

Verifying Dynamic Pointer-Manipulating Threads

Thomas Noll Stefan Rieger

MOVES: Software Modeling and Verification
RWTH Aachen University, Germany

4.3.2008



Duisburg

Multithreading and Pointer Structures

Problem

- unbounded creation of **threads** at runtime
- unbounded creation and destruction of **objects** at runtime
- destructive updates via pointers
- possibly **infinite state space**

Multithreading and Pointer Structures

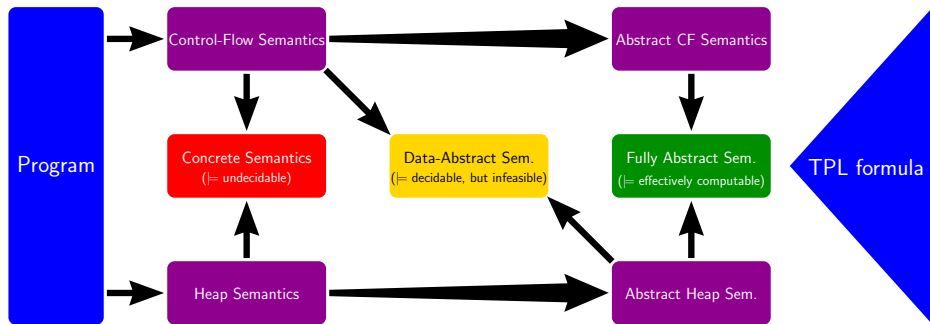
Problem

- unbounded creation of **threads** at runtime
- unbounded creation and destruction of **objects** at runtime
- destructive updates via pointers
- possibly **infinite state space**

Approach

- simple programming language
- temporal pointer logic
- **heap** abstraction
- **control-flow** abstraction
- here: at most **one outgoing edge** (list-like structures)
(extension to arbitrary data-structures in the works)

Abstraction Scheme



Concurrent Server/Worker System

main

```
var x, y;  
proc main(  
01  new(x);  
02  spawn(server);  
)
```

Concurrent Server/Worker System

main

```
var  $x, y$ ;  
proc main(  
01 new( $x$ );  
02 spawn(server);  
)
```

server

```
server(  
11 spawn(worker);  
12 atc(tt);  
13    $y := x$ ;  
14   new( $x$ );  
15    $*x := y$ ;  
16 end atc;  
17 goto 11;  
)
```

Concurrent Server/Worker System

main

```
var  $x, y$ ;  
proc main(  
01 new( $x$ );  
02 spawn(server);  
)
```

server

```
server(  
11 spawn(worker);  
12 atc(tt);  
13    $y := x$ ;  
14   new( $x$ );  
15    $*x := y$ ;  
16 end atc;  
17 goto 11;  
)
```

worker

```
worker(  
21 atc( $x \neq nil$ );  
22    $y := x$ ;  
23    $x := *x$ ;  
24   del( $y$ );  
25 end atc;  
)
```

Concurrent Server/Worker System

main

```

var x, y;
proc main(
01  new(x);
02  spawn(server);
)

```

server

```

server(
11  spawn(worker);
12  atc(tt);
13      y := x;
14      new(x);
15      *x := y;
16  end atc;
17  goto 11;
)

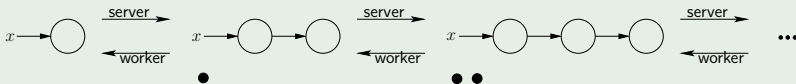
```

worker

```

worker(
21  atc(x ≠ nil);
22      y := x;
23      x := *x;
24      del(y);
25  end atc;
)

```



Overview

- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model
- ④ Model Checking
- ⑤ Conclusion

Programming Language

DLM-Program

For program variables $v_i, v \in PV$ and thread names $p_j, p \in \mathcal{P}$:

$$\pi = \mathbf{var} \ v_1, \dots, v_k; \ \mathbf{proc} \ \mathbf{main}(s_0); \ p_1(s_1); \dots; \ p_l(s_l)$$

Programming Language

DLM-Program

For program variables $v_i, v \in PV$ and thread names $p_j, p \in \mathcal{P}$:

$$\pi = \mathbf{var} \ v_1, \dots, v_k; \ \mathbf{proc} \ \mathbf{main}(s_0); \ p_1(s_1); \dots; \ p_l(s_l)$$

Statements ($s_i = s_{i1}; \dots; s_{ir_i}$)

PExp := PExp	atc (BExp)	spawn (p)	if BExp goto n
new (PExp)	end atc	exit	goto n
del (PExp)			

Programming Language

DLM-Program

For program variables $v_i, v \in PV$ and thread names $p_j, p \in \mathcal{P}$:

$$\pi = \mathbf{var} \ v_1, \dots, v_k; \ \mathbf{proc} \ \mathbf{main}(s_0); \ p_1(s_1); \dots; \ p_l(s_l)$$

Statements ($s_i = s_{i1}; \dots; s_{ir_i}$)

PExp := PExp	atc (BExp)	spawn (p)	if BExp goto n
new (PExp)	end atc	exit	goto n
del (PExp)			

Pointer Expressions

$$\text{PExp} ::= \mathit{nil} \mid v \mid *v \mid \&v$$

A Heap Logic

Pointer Logic (PL)

$$\text{NExp} ::= \text{nil} \mid v (\in PV) \mid x (\in LV) \mid * \text{NExp}$$

A Heap Logic

Pointer Logic (PL)

$\text{NExp} ::= nil \mid v (\in PV) \mid x (\in LV) \mid * \text{NExp}$
 $\text{Atomic} ::= tt \mid ff \mid \text{new}_n \mid \text{del} \mid \text{err} \mid \text{leak} \mid \text{spawn}_p$
 $\mid \text{NExp} = \text{NExp} \mid \text{NExp} \rightsquigarrow \text{NExp}$

A Heap Logic

Pointer Logic (PL)

$\text{NExp} ::= nil \mid v (\in PV) \mid x (\in LV) \mid * \text{NExp}$
 $\text{Atomic} ::= tt \mid ff \mid \text{new}_n \mid \text{del} \mid \text{err} \mid \text{leak} \mid \text{spawn}_p$
 $\quad \mid \text{NExp} = \text{NExp} \mid \text{NExp} \rightsquigarrow \text{NExp}$
 $\text{PL} ::= \text{Atomic} \mid \neg \text{PL} \mid \text{PL} \wedge \text{PL} \mid \exists x : \text{PL}$

A Heap Logic

Pointer Logic (PL)

$$\begin{aligned}
 \text{NExp} &::= \text{nil} \mid v (\in PV) \mid x (\in LV) \mid * \text{NExp} \\
 \text{Atomic} &::= \text{tt} \mid \text{ff} \mid \text{new}_n \mid \text{del} \mid \text{err} \mid \text{leak} \mid \text{spawn}_p \\
 &\quad \mid \text{NExp} = \text{NExp} \mid \text{NExp} \rightsquigarrow \text{NExp} \\
 \text{PL} &::= \text{Atomic} \mid \neg \text{PL} \mid \text{PL} \wedge \text{PL} \mid \exists x : \text{PL}
 \end{aligned}$$

Temporal Pointer Logic (TPL, finite traces allowed)

- PL + LTL temporal operators
 - no temporal operators within quantifiers
- ⇒ binding of logical variables across states not necessary
- ⇒ reduction to standard LTL model checking (with finite traces)

main

```

var x, y;
proc main(
01  new(x);
02  spawn(server); )

```

server

```

server(
11  spawn(worker);
12  atc(tt);
13    y := x;
14    new(x);
15    *x := y;
16  end atc;
17  goto 11; )

```

worker

```

worker(
21  atc(x ≠ nil);
22    y := x;
23    x := *x;
24    del(y);
25  end atc; )

```

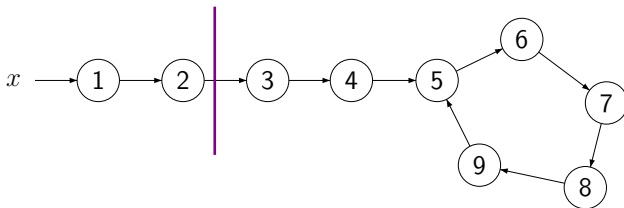
Properties

1. $(\mathbf{GX} \text{ tt}) \wedge (\neg \mathbf{F} \text{ err})$
2. $\mathbf{GF} \exists n : \text{new}_n$
3. $\mathbf{GF} \text{spawn}_{\text{worker}}$
4. $\mathbf{G}(\exists n : \text{new}_n \rightarrow \mathbf{F} \text{spawn}_{\text{worker}})$
5. $\neg \mathbf{G}(\text{spawn}_{\text{worker}} \rightarrow \mathbf{F} \text{del})$

Overview

- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model
- ④ Model Checking
- ⑤ Conclusion

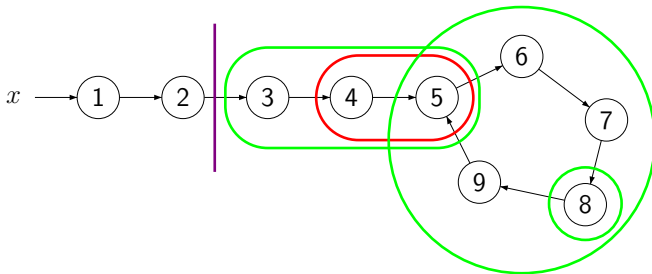
Idea



Approach

- Merge nodes along **chains** into a single **summary node**.

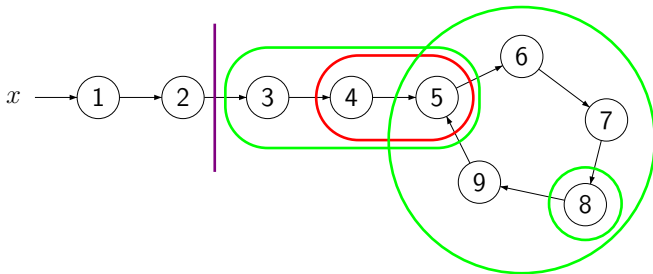
Idea



Approach

- Merge nodes along **chains** into a single **summary node**.

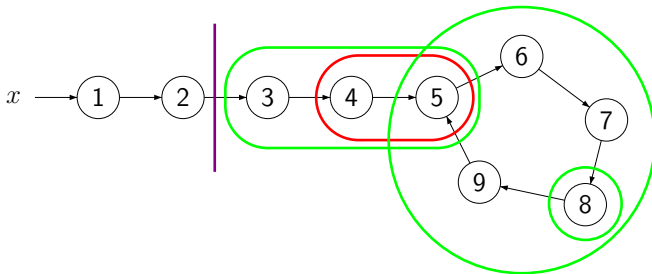
Idea



Approach

- Merge nodes along **chains** into a single **summary node**.
- Do only abstract from chains with $> M$ nodes.

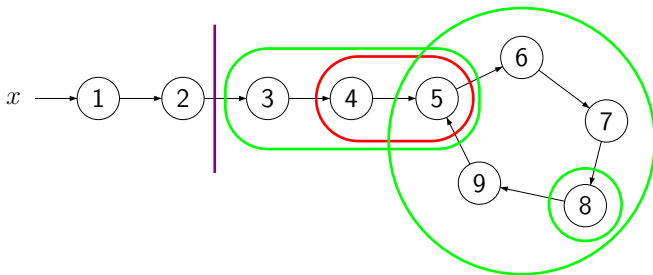
Idea



Approach

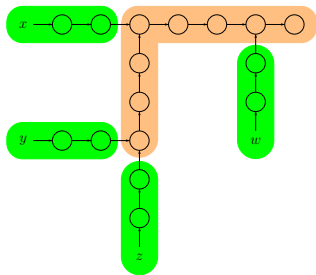
- Merge nodes along **chains** into a single **summary node**.
- Do only abstract from chains with $> M$ nodes.
- Do not merge nodes that are **close** to program variables.

Idea

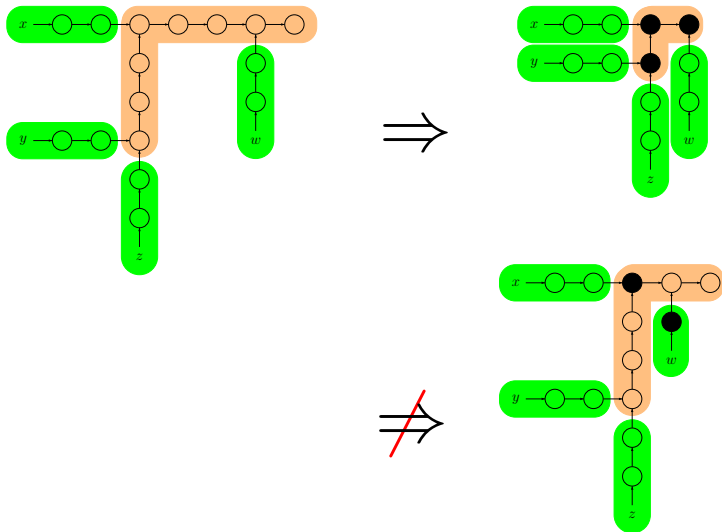


Approach

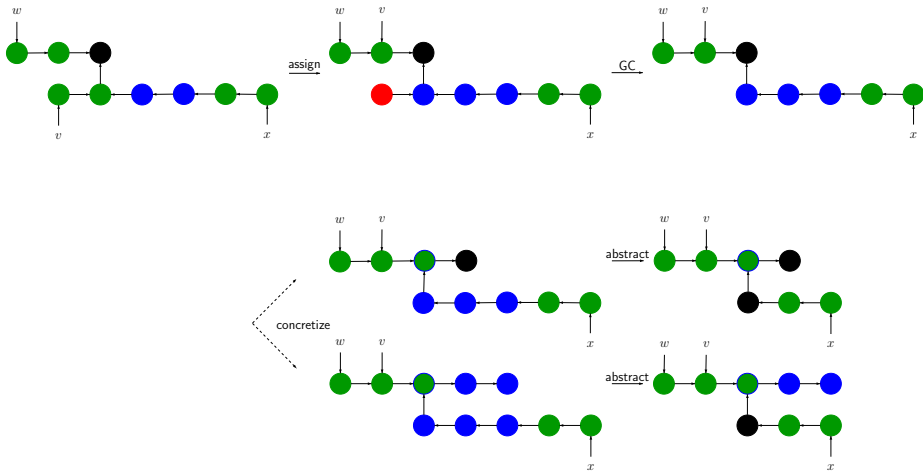
- Merge nodes along **chains** into a single **summary node**.
- Do only abstract from chains with $> M$ nodes.
- Do not merge nodes that are **close** to program variables.
- Automatic **garbage collection** (finiteness) (\Rightarrow **leak-flag**).

Example: $M = 1$ 

Example: $M = 1$



Example: Assignment $v := *w$ ($M = 2$)



Overview

- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model**
- ④ Model Checking
- ⑤ Conclusion

Idea

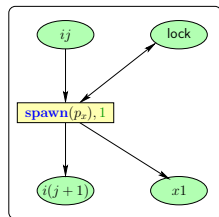
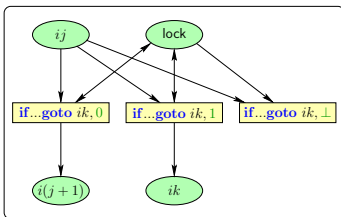
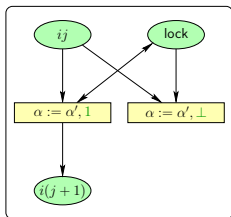
Modeling Control-Flow by Petri net \mathfrak{P}^c

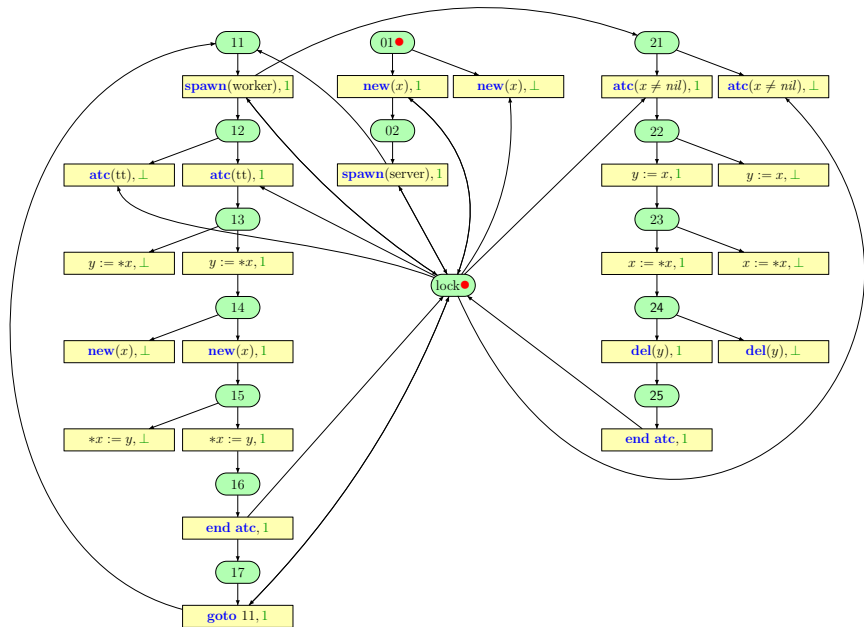
- Every control location corresponds to a place
- Transitions are labeled by instruction and flag $\in \{0, 1, \perp\}$
- Place **lock** for synchronization

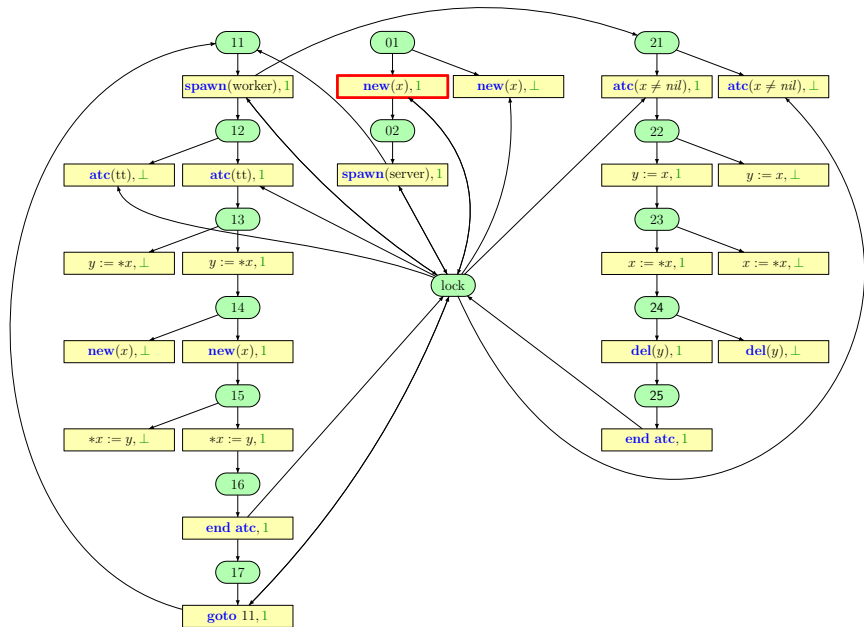
Idea

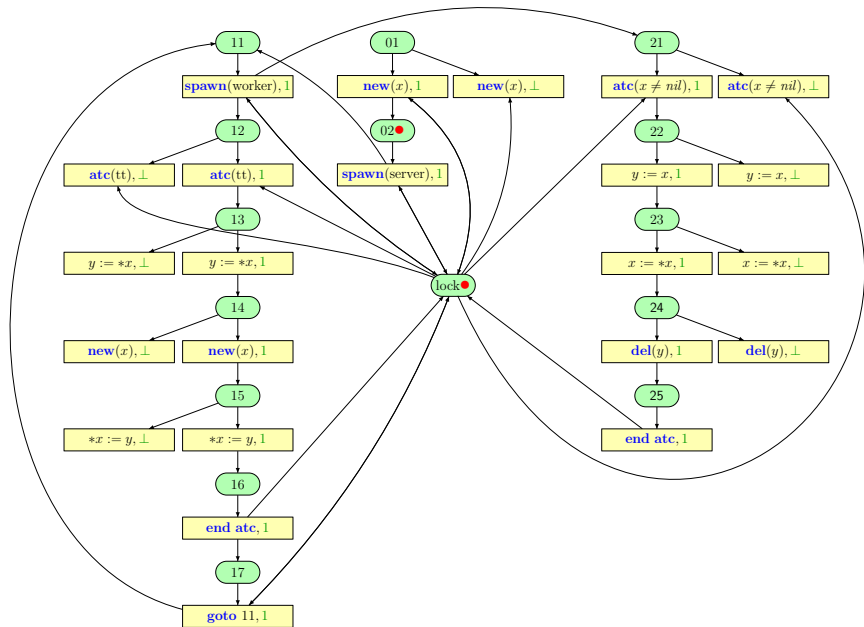
Modeling Control-Flow by Petri net \mathfrak{P}^c

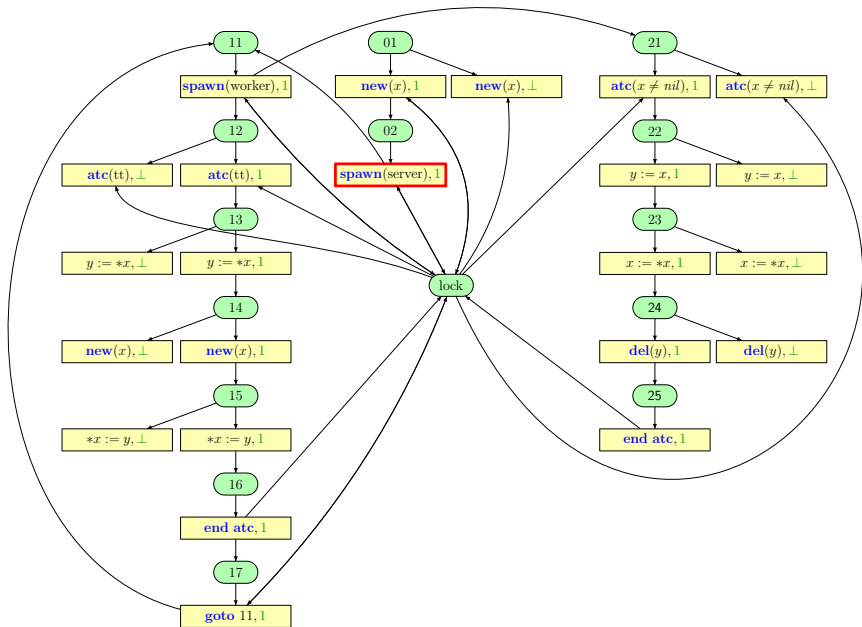
- Every control location corresponds to a place
- Transitions are labeled by instruction and flag $\in \{0, 1, \perp\}$
- Place **lock** for synchronization

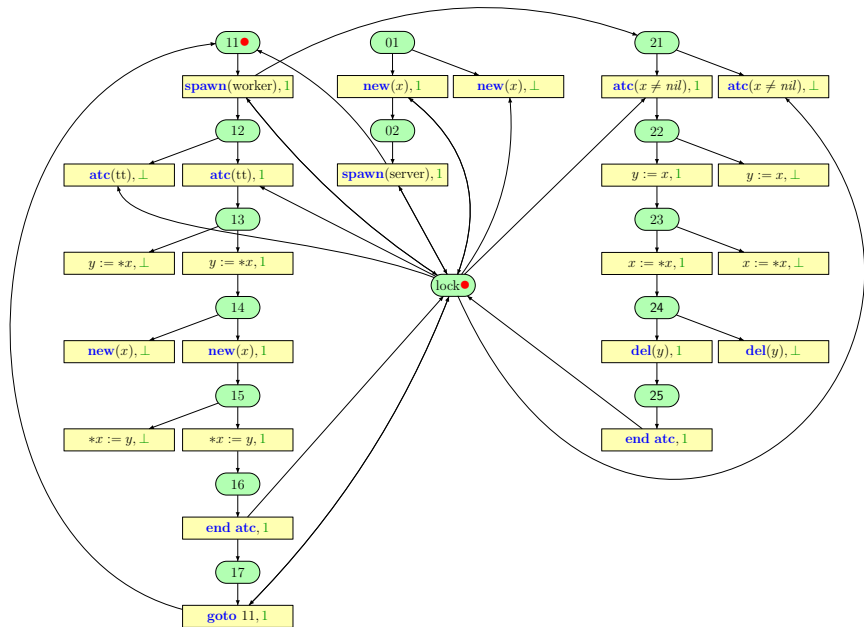


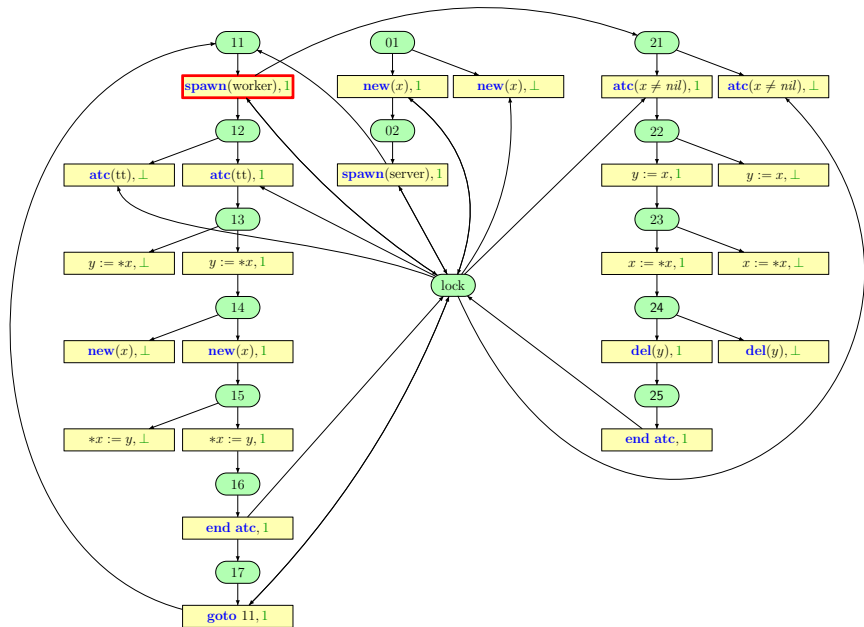


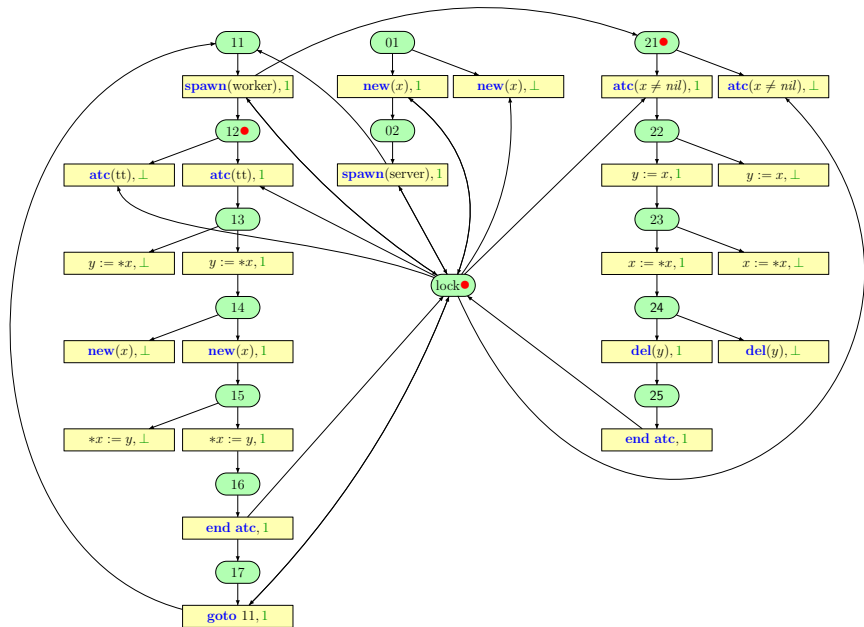


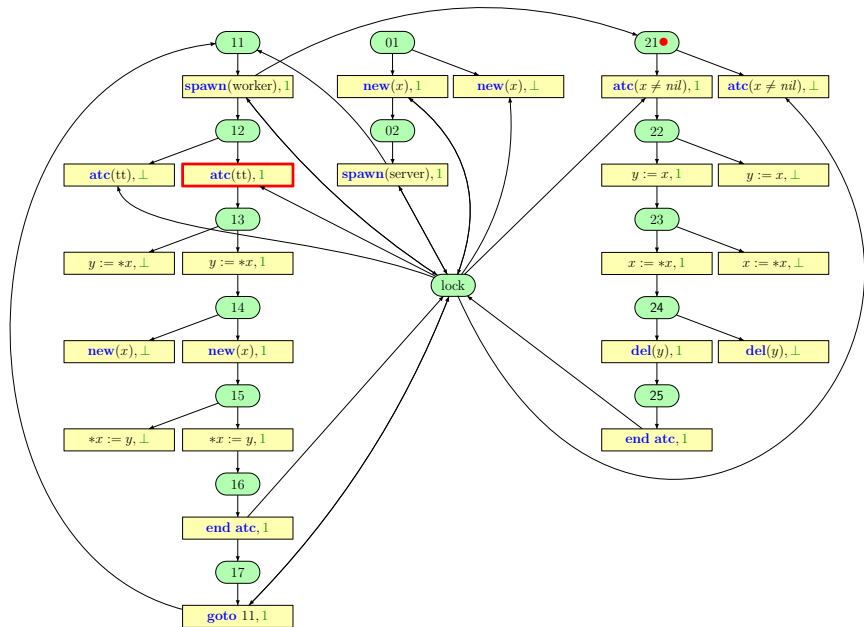


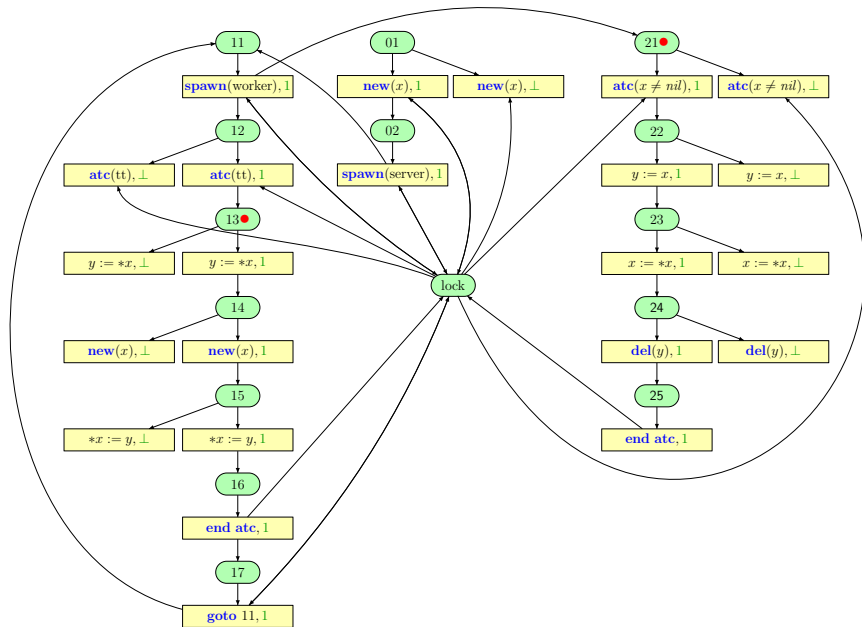


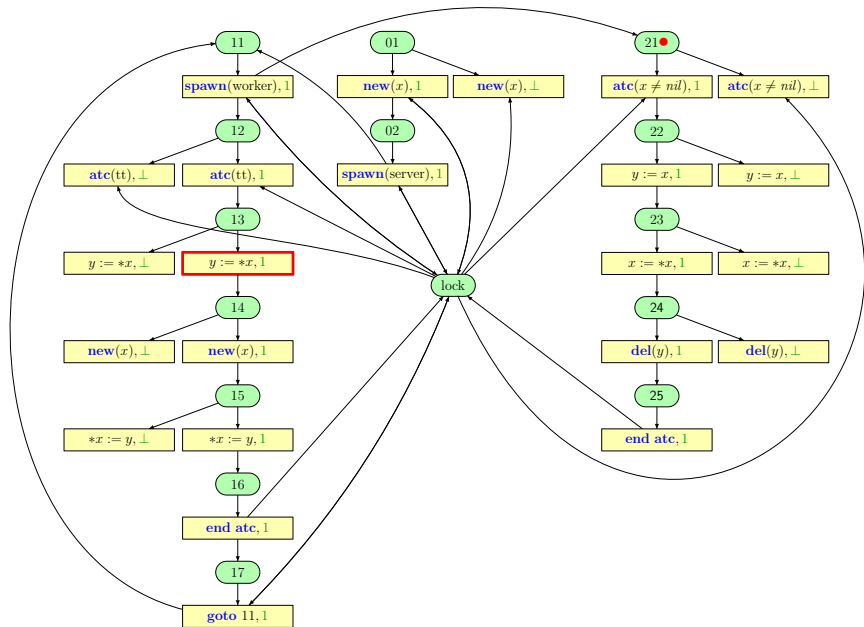


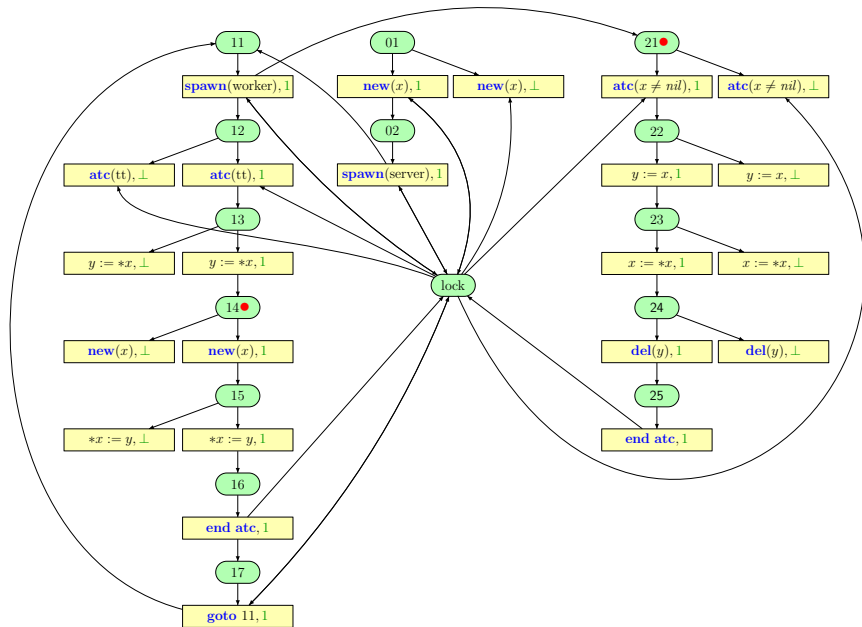


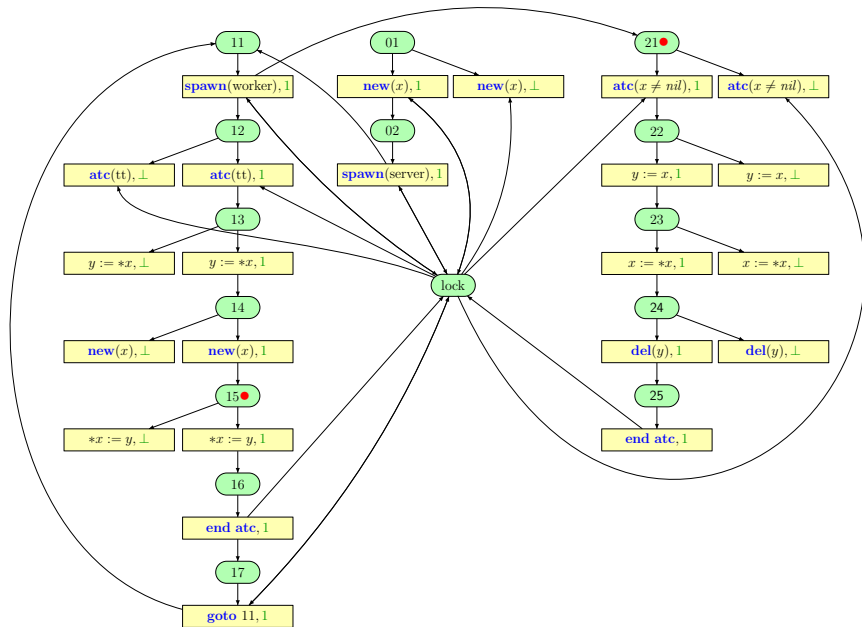


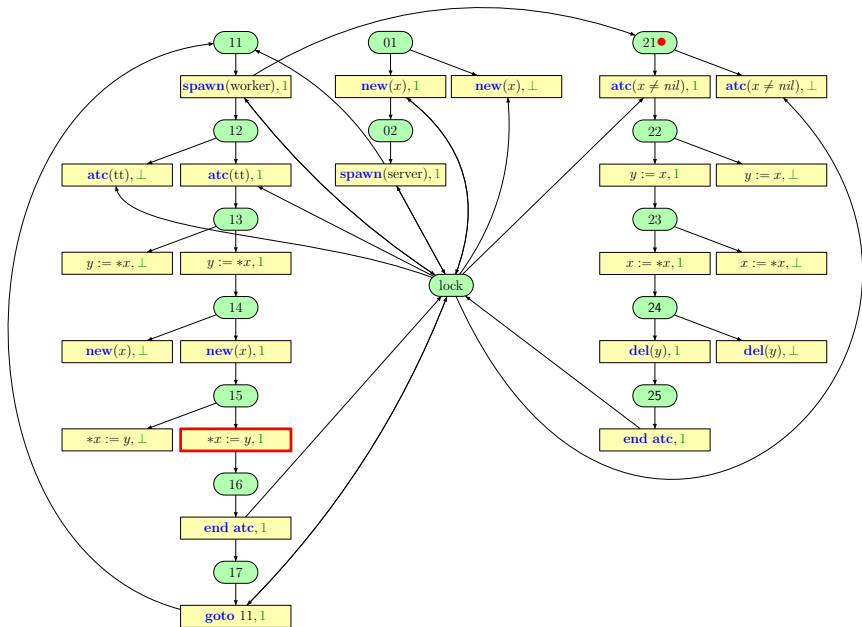


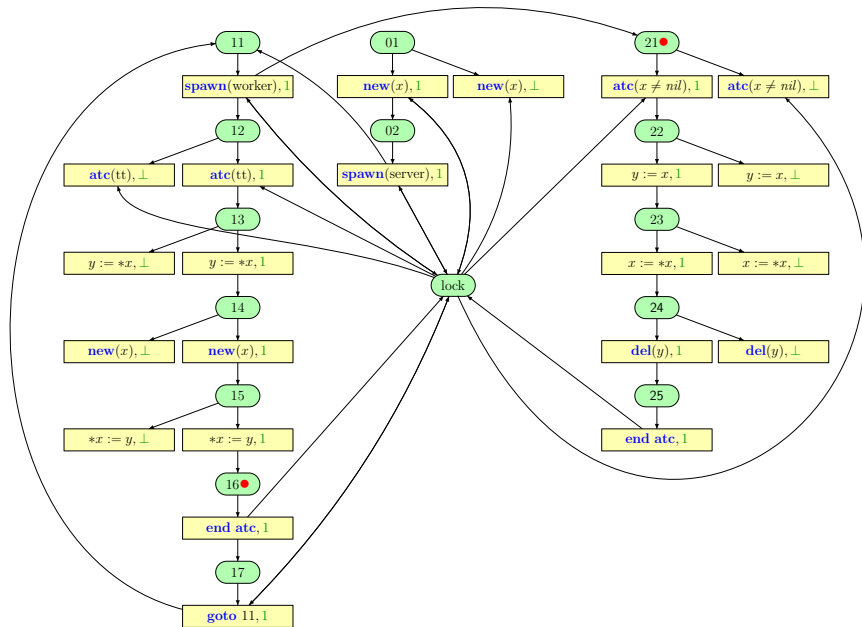


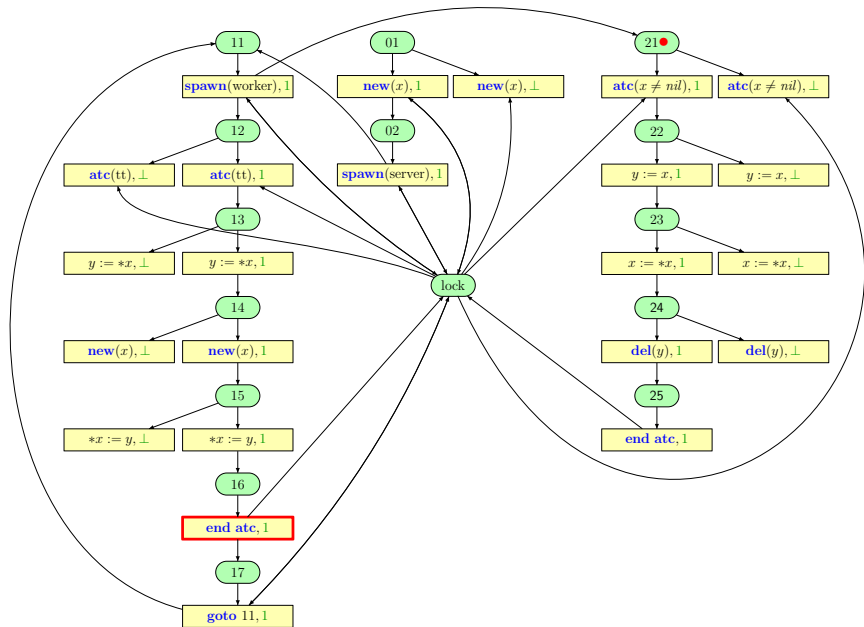


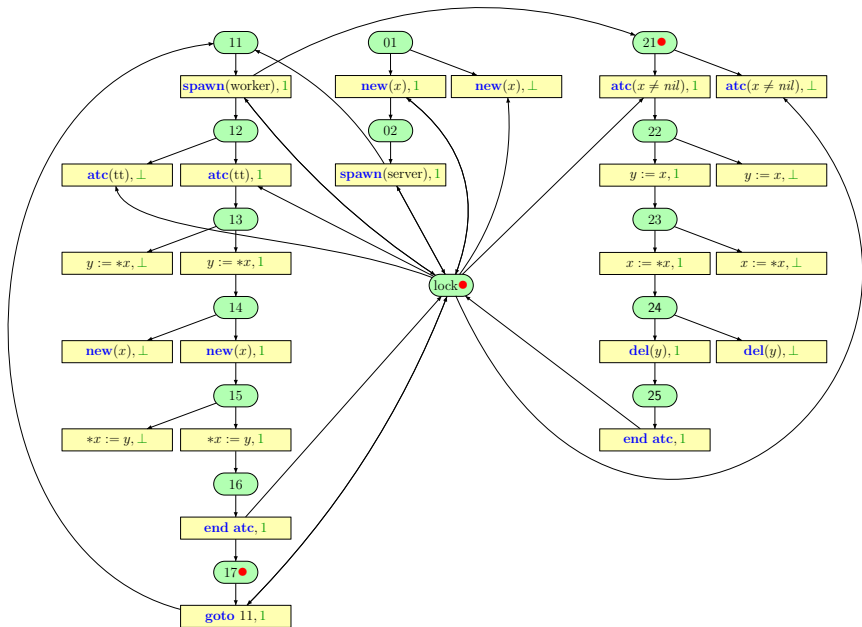


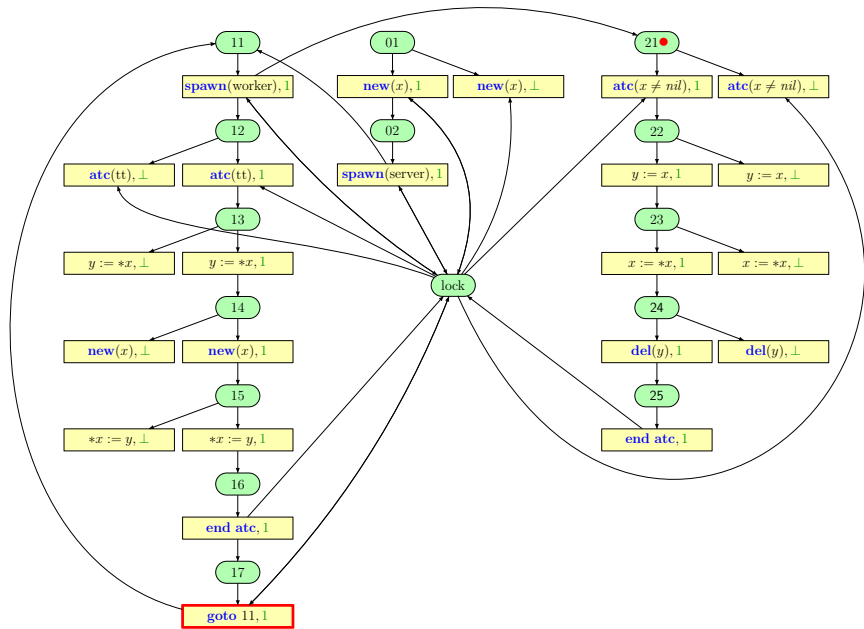


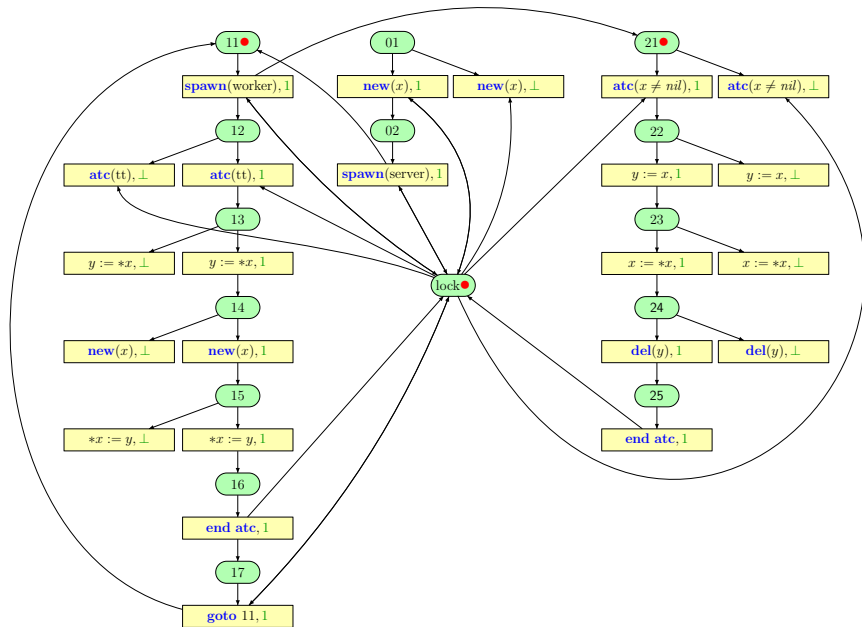


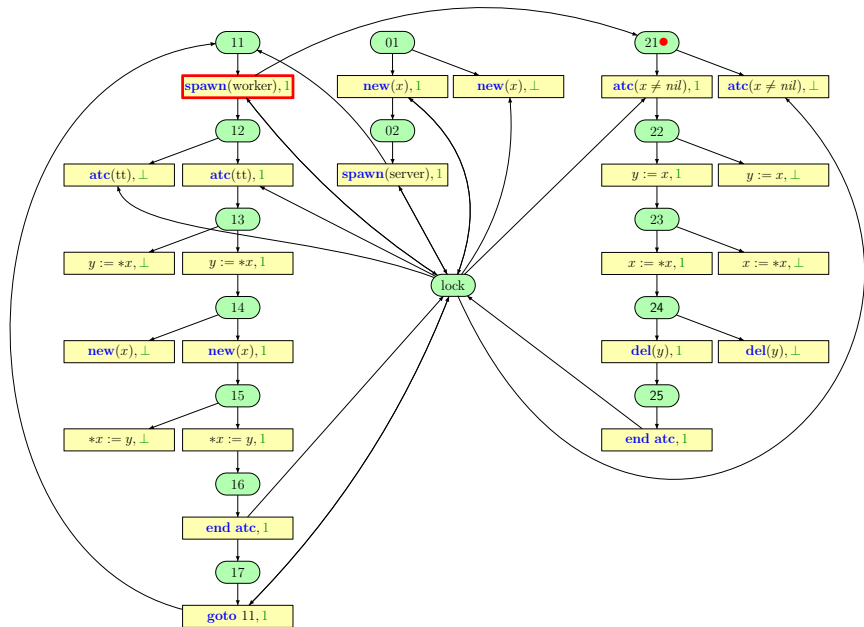


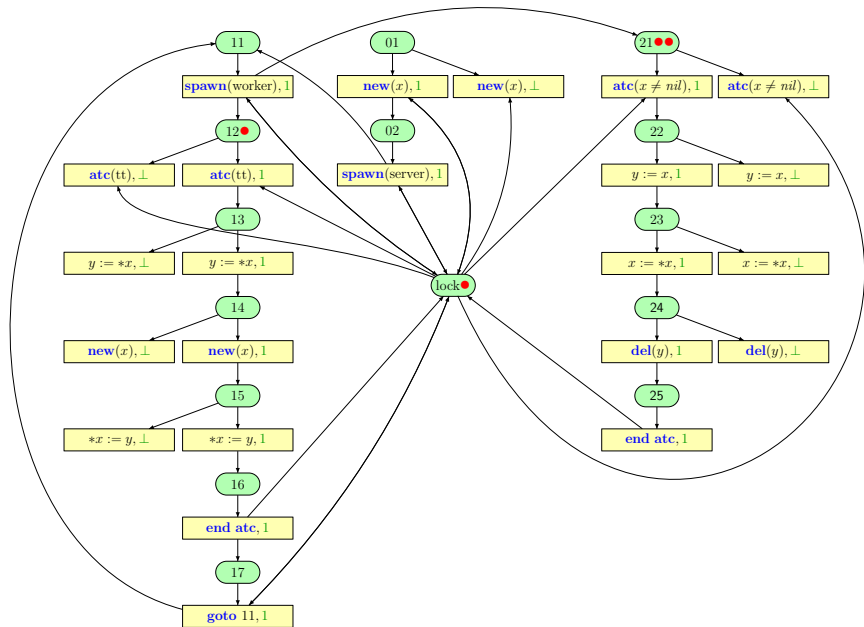


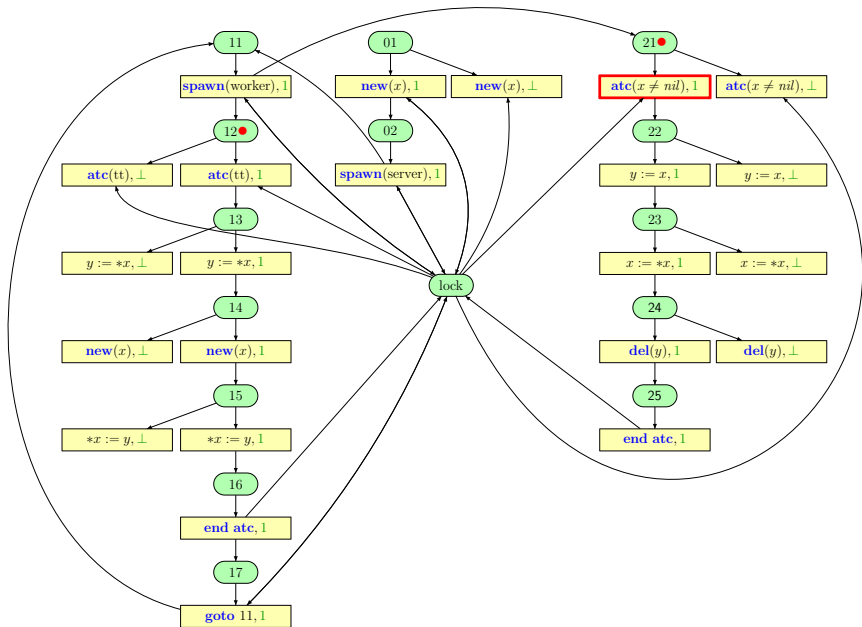


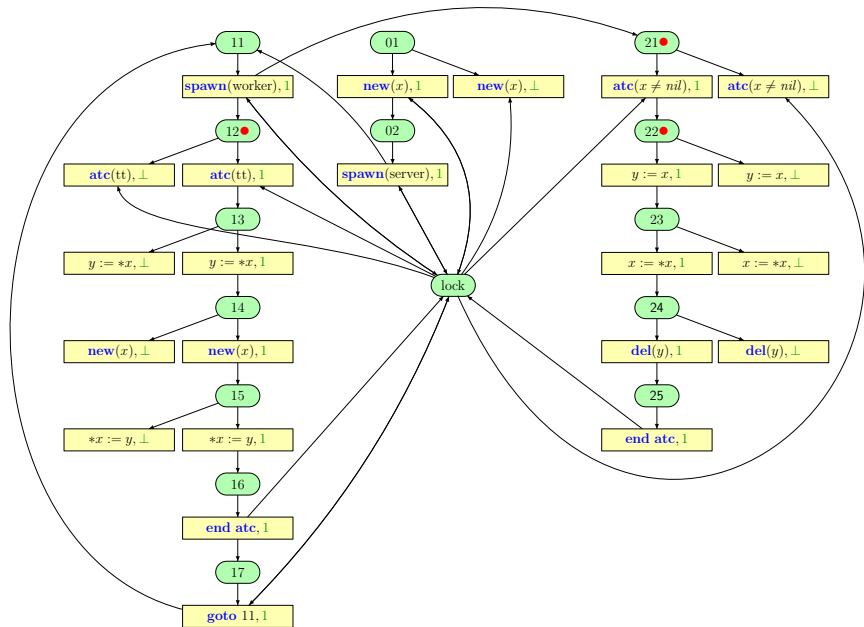


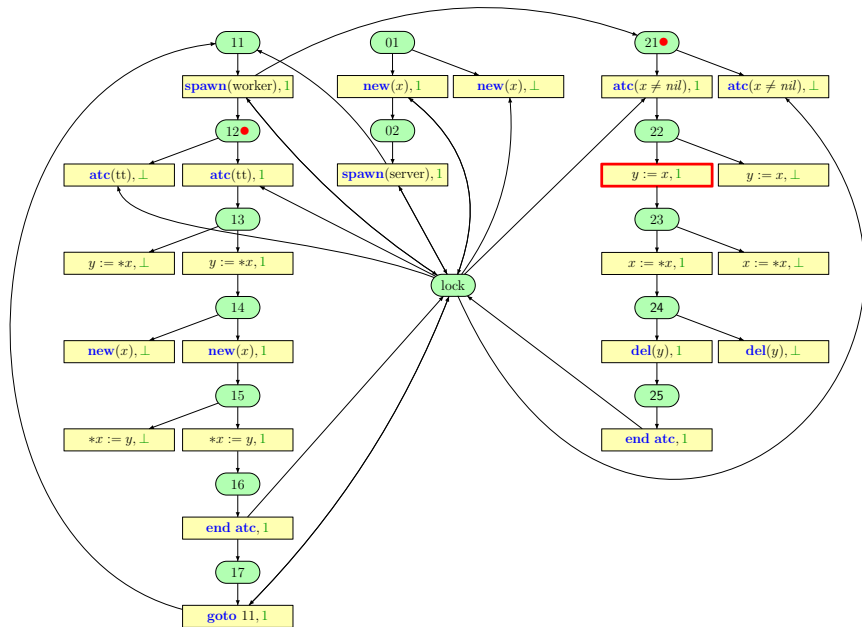


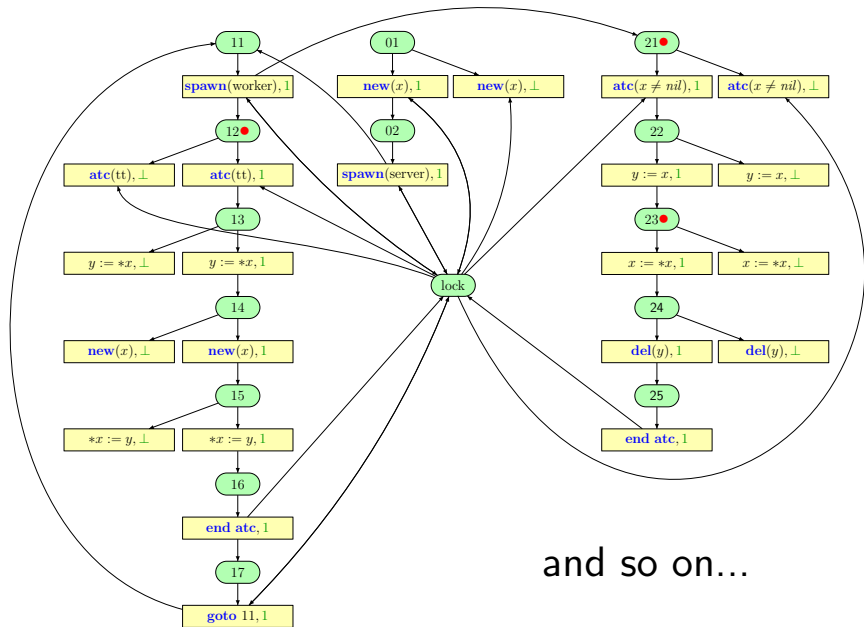












and so on...

Data-Abstract Semantics

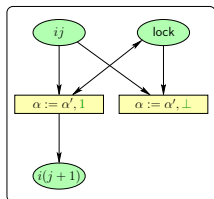
Combining Heap and Control-Flow Semantics

- Interpret abstract heap semantics as 1-safe Petri net \mathfrak{P}^h
- Combine control-flow and abstract heap semantics ($\mathfrak{P}^c \otimes \mathfrak{P}^h$)

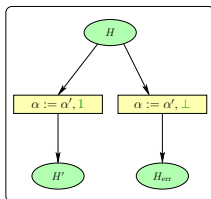
Data-Abstract Semantics

Combining Heap and Control-Flow Semantics

- Interpret abstract heap semantics as 1-safe Petri net \mathfrak{P}^h
- Combine control-flow and abstract heap semantics ($\mathfrak{P}^c \otimes \mathfrak{P}^h$)



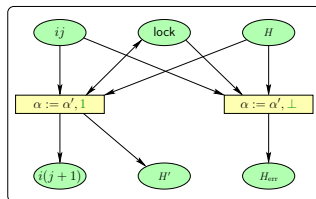
Control-Flow Transitions



Heap Transitions

 \otimes

=

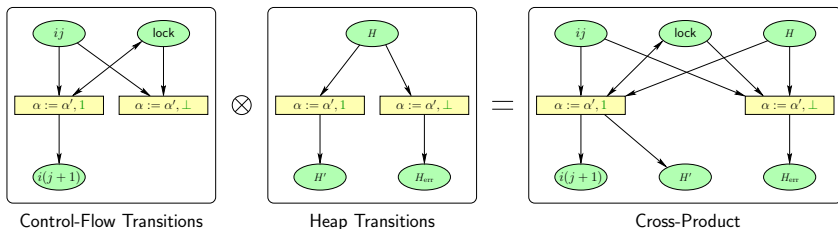


Cross-Product

Data-Abstract Semantics

Combining Heap and Control-Flow Semantics

- Interpret abstract heap semantics as 1-safe Petri net \mathfrak{P}^h
- Combine control-flow and abstract heap semantics ($\mathfrak{P}^c \otimes \mathfrak{P}^h$)



Observation

\mathfrak{P} is **unbounded** and cannot be represented by a finite LTS.

Overview

- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model
- ④ Model Checking
- ⑤ Conclusion

Semantics of Pointer Logic

Proposition (Correctness)

$\mathbf{H} \models \varphi$ for an abstract heap \mathbf{H} if and only if $H \models \varphi$ for all concrete heaps H represented by \mathbf{H} .

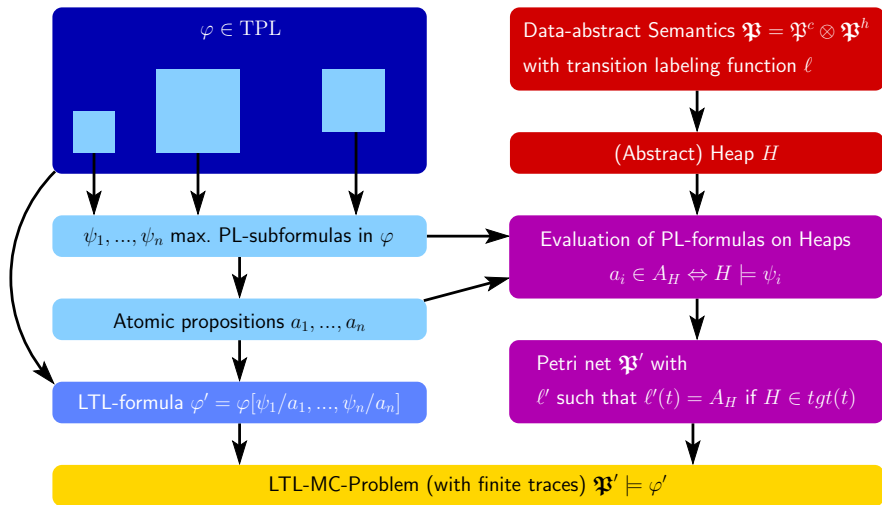
Semantics of Pointer Logic

Proposition (Correctness)

$\mathbf{H} \models \varphi$ for an abstract heap \mathbf{H} if and only if $H \models \varphi$ for all concrete heaps H represented by \mathbf{H} .

Note

To fulfill the proposition, the heap-precision constant M is chosen **dependent on φ** .

TPL \rightarrow LTL

First Result

Theorem

It is **decidable** whether $\mathfrak{P}' \models \varphi'$.

First Result

Theorem

It is **decidable** whether $\mathfrak{P}' \models \varphi'$.

Proof

Reduction to the **reachability problem** for Petri nets

Exponentially many instances, each solvable in EXPSPACE.

First Result

Theorem

It is **decidable** whether $\mathfrak{P}' \models \varphi'$.

Proof

Reduction to the **reachability problem** for Petri nets
Exponentially many instances, each solvable in EXPSPACE.

Consequence

To obtain a practically feasible algorithm, further abstraction is necessary.

Enforcing Boundedness

Abstract Petri Net

- Markings of the form $m : P \rightarrow \mathbb{C}$, where $\mathbb{C} = \{0, \dots, C, \star\}$
- Represent all values $> C$ by \star
- Introduction of **nondeterminism**

Enforcing Boundedness

Abstract Petri Net

- Markings of the form $m : P \rightarrow \mathbb{C}$, where $\mathbb{C} = \{0, \dots, C, \star\}$
- Represent all values $> C$ by \star
- Introduction of **nondeterminism**

Result

The **abstract** Petri net \mathfrak{P}' can equivalently be represented by a finite LTS.

Enforcing Boundedness

Abstract Petri Net

- Markings of the form $m : P \rightarrow \mathbb{C}$, where $\mathbb{C} = \{0, \dots, C, \star\}$
- Represent all values $> C$ by \star
- Introduction of **nondeterminism**

Result

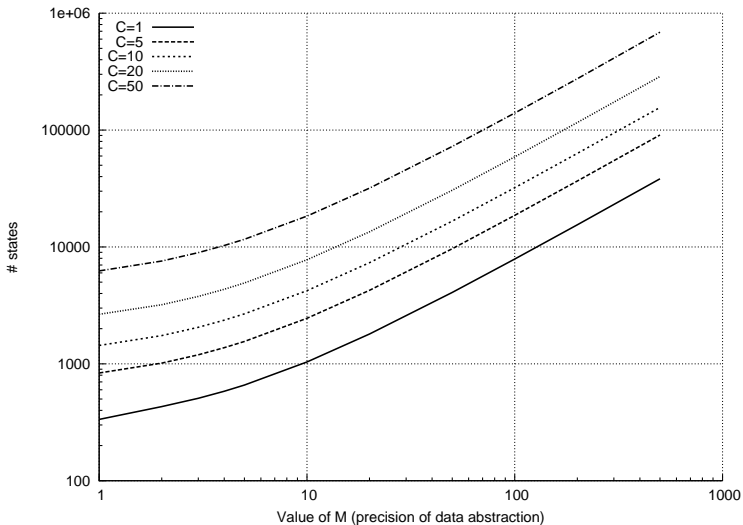
The **abstract** Petri net \mathfrak{P}' can equivalently be represented by a finite LTS.

Model Checking

Generate the LTS T corresponding to \mathfrak{P}' and apply an LTL model checking algorithm^a to verify $T \models \varphi$. If $T \models \varphi$ then also $\pi \models \varphi$.

^aLTL with finite traces

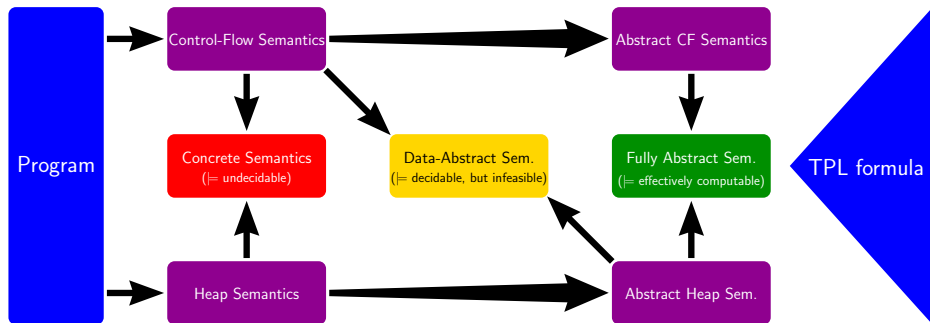
Experimental Results



Overview

- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model
- ④ Model Checking
- ⑤ Conclusion

Abstraction Scheme



Summary

- List-manipulating programming language with
 - dynamic memory allocation
 - dynamic threading
 - destructive updates
- Data and control-flow abstraction handled separately
- Reduction to standard LTL model checking (on finite traces)
- MC problem decidable for data-abstract semantics
- Refinement via global precision parameters

Summary

- **List-manipulating** programming language with
 - dynamic memory allocation
 - dynamic threading
 - destructive updates
- Data and control-flow abstraction handled **separately**
- Reduction to **standard LTL model checking** (on finite traces)
- **MC problem decidable for data-abstract semantics**
- Refinement via **global precision parameters**

Outlook

- Integrating **data-values** (e.g. integers)
- Handling **thread-local variables**
- **Refinement** based on counterexamples
- **Extension towards arbitrary data structures (graph grammars)**