

Verifying Dynamic Pointer-Manipulating Threads

Thomas Noll Stefan Rieger

MOVES: Software Modeling and Verification
RWTH Aachen University, Germany

28.5.2008



Formal Methods 2008, Turku/Finland

Multithreading and Pointer Structures

Problem

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

Multithreading and Pointer Structures

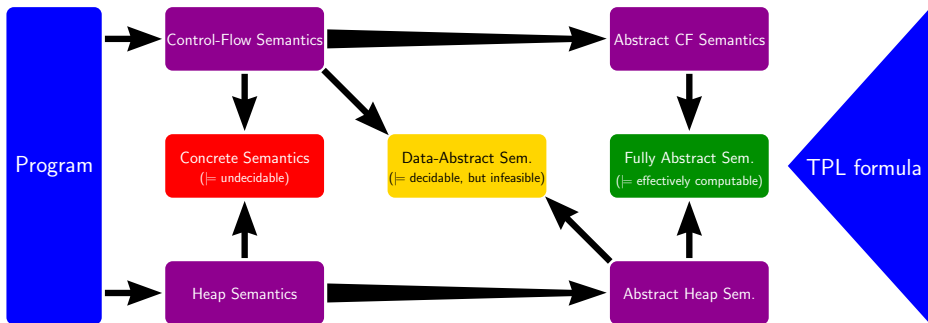
Problem

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

Approach

- simple concurrent programming language with **interleaving** semantics
- **LTL**-like pointer logic
- **heap** abstraction
- **control-flow** abstraction
- here: at most **one outgoing edge** (list-like structures)

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

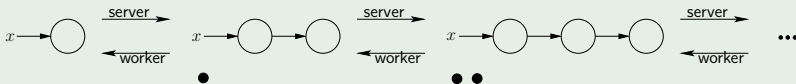
```

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( $x \neq nil$ );
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

Statements

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

Programming Language

Statements

$\text{PExp} ::= \text{PExp} \quad \text{atc}(\text{BExp}) \quad \text{spawn}(p) \quad \text{if BExp goto } n$
 $\text{new}(\text{PExp}) \quad \text{end atc} \quad \text{exit} \quad \text{goto } n$
 $\text{del}(\text{PExp})$

Pointer Expressions

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

No arbitrary dereferencing depths \rightarrow can be emulated

A Heap Logic

Pointer Logic (PL) - Interpretation on Heaps

- (comparison of) **pointer expressions**
- **reachability**
- (error-) **flags**
- **quantification** over heap objects

A Heap Logic

Pointer Logic (PL) - Interpretation on Heaps

- (comparison of) **pointer expressions**
- **reachability**
- (error-) **flags**
- **quantification** over heap objects

Temporal Pointer Logic - Interpretation on (In)finite Traces

- PL + LTL **temporal operators**
- no temporal operators within quantifiers
- safety and **liveness** properties expressible
- 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; )

```

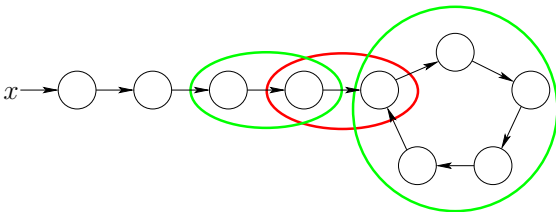
Properties

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

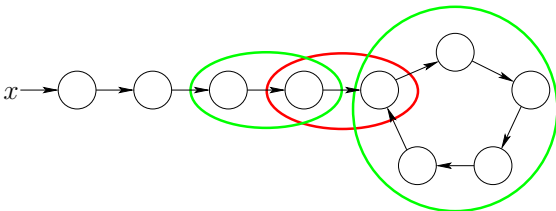
Idea



Approach

- Merge nodes along **chains** into a single **summary node** (similar to [Sagiv et al.], [Distefano]).

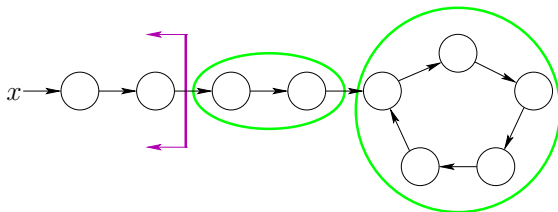
Idea



Approach

- Merge nodes along **chains** into a single **summary node** (similar to [Sagiv et al.], [Distefano]).
- Do only abstract from chains with $> M$ nodes.

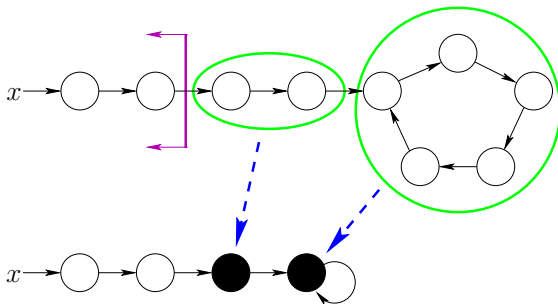
Idea



Approach

- Merge nodes along **chains** into a single **summary node** (similar to [Sagiv et al.], [Distefano]).
- Do only abstract from chains with $> M$ nodes.
- Do not merge nodes that are **close** to program variables.

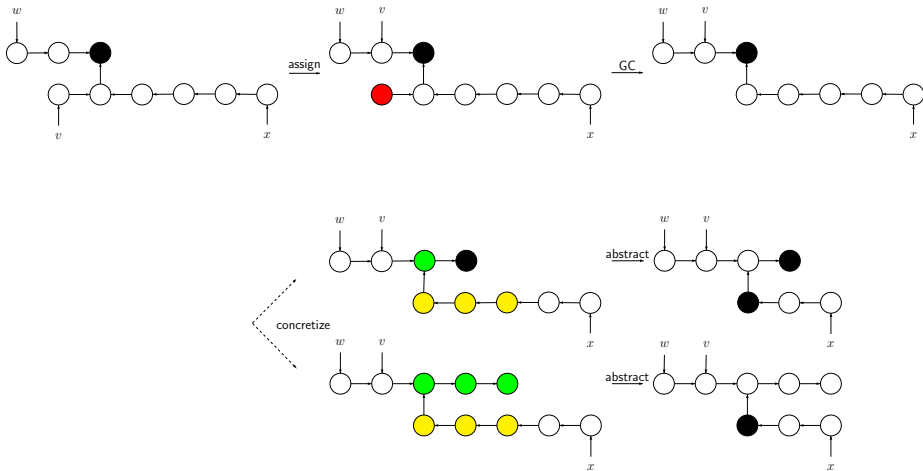
Idea



Approach

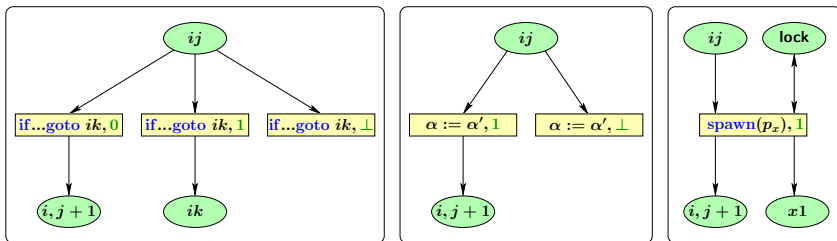
- Merge nodes along **chains** into a single **summary node** (similar to [Sagiv et al.], [Distefano]).
- Do only abstract from chains with $> M$ nodes.
- Do not merge nodes that are **close** to program variables.

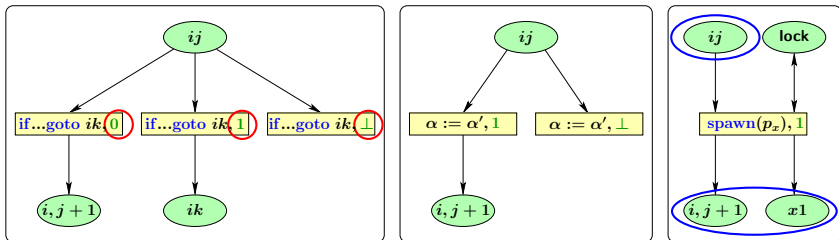
Example: Assignment $v := *w$ ($M = 2$)



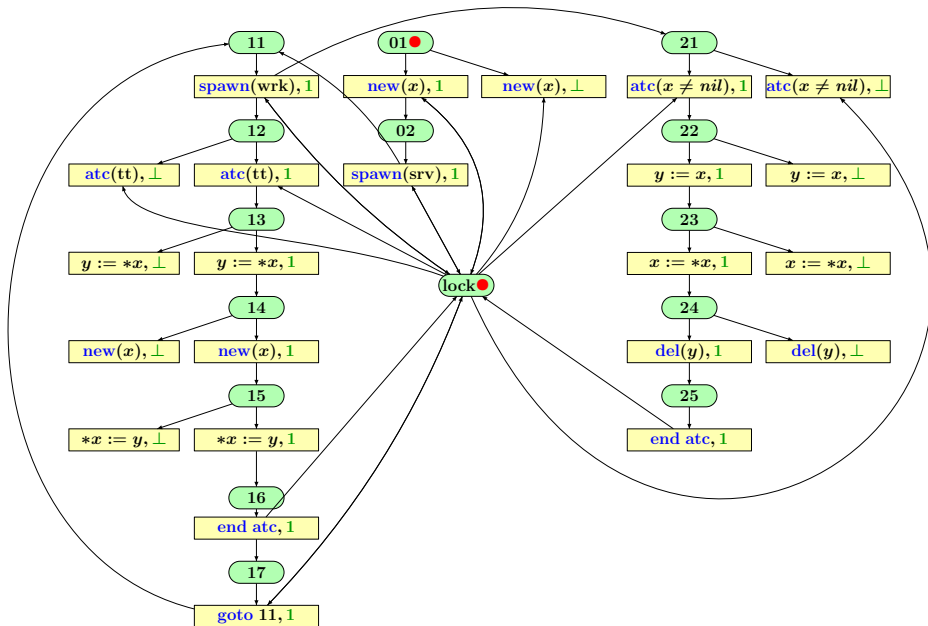
Overview

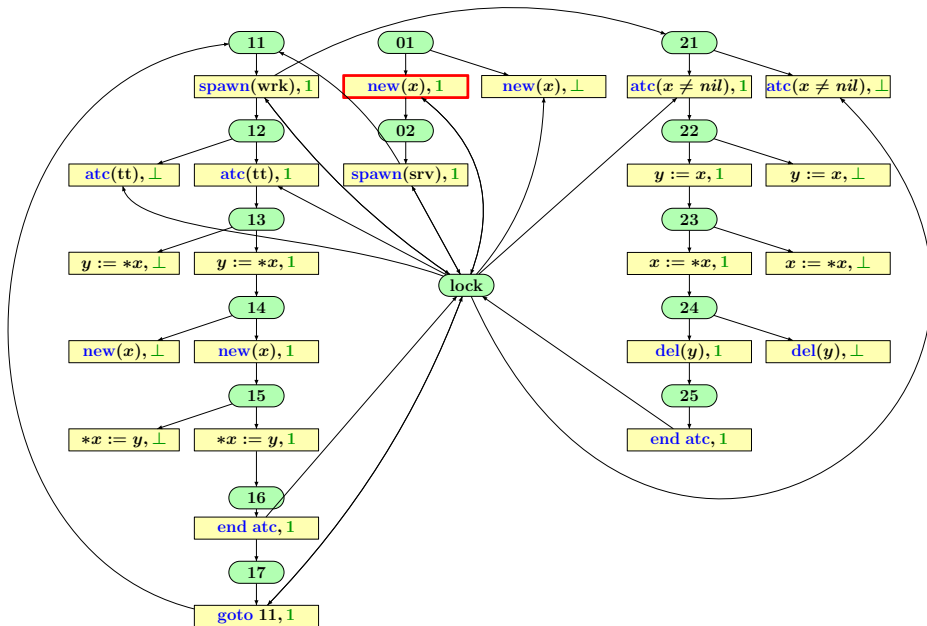
- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model**
- ④ Model Checking
- ⑤ Conclusion

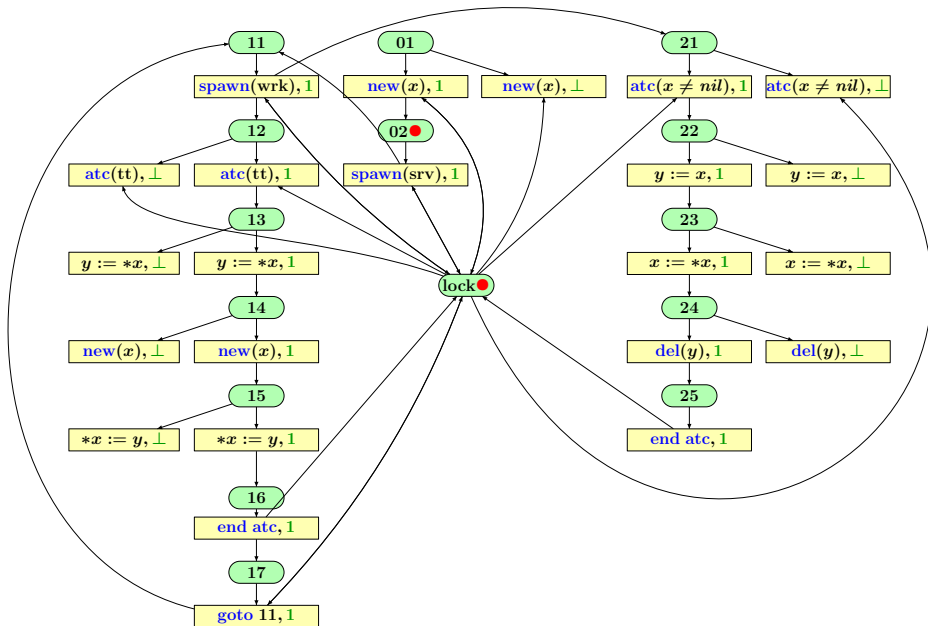
Modeling Control-Flow by Petri net \mathfrak{P}^c 

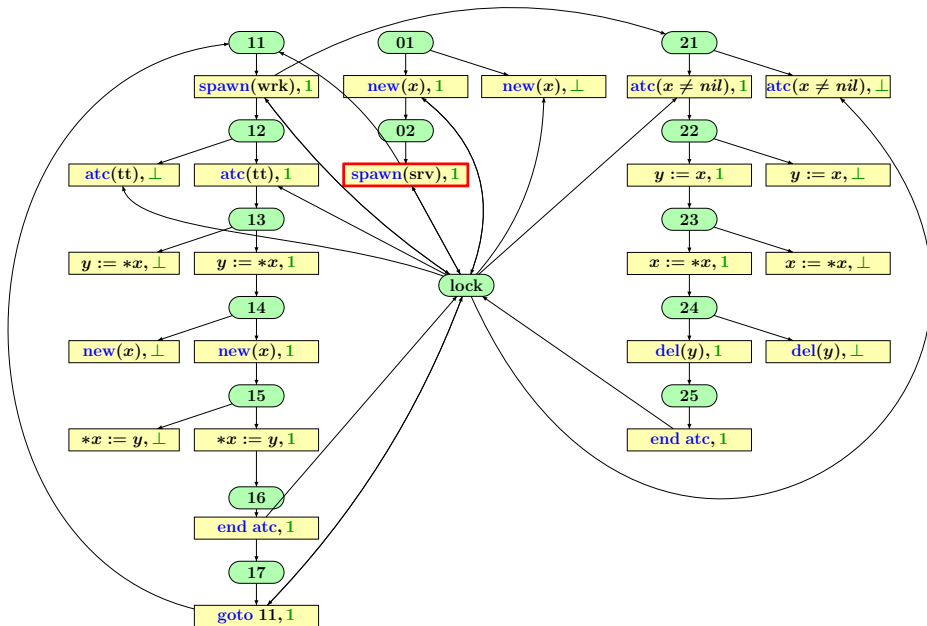
Modeling Control-Flow by Petri net \mathfrak{P}^c 

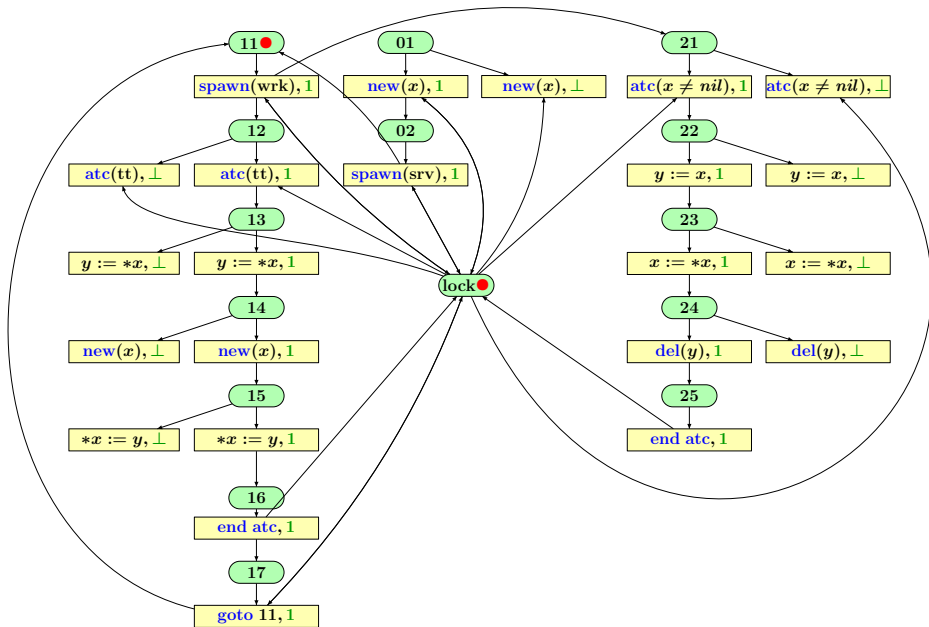
- Every control location corresponds to a place
- Transitions are labeled by instruction and flag $\in \{0, 1, \perp\}$

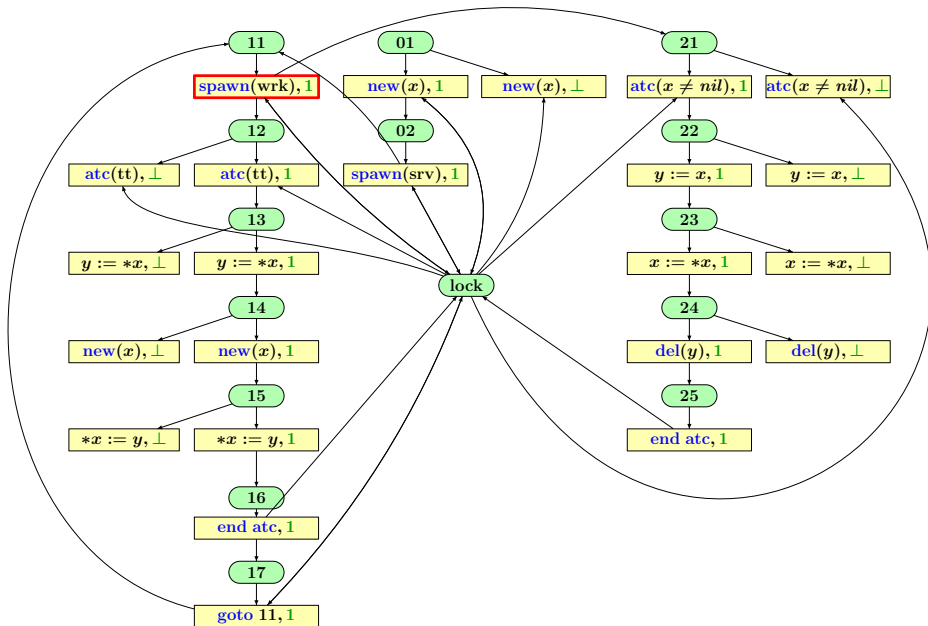


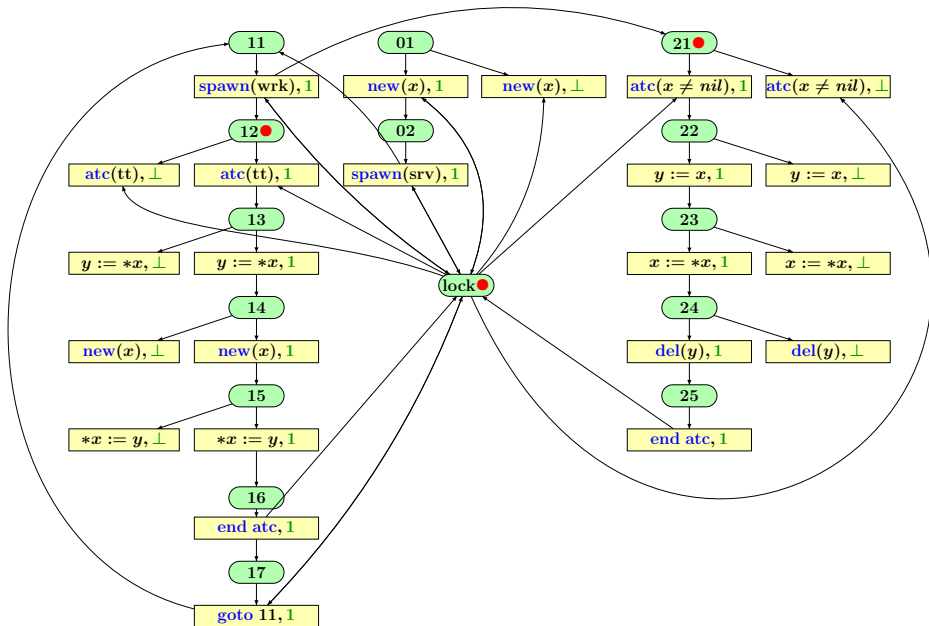


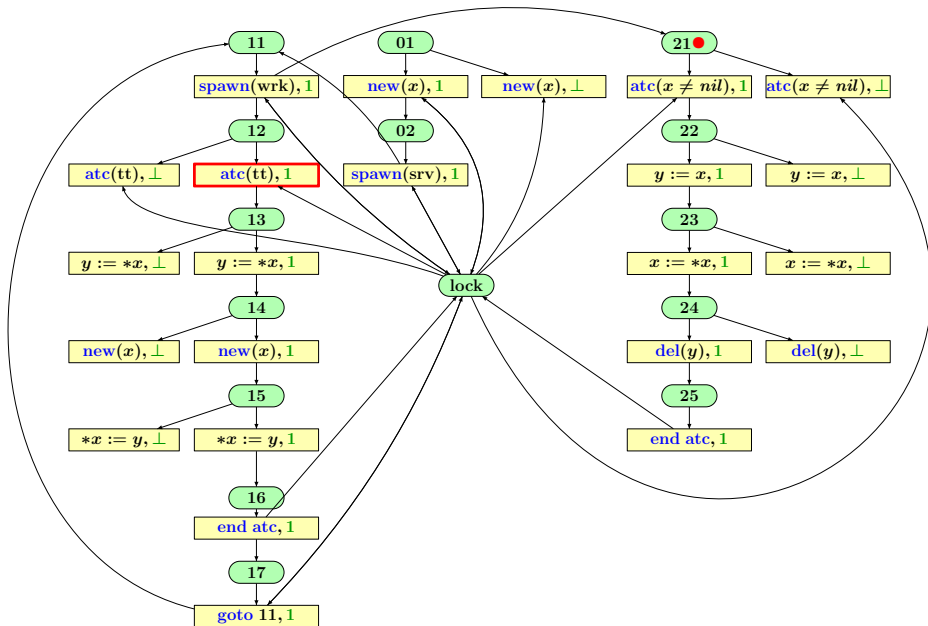


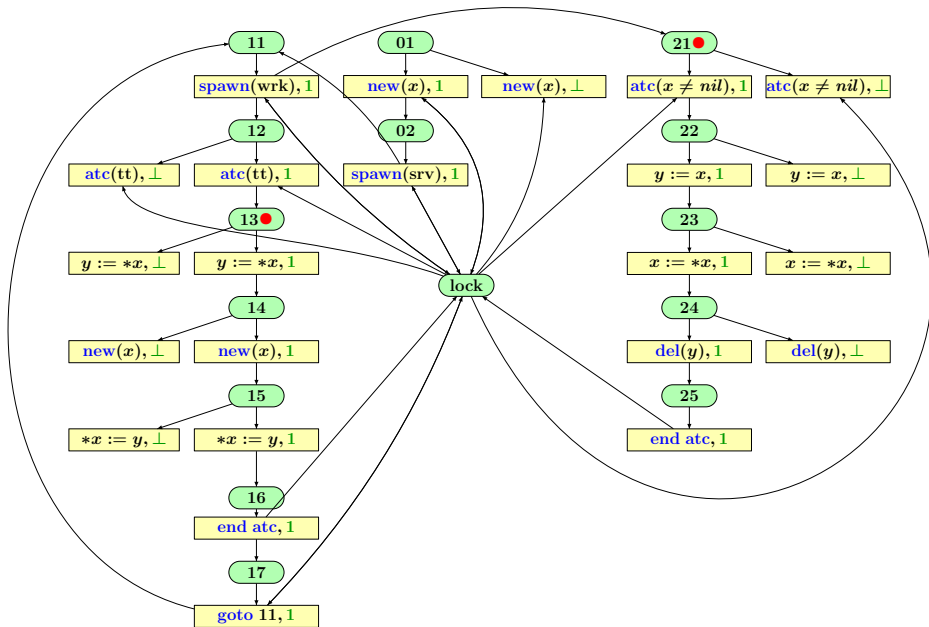


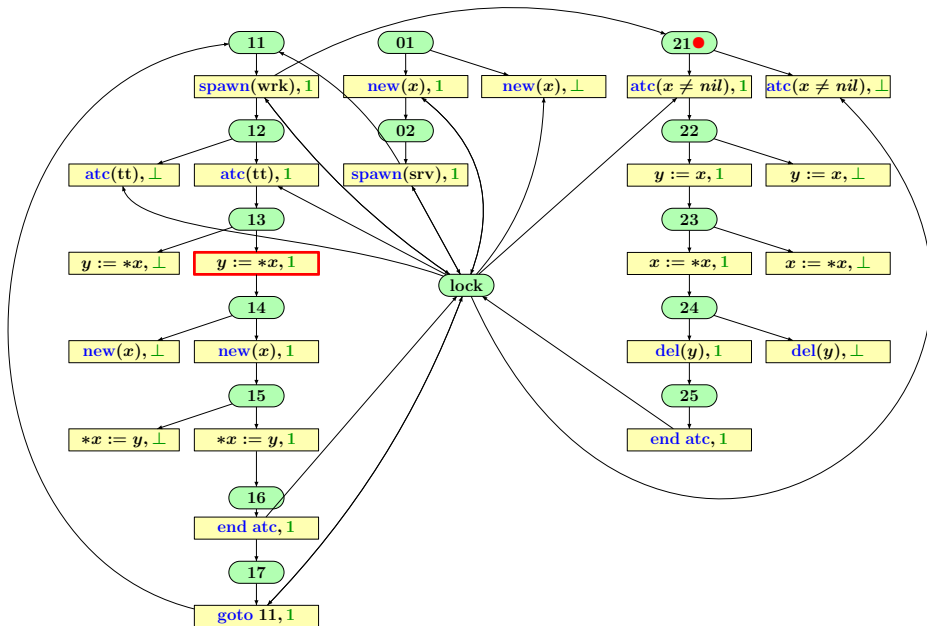


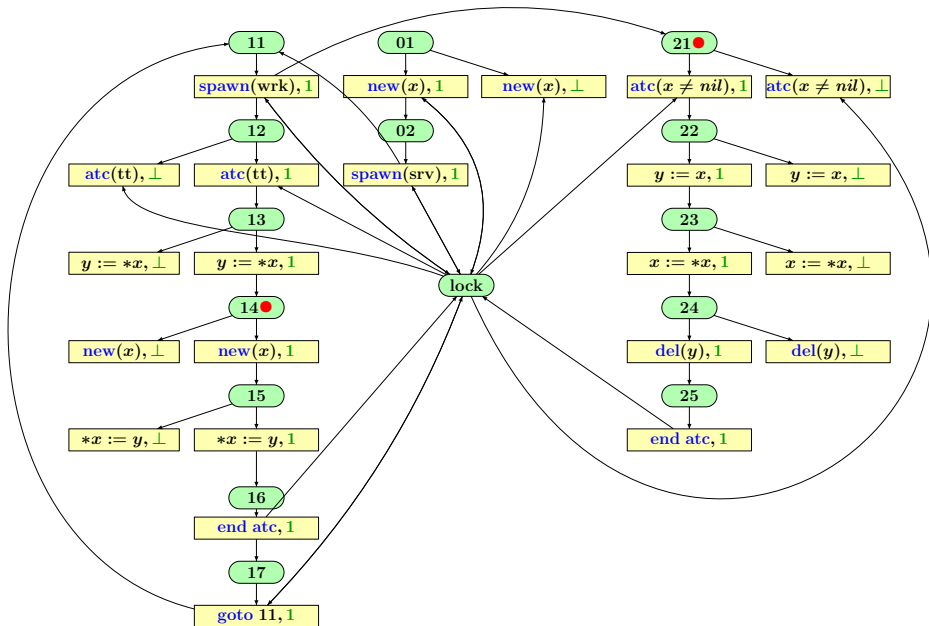


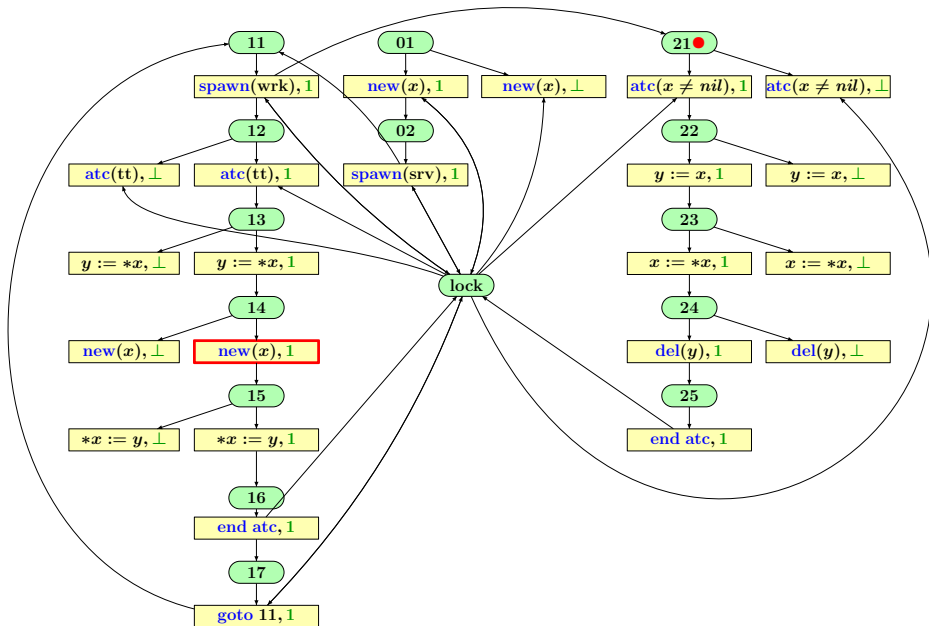


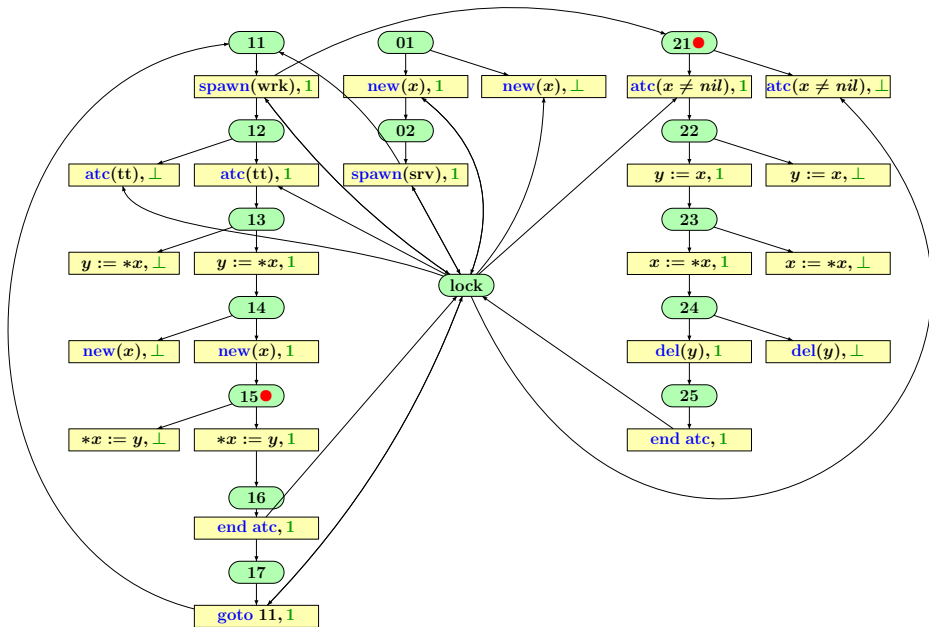


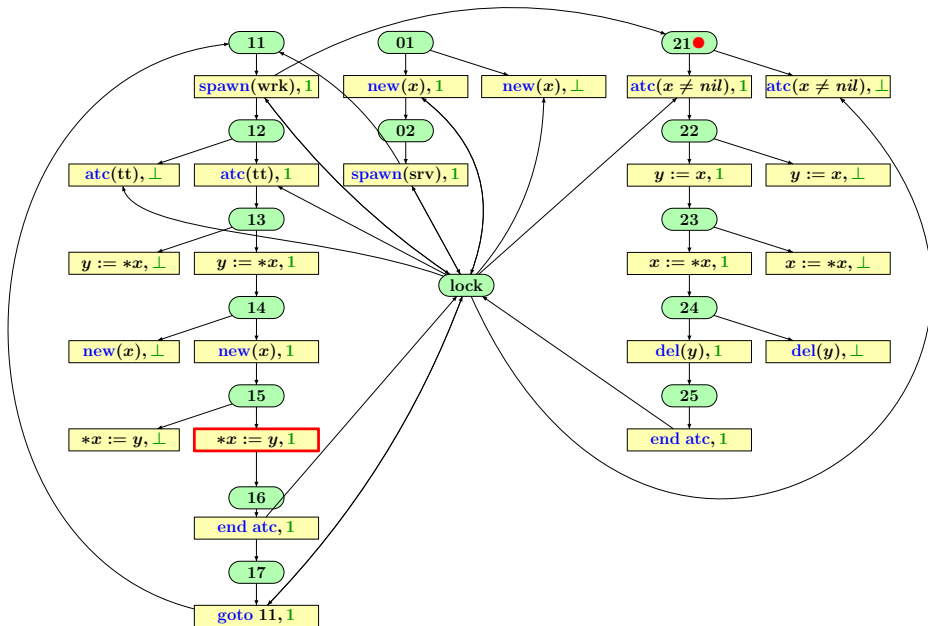


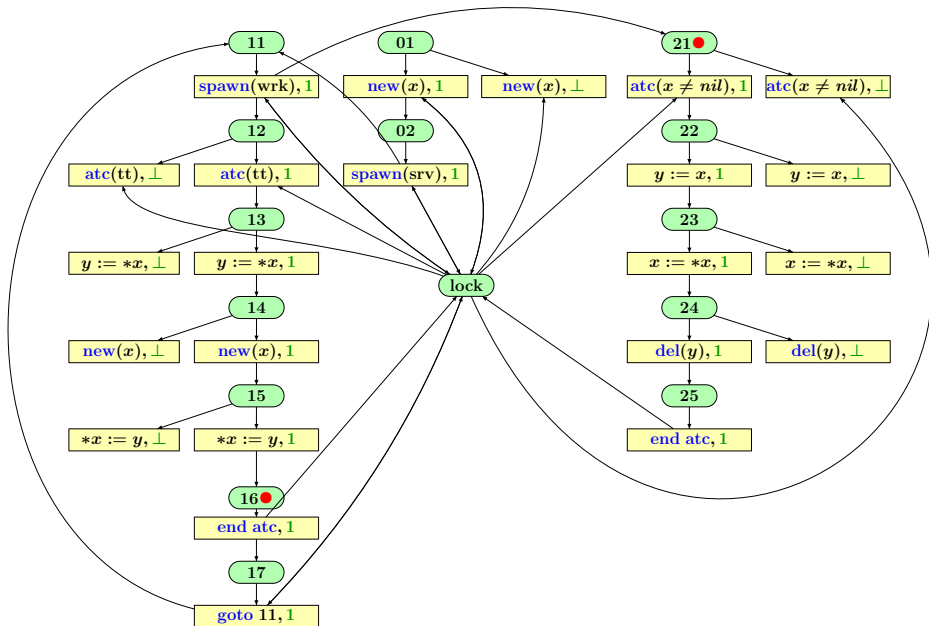


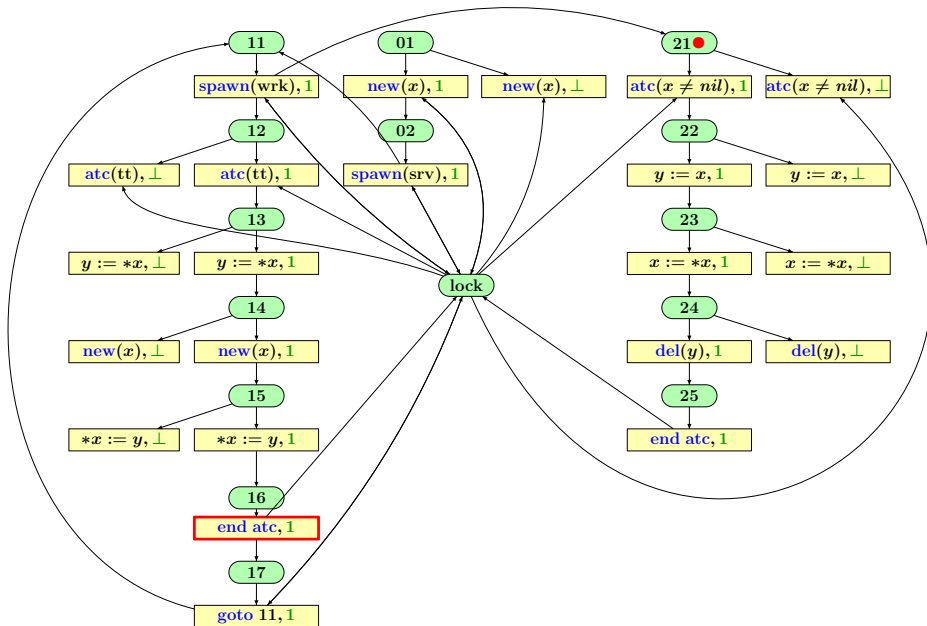


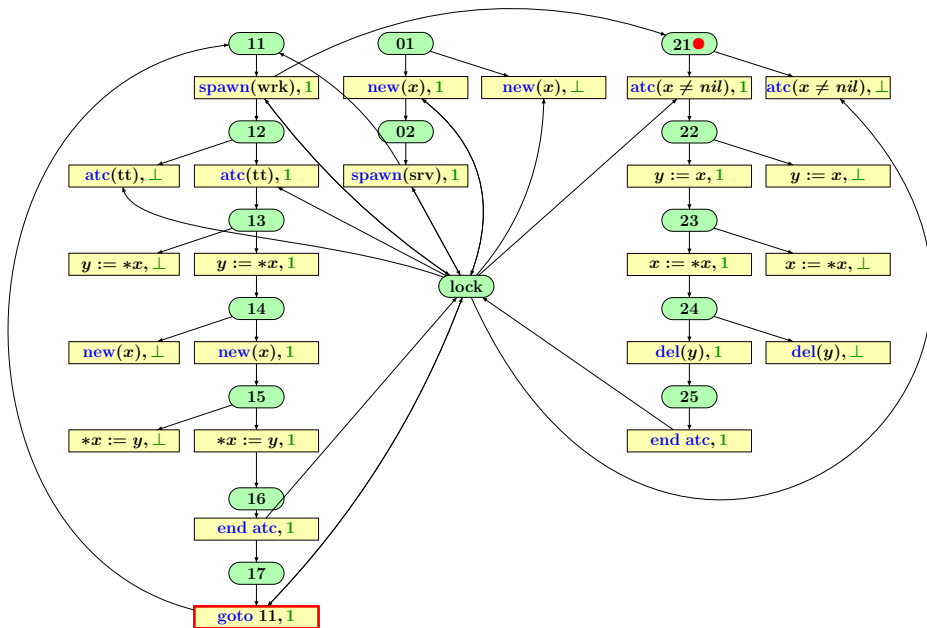


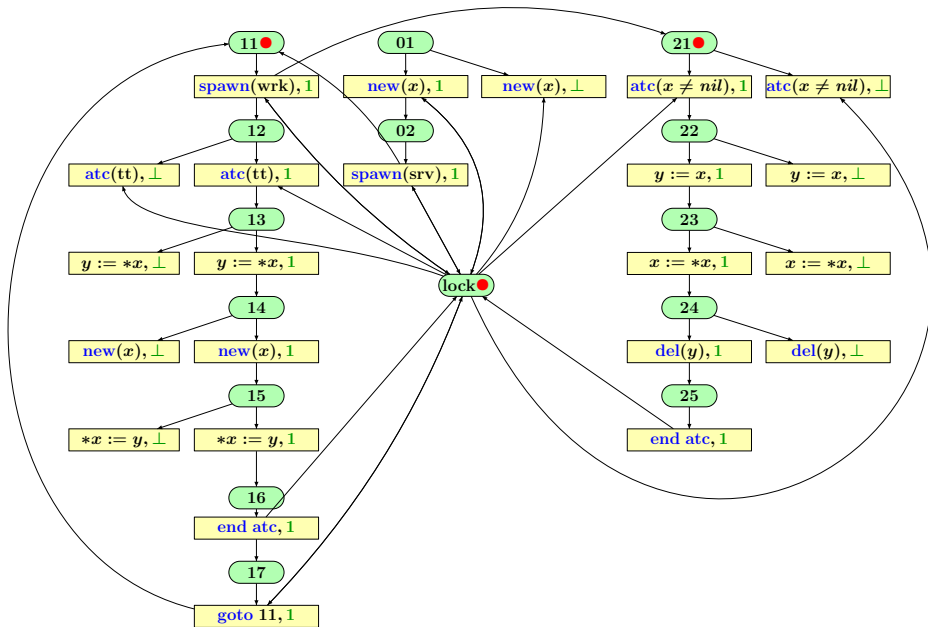


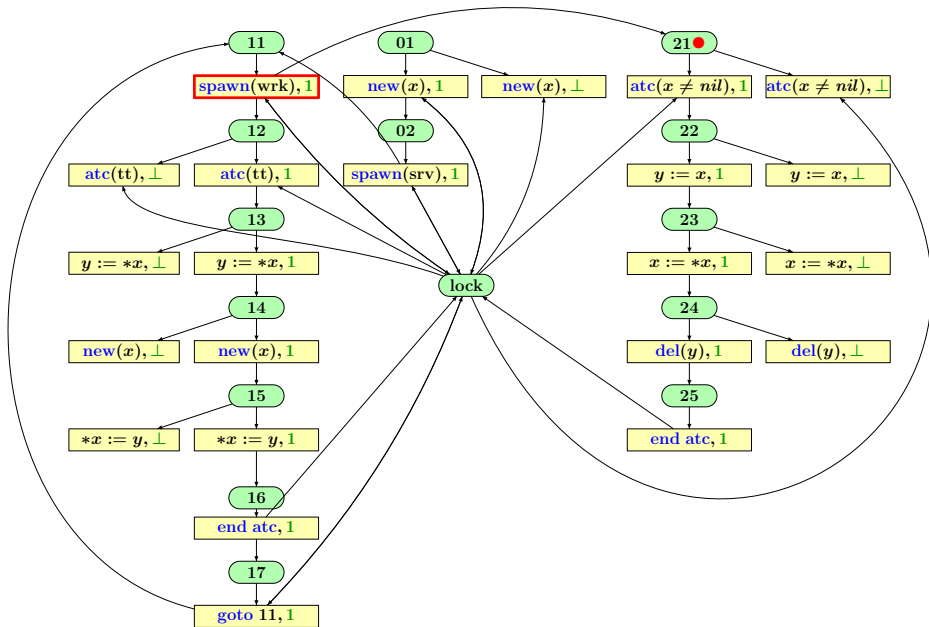


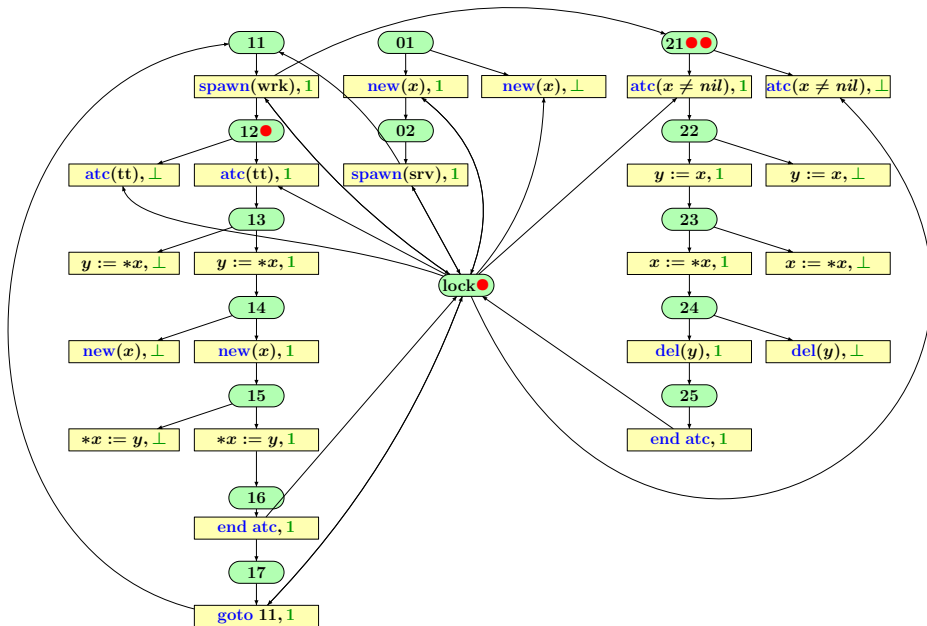


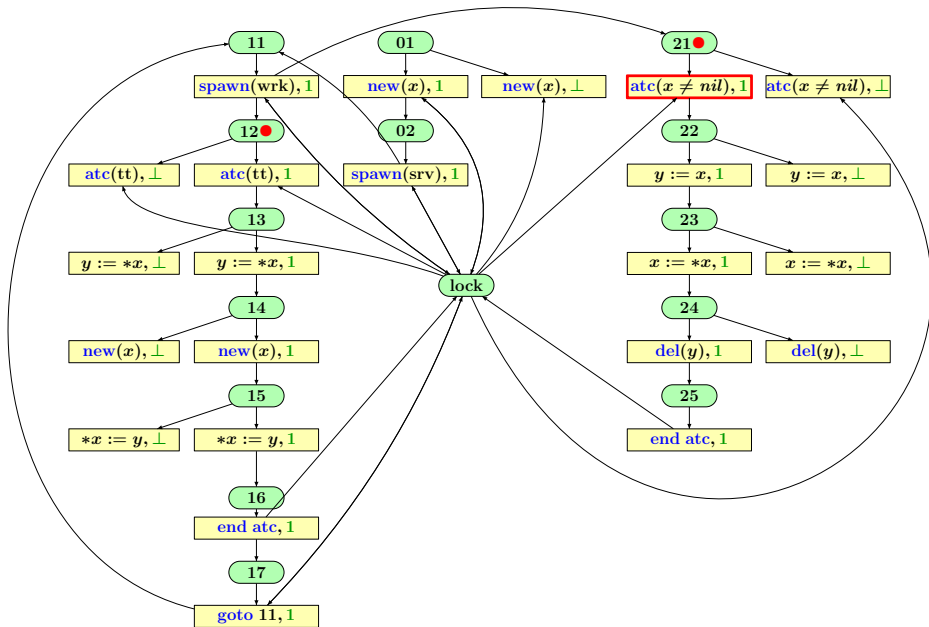


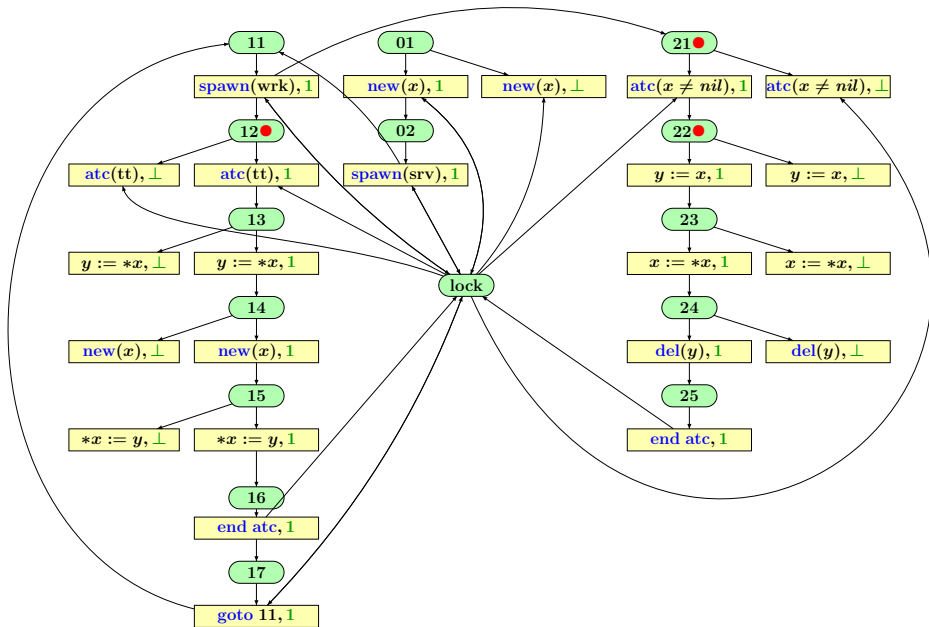


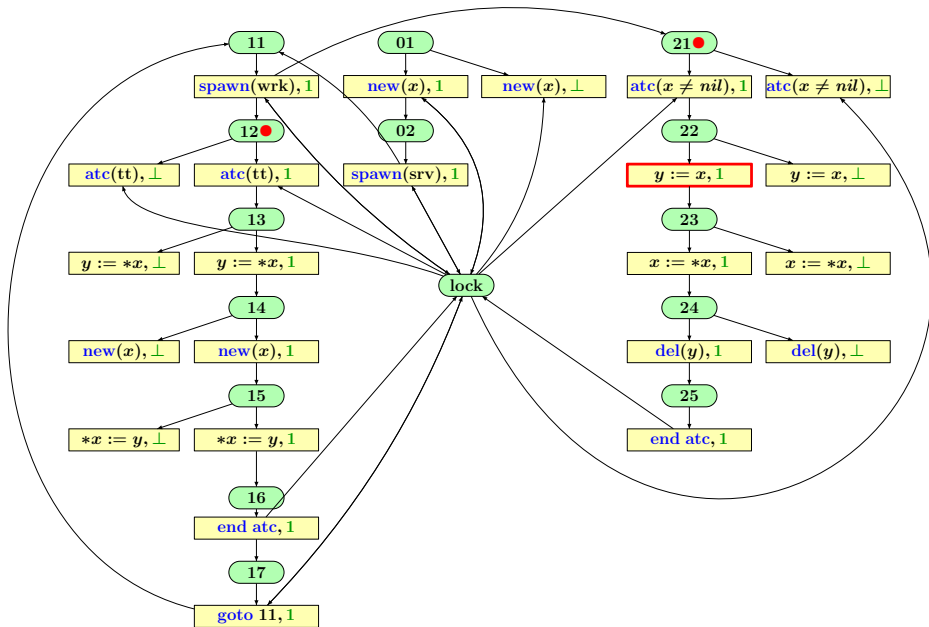


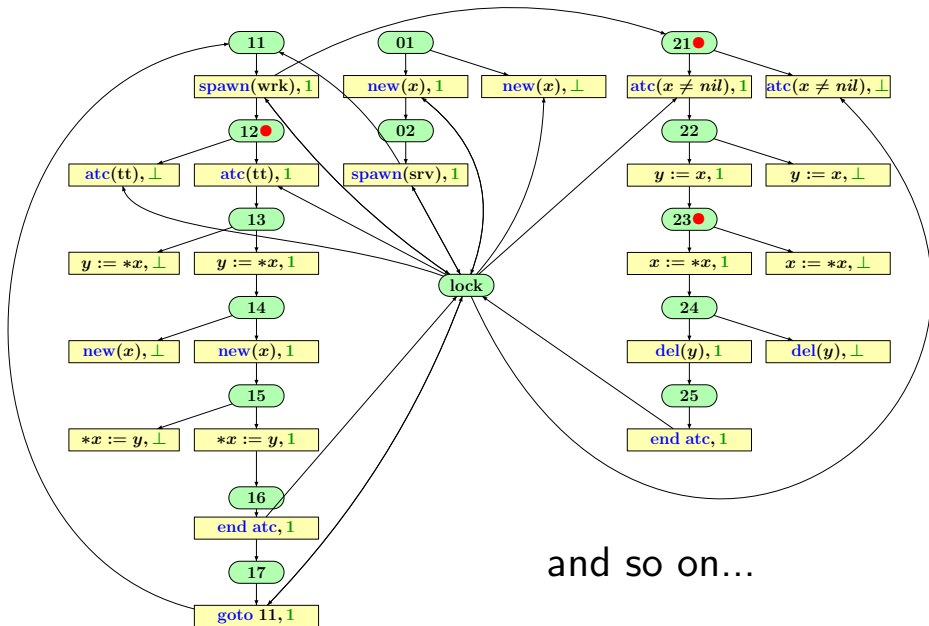












Data-Abstract Semantics

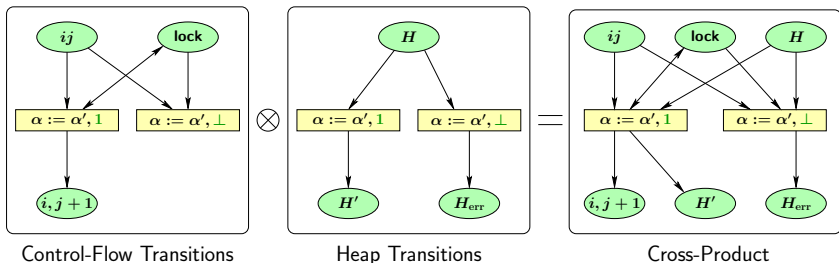
Combining Heap and Control-Flow Semantics

- 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

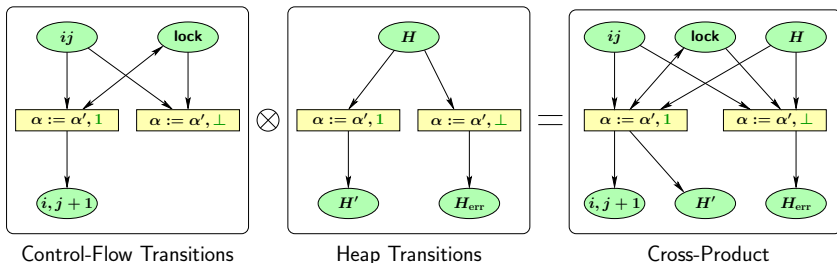
- Interpret abstract heap semantics as 1-safe Petri net \mathfrak{P}^h
- Combine control-flow and abstract heap semantics $\mathfrak{P}^c \otimes \mathfrak{P}^h$



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 $\mathfrak{P}^c \otimes \mathfrak{P}^h$

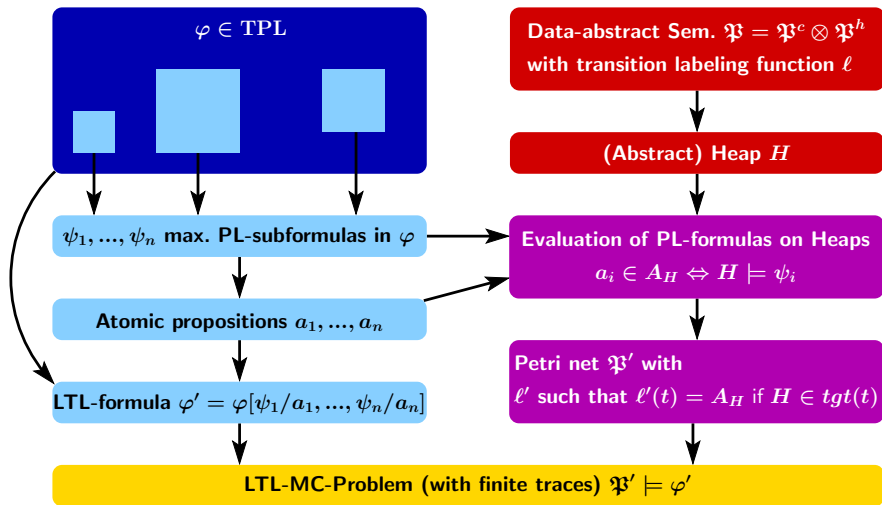


Observation

$\mathfrak{P}^c \otimes \mathfrak{P}^h$ 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

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

Construct two automata \mathfrak{A} and \mathfrak{B} for the words satisfying φ' :

- \mathfrak{A} is a **finite automaton** recognizing the **finite words**, and
- \mathfrak{B} a **Büchi-automaton** accepting the **infinite words**

First Result

Theorem

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

Proof

Construct two automata \mathfrak{A} and \mathfrak{B} for the words satisfying φ' :

- \mathfrak{A} is a **finite automaton** recognizing the **finite words**, and
- \mathfrak{B} a **Büchi-automaton** accepting the **infinite words**

[Esparza(94)]: MC-problem decidable

- using a special formula [Yen(92)] to formulate the Büchi acceptance condition for \mathfrak{B} and
- reduction to reachability problem for Petri net markings that is decidable in **EXPSPACE** [Lambert(92)].

First Result

Theorem

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

Proof

Construct two automata \mathfrak{A} and \mathfrak{B} for the words satisfying φ' :

- \mathfrak{A} is a **finite automaton** recognizing the **finite words**, and
- \mathfrak{B} a **Büchi-automaton** accepting the **infinite words**

[Esparza(94)]: MC-problem decidable

- using a special formula [Yen(92)] to formulate the Büchi acceptance condition for \mathfrak{B} and
- reduction to reachability problem for Petri net markings that is decidable in **EXPSPACE** [Lambert(92)].

⇒ **Further abstraction necessary for practically feasible algorithm**

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

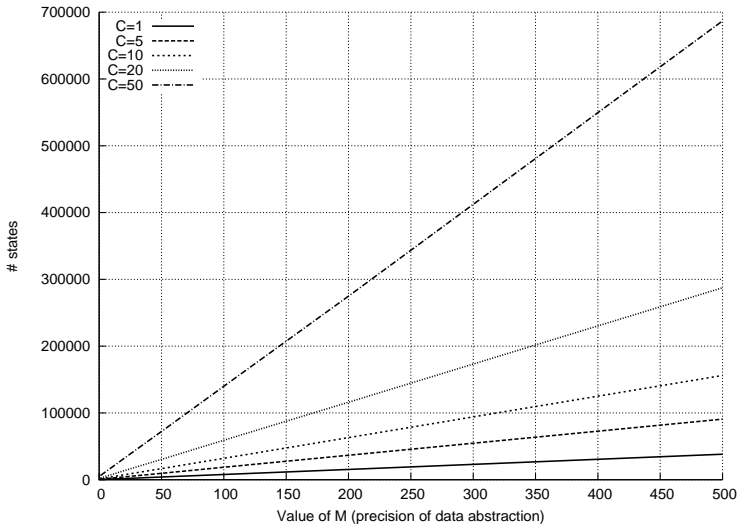
The **abstract** Petri net \mathfrak{P}' can equivalently be represented by a finite LTS.

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

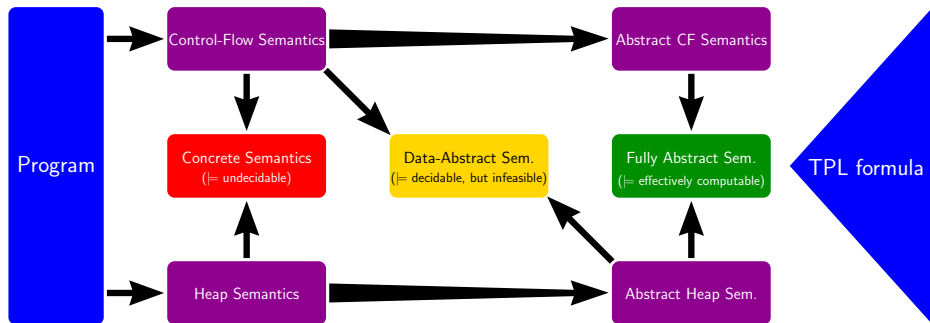
Experimental Results



Overview

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

Abstraction Scheme



Outlook

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