

Bi-Abductive Analysis for Advanced Low-Level Pointer Operations

[ECOOP'22, ESOP'25, TOPLAS'26]

Lukáš Holík Petr Peringer Adam Rogalewicz Florian Sextl Veronika Šoková
Tomáš Vojnar Florian Zuleger

16 October 2026



Informatics



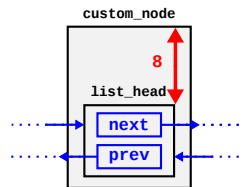
BRNO FACULTY
UNIVERSITY OF INFORMATION
OF TECHNOLOGY TECHNOLOGY

MUNI
FACULTY
OF INFORMATICS

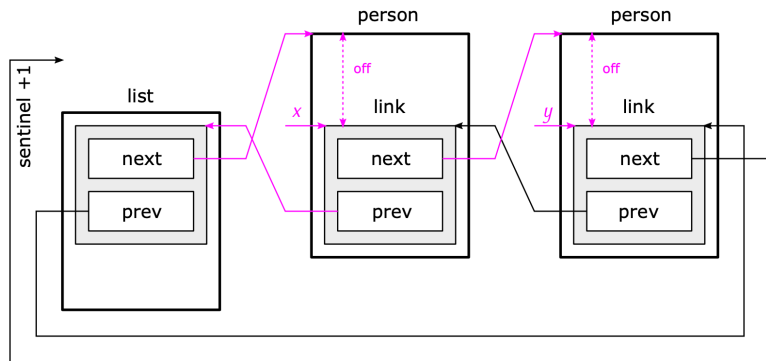
C Programs with Lists and Low-Level Pointer Operations

- Dynamic memory **allocation**, explicit **deallocation**.
- Various kinds of **linked lists** (SLL, DLL, nested, circular, intrusive, ...).
- **Pointer arithmetics**:
 - **Moving** inside (around) allocated objects.
 - **Bit masking** on pointers in use.
 - Address alignment, storing additional information into pointers themselves, ...
- Block operations (memcpy, memmove).
- Safe manipulation of **pointers to unallocated memory**. ETC.

```
// moving to the next item
y = (struct custom_node *) (
    (char *)x->next -
    offsetof(struct custom_node,
             list_head)
);
// testing whether it is the head
if (head != &y->list_head) ...
```



Patrick Wyatt's Intrusive Lists



```
// offset from a node pointer  
// to a link structure  
size_t off = (size_t) x -  
    ((size_t)x->prev->next & ~1);  
// link of the next node  
y = (link *)(  
    (size_t)lnk->next & ~1 + off  
);
```

Modular Analysis of Open Pointer Programs

Desired analysis features:

- **Open programs**: no main function needed, no analysis harnesses.
- **Modular**: each function analysed just once, derived contracts reusable.

Modular Analysis of Open Pointer Programs

Desired analysis features:

- **Open programs**: no main function needed, no analysis harnesses.
- **Modular**: each function analysed just once, derived contracts reusable.

Bi-abductive shape analysis based on **separation logic** [Distefano et al. POPL'09, JACM'11]:

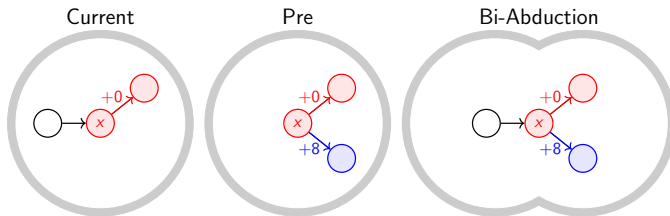
- **Bottom-up analysis** – starts from leaf functions of the call graph.
- Computes a (set of) **contracts** $\{Pre\}f\{Post\}$ for each function f .
 - Given pre-defined contracts for basic statements.
- An analysis step = **applying contracts** on the current state:
 - using bi-abductive reasoning.
- **Under-approximating safe pre-conditions**.
- **Loop acceleration** – widening, **over-approximation**:
 - followed by a classic analysis to guarantee soundness – **2-phase analysis**.

Bi-Abductive Reasoning: Intuition

Given: the *Current* state and a function/statement precondition *Pre*.

Goal: compute *Miss?* and *Frame?* such that

$$\textit{Current} * \textit{Miss?} \models \textit{Pre} * \textit{Frame?}.$$



- *Match* = the shared part between *Current* and *Pre*,
- *Miss* = the missing part of the current state,
- *Frame* = the **untouched** part of the current state.

Low-Level Separation Logic [ECOOP'22]

Separation logic (SL): a logic for describing sets of memory configurations.

We propose a **low-level flavour of SL** with (nested) list predicates:

$$\begin{aligned} \varphi ::= & \varepsilon_1 \mapsto \varepsilon_2 \mid \varepsilon_1 \mapsto \mathbf{c}[\varepsilon_2] \mid \varepsilon_1 \mapsto \mathbf{T}[\varepsilon_2] \mid \exists x. \varphi \mid \varphi_1 * \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \\ & ls_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2) \mid dls_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon'_1, \varepsilon'_2) \mid \text{emp} \mid \text{true} \mid \varepsilon_1 \bowtie \varepsilon_2 \\ \bowtie ::= & = \mid \neq \mid \leq \mid < \mid \geq \mid > \quad \varepsilon ::= \mathbf{c} \mid x \mid \mathbf{b}(\varepsilon) \mid \mathbf{e}(\varepsilon) \mid \mathbf{unop} \ \varepsilon \mid \varepsilon_1 \mathbf{bop} \ \varepsilon_2 \end{aligned}$$

- **Points-to predicates** $\varepsilon_1 \mapsto \varepsilon_2$ – at an address ε_1 , a value ε_2 is stored.

- $\varepsilon_1, \varepsilon_2$ are **expressions**, $\text{size}(\varepsilon_2)$ is constant, given by the type of ε_2 .
- $\varepsilon_1 \mapsto \mathbf{c}[\varepsilon_2], \varepsilon_1 \mapsto \mathbf{T}[\varepsilon_2]$: blocks of bytes of (possibly variable) size ε_2 .

- **Separation** on the level of **fields**, not entire objects.

- **Beginning** and **end of block**:

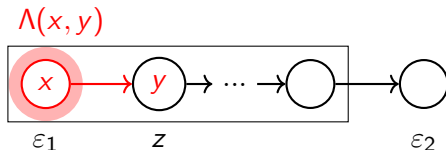
- **Usual notation**: $x \mapsto (\text{next} : y, \text{back} : z, \text{data} : d)$.
- **Our notation**: $x \mapsto y * x + 8 \mapsto z * x + 16 \mapsto d$
 $* \mathbf{b}(x) = x * \mathbf{e}(x) = x + 24$.

```
struct dll{  
    struct dll * next, *back;  
    int data;  
}
```

Separation Logic – List Segments

Singly-linked lists are captured by a special parametric predicate

- $ls_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2)$.



$$ls_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2) \equiv emp * \varepsilon_1 = \varepsilon_2$$

$$ls_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2) \equiv \exists z. \Lambda(\varepsilon_1, z) * ls_{\Lambda(x,y)}(z, \varepsilon_2)$$

- Λ can be more complex: data fields, nested lists, ...

Doubly-linked lists are captured by a parametric predicate

- $dls_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon'_1, \varepsilon'_2)$.

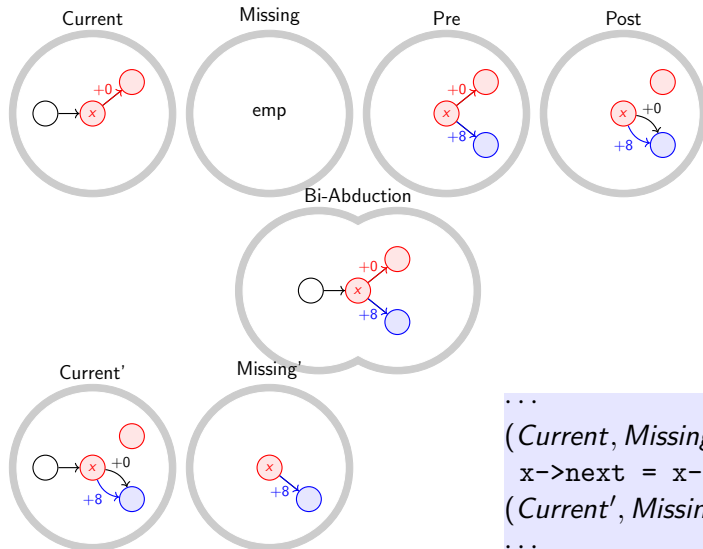
Contracts of Functions/Statements

Contracts are Hoare triples

$$\{Pre\}statement\{Post\}.$$

- Each **basic statement** is captured by a **pre-defined contract**.
- For each **function** a (set of) **contract(s)** is computed.
- A **state of the analysis** of a function $f - (Miss_f, Curr_f)$:
 - $Miss_f$ – the so-far computed precondition of f .
 - $Curr_f$ – the current state of f as seen by the analysis.
- The analysis of a function **starts** with (emp, emp) .
- **One analysis step** = an application of a contract $\{Pre_g\}g(\dots)\{Post_g\}$:
 - Solve the **bi-abduction problem** $Curr_f * Miss_g? \models Pre_g * Frame_f?$.
 - **The next analysis state** is: $(Miss_f * Miss_g, Post_g * Frame_f)$.

Contract Application – An Illustration



...

$(Current, Missing)$
 $x \rightarrow next = x \rightarrow prev;$
 $(Current', Missing')$
...

Example

```
void traverse(struct sll *l){  $P \equiv \text{emp}, Q \equiv \text{emp}$   
    struct sll *t = l;  
     $P \equiv \text{emp}, Q \equiv l = t$   
  
    while (t!=null) {  
  
        t=t->next;  
    }  
  
}
```

Example

```
void traverse(struct sll *l){  $P \equiv \text{emp}, Q \equiv \text{emp}$   
    struct sll *t = l;  
     $P \equiv \text{emp}, Q \equiv l = t$   
  
    while (t!=null) {  
         $P \equiv l \neq \text{null}, Q \equiv l \neq \text{null} * l = t$   
  
        t=t->next;  
    }  
     $P \equiv l = \text{null}, Q \equiv l = \text{null} * t = \text{null}$   
}
```

$\Leftarrow$ Candidate contracts for the function

Example

```
void traverse(struct sll *l){  $P \equiv \text{emp}, Q \equiv \text{emp}$   
    struct sll *t = l;  
     $P \equiv \text{emp}, Q \equiv l = t$   
     $P \equiv l \mapsto a_1, Q \equiv l \mapsto a_1 * a_1 = t$   
  
    while (t!=null) {  
         $P \equiv l \neq \text{null}, Q \equiv l \neq \text{null} * l = t$   
  
        t=t->next;  
    }  
     $P \equiv l = \text{null}, Q \equiv l = \text{null} * t = \text{null}$   
}
```

$\Leftarrow$ Candidate contracts for the function

Example

```
void traverse(struct sll *l){  $P \equiv \text{emp}, Q \equiv \text{emp}$ 
```

```
    struct sll *t = l;
```

```
     $P \equiv \text{emp}, Q \equiv l = t$ 
```

```
     $P \equiv l \mapsto a_1, Q \equiv l \mapsto a_1 * a_1 = t$ 
```

```
    while (t!=null) {
```

```
         $P \equiv l \neq \text{null}, Q \equiv l \neq \text{null} * l = t$ 
```

```
         $P \equiv l \mapsto a_1 * a_1 \neq \text{null}, Q \equiv l \mapsto a_1 * a_1 = t * t \neq \text{null}$ 
```

```
        t=t->next;
```

```
    }
```

```
     $P \equiv l = \text{null}, Q \equiv l = \text{null} * t = \text{null}$ 
```

```
     $P \equiv l \mapsto \text{null}, Q \equiv l \mapsto \text{null} * t = \text{null}$ 
```

```
}
```

$\Leftarrow$ Candidate contracts for the function

Example

```
void traverse(struct sll *l){  $P \equiv \text{emp}, Q \equiv \text{emp}$   
    struct sll *t = l;  
     $P \equiv \text{emp}, Q \equiv l = t$   
     $P \equiv l \mapsto a_1, Q \equiv l \mapsto a_1 * a_1 = t$   
     $P \equiv l \mapsto a_1 * a_1 \mapsto a_2, Q \equiv l \mapsto a_1 * a_1 \mapsto a_2 * a_2 = t$   
    while (t!=null) {  
         $P \equiv l \neq \text{null}, Q \equiv l \neq \text{null} * l = t$   
         $P \equiv l \mapsto a_1 * a_1 \neq \text{null}, Q \equiv l \mapsto a_1 * a_1 = t * t \neq \text{null}$   
  
        t=t->next;  
    }  
     $P \equiv l = \text{null}, Q \equiv l = \text{null} * t = \text{null}$   
     $P \equiv l \mapsto \text{null}, Q \equiv l \mapsto \text{null} * t = \text{null}$   
}
```

$\Leftarrow$ Candidate contracts for the function

Example

```
void traverse(struct sll *l){  $P \equiv \text{emp}, Q \equiv \text{emp}$   
    struct sll *t = l;  
     $P \equiv \text{emp}, Q \equiv l = t$   
     $P \equiv l \mapsto a_1, Q \equiv l \mapsto a_1 * a_1 = t$   
     $P \equiv ls_{x \mapsto y}(l, a_2), Q \equiv ls_{x \mapsto y}(l, a_2) * a_2 = t$   
    while (t!=null) {  
         $P \equiv l \neq \text{null}, Q \equiv l \neq \text{null} * l = t$   
         $P \equiv l \mapsto a_1 * a_1 \neq \text{null}, Q \equiv l \mapsto a_1 * a_1 = t * t \neq \text{null}$   
  
        t=t->next;  
    }  
     $P \equiv l = \text{null}, Q \equiv l = \text{null} * t = \text{null}$   
     $P \equiv l \mapsto \text{null}, Q \equiv l \mapsto \text{null} * t = \text{null}$   
}
```

$\Leftarrow$ Candidate contracts for the function

Example

```
void traverse(struct sll *l){  $P \equiv \text{emp}, Q \equiv \text{emp}$   
    struct sll *t = l;  
     $P \equiv \text{emp}, Q \equiv l = t$   
     $P \equiv l \mapsto a_1, Q \equiv l \mapsto a_1 * a_1 = t$   
     $P \equiv ls_{x \mapsto y}(l, a_2), Q \equiv ls_{x \mapsto y}(l, a_2) * a_2 = t$   
    while (t!=null) {  
         $P \equiv l \neq \text{null}, Q \equiv l \neq \text{null} * l = t$   
         $P \equiv l \mapsto a_1 * a_1 \neq \text{null}, Q \equiv l \mapsto a_1 * a_1 = t * t \neq \text{null}$   
         $P \equiv ls_{x \mapsto y}(l, a_2) * a_2 \neq \text{null}, Q \equiv ls_{x \mapsto y}(l, a_2) * a_2 = t * a_2 \neq \text{null}$   
        t=t->next;  
    }  
     $P \equiv l = \text{null}, Q \equiv l = \text{null} * t = \text{null}$   
     $P \equiv l \mapsto \text{null}, Q \equiv l \mapsto \text{null} * t = \text{null}$   
     $P \equiv ls_{x \mapsto y}(l, \text{null}), Q \equiv ls_{x \mapsto y}(l, \text{null}) * t = \text{null}$   
}
```

$\Leftarrow$ Candidate contracts for the function

Example

```
void traverse(struct sll *l){  $P \equiv \text{emp}, Q \equiv \text{emp}$ 
```

```
    struct sll *t = l;
```

```
     $P \equiv \text{emp}, Q \equiv l = t$ 
```

```
     $P \equiv l \mapsto a_1, Q \equiv l \mapsto a_1 * a_1 = t$ 
```

```
     $P \equiv ls_{x \mapsto y}(l, a_2), Q \equiv ls_{x \mapsto y}(l, a_2) * a_2 = t$ 
```

```
    while (t!=null) {
```

```
         $P \equiv l \neq \text{null}, Q \equiv l \neq \text{null} * l = t$ 
```

```
         $P \equiv l \mapsto a_1 * a_1 \neq \text{null}, Q \equiv l \mapsto a_1 * a_1 = t * t \neq \text{null}$ 
```

```
         $P \equiv ls_{x \mapsto y}(l, a_2) * a_2 \neq \text{null}, Q \equiv ls_{x \mapsto y}(l, a_2) * a_2 = t * a_2 \neq \text{null}$ 
```

```
        t=t->next;
```

```
    }
```

```
     $P \equiv l = \text{null}, Q \equiv l = \text{null} * t = \text{null}$ 
```

```
     $P \equiv l \mapsto \text{null}, Q \equiv l \mapsto \text{null} * t = \text{null}$ 
```

```
     $P \equiv ls_{x \mapsto y}(l, \text{null}), Q \equiv ls_{x \mapsto y}(l, \text{null}) * t = \text{null}$ 
```

```
}
```

$\Leftarrow$ Candidate contracts for the function

Second phase

The contract candidates created by the abductive analysis are **unsafe**:

- The acceleration of the loop is **overapproximating** – fine for postcondition.
 - A **safe precondition** must be **underapproximation**.
- Problem with **nondeterministic branching** – a contract is safe for a single branch only.

Solution:

- **Second phase** – for each Pre , a classic **non-bi-abductive** analysis checks its validity.

Drawback:

- The second phase costs computational resources.
- Many/all preconditions may be discarded by the second phase.

Prototype Implementation

Prototype tool called Broom:

- <https://pajda.fit.vutbr.cz/rogalew/broom>
- Implemented in **OCaml**.
- Front-end based on **Code Listener** – a GCC plugin used in Predator.
- Analysis of **intermediate code** (Gimple, plan to add LLVM IR).
- (UN)SAT **queries reduced** from SL to the **bitvectors SMT theory**.
- Evaluated using the **Z3** and **CVC5** solvers.

Experimental Results – Fragments of List Manipulating Code

Program	Exp res.	#F	Broom			Infer			Gillian		
			T [s]	#F	V	T [s]	#F	V	T [s]	#F	V
circ-dll	ML	4	6	4	ML	0.5	1/1	IE	1.2	1/0/2	OK
circ-dll-err	DF	4	6	4	DF	0.5	1/1	IE	1.2	1/0/2	OK
circ-dll-emb	OK	4	9	4	OK	0.5	1/2	IE	0.6	0	PE
linux-list-1	ML	11	56	10	ML	1.5	2/3	OK	0.6	0	PE
linux-list-2	OK	11	42	11	OK	0.7	1/6	IE	0.6	0	PE
linux-list-3	ML	11	28	11	ML	0.6	1/6	IE	0.6	0	PE
linux-list-all	OK	23	267	21/2	OK	1.0	7/15	IE	44	8/0/9	OK
intrusive-all	OK	15	99	10/5	OK	0.7	4/3	OK	0.6	0	PE
intrusive-2	OK	9	45	6/2	OK	0.7	1/3	IE	0.6	0	PE
intrusive-3	OK	20	133	10/5	OK	0.9	4/3	OK	0.6	0	PE

Expected Result: OK= *no error*, DF= *double free*, ML= *memory leak*.

Errors: IE= *internal error*, PE= *internal parser error*.

#F – functions: complete/partial/error contracts.

Getting Rid of the Second Phase [ESOP'25]

The fact that (over-approximate) biabduction tools need a re-execution phase to filter out unsound candidate triples has been a thorn in the side of any biabduction practitioner since time immemorial (=15 years ago).

- Reviewer B, TPSA Workshop [years from that time]

Getting Rid of the Second Phase [ESOP'25]

The fact that (over-approximate) biabduction tools need a re-execution phase to filter out unsound candidate triples has been a thorn in the side of any biabduction practitioner since time immemorial (=15 years ago).

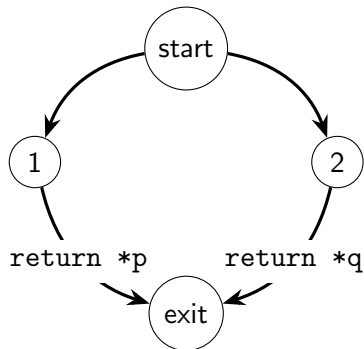
- Reviewer B, TPSA Workshop [years from that time]

Our Achievement

Make a new tool, called Brush, that is single-phase without changing the logic, biabductive reasoning, nor the abstraction procedure used in Broom.

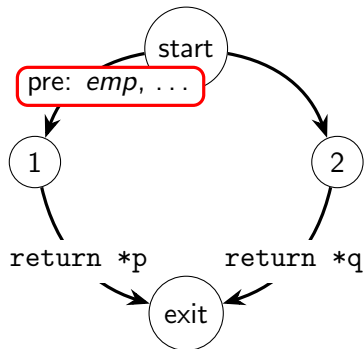
Unsound Branching

```
int choice(int *p, int *q){  
  if (?) {  
    return *p;  
  } else {  
    return *q;  
  }  
}
```



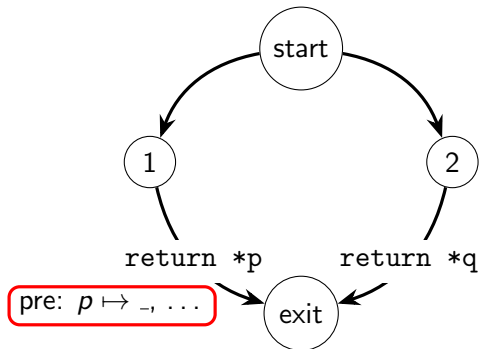
Unsound Branching

```
int choice(int *p, int *q){  
  if (?) {  
    return *p;  
  } else {  
    return *q;  
  }  
}
```



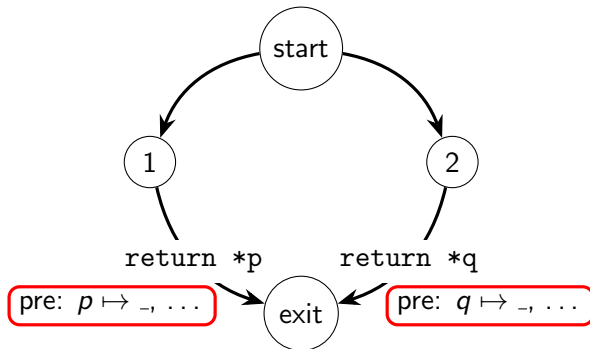
Unsound Branching

```
int choice(int *p, int *q){  
  if (?) {  
    return *p;  
  } else {  
    return *q;  
  }  
}
```



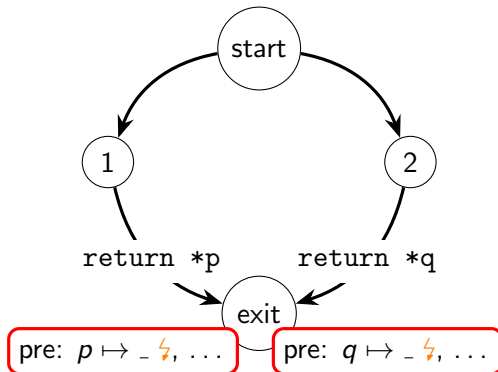
Unsound Branching

```
int choice(int *p, int *q){  
  if (?) {  
    return *p;  
  } else {  
    return *q;  
  }  
}
```



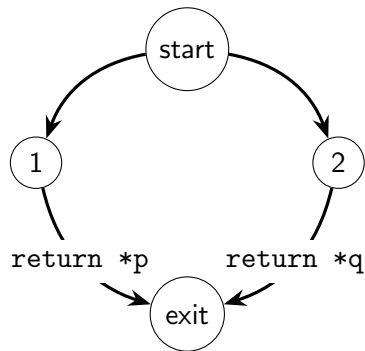
Unsound Branching

```
int choice(int *p, int *q){  
  if (?) {  
    return *p;  
  } else {  
    return *q;  
  }  
}
```

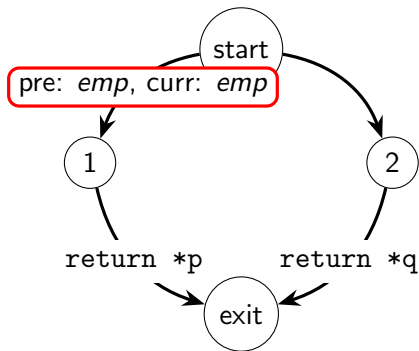


Problem

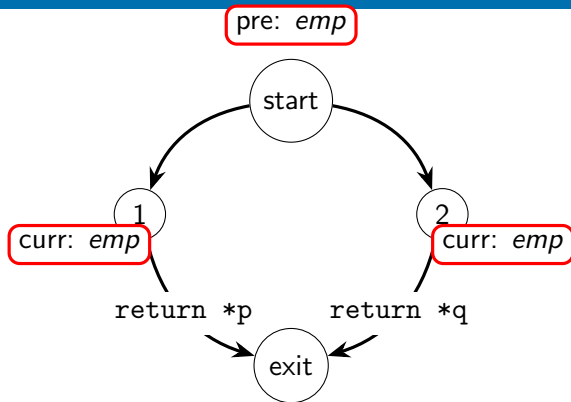
The isolated execution of the branches yields two contracts, each crashing on one branch.



Shared Abduction



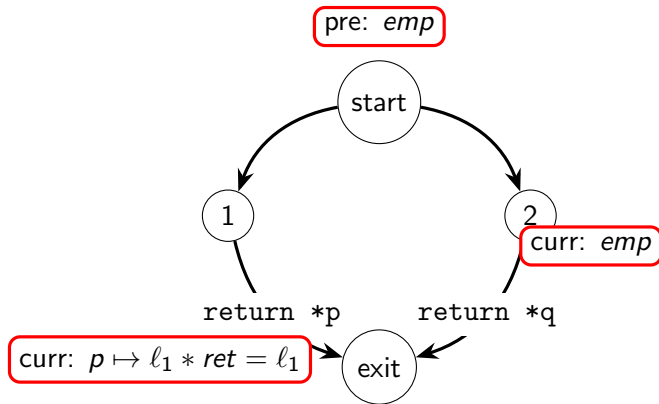
Shared Abduction



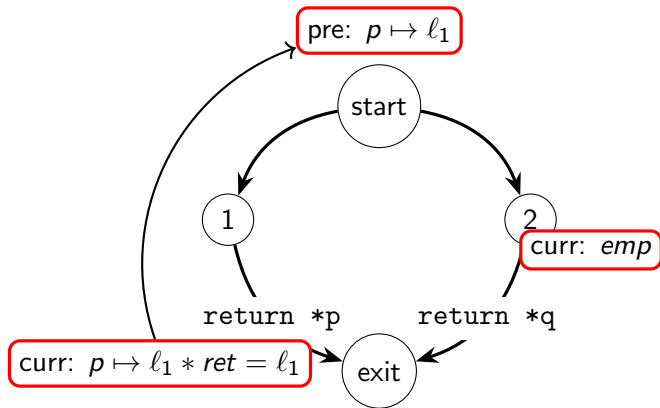
Shared Abduction Idea

Explicitly share a single pre-condition for equally-reachable current states.

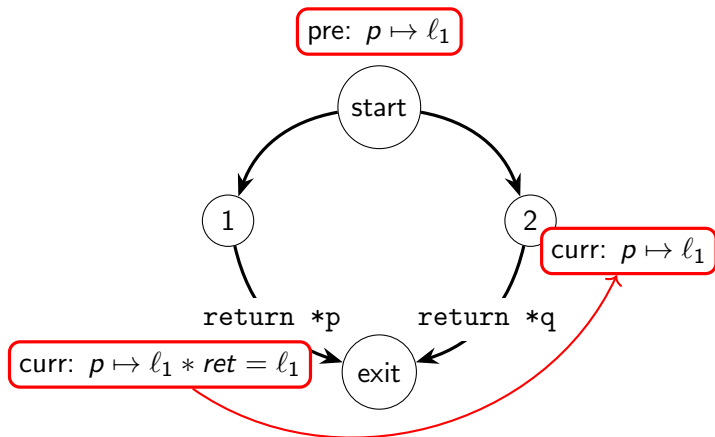
Shared Abduction



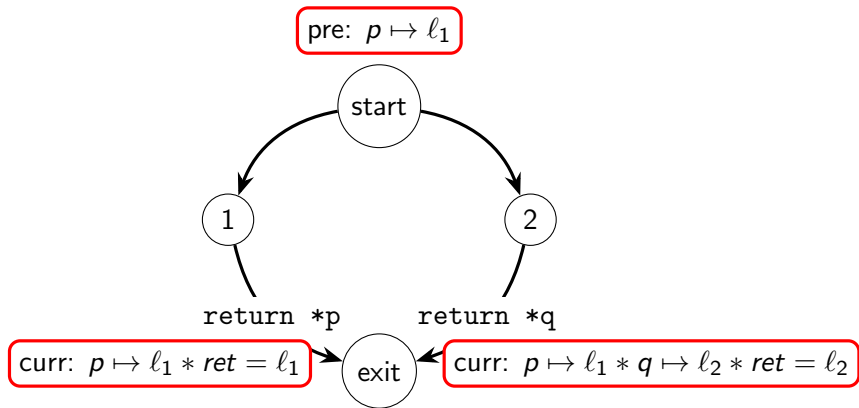
Shared Abduction



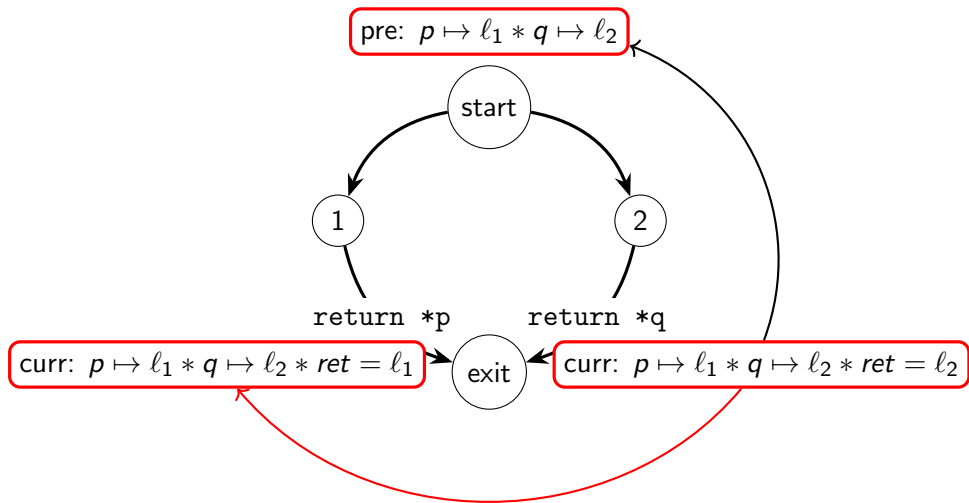
Shared Abduction



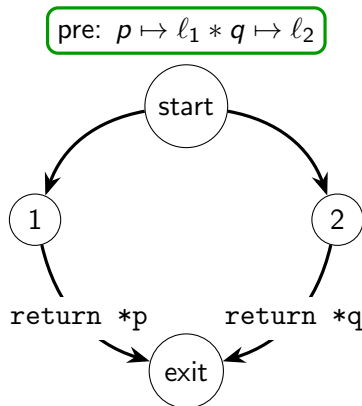
Shared Abduction



Shared Abduction



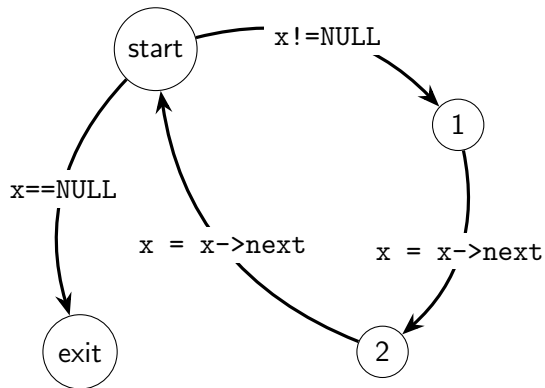
Shared Abduction



post: $(p \mapsto l_1 * q \mapsto l_2 * ret = l_1) \vee (p \mapsto l_1 * q \mapsto l_2 * ret = l_2)$

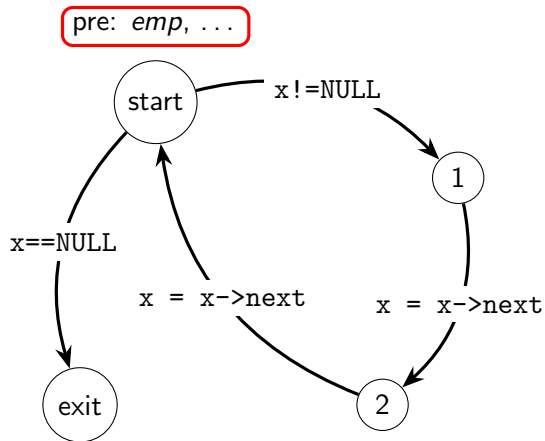
Unsound Loop Acceleration

```
void traverse_even(node *x) {  
  while (x != NULL) {  
    x = x->next;  
    x = x->next;  
  }  
}
```



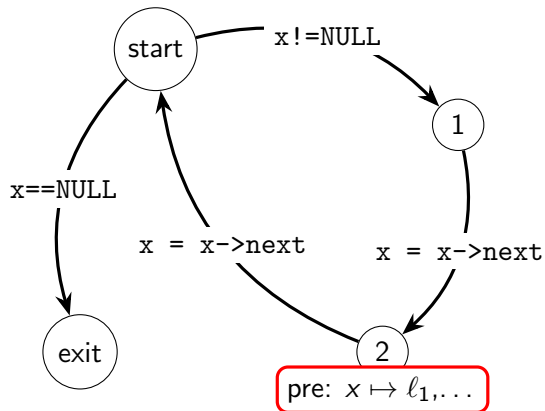
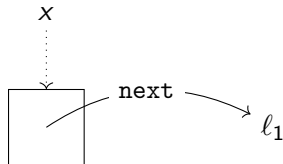
Unsound Loop Acceleration

```
void traverse_even(node *x) {  
  while (x != NULL) {  
    x = x->next;  
    x = x->next;  
  }  
}
```



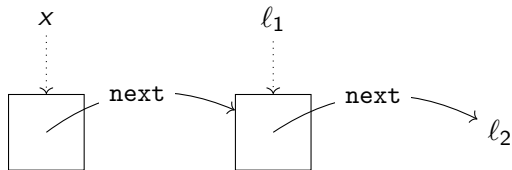
Unsound Loop Acceleration

```
void traverse_even(node *x) {  
  while (x != NULL) {  
    x = x->next;  
    x = x->next;  
  }  
}
```

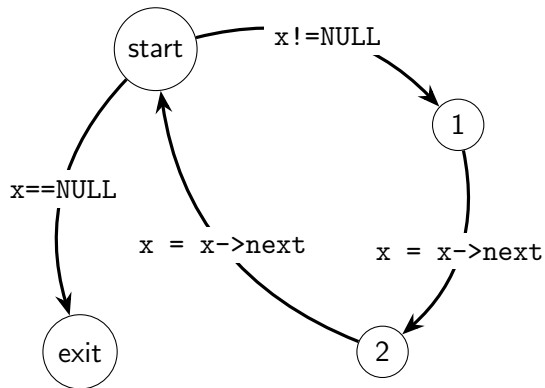


Unsound Loop Acceleration

```
void traverse_even(node *x) {  
  while (x != NULL) {  
    x = x->next;  
    x = x->next;  
  }  
}
```

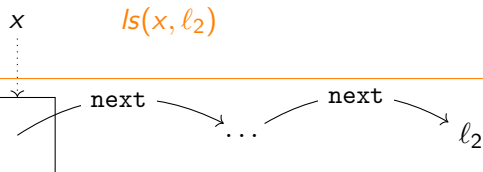


pre: $x \mapsto l_1 * l_1 \mapsto l_2, \dots$

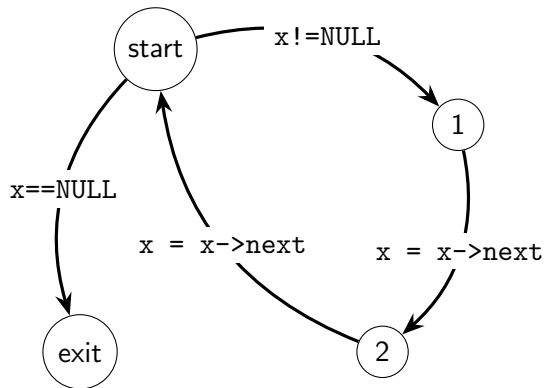


Unsound Loop Acceleration

```
void traverse_even(node *x) {  
  while (x != NULL) {  
    x = x->next;  
    x = x->next;  
  }  
}
```



pre: $ls(x, \ell_2)$ ⚡,...



Safe and Fast Biabductive Loop Acceleration on Lists

Input: A function consisting of a loop L only

Output: A sound contract c for L or FAILURE

Compute $s_1 \leftarrow \text{Analysis}(L)$

$s_{inv} \leftarrow \text{HeuristicExtrapolation}(s_1)$

$\text{CheckIndInv}(L, s_{inv})$ or FAILURE

Construct c from s_{inv}

Safe and Fast Biabductive Loop Acceleration on Lists

Input: A function consisting of a loop L only

Output: A sound contract c for L or FAILURE

Compute $s_1 \leftarrow \text{Analysis}(L)$

$s_{inv} \leftarrow \text{HeuristicExtrapolation}(s_1)$

$\text{CheckIndInv}(L, s_{inv})$ or FAILURE

Construct c from s_{inv}

Heuristic Idea

Extrapolate information from the analysis to an arbitrary number of loop iterations as a **candidate** loop invariant state s_{inv} .

Safe and Fast Biabductive Loop Acceleration on Lists

Input: A function consisting of a loop L only

Output: A sound contract c for L or FAILURE

Compute $s_1 \leftarrow \text{Analysis}(L)$

$s_{inv} \leftarrow \text{HeuristicExtrapolation}(s_1)$

$\text{CheckIndInv}(L, s_{inv})$ or FAILURE

Construct c from s_{inv}

Safe and Fast Biabductive Loop Acceleration on Lists

Input: A function consisting of a loop L only

Output: A sound contract c for L or FAILURE

Compute $s_1 \leftarrow \text{Analysis}(L)$

$s_{inv} \leftarrow \text{HeuristicExtrapolation}(s_1)$

$\text{CheckIndInv}(L, s_{inv})$ or FAILURE

Construct c from s_{inv}

Safe and Fast Biabductive Loop Acceleration on Lists

Input: A function consisting of a loop L only

Output: A sound contract c for L or FAILURE

Compute $s_1 \leftarrow \text{Analysis}(L)$

$s_{inv} \leftarrow \text{HeuristicExtrapolation}(s_1)$

$\text{CheckIndInv}(L, s_{inv})$ or FAILURE

Construct c from s_{inv}

Heuristic extrapolation:

- Either succeed or fail fast!
- If it does not succeed:
 - another symbolic iteration of the loop,
 - classical abstraction,
 - inductiveness test (or resort to 2-phase for the given function).

Heuristic Loop Extrapolation on Lists

Basic principle:

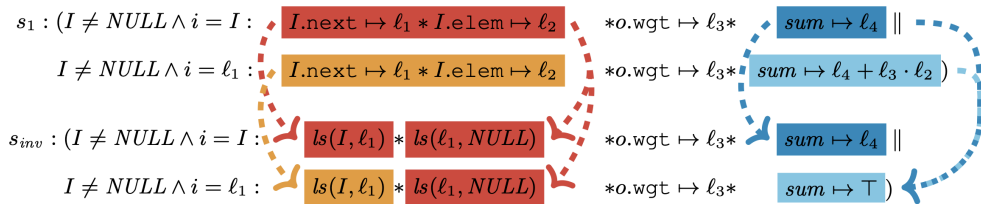
- Find variables with changed values.
- Identify those which **moved along some pointer**.
- **Extrapolate to lists**: both the traversed/to-traverse parts (using the loop exit condition).
- The others considered **data field changes** (currently abstracted to $\top$).

Heuristic Loop Extrapolation on Lists

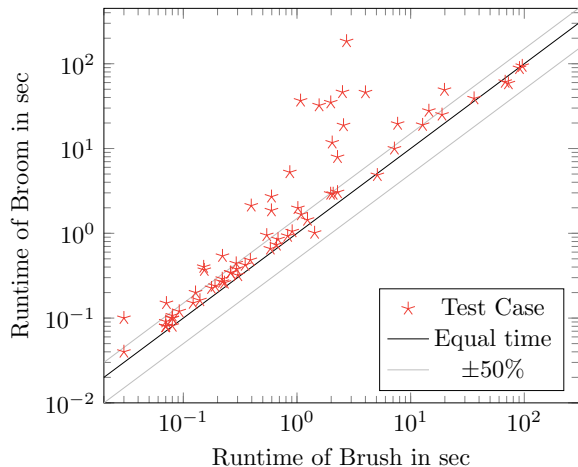
Basic principle:

- Find variables with changed values.
- Identify those which **moved along some pointer**.
- Extrapolate to lists**: both the traversed/to-traverse parts (using the loop exit condition).
- The others considered **data field changes** (currently abstracted to $\top$).

Example: computing a weighted sum of elements stored in a list iterated by the variable i :



Evaluation of the Implementation in Brush



- Overall runtime: -48%
- Runtime for loops: -95%
- Loop iterations analyzed: -66%
- Runtime for branching: -10%
- Handling of non-deterministic branching: ✓

Conclusion and Future Work

- Two related approaches to compositional analysis of memory safety of open programs with lists and low-level pointer manipulation:
 - **BROOM**: wider class of supported of programs, slow.
 - **BRUSH**: single phase analysis, faster, less generic (for now).
- **Future work**:
 - Unify the two branches.
 - Improve the scalability, esp. for BROOM.
 - E.g., reduce the number of SMT queries by static approximations of simple queries.
 - Improve the genericity.
 - Lambda learning and framing.
 - Abstraction/extrapolation on other data structures.
 - Handling non-pointer data.
 - Witnesses of correctness, combinations with AI, other languages (Rust), ...

Conclusion and Future Work

- Two related approaches to compositional analysis of memory safety of open programs with lists and low-level pointer manipulation:
 - **BROOM**: wider class of supported of programs, slow.
 - **BRUSH**: single phase analysis, faster, less generic (for now).
- Future work:
 - Unify the two branches.
 - Improve the scalability, esp. for BROOM.
 - E.g., reduce the number of SMT queries by static approximations of simple queries.
 - Improve the genericity.
 - Lambda learning and framing.
 - Abstraction/extrapolation on other data structures.
 - Handling non-pointer data.
 - Witnesses of correctness, combinations with AI, other languages (Rust), ...
 - What about some competition?

- What about trying some **competition on analyzing open pointer-manipulating programs?**
- Kind of analysis interesting for the practice.
- **Witness checkers for memory safety** under preparation.
 - Should be able to check the witnesses for open programs too.
- No unique result – **different contracts possible.**
 - Use decision procedures for separation logic to **compare their strength?**

Contracts for Basic Statements

- Function `load(x, y)` with the body $x := *y$:

$$\{y = Y * Y \mapsto z\} \text{load}(x, y) \{x = z * y = Y * Y \mapsto z\}$$

- Function `store(x, y)` with the body $*x := y$:

$$\{x = X * y = Y * X \mapsto z\} \text{store}(x, y) \{x = X * y = Y * X \mapsto Y\}$$

- Function `malloc(x, y)` with the body $x := \text{malloc}(y)$:

$$\{y = Y\} \text{malloc}(x, y) \{x = \text{null} * y = Y \vee \\ \exists u. x = u * u \mapsto \top[Y] * \text{b}(u) = u * \text{e}(u) = u + Y * y = Y\}$$

- ...

The variables $X, Y, \dots$, called **anchors**, are used to refer to the original values of pointer parameters in the post conditions.

Abduction Rules (1/3)

Solving the problem $\text{Current} * \text{Miss?} \models \text{Pre} * \text{true}$.

(The *Frame* is obtained as a by-product: the unused part of *Current*.)

- **Match** – matching points-to predicates:

$$\frac{\varphi * \zeta = \zeta' * [M] \triangleright \psi}{\varphi * \varepsilon \mapsto \zeta * [\zeta = \zeta' * M] \triangleright \psi * \varepsilon' \mapsto \zeta'} C$$

where $C : \varphi \models \varepsilon = \varepsilon' * \text{true}$ and $\text{size}(\zeta) = \text{size}(\zeta')$.

- **Slseg-pt-ls-right** (simplified) – matching a points-to against a simple list:

$$\frac{\varphi * \varepsilon \mapsto \zeta * [M] \triangleright \psi * \lambda[\varepsilon'/x, z/y] * \text{ls}_{\Lambda(x,y)}(z, \zeta')}{\varphi * \varepsilon \mapsto \zeta * [M] \triangleright \psi * \text{ls}_{\lambda(x,y)}(\varepsilon', \zeta')} C$$

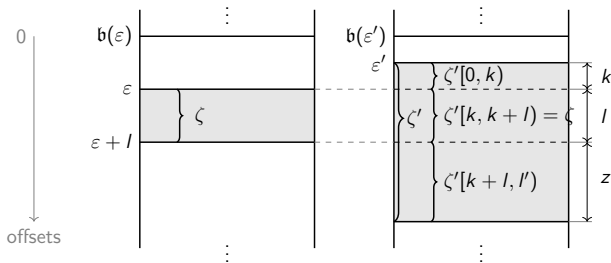
where $C : \varphi \models \mathbf{b}(\varepsilon) = \mathbf{b}(\varepsilon')$ and z is fresh.

Abduction rules (2/3)

- **Split-pt-pt-right** – splits a larger field to several smaller ones:

$$\frac{\varphi * \zeta = \zeta'[k, k + l) * [M] \triangleright \psi * \varepsilon' \mapsto \zeta'[0, k) * (\varepsilon + l) \mapsto \zeta'[k + l, l')}{\varphi * \varepsilon \mapsto \zeta * [\zeta = \zeta'[k, k + l) * M] \triangleright \psi * \varepsilon' \mapsto \zeta'} \quad C$$

$C : \varphi \models \mathbf{b}(\varepsilon) = \mathbf{b}(\varepsilon') * \varepsilon = \varepsilon' + k, \text{size}(\zeta) = l, \text{size}(\zeta') = l', \text{and } l + k \leq l'.$



Abduction rules (3/3)

- ...
- **Learn-finish** – finishing the abduction:

$$\frac{}{\varphi * [\psi] \triangleright \psi * \text{true}} C$$

$C : \varphi * \psi$ is SAT

- **Alias-strong** – introduce speculative aliasing:

$$\frac{\varphi * \chi(\varepsilon) * \varepsilon = \varepsilon' * [M] \triangleright \psi * \chi'(\varepsilon')}{\varphi * \chi(\varepsilon) * [\varepsilon = \varepsilon' * M] \triangleright \psi * \chi'(\varepsilon')} C$$

$C : \text{not } \varphi \models \varepsilon \neq \varepsilon'$

- Either used when **nothing else applies**
- or to **find more possible solutions**.