All-Termination(*T*)

Aaron Turon
Northeastern University
turon@ccs.neu.edu

(joint work with Pete Manolios)

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)))
```

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?

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?

m_1(i, item, list) = |i|

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?

m_1(i, item, list) = |i|
m_2(i, item, list) = 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?

$$m_1(i, item, list) = |i|$$
$$m_2(i, item, list) = length(list)$$
$$m_3(i, item, list) = |i| + length(list)$$

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:

```
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:

```
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?

Measured subsets for insert

{i}
{list}
{i,list}

Measured sets $\rightarrowtail$ induction schemes [*Boyer&Moore, 1979*]

To prove $\forall$ i,item,list :: φ(i,item,list), show:

Measured sets $\rightarrowtail$ induction schemes [*Boyer&Moore, 1979*]

To prove $\forall$ i,item,list :: φ(i,item,list), show:

Measured subset {i}

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

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

Measured sets $\rightarrowtail$ induction schemes [*Boyer&Moore, 1979*]

To prove $\forall$ i,item,list :: φ(i,item,list), show:

Measured subset {i}

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

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

Measured subset {list}

$\varphi(i, item, \mathbf{nil})$

$[\forall x,y :: \varphi(x, y, \mathbf{list})] \Rightarrow \varphi(i, item, \mathbf{cons(a,list)})$

Measured sets $\rightarrowtail$ induction schemes [*Boyer&Moore, 1979*]

To prove $\forall$ i,item,list :: φ(i,item,list), show:

Measured subset {i}

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

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

Measured subset {list}

$\varphi(\text{i},\text{item},\mathbf{nil})$

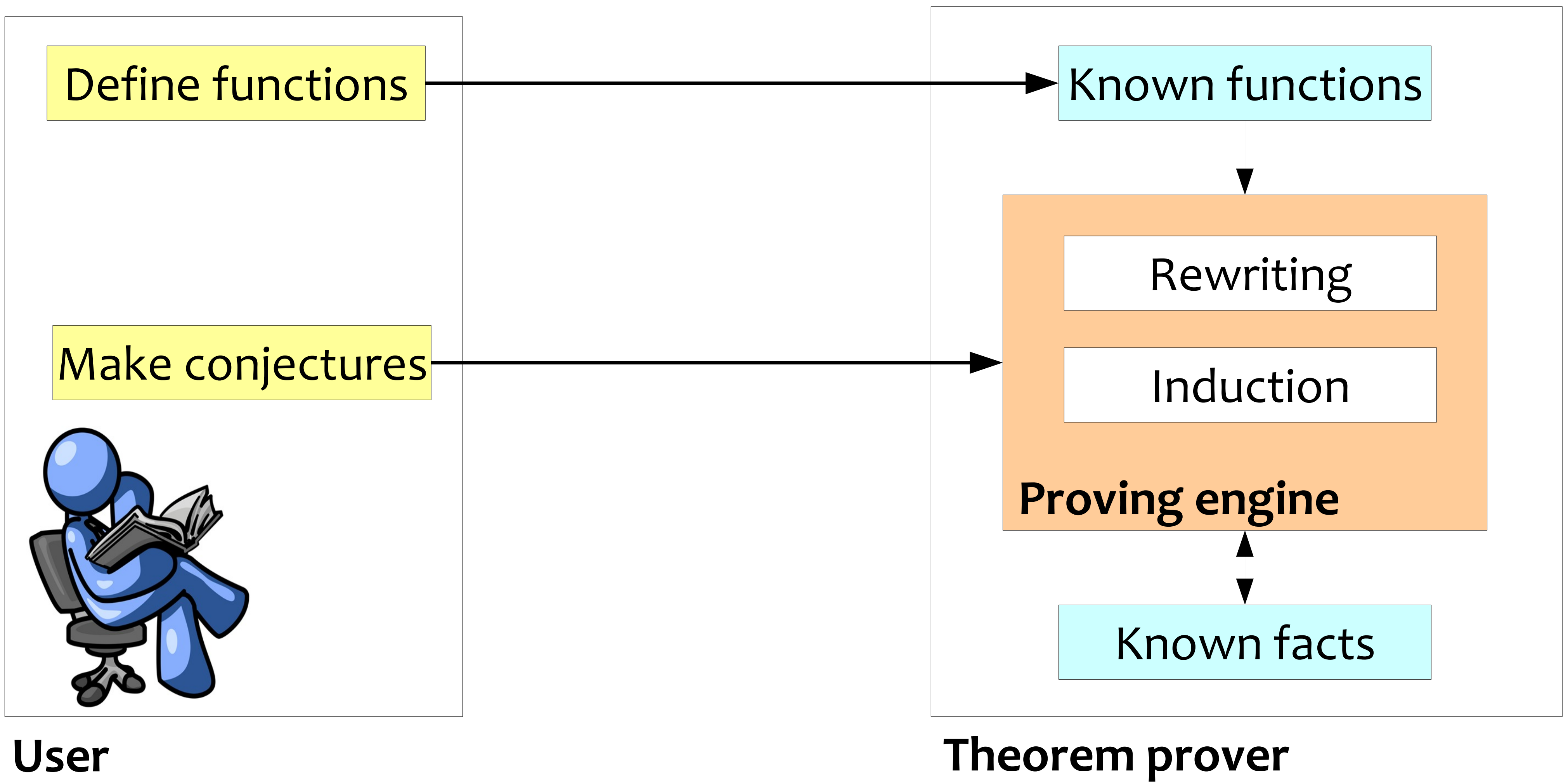
$[\forall x,y :: \varphi(x,y,\mathbf{list})] \Rightarrow \varphi(\text{i},\text{item},\mathbf{cons(a,list)})$

Measured subset {i,list}

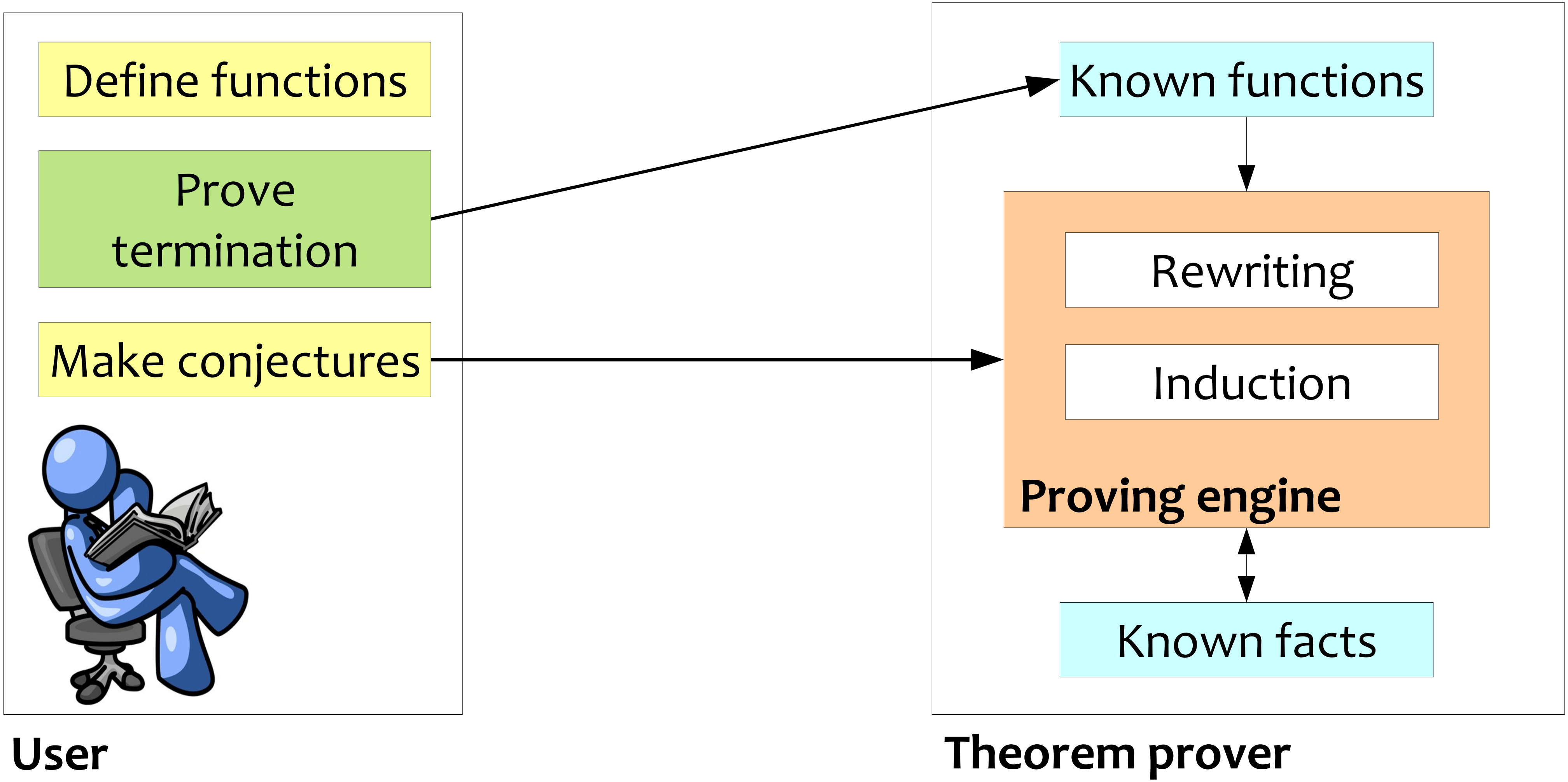
$\varphi(\mathbf{0},\text{item},\mathbf{nil})$

$[\forall x :: \varphi(\mathbf{i},x,\mathbf{list})] \Rightarrow \varphi(\mathbf{i+1},\text{item},\mathbf{cons(a,list)})$

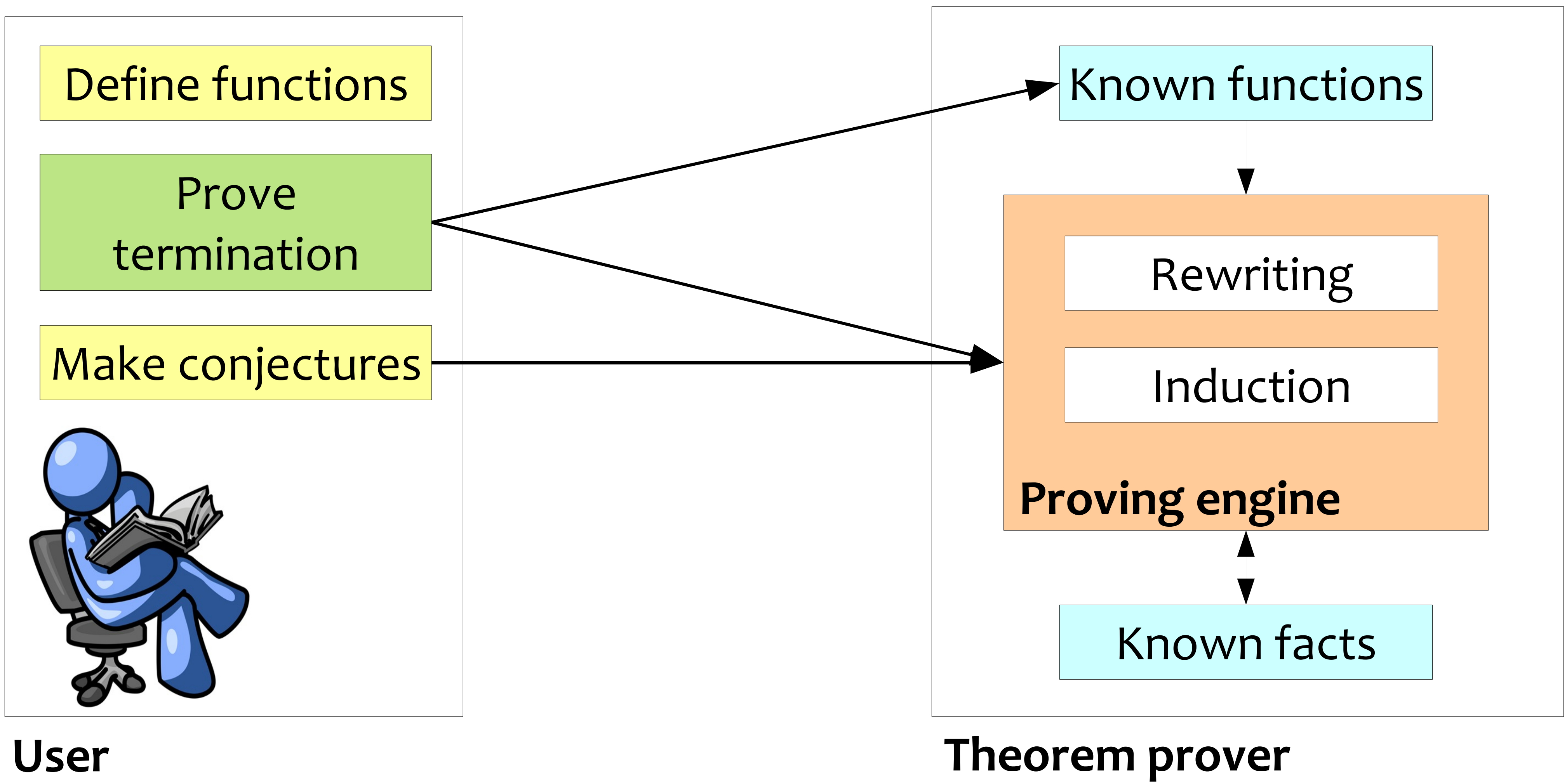
Life with a theorem prover:



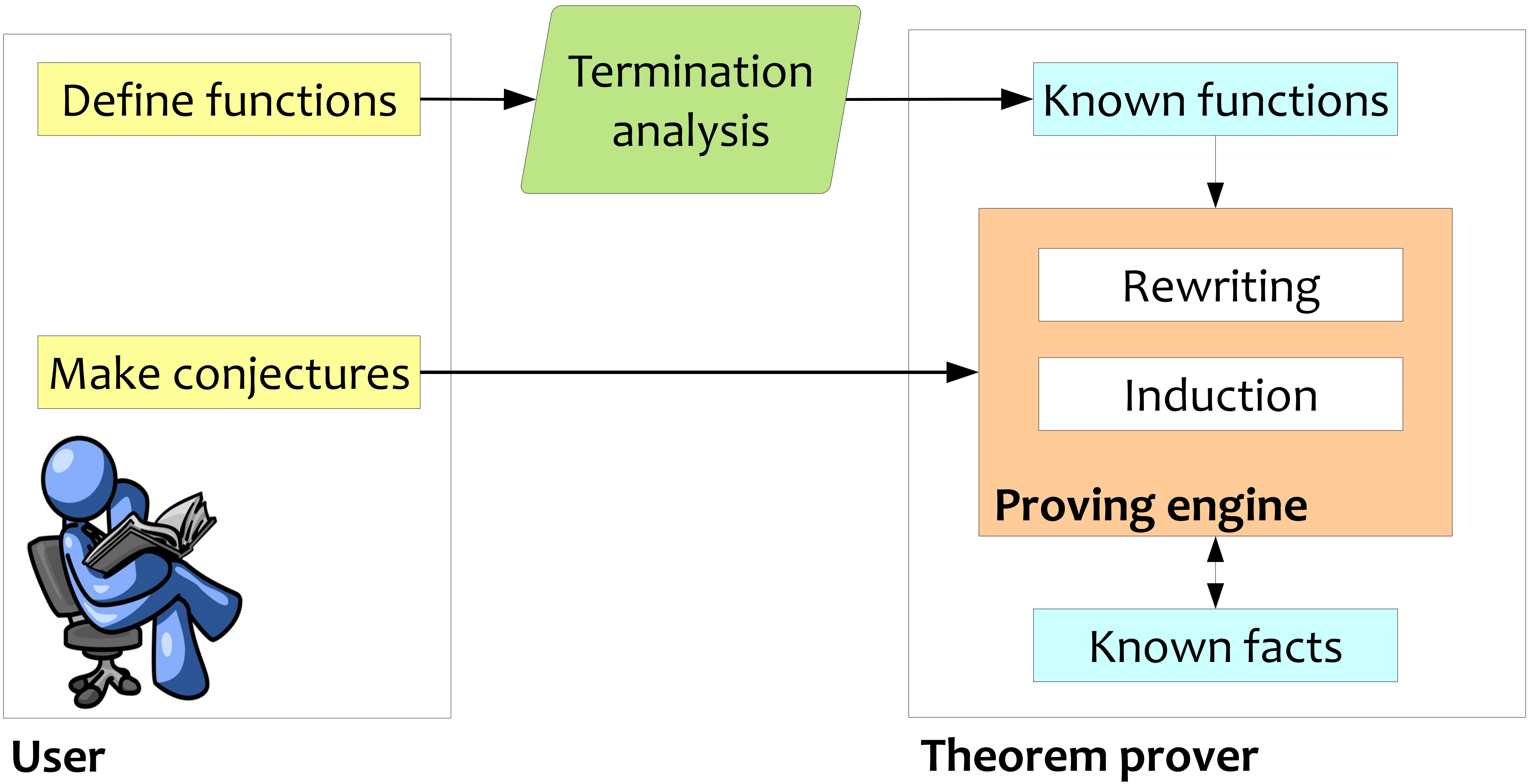
Life with a theorem prover:



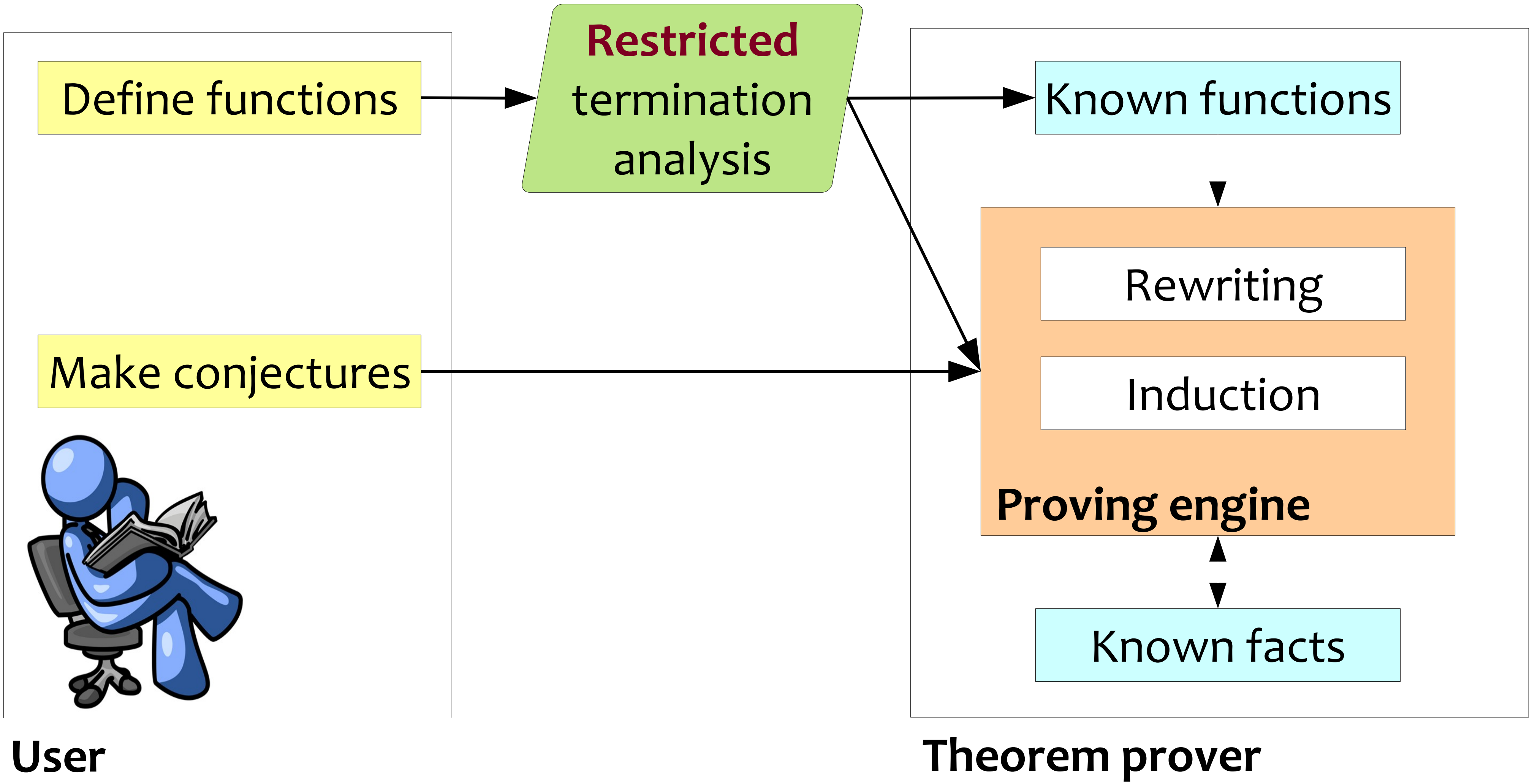
Life with a theorem prover:



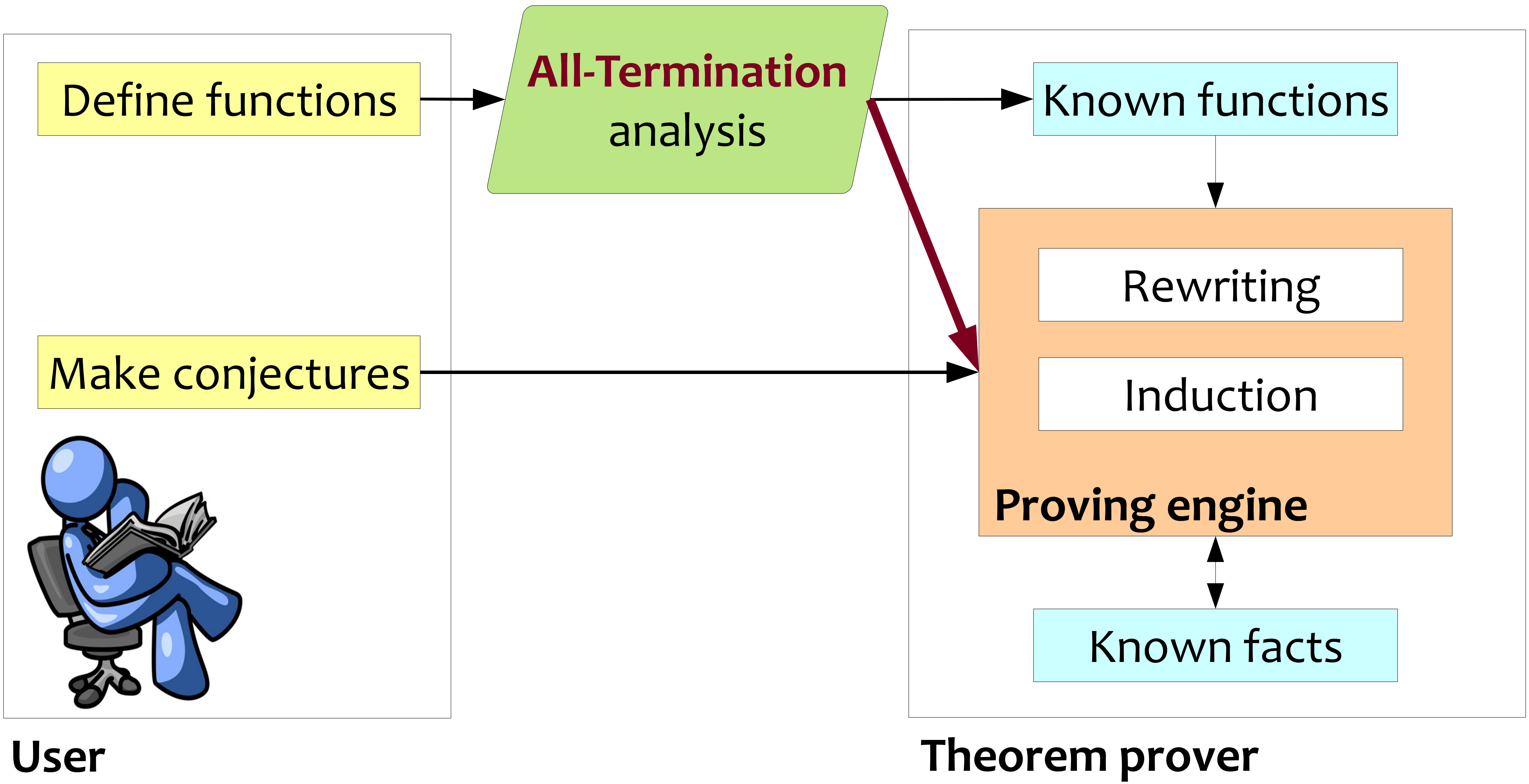
Life with a theorem prover:



Life with a theorem prover:



[A better] life with a 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 : Programs → Bool predicate such that
  if **T**(**P**) then **P** terminates on all inputs

Termination analysis

- Termination undecidable
- Sound, incomplete analyses:

T : Programs → Bool predicate such that

if **T**(**P**) then **P** terminates on all inputs

Restricted termination analysis

$$\mathbf{T} : \text{Programs} \times 2^{\text{Variables}} \to \text{Bool}$$

such that

if **T**(**P**,**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

Measured sets are upward-closed:

if **U** is a measured subset for P, and $\mathbf{U} \subseteq \mathbf{V}$
then **V** is a measured subset for P

All-Termination(T) analysis

$$\text{All-Termination}(\mathbf{T})(\mathrm{P}) \stackrel{\text{def}}{=} \mathit{minimal}\{\mathbf{V} \mid \mathbf{T}(\mathrm{P},\mathbf{V})\}$$

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 $\mathbf{U} \subseteq \mathbf{V}$

then **V** is a measured subset for P

All-Termination(T) analysis

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

where **T** is a restricted termination analysis.

Warning

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

Theorem:

if **T** is in PSPACE then **AllTermination(T)** is in PSPACE.

Proof:

```
All-Termination(T)(P):
  for each V ⊆ vars(P)
     if T(P,V) then
        minimal := true
        for each U ⊊ 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

$$[\exists\, V :: \mathbf{T}(P,V)] \Leftrightarrow \mathbf{A}(P)$$

- *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:

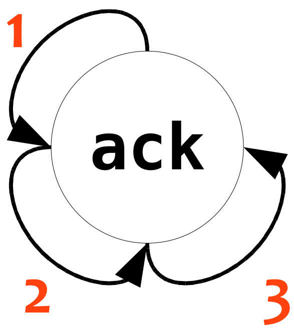
- All-Termination(T)
  - definition
  - research program
- **Size-change termination (SCT)**
- All-Termination(SCT)
  - complexity results
  - experimental results

Size-change termination [*Lee et al, POPL01*] works by analyzing a safe abstraction of the program.

```
ack(0,n) = n+1
ack(m,0) = ¹ack(m-1, 1)
ack(m,n) = ²ack(m-1, ³ack(m, n-1))
```

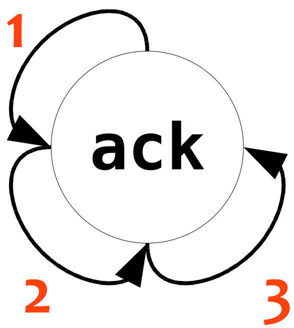
Size-change termination [*Lee et al, POPL01*] works by analyzing a safe abstraction of the program.

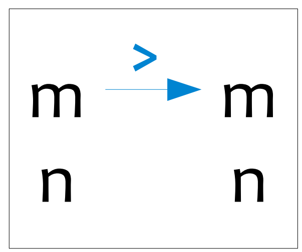
```
ack(0,n) = n+1
ack(m,0) = 1ack(m-1, 1)
ack(m,n) = 2ack(m-1, 3ack(m, n-1))
```



Size-change termination [*Lee et al, POPL01*] works by analyzing a safe abstraction of the program.

$$\mathbf{ack}(0,n) = n+1$$
$$\mathbf{ack}(m,0) = {}^{1}\mathbf{ack}(m-1,\ 1)$$
$$\mathbf{ack}(m,n) = {}^{2}\mathbf{ack}(m-1,\ {}^{3}\mathbf{ack}(m,\ n-1))$$

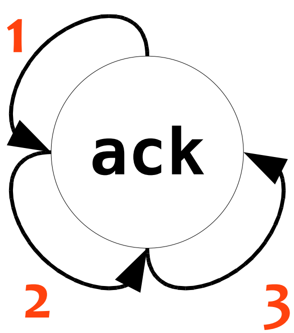


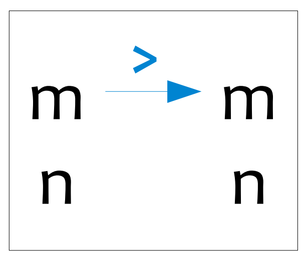


1

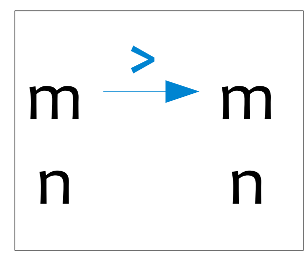
Size-change termination [*Lee et al, POPL01*] works by analyzing a safe abstraction of the program.

```
ack(0,n) = n+1
ack(m,0) = ¹ack(m-1, 1)
ack(m,n) = ²ack(m-1, ³ack(m, n-1))
```





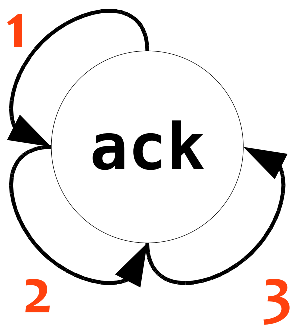
1

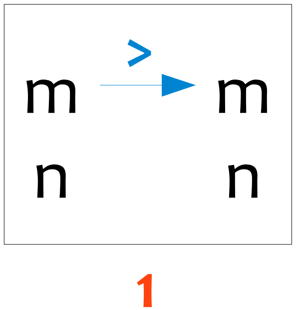


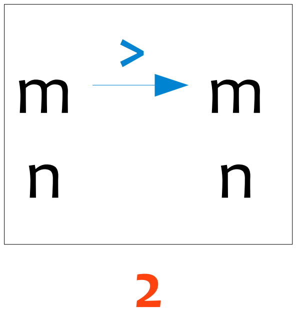
2

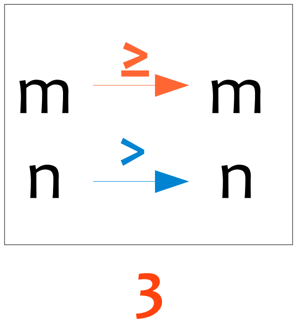
Size-change termination [*Lee et al, POPL01*] works by analyzing a safe abstraction of the program.

```
ack(0,n) = n+1
ack(m,0) = ¹ack(m-1, 1)
ack(m,n) = ²ack(m-1, ³ack(m, n-1))
```

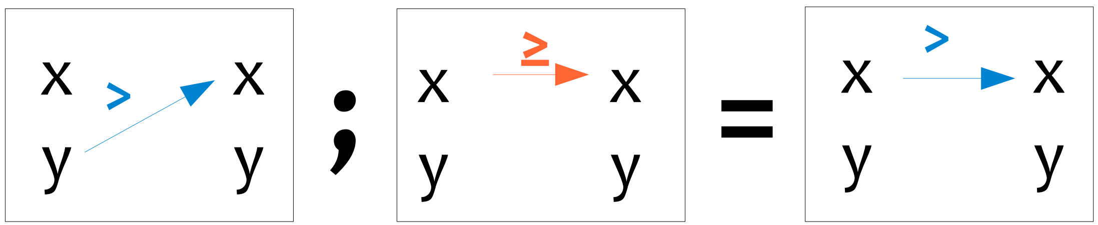




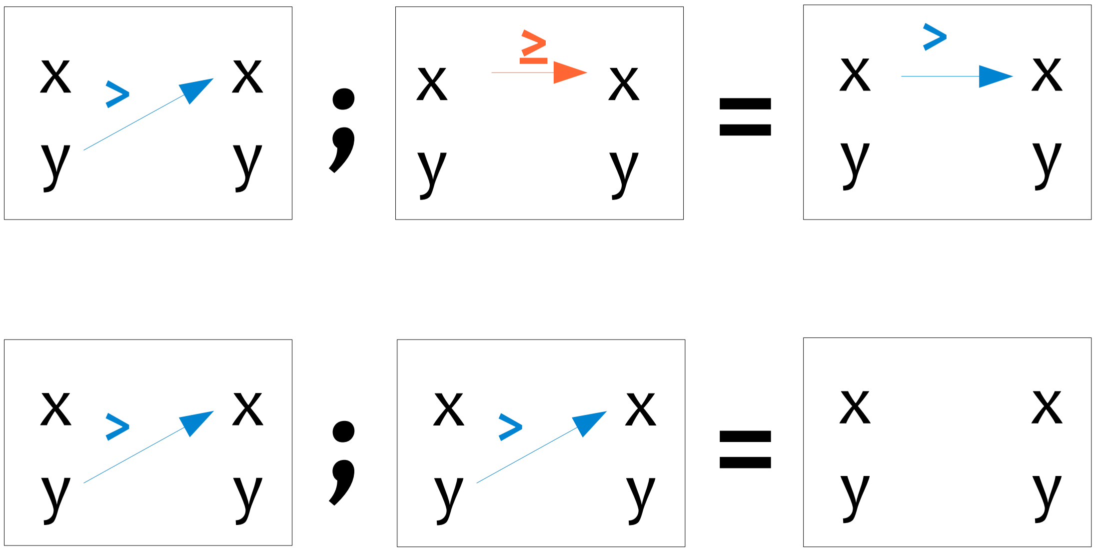




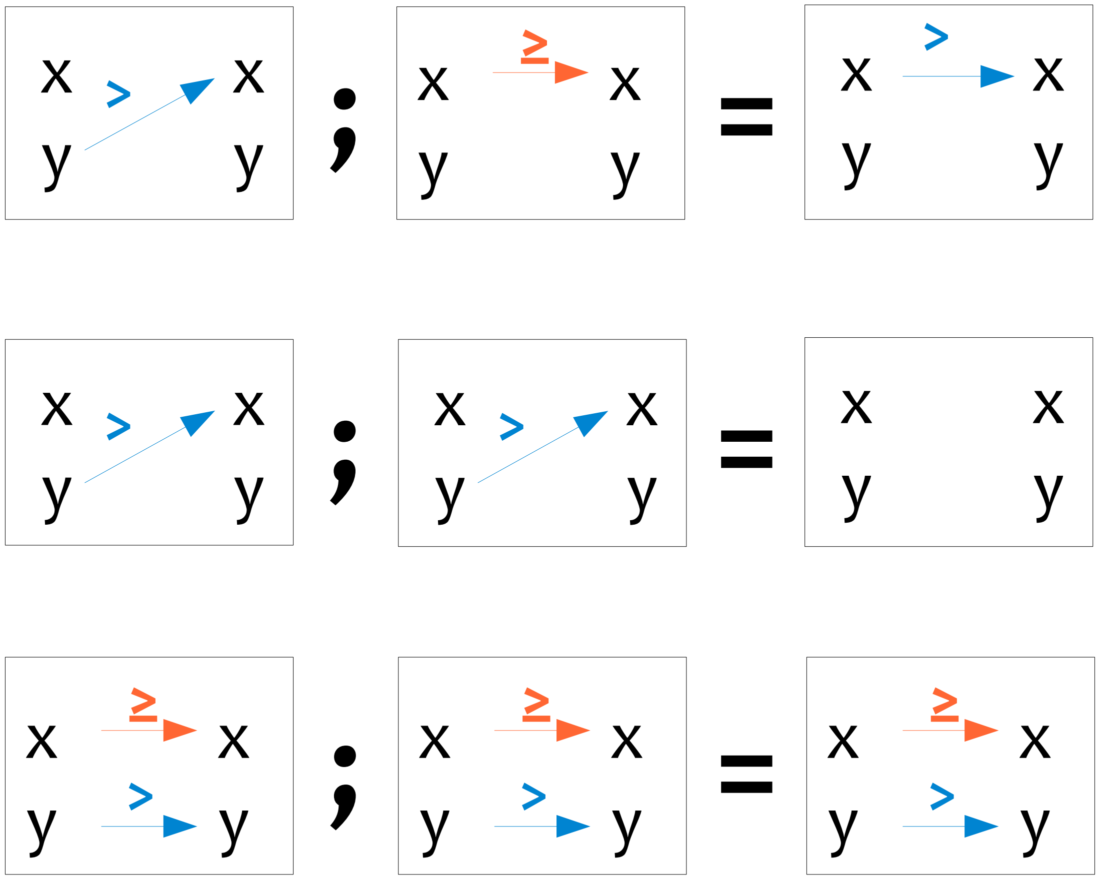
Size-change graph composition



Size-change graph composition



Size-change graph composition



Idempotent

Size-change termination

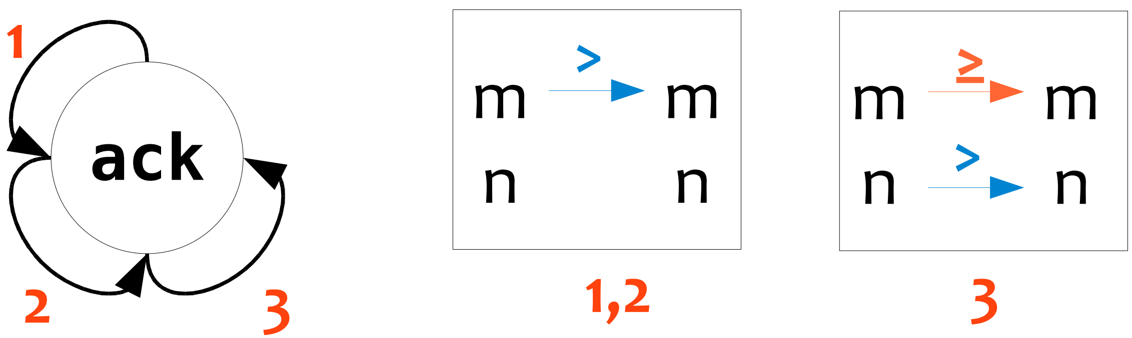
Let ***cl*(ACG)** denote the *composition closure* of **ACG**.

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 *restrict*(ACG, V) be ACG, but with only size-change edges relating variables in V.

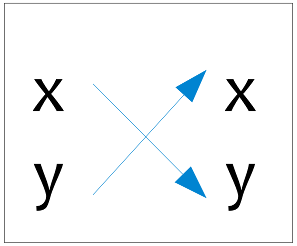
Theorem: if

- ACG is a valid annotated call graph for P
- **SCT**(*restrict*(ACG,V))

then V is a measured subset for P.

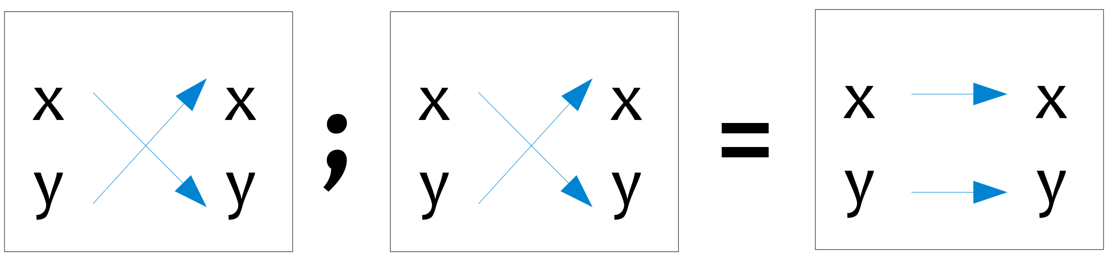
Another example

```
dswap(0,y) = y
dswap(x,0) = x
dswap(x,y) = dswap(y-1,x-1)
```



Another example

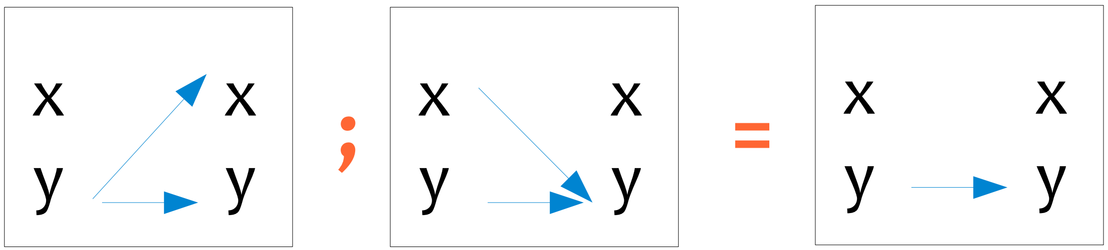
```
dswap(0,y) = y
dswap(x,0) = x
dswap(x,y) = dswap(y-1,x-1)
```



Strategy: annotate edges with the variables they use

Strategy: annotate edges with the variables they use

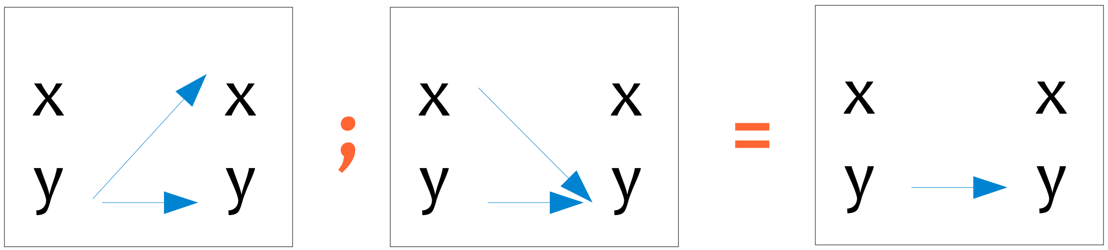
In general, need **sets** of **sets** of variables:



The final y-to-y edge may involve **{x,y}** or just **{y}**.

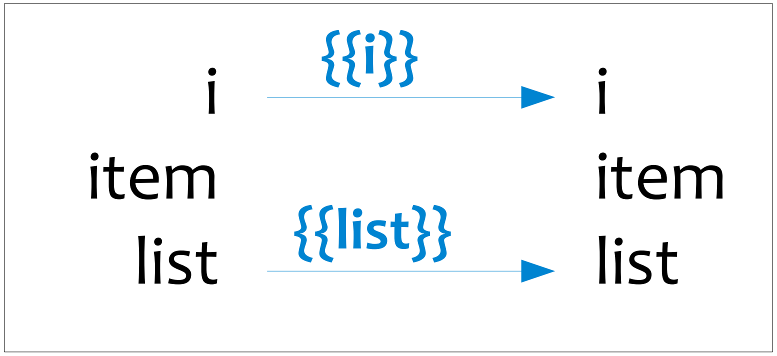
Strategy: annotate edges with the variables they use

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

$$\xrightarrow{\{W_1, \ldots, W_n\}} \;;\; \xrightarrow{\{V_1, \ldots, V_m\}} = \xrightarrow{\{W_1 \cup V_1,\, W_1 \cup V_2, \ldots, W_n \cup V_m\}}$$

Let *acl*(ACG) denote the *annotated closure* of ACG.

Key Theorem:

$$\mathbf{SCT}(\mathbf{restrict}(\texttt{ACG}, \texttt{V}))$$

iff

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

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 φ that captures the measured subset requirements
- $|\varphi| = O(acl|\mathbf{ACG}|)$
- φ can be made *dual-horn:* can find ψ that is
  - equisatisfiable to φ
  - conjunction of clauses,
  - each clause a disjunction of literals
  - at most one *negative* literal per clause
- min solutions to φ can be found from ψ 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 φ, we can find k elements of **All-Termination(SCT)(P)** in time **O(|*acl*(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%** ***(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%	(*note: same as* CCG+*SCT*)

Multiargument functions:

Proved terminating	1728	
With "nontrivial" cores	**90%**	
With multiple cores	7%	
Maximum core count	**3**	***(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