Lec 7. Abstract Interpretation Example 2.

Programs.

Language:

(포인터)

$lv \to x \mid *x$

$e \to n \mid lv \mid \& lv \mid e_1 + e_2 \mid e_1 \times e_2 \mid e_1 - e_2$

$b \to true \mid false \mid e_1 = e_2 \mid e_1 \le e_2$
$\quad \mid \neg b \mid b_1 \wedge b_2$

(memory allocation.)

$c \to lv := e \mid lv := alloc \mid skip \mid c_1 ; c_2$
$\quad \mid if\ b\ c_1\ c_2 \mid while\ b\ c$

- $(N, \to)$
  - N: 노드
  - $\to$: 노드와 노드 사이의 관계.

- Cmd(n)
  → $lv := e$
  $lv := alloc$
  assume (b)

ex)
```
x = 1;
p = &x;
*p = *p + 1;
```
→ { $x \mapsto$ ~~1~~ 2, $p \mapsto x$ } (lookup)

- assign 왼쪽에 있는 *p ⇒ p 자체의 위치
- assign 오른쪽에 있는 *p ⇒ p 안에 있는 값

ex)
```
p = alloc;
q = &p;
**q = 1;
```
→ { $p \mapsto l$, $l \mapsto$ ~~0~~ 1, $q \mapsto p$ } (lookup)

ex)
```
x = 1;
while (x < 10) {
  p = alloc
  *p = *p + x;
  x = x + 1;
}
```
→ { $x \mapsto 1$ }

① { $x \mapsto$ ~~1~~ 2, $p \mapsto l_1$, $l_1 \to$ ~~0~~ 1 }
↳ 정확히는 P를 lookup 후 l_1 값을 변경

② { $x \mapsto 3$, $p \mapsto l_2$, $l_1 \mapsto 1$, $l_2 \mapsto 2$. } ... ⇒ { $x \mapsto 10$, $p \mapsto l_9$, $l_1 \mapsto 1$, $l_2 \mapsto 2$, ⋮, $l_9 \mapsto 9$. }

Concrete Semantics

$Mem = Loc \to Val$ ← Mem은 Loc에 대한 값을 나타냄

$Loc = Var \mid HeapAddr$ ← Loc은 변수이거나 주소이거나

$Val = Int \mid Loc$ ← Val은 정수거나 Loc임

- $[\![lv]\!] : Mem \to Loc$
  → lv를 계산한다는 건 메모리 state를 받아서 Loc을 리턴

  $[\![x]\!](m) = x$ ← lv가 x라고 하면, 메모리를 받았을 때 이에 대한 Loc은 그냥 변수 Var임

  $[\![*x]\!](m) = m(x)$

  ↑ lv가 *x라고 하면, 메모리를 받았을 때 이에 대한 Loc은 heap address임.

- $[\![e]\!] : Mem \to Val$
  ↳ expression을 계산한다는건 메모리 state를 받아서 값을 리턴

  $[\![n]\!](m) = n$
  ↳ e가 n이라고 하면 메모리를 받았을 때 이에 대한 값은 그냥 정수임

  $[\![lv]\!](m) = m([\![lv]\!](m))$
  ↳ e가 lv라고 하면 메모리를 받았을 때 해당 위치에 대한 값을 리턴.

  $[\![\& lv]\!](m) = [\![lv]\!](m)$
  ↳ e가 &lv라고 하면 메모리를 받았을 때 그 위치 자체를 리턴

  $[\![e_1 + e_2]\!](m) = [\![e_1]\!](m) +_{Int} [\![e_2]\!](m)$

  ↑ 이걸 꼭 해야함
  → e에 포인터가 나올수도 있으나 해당 연산은 지원하지 않는다는 가정

- $[\![b]\!] : Mem \to Bool$

$$[\![true]\!](m) = true$$

$$[\![false]\!](m) = false$$

포인터 끼리의 equality는 없다고 가정.

$$[\![e_1 = e_2]\!](m) = [\![e_1]\!](m) =_{Int} [\![e_2]\!](m)$$

$$[\![\neg b]\!](m) = \neg [\![b]\!](m)$$

$$[\![b_1 \wedge b_2]\!](m) = [\![b_1]\!](m) \wedge [\![b_2]\!](m)$$

Input 메모리 집합을 받아 output 메모리 집합을 리턴.

- $fn : p(Mem) \to p(Mem)$ ← transfer function.

$$fn(M) = \{ m[[\![lv]\!](m) \mapsto [\![e]\!](m)] \mid m \in M \}$$

메모리들의 집합 (M)

lv가 뜻하는 로케이션을 먼저 계산

↳ assign 하는 주소에다가 e를 할당

메모리 집합에서 하나씩 꺼내서 update

$\cdots$ cmd(n) = lv := e.

$$fn(M) = \{ m[[\![lv]\!](m) \mapsto l, l \to 0] \mid m \in M \}$$

↳ lv을 하나 새로 할당해주고 그 로케이션 값을 0으로

$\cdots$ cmd(n) = lv := alloc

$$fn(M) = \{ m \in M \mid [\![b]\!](m) = true \}$$

↳ 메모리의 집합을 받아서

↳ 이 안에 있는 값들중 b의 값을 계산했을 때 true가 되는 애들만 살린다.

$\cdots$ cmd(n) = assume.

- $F : (N \to p(Mem)) \to (N \to p(Mem))$

↳ semantic function.

: Node별 reachable한 메모리 계산.

$$F(X) = \lambda n.\ fn\Big(\bigcup_{n' \to n} X(n') \Big)$$

n이랑 연관되어 있는 transfer function에 apply.

↳ $n' \to n$으로 오는 모든 predecessor n'을 모두 모아서

- Collecting semantics

$$fix\ F \in N \to p(Mem)$$

· Abstract Domain.

$Mem = Loc \to Val$

$Loc = Var \mid HeapAddr \to \widehat{Loc} = Var \mid AllocSite$

변수는 요약 X

$Val = Int \mid Loc$

Real Domain.

$\widehat{Mem} = \widehat{Loc} \to \widehat{Val}$

$\widehat{Val} = Interval \times p(\widehat{Loc})$

Abstract Domain

- $P(HeapAddr) \underset{\alpha_{HeapAddr}}{\overset{\gamma_{HeapAddr}}{\rightleftarrows}} P(AllocSite)$

$$\alpha_{HeapAddr}(H) = \{ allocsite(h) \mid h \in H \}$$

Heap 주소가 입력되면 이걸 요약한 것은

↳ location들이 태어난 지점(line number)로 요약.

$\Rightarrow \{l_1, l_2 \cdots l_n\} \to n$ (line number)

- $P(Loc) \underset{\alpha_{Loc}}{\overset{\gamma_{Loc}}{\rightleftarrows}} P(\widehat{Loc})$

출신을 기억하는 disjoint union

$$\alpha_{Loc}(L) = \{ x \mid x \in L \} \uplus \alpha_{HeapAddr}(\{ h \mid h \in L \})$$

Loc의 집합 → Loc은 변수 or HeapAddr.

변수는 요약을 안하니 그대로 가져가고

Heap address를 모아서 allocation site로 요약.

- $P(Int) \underset{\alpha_{Int}}{\overset{\gamma_{Int}}{\rightleftarrows}} Interval$

- $P(Val) \underset{\alpha_{Val}}{\overset{\gamma_{Val}}{\rightleftarrows}} \widehat{Val}$

값은 정수거나, Loc임

Integer 값을 모아서 Interval로 요약.

$$\alpha_{Val}(V) = \langle \alpha_{Int}(\{ z \mid z \in V \}),\ \alpha_{Loc}(\{ l \mid l \in V \}) \rangle$$

☆ Interval과 Loc의 tuple로 정의

, → Union이 아니라 product인 이유는 static type system이 아닌 python 같은 경우 → 이런식으로 될 수도 있기 때문

Branch에서 → x=1 / x=&p

- $P(Mem) \underset{\alpha_{Mem}}{\overset{\gamma_{Mem}}{\rightleftarrows}} \widehat{Mem}$

$$\alpha_{Mem}(M) = \lambda l. \begin{cases} \bigsqcup \alpha_{Val}\{ m(l) \mid m \in M \} & \cdots\ l \in Var \\ \bigsqcup \{ m(a) \mid m \in M, a \in \gamma_{HeapAddr}(l) \} & \cdots\ l \in AllocSite \end{cases}$$

메모리 집합을 받아서 Abstract 메모리 하나로 요약

→ 입력으로 들어온 var에 대해서 var가 가지는 값들을 모두 모아서 value abstraction 수행

↳ 임의의 Heap Address

concretize 해서 실제 어떤 heapaddress인지를 봄.

- $\underbrace{N \to p(Mem)}_{D} \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} \underbrace{N \to \widehat{Mem}}_{\hat{D}}$

$$: \alpha(X) = \lambda n.\ \alpha_{Mem}(X(n))$$

ex) $\alpha_{Mem}\left(\left\{ \begin{bmatrix} x \mapsto 1 \\ y \mapsto x \end{bmatrix}, \begin{bmatrix} x \mapsto 2 \\ y \mapsto l_1 \\ l_1 \mapsto 3 \end{bmatrix} \right\}\right)$

$$\rightarrow = \begin{bmatrix} x \mapsto \langle [1,2], \emptyset \rangle \\ y \mapsto \langle \bot, \{x, n_1\} \rangle \\ n_1 \rightarrow \langle [3,3], \emptyset \rangle \end{bmatrix}$$

(n_1 ← l_1의 allocsite)

- **Abstract Semantics**.

- $[\![lv]\!] : \widehat{Mem} \rightarrow P(\widehat{Loc})$
  - $[\![x]\!](m) = \{x\}$ ← x는 abstract location 계산하면 그냥 자기 자신임.
  - $[\![*x]\!](m) = m(x).2$ ← 메모리에서 x를 lookup 한 다음에 거기서(tuple) 그렇게 원소임.

- $[\![b]\!] : \widehat{Mem} \rightarrow Bool$
  - $[\![true]\!](m) = true$
  - $[\![false]\!](m) = false$
  - $[\![e_1 = e_2]\!](m) = [\![e_1]\!](m) \hat{=} [\![e_2]\!](m)$
  - $[\![e_1 \leq e_2]\!](m) = [\![e_1]\!](m) \hat{\leq} [\![e_2]\!](m)$

    $[\![\neg b]\!](m) = \neg [\![b]\!](m)$
  - $[\![b_1 \wedge b_2]\!](m) = [\![b_1]\!](m) \wedge [\![b_2]\!](m)$

- $[\![e]\!] : \widehat{Mem} \rightarrow \widehat{Val}$

  $[\![n]\!](m) = \langle [n,n], \emptyset \rangle$

  $[\![lv]\!](m) = \bigsqcup_{l \in [\![lv]\!](m)} m(l)$ ← $\{n_1, n_2\}$가 있었다고 가정, m에서 모두 lookup후 join

  $[\![\& lv]\!](m) = \langle \bot, [\![lv]\!](m) \rangle$

  $[\![e_1 + e_2]\!](m) = [\![e_1]\!](m) \hat{+} [\![e_2]\!](m)$

- $\hat{f}_n : \widehat{Mem} \rightarrow \widehat{Mem}$
  - $\hat{f}_n(m) = m[x \mapsto [\![e]\!](m)]$ ⋯ $cmd(n) = x := e$ ← 그냥 바꿔치기 (strong update)
  - $\hat{f}_n(m) = m[x \mapsto [\![e]\!](m)]$ ⋯ $cmd(n) = lv := e$, $[\![lv]\!](m) = \{x\}$

    ↳ 포인터가 가리키는 주소가 1개 밖에 없을 때. strong update
  - $\hat{f}_n(m) = \bigsqcup_{l \in [\![lv]\!](m)} m[l \mapsto m(l) \sqcup [\![e]\!](m)]$ ⋯ $cmd = lv := e$

    ↳ 포인터가 가리키는 주소가 여러개일 때. weak update
  - $\hat{f}_n(m) = m[x \mapsto (\bot, \{n\}), n \mapsto ([0,0], \emptyset)]$ ⋯ $cmd = x := alloc$.
  - $\hat{f}_n(m) = m[x \mapsto (\bot, \{n\}), n \mapsto ([0,0], \emptyset)]$ ⋯ $cmd = lv := alloc$, $[\![lv]\!](m) = \{x\}$

    ↳ strong update
  - $\hat{f}_n(m) = \bigsqcup_{l \in [\![lv]\!](m)} m'[l \mapsto m(l) \sqcup (\bot, \{n\})]$ ⋯ $cmd(n) = lv := alloc$, $m' = m[n \mapsto ([0,0], \emptyset)]$

    ↳ weak update
  - $\hat{f}_n(m) = \bigsqcup \{m' \sqsubseteq m \mid true \sqsubseteq [\![b]\!](m')\}$ ⋯ $cmd(n) = assume(b)$

    ← true가 될 가능성이 있는 m'을 (abstract memory) 살려서 join.

- $\hat{F} : (N \rightarrow \widehat{Mem}) \rightarrow (N \rightarrow \widehat{Mem})$:

  $\hat{F}(X) = \lambda n. \hat{f}_n\left(\bigsqcup_{n' \rightarrow n} X(n')\right)$

  Flow sensitive. 각각의 program location 별로 생성

- Abstract semantics

  $\bigsqcup_{i \geq 0} \hat{F}^i(\bot) \in N \rightarrow \widehat{Mem}$

  ↳ $\nabla \rightarrow \Delta$