UNIVERSITY OF PASSAU

Faculty of Computer Science and Mathematics

Programming Paradigms: Type Systems

Summer Semester 2023

Dr. Abhishek Tiwari, Prof. Dr. Christian Hammer

- Detection of type errors, either at compile time or at runtime
  - type errors occur frequently in programs
  - type errors cannot be prevented/detected by grammar
  - if undetected, type errors can cause severe runtime errors
  - a type system can identify type errors before they occur

Lack of Type checks — Example

```
1. //simplified code
2. size_t lena = input();
3. size_t lenb = input();
4. //some more code
5. for (;;) {
6.  size_t len = lena + lenb + 2;
7.  if (ignore || translate) {
8.    char *copy_a = (char *) xnmalloc (len,
                      MB_CUR_MAX); //denial-of-service
9.    //some more code
10.  }
11.  //some more code
12. }
```

Lack of Type checks — Example

```
1. //simplified code
2. size_t lena = input();
3. size_t lenb = input();
4. //some more code
5. for (;;) {
6.  size_t len = lena + lenb + 2;
7.  if (ignore || translate) {
8.    char *copy_a = (char *) xnmalloc (len,
                        MB_CUR_MAX); //denial-of-service
9.    //some more code
10.  }
11.  //some more code
12. }
```

lena = INT_MAX

Lack of Type checks — Example

```
1. //simplified code
2. size_t lena = input();
3. size_t lenb = input();
4. //some more code
5. for (;;) {
6.  size_t len = lena + lenb + 2;
7.  if (ignore || translate) {
8.    char *copy_a = (char *) xnmalloc (len,
                      MB_CUR_MAX); //denial-of-service
9.   //some more code
10.  }
11.  //some more code
12. }
```

lena = INT_MAX

integer Overflow

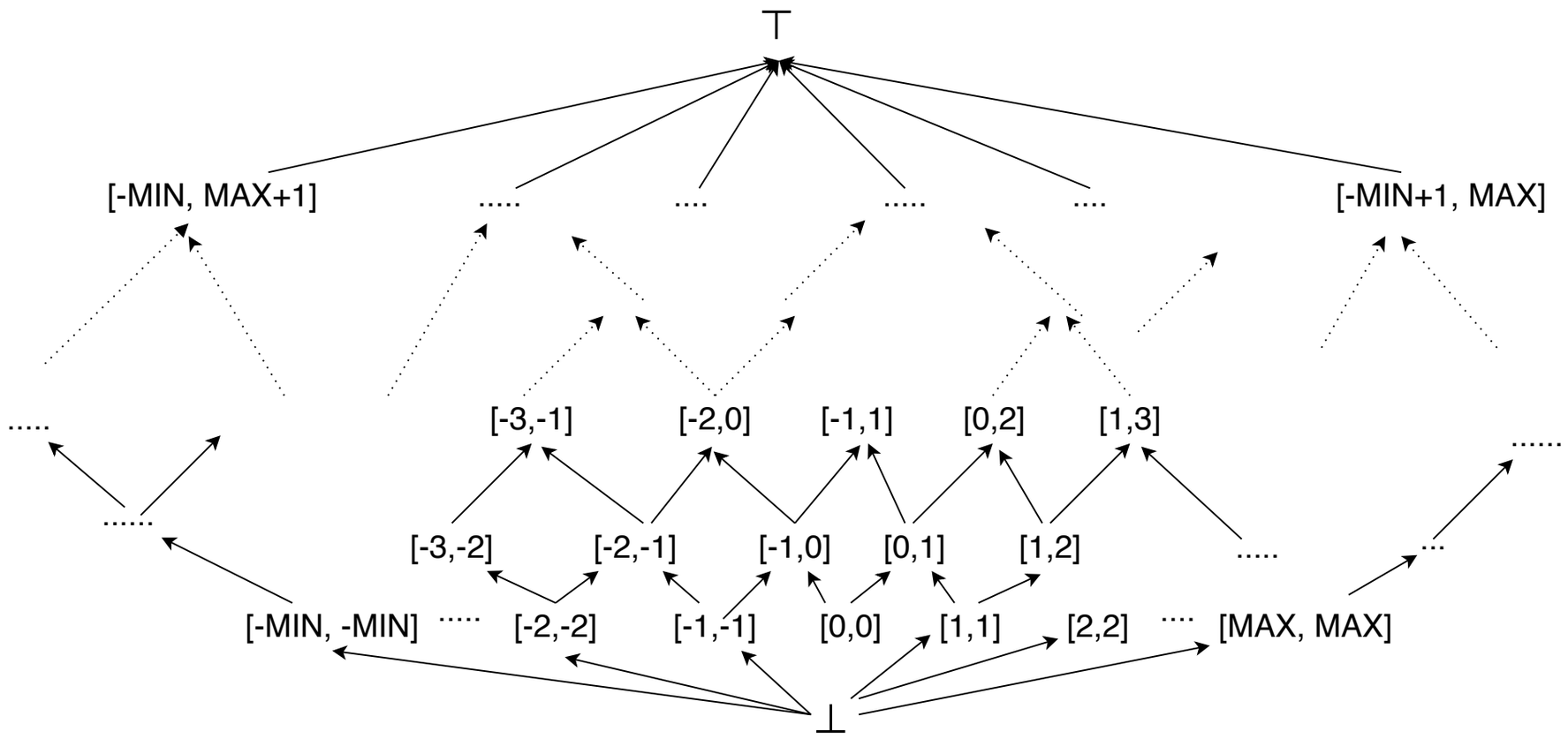
lenb + 2 < INT_MAX - lena

A tiny domain specific language— A C-type language

$$
\begin{array}{rcl}
s \in Stmt & ::= & D \mid v = e \mid \textbf{allocate}(p, c) \mid \textbf{free}\ (p) \mid \\
 & & \textbf{arraystore}(arr, i, v) \mid \textbf{arrayload}(arr, i, v) \mid \textbf{input(x)} \mid \\
 & & \textbf{if}\ (e)\ \{\overrightarrow{s_1}\}\ \textbf{else}\ \{\overrightarrow{s_2}\} \mid \textbf{while}(e)\ \{\overrightarrow{s}\} \mid v_n = func(\overrightarrow{v}) \mid \\
 & & s_1; s_2 \mid skip \\
e \in Exp & ::= & v \mid c \mid v_1 \odot v_2 \mid p \pm c \mid *p \\
D \in Decl & ::= & T\,var \mid T\ arr[n] \mid T\ *p
\end{array}
$$

$$
\begin{array}{rcl}
v \in Var & : & \text{Set of variables} \\
c, i \in Constants & : & \text{Set of integer, char, floating constants} \\
arr[T, n] \in Arrays & : & \text{array variable of type } T \\
T\ *p \in Var & : & \text{pointer variable of type } T \\
T \in Type & : & Integer, Float, Double, Character \\
n \in \mathbb{Z}_+ & : & \text{size of array} \\
\odot & : & \text{Denotes a binary operation}
\end{array}
$$

Integer Interval lattice



Let $\mathbb{Z}$ be the set of all integer and $\mathbb{Z}_B = \{z \mid z \in \mathbb{Z}, \mathbb{Z}_{min} \leq z, z \leq \mathbb{Z}_{max}\}$ denotes a set of integers bounded within the minimum and maximum elements, $\mathbb{Z}_{min}\, and\, \mathbb{Z}_{max}$

An interval $[l, h] = \{z \mid z \in \mathbb{Z}_B, l \leq z \leq h\}$ is the set of integers between (and including) l and h.

Intervals are ordered in the lattice as $[l_1, h_1] \leq [l_2, h_2]$ iff $(l_2 \leq l_1) \wedge (h_1 \leq h_2)$

$$\delta := [(\mathbb{I}_{\mathbb{Z}}, \leq), (\mathbb{I}_{\mathbb{R}}, \leq)]$$

$\mathbb{I}_z$: Integer Interval Lattice

$\mathbb{I}_R$: Floating point Interval Lattice

$$\sigma := \mathit{Variables} \mapsto \delta$$

$\delta := [(\mathbb{I}_{\mathbb{Z}}, \leq), (\mathbb{I}_{\mathbb{R}}, \leq)]$

$\mathbb{I}_z$: Integer Interval Lattice

$\mathbb{I}_R$: Floating point Interval Lattice

$\sigma := \mathit{Variables} \mapsto \delta$

int x = 12, y = 13; —> $\sigma := x \mapsto [12, 12],\ y \mapsto [13, 13]$

$\delta := [(\mathbb{I}_{\mathbb{Z}}, \leq), (\mathbb{I}_{\mathbb{R}}, \leq)]$

$\mathbb{I}_z$: Integer Interval Lattice

$\mathbb{I}_R$: Floating point Interval Lattice

$\sigma := Variables \mapsto \delta$

int x = 12, y = 13; —> $\sigma := x \mapsto [12, 12],\ y \mapsto [13, 13]$

int a[10] -> $\sigma := a \mapsto [0, 9]$

[(x,x)…(y,y)]

Derive Operational Semantics

$$[\textsc{InitVar}](\sigma, T\ v) \to (\sigma[v \leftarrow (\bot, \top)], skip) \quad [\textsc{InitArr}](\sigma, T\ arr[n]) \to (\sigma[arr \leftarrow [0, n-1]], skip) \quad [\textsc{InitPtr}](\sigma, T*\ p) \to (\sigma[p \leftarrow \bot], skip)$$

$$[\textsc{Unknown}](\sigma, v) \to (\sigma[v \leftarrow (\bot, \top)], skip) \quad [\textsc{Assign}]\frac{\sigma(v), \sigma(v') \in \delta}{(\sigma, v = v') \to (\sigma'[v \leftarrow \sigma(v')], skip)} \quad [\textsc{AssignConst}]\frac{\sigma(v) \in \delta \quad c' = [c, c]}{(\sigma, v = c) \to (\sigma'[v \mapsto c'], skip)}$$

$$[\textsc{Skip}](\sigma, skip) \to \sigma \quad [\textsc{Allocation}]\frac{\sigma(p) \in \delta}{(\sigma, allocate(p, c)) \to (\sigma'[p \leftarrow [0, c-1]], skip)} \quad [\textsc{Deallocation}]\frac{\sigma(p) \in \delta}{(\sigma, free(p)) \to (\sigma'[p \leftarrow \bot_{\mathbb{Z}}], skip)}$$

$$[\textsc{BinOp}]\frac{v' = v_1 \odot v_2}{(\sigma, v = v_1 \odot v_2) \to (\sigma'[v \mapsto v'], skip)} \quad [\textsc{IfTrue}]\frac{\sigma(e) \neq \{[0, 0]\}}{(\sigma, \textbf{if}\ (e)\ \{\overrightarrow{s_1}\}\ \textbf{else}\ \{\overrightarrow{s_2}\}) \to (\sigma, s_1)} \quad [\textsc{IfFalse}]\frac{\sigma(e) = \{[0, 0]\}}{(\sigma, \textbf{if}\ (e)\ \{\overrightarrow{s_1}\}\ \textbf{else}\ \{\overrightarrow{s_2}\}) \to (\sigma, s_2)}$$

$$[\textsc{IfUndef}]\frac{\sigma(e) \in \{[0, 0], [1, 1]\}}{(\sigma, \textbf{if}\ (e)\ \{\overrightarrow{s_1}\}\ \textbf{else}\ \{\overrightarrow{s_2}\}) \to ((\sigma, s_1) \sqcup (\sigma, s_2))} \quad [\textsc{WhileTrue}]\frac{e \neq 0}{(\sigma, \textbf{while}(e)\ \{\overrightarrow{s}\}) \to (\sigma, \overrightarrow{s}); \textbf{while}(e)\ \{\overrightarrow{s}\}}$$

$$[\textsc{WhileFalse}]\frac{e = 0}{(\sigma, \textbf{while}(e)\ \{\overrightarrow{s}\}) \to (\sigma, skip)} \quad [\textsc{Dynamic}]\frac{d = trace(s, v) \qquad s = \textbf{input}(v)}{(\sigma, \textbf{input}(v)) \to (\sigma'[v \mapsto d], skip)} \quad [\textsc{Compound}]\frac{(\sigma, s_1) \to \sigma'}{(\sigma, s_1; s_2) \to (\sigma', s_2)}$$

$$[\textsc{CompoundCont}]\frac{(\sigma, s_1) \to (\sigma', s_1')}{(\sigma, s_1; s_2) \to (\sigma', s_1'; s_2)}$$

Derive Operational Semantics

$$[\textsc{InitVar}](\sigma, T\ v) \to (\sigma[v \leftarrow (\bot, \top)], skip) \quad [\textsc{InitArr}](\sigma, T\ arr[n]) \to (\sigma[arr \leftarrow [0, n-1]], skip) \quad [\textsc{InitPtr}](\sigma, T*\ p) \to (\sigma[p \leftarrow \bot], skip)$$

$$[\textsc{Unknown}](\sigma, v) \to (\sigma[v \leftarrow (\bot, \top)], skip) \quad [\textsc{Assign}]\frac{\sigma(v), \sigma(v') \in \delta}{(\sigma, v = v') \to (\sigma'[v \leftarrow \sigma(v')], skip)} \quad [\textsc{AssignConst}]\frac{\sigma(v) \in \delta \quad c' = [c, c]}{(\sigma, v = c) \to (\sigma'[v \mapsto c'], skip)}$$

$$[\textsc{Skip}](\sigma, skip) \to \sigma \quad [\textsc{Allocation}]\frac{\sigma(p) \in \delta}{(\sigma, allocate(p, c)) \to (\sigma'[p \leftarrow [0, c-1]], skip)} \quad [\textsc{Deallocation}]\frac{\sigma(p) \in \delta}{(\sigma, free(p)) \to (\sigma'[p \leftarrow \bot_{\mathbb{Z}}], skip)}$$

$$[\textsc{BinOp}]\frac{v' = v_1 \odot v_2}{(\sigma, v = v_1 \odot v_2) \to (\sigma'[v \mapsto$$

$$[\textsc{Assign}]\frac{\sigma(v), \sigma(v') \in \delta}{(\sigma, v = v') \to (\sigma'[v \leftarrow \sigma(v')], skip)}$$

$$\frac{\sigma(e) = \{[0, 0]\}}{\sigma, \text{if } (e)\ \{\overrightarrow{s_1}\}\ \text{else}\ \{\overrightarrow{s_2}\}) \to (\sigma, s_2)}$$

$$[\textsc{IfUndef}]\frac{\sigma(e) \in \{[0, 0],}{(\sigma, \text{if } (e)\ \{\overrightarrow{s_1}\}\ \text{else}\ \{\overrightarrow{s_2}\}) \to ((\sigma, s_1) \sqcup (\sigma, s_2))} \quad [\textsc{WhileTrue}]\frac{e \neq 0}{(\sigma, \text{while}(e)\ \{\overrightarrow{s}\}) \to (\sigma, \overrightarrow{s}); \text{while}(e)\ \{\overrightarrow{s}\}}$$

$$[\textsc{WhileFalse}]\frac{e = 0}{(\sigma, \text{while}(e)\ \{\overrightarrow{s}\}) \to (\sigma, skip)} \quad [\textsc{Dynamic}]\frac{d = trace(s, v) \qquad s = \text{input}(v)}{(\sigma, \text{input}(v)) \to (\sigma'[v \mapsto d], skip)} \quad [\textsc{Compound}]\frac{(\sigma, s_1) \to \sigma'}{(\sigma, s_1; s_2) \to (\sigma', s_2)}$$

$$[\textsc{CompoundCont}]\frac{(\sigma, s_1) \to (\sigma', s_1')}{(\sigma, s_1; s_2) \to (\sigma', s_1'; s_2)}$$

Type System — Identify violations

$$[\textsc{Allocate}]\frac{\sigma(p) = \bot \vee \mathbb{I}_{\mathbb{Z}}}{[\mathcal{S}] \vdash \textbf{allocate}(p, n) \hookrightarrow \sigma(p) = [0, n-1]}$$

$$[\textsc{SFree}]\frac{\sigma(p) \neq \bot}{[\mathcal{S}] \vdash \textbf{free(p)} \hookrightarrow p = nullptr}$$

$$[\textsc{PointerDeref}]\frac{\sigma(p) \neq \bot}{[\mathcal{S}] \vdash v = *p \hookrightarrow v = *p}$$

$$[\textsc{PointerRef}]\frac{\sigma(p) \neq \bot \qquad [\mathcal{S}] \vdash v}{[\mathcal{S}] \vdash *p = v \hookrightarrow *p = v}$$

$$[\textsc{ArrStore}]\frac{0 \leq min(\sigma(i)) \wedge max(\sigma(i)) \leq \sigma(arr) \qquad [\mathcal{S}] \vdash v}{[\mathcal{S}] \vdash arraystore(arr, i, v) \hookrightarrow arr[i] = v}$$

$$[\textsc{ArrLoad}]\frac{0 \leq min(\sigma(i)) \wedge max(\sigma(i)) \leq \sigma(arr)}{[\mathcal{S}] \vdash arrayload(arr, i, v) \hookrightarrow v = arr[i]}$$

$$[\textsc{While}]\frac{[\mathcal{S}] \vdash s_i \qquad s_i \in \overrightarrow{s}}{[\mathcal{S}] \vdash \textbf{while}(e)\ \{\overrightarrow{s}\}}$$

$$[\textsc{IfElse}]\frac{[\mathcal{S}] \vdash s_i \qquad s_i \in \overrightarrow{s_1}, \overrightarrow{s_2}}{[\mathcal{S}] \vdash \textbf{if}\ (e)\ \{\overrightarrow{s_1}\}\ \textbf{else}\ \{\overrightarrow{s_2}\}}$$

$$[\textsc{ISafeAdd}]\frac{max(\sigma(v_2)) \leq (\mathbb{I}_{z,max} \ominus max(\sigma(v_1)))}{[\mathcal{S}] \vdash v_1 + v_2}$$

$$[\textsc{ISafeSub}]\frac{min(\sigma(v_2)) \leq (\mathbb{I}_{z,min} \oplus max(\sigma(v_1)))}{[\mathcal{S}] \vdash v_1 - v_2}$$

$$[\textsc{FSafeAdd}]\frac{max(\sigma(v_2)) \leq (\mathbb{I}_{f,max} \ominus max(\sigma(v_1)))}{[\mathcal{S}] \vdash v_1 + v_2}$$

$$[\textsc{FSafeSub}]\frac{min(\sigma(v_2)) \leq (\mathbb{I}_{f,min} \oplus max(\sigma(v_1)))}{[\mathcal{S}] \vdash v_1 - v_2}$$

$$[\textsc{SafeDiv}]\frac{[0, 0] \not\leq \sigma(v_2)}{[\mathcal{S}] \vdash v_1 / v_2}$$

$$[\textsc{SafeMul}]\frac{\sigma(v_2) \leq \frac{\top}{v_1}}{[\mathcal{S}] \vdash v_1 * v_2}$$

Type System— Identify violations

$$[\textsc{Allocate}]\frac{\sigma(p) = \bot \vee \mathbb{I}_{\mathbb{Z}}}{[\mathcal{S}] \vdash \textbf{allocate}(p, n) \hookrightarrow \sigma(p) = [0, n-1]} \qquad [\textsc{SFree}]\frac{\sigma(p) \neq \bot}{[\mathcal{S}] \vdash \textbf{free(p)} \hookrightarrow p = nullptr} \qquad [\textsc{PointerDeref}]\frac{\sigma(p) \neq \bot}{[\mathcal{S}] \vdash v = *p \hookrightarrow v = *p}$$

$$[\textsc{PointerRef}]\frac{\sigma(p) \neq \bot \qquad [\mathcal{S}] \vdash v}{[\mathcal{S}] \vdash *p = v \hookrightarrow *p = v} \qquad [\textsc{ArrLoad}]\frac{0 \leq min(\sigma(i)) \wedge max(\sigma(i)) \leq \sigma(arr) \qquad [\mathcal{S}] \vdash v}{\vdash arraystore(arr, i, v) \hookrightarrow arr[i] = v}$$

$$[\textsc{IfElse}]\frac{[\mathcal{S}] \vdash s_i \qquad s_i \in \overrightarrow{s_1}, \overrightarrow{s_2}}{[\mathcal{S}] \vdash \textbf{if}\ (e)\ \overrightarrow{s_1}\ \textbf{else}\ \overrightarrow{s_2}}$$

$$[\textsc{ISafeAdd}]\frac{max(\sigma(v_2)) \leq (\mathbb{I}_{z,max} \ominus max(\sigma(v_1)))}{[\mathcal{S}] \vdash v_1 + v_2} \qquad [\textsc{ISafeSub}]\frac{min(\sigma(v_2)) \leq (\mathbb{I}_{z,min} \oplus max(\sigma(v_1)))}{[\mathcal{S}] \vdash v_1 - v_2}$$

$$[\textsc{FSafeAdd}]\frac{max(\sigma(v_2)) \leq (\mathbb{I}_{f,max} \ominus max(\sigma(v_1)))}{[\mathcal{S}] \vdash v_1 + v_2} \qquad [\textsc{FSafeSub}]\frac{min(\sigma(v_2)) \leq (\mathbb{I}_{f,min} \oplus max(\sigma(v_1)))}{[\mathcal{S}] \vdash v_1 - v_2} \qquad [\textsc{SafeDiv}]\frac{[0,0] \not\leq \sigma(v_2)}{[\mathcal{S}] \vdash v_1 / v_2}$$

$$[\textsc{SafeMul}]\frac{\sigma(v_2) \leq \frac{\top}{v_1}}{[\mathcal{S}] \vdash v_1 * v_2}$$

$$[\textsc{SFree}]\frac{\sigma(p) \neq \bot}{[\mathcal{S}] \vdash \textbf{free(p)} \hookrightarrow p = nullptr}$$

p = null; (σ(p) = $\bot$)

…

free(p);

Type System — Identify violations

$$[\textsc{SFree}]\frac{\sigma(p) \neq \bot}{[\mathcal{S}] \vdash \textbf{free(p)} \hookrightarrow p = nullptr}$$

p = null; (σ(p) = $\bot$)

…

free(p);

An Example

(Op. Sym. , Type System)

```
1.  int x, check;
2.  int buf[10];
3.  scanf("%d", &check);
4.  x=check+1;
5.  int i = 0;
6.  while (i < x) {
7.    buf[i] = i;
8.    i=i+1;
9.  }
```

$[\mathrm{x} \mapsto \bot_{\mathbb{Z}}, \mathtt{check} \mapsto \bot_{\mathbb{Z}}])^{1}$ (InitVar, _)

$[x \mapsto \bot_{\mathbb{Z}}, check \mapsto \bot_{\mathbb{Z}}, buf \mapsto [0,9]]^{2}$ (InitArr, _)

$[x \mapsto \bot_{\mathbb{Z}}, check \mapsto [11,11], buf \mapsto [0,9]]^{3}$ (Dynamic)

$[x \mapsto [12,12], check \mapsto [11,11], buf \mapsto [0,9]]^{4}$ (BinOp, ISafeAdd)

$[x \mapsto [12,12], check \mapsto [11,11], buf \mapsto [0,9], i \mapsto [0,0]]^{5}$ (Assign,_)

$[x \mapsto [12,12], check \mapsto [11,11], buf \mapsto [0,9], i \mapsto [0,0]]^{6}$ (WhileTrue, While)

$[x \mapsto [12,12], check \mapsto [11,11], buf \mapsto [0,9], i \mapsto [0,0]]^{7}$ (Assign, ArrStore)

Type Checking Fails

(BinOp, ISafeAdd) $[x \mapsto [12,12], check \mapsto [11,11], buf \mapsto [0,9], i \mapsto [1,1]]^{8}$

(WhileTrue, While) $[x \mapsto [12,12], check \mapsto [11,11], buf \mapsto [0,9], i \mapsto [0,1]]^{6}$

(Assign, ArrStore) $[x \mapsto [12,12], check \mapsto [11,11], buf \mapsto [0,9], i \mapsto [0,2]]^{7}$

....

(BinOp, ISafeAdd) $[x \mapsto [12,12], check \mapsto [11,11], buf \mapsto [0,9], i \mapsto [10,10]]^{8}$

(WhileTrue, While) $[x \mapsto [12,12], check \mapsto [11,11], buf \mapsto [0,9], i \mapsto [0,10]]^{6}$

(Assign, ArrStore) $[x \mapsto [12,12], check \mapsto [11,11], buf \mapsto [0,9], i \mapsto [0,10]]^{8}$

```c
int check, i = 0;
int buf1[10], buf2[8];
scanf("%d", &check);
while (i < check) {
   buf1[i] = i + 1;
   ++i;
}
while (i >= 0) {
   buf2[i] = i + 1;
   --i;
}
```

Example 2

```
int check, i = 0;
int buf1[10], buf2[8];
scanf("%d", &check);
while (i < check) {
  buf1[i] = i + 1;
  ++i;
}
while (i >= 0) {
  buf2[i] = i + 1;
  --i;
}
```

$[\texttt{check} \mapsto \bot_{\mathbb{Z}}, i \mapsto [0,0]])^1$ (InitVar, _, Assign,_)

$[check \mapsto \bot_{\mathbb{Z}}, i \mapsto [0,0], buf1 \mapsto [0,9], buf2 \mapsto [0,7]]^2$ (InitArr, _)

$[check \mapsto [11,11], i \mapsto [0,0], buf1 \mapsto [0,9], buf2 \mapsto [0,7]]^3$ (Dynamic)

$[check \mapsto [11,11], i \mapsto [0,0], buf1 \mapsto [0,9], buf2 \mapsto [0,7]]^4$ (WhileTrue, While)

$[check \mapsto [11,11], i \mapsto [0,0], buf1 \mapsto [0,9], buf2 \mapsto [0,7]]^5$ (Assign, ArrStore)

(BinOp, ISafeAdd) $[check \mapsto [11,11], i \mapsto [1,1], buf1 \mapsto [0,9], buf2 \mapsto [0,7]]^6$

(WhileTrue, While) $[check \mapsto [11,11], i \mapsto [1,1], buf1 \mapsto [0,9], buf2 \mapsto [0,7]]^6$

(Assign, ArrStore)

....

(BinOp, ISafeAdd) $[check \mapsto [11,11], i \mapsto [10,10], buf1 \mapsto [0,9], buf2 \mapsto [0,7]]^6$

(WhileTrue, While) $[check \mapsto [11,11], i \mapsto [10,10], buf1 \mapsto [0,9], buf2 \mapsto [0,7]]^6$

(Assign, ArrStore) $[check \mapsto [11,11], i \mapsto [10,10], buf1 \mapsto [0,9], buf2 \mapsto [0,7]]^6$

Type Checking Fails

Failing Condition
$\sigma(i) \leq \sigma(buf1)$

(Op. Sym. , Type System)

$i \leq sizeof(buf1) - 1 \; ; (i \leq 9)$

Forward Propagation

```
while (i >= 0) {
  buf2[i] = i + 1;
  --i;
}
```

(WhileTrue, While) $[check \mapsto [11,11], i \leq 9, \ldots]$

(Assign, ArrStore) **Type Checking Fails**

Failing Condition
$\sigma(i) \leq \sigma(buf2)$

$i \leq sizeof(buf2) - 1 \rightarrow (i \leq 7);$

Updated Domain

Reference Materials

- Programming Languages: Principles and Paradigms by Allen B. Tucker and Robert E. Noonan