Ilya Shervin 308640218
Dmitry Kuznetsov 322081183

Problem Set 2

1 Pointer Analysis

The states of the concrete semantics used in this section are functions $S = \text{Loc} \to \text{Loc} \cup Z$. The abstract domain in this section is $A = 2^{\text{Var}^* \times \text{Var}^*}$ and the abstraction function (α) is defined by means of an extraction function (β), where $\beta(s) = \{(x, y) \mid s(loc(x)) = loc(y)\}$. The function $loc : \text{Var}^* \to \text{Loc}$ returns the "address" of each variable.

Recall that as usual in cases in which the Galois connection induced by an extraction function, $\alpha(S) = \cup\{\beta(s) \mid s \in S\}$, and $\gamma(a) = \{s \in 2^{\text{Var}^* \times \text{Var}^*} \mid \beta(s) \subseteq a\}$.

1.1 Question 1

The concrete semantics of the statement $x = y$ is

$$[\![x = y]\!](s) = s[loc(x) \mapsto s(loc(y))]$$

. The abstract transformer associated with this statement is

$$[\![x = y]\!]^\sharp(a) = a \backslash \{(x, z) \mid z \in \text{Var}^*\} \cup \{(x, w) \mid (y, w) \in a\}$$

. Show that the abstract transformer is the best, e.g.

$$[\![x = y]\!]^\sharp(a) = \alpha(\{[\![x = y]\!](s) \mid s \in \gamma(a)\})$$

for any $a \in A$.

Proof. We'll show equality by means of bi-directional set inclusion.

Direction 1 $[\![x = y]\!]^\sharp(a) \subseteq \alpha(\{[\![x = y]\!](s) \mid s \in \gamma(a)\})$

Let $\Delta \in [\![x = y]\!]^\sharp(a) = a \backslash \{(x, z) \mid z \in \text{Var}^*\} \cup \{(x, w) \mid (y, w) \in a\}$, we'll treat 2 cases:

- $\Delta = (x, w)$ then from abstract transformer definition exists (y, w) in a.

  Let us construct a concrete state s in the following form: $s(loc(y)) = loc(w)$ and undefined for all other symbols. It holds that $\beta(s) = \{(y, w)\} \subseteq a$ and therefore $s \in \gamma(a)$.

  We now use the transformer on s and get $s' = s[loc(x) \mapsto s(loc(y))] = s[loc(x) \mapsto loc(w)]$. We got $s' \in \{[\![x = y]\!](s) \mid s \in \gamma(a)\}$ and also $s'(loc(x)) = loc(w)$.

  Finally, it holds that

  $$\Delta = (x, w) \in \beta(s') = \alpha(\{s'\}) \subseteq \alpha(\{[\![x = y]\!](s) \mid s \in \gamma(a)\})$$

- $\Delta = (z, w)$ where $z \neq x$ then from abstract transformer definition $(z, y) \in a$.

  From that we will construct a concrete state s. It will be defined as $s(loc(z)) = loc(w)$ and undefined for all other symbols, it holds that $\beta(s) = \{(z, w)\} \subseteq a$ thus $s \in \gamma(a)$.

  If we transform s with $[\![x = y]\!]$ we'll get $s' = s[loc(x) \mapsto s(loc(y))]$.

  It is true that $s' \in \{[\![x = y]\!](s) \mid s \in \gamma(a)\}$, and $s'(loc(w)) = loc(z)$ therefore:

  $$\Delta = (w, z) \in \beta(s') = \alpha(\{s'\}) \subseteq \alpha(\{[\![x = y]\!](s) \mid s \in \gamma(a)\})$$

Direction 2 $[\![x = y]\!]^\sharp(a) \supseteq \alpha(\{[\![x = y]\!](s) \mid s \in \gamma(a)\})$

Let $\Delta \in \alpha(\{[\![x = y]\!](s) \mid s \in \gamma(a)\})$ then exists $s' \in \{[\![x = y]\!](s) \mid s \in \gamma(a)\}$ s.t. $\Delta \in \beta(s') = \{(a,b) \mid s'(loc(a)) = loc(b)\}$. From the original equation we also know that exists $s \in \gamma(a)$ s.t. $s' = s[loc(x) \mapsto s(loc(y))]$. Similarly, we'll treat 2 cases here:

- $\Delta = (x, z)$ then $s'(loc(x)) = loc(z)$. It then must hold that $s(loc(y)) = loc(z)$ (as $s' = s[loc(x) \mapsto s(loc(y))]$).

  Therefore, $(y, z) \in \beta(s) \subseteq a$. Knowing that $(y, z) \in a$ we can deduce that
  $$(x, z) \in \{(x, w) \mid (y, w) \in a\} \subseteq [\![x = y]\!]^\sharp(a)$$
- $\Delta = (w, z)$ for any $w \neq z$ then $s'(loc(w)) = loc(z)$ and from s''s relation with s also $s(loc(w)) = loc(z)$.

  Therefore $(w, z) \in \beta(s) \subseteq a$. Knowing that $(w, z) \in a$ we can deduce that $\Delta \notin \{(x, z) \mid z \in \text{Var}^*\}$ therefore $\Delta \in [\![x = y]\!]^\sharp(a)$.

□

1.2 Question 2

The abstract transformer of simple assignment
$$[\![x = y]\!]^\sharp(a) = a\backslash\{(x, z) \mid z \in \text{Var}^*\} \cup \{(x, w) \mid (y, w) \in a\}$$
is distributive, i.e.,
$$\forall a_1, a_2 \in A : [\![x = y]\!]^\sharp(a_1) \sqcup [\![x = y]\!]^\sharp(a_2) = [\![x = y]\!]^\sharp(a_1 \sqcup a_2)$$

Proof.

$$\begin{aligned}
&c[\![x = y]\!]^\sharp(a_1) \sqcup [\![x = y]\!]^\sharp(a_2) = \\
&= (a_1\backslash\{(x, z) \mid z \in \text{Var}^*\} \cup \{(x, w) \mid (y, w) \in a_1\}) \cup (a_2\backslash\{(x, z) \mid z \in \text{Var}^*\} \cup \{(x, w) \mid (y, w) \in a_2\}) \\
&= ((a_1 \cup a_2)\backslash\{(x, z) \mid z \in \text{Var}^*\}) \cup \{(x, w) \mid (y, w) \in (a_1 \cup a_2)\} \\
&= ((a_1 \sqcup a_2)\backslash\{(x, z) \mid z \in \text{Var}^*\}) \cup \{(x, w) \mid (y, w) \in (a_1 \sqcup a_2)\} \\
&= [\![x = y]\!]^\sharp(a_1 \sqcup a_2)
\end{aligned}$$

□

1.3 Question 3

The abstract transformer of the statement
$$[\![*x = y]\!]^\sharp(a) = a \cup \{(t, z) \mid (x, t) \in a, (y, z) \in a\}$$
is not distributive, i.e. exists $a_1, a_2 \in A$ s.t.
$$[\![*x = y]\!]^\sharp(a_1) \sqcup [\![*x = y]\!]^\sharp(a_2) \neq [\![*x = y]\!]^\sharp(a_1 \sqcup a_2)$$

Proof. We'll show the sets are not equal by showing elements present in $[\![*x = y]\!]^\sharp(a_1 \sqcup a_2)$ but not in $[\![*x = y]\!]^\sharp(a_1) \sqcup [\![*x = y]\!]^\sharp(a_2)$.
Let a_1 s.t.

- $(x, t_1), (y, w_1) \in a_1$
- $(x, t_2), (y, w_2) \notin a_1$

additionally, let a_2 s.t.

- $(x, t_1), (y, w_1) \notin a_2$
- $(x, t_2), (y, w_2) \in a_2$

then, it holds that:

- $(t_1, w_1) \in [\![*x = y]\!]^\sharp(a_1)$
- $(t_2, w_2) \in [\![*x = y]\!]^\sharp(a_2)$
- $(t_2, w_2) \notin [\![*x = y]\!]^\sharp(a_1)$
- $(t_1, w_1) \notin [\![*x = y]\!]^\sharp(a_2)$

therefore also $(t_2, w_2), (t_1, w_1) \notin [\![*x = y]\!]^\sharp(a_1) \sqcup [\![*x = y]\!]^\sharp(a_2)$.
Conversely, $(t_2, w_2), (t_1, w_1) \in [\![*x = y]\!]^\sharp(a_1 \sqcup a_2)$. □

2 Shape Analysis

In the 3-valued logic framework for shape analysis, the user needs to provide the update formulae which describe the effect of every program statement on the core and instrumentation predicates. In the class, we defined at the update formulae for the core predicates for list-manipulating programs. Define the update formulae for the instrumentation predicates capturing the properties: reach-ability from variable x, heap-sharing (is-shared), and cyclicity for list manipulating programs. Assume that the pointer variables of the program are x, y and z.

- $r_x(v)$ is the predicate that means that node v is reachable from variable x. First, let us define the update formulae for statements that do not change `next` predicate:
  - `x = NULL`:
    $r'_x(v) = 0$
    `x` is null, therefore no node is reachable.

- `x = malloc()`:
  $r'_x(v) = x(v) = isNew(v)$
  `x` is a newly allocated variable, therefore only its node is reachable.
- `x = y`:
  $r'_x(v) = r_y(v)$
  A node v is reachable from `x` if and only if it is reachable from `y`.
- `x = y−>next`:
  $r'_x(v) = (\neg y(v) \wedge r_y(v)) \vee (y(v) \wedge c(v))$
  A node v is reachable from `x` if it is reachable from `y` and is not `y`'s node, unless `y`'s node is on a cycle and hence also reachable from `y−>next`.

Now, consider the case `x−>next = y`. Apart from r_x being updated, for every other variable `z` we need to update r_z. We break it into two statements `x−>next = NULL; x−>next = y`. The update formula for the second statement assumes that `x−>next` is null.

- `x−>next = NULL`:
  * $r'_x(v) = x(v)$
    Only `x`'s node is now reachable from `x`.
  * $r'_z(v) = \begin{cases} \exists v'.x(v') \wedge n^*(v',v), & \text{if } (\exists v'.x(v') \wedge r_z(v')) \wedge r_x(v) \wedge c(v) \\ 0, & \text{if } (\exists v'.x(v') \wedge r_z(v')) \wedge r_x(v) \wedge \neg c(v) \\ r_z(v), & \text{if } \neg(\exists v'.x(v') \wedge r_z(v')) \vee \neg r_x(v) \end{cases}$
    If `x`'s node is reachable from `z` (denoted by $\exists v'.x(v') \wedge r_z(v')$), and v was reachable from `x`, and v was part of a cycle, we must compute $r'_z(v)$ anew, because it might be the case that v is reachable from `z` after `x`, so it won't be reachable any longer, or before `x`, and it will still be reachable.

    If `x`'s node is reachable from `z`, and v was reachable from `x`, and v was not part of a cycle, it won't be reachable any longer.

    Otherwise, `x`'s node is not reachable from `z`, or v was not reachable from `x`, so executing the statement won't change $r_z(v)$.
- `x−>next = y`:
  * $r'_x(v) = x(v) \vee r_y(v)$
    `x`'s node is reachable along with nodes that are reachable from `y`.
  * $r'_z(v) = (\exists v'.x(v') \wedge r_z(v')) \wedge r_y(v)$
    A node v is reachable if `x` is reachable from `z` and v is reachable from `y`

- $c(v)$ is the predicate that means that node v is part of a cycle. Note that statements that do not change `next` predicate do not affect $c(v)$. So we consider only `x−>next = y`. Again we break it into two statements `x−>next = NULL; x−>next = y`.

– $\texttt{x}- > \texttt{next} = \texttt{NULL}$:
$$c'(v) = \begin{cases} 0, & \text{if } \exists v'.x(v') \wedge r_x(v) \wedge c(v') \\ c(v), & \text{otherwise} \end{cases}$$
If v was reachable from `x`'s node and `x`'s node was on a cycle, it means that v was on that cycle as well and now the cycle is cut. Otherwise the value is unchanged

– $\texttt{x}- > \texttt{next} = \texttt{y}$:
$$c'(v) = \begin{cases} 1, & \text{if } \exists v'.x(v') \wedge r_y(v') \wedge r_y(v) \\ c(v), & \text{otherwise} \end{cases}$$
If v is reachable from `y`'s node and also `x`'s node is reachable from `y`'s node, it means that a new cycle is created with v on it. Otherwise the value is unchanged

- $is(v)$ is the predicate that means that at least two different nodes point with `next` predicate to v (v is heap-shared). Note that statements that do not change `next` do not affect $is(v)$. So we consider only $\texttt{x}- > \texttt{next} = \texttt{y}$. Again we break it into two statements
$\texttt{x}- > \texttt{next} = \texttt{NULL}; \texttt{x}- > \texttt{next} = \texttt{y}$.

  – $\texttt{x}- > \texttt{next} = \texttt{NULL}$:
$$is'(v) = \begin{cases} \exists v_1, v_2.n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2, & \text{if } (\exists v'.x(v') \wedge n(v', v)) \wedge is(v) \\ is(v), & \text{otherwise} \end{cases}$$
If `x`'s node pointed directly to v and $is(v) = 1$, we have to recompute $is(v)$ since now it might be pointed by less than two nodes. Otherwise the value is unchanged.

  – $\texttt{x}- > \texttt{next} = \texttt{y}$:
$$is'(v) = \begin{cases} \exists v_1, v_2.n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2, & \text{if } y(v) \wedge \neg is(v) \\ is(v), & \text{otherwise} \end{cases}$$
If we happen to update `y`'s node and also $is(v) = 0$, we have to recompute $is(v)$ since now it might be pointed by two different nodes. Otherwise the value is unchanged.

3 Owiki Gries Logic

Give a (non-trivial) specification for the following program and prove it using Owicki-Griel logic
$$\{X = A \wedge Y = B\} x := X; Y := 1 || y := Y; X := x \ldots$$

Proof. We will prove the following specification for the program:

$$\{\ X = A \wedge Y = B\ \}\ x := X; Y := 1 || y := Y; X := x\ \{\ x = A \vee y = B\ \}$$

First, add two auxiliary variables $done_0$ and $done_1$ that record the state of execution of first and second statements accordingly. Now, rewrite the program as:

$$done_0 = 0; done_1 = 0; (x := X; (Y, done_0) := (1, 1) || y := Y; (X, done_1) := (x, 1))$$

and let $S_0 = x := X; (Y, done_0) := (1, 1)$ and $S_1 = y := Y; (X, done_1) := (x, 1)$. Define $S_{00} = x := X$, $S_{01} = (Y, done_0) := (1, 1)$, $S_{10} = y := Y$ and $S_{11} = (X, done_1) := (x, 1)$. Now we can define the pre conditions of each sub statement:

$$\begin{aligned}
P_{00} &= done_0 = 0 \wedge (done_1 = 0 \rightarrow X = A) \wedge (done_1 = 1 \rightarrow (x = A \vee y = B)) \\
P_{01} &= done_0 = 0 \wedge (done_1 = 0 \rightarrow x = A) \wedge (done_1 = 1 \rightarrow (x = A \vee y = B)) \\
P_{10} &= done_1 = 0 \wedge (done_0 = 0 \rightarrow Y = B) \wedge (done_0 = 1 \rightarrow (x = A \vee y = B)) \\
P_{11} &= done_1 = 0 \wedge (done_0 = 0 \rightarrow y = B) \wedge (done_0 = 1 \rightarrow (x = A \vee y = B))
\end{aligned}$$

Also, define the post conditions:

$$\begin{aligned}
Q_0 &= done_0 = 1 \wedge (done_1 = 0 \rightarrow x = A) \wedge (done_1 = 1 \rightarrow (x = A \vee y = B)) \\
Q_1 &= done_1 = 1 \wedge (done_0 = 0 \rightarrow y = B) \wedge (done_0 = 1 \rightarrow (x = A \vee y = B))
\end{aligned}$$

P_{00} and P_{10} describe the state of events before the assignment to x or y took place, and P_{10} and P_{11} after, accordingly. After the counterpart statement is done, we can have both x and y updated, or only one of them.

Now we wish to show that S_0 and S_1 are interference free. To do this we need to prove the following to show that S_0 is interference free:

$$\{\ P_{00} \wedge Q_1\ \}\ S_0\ \{\ Q_1\ \} \tag{1}$$
$$\{\ P_{00} \wedge P_{10}\ \}\ S_0\ \{\ P_{10}\ \} \tag{2}$$
$$\{\ P_{00} \wedge P_{11}\ \}\ S_0\ \{\ P_{11}\ \} \tag{3}$$

Theorem (1) is the post condition test and theorems (2) and (3) are the pre condition tests that check each "sub" pre-condition in S_1. And we need to prove the following to show that S_1 is interference free:

$$\{\ P_{10} \wedge Q_0\ \}\ S_1\ \{\ Q_0\ \} \tag{4}$$
$$\{\ P_{10} \wedge P_{00}\ \}\ S_1\ \{\ P_{00}\ \} \tag{5}$$
$$\{\ P_{10} \wedge P_{01}\ \}\ S_1\ \{\ P_{01}\ \} \tag{6}$$

We are going to use the following identities:

$$\begin{aligned}
Q_0 \wedge Q_1 &= (x = A \vee y = B) \wedge done_0 = 1 \wedge done_1 = 1 \\
P_{00} \wedge Q_1 &= (x = A \vee y = B) \wedge done_0 = 0 \wedge done_1 = 1 \\
P_{10} \wedge Q_0 &= (x = A \vee y = B) \wedge done_0 = 1 \wedge done_1 = 0 \\
P_{00} \wedge P_{10} &= X = A \wedge Y = B \wedge done_0 = 0 \wedge done_1 = 0 \\
P_{00} \wedge P_{11} &= X = A \wedge y = B \wedge done_0 = 0 \wedge done_1 = 0 \\
P_{10} \wedge P_{01} &= x = A \wedge Y = B \wedge done_0 = 0 \wedge done_1 = 0
\end{aligned}$$

- For (1), we have:

$$\{(x = A \lor y = B) \land done_0 = 0 \land done_1 = 1\}$$
$$x := X; (Y, done_0) := (1, 1)$$
$$\{done_1 = 1 \land (done_0 = 0 \to y = B) \land (done_0 = 1 \to (x = A \lor y = B))\}$$

  We see that $done_0 = 1$ at the end, and we have $y = B$ untouched and obviously $x = A \lor y = B$.

- For (2), we have

$$\{X = A \land Y = B \land done_0 = 0 \land done_1 = 0\}$$
$$x := X; (Y, done_0) := (1, 1)$$
$$\{done_1 = 0 \land (done_0 = 0 \to Y = B) \land (done_0 = 1 \to (x = A \lor y = B))\}$$

  Use the axiom of Hoare logic $\{ X = A \}$ $x := X$ $\{ x = A \}$, along with $done_0 = 1$ at the end, to see that $x = A$ holds.

- For (3), we have

$$\{X = A \land y = B \land done_0 = 0 \land done_1 = 0\}$$
$$x := X; (Y, done_0) := (1, 1)$$
$$\{done_1 = 0 \land (done_0 = 0 \to y = B) \land (done_0 = 1 \to (x = A \lor y = B))\}$$

  The idea is similar to (2). $done_0 = 1$ at the end, and $x = A$ holds.

The idea behind (4), (5) and (6) is symmetrical to (1), (2) and (3), so the proof is skipped.

Now we can show the inference tree for the program □