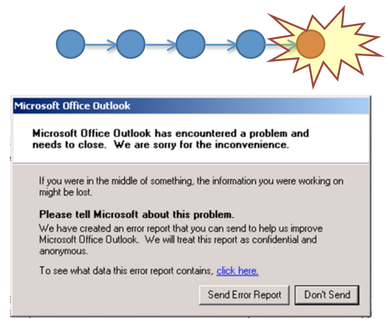
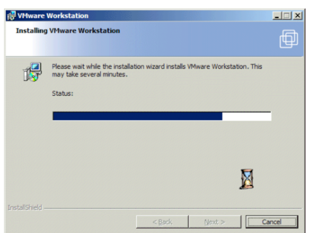
Proving Non-Termination

Ashutosh K. Gupta — MPI-SWS
Thomas A. Henzinger — EPFL
Rupak Majumdar — UCLA
Andrey Rybalchenko — MPI-SWS
Ru-Gang Xu — UCLA

Runtime Errors



Non-Termination Errors

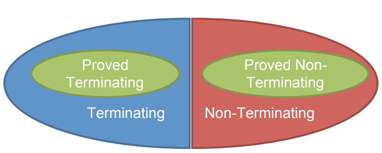


Proving Non-Termination

- Search for infinite executions
- Needs effective (finitary) representation

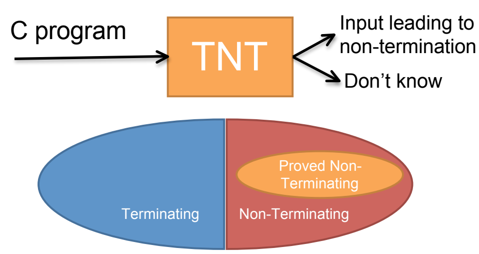
Failed Termination Proof vs. Non-Termination

- Successful termination provers:
  PolyRank, Terminator, ACL2, TerminWeb, AProVE, …
- Inherently incomplete algorithms
- Failed termination proof ≠ non-termination



TNT: Testing Non-Termination

Tool for proving non-termination



Outline

- Example:
  - Non-termination error in a memory protection system
- TNT algorithm:
  - Lasso search
  - Recurrent set computation

Example: Non-termination Error

- Input to TNT:
  - Mondriaan memory protection system
    - (early version courtesy: E. Witchel)
  - Uses recursion for basic operation
  - Termination required
- Output by TNT:
  - non-termination bug (now fixed) in `_mmpt_insert` procedure: cyclic sequences of calls to
    `_mmpt_insert ( … , 0, 3, … …, TAB_ROOT , 0, … )`

Non-Termination in `_mmpt_insert`

```
void _mmpt_insert (struct* mmpt, base, len, prot, tab_t* tab, level, … ) {
    if(len == 0) return;  // Exit condition
    int idx = make_idx(mmpt, base, level);
    if(level < 2 && … … … && len >= tab_len(mmpt, level + 1)) {
        … … …
    } else if(level < 2 && tab[idx] && !uentry_is_data(mmpt, tab[idx])) {
        _mmpt_insert (mmpt, base, len, prot, (tab_t*)tab[idx], level + 1, …);
    } else if(level < 2 && … … … ) {
        … … …
    } else {
        for(; len >= subblock_len(mmpt, level) && … … … ; … … … ) {
            … … …
        }
        _mmpt_insert (mmpt, base, len, prot, mmpt->tab, 0, …);
    }
}
```

Non-Termination in `_mmpt_insert` (first call)

```
void _mmpt_insert (struct* mmpt,    0,   3, prot, TAB_ROOT  ,     0, … ) {
    if(len == 0) return;  // Exit condition
    int idx = make_idx(mmpt, base, level);
    if(level < 2 && … … … && len >= tab_len(mmpt, level + 1)) {
        … … …
    } else if(level < 2 && tab[idx] && !uentry_is_data(mmpt, tab[idx])) {
        _mmpt_insert (mmpt, base, len, prot, (tab_t*)tab[idx], level + 1, …);
    } else if(level < 2 && … … … ) {
        … … …
    } else {
        for(; len >= subblock_len(mmpt, level) && … … … ; … … … ) {
            … … …
        }
        _mmpt_insert (mmpt, base, len, prot, mmpt->tab, 0, …);
    }
}
```

Non-Termination in `_mmpt_insert` (first call)

```
void _mmpt_insert (struct* mmpt,    0,   3, prot, TAB_ROOT  ,     0, … ) {
    if(  3 == 0) return;  // Exit condition
    int   0 = make_idx(mmpt,    0,     0);
    if(    0 < 2 && … … … &&   3 >= 4MB                    ) {
        … … …
    } else if(    0 < 2 && tab[ 0 ] && !uentry_is_data(mmpt, tab[ 0 ])) {
        _mmpt_insert (mmpt,    0,   3, prot, (tab_t*)tab[ 0 ],     0 + 1, …);
    } else if(level < 2 && … … … ) {
        … … …
    } else {
        for(; len >= subblock_len(mmpt, level) && … … … ; … … … ) {
            … … …
        }
        _mmpt_insert (mmpt, base, len, prot, mmpt->tab, 0, …);
    }
}
```

Non-Termination in `_mmpt_insert` (second call)

```
void _mmpt_insert (struct* mmpt,    0,   3, prot, TAB_MID  ,     1, … ) {
    if(len == 0) return;  // Exit condition
    int idx = make_idx(mmpt, base, level);
    if(level < 2 && … … … && len >= tab_len(mmpt, level + 1)) {
        … … …
    } else if(level < 2 && tab[idx] && !uentry_is_data(mmpt, tab[idx])) {
        _mmpt_insert (mmpt, base, len, prot, (tab_t*)tab[idx], level + 1, …);
    } else if(level < 2 && … … … ) {
        … … …
    } else {
        for(; len >= subblock_len(mmpt, level) && … … … ; … … … ) {
            … … …
        }
        _mmpt_insert (mmpt, base, len, prot, mmpt->tab, 0, …);
    }
}
```

Non-Termination in `_mmpt_insert` (second call)

```
void _mmpt_insert (struct* mmpt,    0,   3, prot, TAB_MID   ,     1, … ) {
    if(  3 == 0) return;  // Exit condition
    int   0 = make_idx(mmpt,    0,     1);
    if(    1 < 2 && … … … &&    3 >= 4KB                      ) {
        … … …
    } else if(    1 < 2 && tab[ 0 ] && !uentry_is_data(mmpt, tab[ 0 ])) {
        _mmpt_insert (mmpt,    0,   3, prot, (tab_t*)tab[ 0 ],     1 + 1, …);
    } else if(level < 2 && … … … ) {
        … … …
    } else {
        for(; len >= subblock_len(mmpt, level) && … … … ; … … … ) {
            … … …
        }
        _mmpt_insert (mmpt, base, len, prot, mmpt->tab, 0, …);
    }
}
```

Non-Termination in `_mmpt_insert` (third call)

```
void _mmpt_insert (struct* mmpt,    0,   3, prot, TAB_LEAF  ,     2, … ) {
    if(len == 0) return;  // Exit condition
    int idx = make_idx(mmpt, base, level);
    if(level < 2 && … … … && len >= tab_len(mmpt, level + 1)) {
        … … …
    } else if(level < 2 && tab[idx] && !uentry_is_data(mmpt, tab[idx])) {
        _mmpt_insert (mmpt, base, len, prot, (tab_t*)tab[idx], level + 1, …);
    } else if(level < 2 && … … … ) {
        … … …
    } else {
        for(; len >= subblock_len(mmpt, level) && … … … ; … … … ) {
            … … …
        }
        _mmpt_insert (mmpt, base, len, prot, mmpt->tab, 0, …);
    }
}
```

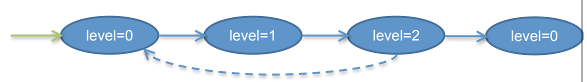
Non-Termination in `_mmpt_insert` (third call)

```
void _mmpt_insert (struct* mmpt,    0,   3, prot, TAB_LEAF  ,     2, … ) {
    if(  3 == 0) return;  // Exit condition
    int   0 = make_idx(mmpt,    0,     2);
    if(    2 < 2 && … … … &&    3 >= tab_len(mmpt,     2 + 1)) {
        … … …
    } else if(    2 < 2 && tab[idx] && !uentry_is_data(mmpt, tab[idx])) {
        _mmpt_insert (mmpt, base, len, prot, (tab_t*)tab[idx], level + 1, …);
    } else if(    2 < 2 && … … … ) {
        … … …
    } else {
        for(;   3 >= 4                            && … … … ; … … … ) {
            … … …
        }
        _mmpt_insert (mmpt,    0,   3, prot, TAB_ROOT , 0, …);
    }
}
```

Same Parameters as we started

What does TNT do?

- TNT finds a cyclic sequence of calls to **`_mmpt_insert`**
- The sequence is lasso – shaped



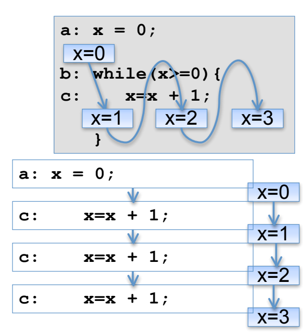
Recursive call sequence

- Non-termination is proved by analyzing the lasso
  - same valuation of input parameters after the call cycle

Outline:

- Search for lassos
- Prove a lasso is non-terminating through recurrent sets

Paths and Executions



Path = seq. of statements

Execution = seq. of states

What do infinite paths look like?

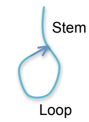
```
a: stmtA
   while( … ){
     if( … )
b:        stmtB
     else
c:        stmtC
   }
```

- General paths:
  e.g. stmtA stmtB^{2}stmtC^{3}stmtB^{5}stmtC^7…
- Periodic paths or Lassos:
  e.g. stmtA stmtB stmtC stmtB stmtC …

stmtA(stmtB stmtC)$^\omega$

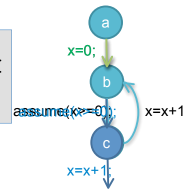
- TNT only considers periodic paths

Lassos



- Pair of paths: (stem, loop)
- Represents infinite periodic path: stem(loop)$^\omega$

```
a: x = 0;
b: while(x>=0){
c:    x=x + 1;
   }
```



- Lasso = (x=0; , assume(x>=0) ; x=x+1;)

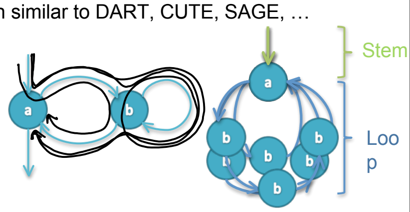
TNT Algorithm

- Two-step algorithm:
  1. Search for feasible lassos
     - Uses symbolic execution
     - Quickly find candidates for non-termination
  2. Check each lasso for non-termination
     - Uses SAT and constraint solving
     - Precise reasoning on small program fragments

Lasso Search

- Find lassos by symbolic execution
- Implementation similar to DART, CUTE, SAGE, …

```
… …
a: while( … ){
b:    while( … ){
        … …
      }
   }
…
```

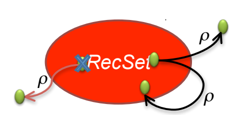


Outline:

- Lasso search
- Recurrent set computation

Recurrent Sets

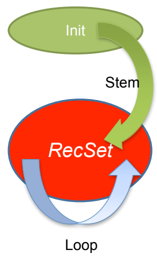
- Proves non-termination using inductive argument
- Set of states *RecSet* is recurrent for relation ρ(x, x') if:
  - non-empty
  - some successor of each state is in *RecSet*



Theorem:
ρ(x, x') is non-terminating iff there exists *RecSet*.

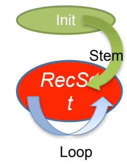
Recurrent Sets for Lassos

- *RecSet* is reachable via stem
- *RecSet* is recurrent for loop



Recurrent Sets to Constraints

- Lasso is non-terminating iff there exists *RecSet* such that:



$\exists x, x'\ RecSet(x') \wedge Stem(x,x')$ Non-empty

$\forall x \exists x'\ RecSet(x) \rightarrow Loop(x,x') \wedge RecSet(x')$ Looping

Computing Recurrent Sets

Bit-vectors:

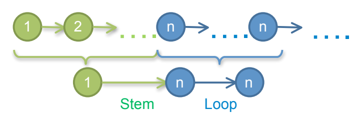
- Implementation level
- Bit-precise
- Full C expressions

Numbers:

- Algorithmic level (more abstract)
- Unbounded integers/rationals
- Linear arithmetic only

Recurrent Sets over Bit-vectors

- Some state has to appear infinitely often:



- Singleton *RecSet* is sufficient:

RecSet = {n}

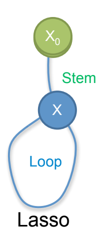
Non-Termination with Singleton Recurrent Sets

- Non-termination reduces to

$\exists X_0, X, X'\ Stem(X_0, X) \wedge Loop(X,X') \wedge (X=X')$

```
a: while (lo < hi) {
b:   mid = (lo + hi)/2;
c:   if ( NONDET ) {
d:     lo = mid + 1;
e:   }
f: }
```

Solve using SAT solver



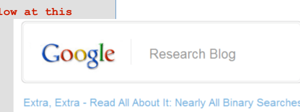
Stem { $true \wedge$

Loop { $lo<hi \wedge mid=(lo+hi)/2 \wedge$
$lo' = mid+1 \wedge$

(X=X') { $lo' = lo \wedge hi' = hi$

Broken Binary Search

```
int bsearch(int a[], int k,unsigned int lo,
                    unsigned int hi){
    unsigned int mid;
    while (lo < hi) {
        mid = (lo + hi)/2; //Overflow at this
  point
        if (a[mid] < k) {
            lo = mid + 1;
        } else if (a[mid] > k) {
            hi = mid - 1;
        } else {
            return mid;
        }
    }
    return -1;
}
```



- Joshua Bloch's blog presents a memory error
- Precondition (signed numbers):
  lo = 1, hi = MAXINT
- TNT discovers non-termination bug
- Precondition (unsigned numbers):
  lo= 1, hi = MAXINT, a[0]<k

Computing Recurrent Sets

Bit-vectors:
- Implementation level
- Bit-precise
- Full C expressions

Numbers:
- Algorithmic level (more abstract)
- Unbounded integers/rationals
- Linear arithmetic only

Recurrent Sets over Numbers

- Apply template-based technique from:
  - invariant generation (Colon et al. 2003)
  - abstract interpretation (Sankaranarayanan et al. 2006, Gulwani et al. 2008)
- *RecSet* is conjunction of linear inequalities

Template: $p_x\, x + p_y\, y =< p \;\wedge\; q_x\, x + q_y\, y =< q$

Parameters: p_x, p_y, p, q_x, q_y, q

Program Variables : x,y

Possible instantiations:

$1*x + 0*y =< 2 \;\wedge\; 6*x + 5*y =< 3$

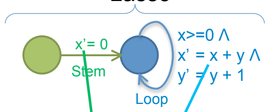
$2*x + 4*y =< 0 \;\wedge\; 6*x - 5*y =< 3$

$2*x + 4*y =< 0 \;\wedge\; 6*x - 5*y =< 3$

Example

```
x = 0;
while(x>=0)
{
    x=x + y;
    y=y + 1;
}
```

Lasso



RecSet template:
$p_x\, x + p_y\, y =< p \;\wedge\; q_x\, x + q_y\, y =< q$

$\exists x, x'\ RecSet(x') \wedge Stem(x,x')$

$\forall x \exists x'\ RecSet(x) \rightarrow Loop(x,x') \wedge RecSet(x')$

RecSet solution:
$-1*x + 0*y =< 0 \;\wedge\; 0*x - 1*y =< 0$

$0 =< x \wedge 0 =< y$

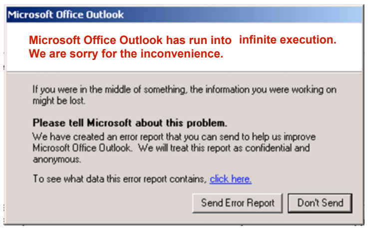
Hence Non-terminating

Non-termination Error in Mondriaan

- Input to TNT:
  - Mondriaan memory protection system (early version courtesy: E. Witchel)
  - Uses recursion for updates to permissions table
- Output by TNT:
  - non-termination bug (now fixed) in `_mmpt_insert` procedure: cyclic sequences of calls to

```
_mmpt_insert ( … , 0, 3, … …, TAB_ROOT , 0, … )
```

Eventually...



Detect and prove non-termination at runtime

Conclusion

- 2-step algorithm for proving non-termination
  - Lasso search
  - Recurrent set computation
- TNT
  - Symbolic execution
  - Constraint-based computation of recurrent sets