1. Definitions

We'll start by defining the collecting semantics for our programming language. We do that directly instead of defining the operational semantics because of the non-determinism (? assignments) which have better abstractions in collecting semantics.
Let VARS = $\{v_1, ..., v_n\}$ represent the list of variables used in the program. We define the base domain as $S :=$ VARS $\to \mathbb{Z}$, and the collecting domain as $D := P(S)$. The collecting domain is a complete lattice as both $\sqcup$ and $\sqcap$ are well defined for sets.
For an If $s \in D$, we'll use the following notation:

$$s[x \to K] := \{d[x \to K] \mid d \in s\}$$
$$s[x \to f(s(y))] := \{d[x \to f(d(y))] \mid d \in s\}$$
$$s[x \to ?] := \cup_{n \in \mathbb{N}}.s[x \to n]$$

Let's define the semantics of the atomic operations:

- [[skip]]$s = s$
- $[[i := j]]s = s[i \to s(j)]$
- $[[i :=?]]s = s[i \to ?]$
- $[[i := K]]s = s[i \to K]$
- $[[i := j \pm 1]]s = s[i \to s(j) \pm 1]$
- [[assert ORC]]$s = s$ (as assertions are considered external checks and not part of the program)
- [[assume E]]$s = \{d \in s \mid [[E]]d$ is true $\}$ where $[[E]]d$ is defined as expected (replacing variable with $d(variable)$)

Note that $[[S]]$ transformer for each S is monotonic. The only non-trivial case is the unknown assignment. However, using the equivalent definition:

$$[[i :=?]]s = \bigcup_{n \in \mathbb{N}} [[i := n]]s$$

It has become trivial too.
Now, one can run any program in our programming language. Starting by the vertex without in-edges, and going over the edges until reaching the vertex with the no out-edges. It easy to see that the language is deterministic. However, as the language is Turing-Complete, the program may never stop.

2. Parity Analysis

a) Domain

We are going to use $A := Disj(\underset{v_i \in \text{Vars}}{\times} \text{Parity})$ as our abstract domain, when Parity is $\{0, 1\}$
$Disj(X)$ is always a complete lattice. Note: We may sometimes treat $\underset{v_i \in \text{Vars}}{\times}$ Parity as Vars $\to Parity$ as those are equivalent definitions.

Now, we have to define the semantics on the abstract domain. We are going to use the same syntactic sugar as before, of:

$$s[x \to K] := \{d[x \to K] \mid d \in s\}$$
$$s[x \to f(s(y))] := \{d[x \to f(d(y))] \mid d \in s\}$$

- [[skip]]$s = s$
- $[[i := j]]s = s[i \to s(j)]$
- $[[i :=?]]s = s[i \to 0] \sqcup s[i \to 1]$
- $[[i := K]]s = s[i \to K\%2]$
- $[[i := j \pm 1]]s = s[i \to (s(j) \pm 1)\ \%\ 2)]$
- [[assert ORC]]$s = s$
- [[assume E]]$s = \{d \in s \mid [[E]]d$ is true $\}$ where $[[E]]d$ is defined as expected, this time on the abstract domain (replacing variable with $d(variable)$).

Note that $[[S]]$ abstract transformer for each S is monotonic. This time the unknown-assignment has the same structure as the others, so every one of them is trivial to prove that is monotone.

b) Galois Connection

The next step is building the Galois connection. As shown in EX2 Q2, it is suffice to build $\beta : S \to A$ in order to build a Galois connection.

$$\beta(s) = \{\lambda v_i.s(v_i)\ \%\ 2\}$$

So we've built a Galois connection and transformers.We want to prove the analysis is sound. After proving that, we can check the assertions. We'll use the notation of $f = [[\_]]_D$ and $f^{\#} = [[\_]]_A$ (same as lecture).

$$\alpha(X) = \bigsqcup\{\beta(s) \mid s \in X\}$$
$$\gamma(a) = \{s \in S \mid \beta(s) \subseteq a\}$$

Let's simplify the terms (or abuse notations :)). β take the ïmage vectoränd returns a singleton of it mod 2. So the abstraction of a collection state, is to go over each individual state and take mod 2 of it. So, we'll write:

$$\alpha(X) = X\ \%\ 2$$

Next, we know that for $s \in S$, $\beta(s) = \{s\ \%\ 2\}$. so $\beta(s) \subseteq a$ iff $s\ \%\ 2 \in a$. Therefore:

$$\gamma(a) = \{s \in S \mid (s\ \%\ 2) \in a\}$$

So the abstraction function sends the values from $\mathbb{Z}$ to $\mathbb{Z}/2\mathbb{Z}$, and the concretization function lifts up the values.

i. Skip and assertions

we know that for skip and assertions: $f \equiv Id_D$ and $f^\# \equiv Id_A$. Then:

1. $\forall_{c,c'} : f(c) = c' \Rightarrow f^\#(\alpha(c)) = Id_A(\alpha(c)) = \alpha(c) = \alpha(Id_D(c)) = \alpha(c)) = \alpha(c')$
2. $\forall_{a,a'} : f^\#(a) = a' \Rightarrow f(\gamma(a)) = Id_D(\gamma(a)) = \gamma(a) = \gamma(Id_A(a)) = \gamma(f^\#(a)) = \gamma(a')$

So the skip transformer is sound.

ii. Simple assignment

We'll handle the case of $x := y$, $x := y \pm 1$ and $x := K$. We'll denote it as $x := g(y)$. Notice that g is commuting with mod 2: $g \circ \text{mod } 2 = \text{mod } 2 \circ g$
Let $c, c' \in D$ s.t. $f(c) = c'$. Then:

$$
\begin{aligned}
f^\#(\alpha(c)) = f^\#(c \,\%\, 2) = \\
= [[x := g(y)]]_A(\{\lambda v_i.s(v_i) \,\%\, 2 \mid s \in c\}) = \\
= \{(\lambda v_i.s(v_i) \,\%\, 2)[x \to g(s(y))] \mid s \in c\} = \\
= \{\lambda v_i. \begin{cases} s(v_i) \,\%\, 2 & \text{if } x \neq v_i \\ g(s(y) \,\%\, 2) & \text{if } x = v_i \end{cases} \mid s \in c\} = \\
= \{\lambda v_i. \begin{cases} s(v_i) \,\%\, 2 & \text{if } x \neq v_i \\ g(s(y)) \,\%\, 2 & \text{if } x = v_i \end{cases} \mid s \in c\}
\end{aligned}
$$

On the other hand:

$$
\begin{aligned}
\alpha(c') = \alpha(f(c)) = \alpha(\{s[x \to g(s(y))] \mid s \in c\}) = \\
= \alpha(\{\lambda v_i. \begin{cases} s(v_i) & \text{if } x \neq v_i \\ g(s(y)) & \text{if } x = v_i \end{cases} \mid s \in c\}) = \\
= \{\lambda v_i. \begin{cases} s(v_i) \,\%\, 2 & \text{if } x \neq v_i \\ g(s(y)) \,\%\, 2 & \text{if } x = v_i \end{cases} \mid s \in c\}
\end{aligned}
$$

Which is the same. So the first soundness requirement is proved.
Now let $a, a' \in A$ s.t. $f^\#(a) = a'$. Then:

$$
\begin{aligned}
f(\gamma(a)) = f(\{s \in S \mid (s \,\%\, 2) \in a\}) = \\
= [[x := g(y)]]_D(\{s \in S \mid (s \,\%\, 2) \in a\}) = \\
= \{s[x \to g(s(y))] \mid s \in S \wedge (s \,\%\, 2) \in a\} = \\
= \{\lambda v_i. \begin{cases} s(v_i) & \text{if } x \neq v_i \\ g(s(y)) & \text{if } x = v_i \end{cases} \mid s \in S \wedge (s \,\%\, 2) \in a\} =
\end{aligned}
$$

On the other hand:

$$
\begin{aligned}
\gamma(a') = \gamma(f^\#(a)) = \gamma([[x := g(y)]]_A(a)) = \\
= \gamma(\{\delta[x \to g(\delta(y))] \mid \delta \in a\}) = \\
= \{s \in S \mid (s \,\%\, 2) \in \{\delta[x \to g(\delta(y))] \mid \delta \in a\}\}
\end{aligned}
$$

Now, let's show every element of the first set is an element of the second set. we have $s \in S$ s.t. $(s \ \% \ 2) \in a$. Now we want to prove that $(s[x \to g(s(y))] \ \% \ 2) \in \{\delta[x \to g(\delta(y))] \mid \delta \in a\}$ We quickly notice that if $s \ \% \ 2 = \delta$ then $s[x \to g(s(y))] \ \% \ 2 = \delta[x \to g(\delta(y))]$. So it is true (as s had its δ in the condition, and now it has the same one).
So the second soundness requirement is proved.

iii. Unknown assignment

Let $c, c' \in D$ s.t. $f(c) = c'$. Then:

$$f^{\#}(\alpha(c)) = f^{\#}(c \ \% \ 2) =$$
$$= [[x :=?]]_A(c \ \% \ 2) =$$
$$= (c \ \% \ 2)[x \to 0] \cup (c \ \% \ 2)[x \to 1]$$

On the other hand:

$$\alpha(c') = \alpha(f(c)) = \alpha(\cup_{n \in \mathbb{N}}.c[x \to n]) =$$
$$\alpha(\cup_{n \in \mathbb{N}}.c[x \to 2n] \cup \cup_{n \in \mathbb{N}}.c[x \to 2n+1]) =$$
$$\alpha(\cup_{n \in \mathbb{N}}.c[x \to 2n]) \cup \alpha(\cup_{n \in \mathbb{N}}.c[x \to 2n+1])$$

Now, for each factor, the result of $s(x) \ \% \ 2$ is the same for every element (and otherwise the elements are the same). So we can ignore the union at all and get:

$$= (c[x \to 0] \ \% \ 2) \cup (c[x \to 0] \ \% \ 2) =$$
$$= (c \ \% \ 2)[x \to 0] \cup (c \ \% \ 2)[x \to 1]$$

So the first soundness requirement is proved.
Now let $a, a' \in A$ s.t. $f^{\#}(a) = a'$. Then:

$$f(\gamma(a)) = f(\{s \in S \mid (s \ \% \ 2) \in a\}) =$$
$$= [[x :=?]]_D(\{s \in S \mid (s \ \% \ 2) \in a\}) =$$
$$= \cup_{n \in \mathbb{N}}\{s[x \to n] \mid s \in S \wedge (s \ \% \ 2) \in a\}$$

On the other hand:

$$\gamma(a') = \gamma(f^{\#}(a)) = \gamma([[x :=?]]_A(a)) =$$
$$= \gamma(\{\delta[x \to 0] \mid \delta \in a\} \cup \{\delta[x \to 1] \mid \delta \in a\}) \geq$$
$$\geq \gamma(\{\delta[x \to 0] \mid \delta \in a\}) \cup \gamma(\{\delta[x \to 1] \mid \delta \in a\}) =$$
$$= \{s \in S \mid (s \ \% \ 2) \in \{\delta[x \to 0] \mid \delta \in a\}\} \cup \{s \in S \mid (s \ \% \ 2) \in \{\delta[x \to 1] \mid \delta \in a\}\}$$

Now, let's show every element of the first set is an element of the second set. we have $s \in S$ s.t. $(s \ \% \ 2) \in a$. Now we have $s[x \to n]$. Without loss of generality, assume n is even. We know that $(s \ \% \ 2) = \delta \in a$, therefore: $(s \ \% \ 2)[x \to n] = \delta[x \to 0] \in \{\delta[x \to 0] \mid \delta \in a\}$. So s is in the second set, as expected.
So the second soundness requirement is proved.

iv. Assumptions

We start with noticing that for $d \in S$, $[[E]]_D d \equiv [[E]]_A(d \ \% \ 2)$.
Let $c, c' \in D$ s.t. $f(c) = c'$. Then:

$$f^{\#}(\alpha(c)) = f^{\#}(c \ \% \ 2) = f^{\#}(\{s \ \% \ 2 \mid s \in c\}) =$$
$$= \{s \ \% \ 2 \mid \ s \in c \wedge [[E]](s \ \% \ 2 \text{) is true } \} =$$
$$\{s \ \% \ 2 \mid \ s \in c \wedge [[E]]s \text{ is true } \} =$$
$$= f(c) \ \% \ 2 = \alpha(f(c)) = \alpha(c')$$

So the first soundness requirement is proved. Now let $a, a' \in A$ s.t. $f^{\#}(a) = a'$. Then:

$$f(\gamma(a)) = f(\{s \in S \mid (s \ \% \ 2) \in a\}) =$$
$$= [[\text{assume E}]]_D(\{s \in S \mid (s \ \% \ 2) \in a\}) =$$
$$= \{s \in S \mid (s \ \% \ 2) \in a \wedge [[E]]s \text{ is true } \}$$

On the other hand,

$$\gamma(a') = \gamma(f^{\#}(a)) = \gamma(\{\delta \in a \mid [[E]]\delta \text{ is true } \}) =$$
$$= \{s \in S \mid (s \ \% \ 2) \in \{\delta \in a \mid [[E]]\delta \text{ is true } \}\} =$$
$$= \{s \in S \mid (s \ \% \ 2) \in a \wedge [[E]](s \ \% \ 2)\} =$$
$$\{s \in S \mid (s \ \% \ 2) \in a \wedge [[E]]s \text{ is true } \}$$

So the second soundness requirement is proved.

c) Assertions

So we proved that our analysis is sound - hooray! Each state on our abstraction is sound compared to the real state. The way our abstraction is defined, we can convert an assertion to a state on the lattice. Then, if the assertion state is greater than the state on the assertion vertex, we know that the assertion is true. If it is less than the state on the assertion vertex, then the assertion is false (but may be true on a specific running). If the assertion is non-comparable, then we can't answer it.
The conversion algorithm is simple:

1. Let $X = \emptyset$
2. For each clause of the assertion:
   (a) Let $E = \underset{v_i \in \text{Vars}}{\times} \text{Parity}$
   (b) For each term of the clause: WLG the term is ËVEN v_i". Set $E = \{e \in E \mid e(v_i) = 0\}$
   (c) $X := X \cup E$
3. Return X

Now $X \in A$, and we want to argue that X represents the assertion.
Let $e \in X$. we want to show that e represents a state expected by the assertion. e comes from some E the belongs to some clause C_i. so we know that for each term in C_i, $term(e)$ is true (because this is how we built E). So e is an accepted state of the assertion. On the other hand, if s is a state accepted by the

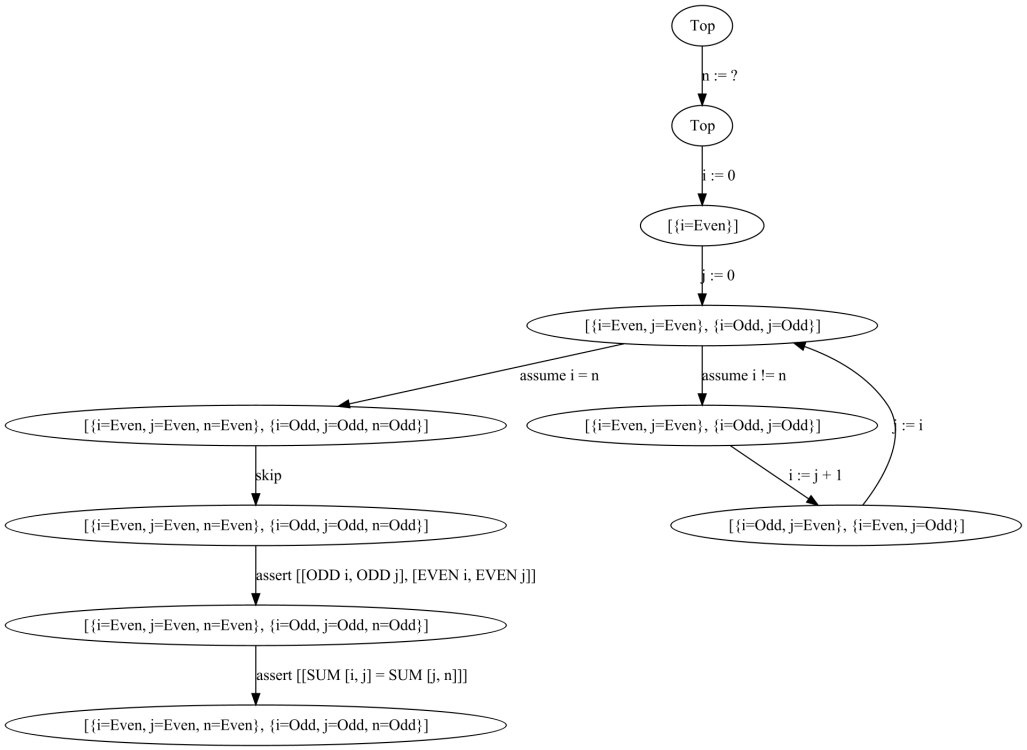
assertion, then it is accepted by one of the clauses of the assertion, c_i. Then, $s \in E_i$ because s was in the beginning, and was never removed, because the predict each time return true.
So X represents exactly the assertion.

d) Implementation

The analysis is implemented in *ParityAnalysis*.
Representing $\underset{v_i \in \text{Vars}}{\times}$ Parity as vectors requires a lot of memory, since we need to store up to 2^n vectors. Instead, I stored them as constraints. I stored $\varphi_1, ...\varphi_n$ for the set of vectors $\{(v_1, ..., v_n) | \forall_i \phi_i(v_i)\}$. Now, since usually we don't have constraints over all the variables, I stored is as an hash-map between variable and constraint.
Therefore, each set from the analysis is represented as a union of sets, each represents by its constraints. Beside the saving in space, it makes the implementation of the assignment very easy - go over each set, remove the old constraint and insert a new one. However, for $x := ?$ assignment, we need to split the constraint set to two (one for each possible parity).
In addition, the assumption case became complex, since we can't represent a connection between two variables as a constraint (as each predict is over a single variable). Check *updateStateFromAssumption* for the implementation.
While I could proved the analysis over the implementation's representation, I think proving it over the normal representation is much cleaner. Since both represents the same abstract domain, just with different representation, it is suffice.

e) The example program

Running the assertion of the example program yields the following result:



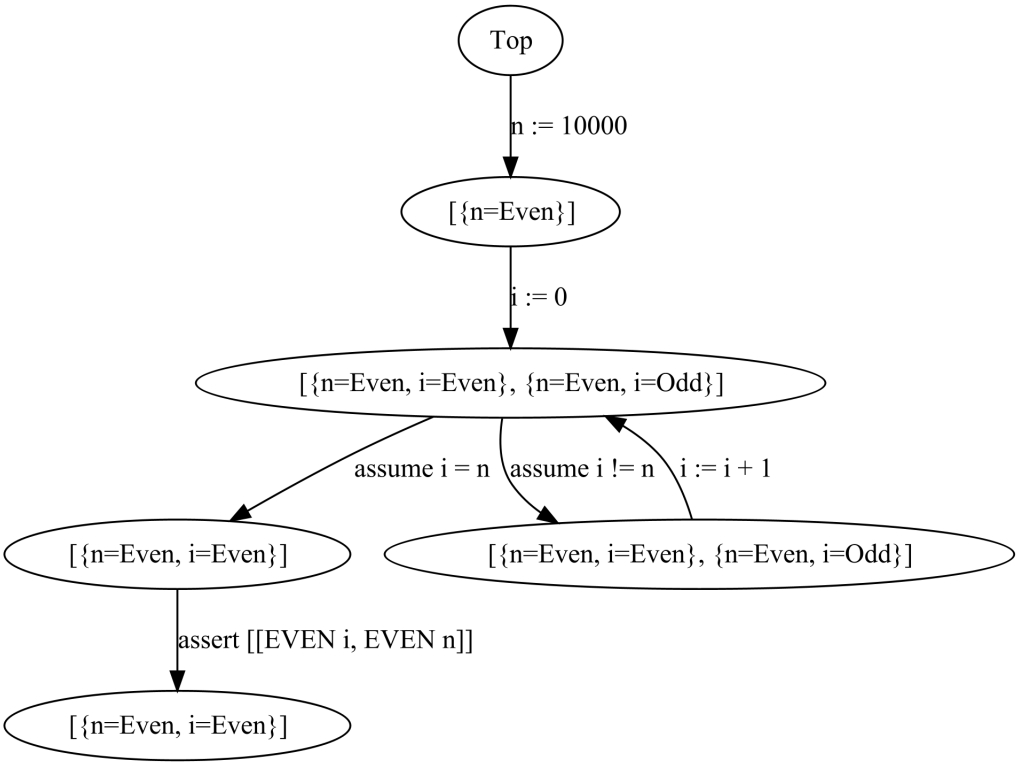
Note that the text of each vertex represent the constraints and not the sets. But, it makes it easier to see it passes the assertion :)

f) Test programs

The first test program checks that the analysis deals with a long for loop, hopefully without running all the rounds of the loop

```
i n

L0 n := 10000 L1
L1 i := 0 L2
L2 assume (i = n) L4
L2 assume (i != n) L3
L3 i := i + 1 L2
L4 assert (EVEN i EVEN n) L5
```

The analysis requires 14 rounds (where round is evaluating the analysis over a single node), and passes the assertions. The reason it finished so fast, is because the disjoint nature of the analysis led to the following fixed-point state:



The second test program checks that the analysis stops even if the program doesn't stop

```
i n

L0 i := 1 L1
L1 n := 2 L2
L2 assume n != i L3
L3 i := i + 1 L4
L4 n := n + 1 L2
L2 assume n = i L5
L5 assert (EVEN i ODD i) L6
```

The analysis requires 16 rounds, and the analysis shows the assertion has passed. The reason for that is that our analysis shows that $Parity(n) \neq Parity(i)$ at L2. Therefore, $n \neq i$ never and we reach bottom. We'll make a small modification to the program as our third test program:

```
i n

L0 i := ? L1
L1 n := 13 L2
L2 assume n != i L3
L3 i := i + 1 L4
```

```
L2 assume n = i L5
L5 assert (ODD i) L6
```

Here we don't know if the program will stop or not until the unknown value resolved. However, we know that if the program reaches L6, then $Parity(i) = Odd$. It took 14 rounds and it passed.
The forth test program is a binary counter. We want to see how the program handles all the possible states:

```
a b c d n

L0 n := 5 L1
L1 a := 1 L2
L2 b := 0 L3
L3 c := 1 L4
L4 d := 0 L5


L5 assume n != 0 L6
L6 assume a = 0 L100
L100 a := 1 L30

L6 assume a = 1 L8
L8 assume b = 0 L200
L200 b := 1 L201
L201 a := 0 L30

L8 assume b = 1 L10
L10 assume c = 0 L300
L300 c := 1 L301
L301 b := 0 L302
L302 a := 0 L30

L10 assume c = 1 L12
L12 assume d = 0 L400
L400 d := 1 L401
L401 c := 0 L402
L402 b := 0 L403
L403 a := 0 L30

L12 assume d = 1 L16
L30 n := n - 1 L5
L5 assume n = 0 L31
L31 assert (EVEN n EVEN a ODD b EVEN c ODD d) L32
```

It takes 127 rounds, and the assertion fails. Why? because the analysis only save the parity of n, it the assume n=0 edge translated to assume n is even, and so L31 consists of the binary counter for 6,8,10,12,14, and not just for 10.
So what if instead of adding int to binary counter, will add a binary counter to binary counter? here we add booleans to booleans, so we expect it to actually run the program and calculate the correct result.

```
a b c d w x y z ww xx yy zz

# init first counter
L1 a := 1 L2
L2 b := 0 L3
L3 c := 1 L4
L4 d := 0 L5

# init second counter
L5 w := 0 L7
L7 x := 1 L8
L8 y := 1 L9
L9 z := 0 L11

# init addition counter
L11 ww := 0 L12
L12 xx := 0 L13
L13 yy := 0 L14
L14 zz := 0 L15
```

```
20
21  # Check if added all
22  L15 assume w = ww L16
23  L16 assume x = xx L17
24  L17 assume y = yy L18
25  L18 assume z = zz L1000
26
27  L15 assume w != ww L19
28  L16 assume x != xx L19
29  L17 assume y != yy L19
30  L18 assume z != zz L19
31
32  # Increment a|b|c|d
33  L19 assume a = 0 L100
34  L100 a := 1 L30
35
36  L19 assume a = 1 L20
37  L20 assume b = 0 L200
38  L200 b := 1 L201
39  L201 a := 0 L30
40
41  L20 assume b = 1 L21
42  L21 assume c = 0 L300
43  L300 c := 1 L301
44  L301 b := 0 L302
45  L302 a := 0 L30
46
47  L21 assume c = 1 L22
48  L22 assume d = 0 L400
49  L400 d := 1 L401
50  L401 c := 0 L402
51  L402 b := 0 L403
52  L403 a := 0 L30
53
54  L22 assume d = 1 L1002
55
56  # increment ww|xx|yy|zz
57  L30 assume ww = 0 L500
58  L500 ww := 1 L60
59
60  L30 assume ww = 1 L31
61  L31 assume xx = 0 L600
62  L600 xx := 1 L601
63  L601 ww := 0 L60
64
65  L31 assume xx = 1 L32
66  L32 assume yy = 0 L700
67  L700 yy := 1 L701
68  L701 xx := 0 L702
69  L702 ww := 0 L60
70
71  L32 assume yy = 1 L33
72  L33 assume zz = 0 L800
73  L800 zz := 1 L801
74  L801 yy := 0 L802
75  L802 xx := 0 L803
76  L803 ww := 0 L60
77
78  L33 assume zz = 1 L1002
79
80  # increment both a|b|c|d and ww|xx|yy|zz, go to the loop condition
81  L60 skip L15
82
83  L1000 assert (ODD a ODD b EVEN c ODD d) L1001
```

And it did! It took 228 rounds, but it has successfully shown that if the program stops, the result of 5+6=11

3. Sum analysis

a) Domain

We are going to use $A = Mat^{\mathbb{Q}}_{nx(n+1)}$ as our domain. Each element represents an equation the variables must satisfy. One can think on each element as a linear affine space $V + v_0$, which is the solution to the affine linear system of equations represented by the matrix.
If we talk about matrices with less than n lines, just assume the rest of the lines are the zero equations $(0 = 0)$.
It is a complete lattice:

- $X \leq Y$ if $X \subseteq Y$
- The gcd is the intersection of the affine spaces (as matrices, just add all the equations of both matrices, and Gauss reduce)
- the lcm of two affine spaces is a minimal linear affine space that contains them both - given $V + v_0$ and $U + u_0$, we take $span\{V, U, \frac{1}{2}(v_0 - u_0)\} + \frac{1}{2}(v_0 + u_0)$. It's easy to prove it contains them both, and it is of the minimal dimension.

  Note: The minimal affine space is unique, as k linear independent points define a $k - 1$ plain uniquely. If they are not linear independent, just take the maximal set of them that is.
- gcd and lcm are both well defined, and return a greater/lesser element as required.
- top is the zero matrix, bottom is a matrix with no solution.

Note that the domain is not really a lattice, as there are multiple matrices represent the same linear space, and thus equal. That's why we limit our domain to reduced row echelon form matrices only (i.e. we convert to rref after every operation).
The variables of the matrix are $v_0, ..., v_n$ - the same variables of the program.
Now, we have to define the semantics on the abstract domain.

- [[skip]]$s = s$
- $[[i := i]]s = s$
- $[[i := j]]s = remove(s, i) \sqcap eq(i = j)$
- $[[i :=?]]s = remove(s, i)$
- $[[i := K]]s = remove(s, i) \sqcap eq(i = K)$
- $[[i := i \pm 1]]s =$ replace every reference of the variable i with $i \mp 1$. (multiply the matrix from left with corresponding basis transition matrix).
- $[[i := j \pm 1]]s = remove(s, i) \sqcap eq(i = j \pm 1)$
- [[assert ORC]]$s = s$
- [[assume E]]$s = s \sqcap eq(E)$

$eq(e)$ is defined as the affine space for the equation e. $remove(V + v_0, v_k) = span\{V, e_k\} + v$ - we allow the value of v_k to be anything. The operation will be either no-op or increase the dimension by one. One can also think of it as multiplying the matrix from left with a matrix the is 1 on the diagonal, except on position (k, k).

Notice that if $s \in S$ then $s[x \to K] \in remove(S, x)$ no matter what the value of K. In addition, if $s \subseteq eq(v_k = C)$ then $remove(s, v_k) \sqcap eq(v_k = C) = s$ (Think about it in geometric way).
Note that $[[S]]$ abstract transformer for each S is monotonic, as $remove(\_, i)$ is monotonic: if $V \leq U$ then $V \subseteq U$. Every element of $remove(\_, i)$ can be written as linear combination element of _ with e_i. So obviously $remove(V, i) \subseteq remove(U, i)$

b) Galois Connection

The next step is building the Galois connection. As shown in EX2 Q2, it is suffice to build $\beta : S \to A$ in order to build a Galois connection.

$$\beta(s) = \{0\} + v$$

Where $v = \{v_0, ..., v_n\}$. So β sends each assignment of variables to the linear system with the assignment as the only solution.

So we've built a Galois connection and transformers.We want to prove the analysis is sound. After proving that, we can check the assertions. We'll use the notation of $f = [[\_]]_D$ and $f^{\#} = [[\_]]_A$ (same as lecture).

$$\alpha(X) = \bigsqcup\{\beta(s) \mid s \in X\}$$
$$\gamma(a) = \{s \in S \mid \beta(s) \subseteq a\}$$

Let's simplify the terms. α takes a set of states, and return a minimal liner affine space that contains them all. γ is much simpler, as if we think of the abstract elements as $V + v_0$, and on elements of S as vector instead of function, then γ is simply the identity function, as $\beta(s) \subseteq a$ iff $s \in a$.

i. Skip and assertions

we know that for skip and assertions: $f \equiv Id_D$ and $f^{\#} \equiv Id_A$. Then:

1. $\forall_{c,c'} : f(c) = c' \Rightarrow f^{\#}(\alpha(c)) = Id_A(\alpha(c)) = \alpha(c) = \alpha(Id_D(c)) = \alpha(c)) = \alpha(c')$
2. $\forall_{a,a'} : f^{\#}(a) = a' \Rightarrow f(\gamma(a)) = Id_D(\gamma(a)) = \gamma(a) = \gamma(Id_A(a)) = \gamma(f^{\#}(a)) = \gamma(a')$

So the skip transformer is sound.

ii. Constant or simple assignment

$x := g(y)$. where $g(y)$ is either y, $y \pm 1$ or K.
Let $c, c' \in D$ s.t. $f(c) = c'$. Then:

$$f^{\#}(\alpha(c)) = remove(\alpha(c), x) \sqcap eq(x = g(y))$$
$$\alpha(c') = \alpha(f(c)) = \alpha(\{s[x \to g(y)] \mid s \in c\})$$

We start by showing the for all $s \in c$, $s[x \to g(y)] \in f^{\#}(\alpha(c))$. We know that:

1. $s \in \alpha(c)$.
2. $s[x \to g(y)] \in eq(x = g(y))$

3. $s[x \to g(y)] \in remove(\alpha(c), x)$ - as the value of x doesn't matter

Then, $s[x \to g(y)] \in remove(\alpha(c), x) \sqcap eq(x = g(y))$. Therefore, $a(c') \subseteq f^{\#}(\alpha(c))$, since $a(c')$ is the minimal affine subspace that contain all those $s[x \to g(y)]$, and $f^{\#}(\alpha(c))$ also contain them.
So the first soundness requirement is proved.
Now let $a, a' \in A$ s.t. $f^{\#}(a) = a'$. Then:

$$f(\gamma(a)) = f(Id(a)) = f(a) = \{s[x \to g(y)] \mid s \in a\}$$
$$\gamma(a') = Id(a') = f^{\#}(a) = remove(a, x) \sqcap eq(x = g(y))$$

Now for each $s \in a$, $s[x \to g(y)] \in remove(a, x), Eq(x \to g(y)$, therefore $s[x \to g(y)] \in remove(a, x) \sqcap Eq(x \to g(y)$ so $f(\gamma(a)) \subseteq \gamma(a')$ as required.
So the second soundness requirement is proved.

iii. Increment or Decrement

$x := x \pm 1$
We'll denote the basis transition matrix as B. Then, $s[x \to x \pm 1] = Bs$.
Let $c, c' \in D$ s.t. $f(c) = c'$. Then:

$$f^{\#}(\alpha(c)) = B \cdot a(c)$$
$$\alpha(c') = \alpha(f(c)) = \alpha(\{s[x \to x \pm 1] \mid s \in c\}) =$$
$$= \alpha(\{Bs \mid s \in c\}) = \alpha(Bc)$$

It easy to see that $\alpha(Bc) \subseteq B\alpha(c)$, as $Bc \subseteq B\alpha(c)$.
So the first soundness requirement is proved.
Now let $a, a' \in A$ s.t. $f^{\#}(a) = a'$. Then:

$$f(\gamma(a)) = f(Id(a)) = f(a) = \{s[x \to x \pm 1] \mid s \in a\} = \{Bs \mid s \in a\} = Ba$$
$$\gamma(a') = Id(a') = f^{\#}(a) = Ba$$

They are equal, So the second soundness requirement is proved.

iv. Unknown assignment

$x :=?$
Let $c, c' \in D$ s.t. $f(c) = c'$. Then:

$$f^{\#}(\alpha(c)) = remove(\alpha(c), x)$$
$$\alpha(c') = \alpha(f(c)) = \alpha(\cup_{n \in \mathbb{N}}.c[x \to n])$$

We know that for each $s \in c, n \in \mathbb{N}$, $s[x \to n] \in f^{\#}(\alpha(c))$ (as we showed earlier), then $\cup_{n \in \mathbb{N}}.c[x \to n] \in f^{\#}(\alpha(c))$, and so $a(c') \subseteq f^{\#}(\alpha(c))$.
So the first soundness requirement is proved.

Now let $a, a' \in A$ s.t. $f^{\#}(a) = a'$. Then:

$$f(\gamma(a)) = f(Id(a)) = f(a) = \cup_{n \in \mathbb{N}}.a[x \to n]$$
$$\gamma(a') = Id(a') = f^{\#}(a) = remove(a, x)$$

We know that for each $s \in a, n \in \mathbb{N}$, $s[x \to n] \in remove(a, x)$, therefore $f(\gamma(a)) \subseteq \gamma(a')$ as required.
So the second soundness requirement is proved.

v. Assumptions

Let $c, c' \in D$ s.t. $f(c) = c'$. Then:

$$f^{\#}(\alpha(c)) = s \sqcap eq(E)$$
$$\alpha(c') = \alpha(f(c)) = \alpha(\{s \in c \mid E(s) \text{ is True}\})$$

- if $s \in c$ then $s \in \alpha(c)$
- if $E(s)$ then $s \in eq(E)$

Therefore $\{s \in c \mid E(s) \text{ is True}\} \in s \sqcap eq(E)$ as required.
So the first soundness requirement is proved.
Now let $a, a' \in A$ s.t. $f^{\#}(a) = a'$. Then:

$$f(\gamma(a)) = f(Id(a)) = f(a) = \{s \in a \mid E(s) \text{ is True}\}$$
$$\gamma(a') = Id(a') = f^{\#}(a) = a \sqcap Eq(E)$$

As before, $s \in a$ and $E(s)$ is true so $s \in Eq(E)$, therefore $s \in a \sqcap Eq(E)$ and $f(\gamma(a)) \subseteq \gamma(a')$ as required.
So the second soundness requirement is proved.

c) Assertions

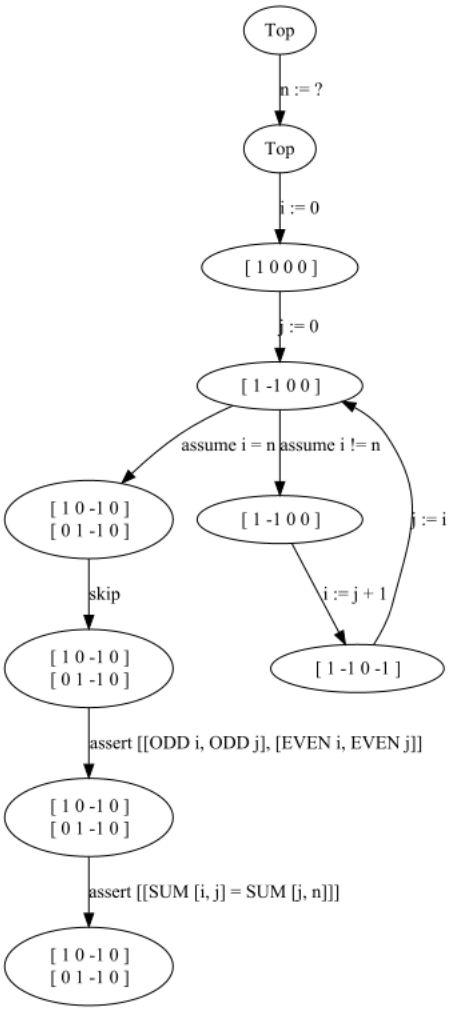
So we proved that our analysis is sound - hooray! Each state on our abstraction is sound compared to the real state. We convert each clause in the assertion to an equation, and convert it to a linear space. Then, the clause is satisfied if the state is a subspace of the converted clause.

d) Implementation

The analysis is implemented in *SumAnalysis*. I represented each abstract element as list of linear equations. The implementation of the remove operation is a little bit optimized, as it tries not to build a basis to the solution space if unnecessary. Also, the assumption implementation is optimized for common cases.

e) The example program

Running the assertion of the example program yields the following result:



where the variables are (i, j, n, 1). As can be observed, the assumption passes successfully, as the linear space is of $i = j = n$.

f) Test programs

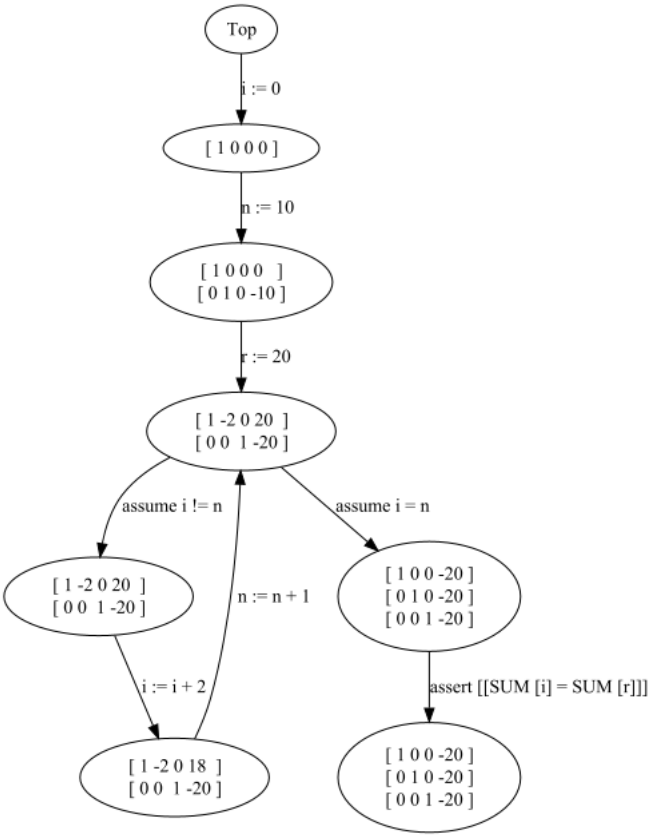
The first test program is the same as the parity's one. It checks that the analysis deals with a long for loop, hopefully without running all the rounds of the loop.

```
i n

L0 n := 10000 L1
L1 i := 0 L2
L2 assume (i = n) L4
L2 assume (i != n) L3
L3 i := i + 1 L2
L4 assert (SUM i = SUM j) L5
```

It takes 14 rounds, and it passes successfully.
The second test program shows that our analysis can understand a linear connection between two variables:

```
i n r

L0 i := 0 L1
L1 n := 10 L2
L2 r := 20 L3
L3 assume (i != n) L4
L4 i := i + 2 L5
L5 n := n + 1 L3
L3 assume (i = n) L6
L6 assert (SUM i = SUM r) L7
```

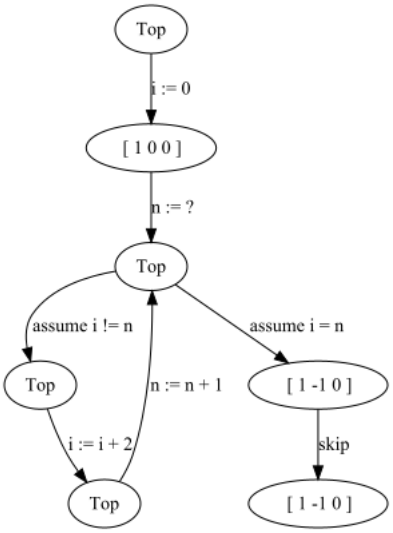
It takes 19 rounds and the assertion passed. The result fixed point state is:



As can be seen, it finds a linear connection as the loop invariant - $i - 2n = -20$! Then, when $i = n$ it can calculate the solution and shows that they must have the value 20.
The third test program deals with unknown:

```
i n

L0 i := 0 L1
L1 n := ? L2
L2 assume (i != n) L3
L3 i := i + 2 L4
L4 n := n + 1 L2
L2 assume (i = n) L5
L5 skip L6
```

It finished after running 18 rounds, and the result is:



We have Top as our state. But it makes total sense, as all we know is that i is even, and n increases - no linear connection of i and n.
The forth test program shows that the analysis is strong enough to prove addition (as expected):

```
i j n

L0 n := ? L1
L1 i := n L2
L2 j := n L3
L3 assume (i != 0) L4
L4 i := i - 1 L5
L5 j := j + 1 L3
L3 assume (i = 0) L6
L6 assert (SUM j = SUM n n) L7
```

It takes rounds to complete and passes successfully. The loop invariant is the key here:

$$i + j = 2n$$

In the end, where $i = 0$, we know that $j = 2n$ as expected.
The last test programs tries to take it to the next level - multiplication by constant.

```
i j k n

L0 n := ? L1
L1 i := 1 L2
L2 j := 5 L3

# outer loop
L3 assume (j != 0) L4
L4 k := n L5

# inner loop
L5 assume (k != 0) L6
L6 i := i + 1 L7
L7 k := k - 1 L5
L5 assume (k = 0) L8

L8 j := j - 1 L3


L3 assume (j = 0) L100
L100 assert (SUM i = SUM n n n n n) L101
```

The analysis run for 39 rounds and the assertion failed (we can't prove it, not that it cannot be) - which is expected, as the relation between the variables is not linear.

4. Combined Analysis

We combined the analysis in 3 ways:

1. The most simple way - the Cartesian's multiplication of the analyses. It's equivalent to running both analysis standalone, and just take special care to run each assertion checker on the correct type of assertions.

2. Using the Sum analysis also for parity - the idea is to look at the vector space mod 2. Then, iterate over the solution space and check if it matches the assertions.

3. Similar to 2, but with homogeneous equations instead of affine. Here, $i = K$ (for $K \neq 0$) and $i := j \pm 1$ are equivalent to $i :=?$ as we cannot store this information.

We'll discuss the cost and precision of the different analyses:

a) Cost

1. The Cartesian analysis cost as running both analyses independently.
2. The advance sum analysis cost the same as running the sum analysis. However, checking the assertions cost more as we have to iterate over the solution space, which might be as big as 2^k where k is the number of variables in the assertion.
3. The homogeneous advance sum analysis is faster than the sum analysis and requires less memory, as operations on homogeneous equation system are much faster. The assertions checking stage should take similar time to (2).

b) Precision

1. This is as precise as the analyses them-self. We gained no precision boost from running them together.
2. This is **sometimes** less precise from (1). The reason is that the original parity analysis was disjoint and discrete, while this analysis is linearly continuous. Let's say there are 2 variables in the program x, y. Let's say there is a vertex in which there are 3 possible options for values: $(0,1), (1,0), (1,2)$. In the original parity analysis, we had no problem to show that both variables has different parity. However, when putting this 2 options in a linear space, we get $\mathbb{Q}^2$, which are literally all the options for x and y. However, it is **sometimes** more precise then (1). For example:

```
i n r

L0 i := 0 L1
L1 n := 20 L2
L2 r := 30 L3
L3 assume (i != n) L4
L4 i := i + 3 L5
L5 n := n + 1 L3
L3 assume (i = n) L6
L6 assert (SUM i = SUM r) L7
L7 assert (EVEN n EVEN i) L8
```

   On the one hand, the parity analysis fail to prove that n and i are both even at the end, since the loop invariant only says that their parity is equal. On the other hand, the linear analysis shows the $i = n = 30$, so it has no problem showing they are both even.

3. As it uses only homogeneous equations, it is less precise from (2). The question arise is whether it became less precise than (1). The answer is not - we are going to exploit the fact we can save whether $x = 0$. Take, for example, the following program:

```
i

L1 i := 0 L2
L2 assume (i != 0) L3
L3 assume(i != 2) L4
L4 i := 1 L20
L3 assume (i = 2) L5
L5 i := 0 L20
L2 assume (i = 0) L20
L20 assert (EVEN i) L21
```

   The homogeneous analysis knows that $i = 0$, so it knows the *assume i != 0* branch will not be taken, while the parity analysis thinks it is possible, since all it knows is the parity of i. So even when taking the power to store all the values of i, saving the value of 0 is strong enough

5. Implementation

I wrote the analyses in Kotlin programming language, using three external libraries:

1. Kotlin-Matrix - Copied for its matrix class. I rewrote it to used fraction and not complex numbers, and even found a bug! I've also added more matrix algorithms that were needed for the project.

2. Better parse - A parser combinators library for kotlin. The library allows you to simply define your tokens:

```
    private val ws by regexToken("""\s+""", ignore = true)
    private val comment by regexToken("""#.+""", ignore = true)
    private val skip by literalToken("skip")
    private val assume by literalToken("assume")
    private val assert by literalToken("assert")
    private val trueValue by literalToken("true")
    private val falseValue by literalToken("false")
    private val lpar by literalToken("(")
    private val rpar by literalToken(")")
    private val plus by literalToken("+")
    private val minus by literalToken("-")
    private val equal by literalToken("=")
    private val notEqual by literalToken("!=")
    private val assign by literalToken(":=")
    private val unknown by literalToken("?")
    private val sum by literalToken("SUM")
    private val even by literalToken("EVEN")
    private val odd by literalToken("ODD")
    private val constValue by regexToken("""\d+""")
    private val vertex by regexToken("""L(\d+)""")
    private val variable by regexToken("""[a-z]+""")
```

and then the combinators:

```
private val constValueParser by constValue use { BigInteger(text) }
    private val variableParser by variable use { text }
    private val vertexParser by vertex use { BigInteger(text.substring(1)) }
    private val opParser by (plus asJust ASTOperation.Plus) or (minus asJust ASTOperation
        .Minus)
    private val equalParser by (equal asJust true) or (notEqual asJust false)
    private val sumParser by -sum * oneOrMore(variableParser)

    private val valueParser: Parser<ASTValue> by
    ((variableParser * opParser * constValueParser).map { (var1, op, var2) ->
        ASTValue.VariableOpConstValue(
            var1,
            if (op == ASTOperation.Plus) var2 else -var2
        )
    }) or
            variableParser.map(ASTValue::VariableValue) or
            constValueParser.map(ASTValue::ConstValue) or
            unknown.asJust(ASTValue.UnknownValue)

    private val assumptionParser: Parser<ASTAssumption> by
    trueValue.asJust(ASTAssumption.TrueAssumption) or falseValue.asJust(ASTAssumption.
        FalseAssumption) or
            ((variableParser * equalParser * valueParser) map { (var1, op, value) ->
                ASTAssumption.VariableAssumption(
                    var1,
                    op,
                    value
                )
            })

    // ...


    override val rootParser: Parser<ASTProgram> by (zeroOrMore(variableParser) *
        zeroOrMore(edgeParser)).map { (variables, edges) ->
```

```
        ASTProgram(
            variables.toSortedSet(),
            edges
        )
    }
```

And you've got a parser that return your AST.

3. Graphviz-Java - used to print the state of the analysis (and for the images in this paper).

After having an AST, we semant the program, and convert it to a graph structure. Next, there are two importants interface: *Lattice* and *Analysis*:

```
interface Lattice<T> {
    val bottom: T
    val top: T
    fun gcd(item1: T, item2: T): T
    fun lcm(item1: T, item2: T): T
    fun compare(item1: T, item2: T): CompareResult
}

interface Analysis<T> {
    val lattice: Lattice<T>

    fun transfer(statement: ASTStatement, state: T): T

    fun join(stateA: T, stateB: T) = lattice.lcm(stateA, stateB)

    fun checkAssertions(assertion: List<List<ASTAssertion>>, state: T): Boolean?
}
```

And finally, there is the *AnalysisRunner*, which receives an analysis and a program, and run the analysis over the program, printing the final state and the results of the assertions.
To run the program:

```
Usage: java -jar analysis.jar MODE PATH_TO_PROGRAM [GRAPHVIZ_MODE]
Mode:
   0 - Parity analysis
   1 - Sum analysis
   2 - Cartesian analysis
   3 - Sum analysis with parity support
   4 - Homogenous sum analysis with parity support
Graphviz mode:
   0 - Disabled
   1 - Print state on each step
   2 - Print only the final step
```