Termination Analysis with Calling Context Graphs

Pete Manolios
Northeastern

Joint work with
Daron Vroon

Austin ACL2 Workshop Nov. 2007

Termination

"The checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number. In this problem the ordinal might be $(n - r)\omega^2 + (r - s)\omega + k$."

-Alan M. Turing (1949)

Termination Analysis

- Quintessential undecidable software verification problem
- Transformational systems: partial vs total correctness
- Reactive systems: liveness
- Theorem proving: consistency and induction schemes
- Domain: fully-featured, first-order, pure functional PLs
- Novel combination of static analysis and theorem proving
- Introduce Calling Context Graphs and Measures [MV'06]
  - New, fully automatic termination analysis
  - General: Can be used to reason about any looping behavior
- Our analysis is implemented in ACL2s

Outline

Overview

Idea

Calling Context Graphs & Measures

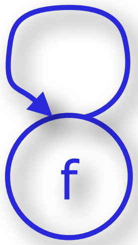
Experimental Results

Conclusions and Future Work

The Idea

- Non-termination in our domain iff
  ∃ some input that leads to an infinite sequence of function calls
- Goal: Conservative, precise, analyzable abstraction
- First attempt: use call graphs
- Example:

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



Not enough

Outline

Governors

- The governors of $e' \subseteq e$ are the branching conditions that need to be true for execution of e to lead to the execution of e′
- Example

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

Governors

- The governors of e′ ⊆ e are the branching conditions that need to be true for execution of e to lead to the execution of e′
- Example

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

- Governors for f(x+1): {intp(x), x > 1, x mod 2 = 1}

Governors

- The governors of $e' \subseteq e$ are the branching conditions that need to be true for execution of e to lead to the execution of e′
- Example

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

- Governors for f(x+1): {intp(x), $x > 1$, $x \bmod 2 = 1$}
- Governors for f(x/2) : {intp(x), $x > 1$, $x \bmod 2 \neq 1$}

Precise Calling Contexts

- A precise calling context for a call e is a triple containing:
  - The name of the function containing e
  - The governors for e in the function body
  - The call, e
- Example:

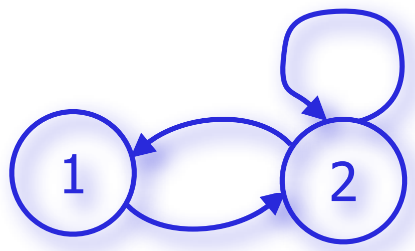
```
define f(x) =
  if  (!intp(x) or x ≤ 1)
    then 0
    else if  (x mod 2 = 1)
      then f(x+1)
      else 1 + f(x/2)
```

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

Calling Context Graphs

- Example:

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

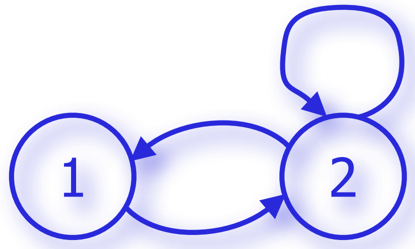

$$\begin{tikzcd} 1 \arrow[r, bend right] & 2 \arrow[l, bend right] \arrow[loop above] \end{tikzcd}$$

1. ⟨f, {intp(x), x > 1, x mod 2 = 1}, f(x+1)⟩
2. ⟨f, {intp(x), x > 1, x mod 2 ≠ 1}, f(x/2)⟩

- Vertices are calling contexts
- An edge from c1 to c2 if it is possible for execution to reach c1 and c2 in consecutive recursive calls
- Set of all paths is an overapproximation of recursive behavior

Building Calling Context Graphs

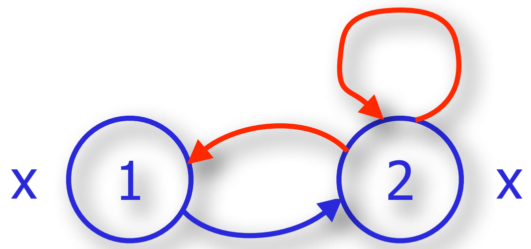
- The edge condition:
  ∃ values that satisfy the governors of both contexts
- Governors are arbitrary predicates
- Building a minimal CCG is undecidable
- Theorem prover queries used to eliminate unnecessary edges
- Edge included if theorem prover cannot disprove the edge condition
- We need more



Calling Context Measures

- Map function formals into some well-founded structure
- Each calling context is given a set of CCMs
- Example:

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



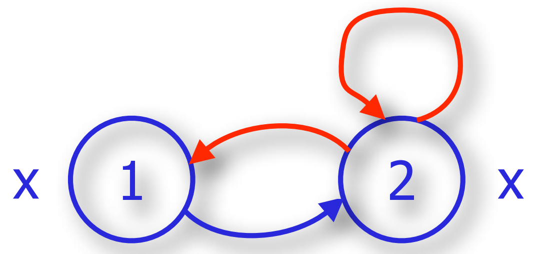
Decreasing edge >:

Non-increasing edge ≥:

Neither X:

The Termination Condition

- For every infinite path through the CCG,
- There should exist a corresponding sequence of CCMs,
- Such that for some tail of the sequence, each adjacent pair of CCMs is never increasing and infinitely decreasing
- Solved by Size Change back-end algorithm



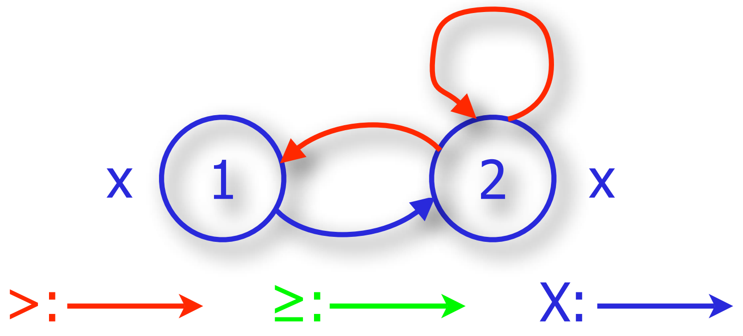
Decreasing edge >: $\longrightarrow$

Non-increasing edge ≥: $\longrightarrow$

Neither X: $\longrightarrow$

Merging

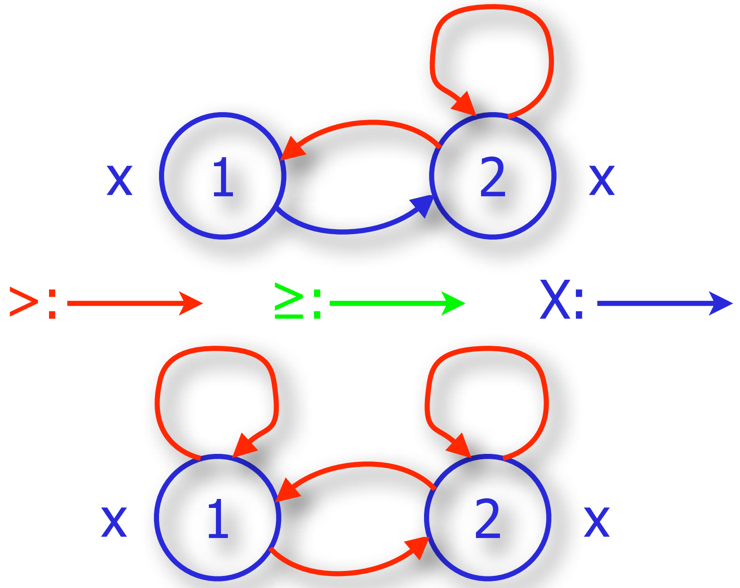
```
define f(x) =
  if  (!intp(x) or x ≤ 1)
    then 0
    else if  (x mod 2 = 1)
      then f(x+1)
      else 1 + f(x/2)
```



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

Merging

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



1. ⟨f, {intp(x), x > 1, x mod 2 = 1, intp(x+1), x+1 > 1, (x+1) mod 2 ≠ 1}, f((x+1)/2)⟩
2. ⟨f, {intp(x), x > 1, x mod 2 ≠ 1}, f(x/2)⟩

Full Algorithm

- We presented a simplified version of our analysis
- Mutual Recursion
- SCC analysis: can have more SCCs than functions
- Hierarchical Analysis
- Merging on a per-node basis: different edges correspond to different numbers of steps
- Multiple CCMs
- CCMs that combine formals (e.g., x - y)
- Different CCMs for different nodes
    - Saturation algorithm for propagating CCMs

Outline

JVM Example

```
(mutual-recursion
 ;; mma2 :: num, counts, s, ac --> [refs]
 (defun mma2 (c1 c2 s ac)
   (declare (xargs :measure (cons (len (cons c1 c2))
                                  (natural-sum (cons c1 c2)))))
   (if (zp c1)
       (mv (heap s) ac)
     (mv-let (new-addr new-heap)
             (mma c2 s)
             (mma2 (- c1 1)
                   c2
                   (make-state (thread-table s)
                               new-heap
                               (class-table s))
                   (cons (list 'REF new-addr) ac)))))

 ;; mma :: [counts], s --> addr, new-heap
 (defun mma (counts s)
   (declare (xargs :measure (cons (+ 1 (len counts))
                                  (natural-sum counts))))
   (if (<= (len counts) 1)

       ;; "Base case"  Handles initializing the final dimension
       (mv (len (heap s))
           (bind (len (heap s))
                 (makearray 'T_REF
                            (car counts)
                            (init-array 'T_REF (car counts))
                            (class-table s))
                 (heap s)))
     ;; "Recursive Case"
     (mv-let (heap-prime lst-of-refs)
             (mma2 (car counts)
                   (cdr counts)
                   s
                   nil)
             (let* ((obj (makearray 'T_REF
                                    (car counts)
                                    lst-of-refs
                                    (class-table s)))
                    (new-addr (len heap-prime))
                    (new-heap (bind new-addr obj heap-prime)))
               (mv new-addr new-heap)))))
 )
```

JVM Example

```
(mutual-recursion
  (defun mma2 (c1 c2 s ac)
    (if (zp c1)
        ...
        (mv-let (new-addr new-heap)
        (mma c2 s)
        (mma2 (- c1 1) c2 e e’)))

  (defun mma (c s)
    (if (<= (len c) 1)
        ...
        (mma2 (car c) (cdr c) s nil)
         ...)))

mma2 measure: (cons (len (cons c1 c2))
                    (natural-sum (cons c1 c2)))
mma measure: (cons (+ 1 (len c)) (natural-sum c))
```

Experimental Results

- Implemented in ACL2s
- Ran our algorithm over the ACL2 Regression Suite
  - Over 100MB, 11,000 function definitions
- Ignored all user hints and other explicit assistance
- Over 98% success rate

Experiment: No Theorems

Problems	Total	CCG	ACL2
Non-Trivial	**1762**	**1408 (80%)**	**1056 (60%)**
Recursive	4348	3394 (92%)	3642(84%)

Experiment: Theorems

Non-Trivial	**1762**	**1544 (87%)**	**1056 (67%)**
Recursive	4348	3394 (95%)	3642(87%)

Outline

Conclusions and Future Work

- Quintessential undecidable software verification problem
- Introduce Calling Context Graphs and Measures (CCGs and CCMs)
  - New, fully automatic termination analysis
  - General: Can be used to reason about any looping behavior
- Domain: fully-featured, first-order, applicative functional PLs
- Novel combination of static analysis and theorem proving
- Our analysis is implemented in ACL2s
- Experimental results: ACL2 regression suite
  - > 100MB of code > 11,000 function definitions
  - 98% success rate [MV'06]
- Use CCG analysis to generate measures
- Reason about imperative programs via SSA-like transformation

ACL2s

- ACL2 theorem prover
  - Runs like a well-tuned race car in the hands of an expert
  - Unfortunately, novices don't have the same experience
  - Disseminate: wrote a book
  - Not enough: undergrads
- ACL2s: The ACL2 Sedan
  - From race car to sedan
  - Self-teaching
  - Control a machine that is thinking about other machines
  - Visualize what ACL2 is doing
  - Levels
  - Termination: e.g., my-merge
  - Used in several classes
  - Available for download [DMMV'07]