

Generating Abstract Graph-Based Procedure Summaries for Pointer Programs

Christina Jansen Thomas Noll

Software Modeling and Verification Group



<http://moves.rwth-aachen.de/>

ICGT 2014

22 July, 2014; York, UK

Pointer-Related Software Errors

Typical programming errors

- ▶ Dereferencing null (or disposed) pointers
- ▶ Creation of memory leaks
- ▶ Accidental invalidation of data structures

Pointer-Related Software Errors

Typical programming errors

- ▶ Dereferencing null (or disposed) pointers
- ▶ Creation of memory leaks
- ▶ Accidental invalidation of data structures

Problem: unbounded state spaces

- ▶ Infinite data domains
- ▶ Dynamic storage (de-)allocation
- ▶ Recursive procedures
- ▶ [Dynamic thread creation]

Pointer-Related Software Errors

Typical programming errors

- ▶ Dereferencing null (or disposed) pointers
- ▶ Creation of memory leaks
- ▶ Accidental invalidation of data structures

Problem: unbounded state spaces

- ▶ Infinite data domains
- ▶ Dynamic storage (de-)allocation
- ▶ Recursive procedures
- ▶ [Dynamic thread creation]

Abstraction techniques

- ▶ Automata-based symbolic verification (regular model checking)
- ▶ Logic-based symbolic verification (separation logic)
- ▶ Shape analysis
- ▶ **Heap abstraction grammars**

The Programming Language

Syntax

Programs	$Pgm ::= Prc^+$
Procedures	$Prc ::= p(x_1, \dots, x_k); \{S\}$ (with variables $x \in Var$)
Statements	$S \in Stm ::= l := v \mid \mathbf{new}(x) \mid p(l_1, \dots, l_k) \mid$ $\quad \mathbf{if}(C)\{S\} \mid \mathbf{while}(C)\{S\} \mid Stm; Stm$
Values	$Val ::= Loc \mid \mathbf{null}$
Locations	$l \in Loc ::= x \mid x.s$ (with selectors $s \in Sel$)
Conditions	$C \in Cnd ::= \dots$

The Programming Language

Syntax

Programs	$Pgm ::= Prc^+$
Procedures	$Prc ::= p(x_1, \dots, x_k); \{S\}$ (with variables $x \in Var$)
Statements	$S \in Stm ::= l := v \mid \mathbf{new}(x) \mid p(l_1, \dots, l_k) \mid$ $\quad \mathbf{if} (C)\{S\} \mid \mathbf{while} (C)\{S\} \mid Stm; Stm$
Values	$Val ::= Loc \mid \mathbf{null}$
Locations	$l \in Loc ::= x \mid x.s$ (with selectors $s \in Sel$)
Conditions	$C \in Cnd ::= \dots$

Example (doubly-linked list reversal)

```

main(head, tail){
  reverse(head, tail);
  tmp := head;
  head := tail;
  tail := tmp;
}

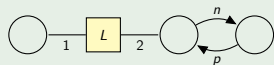
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}

```

Abstraction by HRGs in a Nutshell

- ▶ Represent heaps as (hyper-)graphs
- ▶ Abstraction: replace subgraphs by placeholders

Example (doubly-linked lists)



Definition (hypergraph)

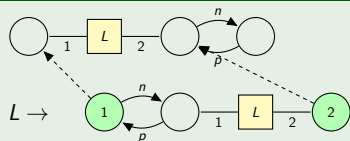
$H = (V, E, att, lab, ext) \in HG_{\Sigma}$

- ▶ Set of **nodes** V
- ▶ Set of **hyperedges** E
- ▶ **Attachment** $att : E \rightarrow V^*$
- ▶ **Labelling** $lab : E \rightarrow \Sigma$
 - ▶ $\Sigma = NT \cup Sel \cup Var$
- ▶ Sequence of (pairwise distinct) **external nodes** $ext \in V^*$

Abstraction by HRGs in a Nutshell

- ▶ Represent heaps as (hyper-)graphs
- ▶ Abstraction: replace subgraphs by placeholders

Example (doubly-linked lists)



Definition (HRG)

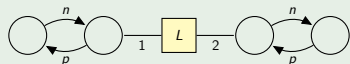
$$G \subseteq_{fin} NT \times HG_{\Sigma}$$

- ▶ **Production rules** $X \rightarrow H \in G$
- ▶ Ranking function $rk : \Sigma \rightarrow \mathbb{N}$
 - ▶ $rk(SEL) = \{2\}$
 - ▶ $rk(VAR) = \{1\}$
 - ▶ $|\text{ext}_H| = rk(X)$ for each $X \rightarrow H \in G$

Abstraction by HRGs in a Nutshell

- ▶ Represent heaps as (hyper-)graphs
- ▶ Abstraction: replace subgraphs by placeholders
- ▶ Hyperedge replacement grammars (HRGs): define the data structure represented by placeholder

Example (doubly-linked lists)



Definition (HRG)

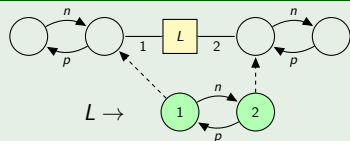
$$G \subseteq_{fin} NT \times HG_{\Sigma}$$

- ▶ **Production rules** $X \rightarrow H \in G$
- ▶ Ranking function $rk : \Sigma \rightarrow \mathbb{N}$
 - ▶ $rk(SEL) = \{2\}$
 - ▶ $rk(Var) = \{1\}$
 - ▶ $|ext_H| = rk(X)$ for each $X \rightarrow H \in G$

Abstraction by HRGs in a Nutshell

- ▶ Represent heaps as (hyper-)graphs
- ▶ Abstraction: replace subgraphs by placeholders
- ▶ Hyperedge replacement grammars (HRGs): define the data structure represented by placeholder

Example (doubly-linked lists)



Definition (HRG)

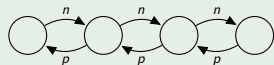
$$G \subseteq_{fin} NT \times HG_{\Sigma}$$

- ▶ **Production rules** $X \rightarrow H \in G$
- ▶ Ranking function $rk : \Sigma \rightarrow \mathbb{N}$
 - ▶ $rk(SEL) = \{2\}$
 - ▶ $rk(Var) = \{1\}$
 - ▶ $|ext_H| = rk(X)$ for each $X \rightarrow H \in G$

Abstraction by HRGs in a Nutshell

- ▶ Represent heaps as (hyper-)graphs
- ▶ Abstraction: replace subgraphs by placeholders
- ▶ Hyperedge replacement grammars (HRGs): define the data structure represented by placeholder

Example (doubly-linked lists)



Definition (HRG)

$$G \subseteq_{fin} NT \times HG_{\Sigma}$$

- ▶ **Production rules** $X \rightarrow H \in G$
- ▶ Ranking function $rk : \Sigma \rightarrow \mathbb{N}$
 - ▶ $rk(SEL) = \{2\}$
 - ▶ $rk(VAR) = \{1\}$
 - ▶ $|ext_H| = rk(X)$ for each $X \rightarrow H \in G$

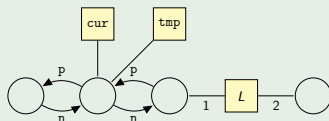
Execution & Concretisation

- ▶ Statements executed on concrete part of heap

Example

```
tmp := cur.next;
```

```
tmp := tmp.next;
```

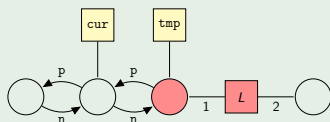


Execution & Concretisation

- ▶ Statements executed on concrete part of heap
- ▶ **Violation point** triggers concretisation

Example

```
tmp := cur.next;
tmp := tmp.next;
```

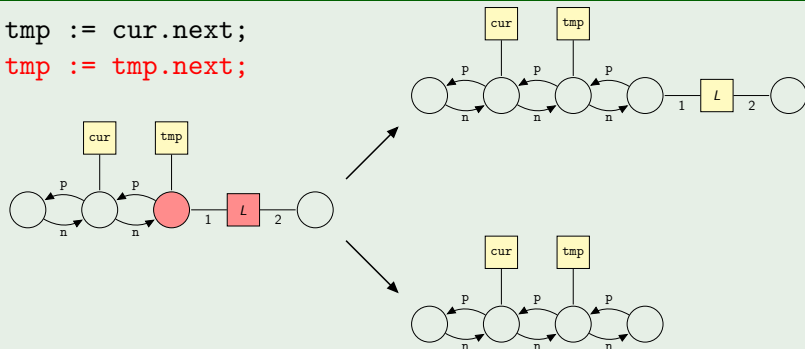


Execution & Concretisation

- ▶ Statements executed on concrete part of heap
- ▶ **Violation point** triggers concretisation
- ▶ **Concretisation** by (forward) rule application
- ▶ One configuration for each applicable rule
- ▶ Yields **nondeterministic branching** in state space

Example

```
tmp := cur.next;
tmp := tmp.next;
```

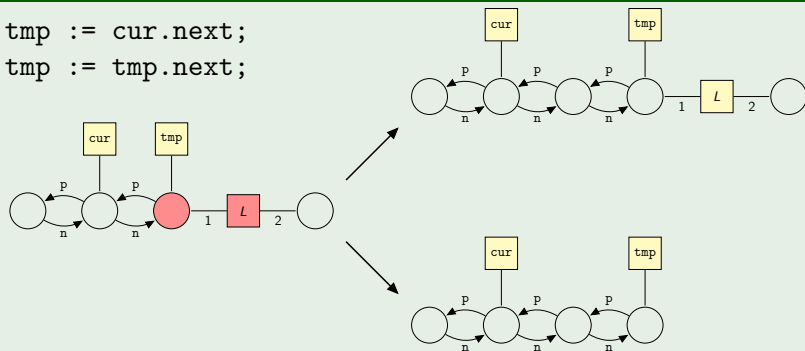


Execution & Concretisation

- ▶ Statements executed on concrete part of heap
- ▶ **Violation point** triggers concretisation
- ▶ **Concretisation** by (forward) rule application
- ▶ One configuration for each applicable rule
- ▶ Yields **nondeterministic branching** in state space

Example

```
tmp := cur.next;
tmp := tmp.next;
```



Taking Procedures into Account

Previous approach: HRG abstraction of runtime stack

- ▶ Encode runtime stack (incl. local variables) in heap
- ▶ Expand procedure bodies during execution
- + Fits nicely into HRG framework
- Requires manual development of (complex) abstraction rules in case of recursion

[Heinen/Jansen/Barthels FoVeOOS 2011]

Taking Procedures into Account

Previous approach: HRG abstraction of runtime stack

- ▶ Encode runtime stack (incl. local variables) in heap
- ▶ Expand procedure bodies during execution
- + Fits nicely into HRG framework
- Requires manual development of (complex) abstraction rules in case of recursion

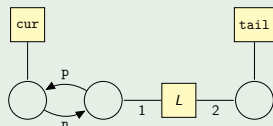
[Heinen/Jansen/Barthels FoVeOOS 2011]

Now: **procedure contracts**

- ▶ Map (reachable part of) input heap to set of output heaps
- ▶ Represent overall effect of procedure (“summary”)
- + Separates data and control abstraction
- + Fully automated computation
- + Supports cutpoints, i.e., node sharing between caller and callee

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)



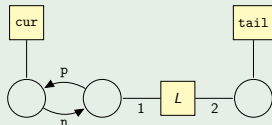
```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

+ concretisation

+ abstraction

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)



```

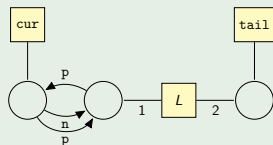
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
  
```

+ concretisation

+ abstraction

Procedure Contracts by Example

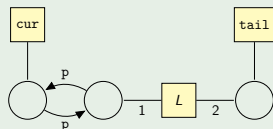
Example (reversal of list with ≥ 3 entries)



```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
+ concretisation
+ abstraction
```

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)



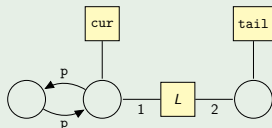
```

reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
+ concretisation
+ abstraction

```

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)



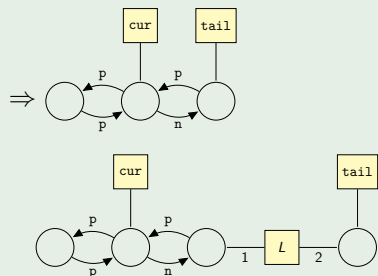
```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

+ concretisation

+ abstraction

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)

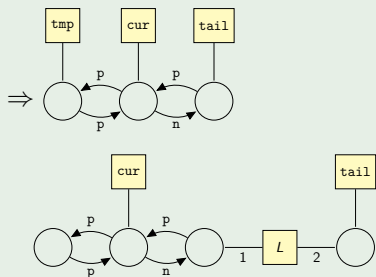


```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

+ concretisation
+ abstraction

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)

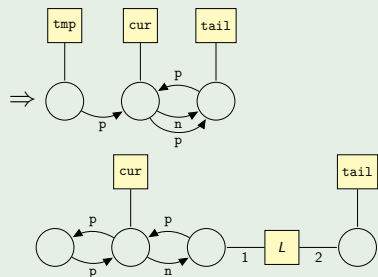


```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

+ concretisation
+ abstraction

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)

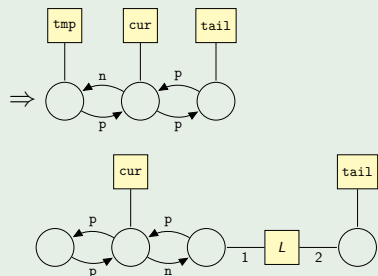


```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

+ concretisation
+ abstraction

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)

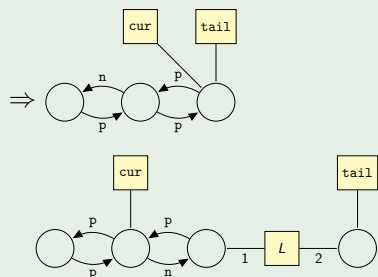


```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

+ concretisation
+ abstraction

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)



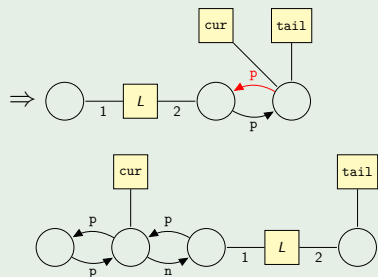
```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

+ concretisation

+ **abstraction**

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)



```

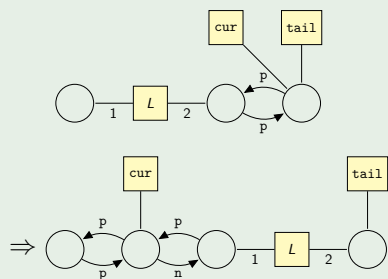
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}

```

+ concretisation
+ abstraction

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)

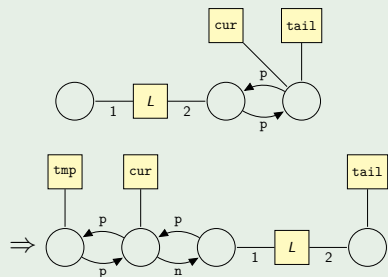


```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

+ concretisation
+ abstraction

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)

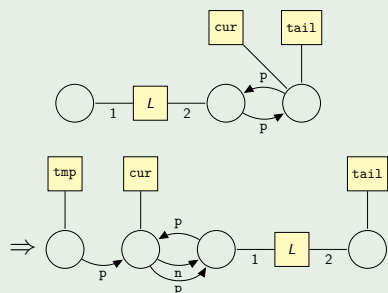


```

reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
+ concretisation
+ abstraction
  
```

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)

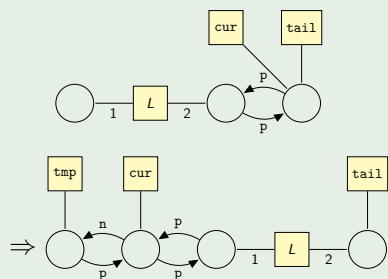


```

reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
+ concretisation
+ abstraction
  
```

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)

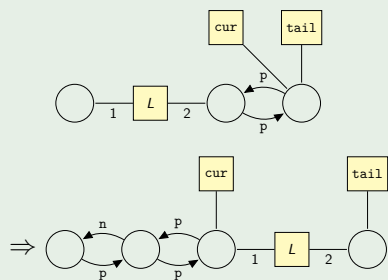


```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

+ concretisation
+ abstraction

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)

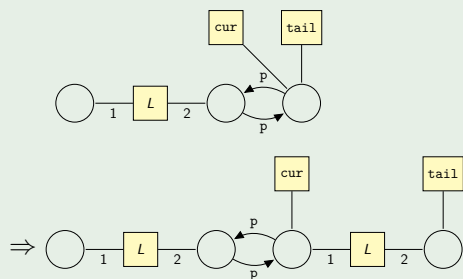


```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

+ concretisation
+ **abstraction**

Procedure Contracts by Example

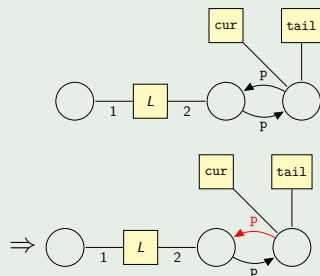
Example (reversal of list with ≥ 3 entries)



```
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
+ concretisation
+ abstraction
```

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)



```

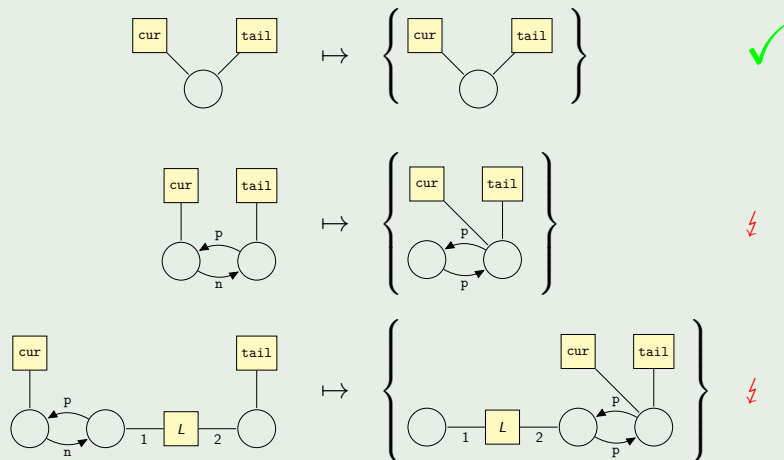
reverse(cur, tail){
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
+ concretisation
+ abstraction

```

Procedure Contracts by Example

Example (reversal of list with ≥ 3 entries)

(Excerpt of) resulting reverse-contract:



Interprocedural Dataflow Analysis

Dataflow analysis approach

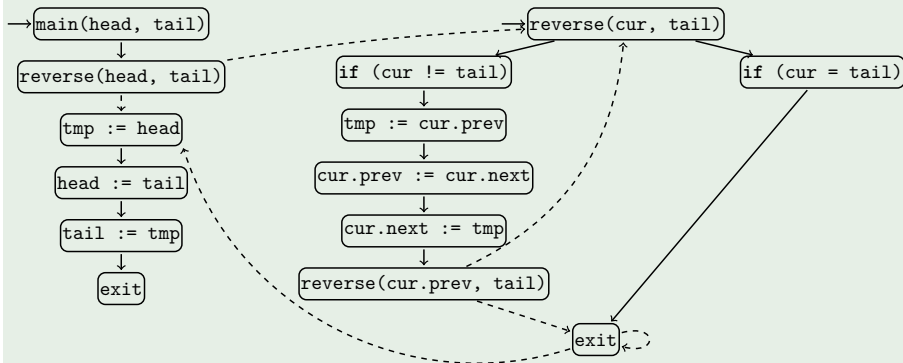
Compute **analysis information** as (least) **solution of equation system** over **control-flow graph** of program

Interprocedural Dataflow Analysis

Dataflow analysis approach

Compute **analysis information** as (least) **solution of equation system** over **control-flow graph** of program

Example (Interprocedural control-flow graph)



Analysis Information

Definition (Complete lattice)

A **complete lattice** is a partial order (D, \sqsubseteq) such that all subsets of D have least upper bounds. In this case, $\perp = \bigsqcup \emptyset$ denotes the **least element** of D .

Analysis Information

Definition (Complete lattice)

A **complete lattice** is a partial order (D, \sqsubseteq) such that all subsets of D have least upper bounds. In this case, $\perp = \bigsqcup \emptyset$ denotes the **least element** of D .

Example (procedure contracts)

$$\triangleright D = \underbrace{HG_{\Sigma}}_{\text{pre-heap}} \rightarrow \underbrace{2^{HG_{\Sigma}}}_{\text{post-heaps}}$$

contract

- $\triangleright d_1 \sqsubseteq d_2 \iff \forall H \in HG_{\Sigma} : d_1(H) \subseteq d_2(H)$
- \triangleright Least element $d_{\emptyset} = \{H \mapsto \emptyset \mid H \in HG_{\Sigma}\}$
- \triangleright Least upper bound $\bigsqcup C = \{H \mapsto \bigcup_{d \in C} d(H)\}$ for $C \subseteq D$

Local Transformers

- ▶ Formalise effect of **executing a single statement**
- ▶ For **non-procedural** control-flow nodes $n \in N$: $\llbracket n \rrbracket : D \rightarrow D$ with

$$\llbracket n \rrbracket(d)(H) = \text{abs}(t_n(\text{con}(d(H))))$$

where

$\text{con}: HG_\Sigma \rightarrow 2^{HG_\Sigma}$	concretisation
$t_n: HG_\Sigma \rightarrow 2^{HG_\Sigma} \cup \{\text{error}\}$	transfer function of n
$\text{abs}: HG_\Sigma \rightarrow HG_\Sigma$	abstraction

Local Transformers

- ▶ Formalise effect of **executing a single statement**
- ▶ For **non-procedural** control-flow nodes $n \in N$: $\llbracket n \rrbracket : D \rightarrow D$ with

$$\llbracket n \rrbracket(d)(H) = \text{abs}(t_n(\text{con}(d(H))))$$

where $\text{con}: HG_\Sigma \rightarrow 2^{HG_\Sigma}$

$t_n: HG_\Sigma \rightarrow 2^{HG_\Sigma} \cup \{\text{error}\}$

$\text{abs}: HG_\Sigma \rightarrow HG_\Sigma$

concretisation

transfer function of n

abstraction

Example (transfer functions)

▶ $t_n: \text{tmp} := \text{cur.next}$

$$\left(\begin{array}{c} \text{cur} \\ \downarrow \\ \text{Loop} \\ \uparrow \text{p} \\ \text{Loop} \\ \downarrow \text{n} \\ \text{Loop} \\ \downarrow 1 \\ \text{L} \\ \downarrow 2 \\ \text{tail} \end{array} \right) = \left\{ \begin{array}{c} \text{cur} \quad \text{tmp} \quad \text{tail} \\ \downarrow \quad \downarrow \quad \downarrow \\ \text{Loop} \\ \uparrow \text{p} \\ \text{Loop} \\ \downarrow \text{n} \\ \text{Loop} \\ \downarrow 1 \\ \text{L} \\ \downarrow 2 \\ \text{tail} \end{array} \right\}$$

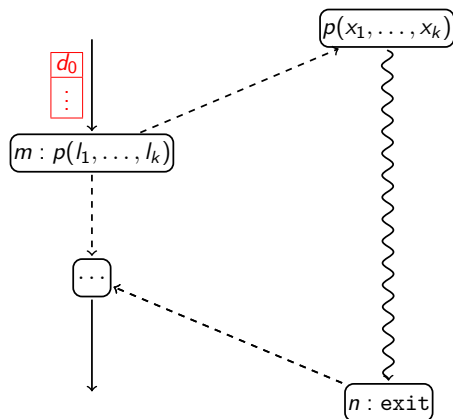
▶ $t_n: \text{if}(C)(H) = \begin{cases} \{H\} & \text{if } H \text{ (possibly) satisfies } C \\ \emptyset & \text{otherwise} \end{cases}$

Taking Procedures into Account

- ▶ Extend analysis information to (non-empty) **stacks of contracts**: D^+
- ▶ **Non-procedural statements** operate on topmost entry

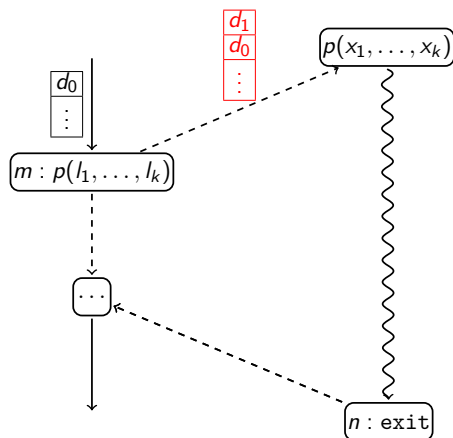
Taking Procedures into Account

- ▶ Extend analysis information to (non-empty) **stacks of contracts**: D^+
- ▶ **Non-procedural statements** operate on topmost entry
- ▶ Handling of **procedure calls**:



Taking Procedures into Account

- ▶ Extend analysis information to (non-empty) **stacks of contracts**: D^+
- ▶ **Non-procedural statements** operate on topmost entry
- ▶ Handling of **procedure calls**:

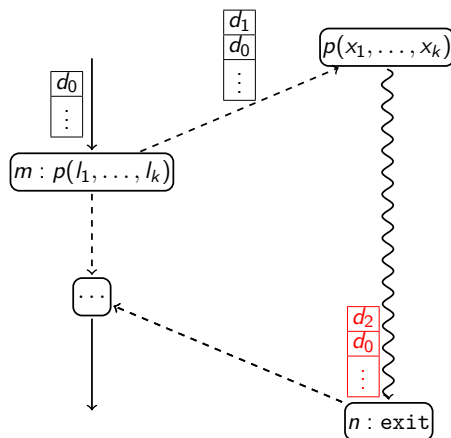


1. **push new contract**
 $d_1 = \text{call}_m(d_0)$

- ▶ cut d_0 -pre-heaps to part reachable from l_1, \dots, l_k
- ▶ insert edges for x_1, \dots, x_k
- ▶ remove other variables

Taking Procedures into Account

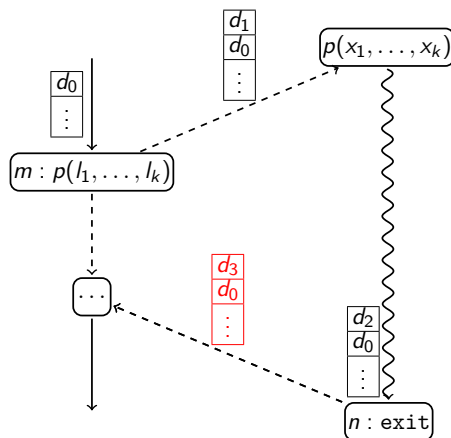
- ▶ Extend analysis information to (non-empty) **stacks of contracts**: D^+
- ▶ **Non-procedural statements** operate on topmost entry
- ▶ Handling of **procedure calls**:



1. push new contract
 $d_1 = \text{call}_m(d_0)$
2. obtain d_2 by applying global transformer $\llbracket n \rrbracket(d_1)$
(see next slide)

Taking Procedures into Account

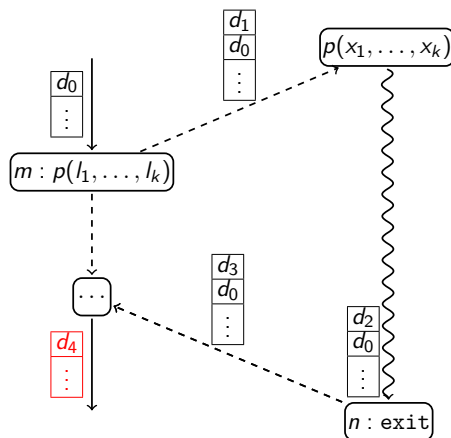
- ▶ Extend analysis information to (non-empty) **stacks of contracts**: D^+
- ▶ **Non-procedural statements** operate on topmost entry
- ▶ Handling of **procedure calls**:



1. push new contract
 $d_1 = \text{call}_m(d_0)$
2. obtain d_2 by applying global transformer $\llbracket n \rrbracket(d_1)$
3. obtain d_3 from d_2 by removing p -local variables

Taking Procedures into Account

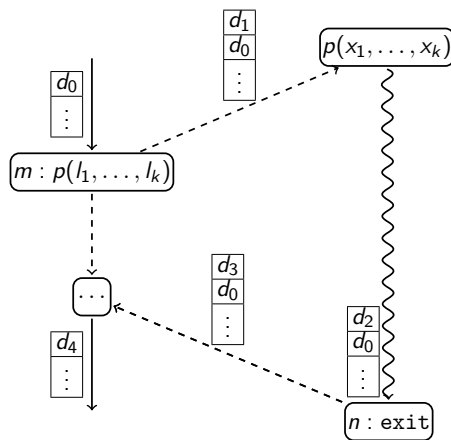
- ▶ Extend analysis information to (non-empty) **stacks of contracts**: D^+
- ▶ **Non-procedural statements** operate on topmost entry
- ▶ Handling of **procedure calls**:



1. push new contract
 $d_1 = \text{call}_m(d_0)$
2. obtain d_2 by applying global transformer $\llbracket n \rrbracket(d_1)$
3. obtain d_3 from d_2 by removing p -local variables
4. obtain d_4 from d_0 by replacing subheaps according to d_3

Taking Procedures into Account

- ▶ Extend analysis information to (non-empty) **stacks of contracts**: D^+
- ▶ **Non-procedural statements** operate on topmost entry
- ▶ Handling of **procedure calls**:



1. push new contract
 $d_1 = \text{call}_m(d_0)$
2. obtain d_2 by applying global transformer $\llbracket n \rrbracket(d_1)$
3. obtain d_3 from d_2 by removing p -local variables
4. obtain d_4 from d_0 by replacing subheaps according to d_3

Yields **local transformer**

$$\llbracket \cdot \rrbracket^* : N \rightarrow (D^+ \rightarrow D^+)$$

Global Transformers I

The **global transformer**

$$\llbracket \cdot \rrbracket : N \rightarrow (D^+ \rightarrow D^+)$$

captures the overall effect of executing a procedure:

$$\llbracket n \rrbracket(ds) = \begin{cases} ds & \text{if } n \text{ procedure entry} \\ \llbracket n \rrbracket (\sqcup_{m \rightarrow n} \llbracket m \rrbracket(ds)) & \text{otherwise} \end{cases}$$

Global Transformers II

Example (main procedure)

Procedure	Result of global transformer
-----------	------------------------------

```
main(head, tail){
```

```
    reverse(head,  
            tail);
```

```
    tmp := head;
```

```
    head := tail;
```

```
    tail := head;}
```

Global Transformers II

Example (main procedure)

Procedure

```
main(head, tail){
```

```
  reverse(head,  
          tail);
```

```
  tmp := head;
```

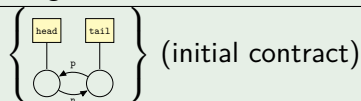
```
  head := tail;
```

```
  tail := head;}
```

Result of global transformer



\mapsto



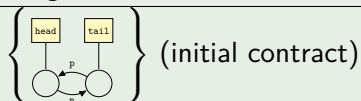
Global Transformers II

Example (main procedure)

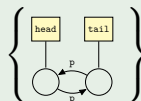
Procedure

Result of global transformer

```
main(head, tail){
```


 \mapsto


```
reverse(head,  
tail);
```


 \mapsto


```
tmp := head;
```

```
head := tail;
```

```
tail := head;}
```

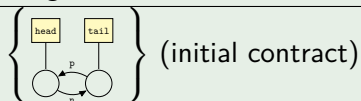
Global Transformers II

Example (main procedure)

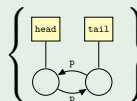
Procedure

Result of global transformer

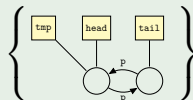
```
main(head, tail){
```


 \mapsto


```
reverse(head,  
tail);
```


 \mapsto


```
tmp := head;
```


 \mapsto


```
head := tail;
```

```
tail := head;}
```

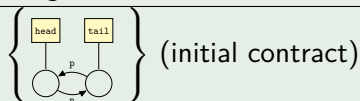
Global Transformers II

Example (main procedure)

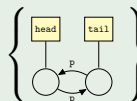
Procedure

Result of global transformer

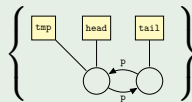
```
main(head, tail){
```


 \mapsto


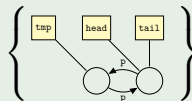
```
reverse(head,  
tail);
```


 \mapsto


```
tmp := head;
```


 \mapsto


```
head := tail;
```


 \mapsto


```
tail := head;}
```

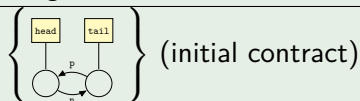
Global Transformers II

Example (main procedure)

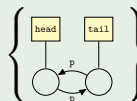
Procedure

Result of global transformer

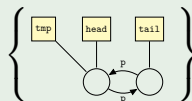
```
main(head, tail){
```


 \mapsto


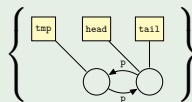
```
reverse(head,  
tail);
```


 \mapsto


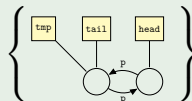
```
tmp := head;
```


 \mapsto


```
head := tail;
```


 \mapsto


```
tail := head;}
```


 \mapsto


The Equation System

Interprocedural dataflow equation system

- ▶ Information at **entry** of node $n \in N$:

$$pre(n) = \begin{cases} \{init \mapsto \{init\}\} & \text{if } n : \text{main}(\dots)\{\dots\} \\ \sqcup_{m \text{ calls } p} call_m(pre(m)) & \text{if } n : p(\dots)\{\dots\}, p \neq \text{main} \\ \sqcup_{m \rightarrow n} post(m) & \text{otherwise} \end{cases}$$

- ▶ Information at **exit** of node $n \in N$:

$$post(n) = \llbracket n \rrbracket^*(pre(n))$$

The Equation System

Interprocedural dataflow equation system

- Information at **entry** of node $n \in N$:

$$pre(n) = \begin{cases} \{init \mapsto \{init\}\} & \text{if } n : \text{main}(\dots)\{\dots\} \\ \bigsqcup_{m \text{ calls } p} call_m(pre(m)) & \text{if } n : p(\dots)\{\dots\}, p \neq \text{main} \\ \bigsqcup_{m \rightarrow n} post(m) & \text{otherwise} \end{cases}$$

- Information at **exit** of node $n \in N$:

$$post(n) = \llbracket n \rrbracket^*(pre(n))$$

Facts:

- $\llbracket n \rrbracket^*$ **monotonic** w.r.t. \sqsubseteq^+
 - (D^+, \sqsubseteq^+) satisfies **ascending chain condition** if HRG yields finite abstraction
- \Rightarrow solvable by **demand-driven fixpoint iteration** over stacks of height 2
[Knoop/Steffen CC 1992]

Fixing the Bug

Example (corrected code)

```
main(head, tail){
  reverse(head, tail);
  tmp := head;
  head := tail;
  tail := tmp;
}

reverse(cur, tail){
  if (cur != null){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev,
            tail);
  }
}
```

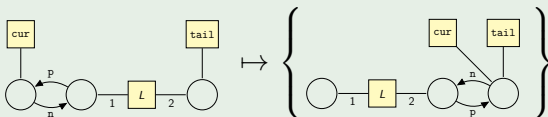
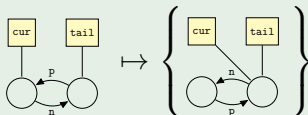
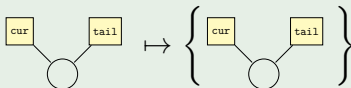
Fixing the Bug

Example (corrected code)

```
main(head, tail){
  reverse(head, tail);
  tmp := head;
  head := tail;
  tail := tmp;
}
```

```
reverse(cur, tail){
  if (cur != null){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev,
            tail);
  }
}
```

reverse contract (excerpt):



Conclusion

Summary

- ▶ Novel interprocedural dataflow analysis for automatically deriving **procedure contracts** of pointer programs
- ▶ Builds upon **abstraction framework** using HRGs
- ▶ Supports **recursion**, **local variables** and **cutpoints**

Conclusion

Summary

- ▶ Novel interprocedural dataflow analysis for automatically deriving **procedure contracts** of pointer programs
- ▶ Builds upon **abstraction framework** using HRGs
- ▶ Supports **recursion**, **local variables** and **cutpoints**

Outlook

- ▶ **Implementation** for evaluation (esp. comparison with explicit encoding of runtime stack)
- ▶ Extension to concurrent programs by **thread contracts using permissions**
- ▶ Use HRG fragment with **decidable language inclusion problem** (cycle check)