease of the value of `x` despite the initial increase.

In order to gain more accuracy and overcome many of the problems caused by the local nature of our analysis, we introduce the idea of context merging. This essentially enables us to consider multiple steps instead of single steps.

$1;2$: $\langle$f, {intp(x), 0 $\leq$ x, x$\neq$1, x mod 2 = 1, intp(x+1), 0 $\leq$ x+1, (x+1) mod 2 $\neq$ 1}, f((x+1)/2)$\rangle$
$2;1$: $\langle$f, {intp(x), 0 $\leq$ x, x$\neq$1, x mod 2 $\neq$ 1, intp(x/2), 0 $\leq$ x/2, (x/2) mod 2 = 1}, f(x/2+1)$\rangle$
$2;2$: $\langle$f, {intp(x), 0 $\leq$ x, x$\neq$1, x mod 2 $\neq$ 1, intp(x+1), 0 $\leq$ x+1, (x+1) mod 2 $\neq$ 1}, f((x/2)/2)$\rangle$

Fig. 8. Merging and compaction results for Figure 7.

Definition 9. *The* call substitution *of* $e = f(e_1, e_2, \ldots, e_n)$*, denoted* σ_e*, maps* x_i *to* e_i *for all* $1 \leq i \leq n$*, where* $x_1, x_2, \ldots, x_n$ *are the parameters of* f*.*

Definition 10. *Let* $\langle c_1, c_2 \rangle$ *be a well-formed sequence of calling contexts, where* $c_1 = \langle f_1, G_1, e_1 \rangle$ *and* $c_2 = \langle f_2, G_2, e_2 \rangle$*. The* merging *of* c_1 *and* c_2*, denoted* $c_1; c_2$*, is the calling context* $\langle f_1, G_1 \cup \{p\sigma_{e_1} \mid p \in G_2\}, e_2\sigma_{e_1} \rangle$*.*

As an example, note that if in Figure 6 we merge context 3 with context 1 and context 4 with context 2, we get contexts 1 and 2 of Figure 2, respectively. This makes sense as the example in Figure 6 was obtained by splitting f into several functions and merging essentially recombines the contexts. For a more interesting example, in Figure 7 consider merging context 1 with context 2, context 2 with context 1, and context 2 with itself; the result appears in Figure 8.

We now use merging to define the notion of absorption and show that given a CCG, we can define an infinite sequence of CCGs such that if we can prove that at least one CCG in the sequence terminates, then so does the original CCG. This can greatly extend the applicability of our analysis.

Definition 11. *Given a CCG,* $\mathcal{G} = (C, E)$*, the result of* absorbing $c \in C$ *is a CCG* $\mathcal{G}' = (C', E')$ *where* $C' = C \backslash \{c\} \cup \{c; c' \mid \langle c, c' \rangle \in E\}$*.*

Theorem 3. *Let* $\mathcal{G}_0, \mathcal{G}_1, \ldots$ *be a sequence of CCGs such that* $\mathcal{G}_0$ *is a precise CCG of* d*, and* $\mathcal{G}_{i+1}$ *is obtained from* $\mathcal{G}_i$ *by absorbing a context. If for some* i*, every maximal SCC of* $\mathcal{G}_i$ *is well-founded, then every function in* d *terminates on all inputs.*

4 Algorithm

The definitions given in Section 3 suggest the following algorithm for the termination analysis of a set of function definitions, d, using static analysis and theorem proving.

1. Using static analysis, construct the precise calling contexts of d.
2. Using theorem proving, build a precise CCG.
3. Absorb contexts that have only one successor.
4. Divide the CCG into SCCs
5. Choose a well-founded structure for each SCC, and a set of CCMs for each context.
6. Use theorem proving to construct safe approximations of the CCM functions.
7. Perform analysis to decide the CCM predicate for all paths through each SCC.

Step 1 is straightforward, and one can construct the algorithm from Definition 1. Step 2 involves building a CCG. We wish to construct as minimal a CCG as we can, in order to avoid spurious paths through the CCG, which complicate the rest of the

algorithm and can lead to less accurate analysis. Therefore, for every pair of contexts, $c_1 = \langle f, G_1, e_1 \rangle$ and $c_2 = \langle g, G_2, e_2 \rangle$, such that e_1 is a call to g, we query the theorem prover to prove that $\langle c_1, c_2 \rangle$ is not well-formed, and therefore no edge needs to be added from c_1 to c_2. The corresponding theorem prover query is $\langle \forall \boldsymbol{v} \in \mathit{Val}^* :: (\bigwedge_{p \in G_1} [\![p]\!]^h \epsilon_1^{\boldsymbol{v}} \neq \texttt{nil}) \Rightarrow \neg(\bigwedge_{q \in G_2} [\![q\sigma_{e_1}]\!]^h \epsilon_1^{\boldsymbol{v}} \neq \texttt{nil}) \rangle$. If the proof is successful, we omit the edge $\langle c_1, c_2 \rangle$.

For this algorithm, we choose a simple absorption strategy. While absorbing a context in a CCG may result in a CCG that is more amenable to analysis, it may also increase the size of the CCG (by up to a factor of 2). However, if a context has only one successor in the CCG, absorbing it creates a CCG at most the size of the original. We therefore perform several passes through the graph, absorbing all such contexts with each pass. This simple absorption strategy is quite effective, *e.g.*, it allows us to automatically prove the termination of the functions in Figure 6. We plan to explore other strategies, such as looping from step 3 through step 7 and using the result of the previous failed termination analysis to guide absorption.

Once absorption is completed, we choose well-founded structures and CCMs. Currently, we always default to natural numbers for our well-founded structure. We use heuristics to automatically choose CCMs. Currently, these include the following.

- We use a version of the `size` function from Figure 3, called `acl2-count`, that is extended to deal with more types, adding the `size` of each parameter of a function to the CCMs of each context from that function.
- When $e_1 < e_2$ or $e_1 \leq e_2$ is a governor of a context, we add e_1 `-` e_2 as a CCM.
- When `intp(`e`)` and `0` $< e$ are governors of the context, we add e as a CCM.

Finally, we propagate measures other than the `size` of the parameters through the rest of the contexts. That is, if we add a CCM s, to a context, then to each of its predecessors in the CCG we add the CCM $s\sigma_e$, where e is the call of the predecessor. We repeat this until the CCM is propagated to each of the contexts in the CCG.

In step 6, we approximate the CCM functions using the theorem prover. Given two adjacent contexts, $c_1 = \langle f, G_1, e_1 \rangle$ and $c_2 = \langle g, G_2, e_2 \rangle$, in an SCC, then for every CCM, s_1, for c_1 and every CCM, s_2, for c_2, we perform the following analysis. We first attempt to prove that for all ϵ, $[(\bigwedge_{p \in G_1} [\![p]\!]^h \epsilon \neq \texttt{nil}) \wedge (\bigwedge_{q \in G_2} [\![q\sigma_{e_1}]\!]^h \epsilon \neq \texttt{nil})] \Rightarrow [\![s_1]\!]^h \epsilon \prec [\![s_2\sigma_{e_1}]\!]^h \epsilon$. If this succeeds, we set $\phi(s_1, s_2)$ to be $>$. Otherwise, we attempt to prove that for all ϵ, $[(\bigwedge_{p \in G_1} [\![p]\!]^h \epsilon \neq \texttt{nil}) \wedge (\bigwedge_{q \in G_2} [\![q\sigma_{e_1}]\!]^h \epsilon \neq \texttt{nil})] \Rightarrow [\![s_1]\!]^h \epsilon \preceq [\![s_2\sigma_{e_1}]\!]^h \epsilon$. If this succeeds, we set $\phi(s_1, s_2)$ to be $\geq$. If neither proof succeeds, we set $\phi(s_1, s_2)$ to be $\times$.

The final step of the algorithm is to determine the value of the CCM predicate. In other words, we wish to determine that for every path through the graph, we can choose one of the CCMs from each context in the path such that they never increase in value, and infinitely decrease in value. A basic algorithm for doing this appears in [13].

5 Experimental Results

In this section, we experimentally evaluate the theory of calling context graphs we have introduced in this paper. As we saw in the previous section, our analysis is param-

eterized by the CCMs used and by the merging and absorption strategies employed. Our goal is to evaluate a simple, baseline version of the algorithms we have presented. Therefore, we use a simple absorption strategy and a small, simple collection of CCMs.

We have implemented our termination algorithm and used it on the ACL2 system, an industrial-strength theorem proving system that consists of a feature-rich functional programming language, a first-order logic for reasoning about this language, and a theorem prover for the automation of this reasoning. The ACL2 language can roughly be thought of as an applicative (pure, functional) subset of Common Lisp. The reality is more complicated because ACL2 has many advanced features such as single-threaded objects, which have been shown to enable execution at close to C speeds. ACL2 is actively used by a worldwide user-base to perform tasks as diverse as microprocessor modeling and simulation, the analysis of graph algorithms, algebraic reasoning, the analysis of imperative programs written in languages such as Java, etc. For more information on ACL2 see [11, 10, 9].

ACL2 is a good choice for us because termination arguments play a key role in its logic. First, every program admitted by the definitional principle must be shown to terminate before it is accepted by ACL2. This guarantees that definitions do not render ACL2 inconsistent. Second, inductive reasoning, ACL2's forte, is justified using termination arguments (to show that the induction is well-founded). Currently, termination in ACL2 is proven by providing an ordinal-valued measure and showing that it decreases on every recursive call. The ordinals are a transfinite extension of the natural numbers that form the basis of set theory; in fact, any well-founded argument can be phrased in terms of the ordinals. In recent work, we improved ACL2's handling of the ordinals, defined algorithms for ordinal arithmetic, and created a library of theorems for reasoning about the ordinals and ordinal arithmetic. The result was a significant improvement in ACL2's ability to reason about termination, once an ordinal measure is provided [14–17]. ACL2 tries to automate termination analysis by guessing a measure of the form `acl2-count(x)`, where `x` is some parameter of the function. Unfortunately, it is often the case that this simple heuristic fails and the user must discover and provide an appropriate ordinal measure.

Another advantage of using ACL2 is that it has a regression suite consisting of 137 MB of definitions and theorems. There are over 10,400 function definitions arising in the work of various researchers around the world and ranging from bit-vector libraries used by AMD (to prove the correctness of their floating point units) to set theory libraries to graph algorithms to model checkers, *etc*. The termination of all of these functions has already been proven with ACL2. In the cases where ACL2 does not automatically prove termination, human guidance is required. We distinguish two types of guidance.

Implicit guidance is given when users prove auxiliary lemmas which help ACL2 to complete the termination proof. While it is difficult to identify the theorems used solely to prove termination, it is clear that many termination proofs require auxiliary lemmas and substantial human effort. For example, in a recent posting to the ACL2 mailing list, an experienced ACL2 user asked whether a particular proof could be simplified. After some discussion, he simplified his proof and posted a proof challenge to see if anyone could simplify it further. The point was to establish the termination of function

Which Functions	Total	# Correct	% Correct
All	10,442	10,308	98.7%
With Explicit Guidance	421	287	68.2%

Table 1. Results of experiments on the regression suite

`fringep`. The simplified proof included a library for reasoning about arithmetic, seven lemmas, one theory command, and five function definitions. Two of the functions were needed to define `fringep`, but the other three functions were needed for the proof. The proof script also contained several hints, the use of the proof checker, and several theorems that were classified as `:linear` rules (which are handled in a special way by ACL2). The proof was simplified by another experienced ACL2 user, but it still required the library, five function definitions, and five theorems. Using our system, we proved termination directly in seconds, without using the library, without the extra definitions, without any lemmas, and most importantly, without thinking.

Explicit guidance is given when users provide the measure explicitly or when they provide hints on how to prove termination. Such guidance is easy to detect and of the 10,442 functions in the regression suite, 404 required the user to provide explicit measures and 17 more requited hints. For example, here is a part of a function from the regression suite that specifies an explicit measure: an ordinal constructed using ordinal multiplication (`o*`), ordinal addition (`o+`), the first infinite ordinal (`(omega)`), and several auxiliary functions (*e.g.*, `tuple-set-max-first`).

```
(defun tuple-set->ordinal-partial-sum (k S i)
  (declare (xargs
            :measure (o+ (o* (omega) (nfix k))
                         (nfix (- (tuple-set-max-first S) i)))))
  ...)
```

The actual function definition is too long to list here, but discovering infinite measures requires some skill. Our system automatically proves that the above function terminates.

To quantitatively evaluate our work, we removed all sources of explicit hints and ran our termination method on the full regression suite. Since identifying the implicit guidance is difficult, we did not attempt to remove such lemmas, but we note that since our termination analysis is very different from ACL2's, such lemmas are not very likely to provide much help for us. The results of our experiments are presented in Table 1. Out of all 10,442 functions analyzed by our system, 10,308 (over 98%) were automatically proven to terminate. Included in these are 287 of the 421 functions which required the user to provide explicit measures or hints for ACL2 to prove termination. In other words, of the most difficult 4% of functions to analyze, our tool successfully and fully automatically analyzed almost 70% of them.

6 Related Work

Termination is one of the oldest problems in computing science and it has received a significant amount of attention. Here we will briefly review recent work on automating termination analysis.

One of the most often cited techniques for the proving termination of programs is called the *size change principle* [13]. This method involves using a well-order on function parameters, analyzing recursive calls to label any clearly decreasing or non-increasing parameters. Then, all infinite paths are analyzed to ensure that some parameter never increases and infinitely decreases over each path. We use this path analysis in step 7 of our algorithm. The size change principle has several limitations, *e.g.*, it does not show how to take governors into account and it does not provide any method for determining the sizes of the outputs of user-defined functions. Both of these considerations are almost always important for establishing termination in realistic programming languages.

Much work has gone into developing termination analyses for term rewriting systems and logic programs, *e.g.*, [2, 8, 4]. However, these methods do not scale to the complexity of functional programming languages. For example, the AProVE tool [8], cannot prove the termination of a function that takes two integer arguments, x and y, and increments x until it is greater than y, which is the behavior of a simple `for` loop.

There has been a significant amount of work on proving the termination of programs written in high-level imperative languages such as C. This work tends to focus on semi-algebraic functions, whose termination behavior is governed by integer arithmetic. Most of it has been even more narrowly defined than that, dealing only with systems whose behavior is linear [19, 20]. Recently, this work has been extended to programs with polynomial behavior [3, 6]. While successful in dealing with semi-algebraic programs, these methods are not applicable outside of this domain, *e.g.*, they cannot reason about data structures, which often play a crucial role in termination proofs, or non-polynomial arithmetic. A recent paper presents an abstraction-refinement algorithm for termination analysis. The algorithm deals with loops, but cannot currently handle recursion and was not implemented [5].

7 Conclusion

We introduced the notion of calling context graphs and various related static and theorem proving based analyses that together led to a powerful new method for proving termination of programs written in feature-rich, first-order, purely functional languages. We implemented our algorithm and were able to automatically detect the termination of over 98% of the more than 10,000 function definitions in the ACL2 regression suite. For future work, we are developing an abstraction-refinement framework that uses more advanced absorption and merging strategies to refine CCGs. We are also looking at extending our analysis to deal with imperative languages such as C by taking advantage of various static analyses (such as alias analysis, data-flow, and control-flow) and taking advantage of the fact that Static Single Assignment (SSA), a popular intermediate language used for the analysis and optimization of imperative programs, is essentially a pure functional language [1]. More generally, we are interested in exploring algorithms that combine static analysis methods with theorem proving [18].

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. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In Cousot [7], pages 113–129.
4. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
5. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *Static Analysis: 12th International Symposium, SAS 2005*, volume 3672 of *LNCS*, pages 87–102, September 2005.
6. Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In Cousot [7], pages 1–24.
7. Radhia Cousot, editor. *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*. Springer, 2005.
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. Matt Kaufmann and J. Strother Moore. Structured theory development for a mechanized logic. *J. Autom. Reason.*, 26(2):161–203, 2001.
13. 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.
14. Panagiotis Manolios and Daron Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal Of Automated Reasoning*. To Appear.
15. 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.
16. 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/`.
17. 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.
18. Panagiotis Manolios and Daron Vroon. Integrating static analysis and general-purpose theorem proving for termination analysis. In *ICSE'06, The 28th International Conference on Softwar Engineering, Emerging Results*. ACM, May 2006.
19. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.
20. A. Tiwari. Termination of linear programs. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV*, volume 3114 of *LNCS*, pages 70–82. Springer, July 2004.
21. Alan Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of *Series 2*, pages 230–265, 1936.