Abstract Interpretation (an introduction)

SCSSE Summer School 2017

Part 1

David Pichardie

ENS Rennes, France

these slides are available at
`http://www.irisa.fr/celtique/pichardie/teaching/ecnu-2017-part1.pdf`

Static program analysis

The goals of static program analysis

- to prove properties about the run-time behaviour of a program
- in a fully automatic way
- without actually executing this program

Applications

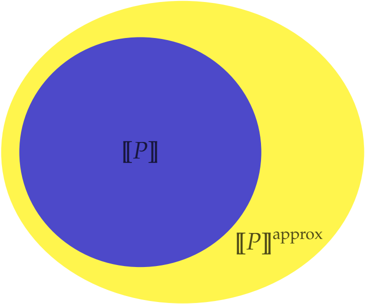
- code optimisation
- error detection (array out of bound access, null pointers)
- proof support (invariant extraction)

Abstract Interpretation

A theory which unifies a large variety of static analysis

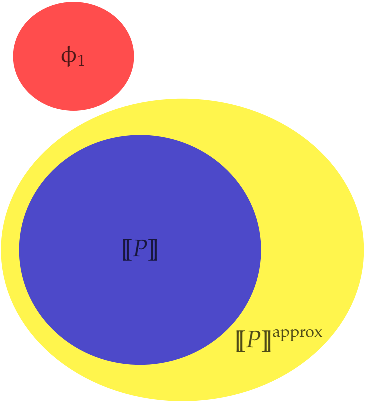
- formalises the approximated analyse of programs
- allows to compare relative precision of analyses
- facilitates the conception of sophisticated analyses
- discovered by Patrick Cousot and Radhia Cousot in 1977

Static analysis computes approximations



$[\![P]\!]$:	concrete semantics (e.g. set of reachable states)	(not computable)
$[\![P]\!]^{\text{approx}}$:	analyser result (here over-approximation)	(computable)

Static analysis computes approximations

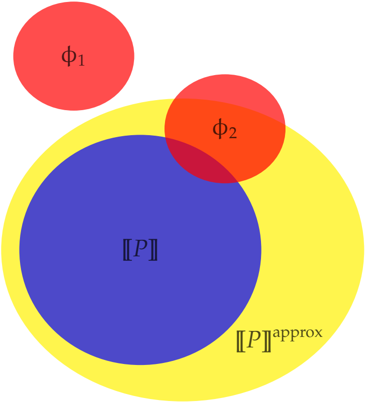


- P is safe w.r.t. ϕ_1 and the analyser proves it

$$[\![P]\!] \cap \phi_1 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_1 = \emptyset$$

$[\![P]\!]$:	concrete semantics (e.g. set of reachable states)	(not computable)
ϕ_1 :	erroneous/dangerous set of states	(computable)
$[\![P]\!]^{\text{approx}}$:	analyser result (here over-approximation)	(computable)

Static analysis computes approximations



- P is safe w.r.t. ϕ_1 and the analyser proves it

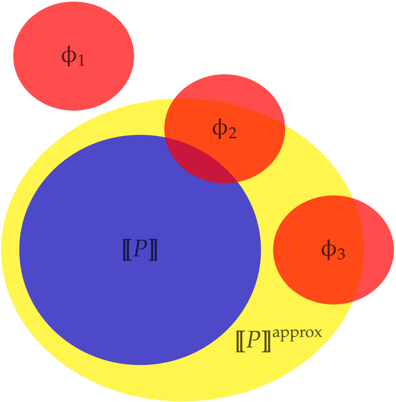
$$[\![P]\!] \cap \phi_1 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_1 = \emptyset$$

- P is unsafe w.r.t. ϕ_2 and the analyser warns about it

$$[\![P]\!] \cap \phi_2 \neq \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_2 \neq \emptyset$$

$[\![P]\!]$:	concrete semantics (e.g. set of reachable states)	(not computable)
ϕ_1, ϕ_2 :	erroneous/dangerous set of states	(computable)
$[\![P]\!]^{\text{approx}}$:	analyser result (here over-approximation)	(computable)

Static analysis computes approximations



- P is safe w.r.t. ϕ_1 and the analyser proves it

$$[\![P]\!] \cap \phi_1 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_1 = \emptyset$$

- P is unsafe w.r.t. ϕ_2 and the analyser warns about it

$$[\![P]\!] \cap \phi_2 \neq \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_2 \neq \emptyset$$

- but P is safe w.r.t. ϕ_3 and the analyser can't prove it (this is called a *false alarm*)

$$[\![P]\!] \cap \phi_3 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_3 \neq \emptyset$$

$[\![P]\!]$:	concrete semantics (e.g. set of reachable states)	(not computable)
ϕ_1, ϕ_2, ϕ_3 :	erroneous/dangerous set of states	(computable)
$[\![P]\!]^{\text{approx}}$:	analyser result (here over-approximation)	(computable)

Example

Concrete semantics says :

- at point 2, $\mathtt{x} = 100$

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:   res = res * x;
5:   x = x - 1;
6: };
7: y = 1 / res;
```

Example

Concrete semantics says :

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:   res = res * x;
5:   x = x - 1;
6: };
7: y = 1 / res;
```

- at point 2, $\mathtt{x} = 100$
- at point 3 (before entering the loop), $\mathtt{x} = 100$ and $\mathtt{res} = 1$

Example

Concrete semantics says :

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:  res = res * x;
5:  x = x - 1;
6: };
7: y = 1 / res;
```

- at point 2, $\mathtt{x} = 100$
- at point 3 (before entering the loop), $\mathtt{x} = 100$ and $\mathtt{res} = 1$
- at point 5, $\mathtt{x} > 0$ and $\mathtt{res} = 100 \times 99 \times \cdots \times \mathtt{x}$

Example

Concrete semantics says :

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:   res = res * x;
5:   x = x - 1;
6: };
7: y = 1 / res;
```

- at point 2, $\mathtt{x} = 100$
- at point 3 (before entering the loop), $\mathtt{x} = 100$ and $\mathtt{res} = 1$
- at point 5, $\mathtt{x} > 0$ and $\mathtt{res} = 100 \times 99 \times \cdots \times \mathtt{x}$
- at point 6, $\mathtt{x} \geqslant 0$ and $\mathtt{res} = 100 \times 99 \times \cdots \times (\mathtt{x} + 1)$

Example

Concrete semantics says :

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:   res = res * x;
5:   x = x - 1;
6: };
7: y = 1 / res;
```

- at point 2, $\texttt{x} = 100$
- at point 3 (before entering the loop), $\texttt{x} = 100$ and $\texttt{res} = 1$
- at point 5, $\texttt{x} > 0$ and $\texttt{res} = 100 \times 99 \times \cdots \times \texttt{x}$
- at point 6, $\texttt{x} \geqslant 0$ and $\texttt{res} = 100 \times 99 \times \cdots \times (\texttt{x} + 1)$
- at point 7, $\texttt{x} = 0$ and $\texttt{res} = 100!$
  Hence, we can prove there is no division by zero at point 7.

Example

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:  res = res * x;
5:  x = x - 1;
6: };
7: y = 1 / res;
```

A **correct** static analysis could says :

- at point 2, $\mathtt{x} = 100$ (yes/no ?), $\mathtt{x} > 0$ (yes/no ?), $\mathtt{x} < 10$ (yes/no ?)

Example

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:   res = res * x;
5:   x = x - 1;
6: };
7: y = 1 / res;
```

A **correct** static analysis could says :

- at point 2, $\mathtt{x} = 100$ (yes/no ?), $\mathtt{x} > 0$ (yes/no ?), $\mathtt{x} < 10$ (yes/no ?)

Example

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:  res = res * x;
5:  x = x - 1;
6: };
7: y = 1 / res;
```

A **correct** static analysis could says :

- at point 2, $\mathtt{x} = 100$ (**yes**/no ?), $\mathtt{x} > 0$ (yes/no ?), $\mathtt{x} < 10$ (yes/no ?)

Example

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:  res = res * x;
5:  x = x - 1;
6: };
7: y = 1 / res;
```

A **correct** static analysis could says :

- at point 2, $\mathtt{x} = 100$ (**yes**/no ?), $\mathtt{x} > 0$ (**yes**/no ?), $\mathtt{x} < 10$ (yes/no ?)

Example

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:  res = res * x;
5:  x = x - 1;
6: };
7: y = 1 / res;
```

A **correct** static analysis could says :

- at point 2, $\mathtt{x} = 100$ (**yes**/no ?), $\mathtt{x} > 0$ (**yes**/no ?), $\mathtt{x} < 10$ (yes/**no** ?)

Example

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:  res = res * x;
5:  x = x - 1;
6: };
7: y = 1 / res;
```

A **correct** static analysis could says :

- at point 2, $\mathtt{x} = 100$ (**yes**/no ?), $\mathtt{x} > 0$ (**yes**/no ?), $\mathtt{x} < 10$ (yes/**no** ?)
- at point 3 (before entering the loop), $\mathtt{x} = 100 \times \mathtt{res}$ (yes/no ?)

Example

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:   res = res * x;
5:   x = x - 1;
6: };
7: y = 1 / res;
```

A **correct** static analysis could says :

- at point 2, $\texttt{x} = 100$ (**yes**/no ?), $\texttt{x} > 0$ (**yes**/no ?), $\texttt{x} < 10$ (yes/**no** ?)
- at point 3 (before entering the loop), $\texttt{x} = 100 \times \texttt{res}$ (**yes**/no ?)

Example

A **correct** static analysis could says :

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:   res = res * x;
5:   x = x - 1;
6: };
7: y = 1 / res;
```

- at point 2, $\texttt{x} = 100$ (**yes**/no ?), $\texttt{x} > 0$ (**yes**/no ?), $\texttt{x} < 10$ (yes/**no** ?)
- at point 3 (before entering the loop), $\texttt{x} = 100 \times \texttt{res}$ (**yes**/no ?)
- at point 7, $\texttt{x} > 0$ (yes/no), $\texttt{res} > 0$ (yes/no ?), $\texttt{res} > x$ (yes/no ?)

Example

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:  res = res * x;
5:  x = x - 1;
6: };
7: y = 1 / res;
```

A **correct** static analysis could says :

- at point 2, $\mathtt{x} = 100$ (**yes**/no ?), $\mathtt{x} > 0$ (**yes**/no ?), $\mathtt{x} < 10$ (yes/**no** ?)
- at point 3 (before entering the loop), $\mathtt{x} = 100 \times \mathtt{res}$ (**yes**/no ?)
- at point 7, $\mathtt{x} > 0$ (yes/**no**), $\mathtt{res} > 0$ (yes/no ?), $\mathtt{res} > x$ (yes/no ?)

Example

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:   res = res * x;
5:   x = x - 1;
6: };
7: y = 1 / res;
```

A **correct** static analysis could says :

- at point 2, $\mathtt{x} = 100$ (**yes**/no ?), $\mathtt{x} > 0$ (**yes**/no ?), $\mathtt{x} < 10$ (yes/**no** ?)
- at point 3 (before entering the loop), $\mathtt{x} = 100 \times \mathtt{res}$ (**yes**/no ?)
- at point 7, $\mathtt{x} > 0$ (yes/**no**), $\mathtt{res} > 0$ (**yes**/no ?), $\mathtt{res} > x$ (yes/no ?)

Example

A **correct** static analysis could says :

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:   res = res * x;
5:   x = x - 1;
6: };
7: y = 1 / res;
```

- at point 2, $\mathtt{x} = 100$ (**yes**/no ?), $\mathtt{x} > 0$ (**yes**/no ?), $\mathtt{x} < 10$ (yes/**no** ?)
- at point 3 (before entering the loop), $\mathtt{x} = 100 \times \mathtt{res}$ (**yes**/no ?)
- at point 7, $\mathtt{x} > 0$ (yes/**no**), $\mathtt{res} > 0$ (**yes**/no ?), $\mathtt{res} > x$ (**yes**/no ?)

Example

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:  res = res * x;
5:  x = x - 1;
6: };
7: y = 1 / res;
```

A **correct** static analysis could says :

- at point 2, $\mathtt{x} = 100$ (**yes**/no ?), $\mathtt{x} > 0$ (**yes**/no ?), $\mathtt{x} < 10$ (yes/**no** ?)
- at point 3 (before entering the loop), $\mathtt{x} = 100 \times \mathtt{res}$ (**yes**/no ?)
- at point 7, $\mathtt{x} > 0$ (yes/**no**), $\mathtt{res} > 0$ (**yes**/no ?), $\mathtt{res} > x$ (**yes**/no ?)
  But only the property $\mathtt{res} > 0$ can prove the absence of division by zero.

Example

A **correct** static analysis could says :

```
1: x = 100;
2: res = 1;
3: while (x>0) {
4:   res = res * x;
5:   x = x - 1;
6: };
7: y = 1 / res;
```

- at point 2, $\mathtt{x} = 100$ (**yes**/no ?), $\mathtt{x} > 0$ (**yes**/no ?), $\mathtt{x} < 10$ (yes/**no** ?)
- at point 3 (before entering the loop), $\mathtt{x} = 100 \times \mathtt{res}$ (**yes**/no ?)
- at point 7, $\mathtt{x} > 0$ (yes/**no**), $\mathtt{res} > 0$ (**yes**/no ?), $\mathtt{res} > x$ (**yes**/no ?)
  But only the property $\mathtt{res} > 0$ can prove the absence of division by zero.

  $\mathtt{res} > x$ will raise a false alarm.

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {                    }
while (x<6) {
  if (?) {
      {                    }
    y = y+2;
      {                    }
  };
      {                                }
  x = x+1;
      {                                }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0)          }
while (x<6) {
  if (?) {
      {               }
    y = y+2;
      {               }
  };
      {                          }
  x = x+1;
      {                          }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0)              }
while (x<6) {
  if (?) {
      {(0,0)              }
    y = y+2;
      {                   }
  };
      {                              }
  x = x+1;
      {                              }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0)                }
while (x<6) {
  if (?) {
      {(0,0)                }
    y = y+2;
      {(0,2)                }
  };
      {                               }
  x = x+1;
      {                               }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0)              }
while (x<6) {
  if (?) {
      {(0,0)              }
    y = y+2;
      {(0,2)              }
  };
      {(0,0),(0,2)                  }
  x = x+1;
      {                              }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0)              }
while (x<6) {
  if (?) {
      {(0,0)              }
    y = y+2;
      {(0,2)              }
  };
      {(0,0),(0,2)                }
  x = x+1;
      {(1,0),(1,2)                }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0),(1,0),(1,2)    }
while (x<6) {
  if (?) {
      {(0,0)                }
    y = y+2;
      {(0,2)                }
  };
      {(0,0),(0,2)          }
  x = x+1;
      {(1,0),(1,2)          }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0),(1,0),(1,2)    }
while (x<6) {
  if (?) {
      {(0,0),(1,0),(1,2)    }
    y = y+2;
      {(0,2)                }
  };
      {(0,0),(0,2)                        }
  x = x+1;
      {(1,0),(1,2)                        }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
        {(0,0), (1,0), (1,2)     }
while (x<6) {
  if (?) {
        {(0,0), (1,0), (1,2)     }
    y = y+2;
        {(0,2), (1,2), (1,4)     }
  };
        {(0,0), (0,2)                        }
  x = x+1;
        {(1,0), (1,2)                        }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0),(1,0),(1,2)    }
while (x<6) {
  if (?) {
      {(0,0),(1,0),(1,2)    }
    y = y+2;
      {(0,2),(1,2),(1,4)    }
  };
      {(0,0),(0,2),(1,0),(1,2),(1,4)    }
  x = x+1;
      {(1,0),(1,2)                      }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0),(1,0),(1,2)    }
while (x<6) {
  if (?) {
      {(0,0),(1,0),(1,2)    }
    y = y+2;
      {(0,2),(1,2),(1,4)    }
  };
      {(0,0),(0,2),(1,0),(1,2),(1,4)    }
  x = x+1;
      {(1,0),(1,2),(2,0),(2,2),(2,4)    }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
        {(0,0),(1,0),(1,2),...}
while (x<6) {
  if (?) {
        {(0,0),(1,0),(1,2),...}
    y = y+2;
        {(0,2),(1,2),(1,4),...}
  };
        {(0,0),(0,2),(1,0),(1,2),(1,4),...}
  x = x+1;
        {(1,0),(1,2),(2,0),(2,2),(2,4),...}
}
        {(6,0),(6,2),(6,4),(6,6),...}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$\begin{array}{lcl} P & ::= & x\,C\,0 \wedge y\,C\,0 \\ C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant \end{array}$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
```

$\mathtt{x} = 0 \wedge \mathtt{y} = 0$

```
while (x<6) {
  if (?) {

    y = y+2;

  };

  x = x+1;

}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$\begin{array}{rcl} P & ::= & x\,C\,0 \wedge y\,C\,0 \\ C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant \end{array}$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x = 0 ∧ y = 0
while (x<6) {
  if (?) {
        x = 0 ∧ y = 0
    y = y+2;

  };

  x = x+1;

}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$P \;::=\; x\,C\,0 \wedge y\,C\,0$$
$$C \;::=\; < \mid \leqslant \mid = \mid > \mid \geqslant$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x = 0 ∧ y = 0
while (x<6) {
  if (?) {
        x = 0 ∧ y = 0
    y = y+2;
        x = 0 ∧ y > 0   over-approximation!
  };

  x = x+1;

}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$P ::= x \, C \, 0 \wedge y \, C \, 0$$
$$C ::= < \mid \leqslant \mid = \mid > \mid \geqslant$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x = 0 ∧ y = 0
while (x<6) {
  if (?) {
        x = 0 ∧ y = 0
    y = y+2;
        x = 0 ∧ y > 0
  };
        x = 0 ∧ y ⩾ 0
  x = x+1;

}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x \, C \, 0 \wedge y \, C \, 0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x = 0 ∧ y = 0
while (x<6) {
  if (?) {
        x = 0 ∧ y = 0
    y = y+2;
        x = 0 ∧ y > 0
  };
        x = 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0 over-approximation!
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x\,C\,0 \wedge y\,C\,0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x = 0 ∧ y = 0
    y = y+2;
        x = 0 ∧ y > 0
  };
        x = 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x\,C\,0 \wedge y\,C\,0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x ⩾ 0 ∧ y ⩾ 0
    y = y+2;
        x = 0 ∧ y > 0
  };
        x = 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$P \;::=\; x\,C\,0 \wedge y\,C\,0$$
$$C \;::=\; < \mid \leqslant \mid = \mid > \mid \geqslant$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x ⩾ 0 ∧ y ⩾ 0
    y = y+2;
        x ⩾ 0 ∧ y > 0
  };
        x = 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x\, C\, 0 \wedge y\, C\, 0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x ⩾ 0 ∧ y ⩾ 0
    y = y+2;
        x ⩾ 0 ∧ y > 0
  };
        x ⩾ 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{aligned}
P &::= x\, C\, 0 \wedge y\, C\, 0 \\
C &::= < \mid \leqslant \mid = \mid > \mid \geqslant
\end{aligned}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x ⩾ 0 ∧ y ⩾ 0
    y = y+2;
        x ⩾ 0 ∧ y > 0
  };
        x ⩾ 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x\,C\,0 \wedge y\,C\,0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x ⩾ 0 ∧ y ⩾ 0
    y = y+2;
        x ⩾ 0 ∧ y > 0
  };
        x ⩾ 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
        x ⩾ 0 ∧ y ⩾ 0
```

An other example : the interval analysis

For each point k and each numeric variable x, we infer an interval in which x must belong to.

Example : insertion sort, array access verification

```
assert(T.length=100); i=1;
                                        {i ∈ [1,100]}
while (i<T.length) {
                                        {i ∈ [1,99]}
    p = T[i]; j = i-1;
                                        {i ∈ [1,99], j ∈ [−1,98]}
    while (0<=j and T[j]>p) {
                                        {i ∈ [1,99], j ∈ [0,98]}
        T[j]=T[j+1]; j = j-1;
                                        {i ∈ [1,99], j ∈ [−1,97]}
    };
                                        {i ∈ [1,99], j ∈ [−1,98]}
    T[j+1]=p; i = i+1;
                                        {i ∈ [2,100], j = [−1,98]}
};
                                        {i = 100}
```

An other example : the polyhedral analysis

For each point k and we infer invariant linear equality and inequality relationships among variables.

Example : insertion sort, array access verification

```
assert(T.length>=1); i=1;

while i<T.length {

    p = T[i]; j = i-1;

    while 0<=j and T[j]>p {

        T[j]=T[j+1]; j = j-1;

    };

    T[j+1]=p; i = i+1;

};
```

$\{1 \leqslant i \leqslant T.length\}$

$\{1 \leqslant i \leqslant T.length - 1\}$

$\{1 \leqslant i \leqslant T.length - 1 \ \wedge \ -1 \leqslant j \leqslant i - 1\}$

$\{1 \leqslant i \leqslant T.length - 1 \ \wedge \ 0 \leqslant j \leqslant i - 1\}$

$\{1 \leqslant i \leqslant T.length - 1 \ \wedge \ -1 \leqslant j \leqslant i - 2\}$

$\{1 \leqslant i \leqslant T.length - 1 \ \wedge \ -1 \leqslant j \leqslant i - 1\}$

$\{2 \leqslant i \leqslant T.length + 1 \ \wedge \ -1 \leqslant j \leqslant i - 2\}$

$\{i = T.length\}$

This lecture

Outline

Outline

Outline

Outline

Outline

Outline

While syntax

$$
\begin{array}{rcll}
Exp & ::= & n & n \in \mathbb{Z} \\
& | & ? & \\
& | & x & x \in \mathbb{V} \\
& | & Exp\ o\ Exp & o \in \{+,-,\times\} \\
\\
Test & ::= & Exp\ c\ Exp & c \in \{=,\neq,<,\leqslant\} \\
& | & Test\ \textbf{and}\ Test & \\
& | & Test\ \textbf{or}\ Test & \\
\\
Stm & ::= & {}^{l}[x\ :=\ Exp] & l \in \mathbb{P} \\
& | & {}^{l}[\texttt{skip}] & \\
& | & \texttt{if}\ {}^{l}[Test]\ \{\ Stm\ \}\ \{\ Stm\ \} & \\
& | & \texttt{while}\ {}^{l}[Test]\ \{\ Stm\ \} & \\
& | & Stm\ ;\ Stm & \\
Prog & ::= & [Stm]^{end} & end \in \mathbb{P}
\end{array}
$$

$\mathbb{P}$: set of program points $\mathbb{V}$: set of program variables

While syntax : example

$[^0[x := ?];$
$\texttt{if}\ ^1[x < 0]\ \{$
$\quad \texttt{while}\ ^2[x < 0]\ \{$
$\quad\quad ^3[x := x + 1];$
$\quad \};$
$\quad ^4[y := x];$
$\}\ \texttt{else}\ \{$
$\quad ^5[y := 0];$
$\};]^6$

Syntax : Ocaml code

```
type var = string

type binop =
  | Add | Sub | Mult

type expr =
  | Const of int
  | Unknown
  | Var of var
  | Binop of binop * expr * expr

type comp = Eq | Neq | Le | Lt

type test =
  | Comp of comp * expr * expr
  | And of test * test
  | Or of test * test
```

```
type label = int

type stmt =
  | Assign of label * var * expr
  | Skip of label
  | If of label * test * stmt * stmt
  | While of label * test * stmt
  | Seq of stmt * stmt

type program = stmt * label
```

While semantics

Semantic domains

$$\begin{array}{rcl} Env & \stackrel{\text{def}}{=} & \mathbb{V} \to \mathbb{Z} \\ State & \stackrel{\text{def}}{=} & \mathbb{P} \times Env \end{array}$$

Semantics of expressions

$$\mathcal{A}[\![e]\!]\,\rho \in \mathcal{P}(\mathbb{Z}), \quad e \in Exp,\ \rho \in Env$$

$$\begin{array}{lcl} \mathcal{A}[\![n]\!]\,\rho & = & \{\, n \,\} \\ \mathcal{A}[\![?]\!]\,\rho & = & \mathbb{Z} \\ \mathcal{A}[\![x]\!]\,\rho & = & \{\, \rho(x) \,\},\ x \in \mathbb{V} \\ \mathcal{A}[\![e_1\ o\ e_2]\!]\,\rho & = & \{\, v_1 o\, v_2 \mid v_1 \in \mathcal{A}[\![e_1]\!]\,\rho,\ v_2 \in \mathcal{A}[\![e_2]\!]\,\rho \,\} \\ & & \qquad\qquad o \in \{+, -, \times\} \end{array}$$

Remark : $\mathcal{A}[\![\cdot]\!]\,\rho$ is non-deterministic because of the expression ?.

Semantics of tests

$$\mathcal{B}[\![t]\!]\,\rho \in \mathcal{P}(\mathbb{B}), \quad t \in \mathit{Test},\ \rho \in \mathit{Env} \quad \mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$$

$$\frac{v_1 \in \mathcal{A}[\![e_1]\!]\,\rho \quad v_2 \in \mathcal{A}[\![e_2]\!]\,\rho \quad v_1\ \bar{c}\ v_2}{\mathbf{tt} \in \mathcal{B}[\![e_1\ c\ e_2]\!]\,\rho}$$

$$\frac{v_1 \in \mathcal{A}[\![e_1]\!]\,\rho \quad v_2 \in \mathcal{A}[\![e_2]\!]\,\rho \quad \neg\,(v_1\ \bar{c}\ v_2)}{\mathbf{ff} \in \mathcal{B}[\![e_1\ c\ e_2]\!]\,\rho}$$

$$\frac{b_1 \in \mathcal{B}[\![t_1]\!]\,\rho \quad b_2 \in \mathcal{B}[\![t_2]\!]\,\rho}{b_1 \wedge_{\mathbb{B}} b_2 \in \mathcal{B}[\![t_1\ \mathtt{and}\ t_2]\!]\,\rho}$$

$$\frac{b_1 \in \mathcal{B}[\![t_1]\!]\,\rho \quad b_2 \in \mathcal{B}[\![t_2]\!]\,\rho}{b_1 \vee_{\mathbb{B}} b_2 \in \mathcal{B}[\![t_1\ \mathtt{or}\ t_2]\!]\,\rho}$$

Structural Operational Semantics

Small-step semantics

$$\frac{v \in \mathcal{A}[\![a]\!]\rho}{({}^{l}[x := a], \rho) \Rightarrow \rho[x \mapsto v]} \qquad \frac{}{({}^{l}[\mathbf{nop}], \rho) \Rightarrow \rho}$$

$$\frac{(S_1, \rho) \Rightarrow \rho'}{(S_1 \,;\, S_2, \rho) \Rightarrow (S_2, \rho')} \qquad \frac{(S_1, \rho) \Rightarrow (S_1', \rho')}{(S_1 \,;\, S_2, \rho) \Rightarrow (S_1' \,;\, S_2, \rho')}$$

$$\frac{\mathbf{tt} \in \mathcal{B}[\![b]\!]\rho}{(\mathtt{if}\ {}^{l}[b]\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \rho) \Rightarrow (S_1, \rho)}$$

$$\frac{\mathbf{ff} \in \mathcal{B}[\![b]\!]\rho}{(\mathtt{if}\ {}^{l}[b]\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \rho) \Rightarrow (S_2, \rho)}$$

$$\frac{\mathbf{tt} \in \mathcal{B}[\![b]\!]\rho}{(\mathtt{while}\ {}^{l}[b]\ \mathtt{do}\ S, \rho) \Rightarrow (S \,;\, \mathtt{while}\ {}^{l}[b]\ \mathtt{do}\ S, \rho)}$$

$$\frac{\mathbf{ff} \in \mathcal{B}[\![b]\!]\rho}{(\mathtt{while}\ {}^{l}[b]\ \mathtt{do}\ S, \rho) \Rightarrow \rho}$$

Program point manipulation

We define the entry point of a statement :

$$
\begin{aligned}
\mathit{entry}\left({}^{l}[x := e]\right) &= \mathit{entry}\left({}^{l}[\texttt{skip}]\right) = l \\
\mathit{entry}\left(\texttt{if}\ {}^{l}[t]\ \{\ S_1\ \}\ \{\ S_2\ \}\right) &= \mathit{entry}\left(\texttt{while}\ {}^{l}[t]\ \{\ S\ \}\right) = l \\
\mathit{entry}\left(S_1;\ S_2\right) &= \mathit{entry}(S_1)
\end{aligned}
$$

We define the set of points of a statement :

$$
\begin{aligned}
\mathit{labels}\left({}^{l}[x := e]\right) &= \mathit{labels}\left({}^{l}[\texttt{skip}]\right) = \{l\} \\
\mathit{labels}\left(\texttt{if}\ {}^{l}[t]\ \{\ S_1\ \}\ \{\ S_2\ \}\right) &= \{l\} \cup \mathit{labels}(S_1) \cup \mathit{labels}(S_2) \\
\mathit{labels}\left(\texttt{while}\ {}^{l}[t]\ \{\ S\ \}\right) &= \{l\} \cup \mathit{labels}(S) \\
\mathit{labels}\left(S_1;\ S_2\right) &= \mathit{labels}(S_1) \cup \mathit{labels}(S_2)
\end{aligned}
$$

SOS-Reachable states

Reachable states :

$$\left\llbracket [P]^{\text{end}} \right\rrbracket_{\text{SOS}} = \left\{ (k, \rho) \;\middle|\; \begin{array}{l} \exists \rho_0 \in \mathit{Env}, \\ \quad \exists S \in \mathit{Stm},\ (P, \rho_0) \Rightarrow^* (S, \rho) \text{ and } k = \text{entry}(S) \\ \quad \text{or}\ \ (P, \rho_0) \Rightarrow^* \rho \text{ and } k = \text{end} \end{array} \right\}$$

Outline

A flowchart representation of program

The standard program model in static analysis : the *control flow graph*.

The graph model used here :

- the nodes are program point $k \in \mathbb{P}$,
- the edges are labeled with *basic instructions*

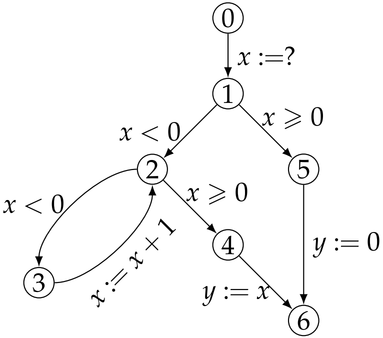
$$\begin{array}{rlll} \mathit{Instr} ::= & x := \mathit{Exp} & \textit{assignment} \\ | & \texttt{assume}\ \mathit{Test} & \textit{execution continues only if} \\ & & \textit{the test succeds} \end{array}$$

- formally, a cfg is a triplet $(k_{\mathrm{init}}, S, k_{\mathrm{end}})$ with
  - $k_{\mathrm{init}} \in \mathbb{P}$: the entry point,
  - $k_{\mathrm{end}} \in \mathbb{P}$: the exit point,
  - $S \subseteq \mathbb{P} \times \mathit{Instr} \times \mathbb{P}$ the set of edges.

Remark : data-flow analyses are generally based on other versions of control flow graphs (instructions are put in nodes).

While syntax : example

$[^0[x :=?];$
$\texttt{if}\ ^1[x < 0]\ \{$
$\quad \texttt{while}\ ^2[x < 0]\ \{$
$\quad\quad ^3[x := x + 1];$
$\quad \};$
$\quad ^4[y := x];$
$\}\ \texttt{else}\ \{$
$\quad ^5[y := 0];$
$\};]^6$



assume is left implicit

Control flow graph generation (1/2)

$cfg_l(S)$ computes the edges of the control flow graph of S using l as final label.

$$
\begin{array}{rcl}
cfg_l & \in & Stm \to \mathcal{P}(\mathbb{P} \times Instr \times \mathbb{P}), \quad l \in \mathbb{P} \\
cfg_{l'}\left({}^{l}[x := e]\right) & = & \{(l, x := e, l')\} \\
cfg_{l'}\left({}^{l}[\mathtt{skip}]\right) & = & \{(l, \mathtt{assume}\ T, l')\} \qquad \text{with } T \equiv 0 = 0 \\
cfg_{l'}\left(\mathtt{if}\ {}^{l}[t]\ \{\ S_1\ \}\ \{\ S_2\ \}\right) & = & \{(l, \mathtt{assume}\ t, entry(S_1))\} \cup \\
& & \{(l, \mathtt{assume}\ neg(t), entry(S_2))\} \cup cfg_{l'}(S_1) \cup cfg_{l'}(S_2) \\
cfg_{l'}\left(\mathtt{while}\ {}^{l}[t]\ \{\ S\ \}\right) & = & \\
 & & \\
cfg_{l'}(S_1;\ S_2) & = & \\
 & & \\
cfg & \in & Prog \to \mathbb{P} \times \mathcal{P}(\mathbb{P} \times Instr \times \mathbb{P}) \times \mathbb{P} \\
cfg([P]^{end}) & = & (entry(P), cfg_{end}(P), end)
\end{array}
$$

Control flow graph generation (1/2)

$cfg_l(S)$ computes the edges of the control flow graph of S using l as final label.

$$
\begin{array}{rcl}
cfg_l & \in & Stm \to \mathcal{P}(\mathbb{P} \times Instr \times \mathbb{P}), \quad l \in \mathbb{P} \\
cfg_{l'}\left({}^{l}[x := e]\right) & = & \{(l, x := e, l')\} \\
cfg_{l'}\left({}^{l}[\mathtt{skip}]\right) & = & \{(l, \mathtt{assume}\ T, l')\} \qquad \text{with } T \equiv 0 = 0 \\
cfg_{l'}\left(\mathtt{if}\ {}^{l}[t]\ \{\ S_1\ \}\ \{\ S_2\ \}\right) & = & \{(l, \mathtt{assume}\ t, entry(S_1))\} \cup \\
 & & \{(l, \mathtt{assume}\ neg(t), entry(S_2))\} \cup cfg_{l'}(S_1) \cup cfg_{l'}(S_2) \\
cfg_{l'}\left(\mathtt{while}\ {}^{l}[t]\ \{\ S\ \}\right) & = & \{(l, \mathtt{assume}\ t, entry(S))\} \cup \\
 & & cfg_l(S) \cup \{(l, \mathtt{assume}\ neg(t), l')\} \\
cfg_{l'}\left(S_1;\ S_2\right) & = &
\end{array}
$$

$$
\begin{array}{rcl}
cfg & \in & Prog \to \mathbb{P} \times \mathcal{P}(\mathbb{P} \times Instr \times \mathbb{P}) \times \mathbb{P} \\
cfg([P]^{end}) & = & (entry(P), cfg_{end}(P), end)
\end{array}
$$

Control flow graph generation (1/2)

$cfg_l(S)$ computes the edges of the control flow graph of S using l as final label.

$$
\begin{array}{rcl}
cfg_l & \in & Stm \to \mathcal{P}(\mathbb{P} \times Instr \times \mathbb{P}), \quad l \in \mathbb{P} \\
cfg_{l'}\left({}^{l}[x := e]\right) & = & \{(l, x := e, l')\} \\
cfg_{l'}\left({}^{l}[\texttt{skip}]\right) & = & \{(l, \texttt{assume}\ T, l')\} \qquad \text{with } T \equiv 0 = 0 \\
cfg_{l'}\left(\texttt{if}\ {}^{l}[t]\ \{\ S_1\ \}\ \{\ S_2\ \}\right) & = & \{(l, \texttt{assume}\ t, entry(S_1))\} \cup \\
& & \{(l, \texttt{assume}\ neg(t), entry(S_2))\} \cup cfg_{l'}(S_1) \cup cfg_{l'}(S_2) \\
cfg_{l'}\left(\texttt{while}\ {}^{l}[t]\ \{\ S\ \}\right) & = & \{(l, \texttt{assume}\ t, entry(S))\} \cup \\
& & cfg_l(S) \cup \{(l, \texttt{assume}\ neg(t), l')\} \\
cfg_{l'}\left(S_1;\ S_2\right) & = & cfg_{entry(S_2)}(S_1) \cup cfg_{l'}(S_2)
\end{array}
$$

$$
\begin{array}{rcl}
cfg & \in & Prog \to \mathbb{P} \times \mathcal{P}(\mathbb{P} \times Instr \times \mathbb{P}) \times \mathbb{P} \\
cfg([P]^{end}) & = & (entry(P), cfg_{end}(P), end)
\end{array}
$$

Control flow graph generation (2/2)

Test negation :

$$
\begin{array}{rcl}
neg(e_1 = e_2) & = & e_1 \neq e_2 \\
neg(e_1 \neq e_2) & = & e_1 = e_2 \\
neg(e_1 < e_2) & = & e_2 \leqslant e_1 \\
neg(e_1 \leqslant e_2) & = & e_2 < e_1 \\
neg(t_1 \ \textbf{and}\ t_2) & = & neg(t_1)\ \textbf{or}\ neg(t_1) \\
neg(t_1 \ \textbf{or}\ t_2) & = & neg(t_1)\ \textbf{and}\ neg(t_1)
\end{array}
$$

Small-step semantics of cfg

We first define the semantics of instructions : $\xrightarrow{i} \subseteq Env \times Env$

$$\frac{v \in \mathcal{A}[\![a]\!]\rho}{\rho \xrightarrow{x\ :=\ a} \rho[x \mapsto v]} \qquad \frac{\mathbf{tt} \in \mathcal{B}[\![t]\!]\rho}{\rho \xrightarrow{\texttt{assume}\ t} \rho}$$

Then, a small-step relation $\rightarrow_{cfg} \subseteq State \times State$ for a $cfg = (k_{\text{init}}, S, k_{\text{end}})$

$$\frac{(k_1, i, k_2) \in S \qquad \rho_1 \xrightarrow{i} \rho_2}{(k_1, \rho_1) \rightarrow_{cfg} (k_2, \rho_2)}$$

Reachable states for control flow graphs

$$[\![(k_{\text{init}}, S, k_{\text{end}})]\!]_{\text{CFG}} = \{\ (k, \rho) \mid \exists \rho_0 \in Env,\ (k_{\text{init}}, \rho_0) \rightarrow^*_{(k_{\text{init}}, S, k_{\text{end}})} (k, \rho)\ \}$$

Correctness of *cfg*

Lemma

For any $l \in \mathbb{P}$*,* $S, S' \in Stm$ *and* $s, s' \in State$

$$(S, s) \Rightarrow s' \text{ iff } \exists i \in Instr, s \xrightarrow{i} s' \text{ and } (entry(S), i, l) \in cfg_l(S)$$

$$(S, s) \Rightarrow (S', s') \text{ iff } \exists i \in Instr, s \xrightarrow{i} s' \text{ and } (entry(S), i, entry(S')) \in cfg_l(S)$$

Theorem

For all program p,

$$[\![cfg(p)]\!]_{CFG} = [\![p]\!]_{SOS}$$

From now on, we write $[\![p]\!]$ instead of $[\![cfg(p)]\!]_{\mathrm{CFG}}$ and we confound $cfg(p)$ and p.

Proof of the lemma (1/4)

Auxiliary lemmas :

1. $b \in \mathcal{B}[\![t]\!]\rho$ implies $\neg b \in \mathcal{B}[\![neg(t)]\!]\rho$.
   Proof : by induction on t.
2. $(S, \rho) \Rightarrow (S', \rho')$ implies $cfg_l(S') \subseteq cfg_l(S)$.
   Proof : by induction on $(S, \rho) \Rightarrow (S', \rho')$.

Proof of the lemma (2/4)

We prove that $(S, \rho) \Rightarrow \rho'$ implies $\exists i \in \mathit{Instr}, \rho \xrightarrow{i} \rho'$ and $(\mathit{entry}(S), i, l) \in \mathit{cfg}_l(S)$ by case analysis on $(S, \rho) \Rightarrow \rho'$.

- $({}^{l'}[x := a], \rho) \Rightarrow \rho[x \mapsto v]$ with $v \in \mathcal{A}[\![a]\!]\rho$ and $\mathit{cfg}_l(S) = \{(l', x := a, l)\}$.
  Hence $\rho \xrightarrow{x := a} \rho[x \mapsto v]$ and since $\mathit{entry}(S) = l'$ we have $(\mathit{entry}(S), i, l) \in \mathit{cfg}_l(S)$.
- $({}^{l'}[\mathbf{nop}], \rho) \Rightarrow \rho$ and therefore $\mathit{cfg}_l(S) = \{(l', \mathtt{assume}\ T, l)\}$.
  Hence, $\rho \xrightarrow{\mathtt{assume}\ T} \rho$ and, since $\mathit{entry}(S) = l'$, we have $(\mathit{entry}(S), i, l) \in \mathit{cfg}_l(S)$.
- $(\mathtt{while}\ {}^{l'}[t]\ \mathtt{do}\ S, \rho) \Rightarrow \rho$ with $\mathbf{ff} \in \mathcal{B}[\![t]\!]\rho$ and $\mathit{cfg}_l\left(\mathtt{while}\ {}^{l'}[t]\ \{\ S\ \}\right) = \{(l', \mathtt{assume}\ t, \mathit{entry}(S))\} \cup \mathit{cfg}_{l'}(S) \cup \{(l', \mathtt{assume}\ \mathit{neg}(t), l)\}$.
  Hence $\rho \xrightarrow{\mathtt{assume}\ \mathit{neg}(t)} \rho$ since $\mathbf{tt} \in \mathcal{B}[\![\mathit{neg}(t)]\!]\rho$.
  Furthermore, $\mathit{entry}(S) = l'$ so we have $(\mathit{entry}(S), \mathtt{assume}\ \mathit{neg}(t), l) \in \mathit{cfg}_l(S)$.

Proof of the lemma (3/4)

We prove that $(S, \rho) \Rightarrow (S', \rho')$ implies $\exists i \in \mathit{Instr}, \rho \xrightarrow{i} \rho'$ and $(\mathit{entry}(S), i, \mathit{entry}(S')) \in \mathit{cfg}_l(S)$ by induction on $(S, \rho) \Rightarrow (S', \rho')$.

- $(S_1\,; S_2, \rho) \Rightarrow (S_2, \rho')$ because $(S_1, \rho) \Rightarrow \rho'$ and $\mathit{cfg}_l\,(S_1;\ S_2) = \mathit{cfg}_{\mathit{entry}(S_2)}(S_1) \cup \mathit{cfg}_l(S_2)$. Using the previous result with $\mathit{entry}(S_2)$, we know there exists $i \in \mathit{Instr}$ such that $\rho \xrightarrow{i} \rho'$ and $(\mathit{entry}(S_1), i, \mathit{entry}(S_2)) \in \mathit{cfg}_{\mathit{entry}(S_2)}(S_1) \subseteq \mathit{cfg}_l\,(S_1;\ S_2)$ and $\mathit{entry}(S_1\,; S_2) = \mathit{entry}(S_1)$.
- $(S_1\,; S_2, \rho) \Rightarrow (S'_1\,; S_2, \rho')$ because $(S_1, \rho) \Rightarrow (S'_1, \rho')$ and $\mathit{cfg}_l\,(S_1;\ S_2) = \mathit{cfg}_{\mathit{entry}(S_2)}(S_1) \cup \mathit{cfg}_l(S_2)$. Using induction hypothesis with $\mathit{entry}(S_2)$ instead of l, we know there exists $i \in \mathit{Instr}$ such that $\rho \xrightarrow{i} \rho'$ and $(\mathit{entry}(S_1), i, \mathit{entry}(S'_1)) \in \mathit{cfg}_{\mathit{entry}(S_2)}(S_1) \subseteq \mathit{cfg}_l\,(S_1;\ S_2)$ and $\mathit{entry}(S_1 :\ S_2) = \mathit{entry}(S_1)$. We easily conclude since $\mathit{entry}(S_1\,; S_2) = \mathit{entry}(S_1)$ and $\mathit{entry}(S'_1\,; S_2) = \mathit{entry}(S'_1)$ and as a consequence $(\mathit{entry}(S_1\,; S_2), i, \mathit{entry}(S'_1\,; S_2)) \in \mathit{cfg}_l\,(S_1;\ S_2)$.

Proof of the lemma (4/4)

- $(\texttt{if}^{\,l'}[t]\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \rho) \Rightarrow (S_1, \rho)$ with $\mathbf{tt} \in \mathcal{B}[\![b]\!]\rho$ and $(l', \texttt{assume}\ t, entry(S_1)) \in cfg_l(S)$.
  Hence $\rho \xrightarrow{\texttt{assume}\ t} \rho$ and $(entry(S), \texttt{assume}\ t, entry(S_1)) \in cfg_l(S)$ since $entry(S) = l'$.
- $\ldots$

Exercise : Prove the other direction : $\exists i, \ldots$ implies ...

Outline

Collecting Semantics

We will consider a collecting semantics that give us the set of reachable states $[\![p]\!]_k^{\text{col}}$ at each program points k.

$$\forall k \in \mathbb{P},\ [\![p]\!]_k^{\text{col}} = \{\ \rho \mid (k, \rho) \in [\![p]\!]\ \}$$

Theorem

$[\![p]\!]^{col}$ *may be characterized as the least fixpoint of the following equation system.*

$$\forall k \in labels(p),\ X_k = X_k^{init} \cup \bigcup_{(k',i,k)\in p} [\![i]\!]\,(X_{k'})$$

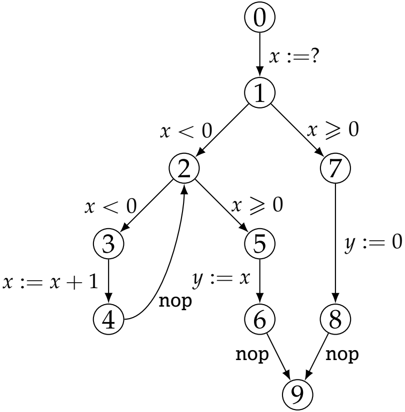
with $X_k^{\text{init}} = \begin{cases} Env & \text{if } k = k_{\text{init}} \\ \emptyset & \text{otherwise} \end{cases}$

and

$$\forall i \in Instr,\ \forall X \subseteq Env,\ [\![i]\!]\,(X) = \left\{\ \rho_2 \mid \exists \rho_1 \in X,\ \rho_1 \xrightarrow{i} \rho_2\ \right\} = \text{post}\left[\xrightarrow{i}\right](X)$$

Example

For the following program, $[\![P]\!]^{\mathrm{col}}$ is the least solution of the following equation system :

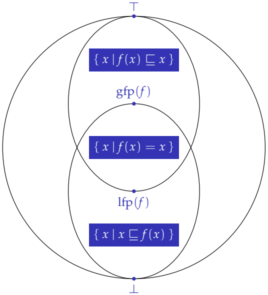


$$
\begin{array}{rcl}
X_0 & = & Env \\
X_1 & = & [\![x :=?]\!]\,(X_0) \\
X_2 & = & [\![x < 0]\!]\,(X_1) \cup X_4 \\
X_3 & = & [\![x < 0]\!]\,(X_2) \\
X_4 & = & [\![x := x + 1]\!]\,(X_3) \\
X_5 & = & [\![x \geqslant 0]\!]\,(X_2) \\
X_6 & = & [\![y := x]\!]\,(X_5) \\
X_7 & = & [\![x \geqslant 0]\!]\,(X_1) \\
X_8 & = & [\![y := 0]\!]\,(X_7) \\
X_9 & = & X_6 \cup X_8
\end{array}
$$

Fixpoint Lattice Theory

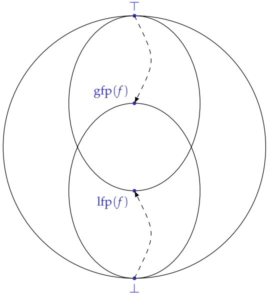
Theorem (Knaster-Tarski)

In a complete lattice $(A, \sqsubseteq, \bigsqcup)$, for all monotone functions $f \in A \to A$, the least fixpoint $\mathrm{lfp}(f)$ of f exists and is $\bigsqcap\{x \in A \mid f(x) \sqsubseteq x\}$.



Theorem (Kleene fixpoint theorem)

In a complete lattice $(A, \sqsubseteq, \bigsqcup)$, for all continuous function $f \in A \to A$, the least fixpoint $\mathrm{lfp}(f)$ of f is equal to $\bigsqcup\{ f^n(\perp) \mid n \in \mathbb{N} \}$.



Theorem

Let $(A, \sqsubseteq)$ a poset that verifies the ascending chain condition and f a monotone function. The sequence $\perp, f(\perp), \ldots, f^n(\perp), \ldots$ eventually stabilises. Its limit is the least fixpoint of f.

Collecting semantics and exact analysis

The $(X_k)_{i=1..N}$ are hence specified as the least solution of a fixpoint equation system

$$X_k = F_k(X_1, X_2, \ldots, X_N) \ \ , \ \ k \in labels(p)$$

or, equivalently $\vec{X} = \vec{F}(\vec{X})$.

Exact analysis :

- Thanks to Knaster-Tarski, the least solution exists (complete lattice, F_k are monotone functions),
- Kleen fixpoint theorem (F_k are continuous functions) says it is the limit of

$$X_k^0 = \emptyset \ \ , \ \ X_k^{n+1} = F_k(X_1^n, X_2^n, \ldots, X_N^n)$$

Uncomputable problem :

- Representing the X_k may be hard (infinite sets)
- The limit may not be reachable in a finite number of steps

Approximate analysis

Exact analysis :
Least solution of $X = F(X)$ in the complete lattice $(\mathcal{P}(Env)^N, \subseteq, \cup, \cap)$
or limit of $X^0 = \bot, X^{n+1} = F(X^n)$

Approximate analysis :

- Static approximation : we replace the concrete lattice $(\mathcal{P}(Env), \subseteq, \cup, \cap)$ by an abstract lattice $(L^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp)$
  - whose elements can be (efficiently) represented in computers,
  - in which we know how to compute $\sqcup^\sharp, \sqcap^\sharp, \sqsubseteq^\sharp, \ldots$

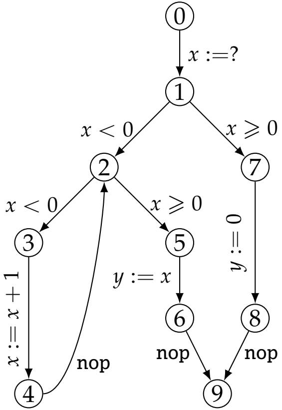
  and we "transpose" the equation $X = F(X)$ of $\mathcal{P}(Env)^N$ into $(L^\sharp)^N$.
- Dynamic approximation : when $L^\sharp$ does not verifies the ascending chain condition, the iterative computation may not terminate in a finite number of steps (or sometimes too slowly). In this case, we can only approximate the limit (see widening/narrowing).

Outline

Just put some $^\sharp$...

From $\mathcal{P}(Env)$ to $Env^\sharp$

control flow graph



collecting semantics

$$
\begin{array}{lcl}
X_0 & = & Env \\
X_1 & = & [\![x :=?]\!]\,(X_0) \\
X_2 & = & [\![x < 0]\!]\,(X_1) \cup X_4 \\
X_3 & = & [\![x < 0]\!]\,(X_2) \\
X_4 & = & [\![x := x + 1]\!]\,(X_3) \\
X_5 & = & [\![x \geqslant 0]\!]\,(X_2) \\
X_6 & = & [\![y := x]\!]\,(X_5) \\
X_7 & = & [\![x \geqslant 0]\!]\,(X_1) \\
X_8 & = & [\![y := 0]\!]\,(X_7) \\
X_9 & = & X_6 \cup X_8
\end{array}
$$

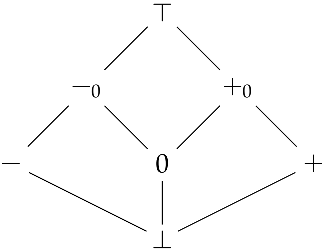
abstract semantics

$$
\begin{array}{lcl}
X_0^\sharp & = & \top_{Env}^\sharp \\
X_1^\sharp & = & [\![x :=?]\!]^\sharp\,(X_0^\sharp) \\
X_2^\sharp & = & [\![x < 0]\!]^\sharp\,(X_1^\sharp) \sqcup^\sharp X_4^\sharp \\
X_3^\sharp & = & [\![x < 0]\!]^\sharp\,(X_2^\sharp) \\
X_4^\sharp & = & [\![x := x + 1]\!]^\sharp\,(X_3^\sharp) \\
X_5^\sharp & = & [\![x \geqslant 0]\!]^\sharp\,(X_2^\sharp) \\
X_6^\sharp & = & [\![y := x]\!]^\sharp\,(X_5^\sharp) \\
X_7^\sharp & = & [\![x \geqslant 0]\!]^\sharp\,(X_1^\sharp) \\
X_8^\sharp & = & [\![y := 0]\!]^\sharp\,(X_7^\sharp) \\
X_9^\sharp & = & X_6^\sharp \sqcup^\sharp X_8^\sharp
\end{array}
$$

Abstract semantics : the ingredients

- A lattice structure $(Env^\sharp, \sqsubseteq^\sharp_{Env}, \sqcup^\sharp_{Env}, \sqcap^\sharp_{Env}, \perp^\sharp_{Env}, \top^\sharp_{Env})$
  - $\sqsubseteq^\sharp_{Env}$ is an approximation of $\subseteq$
  - $\sqcup^\sharp_{Env}$ is an approximation of $\cup$
  - $\sqcap^\sharp_{Env}$ is an approximation of $\cap$
  - $\perp^\sharp_{Env}$ is an approximation of $\emptyset$
  - $\top^\sharp_{Env}$ is an approximation of Env
- For all $x \in \mathbb{V}$,
  $[\![x :=?]\!]^\sharp \in Env^\sharp \to Env^\sharp$ an approximation of $[\![x :=?]\!]$
- For all $x \in \mathbb{V}$, $e \in Exp$,
  $[\![x := e]\!]^\sharp \in Env^\sharp \to Env^\sharp$ an approximation of $[\![x := e]\!]$
- For all $t \in Test$,
  $[\![t]\!]^\sharp \in Env^\sharp \to Env^\sharp$ an approximation of $[\![t]\!]$
- A concretisation $\gamma \in Env^\sharp \to \mathcal{P}(Env)$ that explains which property $\gamma(x^\sharp) \in \mathcal{P}(Env)$ is represented by each abstract element $x^\sharp \in Env^\sharp$.

An abstraction by signs



$\perp$	represents the property	$\emptyset$
$-$	represents the property	$\{ z \mid z < 0 \}$
0	represents the property	$\{0\}$
$+$	represents the property	$\{ z \mid z > 0 \}$
$-_0$	represents the property	$\{ z \mid z \leqslant 0 \}$
$+_0$	represents the property	$\{ z \mid z \geqslant 0 \}$
$\top$	represents the property	$\mathbb{Z}$

$\boldsymbol{Env}^\sharp \stackrel{\text{def}}{=} \mathbb{V} \rightarrow \boldsymbol{Sign}$: a sign is associated to each variable.

An abstraction by signs : example

$$
\begin{array}{lcl}
X_0^\sharp & = & \top^\sharp_{Env} \\
X_1^\sharp & = & [\![x := ?]\!]^\sharp\,(X_0^\sharp) \\
X_2^\sharp & = & [\![x < 0]\!]^\sharp\,(X_1^\sharp) \sqcup^\sharp X_4^\sharp \\
X_3^\sharp & = & [\![x < 0]\!]^\sharp\,(X_2^\sharp) \\
X_4^\sharp & = & [\![x := x + 1]\!]^\sharp\,(X_3^\sharp) \\
X_5^\sharp & = & [\![x \geqslant 0]\!]^\sharp\,(X_2^\sharp) \\
X_6^\sharp & = & [\![y := x]\!]^\sharp\,(X_5^\sharp) \\
X_7^\sharp & = & [\![x \geqslant 0]\!]^\sharp\,(X_1^\sharp) \\
X_8^\sharp & = & [\![y := 0]\!]^\sharp\,(X_7^\sharp) \\
X_9^\sharp & = & X_6^\sharp \sqcup^\sharp X_8^\sharp
\end{array}
\quad \xrightarrow[\text{simplifies into}]{\text{which}} \quad
\begin{array}{lcl}
X_0^\sharp & = & [x : \top;\ y : \top] \\
X_1^\sharp & = & X_0^\sharp[x \mapsto \top] \\
X_2^\sharp & = & X_1^\sharp[x \mapsto -] \sqcup^\sharp X_4^\sharp \\
X_3^\sharp & = & X_2^\sharp[x \mapsto -] \\
X_4^\sharp & = & X_3^\sharp[x \mapsto \mathrm{succ}^\sharp(X_3^\sharp(x))] \\
X_5^\sharp & = & X_2^\sharp[x \mapsto +_0] \\
X_6^\sharp & = & X_5^\sharp[y \mapsto X_5^\sharp(x)] \\
X_7^\sharp & = & X_1^\sharp[x \mapsto +_0] \\
X_8^\sharp & = & X_7^\sharp[y \mapsto 0] \\
X_9^\sharp & = & X_6^\sharp \sqcup^\sharp X_8^\sharp
\end{array}
$$

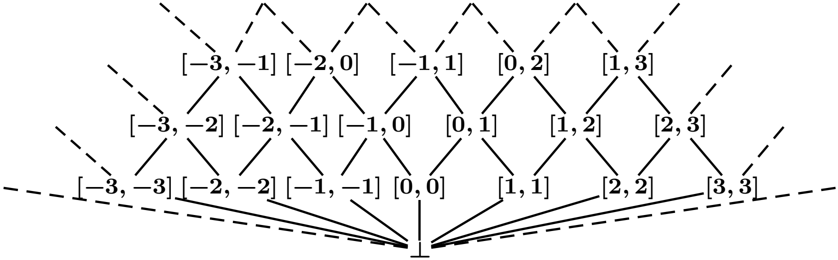
with

$$
\begin{array}{rcl}
\mathrm{succ}^\sharp(\bot) & = & \bot \\
\mathrm{succ}^\sharp(-) & = & -_0 \\
\mathrm{succ}^\sharp(0) & = & \mathrm{succ}^\sharp(+) = \mathrm{succ}^\sharp(+_0) = + \\
\mathrm{succ}^\sharp(-_0) & = & \mathrm{succ}^\sharp(\top) = \top
\end{array}
$$

Abstraction by intervals

$$Int \stackrel{\text{def}}{=} \{\, [a,b] \mid a,b \in \overline{\mathbb{Z}},\ a \leqslant b \,\} \cup \{\bot\}$$

with $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, +\infty\}$.
$\bot$ represents $\emptyset$ and $[a,b]$ the property $\{z \mid a \leqslant z \leqslant b\}$.



$\boldsymbol{Env^\sharp} \stackrel{\text{def}}{=} \mathbb{V} \rightarrow \boldsymbol{Int}$: an interval is associated to each variable.

Abstraction by intervals : example

$$
\begin{array}{rcl}
X_0^\sharp &=& \top_{Env}^\sharp \\
X_1^\sharp &=& [\![x := ?]\!]^\sharp\,(X_0^\sharp) \\
X_2^\sharp &=& [\![x < 0]\!]^\sharp\,(X_1^\sharp) \sqcup^\sharp X_4^\sharp \\
X_3^\sharp &=& [\![x < 0]\!]^\sharp\,(X_2^\sharp) \\
X_4^\sharp &=& [\![x := x + 1]\!]^\sharp\,(X_3^\sharp) \\
X_5^\sharp &=& [\![x \geqslant 0]\!]^\sharp\,(X_2^\sharp) \\
X_6^\sharp &=& [\![y := x]\!]^\sharp\,(X_5^\sharp) \\
X_7^\sharp &=& [\![x \geqslant 0]\!]^\sharp\,(X_1^\sharp) \\
X_8^\sharp &=& [\![y := 0]\!]^\sharp\,(X_7^\sharp) \\
X_9^\sharp &=& X_6^\sharp \sqcup^\sharp X_8^\sharp
\end{array}
\qquad
\begin{array}{rcl}
X_0^\sharp &=& [x : [-\infty, +\infty];\ y : [-\infty, +\infty]] \\
X_1^\sharp &=& X_0^\sharp[x \mapsto [-\infty, +\infty]] \\
X_2^\sharp &=& X_1^\sharp[x \mapsto X_1^\sharp(x) \sqcap^\sharp [-\infty, -1]] \sqcup^\sharp X_4^\sharp \\
X_3^\sharp &=& X_2^\sharp[x \mapsto X_2^\sharp(x) \sqcap^\sharp [-\infty, -1]] \\
X_4^\sharp &=& X_3^\sharp[x \mapsto \mathrm{succ}^\sharp(X_3^\sharp(x))] \\
X_5^\sharp &=& X_2^\sharp[x \mapsto X_2^\sharp(x) \sqcap^\sharp [0, +\infty]] \\
X_6^\sharp &=& X_5^\sharp[y \mapsto X_5^\sharp(x)] \\
X_7^\sharp &=& X_1^\sharp[x \mapsto X_1^\sharp(x) \sqcap^\sharp [0, +\infty]] \\
X_8^\sharp &=& X_7^\sharp[y \mapsto [0, 0]] \\
X_9^\sharp &=& X_6^\sharp \sqcup^\sharp X_8^\sharp
\end{array}
$$

with

$$
\begin{array}{rcl}
\mathrm{succ}^\sharp(\bot) &=& \bot \\
\mathrm{succ}^\sharp([a, b]) &=& [a + 1, b + 1]
\end{array}
$$