A Context Sensitive Memory Model for Software Model Checking

Arie Gurfinkel

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

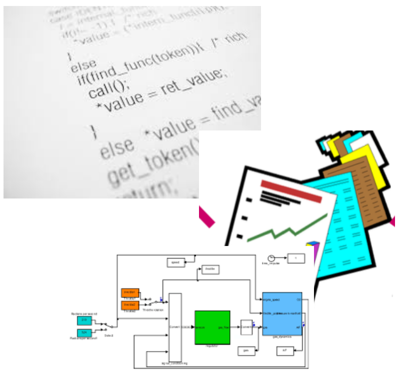
http://ece.uwaterloo.ca/~agurfink

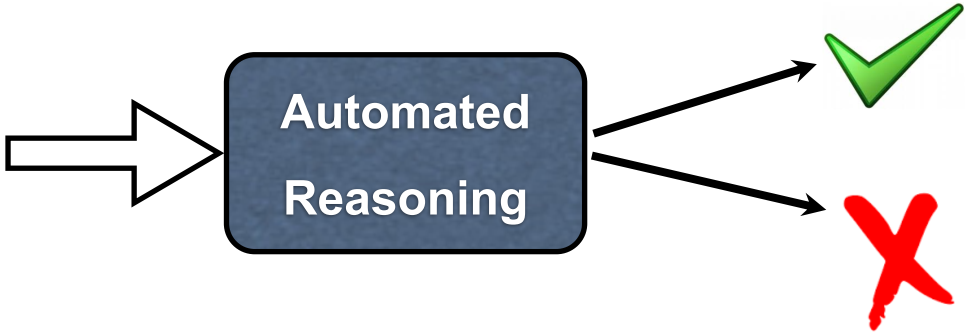
joint work with **Jorge A. Navas** (SRI)

UNIVERSITY OF WATERLOO

Automated (Software) Verification

Program and/or model





Correct

Incorrect

Alan M. Turing. 1936: "Undecidable"

Alan M. Turing. "Checking a large routine" 1949

How can one check a routine in the sense of making sure that it is right?

programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Automated Software Analysis

Model Checking



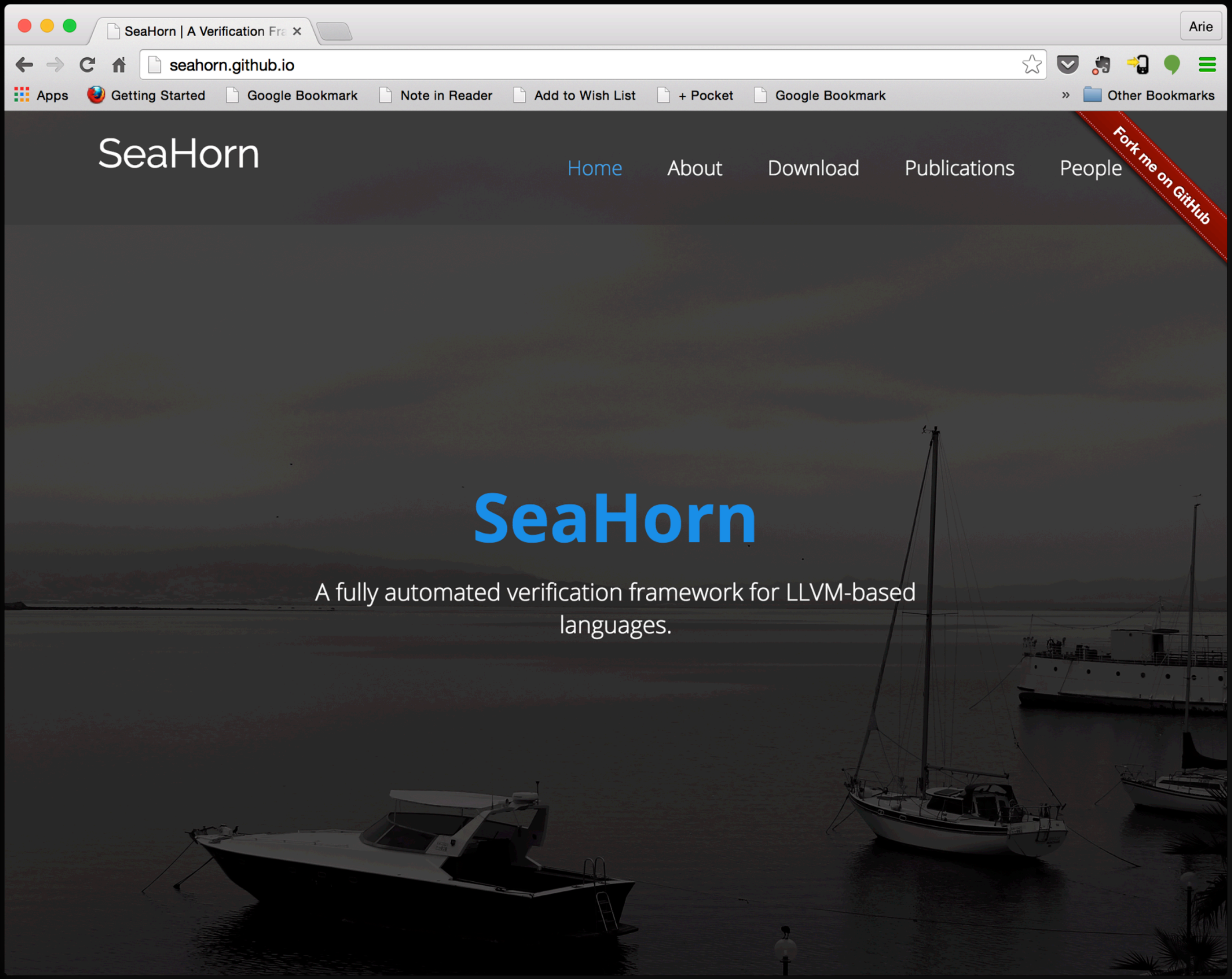
[Clarke and Emerson, 1981] [Queille and Sifakis, 1982]

Abstract Interpretation


[Cousot and Cousot, 1977]

Symbolic Execution

[King, 1976]



http://seahorn.github.io

http://seahorn.github.io

Algorithmic Logic-based Program Verification

Low-Level Bounded Model Checking (BMC)

- decide whether a low level program/circuit has an execution of a given length that violates a safety property
- effective decision procedure via encoding to propositional SAT

High-Level (Word-Level) Bounded Model Checking

- decide whether a program has an execution of a given length that violates a safety property
- efficient decision procedure via encoding to SMT

What is an SMT-like equivalent for Safety Verification?

- Logic: SMT-Constrained Horn Clauses
- Decision Procedure: Spacer / GPDR
  - extend IC3/PDR algorithms from Hardware Model Checking

Algorithmic Logic-Based Verification

Safety Properties

Program + Spec

Constrained Horn Clauses

Verification Condition (in Logic)

Spacer

Decision Procedure

Yes **No**

Horn Clauses for Program Verification

$\ell_{out}(x_0, w', e_o)$, which is an entry point into successor edges.
with the edges are formulated as follows:

$$p_{init}(x_0, w, \bot) \leftarrow x = x_0 \qquad \text{where } x \text{ occurs in } w$$
$$p_{exit}(x_0, ret, \top) \leftarrow \ell(x_0, w, \top) \quad \text{for each label } \ell \text{, and re}$$
$$p(x, ret, \bot, \bot) \leftarrow p_{exit}(x, ret, \bot)$$
$$p(x, ret, \bot, \top) \leftarrow p_{exit}(x, ret, \top)$$
$$\ell_{out}(x_0, w', e_o) \leftarrow \ell_{in}(x_0, w, e_i) \wedge \neg e_i \wedge \neg wlp(S, \neg(e_i =$$

```
5. incorrect :- Z=W+1, W≥0, W+1<
                read(A,W,U), read(A,Z
6. p(I1,N,B) :- 1≤I, I<N, D=I−1, I1=I+1, V=U+1,
                read(A,D,U), write(A
7. p(I,N,A) :- I=1, N>1.
```

De Angelis et al. Verifying Array Programs by Transforming Verification Conditions. VMCAI'14

Weakest Preconditions If we apply Boogie directly we obtain a translation from programs to Horn logic using a weakest liberal pre-condition calculus [26]:

$$\mathsf{ToHorn}(program) := wlp(Main(), \top) \wedge \bigwedge_{decl \in program} \mathsf{ToHorn}(decl)$$
$$\mathsf{ToHorn}(\mathbf{def}\ p(x)\ \{S\}) := wlp\begin{pmatrix}\mathbf{havoc}\ x_0; \mathbf{assume}\ x_0 = x; & \\ \mathbf{assume}\ p_{pre}(x); S, & p(x_0, ret)\end{pmatrix}$$
$$wlp(x := E, Q) := \mathbf{let}\ x = E\ \mathbf{in}\ Q$$
$$wlp((\mathbf{if}\ E\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2), Q) := wlp(((\mathbf{assume}\ E; S_1)\Box(\mathbf{assume}\ \neg E; S_2)), Q)$$
$$wlp((S_1 \Box S_2), Q) := wlp(S_1, Q) \wedge wlp(S_2, Q)$$
$$wlp(S_1; S_2, Q) := wlp(S_1, wlp(S_2, Q))$$
$$wlp(\mathbf{havoc}\ x, Q) := \forall x\ .\ Q$$
$$wlp(\mathbf{assert}\ \varphi, Q) := \varphi \wedge Q$$
$$wlp(\mathbf{assume}\ \varphi, Q) := \varphi \rightarrow Q$$
$$wlp((\mathbf{while}\ E\ \mathbf{do}\ S), Q) := inv(w) \wedge \forall w\ .\ \begin{pmatrix}((inv(w) \wedge E) & \rightarrow & wlp(S, inv(w))) \\ \wedge\ ((inv(w) \wedge \neg E) & \rightarrow & Q)\end{pmatrix}$$

To translate a procedure call $\ell : y := q(E); \ell'$ within a procedure p, create he clauses:

$$p(w_0, w_4) \leftarrow p(w_0, w_1), call(w_1, w_2), q(w_2, w_3), return(w_1, w_3, w_4)$$
$$q(w_2, w_2) \leftarrow p(w_0, w_1), call(w_1, w_2)$$
$$call(w, w') \leftarrow \pi = \ell, x' = E, \pi' = \ell_{q_{init}}$$
$$return(w, w', w'') \leftarrow \pi' = \ell_{q_{exit}}, w'' = w[ret'/y, \ell'/\pi]$$

Bjørner, Gurfinkel, McMillan, and Rybalchenko:

Horn Clause Solvers for Program Verification

UNIVERSITY OF WATERLOO

Horn Clauses for Concurrent / Distributed / Parameterized Systems

For assertions $R_1, \ldots, R_N$ over V and $E_1, \ldots, E_N$ over V, V',

$$
\begin{array}{lll}
\text{CM1}: & init(V) & \to R_i(V) \\
\text{CM2}: & R_i(V) \wedge \rho_i(V, V') & \to R_i(V') \\
\text{CM3}: & (\bigvee_{i \in 1..N \setminus \{j\}} R_i(V) \wedge \rho_i(V, V')) & \to E_j(V, V') \\
\text{CM4}: & R_i(V) \wedge E_i(V, V') \wedge \rho_i^{=}(V, V') & \to R_i(V') \\
\text{CM5}: & R_1(V) \wedge \cdots \wedge R_N(V) \wedge error(V) & \to false \\
\hline
 & \text{multi-threaded program } P \text{ is safe} &
\end{array}
$$

Rybalchenko et al. Synthesizing Software Verifiers from Proof Rules. PLDI'12

$$\left\{ R(\mathsf{g}, \mathsf{p}_{\sigma(1)}, \mathsf{l}_{\sigma(1)}, \ldots, \mathsf{p}_{\sigma(k)}, \mathsf{l}_{\sigma(k)}) \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \right\}_{\sigma \in S_k} \tag{6}$$

$$R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge Init(\mathsf{g}, \mathsf{l}_1) \wedge \cdots \wedge Init(\mathsf{g}, \mathsf{l}_k) \tag{7}$$

$$R(\mathsf{g}', \mathsf{p}_1, \mathsf{l}_1', \ldots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge \left((\mathsf{g}, \mathsf{l}_1) \xrightarrow{\mathsf{p}_1} (\mathsf{g}', \mathsf{l}_1') \right) \wedge R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \tag{8}$$

$$R(\mathsf{g}', \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_0, \mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge \left((\mathsf{g}, \mathsf{l}_0) \xrightarrow{\mathsf{p}_0} (\mathsf{g}', \mathsf{l}_0') \right) \wedge RConj(0, \ldots, k) \tag{9}$$

$$false \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_r) \wedge \Big(\bigwedge_{j=1,\ldots,m} (\mathsf{p}_j = p_j \wedge (\mathsf{g}, \mathsf{l}_j) \in E_j) \Big) \wedge RConj(1, \ldots, r) \tag{10}$$

Figure 4: Horn constraints encoding a homogeneous infinite system with the help of a k-indexed invariant. S_k is the symmetric group on $\{1, \ldots, k\}$, i.e., the group of all permutations of k numbers; as an optimisation, any generating subset of S_k, for instance transpositions, can be used instead of S_k. In (10), we define $r = \max\{m, k\}$.

Hojjat et al. Horn Clauses for Communicating Timed Systems. HCVS'14

$$
\begin{array}{ll}
\text{(initial)} & \mathsf{init}(g, x_1) \wedge \cdots \wedge \mathsf{init}(g, x_n) \to Inv(g, \ell_{\mathsf{init}}, x_1, \ldots, \ell_{\mathsf{init}}, x_k) \\
\text{(inductive)} & Inv(g, \ell_1, x_1, \ldots, \ell_i, x_i, \ldots, \ell_k, x_k) \wedge s(g, x_i, g', x_i') \to Inv(g', \ell_1, x_1, \ldots, \ell_i', x_i', \ldots, \ell_k, \\
\text{(non-interference)} & Inv(g, \ell_1, x_1, \ldots, \ell_k, x_k) \wedge \\
 & Inv(g, \ell^\dagger, x^\dagger, \ell_2, x_2, \ldots, \ell_k, x_k) \wedge \\
 & \vdots \\
 & Inv(g, \ell_1, x_1, \ldots, \ell_{k-1}, x_{k-1}, \ell^\dagger, x^\dagger) \wedge s(g, x^\dagger, g', \cdot) \to Inv(g', \ell_1, x_1, \ldots, \ell_k, x_k) \\
\text{(safe)} & Inv(g, \ell_1, x_1, \ldots, \ell_k, x_k) \wedge \mathsf{err}(g, \ell_1, x_1, \ldots, \ell_m, x_m) \to false
\end{array}
$$

Figure 6. Horn clause encoding for thread modularity at level k (where (ℓ_i, s, ℓ_i') and $(\ell^\dagger, s, \cdot)$ refer to statement s on a from ℓ_i to ℓ_i' and, respectively, from $\ell^\dagger$ to some other location in the control flow graph)

Hoenicke et al. Thread Modularity at Many Levels. POPL'17

$$Init(i, j, \overline{v}) \wedge Init(j, i, \overline{v}) \wedge$$

$$Init(i, i, \overline{v}) \wedge Init(j, j, \overline{v}) \Rightarrow I_2(i, j, \overline{v})$$

$$I_2(i, j, \overline{v}) \wedge Tr(i, \overline{v}, \overline{v}') \Rightarrow I_2(i, j, \overline{v}') \tag{3}$$

$$I_2(i, j, \overline{v}) \wedge Tr(j, \overline{v}, \overline{v}') \Rightarrow I_2(i, j, \overline{v}') \tag{4}$$

$$I_2(i, j, \overline{v}) \wedge I_2(i, k, \overline{v}) \wedge I_2(j, k, \overline{v}) \wedge$$

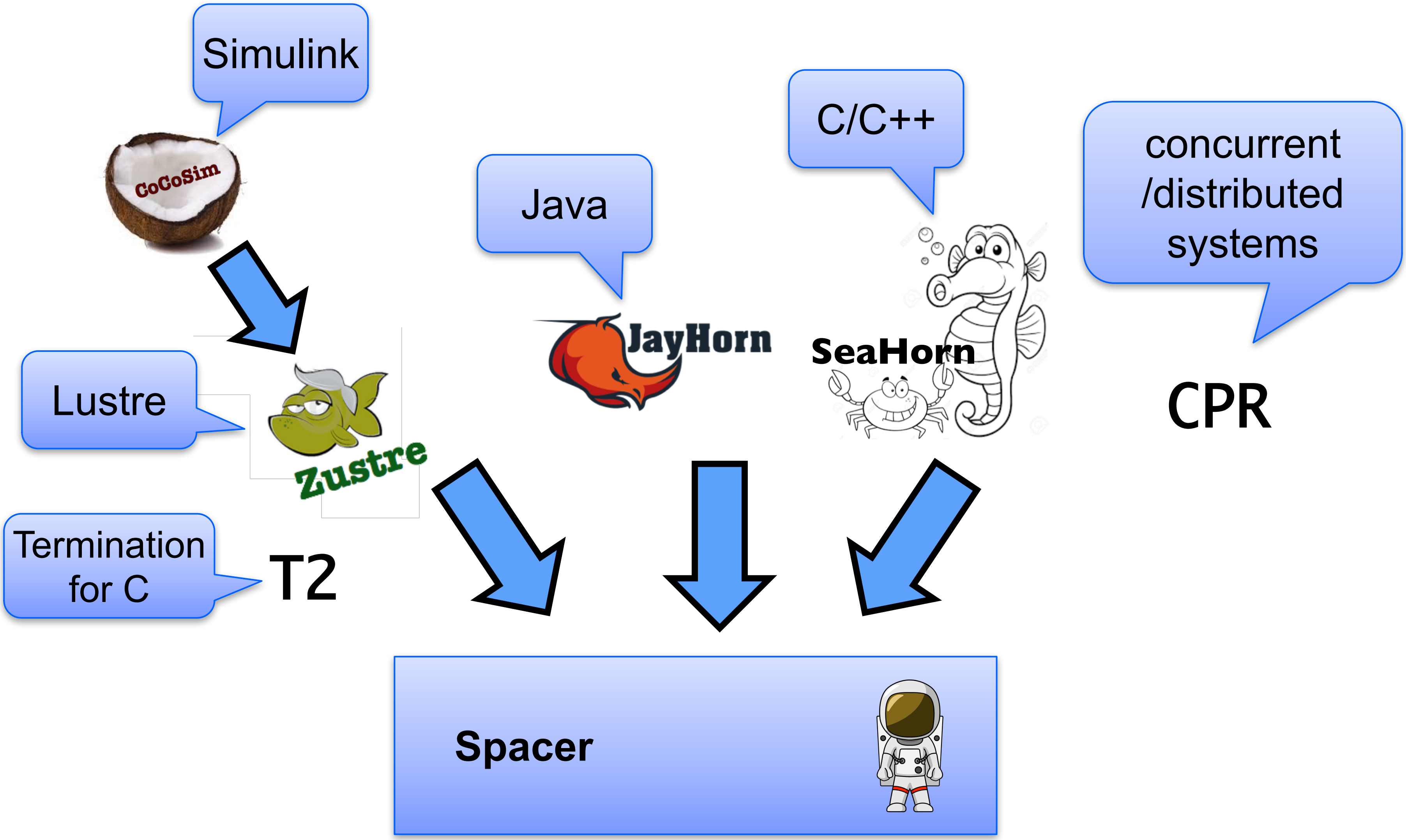
$$Tr(k, \overline{v}, \overline{v}') \wedge k \neq i \wedge k \neq j \Rightarrow I_2(i, j, \overline{v}') \tag{5}$$

$$I_2(i, j, \overline{v}) \Rightarrow \neg Bad(i, j, \overline{v})$$

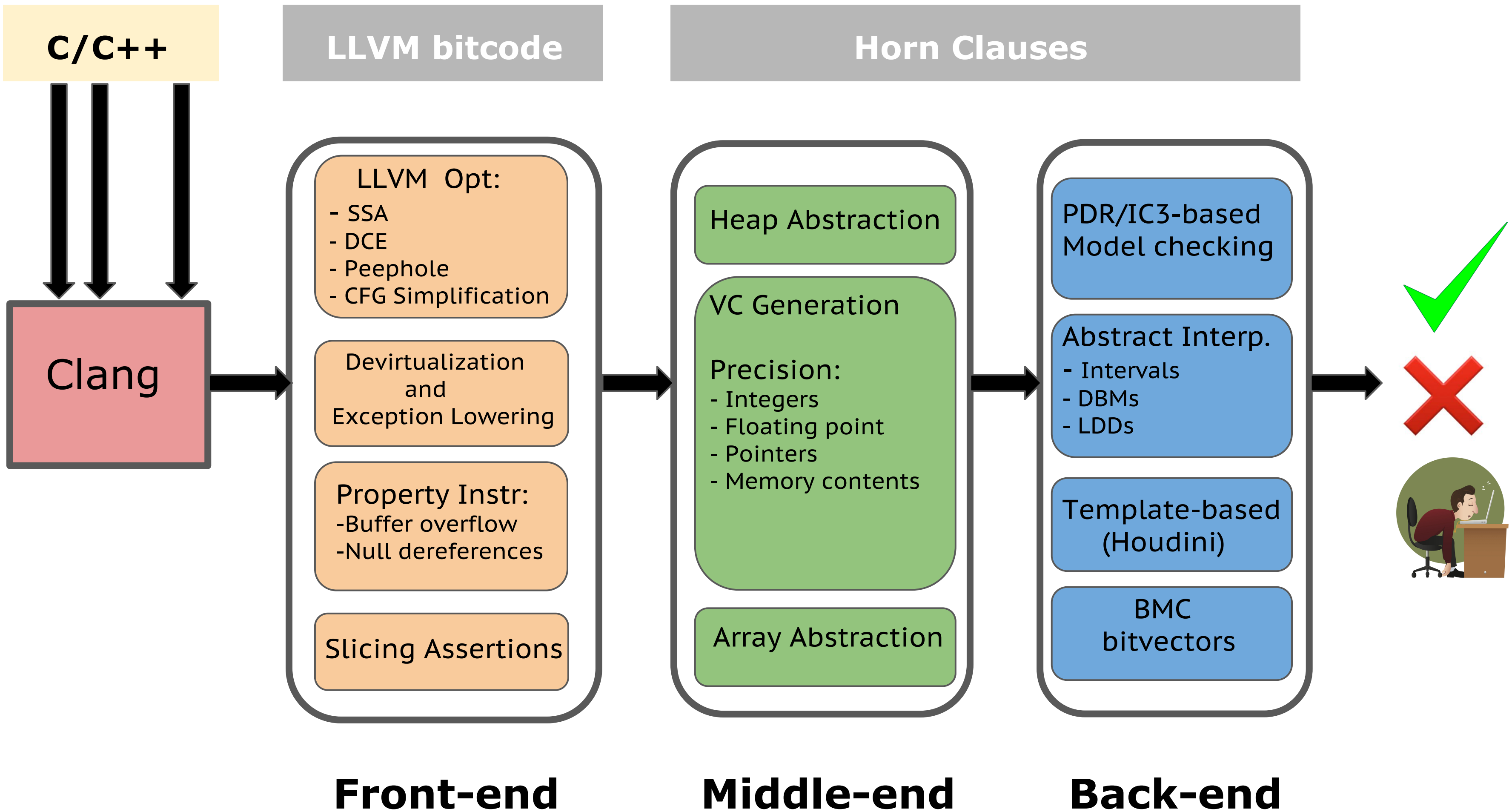
Figure 3: $VC_2(T)$ for two-quantifier invariants.

Gurfinkel et al. SMT-Based Verification of Parameterized Systems. FSE 2016

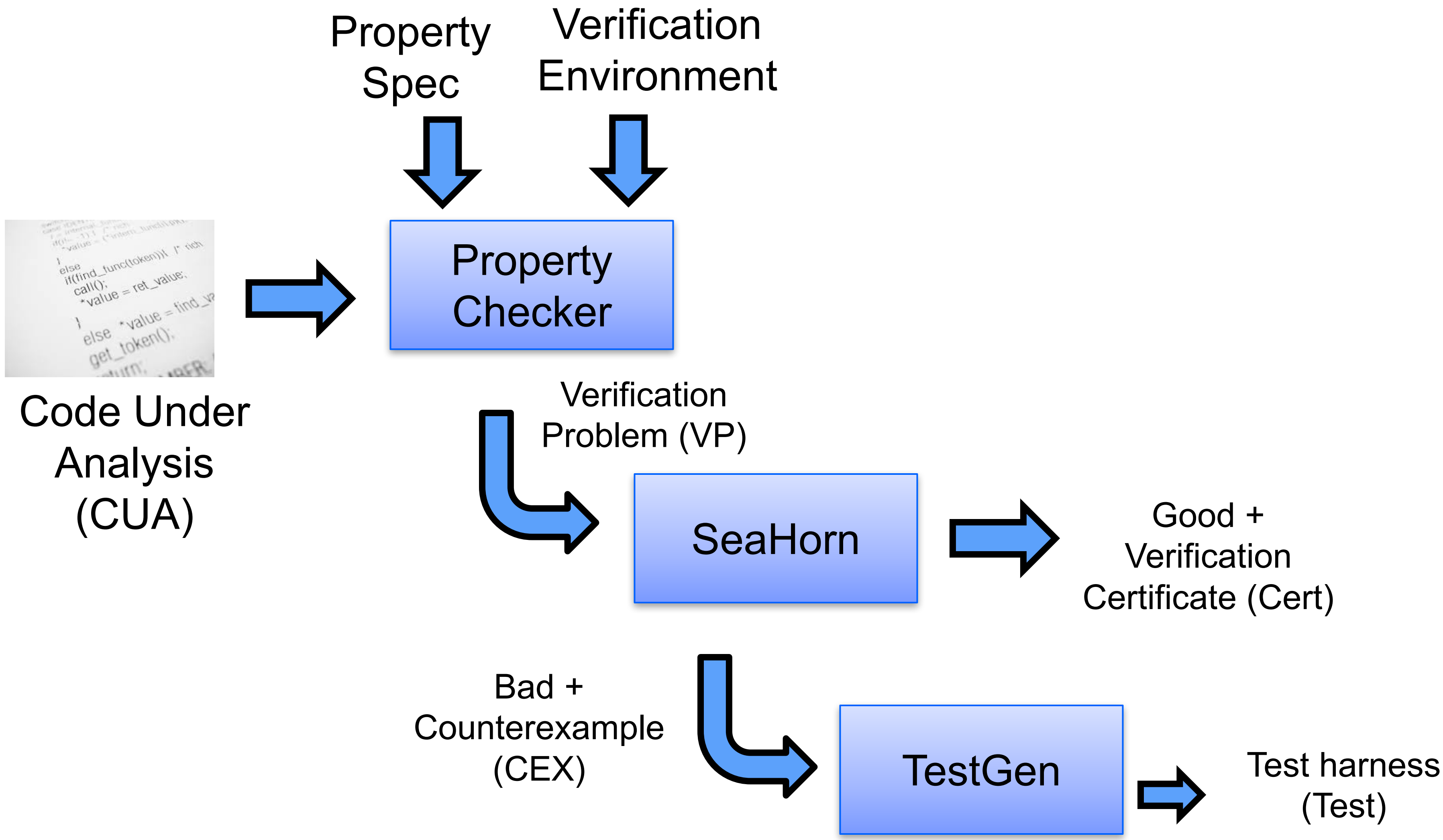
Logic-based Algorithmic Verification



Architecture of Seahorn



SeaHorn Workflow



SeaHorn workflow components

Code Under Analysis (CUA)

- code being analyzed. Device driver, component, library, etc.

Verification environment

- stubs for the environment with which CUA interacts
- e.g., libc, memcpy, malloc, OS system calls, user input, socket, file, …

Property Checker

- static instrumentation of a program with a monitor that indicates when an error has happened
- similar to dynamic sanitizers, but can use verifier-specific API to perform symbolic actions
- property spec is specific to a property checker

Verification Problem

- a prepared instance of program with embedded assertions, potentially simplified by abstracting away irrelevant parts of execution

Test Gen

- generates a test harness that includes all stubs and stimuli to guide CUA to a property failure discovered by the verifier

Developing a Static Property Checker

A static property checker is similar to a dynamic checker

- e.g., clang sanitizer (address, thread, memory, etc.)

A significant development effort for each new property

- new specialized static analyses to rule out trivial cases
- different instrumentations have affect on performance

Developed by a domain expert

- understanding of verification techniques is useful (but not required)
- 3-6 month effort for a new property
  - but many things can be reused between similar properties
  - e.g., memory safety, null-dereference, taint checking, use-after-free, etc.

SeaHorn property checkers:

- memory safety (out of bound uses, null pointer)
  - ongoing work to improve scalability and usability
- taint analysis (being developed by Princeton)

Classical Memory Models for C/C++

- Byte-level model: a large array of bytes and every allocation returns a new offset in that array

$$\mathrm{Ptr} = \mathrm{Int} \qquad \mathit{Mem} : \mathrm{Ptr} \to \mathrm{Byte}$$

- Untyped Block-level model: a pointer is a pair $\langle \mathit{ref}, o \rangle$ where *ref* uniquely defines a memory object and *o* defines the byte in the object being point to

$$\mathrm{Ptr} = \mathrm{Ref} \times \mathrm{Int} \qquad \mathit{Mem} : \mathrm{Ptr} \to \mathrm{Ptr}$$

- Typed Block-level model: refines the block-level model by having a separate block for each distinct type:

$$\mathrm{Ptr} = \mathrm{Ref} \times \mathrm{Int} \qquad \mathit{Mem} : \mathrm{Type} \times \mathrm{Ptr} \to \mathrm{Ptr}$$

Classical Memory Models for C/C++

- Byte-level model: a large array of bytes and every allocation returns a new offset in that array

$$\text{Ptr} = \text{Int} \qquad \mathit{Mem} : \text{Ptr} \to \text{Byte}$$

- Untyped Block-level model: a pointer is a pair $\langle \mathit{ref}, o \rangle$ where *ref* uniquely defines a memory object and *o* defines the byte in the object being point to

$$\text{Ptr} = \text{Ref} \times \text{Int} \qquad \mathit{Mem} : \text{Ptr} \to \text{Ptr}$$

- Typed Block-level model: refines the block-level model by having a separate block for each distinct type:

$$\text{Ptr} = \text{Ref} \times \text{Int} \qquad \mathit{Mem} : \text{Type} \times \text{Ptr} \to \text{Ptr}$$

Byte-Level vs Block-level Memory Model

- Let $\mathcal{P}$ be a property of an array segment $\{\mathtt{A+l},\ldots,\mathtt{A+h}\}$
- Let q be a pointer that is is disjoint from $\{\mathtt{A+l},\ldots,\mathtt{A+h}\}$
- Show that $\mathcal{P}$ is true of $\{\mathtt{A+l},\ldots,\mathtt{A+h}\}$ after $*\mathtt{q} = 5$
- Using byte-level model:

$$\mathcal{P}(M_0,\mathtt{l},\mathtt{h}) \wedge \mathit{disjoint}(\mathtt{q},\mathtt{l},\mathtt{h}) \wedge M_1 = \mathtt{store}(M_0,\mathtt{q},5) \Rightarrow \mathcal{P}(M_1,\mathtt{l},\mathtt{h})$$

  where

  - $\mathit{disjoint}(q,l,h) = q + \mathrm{size}(q) \leq l \vee q \geq h$
  - auxiliary lemma:

$$\forall M_i, M_j \in \mathrm{Mem}, \forall a,b,x \in \mathrm{Int}.(a \leq x \leq b \wedge M_i[x] = M_j[x]) \Rightarrow (\mathcal{P}(M_i,a,b) \Rightarrow \mathcal{P}(M_j,a,b))$$

- Using block-level model:

$$\mathcal{P}(B^A,\mathtt{l},\mathtt{h}) \wedge B^q_1 = \mathtt{store}(B^q_0,\mathtt{q},5) \Rightarrow \mathcal{P}(B^A,\mathtt{l},\mathtt{h})$$

Our Memory Model

- Untyped block-level based memory model
- Memory objects are infinitely apart from each other
- Implicit separation given by distinctness of block references
- Cover a relevant subset of C/C++ programs that supports:
  - dynamic memory allocation
  - type unions, pointer arithmetic, pointer casts
  - inheritance, function/method calls, etc

From Pointer Analysis to Verification Conditions

- Run a pointer analysis to disambiguate memory
- Produce a side-effect-free encoding by:
  - Replacing each memory object o to a logical array A_o
  - Replacing memory accesses to a pointer p (within object o) to array reads and writes over A_o
  - Each array write on A_o produces a new version of A'_o representing the array after the execution of the memory write
- Logical arrays are unbounded and the "whole array" is updated in its entirety:
  - $A[1] = 5 \;\rightarrow\; A_1 = \lambda i : i = 1 \;?\; 5 \;:\; A_0$
  - $A[k] = 7 \;\rightarrow\; A_2 = \lambda i : i = k \;?\; 7 \;:\; A_1$

Pointer Analysis with C++ Inheritance

```
class X {
   X() {....}
};

class Y: public X {
   Y(): X() {....}
};

class Z: public X {
   Z(): X() {....}
};

Y* y = new Y();
Z* z = new Z();
```

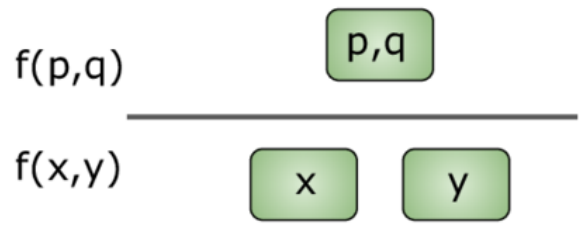
```
% Constructor for Y
_Y_c(this) {
   _X_c(this);
   ...
}
% Constructor for Z
_Z_c(this) {
   _X_c(this);
   ...
}

% Y y = new Y();
y = _Znwm(sizeof(Y));
_Y_c(y);

% Z z = new Z();
z = _Znwm(sizeof(Z));
_Z_c(z);
```

UNIVERSITY OF WATERLOO

VCs Using a Context-Insensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias



VCs Using a Context-Insensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias

f(p,q)

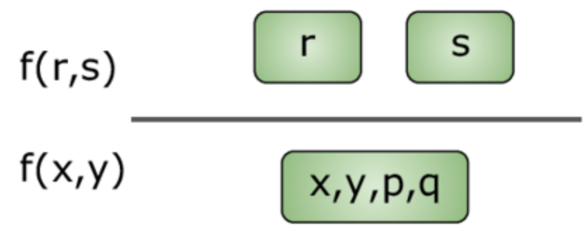
f(x,y)

x,y,p,q

VCs Using a Context-Insensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
        int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias



VCs Using a Context-Insensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias

f(r,s)

x,y,p,q,r,s

f(x,y)

VCs Using a Context-Insensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Verification conditions:

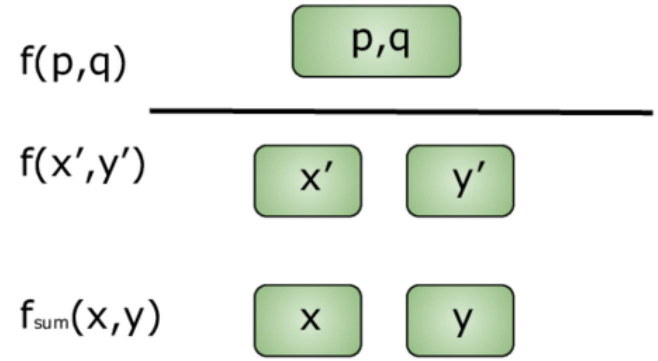
$$
\begin{array}{l}
f(x, y, A_{xy}, A''_{xy})\{ \\
\quad A'_{xy} = \mathtt{store}(A_{xy}, x, 1) \\
\quad A''_{xy} = \mathtt{store}(A'_{xy}, y, 2) \\
\}
\end{array}
$$

$$
\begin{array}{l}
g(p, q, r, s, A_{pqrs}, A''_{pqrs})\{ \\
\quad f(p, q, A_{pqrs}, A'_{pqrs}) \\
\quad f(r, s, A'_{pqrs}, A''_{pqrs}) \\
\}
\end{array}
$$

VCs Using a Context-Sensitive Pointer Analysis

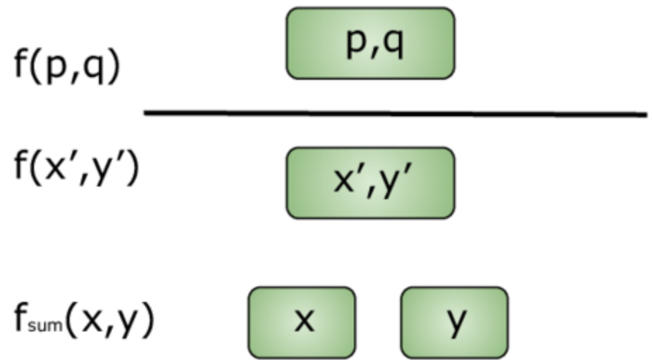
```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias



```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

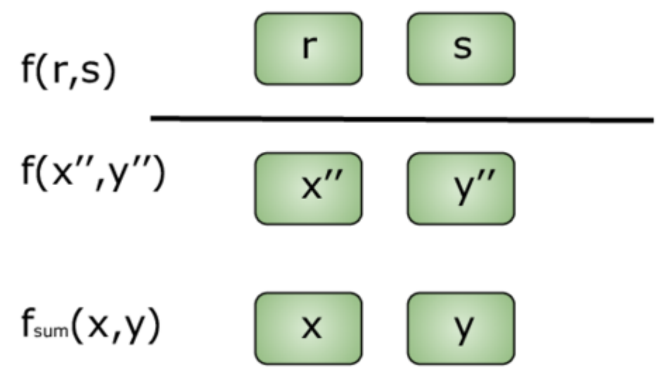
Assume p and q may alias



VCs Using a Context-Sensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias



VCs Using a Context-Sensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
         int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Verification conditions:

$f(x, y, A_x, A_y, A'_x, A'_y)\{$
$\quad A'_x = \mathtt{store}(A_x, x, 1)$
$\quad A'_y = \mathtt{store}(A_y, y, 2)$
$\}$

$g(p, q, r, s, A_{pq}, A_r, A_s, A'_{pq}, A'_r, A'_s)\{$
$\quad f(p, q, A_{pq}, A_{pq}, A'_{pq}, A'_{pq})$
$\quad f(r, s, A_r, A_s, A'_r, A'_s)$
$\}$

VCs Using a Context-Sensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Verification conditions:

$f(x, y, A_x, A_y, A'_x, A'_y)\{$
$\quad A'_x = \mathtt{store}(A_x, x, 1)$
$\quad A'_y = \mathtt{store}(A_y, y, 2)$
$\}$

$g(p, q, r, s, A_{pq}, A_r, A_s, A'_{pq}, A'_r, A'_s)\{$
$\quad f(p, q, A_{pq}, A_{pq}, A'_{pq}, A'_{pq})$
$\quad f(r, s, A_r, A_s, A'_r, A'_s)$
$\}$

A direct VC encoding is unsound:

First call to f: $A'_{pq} = \mathtt{store}(A_{pq}, p, 1)$ and $A'_{pq} = \mathtt{store}(A_{pq}, q, 2)$

The update of `p` is lost!

UNIVERSITY OF WATERLOO

Ensuring Sound VCs using a CS Pointer Analysis

- Arbitrary CS pointer analysis cannot be directly leveraged for modular verification

- They must satisfy this Correctness Condition (CC):

  "No two disjoint memory objects modified in a function can be aliased at any particular call site"

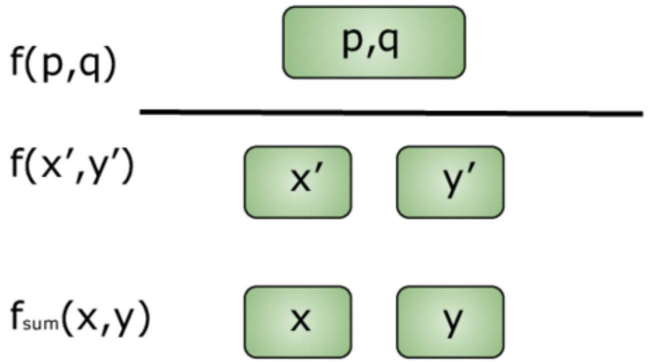
- Observed by Reynolds'78, Moy's PhD thesis'09, and many others

- Proposed solutions:
  - ignore context-sensitivity: SMACK and Cascade
  - generate contracts that ensure CC holds, otherwise reject programs: Frama-C + Jessie plugin

Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

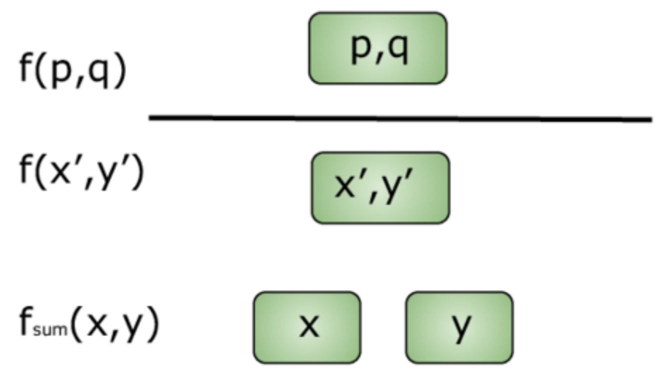
Assume p and q may alias



Ensuring Sound Modular VC Generation: Our Solution

Assume p and q may alias

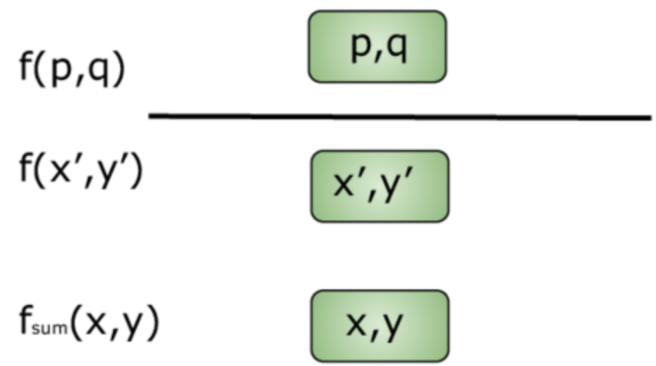
```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```



Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
        int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

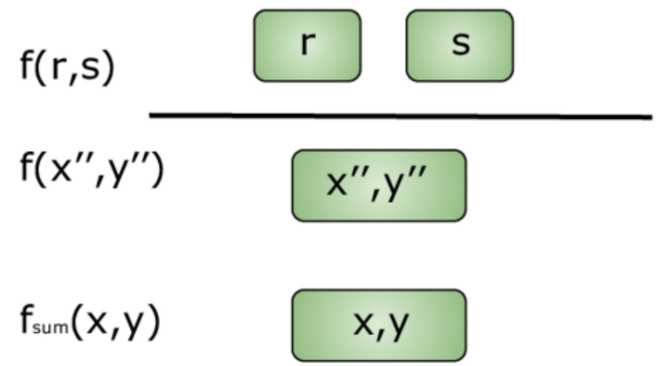
Assume p and q may alias



Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
         int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

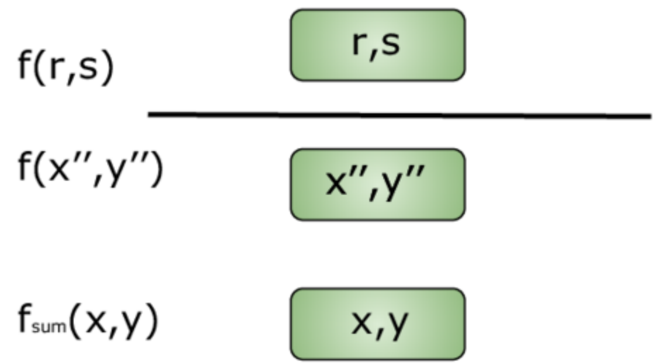
Assume p and q may alias



Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
        int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias



Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Sound verification conditions:

$$
\begin{array}{l}
f(x, y, A_{xy}, A''_{xy})\{ \\
\quad A'_{xy} = \mathtt{store}(A_{xy}, x, 1) \\
\quad A''_{xy} = \mathtt{store}(A'_{xy}, y, 2) \\
\} \\
g(p, q, r, s, A_{pq}, A_{rs}, A'_{pq}, A'_{rs})\{ \\
\quad f(p, q, A_{pq}, A'_{pq}) \\
\quad f(r, s, A_{rs}, A'_{rs}) \\
\}
\end{array}
$$

Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Sound verification conditions:

$$
\begin{array}{l}
f(x, y, A_{xy}, A''_{xy})\{ \\
\quad A'_{xy} = \mathtt{store}(A_{xy}, x, 1) \\
\quad A''_{xy} = \mathtt{store}(A'_{xy}, y, 2) \\
\} \\
g(p, q, r, s, A_{pq}, A_{rs}, A'_{pq}, A'_{rs})\{ \\
\quad f(p, q, A_{pq}, A'_{pq}) \\
\quad f(r, s, A_{rs}, A'_{rs}) \\
\}
\end{array}
$$

Good compromise:

- context-sensitive: calls to f do not merge {p,q} and {r,s}
- ensure that CC holds!

Field- and Array-Sensitive Pointer Analysis

```
typedef struct list{
  struct list *n;
  int e;
} ll;

ll* mkList(int s,int e){
 if (s <= 0)
   return NULL;
 ll*p=malloc(sizeof(ll));
 p->e=e;
 p->n=mkList(s-1,e);
 return p;
}

void main(){
 ll* a[N];
 int i;
 for(i=0;i<N;++i)
   a[i] = mkList(M,0);
}
```

Our pointer analysis infers:

1. `&a[0]` points to an object O_A which has ≥ 1 elements of size of a pointer
2. O_A points to another object O_L with 0 and 4 offsets

Similar pointer analyses do not distinguish O_A from O_L

Our contributions

We present a new pointer analysis for verification of C/C++ that:

1. is context-, field-, and array-sensitive
2. has been implemented and publicly available

   `https://github.com/seahorn/sea-dsa`

3. has been evaluated on flight control components written in C++ and SV-COMP benchmarks in C

Concrete Semantics

- A concrete cell is a pair of an object reference and offset
- A concrete points-to graph $g \in \mathcal{G}_{\mathbb{C}}$ is a triple $\langle V, E, \sigma \rangle$:

$$V \subseteq \mathcal{C}_{\mathbb{C}} \quad E \subseteq \mathcal{C}_{\mathbb{C}} \times \mathcal{C}_{\mathbb{C}} \quad \sigma : \mathcal{V}_{\mathcal{P}} \mapsto \mathcal{C}_{\mathbb{C}}$$

- A concrete state is a triple $\langle g, \pi, pc \rangle$ where

$$g \in \mathcal{G}_{\mathbb{C}} \quad \pi : \mathcal{V}_{\mathcal{I}} \mapsto \mathbb{Z} \quad pc \in \mathbb{L}$$

- **malloc** returns a fresh memory object

Concrete Semantics: Assumptions

1. Freed memory is not reused:

```
int *p = (int*) malloc(..);
int *q = p;
free(p);
int *r = (int*) malloc(..)
```

it assumes that `r` cannot alias with `q`

2. It does not distinguish between valid and invalid pointers:

```
int *p = (int*) malloc(..);
free(p);
int *q = (int*) malloc(..);
if (p == q) *p=0;
```

it assumes no null dereference

UNIVERSITY OF WATERLOO

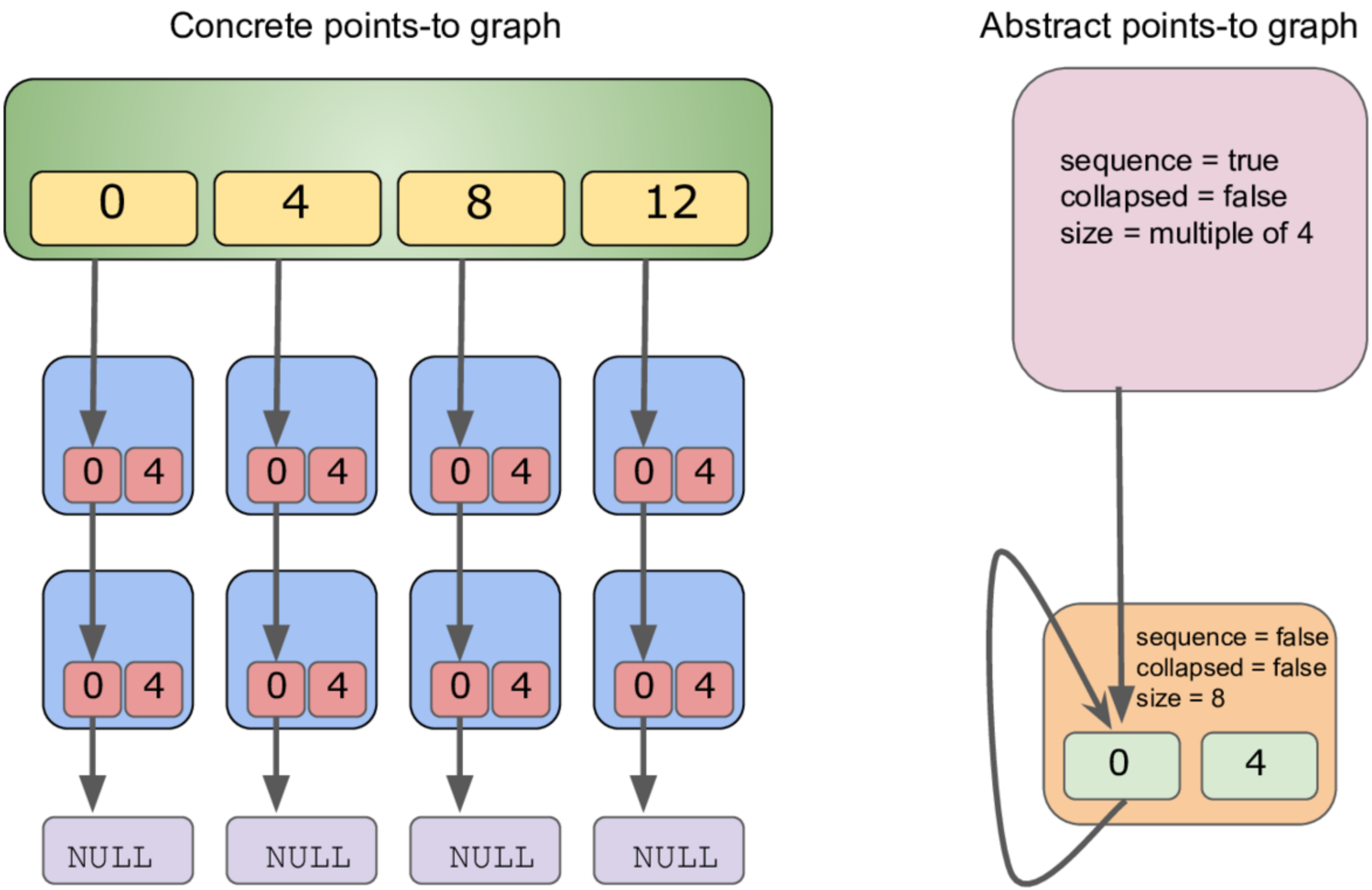
Abstract Semantics

- An abstract cell is a pair of an abstract object and byte offset
- An abstract object has an identifier and:
  1. is_sequence: unknown sequence of consecutive bytes
  2. is_collapsed: all outgoing cells have been merged
  3. size in bytes (see paper for details)
- An abstract points-to graph $\mathcal{G}_{\mathbb{A}}$ is a triple $\langle V, E, \sigma \rangle$:

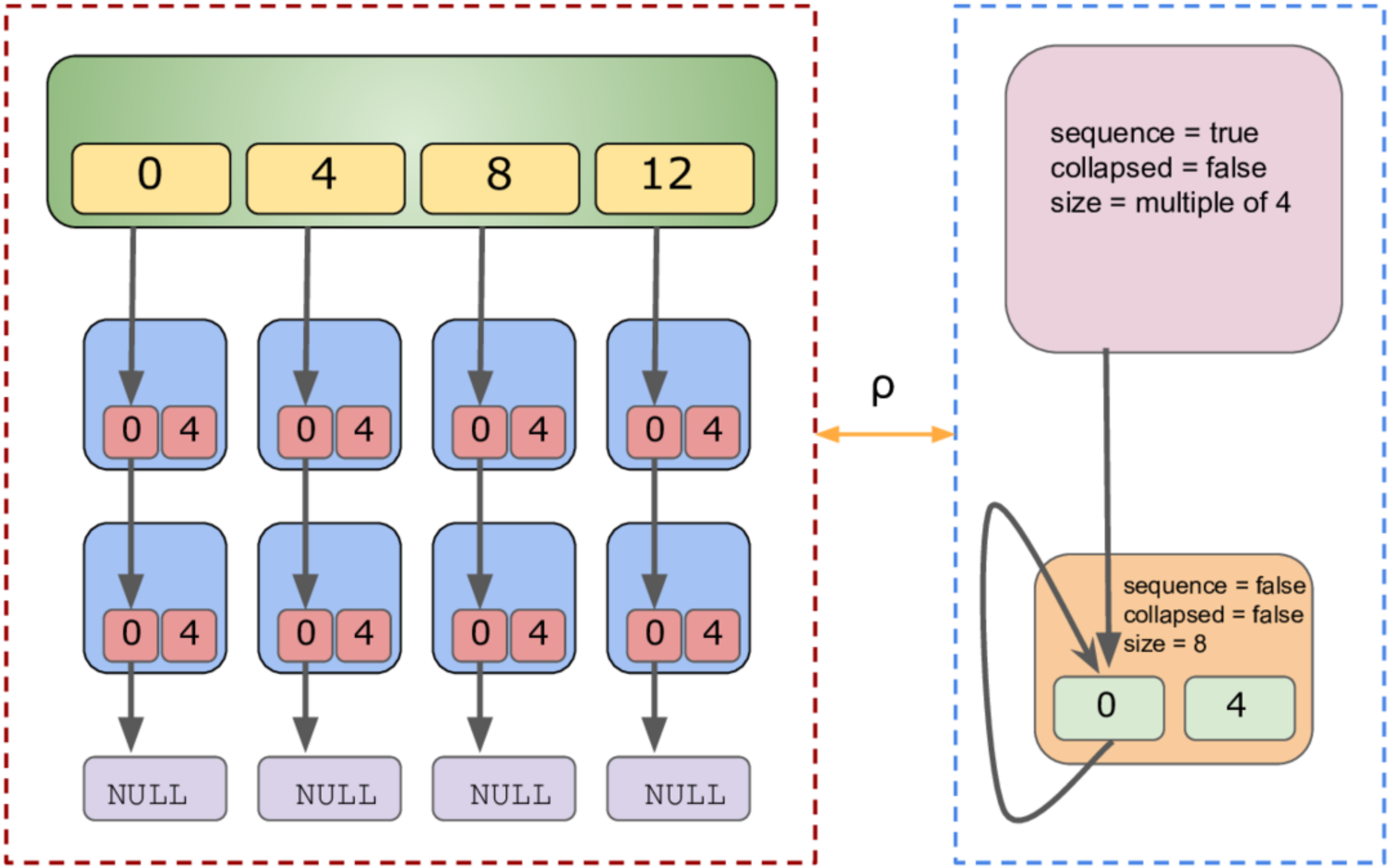
$$V \subseteq \mathcal{C}_{\mathbb{A}} \quad E \subseteq \mathcal{C}_{\mathbb{A}} \times \mathcal{C}_{\mathbb{A}} \quad \sigma : \mathcal{V}_{\mathcal{P}} \mapsto \mathcal{C}_{\mathbb{A}}$$

  The number of abstract objects is finite
- An abstract state is represented by an abstract points-to graph
  - it does not keep track of an environment for integer variables
  - it is flow-insensitive

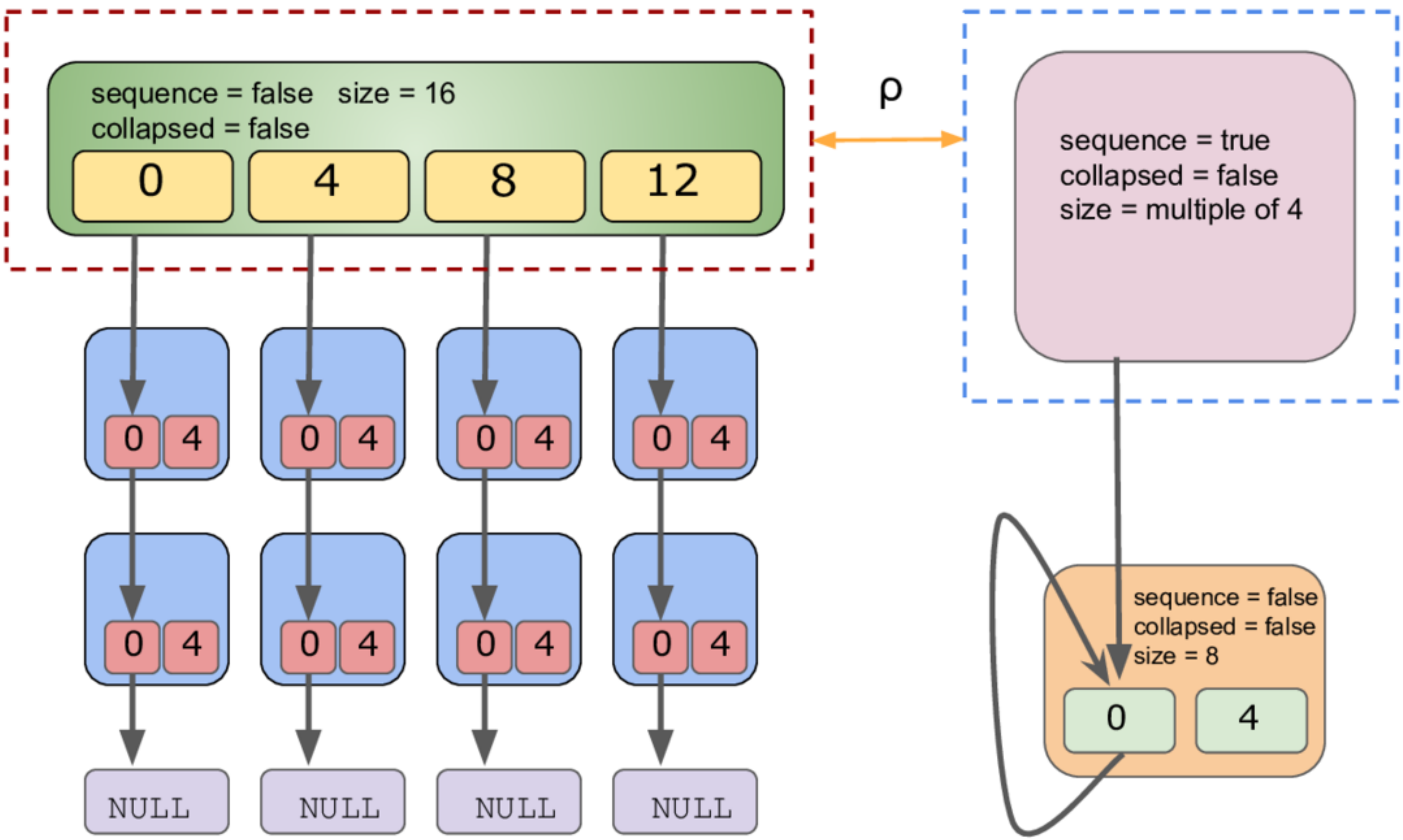
Concrete vs Abstract points-to Graphs



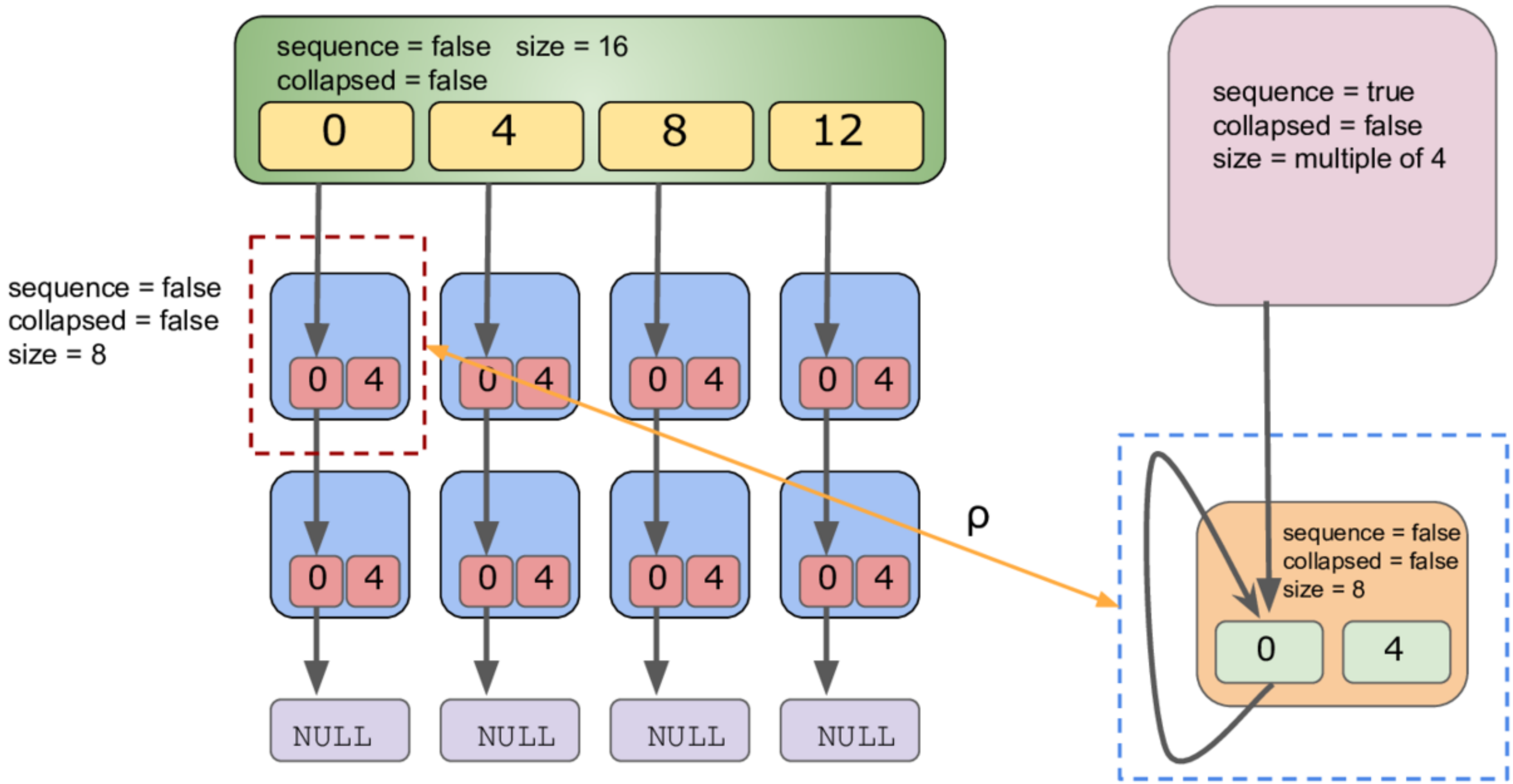
Simulation Relation between Graphs



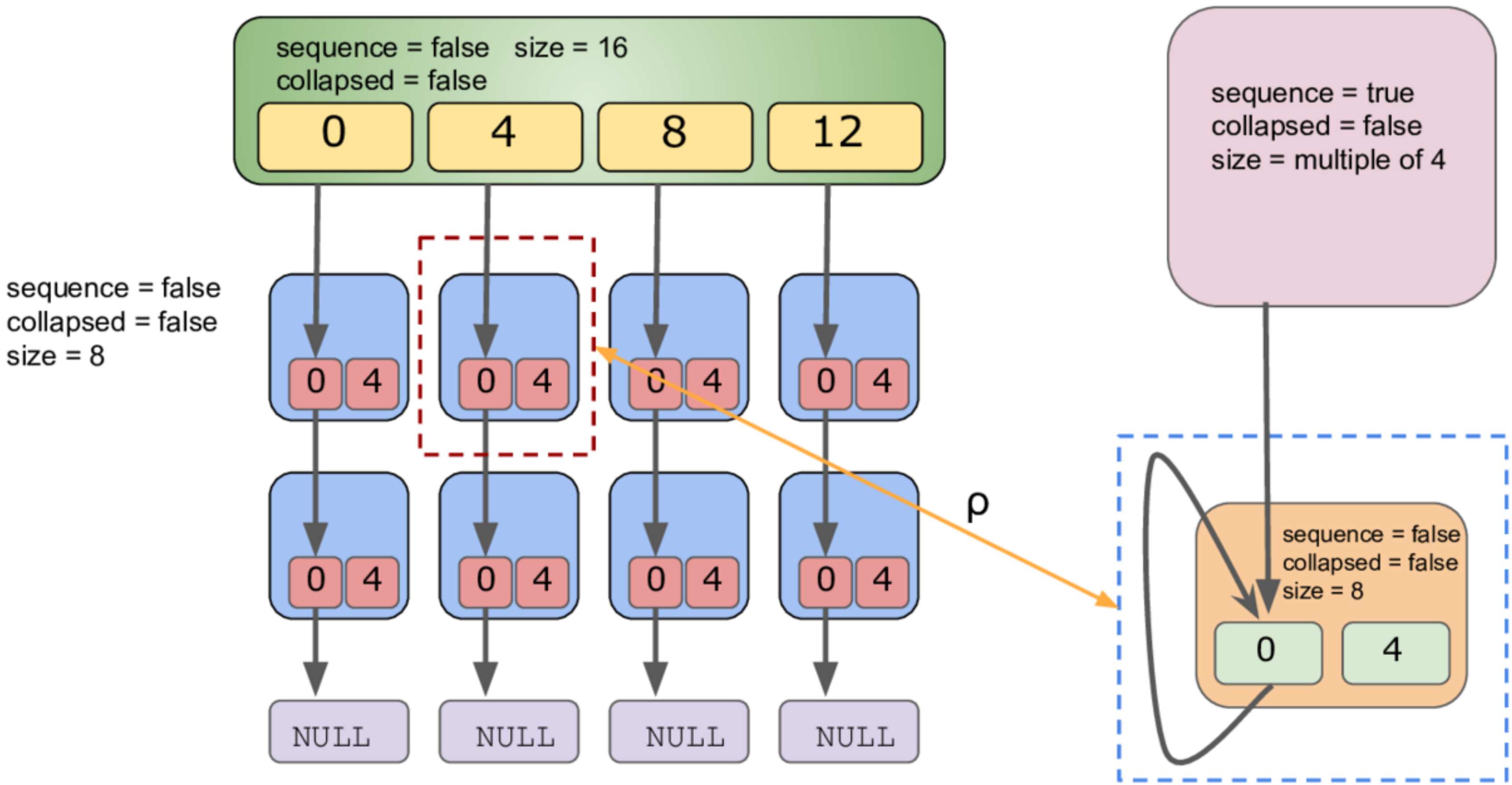
Simulation Relation between Graphs



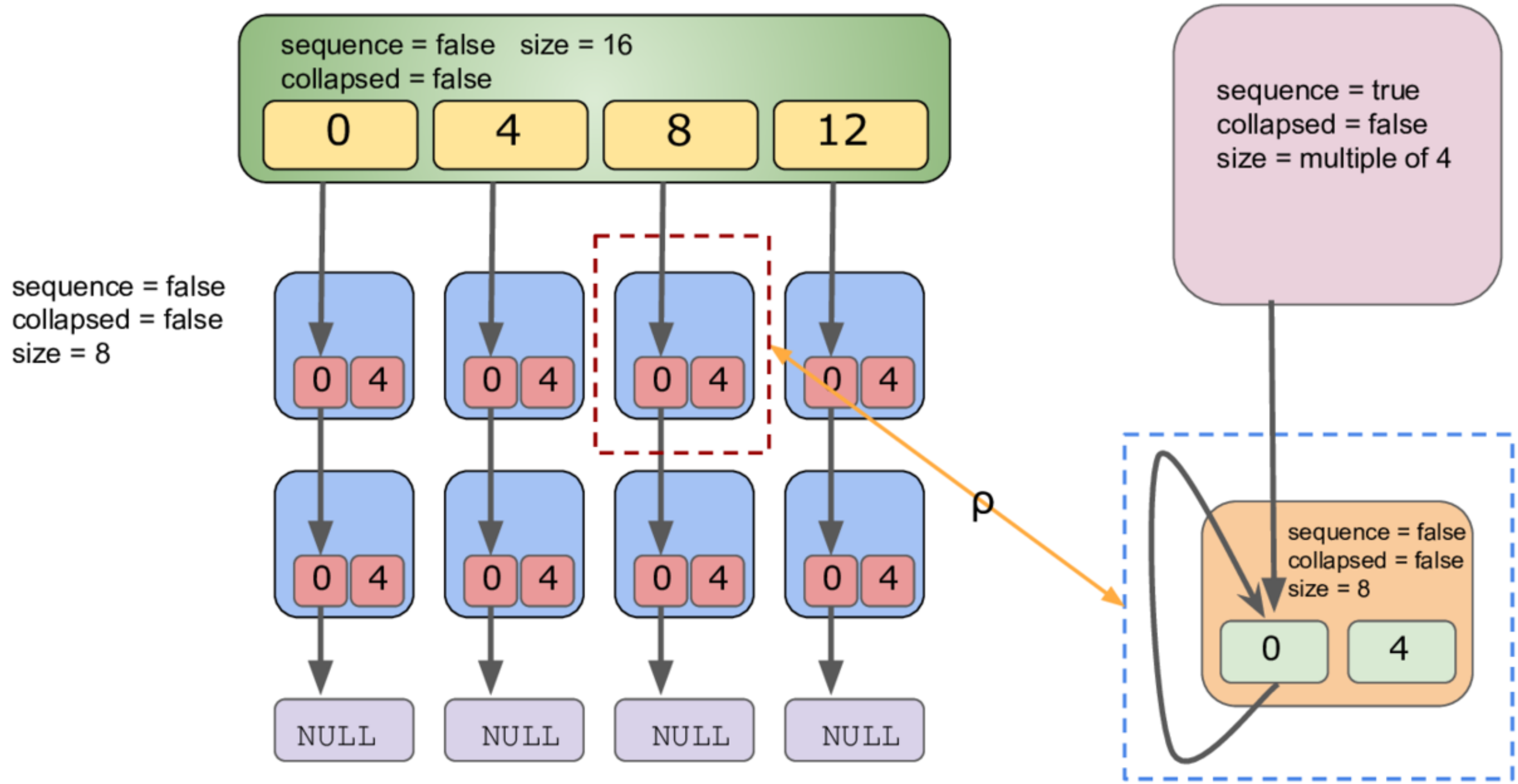
Simulation Relation between Graphs



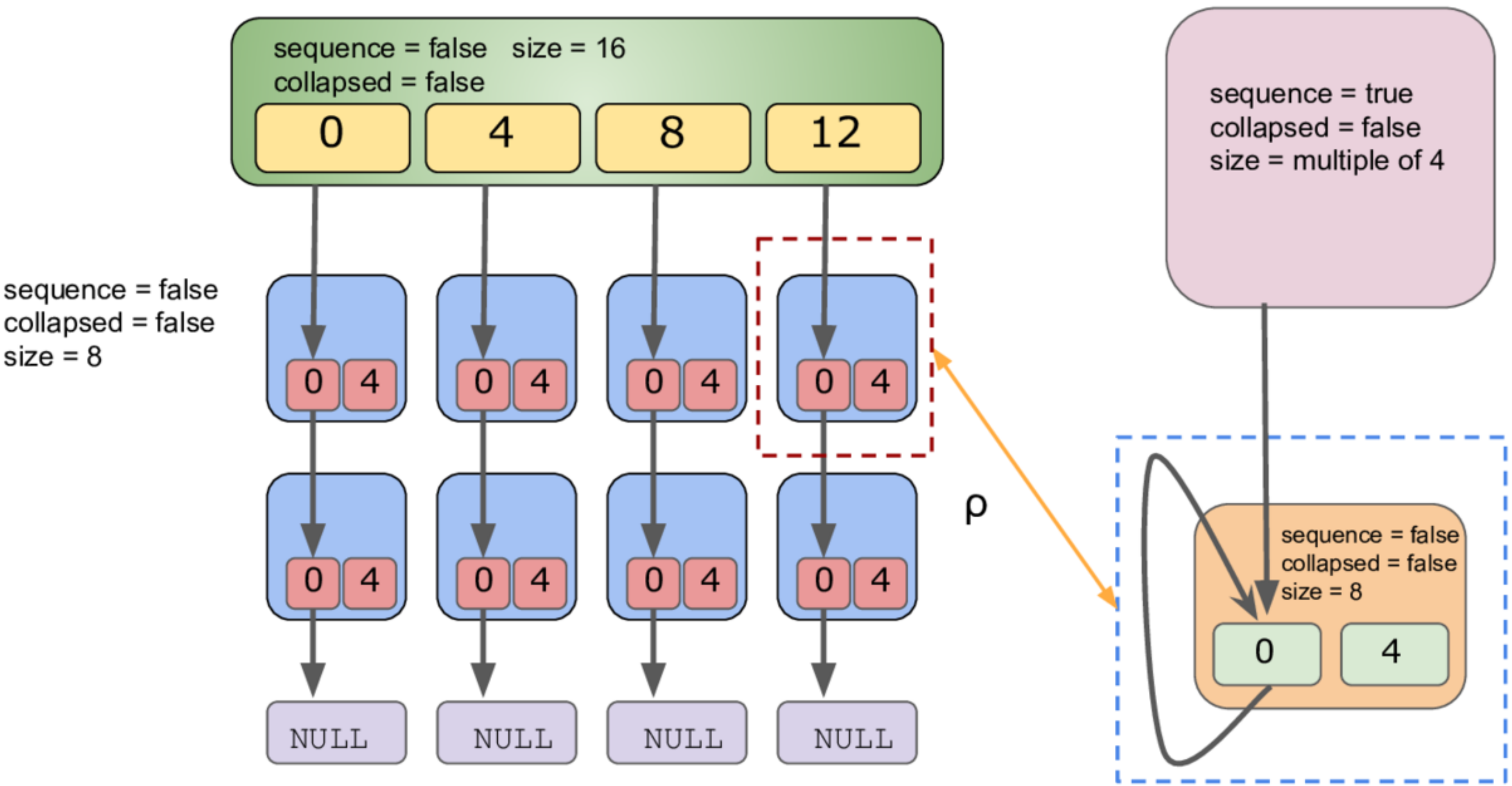
Simulation Relation between Graphs



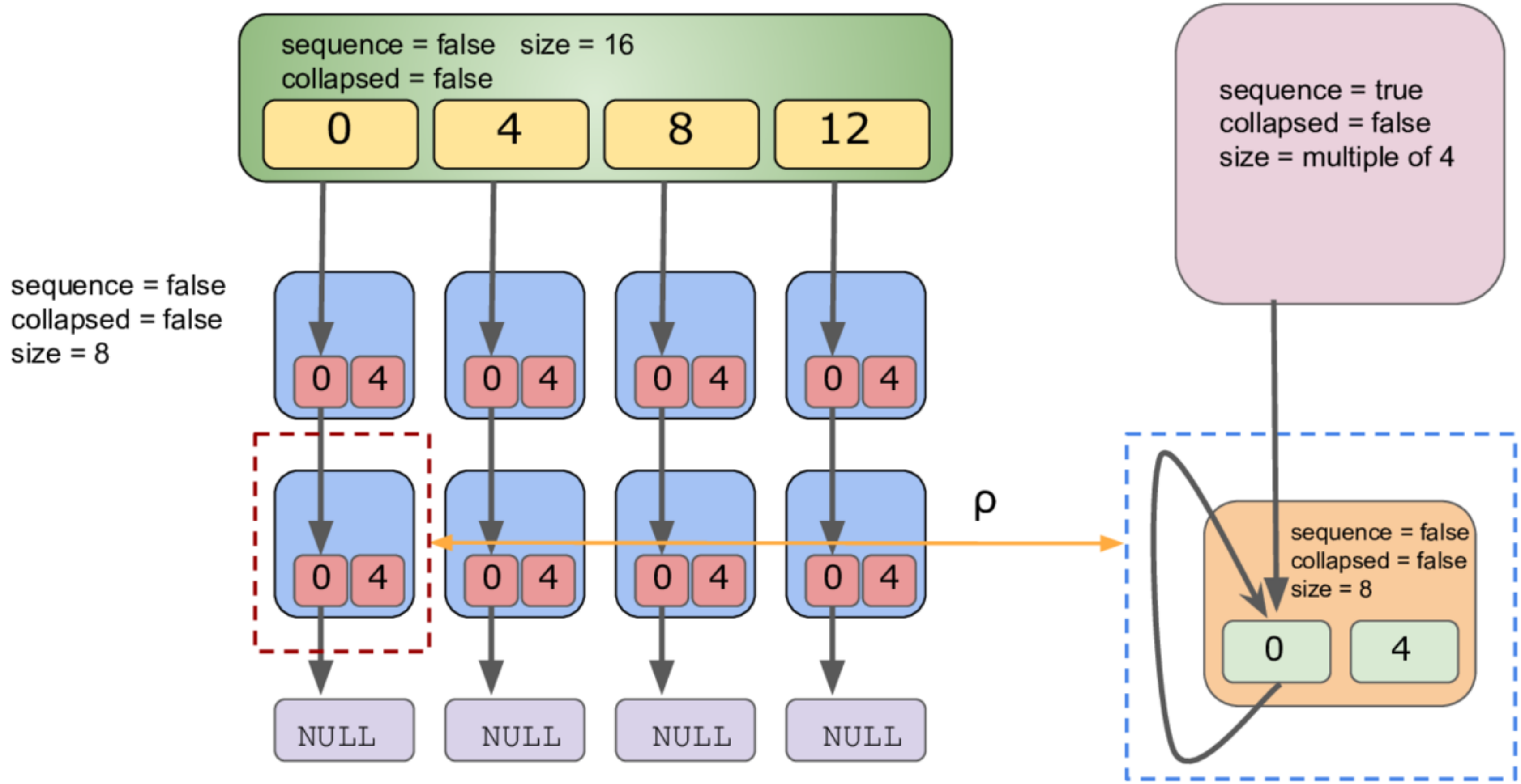
Simulation Relation between Graphs



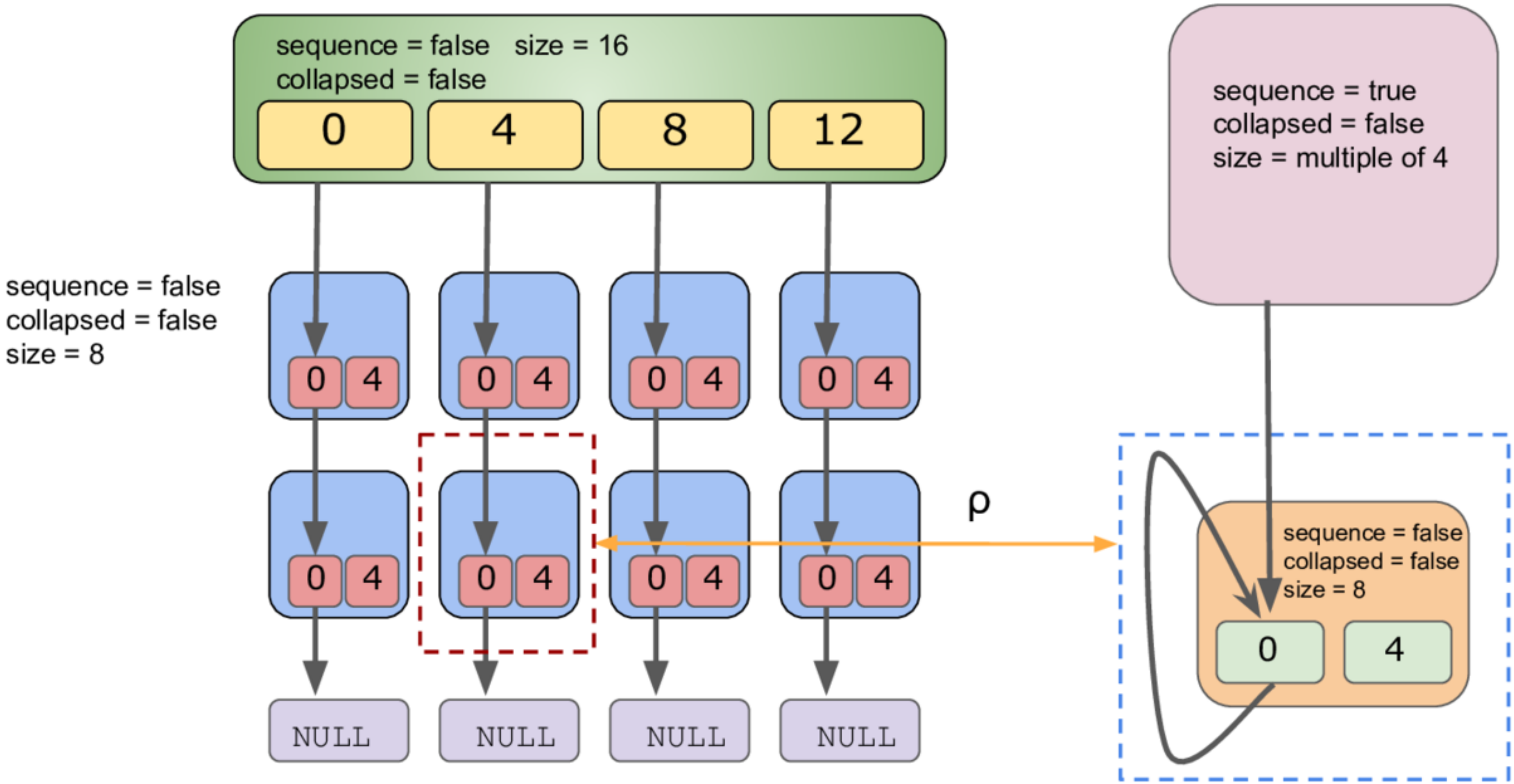
Simulation Relation between Graphs



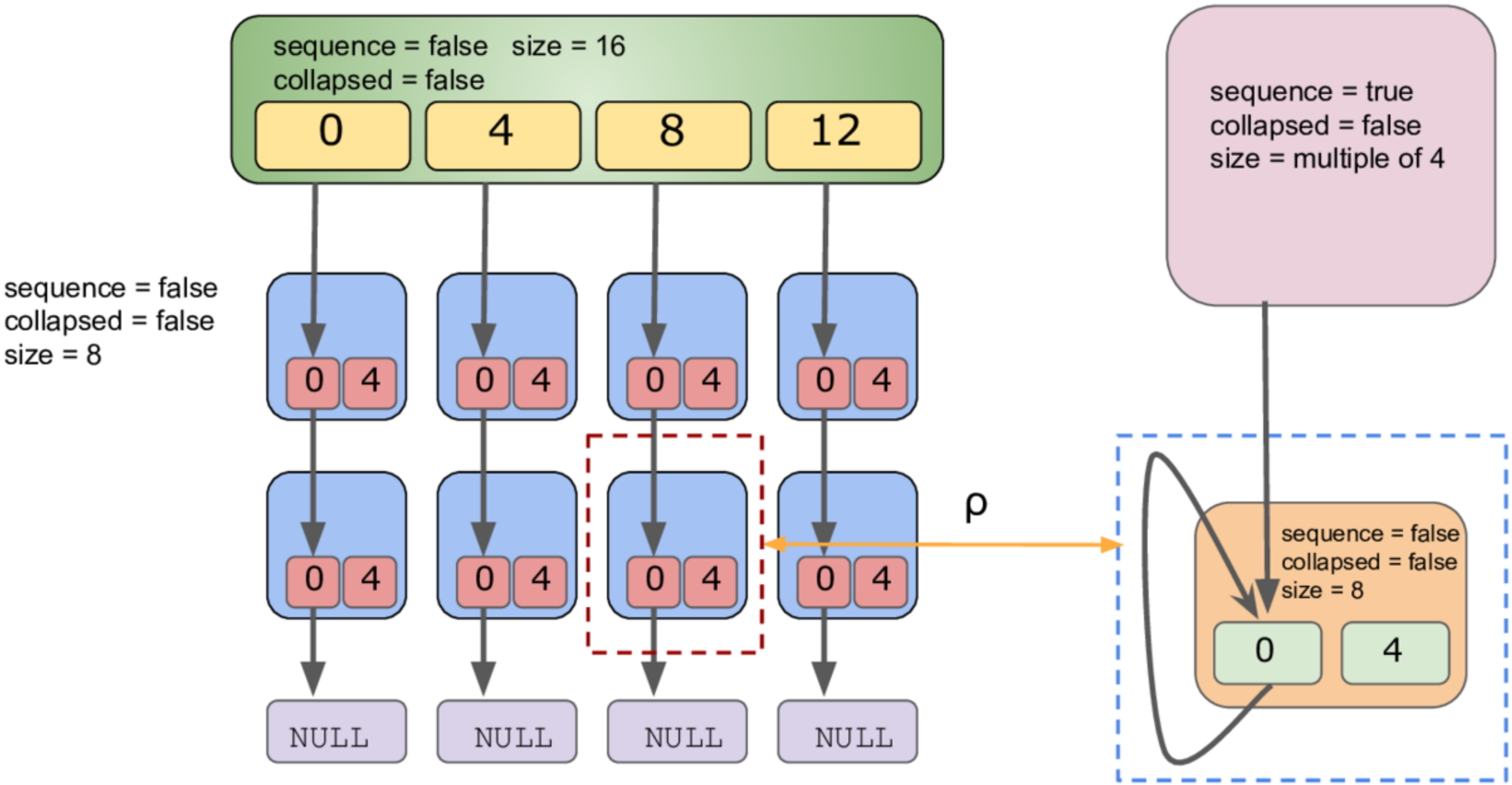
Simulation Relation between Graphs



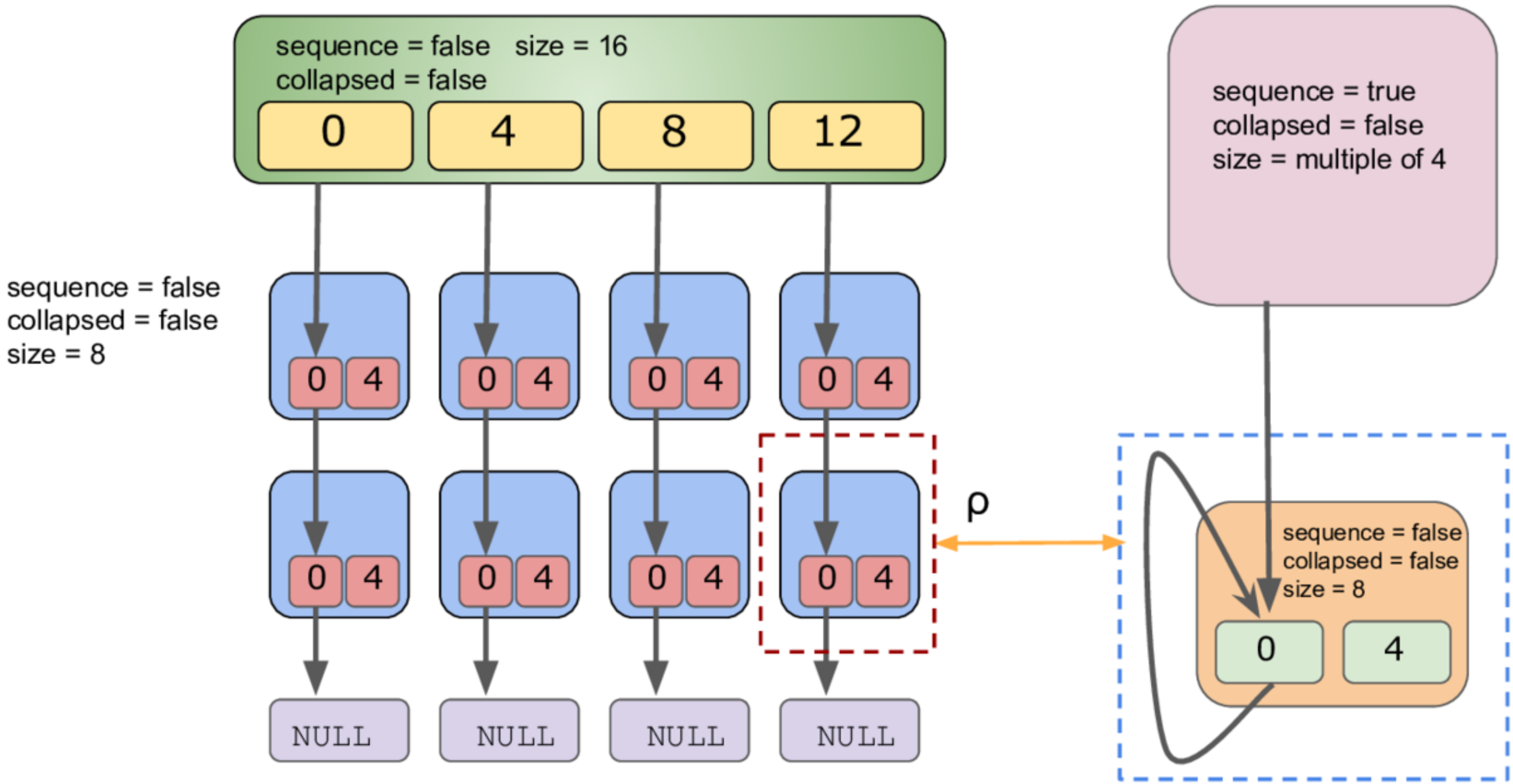
Simulation Relation between Graphs



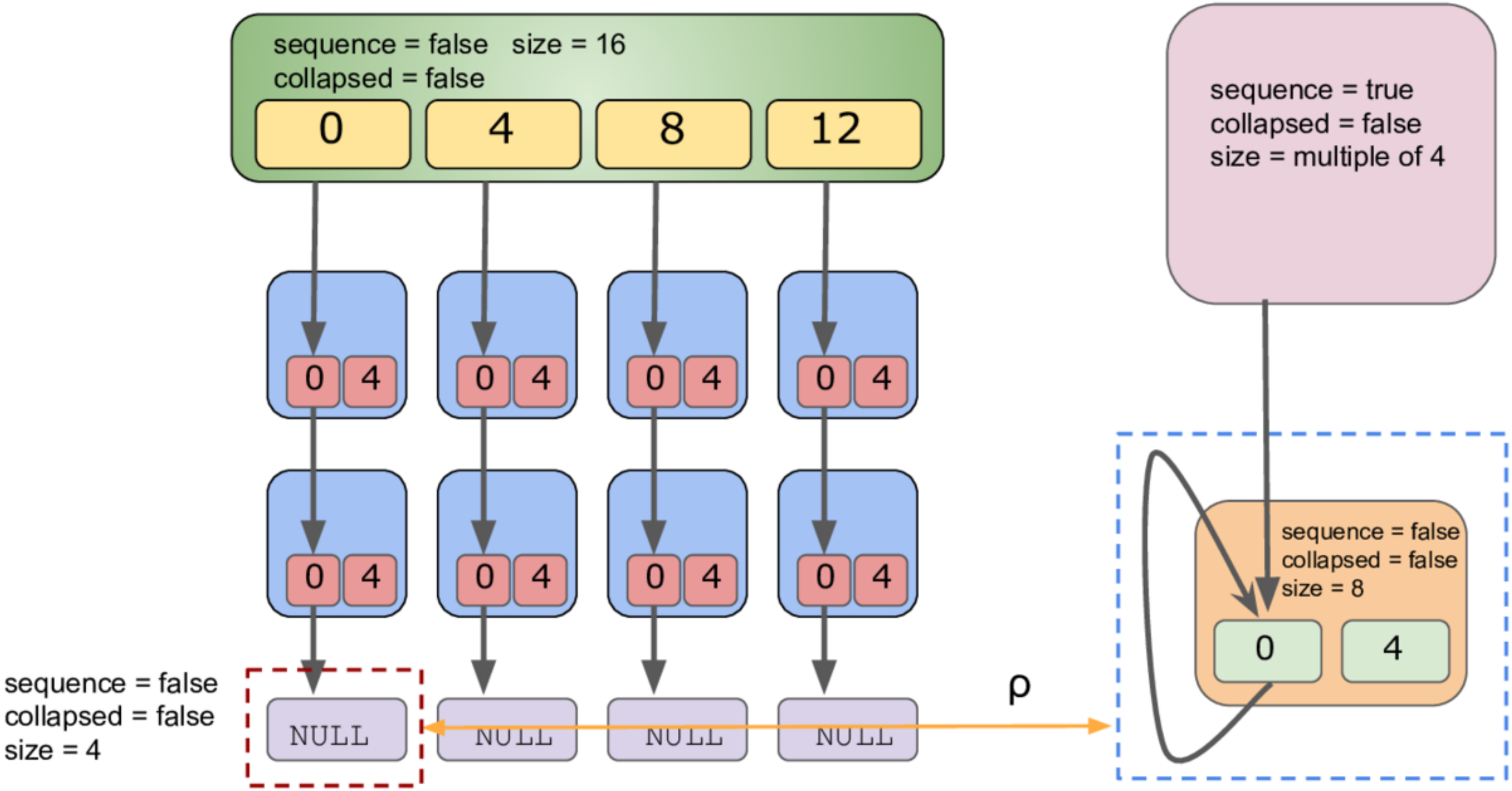
Simulation Relation between Graphs



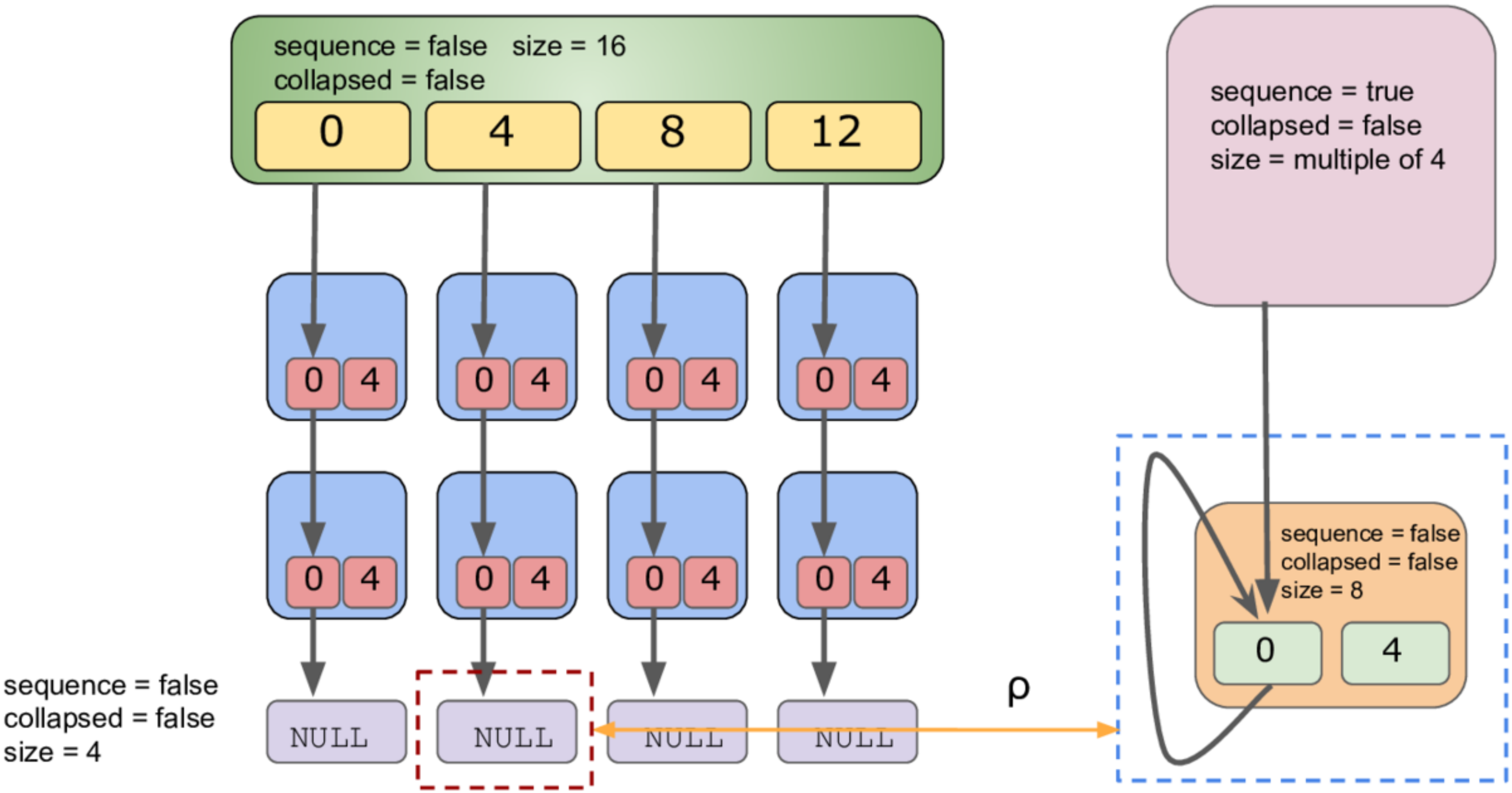
Simulation Relation between Graphs



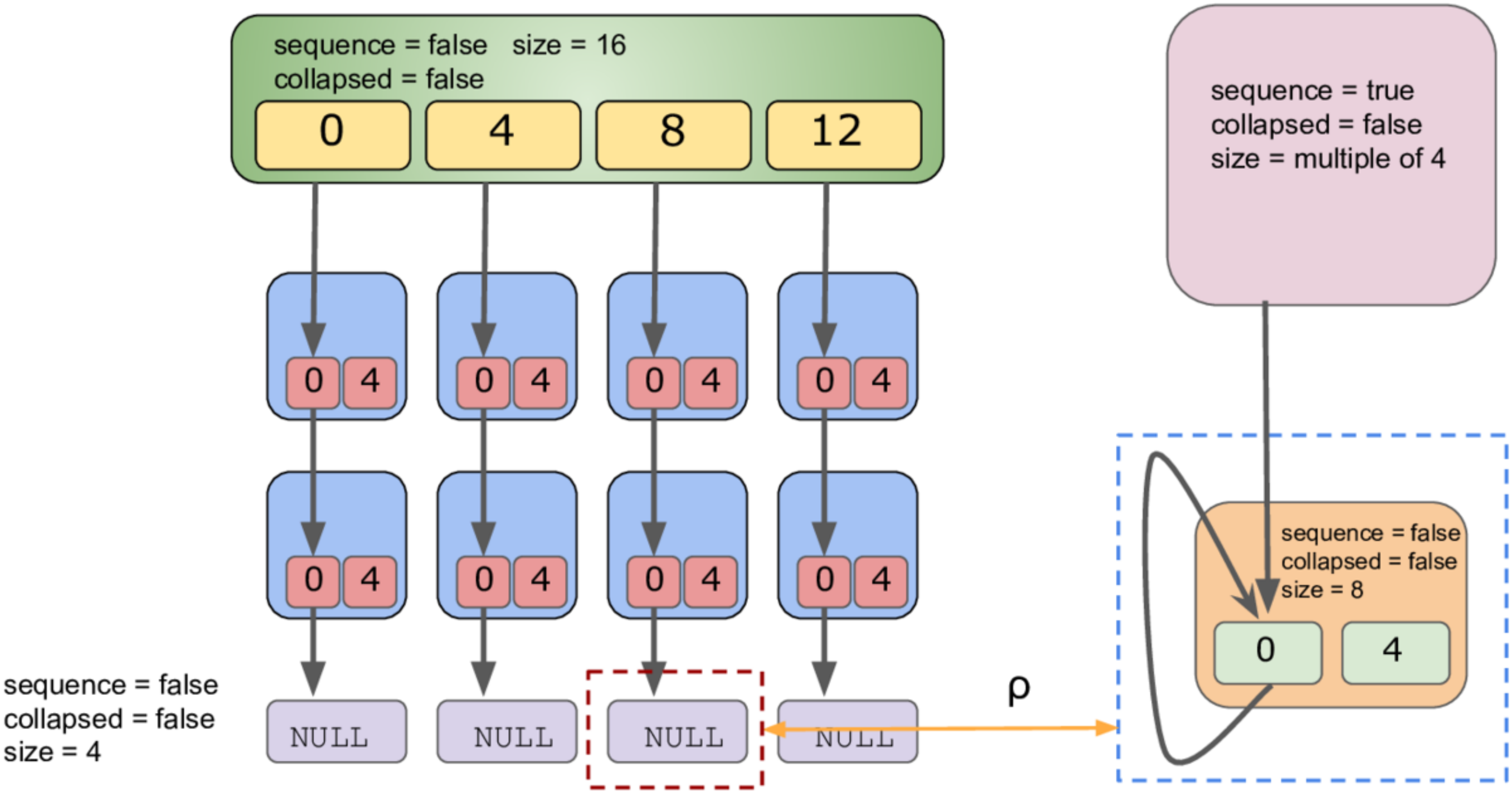
Simulation Relation between Graphs



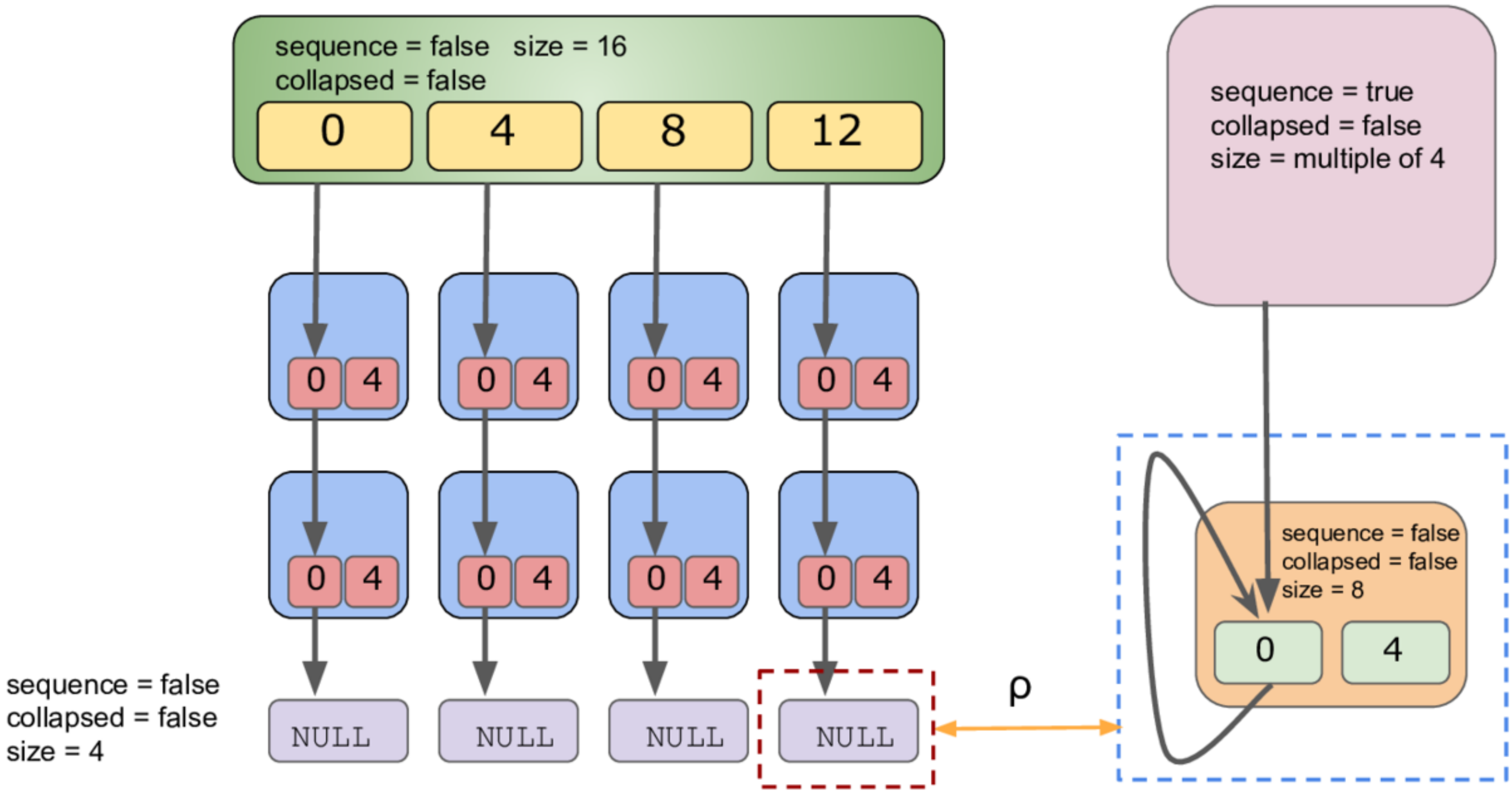
Simulation Relation between Graphs



Simulation Relation between Graphs



Simulation Relation between Graphs



Simulation Relation between Graphs

- $\gamma : \mathcal{G}_{\mathbb{A}} \mapsto 2^{\mathcal{G}_{\mathbb{C}}}$ defined as

$$\gamma(g_a) = \{g_c \in \mathcal{G}_{\mathbb{C}} \mid g_c \text{ simulated by } g_a\}$$

- It defines also an ordering between abstract graphs $g, g' \in \mathcal{G}_{\mathbb{A}}$

$$g \sqsubseteq_{\mathcal{G}_{\mathbb{A}}} g' \text{ if and only if } g \text{ is simulated by } g'$$

- It will play an essential role during the context-sensitive analysis (later in this talk)

Intra-Procedural Pointer Analysis

- Based on field-sensitive Steensgaard's
- Key operation: cell unification
- Ensure $c_1 = (n_1, o_1)$ and $c_2 = (n_2, o_2)$ are the same address
- If $o_1 < o_2$ then (other case symmetric)

  map $(n_1, 0)$ to $(n_2, o_2 - o_1)$

  $(n_1, o_1) = (n_2, o_2 - o_1 + o_1) = (n_2, o_2)$

  unify each (n_1, o_k) with $(n_2, o_2 - o_1 + o_k)$

UNIVERSITY OF WATERLOO

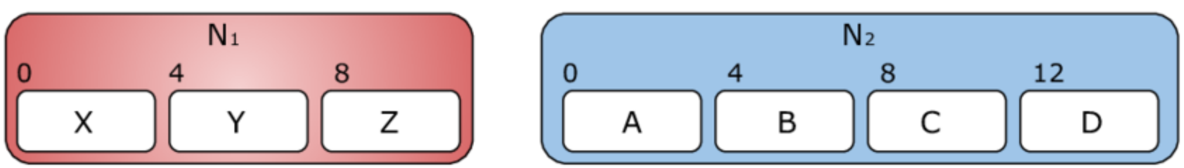
Intra-Procedural Pointer Analysis

- Based on field-sensitive Steensgaard's
- Key operation: cell unification
- Ensure $c_1 = (n_1, o_1)$ and $c_2 = (n_2, o_2)$ are the same address
- If $o_1 < o_2$ then (other case symmetric)

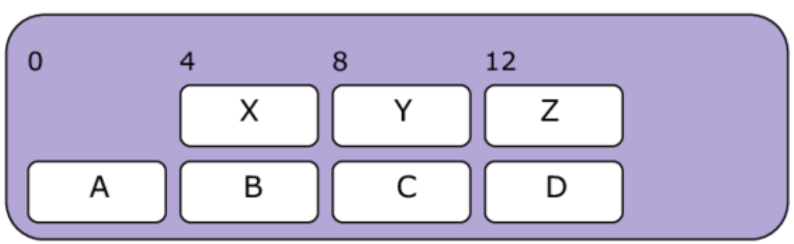
  map $(n_1, 0)$ to $(n_2, o_2 - o_1)$

  $$(n_1, o_1) = (n_2, o_2 - o_1 + o_1) = (n_2, o_2)$$

  unify each (n_1, o_k) with $(n_2, o_2 - o_1 + o_k)$

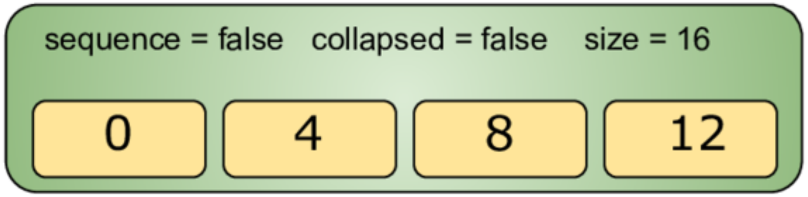


unify(Y,C) = unify((N_1,4),(N_2,8))



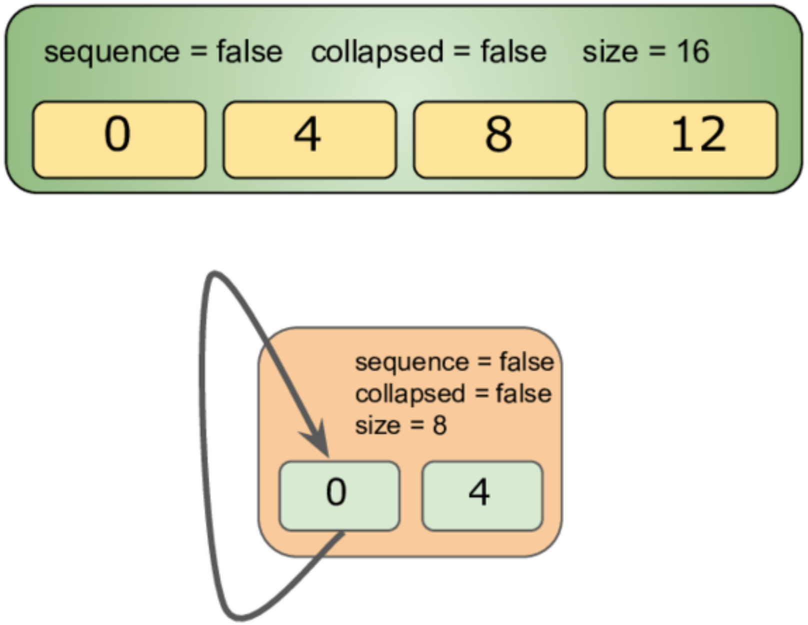
Array-Sensitivity

```
typedef struct list{
  struct list *n;
  int e;
} ll;

ll* mkList(int s,int e){
 if (s <= 0)
   return NULL;
 ll*p=malloc(sizeof(ll));
 p->e=e;
 p->n=mkList(s-1,e);
 return p;
}
#define N 4
void main(){
 ll* a[N];
 int i;
 for(i=0;i<N;++i)
   a[i] = mkList(M,0);
}
```



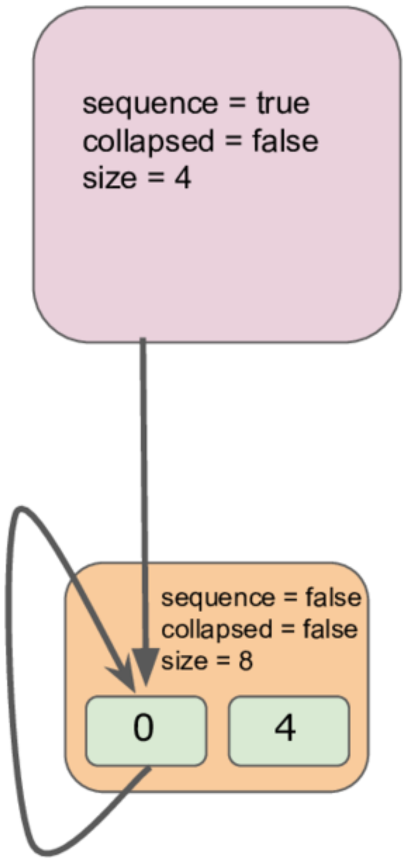
Array-Sensitivity

```c
typedef struct list{
  struct list *n;
  int e;
} ll;

ll* mkList(int s,int e){
 if (s <= 0)
   return NULL;
 ll*p=malloc(sizeof(ll));
 p->e=e;
 p->n=mkList(s-1,e);
 return p;
}
#define N 4
void main(){
 ll* a[N];
 int i;
 for(i=0;i<N;++i)
   a[i] = mkList(M,0);
}
```



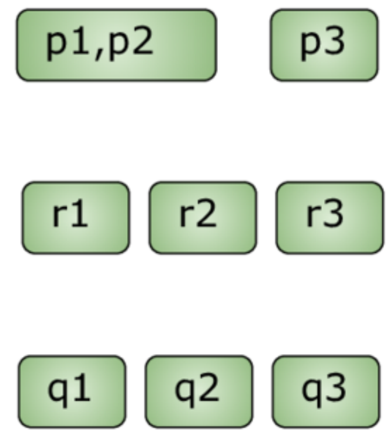
Array-Sensitivity

```
typedef struct list{
  struct list *n;
  int e;
} ll;

ll* mkList(int s,int e){
 if (s <= 0)
   return NULL;
 ll*p=malloc(sizeof(ll));
 p->e=e;
 p->n=mkList(s-1,e);
 return p;
}
#define N 4
void main(){
 ll* a[N];
 int i;
 for(i=0;i<N;++i)
   a[i] = mkList(M,0);
}
```



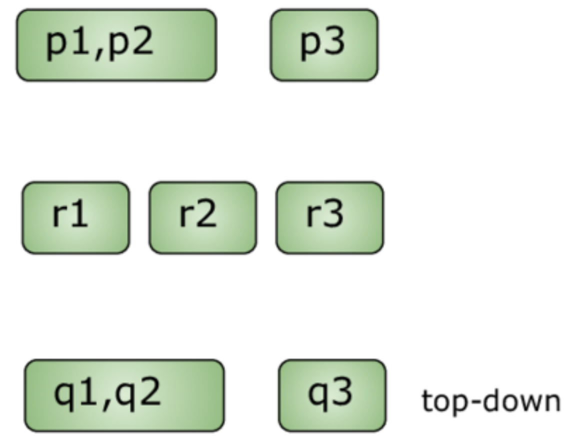
Context-Sensitive Pointer Analysis

```
void g(...) {
  f(p1,p2,p3);
}
void h(...) {
  f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
  ...
}
```



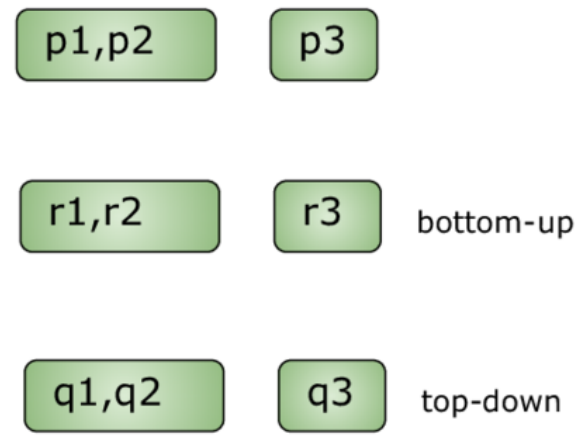
Context-Sensitive Pointer Analysis

```
void g(...) {
  f(p1,p2,p3);
}
void h(...) {
  f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
  ...
}
```



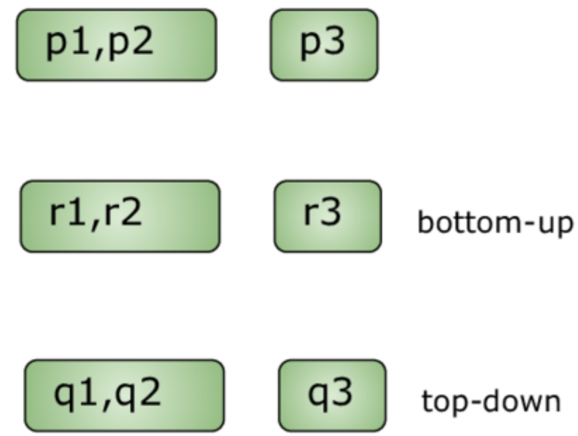
Context-Sensitive Pointer Analysis

```
void g(...) {
  f(p1,p2,p3);
}
void h(...) {
  f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
  ...
}
```



Context-Sensitive Pointer Analysis

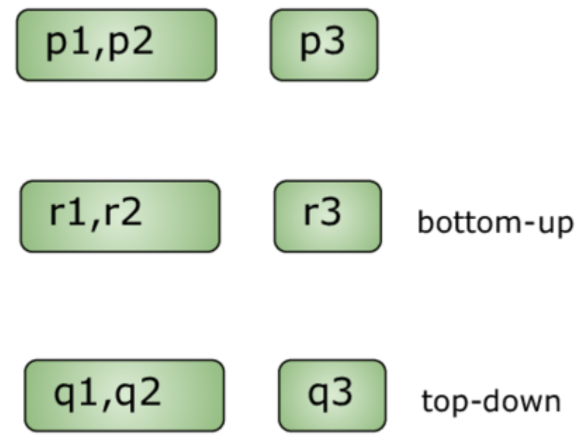
```
void g(...) {
  f(p1,p2,p3);
}
void h(...) {
  f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
  ...
}
```



- Next, `h`'s callsites and callsites where `h` is called must be re-analyzed, and so on

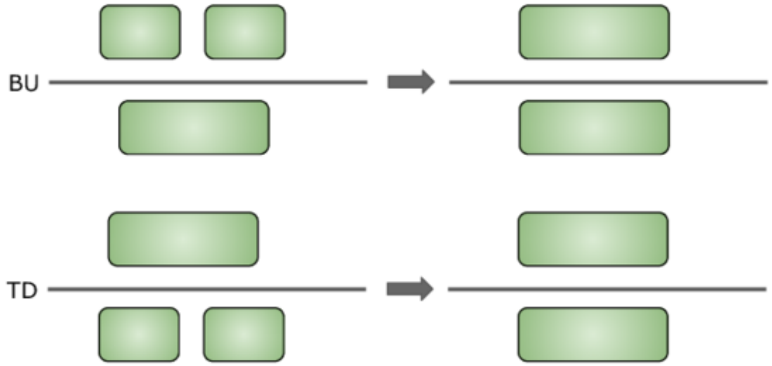
Context-Sensitive Pointer Analysis

```
void g(...) {
  f(p1,p2,p3);
}
void h(...) {
  f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
  ...
}
```



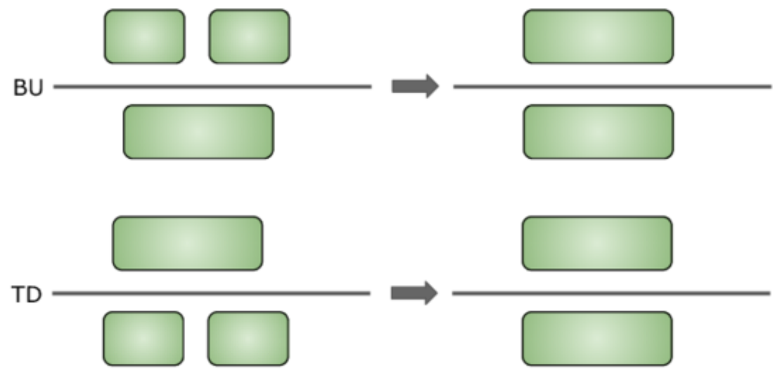
- Next, `h`'s callsites and callsites where `h` is called must be re-analyzed, and so on
- In general, after an unification we need to re-analyze:
  - if top-down: callsites with same callee and callsites within the callee
  - if bottom-up: callsites with same caller and callsites within the caller
- However, no need to re-analyze the whole function!
- Fixpoint over all callsites until no more bottom-up or top-down unifications

Bottom-Up and Top-Down Unifications



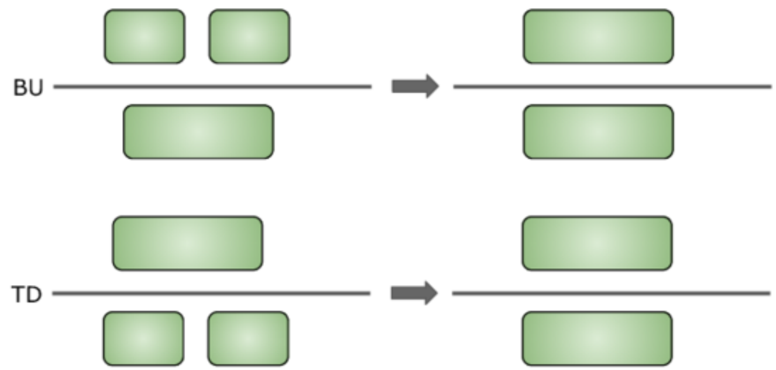
Q: How to decide whether BU, TD or no more unifications?

Bottom-Up and Top-Down Unifications



Q: How to decide whether BU, TD or no more unifications?
A: Simulation relation!

Bottom-Up and Top-Down Unifications



Q: How to decide whether BU, TD or no more unifications?
A: Simulation relation!

Build a simulation relation ρ between callee and caller graphs:

1. if ρ is not a function then BU
2. else if ρ is a function but not injective then TD
3. else ρ is an injective function then do nothing

Context-Sensitive Pointer Analysis: All Pieces Together

1. for each function in reverse topological order of the call graph

   compute summary

2. for each callsite

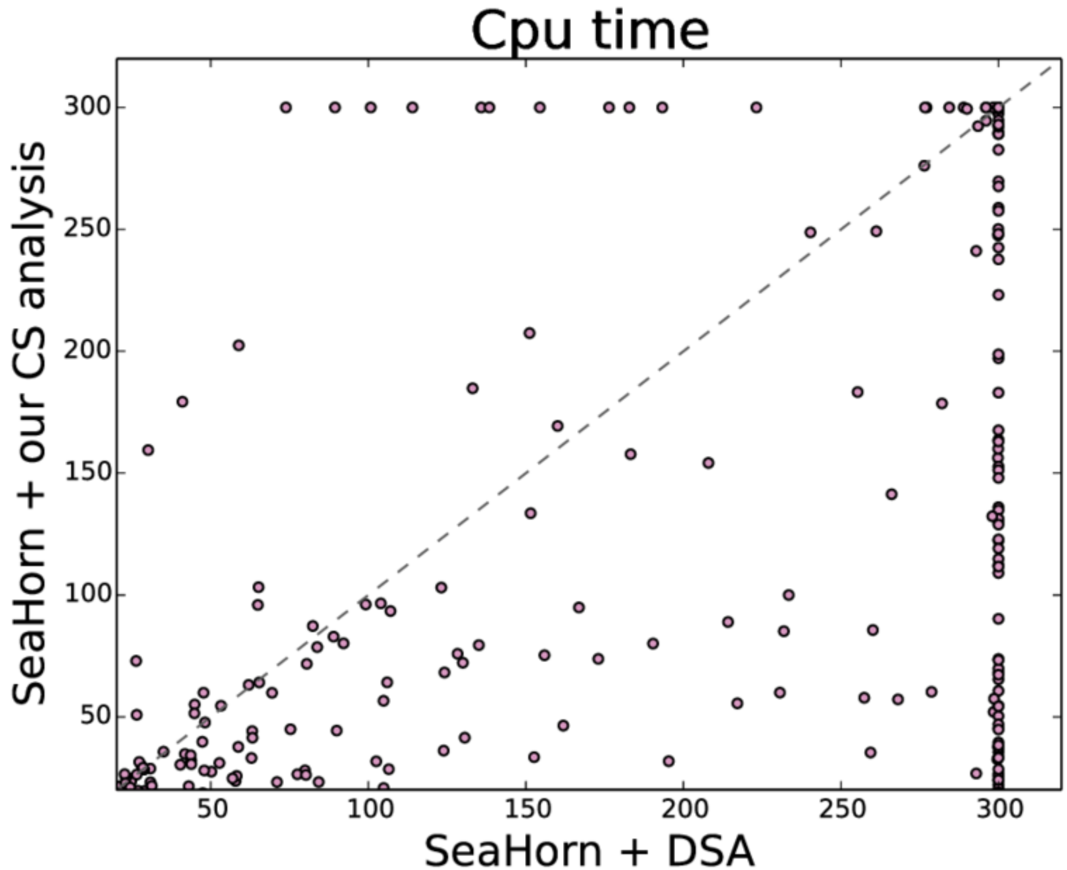
   clone callee's summary into the caller graph and unify formal/actual cells

3. apply BU and TD unifications until CC holds for all callsites

Experiments

- Integrated the pointer analysis in SeaHorn
- The pointer analysis is used during VC generation
- Compared SeaHorn verification time using:
  - (CI) DSA Pointer analysis from LLVM PoolAlloc project
  - Our pointer analysis

Experiments on SV-COMP C Programs



- 2000 benchmarks from SV-COMP DeviceDrivers64 category
- Verification time with timeout of 5m and 4GB memory limit
- With our analysis SeaHorn proved 81 more programs

(Ongoing) C++ Case Study

Goal:

Verify absence of buffer overflows on the flight control system of the Core Autonomous Safety Software (CASS) of an Autonomous Flight Safety System

- 13,640 LOC (excluding blanks/comments) written in C++ using standard C++ 2011 and following MISRA C++ 2008
- It follows an object-oriented style and makes heavy use of dynamic arrays and singly-linked lists

	#Objects	#Collapsed	Max. Density	% Proven
Sea + DSA	258	49%	80%	13
Sea + our CS	12,789	4%	13%	21

Related Work (1/2)

- Our work is inspired by Data Structure Analysis (DSA) of Lattner et al.: a context and field-sensitive (FS) PA with explicit heap representation:
  - context-sensitivity (CS) cannot be exploited: **CC** is not guaranteed
  - array-insensitive
- Many Software Model Checkers (e.g., CBMC, Smack, Cascade) are based on static memory partitioning via a FS unification-based pointer analysis
  - they are context-insensitive (CI)
  - Smack and Cascade perform type inference to refine partitions

Related Work (2/2)

- Deductive verification systems (HAVOC, VCC, and Frama-C):
  - More precise memory models
  - require quantified axioms
  - Frama-C/Jessie is CS but it rejects programs that does not satisfy **CC**
- Pointer analyses combined with other abstractions:
  - Miné's cell-based memory model: CI, flow-sensitive, FS with numerical abstraction of offsets
- Shape analysis and its combination with numerical abstractions can infer more expressive invariants but scalability is challenging

Conclusions

- Modular proofs require context-sensitive heap reasoning
- We adopted a very high-level memory model that can still express low-level C/C++ features such as:
  - pointer arithmetic, pointer casts and type unions
- We presented a scalable field-,array-,context-sensitive pointer analysis tailored for VC generation
  - A simulation relation between points-to graphs plays a major role in the analysis of function calls
- It can produce a finer-grained partition of memory that often results in faster verification times

All Software Publicly Available

- https://github.com/seahorn/sea-dsa
- https://github.com/seahorn/llvm-dsa
- https://github.com/seahorn/seahorn
- https://bitbucket.org/spacer/code
- https://github.com/seahorn/crab-llvm
- https://github.com/seahorn/crab

?

?

?

?

?

?