All-Termination($T$)

Aaron Turon
Northeastern University
turon@ccs.neu.edu

(joint work with Pete Manolios)
Consider a recursive list-insertion procedure:

\[
\text{define } \textbf{insert}(i, \text{item}, \text{list}) = \\
\text{if } i \leq 0 \text{ or empty(list)} \\
\text{then cons(item, list)} \\
\text{else cons(first(list),} \\
\text{insert}(i-1, \text{item}, \text{rest(list)})
\]
Consider a recursive list-insertion procedure:

\[
\text{define } \textbf{insert}(i, \text{item}, \textbf{list}) = \\
\text{if } i \leq 0 \text{ or empty} (\textbf{list}) \\
\text{then cons(item, list)} \\
\text{else cons(first(list),} \\
\text{insert(i-1, item, rest(list)))}
\]

How do we prove that \textbf{insert} terminates?
Consider a recursive list-insertion procedure:

\[
\text{define } \textbf{insert}(i, \text{item, list}) = \\
\quad \text{if } i \leq 0 \text{ or empty(list)} \\
\quad \text{then } \text{cons(item, list)} \\
\quad \text{else } \text{cons(first(list), insert(i-1, item, rest(list)))}
\]

How do we prove that \textbf{insert} terminates?

\[
m_1(i, \text{item, list}) = |i|
\]
Consider a recursive list-insertion procedure:

\[
\text{define } \text{insert}(i, \text{item}, \text{list}) = \\
\text{if } i \leq 0 \text{ or empty(list) then cons(item, list) \\
else cons(first(list),}\ \\
\text{insert}(i-1, \text{item}, \text{rest(list)})\]

How do we prove that insert terminates?

\[
m_1(i, \text{item}, \text{list}) = |i| \\
m_2(i, \text{item}, \text{list}) = \text{length(list)}
\]
Consider a recursive list-insertion procedure:

```plaintext
define insert(i, item, list) =
  if i <= 0 or empty(list)
  then cons(item, list)
  else cons(first(list),
            insert(i-1, item, rest(list)))
```

How do we prove that `insert` terminates?

```plaintext
m_1(i, item, list) = |i|
m_2(i, item, list) = length(list)
m_3(i, item, list) = |i| + length(list)
```
Consider a recursive list-insertion procedure:

```
define insert(i, item, list) =
  if i <= 0 or empty(list)
  then cons(item, list)
  else cons(first(list),
            insert(i-1, item, rest(list)))
```

How do we prove that `insert` terminates?

Are these proofs different in an important way?

```
m_1(i, item, list) = |i|
m_2(i, item, list) = length(list)
m_3(i, item, list) = |i| + length(list)
```
Consider a recursive list-insertion procedure:

\[
\text{define } \text{insert}(i, \text{item}, \text{list}) = \\
\text{if } i \leq 0 \text{ or empty}() \text{ then cons(item, list)} \\
\text{else cons(first(list), insert}(i-1, \text{item}, \text{rest(list)}))
\]

How do we prove that \text{insert} terminates?

Are these proofs different in an important way?

\[
m_1(i, \text{item}, \text{list}) = |i| \quad m_2(i, \text{item}, \text{list}) = \text{length(list)} \quad m_3(i, \text{item}, \text{list}) = |i| + \text{length(list)}
\]
Consider a recursive list-insertion procedure:

\[
\text{define } \text{insert}(i, \text{item}, \text{list}) = \\
\text{if } i \leq 0 \text{ or empty(list)} \\
\text{then cons(item, list)} \\
\text{else cons(first(list), insert(i-1, item, rest(list)))}
\]

How do we prove that \text{insert} terminates?

Are these proofs different in an important way?

Measured subsets for \text{insert}:

- \{i\}
- \{\text{list}\}
- \{i, \text{list}\}
Measured sets $\rightarrow$ induction schemes [BoyeroMoore, 1979]

To prove $\forall i,\text{item},\text{list} :: \varphi(i,\text{item},\text{list})$, show:
Measured sets $\rightarrow$ induction schemes [Boyer&Moore, 1979]

To prove $\forall i,\text{item},\text{list} :: \varphi(i,\text{item},\text{list})$, show:

Measured subset $\{i\}$

$\varphi(0,\text{item},\text{list})$

$[\forall y,z :: \varphi(i,y,z)] \Rightarrow \varphi(i+1,\text{item},\text{list})$
Measured sets $\rightarrow$ induction schemes \cite{BoyerMoore, 1979}

To prove $\forall i, \text{item}, \text{list} :: \varphi(i, \text{item}, \text{list})$, show:

Measured subset $\{i\}$

- $\varphi(0, \text{item}, \text{list})$
- $[\forall y, z :: \varphi(i, y, z)] \Rightarrow \varphi(i+1, \text{item}, \text{list})$

Measured subset $\{\text{list}\}$

- $\varphi(i, \text{item}, \text{nil})$
- $[\forall x, y :: \varphi(x, y, \text{list})] \Rightarrow \varphi(i, \text{item}, \text{cons}(a, \text{list}))$
Measured sets ↦ induction schemes [Boyer&Moore, 1979]

To prove ∀i,item,list :: φ(i,item,list), show:

Measured subset \{i\}

φ(0,item,list)

[ ∀y,z :: φ(i,y,z) ] ⇒ φ(i+1,item,list)

Measured subset \{list\}

φ(i,item,nil)

[ ∀x,y :: φ(x,y,list) ] ⇒ φ(i,item,cons(a,list))

Measured subset \{i,list\}

φ(0,item,nil)

[ ∀x :: φ(i,x,list) ] ⇒ φ(i+1,item,cons(a,list))
Define functions

Make conjectures

User

Known functions

Rewriting

Induction

Proving engine

Known facts

Theorem prover

Life with a theorem prover:
Life with a theorem prover:

- Define functions
- Prove termination
- Make conjectures

Theorem prover:

- Known functions
  - Rewriting
  - Induction
- Known facts

User

TACAS 2009 - March 26
Aaron Turon - All-Termination(T)
Life with a theorem prover:

User

- Define functions
- Prove termination
- Make conjectures

Theorem prover

- Known functions
- Rewriting
- Induction

Proving engine

- Known facts
Life with a theorem prover:

Define functions → Termination analysis → Known functions → Proving engine → Theorem prover

Make conjectures

User

Known functions

Rewriting

Induction

Known facts

Proving engine
Life with a theorem prover:

- Define functions
- Make conjectures

Restricted termination analysis

- Known functions
  - Rewriting
  - Induction

Proving engine

- Known facts

Theorem prover
[A better] life with a theorem prover:

Define functions  →  All-Termination analysis  →  Known functions

Make conjectures  →  Proving engine

- Rewriting
- Induction

Known facts

User

Theorem prover
The rest of the talk:

- All-Termination($T$)
  - definition
  - research program

- Size-change termination (SCT)

- All-Termination(SCT)
  - complexity results
  - experimental results
The rest of the talk:

- **All-Termination**($T$)
  - definition
  - research program
- Size-change termination (SCT)
- All-Termination(SCT)
  - complexity results
  - experimental results
Termination analysis

• Termination undecidable

• Sound, incomplete analyses:

\[ T : \text{Programs} \rightarrow \text{Bool predicate such that if } T(P) \text{ then } P \text{ terminates on all inputs} \]
Termination analysis

- Termination undecidable
- Sound, incomplete analyses:

\[ T : \text{Programs} \rightarrow \text{Bool predicate such that} \]
\[ \text{if } T(P) \text{ then } P \text{ terminates on all inputs} \]

Restricted termination analysis

\[ T : \text{Programs} \times 2^{\text{Variables}} \rightarrow \text{Bool} \]

such that

\[ \text{if } T(P,V) \text{ then } V \text{ is a measured subset for } P \]
Measured sets are upward-closed:

if $U$ is a measured subset for $P$, and $U \subseteq V$
then $V$ is a measured subset for $P$
Measured sets are upward-closed:

if $U$ is a measured subset for $P$, and $U \subseteq V$
then $V$ is a measured subset for $P$

**All-Termination(T) analysis**

\[
\text{All-Termination}(T)(P) \overset{\text{def}}{=} \text{minimal}\{V \mid T(P,V)\}
\]

where $T$ is a restricted termination analysis.

The “**termination cores of $P$ modulo $T$**”. 
Measured sets are upward-closed:

if $U$ is a measured subset for $P$, and $U \subseteq V$
then $V$ is a measured subset for $P$

All-Termination($T$) analysis

$$\text{All-Termination}(T)(P) \overset{\text{def}}{=} \text{minimal}\{V \mid T(P,V)\}$$

where $T$ is a restricted termination analysis.

Warning

$|\text{All-Termination}(T)(P)|$ can be exponential in $|P|$. 
Theorem:

if $T$ is in PSPACE then $\text{AllTermination}(T)$ is in PSPACE.

Proof:

$\text{All-Termination}(T)(P)$:

for each $V \subseteq \text{vars}(P)$

if $T(P,V)$ then

minimal := true

for each $U \subset V$

if $T(P,U)$ then minimal := false

if minimal then output($V$)
Research program

• Begin with standard termination analysis, \( A \)

• Define restricted version, \( T \), so that
  \[
  \exists V :: T(P,V) \iff A(P)
  \]

• *Instrument* \( A \) to produce a “certificate” \( C \)

• Implement All-Termination(\( T \))(\( P \)) by
  – running \( A \) on \( P \) to produce \( C \)
  – *extracting* termination cores from \( C \)
The rest of the talk:

- All-Termination($T$)
  - definition
  - research program
- Size-change termination (SCT)
- All-Termination(SCT)
  - complexity results
  - experimental results
Size-change termination [Lee et al, POPL01] works by analyzing a safe abstraction of the program.

\[
\begin{align*}
\text{ack}(\emptyset, n) &= n+1 \\
\text{ack}(m, 0) &= 1\text{ack}(m-1, 1) \\
\text{ack}(m, n) &= 2\text{ack}(m-1, 3\text{ack}(m, n-1))
\end{align*}
\]
Size-change termination \cite{Lee et al, POPL01} works by analyzing a safe abstraction of the program.

\[
\begin{align*}
\text{ack}(0, n) & = n+1 \\
\text{ack}(m, 0) & = 1\text{ack}(m-1, 1) \\
\text{ack}(m, n) & = 2\text{ack}(m-1, 3\text{ack}(m, n-1))
\end{align*}
\]
Size-change termination [Lee et al, POPL01] works by analyzing a safe abstraction of the program.

\[
\begin{align*}
\text{ack}(0,n) &= n+1 \\
\text{ack}(m,0) &= 1\text{ack}(m-1, 1) \\
\text{ack}(m,n) &= 2\text{ack}(m-1, 3\text{ack}(m, n-1))
\end{align*}
\]
Size-change termination [Lee et al, POPL01] works by analyzing a safe abstraction of the program.

\[
\begin{align*}
\text{ack}(0, n) &= n+1 \\
\text{ack}(m, 0) &= 1 \text{ack}(m-1, 1) \\
\text{ack}(m, n) &= 2 \text{ack}(m-1, 3 \text{ack}(m, n-1))
\end{align*}
\]
Size-change termination \cite{Lee et al, POPL01} works by analyzing a safe abstraction of the program.

\[
\begin{align*}
\text{ack}(\emptyset, n) &= n + 1 \\
\text{ack}(m, \emptyset) &= \text{ack}(m-1, 1) \\
\text{ack}(m, n) &= \text{ack}(m-1, \text{ack}(m, n-1))
\end{align*}
\]
Size-change graph composition

\[
\begin{array}{c}
\begin{array}{c}
\text{x} > x \\
y & y
\end{array} \\
\end{array}
; \\
\begin{array}{c}
\begin{array}{c}
\text{x} \geq x \\
y & y
\end{array}
\end{array}
= \\
\begin{array}{c}
\begin{array}{c}
\text{x} > x \\
y & y
\end{array}
\end{array}
\]

Size-change graph composition

\[
\begin{align*}
\begin{array}{c}
\text{x > x} \\
y \quad y
\end{array}
\end{align*}
\begin{align*}
; &= \begin{array}{c}
\text{x \geq x} \\
y \quad y
\end{array}
\end{align*}
\begin{align*}
\begin{array}{c}
x > x \\
y \quad y
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
x > x \\
y \quad y
\end{array}
\end{align*}
\begin{align*}
; &= \begin{array}{c}
x > x \\
y \quad y
\end{array}
\end{align*}
\begin{align*}
\begin{array}{c}
x \quad x \\
y \quad y
\end{array}
\end{align*}
\]

Size-change graph composition

\[
\begin{align*}
\begin{array}{c}
\begin{array}{c}
\xrightarrow{x > x} \\
y & y
\end{array} \\
\; ; \\
\begin{array}{c}
\xrightarrow{x \geq x} \\
y & y
\end{array}
\end{array}
& = \\
\begin{array}{c}
\begin{array}{c}
\xrightarrow{x > x} \\
y & y
\end{array}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\begin{array}{c}
\xrightarrow{x > x} \\
y & y
\end{array} \\
\; ; \\
\begin{array}{c}
\xrightarrow{x > x} \\
y & y
\end{array}
\end{array}
& = \\
\begin{array}{c}
\begin{array}{c}
x \\
y & y
\end{array}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\begin{array}{c}
\xrightarrow{x \geq x} \\
y & > y
\end{array} \\
\; ; \\
\begin{array}{c}
\xrightarrow{x \geq x} \\
y & > y
\end{array}
\end{array}
& = \\
\begin{array}{c}
\begin{array}{c}
\xrightarrow{x \geq x} \\
y & > y
\end{array}
\end{array}
\end{align*}
\]

Idempotent
Size-change termination

Let $cl(ACG)$ denote the composition closure of $ACG$.

Definition: $ACG$ is size-change terminating if every idempotent in $cl(ACG)$ has a strict self-edge.
Size-change termination

Let $cl(ACG)$ denote the composition closure of $ACG$.

**Definition:** $ACG$ is size-change terminating if every idempotent in $cl(ACG)$ has a strict self-edge.

$$cl\{1,3\} = \{1,3\}$$

$1,3$ are idempotent, and have strict self-edges
Size-change termination

Let $cl(ACG)$ denote the composition closure of $ACG$. 

**Definition:** $ACG$ is size-change terminating if every idempotent in $cl(ACG)$ has a strict self-edge.

Size-change termination is PSPACE-complete.

But size-change analysis needs an ACG.

We use *Calling Context Graphs* [Manolios, Vroon CAV2006] to find ACGs.
The rest of the talk:

- All-Termination($T$)
  - definition
  - research program
- Size-change termination (SCT)
- All-Termination(SCT)
  - complexity results
  - experimental results
The restricted version of SCT

Let \textit{restrict}(ACG, V) be ACG, but with only size-change edges relating variables in V.

\textbf{Theorem: if}

- ACG is a valid annotated call graph for P
- \textit{SCT}(restrict(ACG,V))

then V is a measured subset for P.
Another example

\[\text{dswap}(\theta, y) = y\]
\[\text{dswap}(x, \theta) = x\]
\[\text{dswap}(x, y) = \text{dswap}(y-1, x-1)\]
Another example

\[ \text{dswap}(\theta, y) = y \]
\[ \text{dswap}(x, \theta) = x \]
\[ \text{dswap}(x, y) = \text{dswap}(y-1, x-1) \]

\[ \begin{array}{c}
x & x \\
y & y
\end{array} \quad ; \quad \begin{array}{c}
x & x \\
y & y
\end{array} = \begin{array}{c}
x & x \\
y & y
\end{array} \]
**Strategy**: annotate edges with the variables they use
**Strategy**: annotate edges with the variables they use.

In general, need *sets* of *sets* of variables:

\[
\begin{array}{ccc}
  \text{\text{\text{x}}} & \text{\text{\text{x}}} & ; \\
  \text{\text{\text{y}}} & \text{\text{\text{y}}} & \text{\text{\text{y}}} \\
\end{array}
\]

\[
\begin{array}{ccc}
  \text{\text{\text{x}}} & \text{\text{\text{x}}} & = \\
  \text{\text{\text{y}}} & \text{\text{\text{y}}} & \text{\text{\text{y}}} \\
\end{array}
\]

The final y-to-y edge may involve \{x, y\} or just \{y\}.
**Strategy:** annotate edges with the variables they use

In general, need *sets* of *sets* of variables:

The final y-to-y edge may involve \( \{x,y\} \) or just \( \{y\} \).

For programs such as *insert*, the graphs are simple:
Composition of edge-annotated graphs

• results in same edges as before

• new edge annotations are union-of-crossproduct

\[ \{w_1, ..., w_n\} \cdot \{v_1, ..., v_m\} = \{w_1 \cup v_1, w_1 \cup v_2, ..., w_n \cup v_m\} \]

Let \( acl(ACG) \) denote the annotated closure of ACG.
Key Theorem:

\[ \text{SCT}(\text{restrict}(\text{ACG}, V)) \]

iff

every idempotent in \( \text{acl}(\text{ACG}) \)
has a strict self-edge, labeled with some set \( \{ W_1, \ldots, W_n \} \), such that \( W_i \subseteq V \) for some \( I \)

This shows we can extract measured subsets from the instrumented analysis.
After running SCT with edge annotations, we have:

- A set of idempotent size-change graphs
- Each of which has a set of strict self-edges
- Each of which has a set of variable sets

**To find a single measured subset, we choose:**

- One set of variables per strict self-edge
- From one strict self-edge per graph
After running SCT with edge annotations, we have:

- A set of idempotent size-change graphs
- Each of which has a set of strict self-edges
- Each of which has a set of variable sets

To find a **single** measured subset, we choose:

- One set of variables per strict self-edge
- From one strict self-edge per graph

The measured subset is the union of the annotations we chose.
To find all the (minimal) measured subsets:

- Build a boolean constraint system $\varphi$ that captures the measured subset requirements
- $|\varphi| = O(acl|ACG|)$
- $\varphi$ can be made dual-horn: can find $\psi$ that is
  - equisatisfiable to $\varphi$
  - conjunction of clauses,
  - each clause a disjunction of literals
  - at most one negative literal per clause
- min solutions to $\varphi$ can be found from $\psi$ efficiently
Complexity result

Output may be exponential, so what can we say?

Can look for output-sensitive complexity: running time reflects actual number of outputs.

Theorem:

After computing $\varphi$, we can find $k$ elements of $\text{All-Termination}(\text{SCT})(P)$ in time $O(|\text{acl}(\text{ACG})|^k)$

This leads to a pay-as-you-go, incremental algorithm for finding termination cores.
Experimental results

We implemented our algorithm for ACL2, on top of calling context graphs.

ACL2 has a large regression suite:

- >100MB
- >11,000 function definitions (each of which must be proved terminating)
- Code ranging from bit-vector libraries to model checkers
Experimental results

The setup: we ran CCG + All-Termination(SCT) on the entire regression suite.

Number of functions: \( >11,000 \)
Proved terminating: 98%  
\textit{(note: same as CCG+SCT)}

Multiargument functions:

- Proved terminating: 1728
- With “nontrivial” cores: 90%
- With multiple cores: 7%
- Maximum core count: 3

Running time (not including CCG): 30 seconds
Experimental results

The setup: we ran CCG + All-Termination(SCT) on the entire regression suite.

Number of functions: >11,000  
Proved terminating: 98%  (note: same as CCG+SCT)

Multiargument functions:
  Proved terminating 1728
  With “nontrivial” cores 90%
  With multiple cores 7%
  Maximum core count 3

Running time (not including CCG): 30 seconds
Experimental results

The setup: we ran CCG + All-Termination(SCT) on the entire regression suite.

Number of functions:  >11,000
Proved terminating:  98%  \textit{(note: same as CCG+SCT)}

Multiargument functions:
   Proved terminating  1728
   \textbf{With “nontrivial” cores}  90%
   With multiple cores  7%
   \textbf{Maximum core count}  3  \textit{(note: this is the $k$ parameter)}

Running time (not including CCG):  30 seconds
Further work

- Study All-Termination(T) for additional T
  - We've explored polynomial size-change
- Extend our prototype to the ACL2 Sedan
  - Will help our freshman users at Northeastern
- Explore new applications of measured subsets
  - We've got a few in mind, but want to hear yours
Conclusion

- Measured sets known, but unstudied until now
- We introduced **All-Termination(T):**
  find all measurable sets for a program P, modulo T
- We studied **All-Termination(SCT),** showed it
  - PSPACE-complete
  - Workable in practice