Building Verified Program Analyzers in Coq

Lecture 3: A verified abstract interpreter for a simple imperative language

David Pichardie - INRIA Rennes / Harvard University

Lecture 1

Motivations

Examples of verified analysers

Lecture 2

Coq crash course

Lecture 3

A flavor of abstraction interpretation

Verified abstract interpreter for a simple imperative language

Lecture 4

CompCert

A verified value analysis for CompCert

Lecture 1

Motivations

Examples of verified analysers

Lecture 2

Coq crash course

Lecture 3

A flavor of abstraction interpretation

Verified abstract interpreter for a simple imperative language

Lecture 4

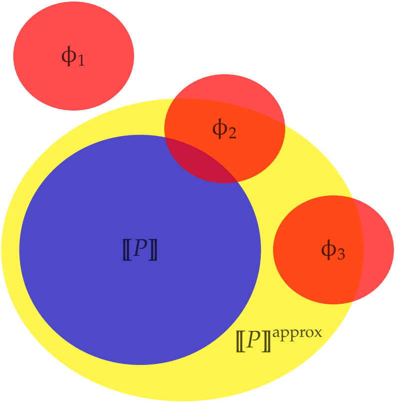
CompCert

A verified value analysis for CompCert

A flavor of Abstract Interpretation

David Pichardie

Static analysis computes approximations [1]



- 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)

1. see `http://www.astree.ens.fr/IntroAbsInt.html`

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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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 (monotone operator on a complete lattice)

```
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

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

- 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 (?) {

    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

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

- 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

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

- 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;


}
```

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;
        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

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

- 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

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

- 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

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

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

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

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

```
while i<T.length {
```

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

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

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

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

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

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

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

```
    };
```

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

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

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

```
};
```

$$\{i = T.length\}$$

Polyhedral abstract interpretation

Automatic discovery of linear restraints among variables of a program.
P. Cousot and N. Halbwachs. POPL'78.

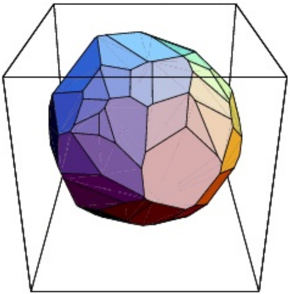
Patrick Cousot

Nicolas Halbwachs

Polyhedral analysis seeks to discover invariant linear equality and inequality relationships among the variables of an imperative program.

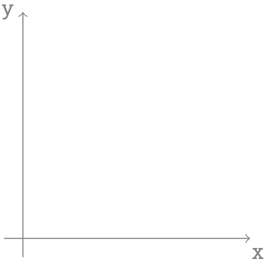
Convex polyhedra

A convex polyhedron can be defined algebraically as the set of solutions of a system of linear inequalities.
Geometrically, it can be defined as a finite intersection of half-spaces.



Polyhedral analysis

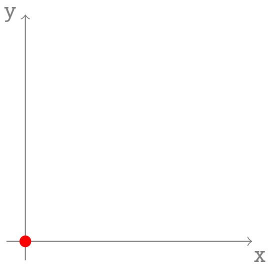
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;


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

    y = y+2;

  };


  x = x+1;

}
```

Polyhedral analysis

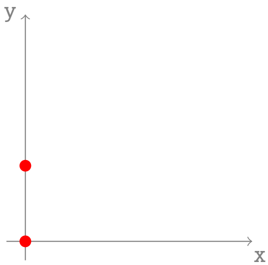
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
        {x = 0 ∧ y = 0}

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

  };


  x = x+1;

}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



At junction points, we over-approximates union by a convex union.

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

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

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

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

```
    y = y+2;
```

$\{x = 0 \wedge y = 2\}$

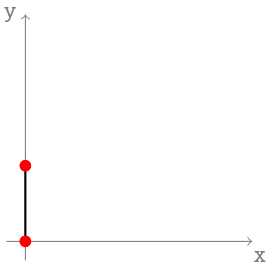
```
  };
```

$\{x = 0 \wedge y = 0\} \uplus \{x = 0 \wedge y = 2\}$

```
  x = x+1;

}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



At junction points, we over-approximates union by a convex union.

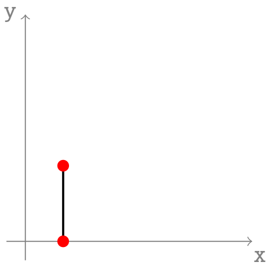
```
x = 0; y = 0;
```
$\{x = 0 \wedge y = 0\}$

```
while (x<6) {
  if (?) {
```
$\{x = 0 \wedge y = 0\}$
```
    y = y+2;
```
$\{x = 0 \wedge y = 2\}$
```
  };
```
$\{x = 0 \wedge 0 \leqslant y \leqslant 2\}$

```
  x = x+1;

}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



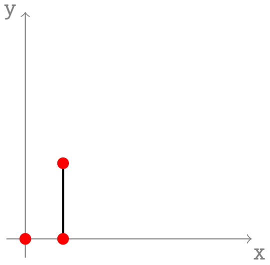

```
x = 0; y = 0;
```
$\{x = 0 \land y = 0\}$

```
while (x<6) {
  if (?) {
```
$\{x = 0 \land y = 0\}$
```
    y = y+2;
```
$\{x = 0 \land y = 2\}$
```
  };
```
$\{x = 0 \land 0 \leqslant y \leqslant 2\}$

```
  x = x+1;
```
$\{x = 1 \land 0 \leqslant y \leqslant 2\}$
```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




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

$\{x = 0 \wedge y = 0\} \uplus \{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

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

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

```
    y = y+2;
```

$\{x = 0 \wedge y = 2\}$

```
  };
```

$\{x = 0 \wedge 0 \leqslant y \leqslant 2\}$

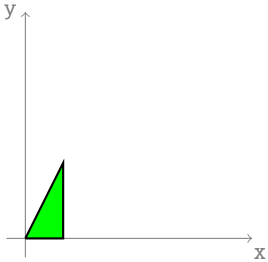
```
  x = x+1;
```

$\{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




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

$\{x \leqslant 1 \ \wedge \ 0 \leqslant y \leqslant 2x\}$

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

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

```
    y = y+2;
```

$\{x = 0 \ \wedge \ y = 2\}$

```
  };
```

$\{x = 0 \ \wedge \ 0 \leqslant y \leqslant 2\}$

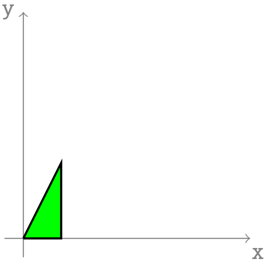
```
  x = x+1;
```

$\{x = 1 \ \wedge \ 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




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

$\{x \leqslant 1 \ \wedge\ 0 \leqslant y \leqslant 2x\}$

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

$\{x \leqslant 1 \ \wedge\ 0 \leqslant y \leqslant 2x\}$

```
    y = y+2;
```

$\{x = 0 \ \wedge\ y = 2\}$

```
  };
```

$\{x = 0 \ \wedge\ 0 \leqslant y \leqslant 2\}$

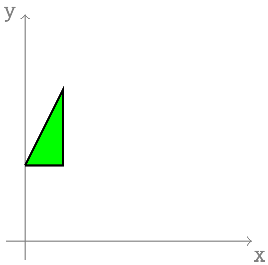
```
  x = x+1;
```

$\{x = 1 \ \wedge\ 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




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

$\{x \leqslant 1 \ \wedge\ 0 \leqslant y \leqslant 2x\}$

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

$\{x \leqslant 1 \ \wedge\ 0 \leqslant y \leqslant 2x\}$

```
    y = y+2;
```

$\{x \leqslant 1 \ \wedge\ 2 \leqslant y \leqslant 2x + 2\}$

```
  };
```

$\{x = 0 \ \wedge\ 0 \leqslant y \leqslant 2\}$

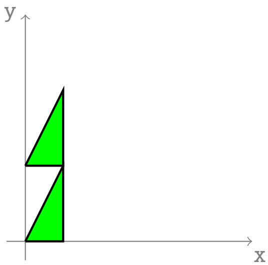
```
  x = x+1;
```

$\{x = 1 \ \wedge\ 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
```
$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

```
while (x<6) {
  if (?) {
```
$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

```
    y = y+2;
```
$\{x \leqslant 1 \wedge 2 \leqslant y \leqslant 2x + 2\}$

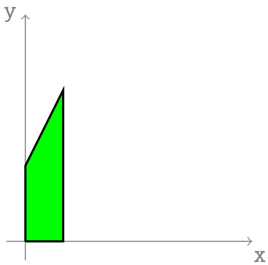
```
  };
```
$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$
$\uplus\{x \leqslant 1 \wedge 2 \leqslant y \leqslant 2x + 2\}$

```
  x = x+1;
```
$\{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




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

$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

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

$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

```
    y = y+2;
```

$\{x \leqslant 1 \wedge 2 \leqslant y \leqslant 2x + 2\}$

```
  };
```

$\{0 \leqslant x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x + 2\}$

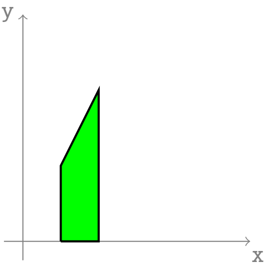
```
  x = x+1;
```

$\{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



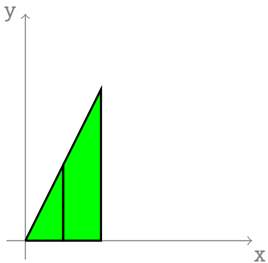

```
x = 0; y = 0;
```
$\{x \leqslant 1 \ \wedge \ 0 \leqslant y \leqslant 2x\}$

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

```
  x = x+1;
```
$\{1 \leqslant x \leqslant 2 \ \wedge \ 0 \leqslant y \leqslant 2x\}$
```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




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

$\{x \leqslant 1 \land 0 \leqslant y \leqslant 2x\}$

$\nabla \{x \leqslant 2 \land 0 \leqslant y \leqslant 2x\}$

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

$\{x \leqslant 1 \land 0 \leqslant y \leqslant 2x\}$

```
    y = y+2;
```

$\{x \leqslant 1 \land 2 \leqslant y \leqslant 2x+2\}$

```
  };
```

$\{0 \leqslant x \leqslant 1 \land 0 \leqslant y \leqslant 2x+2\}$

```
  x = x+1;
```

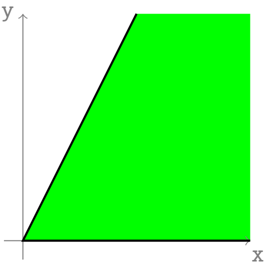
$\{1 \leqslant x \leqslant 2 \land 0 \leqslant y \leqslant 2x\}$

```
}
```

At loop headers, we use heuristics (widening) to ensure finite convergence.

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



At loop headers, we use heuristics (widening) to ensure finite convergence.

```
x = 0; y = 0;
        {0 ⩽ y ⩽ 2x}

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

  x = x+1;
        {1 ⩽ x ⩽ 2 ∧ 0 ⩽ y ⩽ 2x}
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.

By propagation we obtain a post-fixpoint

```
x = 0; y = 0;
```
$\{0 \leqslant y \leqslant 2x\}$

```
while (x<6) {
  if (?) {
```
$\{0 \leqslant y \leqslant 2x \ \wedge \ x \leqslant 5\}$
```
    y = y+2;
```
$\{2 \leqslant y \leqslant 2x+2 \ \wedge \ x \leqslant 5\}$
```
  };
```
$\{0 \leqslant y \leqslant 2x+2 \ \wedge \ 0 \leqslant x \leqslant 5\}$

```
  x = x+1;
```
$\{0 \leqslant y \leqslant 2x \ \wedge \ 1 \leqslant x \leqslant 6\}$
```
}
```
$\{0 \leqslant y \leqslant 2x \ \wedge \ 6 \leqslant x\}$

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.

By propagation we obtain a post-fixpoint which is enhanced by downward iteration.

```
x = 0; y = 0;
```
$\{0 \leqslant y \leqslant 2x \wedge x \leqslant 6\}$

```
while (x<6) {
  if (?) {
```
$\{0 \leqslant y \leqslant 2x \wedge x \leqslant 5\}$
```
    y = y+2;
```
$\{2 \leqslant y \leqslant 2x+2 \wedge x \leqslant 5\}$
```
  };
```
$\{0 \leqslant y \leqslant 2x+2 \wedge 0 \leqslant x \leqslant 5\}$

```
  x = x+1;
```
$\{0 \leqslant y \leqslant 2x \wedge 1 \leqslant x \leqslant 6\}$
```
}
```
$\{0 \leqslant y \leqslant 2x \wedge 6 = x\}$

Polyhedral analysis

A more complex example.

The analysis accepts to replace some constants by parameters.

```
x = 0; y = A;
```
$$\{A \leqslant y \leqslant 2x + A \wedge x \leqslant N\}$$

```
while (x<N) {
  if (?) {
```
$$\{A \leqslant y \leqslant 2x + A \;\wedge\; x \leqslant N - 1\}$$
```
    y = y+2;
```
$$\{A + 2 \leqslant y \leqslant 2x + A + 2 \;\wedge\; x \leqslant N - 1\}$$
```
  };
```
$$\{A \leqslant y \leqslant 2x + A + 2 \;\wedge\; 0 \leqslant x \leqslant N - 1\}$$

```
  x = x+1;
```
$$\{A \leqslant y \leqslant 2x + A \;\wedge\; 1 \leqslant x \leqslant N\}$$
```
}
```
$$\{A \leqslant y \leqslant 2x + A \;\wedge\; N = x\}$$

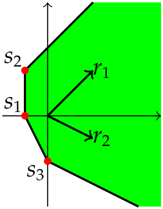
The four polyhedra operations

- $\uplus \in \mathbb{P}_n \times \mathbb{P}_n \to \mathbb{P}_n$: convex union
  - over-approximates the concrete union at junction points
- $\cap \in \mathbb{P}_n \times \mathbb{P}_n \to \mathbb{P}_n$: intersection
  - over-approximates the concrete intersection after a conditional intruction
- $[\![\mathtt{x} := e]\!] \in \mathbb{P}_n \to \mathbb{P}_n$: affine transformation
  - over-approximates the assignment of a variable by a linear expression
- $\nabla \in \mathbb{P}_n \times \mathbb{P}_n \to \mathbb{P}_n$: widening
  - ensures (and accelerates) convergence of (post-)fixpoint iteration
  - includes heuristics to infer loop invariants

```
x = 0; y = 0;
```
$$P_0 = [\![\mathtt{y} := \mathtt{0}]\!]\,[\![\mathtt{x} := \mathtt{0}]\!]\,(\mathbb{Q}^2) \,\nabla\, P_4$$
```
while (x<6) {
  if (?) {
```
$$P_1 = P_0 \cap \{x < 6\}$$
```
    y = y+2;
```
$$P_2 = [\![\mathtt{y} := \mathtt{y} + \mathtt{2}]\!]\,(P_1)$$
```
  };
```
$$P_3 = P_1 \uplus P_2$$
```
  x = x+1;
```
$$P_4 = [\![\mathtt{x} := \mathtt{x} + \mathtt{1}]\!]\,(P_3)$$
```
}
```
$$P_5 = P_0 \cap \{x \geqslant 6\}$$

Library for manipulating polyhedra

- Parma Polyhedra Library [2] (PPL), NewPolka
- They rely on the Double Description Method
  - polyhedra are managed using two representations in parallel



  - by set of inequalities

$$P = \left\{ (x,y) \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} x \geqslant -1 \\ x - y \geqslant -3 \\ 2x + y \geqslant -2 \\ x + 2y \geqslant -4 \end{array} \right\}$$

  - by set of generators

$$P = \left\{ \lambda_1 s_1 + \lambda_2 s_2 + \lambda_3 s_3 + \mu_1 r_1 + \mu_2 r_2 \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} \lambda_1, \lambda_2, \lambda_3, \mu_1, \mu_2 \in \mathbb{R}^+ \\ \lambda_1 + \lambda_2 + \lambda_3 = 1 \end{array} \right\}$$

  - operations efficiency strongly depends on the chosen representations, so they keep both

2. Previous tutorial on polyhedra partially comes from `http://www.cs.unipr.it/ppl/`

This lecture

We study a small abstract interpreter

- following Cousot's lecture notes
- represents an embryo of the Astrée analyser

Challenges

- be able to follow the textbook approach without remodeling the algorithms and the proofs
- first machine-checked instance of the motto
  « *my abstract interpreter is correct by construction* »

Language Syntax

```
Inductive stmt :=
    Assign            (x:var) (e:expr)
  | Skip
  | Assert            (t:test)
  | If          (t:test) (b1 b2:stmt)
  | While           (t:test) (stmt)
  | Seq (i1 i2:stmt).
```

Language Syntax

```
Inductive stmt :=
    Assign (p:pp) (x:var) (e:expr)
  | Skip (p:pp)
  | Assert (p:pp) (t:test)
  | If (p:pp) (t:test) (b1 b2:stmt)
  | While (p:pp) (t:test) (stmt)
  | Seq (i1 i2:stmt).
```

Instructions are labelled (program points)

Language Syntax

```
Definition word := bin 32.
Definition var := word.
Definition pp := word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
   Const (n:Z)
 | Unknown
 | Var (x:var)
 | Numop (o:op) (e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
 | Numcomp (c:comp) (e1 e2:expr)
 | Not (t:test)
 | And (t1 t2:test)
 | Or (t1 t2:test).
```

```
Inductive stmt :=
   Assign (p:pp) (x:var) (e:expr)
 | Skip (p:pp)
 | Assert (p:pp) (t:test)
 | If (p:pp) (t:test) (b1 b2:stmt)
 | While (p:pp) (t:test) (stmt)
 | Seq (i1 i2:stmt).
```

```
Record program := {
  p_stmt: stmt;
  p_end: pp;
  vars: list var
}.
```

Language Syntax

binary numbers with at most 32 bits (see Lecture 2), useful to prove termination

```
Definition word := bin 32.
Definition var := word.
Definition pp := word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
   Const (n:Z)
 | Unknown
 | Var (x:var)
 | Numop (o:op) (e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
 | Numcomp (c:comp) (e1 e2:expr)
 | Not (t:test)
 | And (t1 t2:test)
 | Or (t1 t2:test).
```

```
Inductive stmt :=
   Assign (p:pp) (x:var) (e:expr)
 | Skip (p:pp)
 | Assert (p:pp) (t:test)
 | If (p:pp) (t:test) (b1 b2:stmt)
 | While (p:pp) (t:test) (stmt)
 | Seq (i1 i2:stmt).

Record program := {
  p_stmt: stmt;
  p_end: pp;
  vars: list var
}.
```

Language Syntax

```
Definition word := bin 32.
Definition var := word.
Definition pp := word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
   Const (n:Z)
 | Unknown
 | Var (x:var)
 | Numop (o:op) (e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
 | Numcomp (c:comp) (e1 e2:expr)
 | Not (t:test)
 | And (t1 t2:test)
 | Or (t1 t2:test).
```

```
Inductive stmt :=
   Assign (p:pp) (x:var) (e:expr)
 | Skip (p:pp)
 | Assert (p:pp) (t:test)
 | If (p:pp) (t:test) (b1 b2:stmt)
 | While (p:pp) (t:test) (stmt)
 | Seq (i1 i2:stmt).
```

```
Record program := {
  p_stmt: stmt;
  p_end: pp;
  vars: list var
}.
```

main statement

Language Syntax

```
Definition word := bin 32.
Definition var := word.
Definition pp := word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
   Const (n:Z)
 | Unknown
 | Var (x:var)
 | Numop (o:op) (e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
 | Numcomp (c:comp) (e1 e2:expr)
 | Not (t:test)
 | And (t1 t2:test)
 | Or (t1 t2:test).
```

```
Inductive stmt :=
   Assign (p:pp) (x:var) (e:expr)
 | Skip (p:pp)
 | Assert (p:pp) (t:test)
 | If (p:pp) (t:test) (b1 b2:stmt)
 | While (p:pp) (t:test) (stmt)
 | Seq (i1 i2:stmt).
```

```
Record program := {
  p_stmt: stmt;
  p_end: pp;
  vars: list var
}.
```

last
label

Language Syntax

```
Definition word := bin 32.
Definition var := word.
Definition pp := word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
   Const (n:Z)
 | Unknown
 | Var (x:var)
 | Numop (o:op) (e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
 | Numcomp (c:comp) (e1 e2:expr)
 | Not (t:test)
 | And (t1 t2:test)
 | Or (t1 t2:test).
```

```
Inductive stmt :=
   Assign (p:pp) (x:var) (e:expr)
 | Skip (p:pp)
 | Assert (p:pp) (t:test)
 | If (p:pp) (t:test) (b1 b2:stmt)
 | While (p:pp) (t:test) (stmt)
 | Seq (i1 i2:stmt).
```

```
Record program := {
  p_stmt: stmt;
  p_end: pp;
  vars: list var
}.
```

variable declaration

Language Semantics

Semantic Domains

```
Definition env := var → Z.
Inductive config := Final (ρ:env) | Inter (i:instr) (ρ:env).
```

Language Semantics

Semantic Domains

```
Definition env := var → Z.
Inductive config := Final (ρ:env) | Inter (i:instr) (ρ:env).
```

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ l x e n ρ1 ρ2,
  sem_expr p ρ1 e n → subst ρ1 x n ρ2 → In x (vars p) →
  sos p (Assign l x e,ρ1) (Final ρ2)
[...]
```

Language Semantics

Semantic Domains

```
Definition env := var → Z.
Inductive config := Final (ρ:env) | Inter (i:instr) (ρ:env).
```

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ l x e n ρ1 ρ2,
  sem_expr p ρ1 e n → subst ρ1 x n ρ2 → In x (vars p) →
  sos p (Assign l x e,ρ1) (Final ρ2)
[...]
```

$$\frac{\mathtt{sem\_expr}\ p\ \rho_1\ e\ n \quad \rho_2 = \rho_1[x \mapsto n] \quad x \in (\mathtt{vars}\ p)}{\mathtt{sos}\ p\ (\mathtt{Assign}\ l\ x\ e, \rho_1)\ (\mathtt{Final}\ \rho_2)}$$

Language Semantics

Semantic Domains

```
Definition env := var → Z.
Inductive config := Final (ρ:env) | Inter (i:instr) (ρ:env).
```

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ l x e n ρ1 ρ2,
  sem_expr p ρ1 e n → subst ρ1 x n ρ2 → In x (vars p) →
  sos p (Assign l x e,ρ1) (Final ρ2)
[...]
```

$$\frac{\texttt{sem\_expr}\ p\ \rho_1\ e\ n \qquad \rho_2 = \rho_1[x \mapsto n] \qquad x \in (\texttt{vars}\ p)}{\texttt{sos}\ p\ (\texttt{Assign}\ l\ x\ e, \rho_1)\ (\texttt{Final}\ \rho_2)}$$

Language Semantics

Semantic Domains

```
Definition env := var → Z.
Inductive config := Final (ρ:env) | Inter (i:instr) (ρ:env).
```

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ l x e n ρ1 ρ2,
  sem_expr p ρ1 e n → subst ρ1 x n ρ2 → In x (vars p) →
  sos p (Assign l x e,ρ1) (Final ρ2)
[...]
```

$$\frac{\texttt{sem\_expr}\ p\ \rho_1\ e\ n \qquad \rho_2 = \rho_1[x \mapsto n] \qquad x \in (\texttt{vars}\ p)}{\texttt{sos}\ p\ (\texttt{Assign}\ l\ x\ e, \rho_1)\ (\texttt{Final}\ \rho_2)}$$

Language Semantics

Semantic Domains

```
Definition env := var → Z.
Inductive config := Final (ρ:env) | Inter (i:instr) (ρ:env).
```

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ l x e n ρ1 ρ2,
  sem_expr p ρ1 e n → subst ρ1 x n ρ2 → In x (vars p) →
  sos p (Assign l x e,ρ1) (Final ρ2)
[...]
```

$$\frac{\texttt{sem\_expr}\ p\ \rho_1\ e\ n \qquad \boxed{\rho_2 = \rho_1[x \mapsto n]} \qquad x \in (\texttt{vars}\ p)}{\texttt{sos}\ p\ (\texttt{Assign}\ l\ x\ e, \rho_1)\ (\texttt{Final}\ \rho_2)}$$

Language Semantics

Semantic Domains

```
Definition env := var → Z.
Inductive config := Final (ρ:env) | Inter (i:instr) (ρ:env).
```

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ l x e n ρ1 ρ2,
  sem_expr p ρ1 e n → subst ρ1 x n ρ2 → In x (vars p) →
  sos p (Assign l x e,ρ1) (Final ρ2)
[...]
```

$$\frac{\texttt{sem\_expr}\ p\ \rho_1\ e\ n \quad \rho_2 = \rho_1[x \mapsto n] \quad x \in (\texttt{vars}\ p)}{\texttt{sos}\ p\ (\texttt{Assign}\ l\ x\ e, \rho_1)\ (\texttt{Final}\ \rho_2)}$$

Language Semantics

Structural Operational Semantics

```
Inductive sos (p:program) : Kind → (instr * env) → config → Prop :=
 | sos_affect : ∀ l x e n ρ1 ρ2,
     sem_expr p ρ1 e n →
     subst ρ1 x n ρ2 →
     In x (vars p) →
     sos p (KAssign x e) (Assign l x e,ρ1) (Final ρ2)
 | sos_skip : ∀ l ρ,
     sos p KSkip (Skip l,ρ) (Final ρ)
 | sos_assert_true : ∀ l t ρ,
     sem_test p ρ t true →
     sos p (KAssert t) (Assert l t,ρ) (Final ρ)
 | sos_if_true : ∀ l t b1 b2 ρ,
     sem_test p ρ t true →
     sos p (KAssert t) (If l t b1 b2,ρ) (Inter b1 ρ)
 | sos_if_false : ∀ l t b1 b2 ρ,
     sem_test p ρ t false →
     sos p (KAssert (Not t)) (If l t b1 b2,ρ) (Inter b2 ρ)
 | sos_while_true : ∀ l t b ρ,
     sem_test p ρ t true →
     sos p (KAssert t) (While l t b,ρ) (Inter (Seq b (While l t b)) ρ)
```

Language Semantics

Structural Operational Semantics

```
    sem_test p ρ t true →
    sos p (KAssert t) (Assert l t,ρ) (Final ρ)
 | sos_if_true : ∀ l t b1 b2 ρ,
    sem_test p ρ t true →
    sos p (KAssert t) (If l t b1 b2,ρ) (Inter b1 ρ)
 | sos_if_false : ∀ l t b1 b2 ρ,
    sem_test p ρ t false →
    sos p (KAssert (Not t)) (If l t b1 b2,ρ) (Inter b2 ρ)
 | sos_while_true : ∀ l t b ρ,
    sem_test p ρ t true →
    sos p (KAssert t) (While l t b,ρ) (Inter (Seq b (While l t b)) ρ)
 | sos_while_false : ∀ l t b ρ,
    sem_test p ρ t false →
    sos p (KAssert (Not t)) (While l t b,ρ) (Final ρ)
 | sos_seq1 : ∀ k i1 i2 ρ ρ',
    sos p k (i1,ρ) (Final ρ') →
    sos p (KSeq1 i1 (first i2)) (Seq i1 i2,ρ) (Inter i2 ρ')
 | sos_seq2 : ∀ k i1 i1' i2 ρ ρ',
    sos p k (i1,ρ) (Inter i1' ρ') →
    sos p (KSeq2 k) (Seq i1 i2,ρ) (Inter (Seq i1' i2) ρ').
```

Language Semantics

Reachable states from any initial environment

```
Inductive sos_plus (p:program) : (instr * env) → config → Prop :=
| sos_plus0 : ∀ i ρ, sos_plus p (i,ρ) (Inter i ρ)
| sos_plus1 : ∀ k s1 s2, sos p k s1 s2 → sos_plus p s1 s2
| sos_trans : ∀ k s1 i ρ s3,
  sos p k s1 (Inter i ρ) →
  sos_plus p (i,ρ) s3 → sos_plus p s1 s3.

Inductive reachable_sos (p:program) : pp*env → Prop :=
| reachable_sos_intermediate : ∀ ρ0 i ρ,
  sos_plus p (p_instr p,ρ0) (Inter  i ρ) →
  reachable_sos p (first i,ρ)
| reachable_sos_final : ∀ ρ0 ρ,
  sos_plus p (p_instr p,ρ0) (Final ρ) →
  reachable_sos p (p_end p,ρ).
```

Language Semantics

Reachable states from any initial environment

```
Fixpoint first (i:instr) : pp :=
  match i with
    | Assign p x e => p
    | Skip p => p
    | Assert p t => p
    | If p t i1 i2 => p
    | While p t i => p
    | Seq i1 i2 => first i1
  end.
```

```
Inductive reachable_sos (p:program) : pp*env → Prop :=
| reachable_sos_intermediate : ∀ ρ0 i ρ,
  sos_plus p (p_instr p,ρ0) (Inter  i ρ) →
  reachable_sos p (first i,ρ)
| reachable_sos_final : ∀ ρ0 ρ,
  sos_plus p (p_instr p,ρ0) (Final ρ) →
  reachable_sos p (p_end p,ρ).
```

Final Objective for today

Final Objective for today

The analyzer computes an abstract representation of the program semantics

```
Definition analyse : program → abdom := [...]
```

Final Objective for today

The analyzer computes an abstract representation of the program semantics

```
Definition analyse : program → abdom := [...]
```

Each abstract element is given a concretization in $\mathcal{P}(\mathrm{pp} \times \mathrm{env})$

```
Definition γ : abdom → (pp*env → Prop) := [...]
```

Final Objective for today

The analyzer computes an abstract representation of the program semantics

```
Definition analyse : program → abdom := [...]
```

Each abstract element is given a concretization in $\mathcal{P}(\mathrm{pp} \times \mathrm{env})$

```
Definition γ : abdom → (pp*env → Prop) := [...]
```

The analyzer must compute a correct over-approximation of the reachable states

```
Theorem analyse_correct : ∀ prog:program,
  reachable_sos prog ⊆ γ (analyse prog).
```

Roadmap

Standard Semantics

Abstract Semantics

Roadmap

Standard Semantics

direct soundness proof

Abstract Semantics

Previous works:

Y. Bertot. *Structural abstract interpretation, a formal study in Coq*. ALFA Summer School 2008

X. Leroy. *Mechanized semantics, with applications to program proof and compiler verification.* Marktoberdorf Summer School 2009

Roadmap

Standard Semantics

~~direct soundness proof~~

Abstract Semantics

Previous works:

> Y. Bertot. *Structural abstract interpretation, a formal study in Coq*. ALFA Summer School 2008
>
> X. Leroy. *Mechanized semantics, with applications to program proof and compiler verification.* Marktoberdorf Summer School 2009

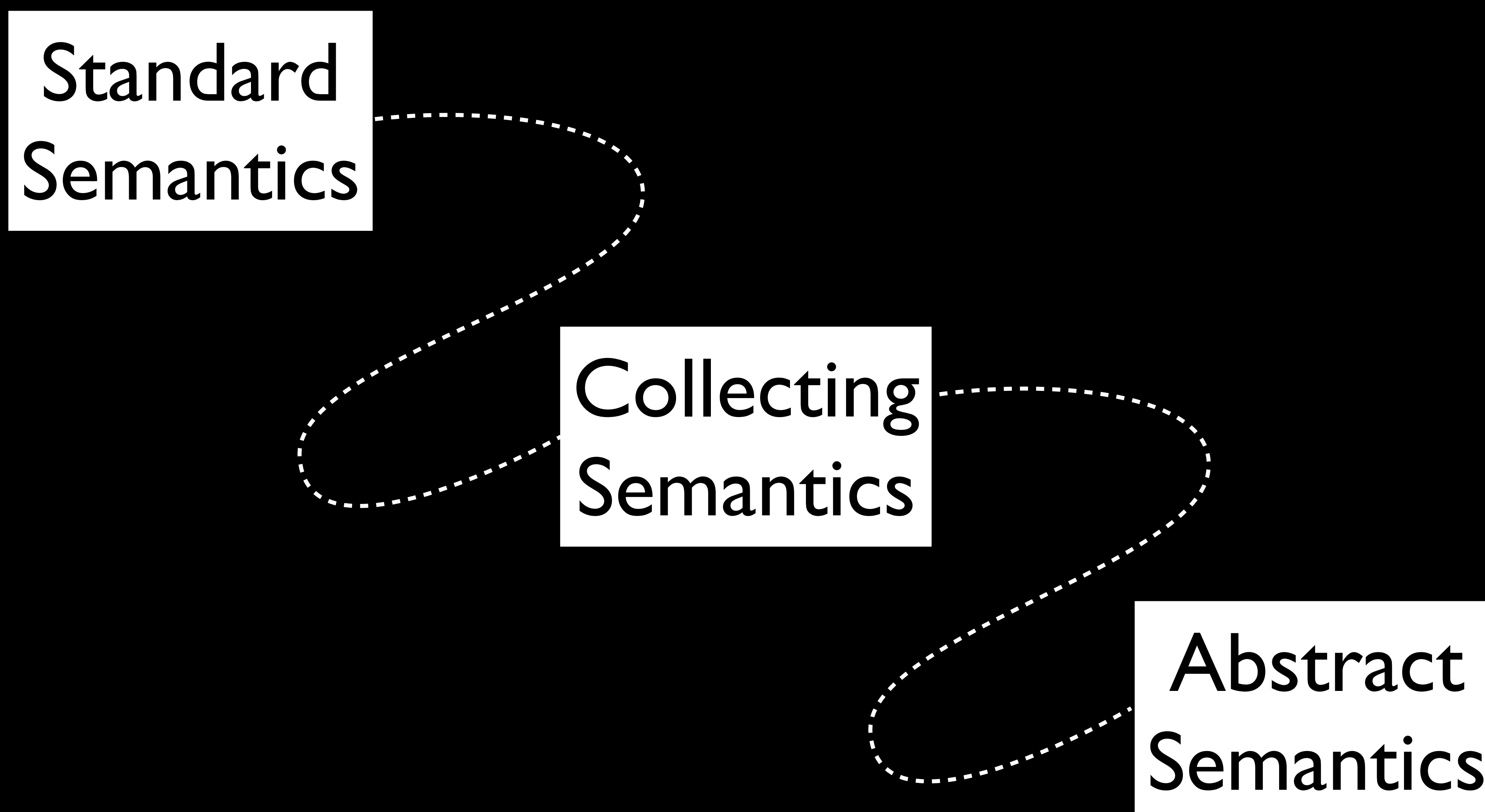
Roadmap

Standard Semantics

Collecting Semantics

Abstract Semantics

Roadmap


Standard Semantics
Collecting Semantics
Abstract Semantics

Roadmap

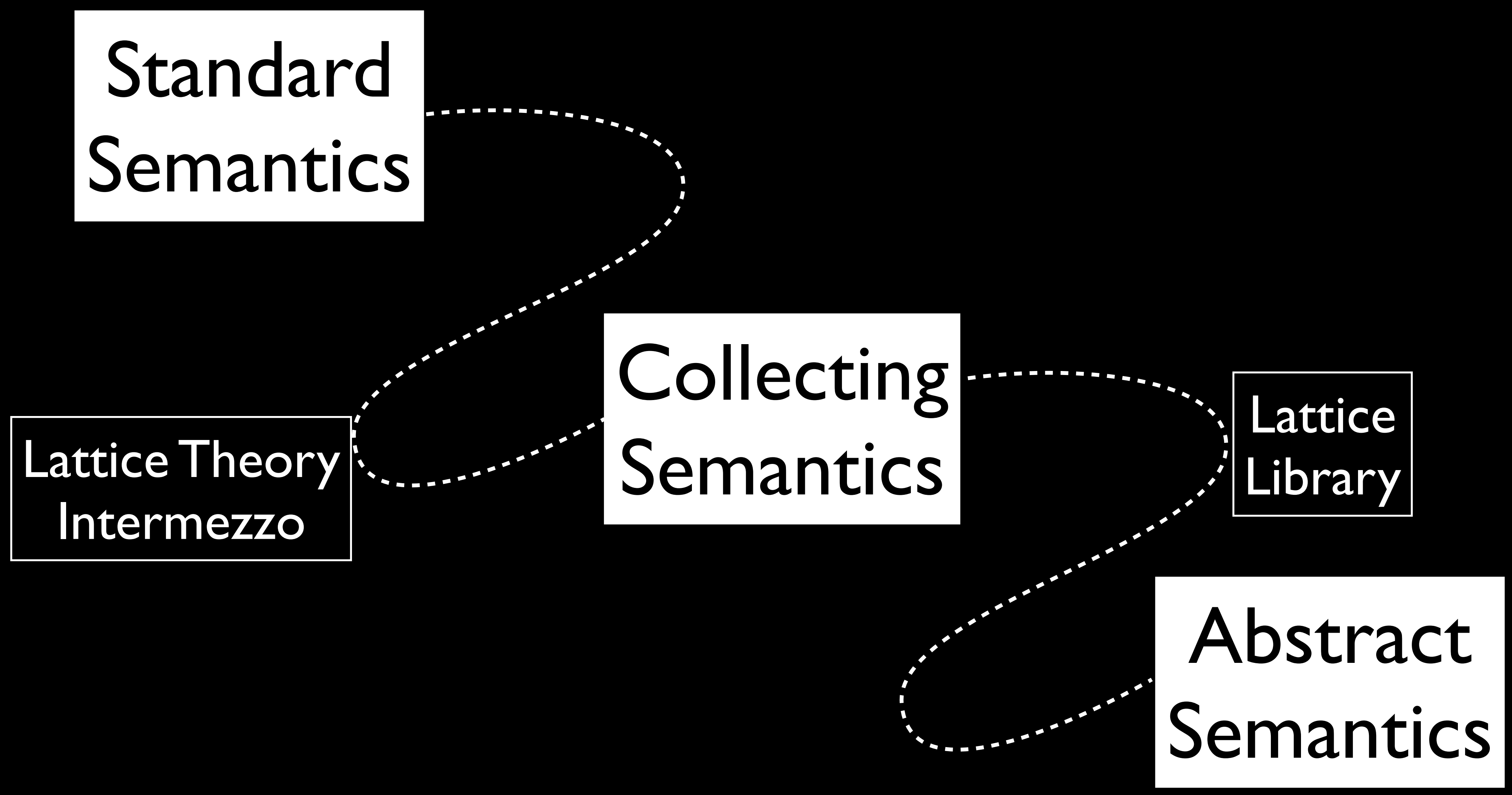
Standard Semantics

Lattice Theory Intermezzo

Collecting Semantics

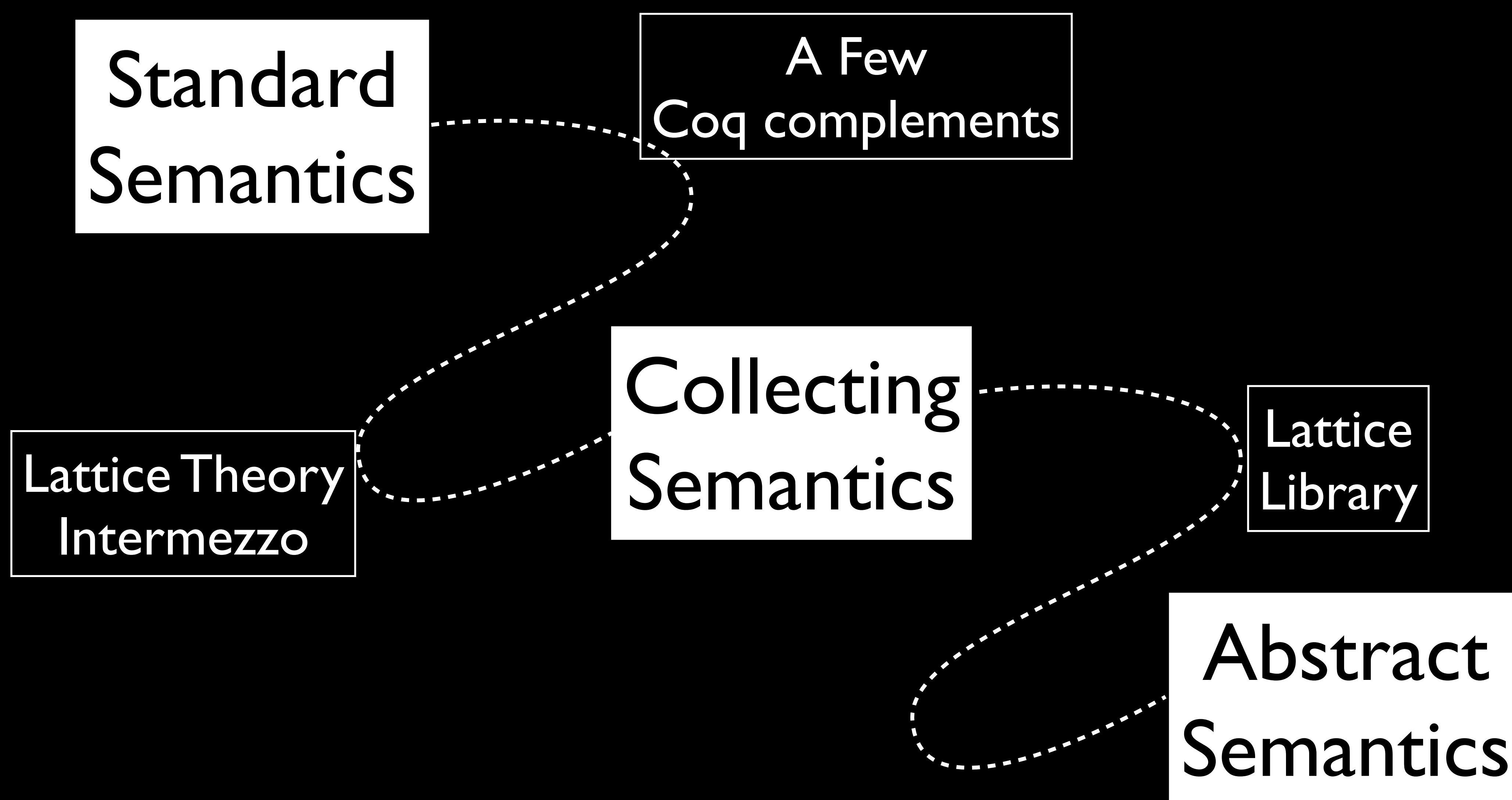
Abstract Semantics

Roadmap



Standard Semantics
Lattice Theory Intermezzo
Collecting Semantics
Lattice Library
Abstract Semantics

Roadmap


Standard Semantics
A Few Coq complements
Lattice Theory Intermezzo
Collecting Semantics
Lattice Library
Abstract Semantics

A Few Coq Complements

Programming in Coq

Coq allows to mix

- data types
- programs
- predicates
- proofs

```
Record t := {
  A : Type;
  f1 : A → A;
  f2 : A → A;
  P : A → A → Prop;
  prop : ∀a:A, P (f1 a) (f2 a)
}.
```

All elements share a same representation: typed λ-term in the Calculus of Construction.

Extracting to OCaml

Extraction mechanism is automatic but may fail to generate well-typed OCaml programs.

```
Record t := {
  A : Type;
  f1 : A → A;
  f2 : A → A;
  P : A → A → Prop;
  prop : ∀ a:A,
    P (f1 a) (f2 a)
}.
```

Coq

```
type __ = Obj.t

type t = {
  f1 : (__ → __);
  f2 : (__ → __)
}
```

OCaml

Extracting to OCaml

A better choice.

```
Record t (A:Type) := {
  f1 : A → A;
  f2 : A → A;
  P : A → A → Prop;
  prop : ∀ a,
    P (f1 a) (f2 a)
}.
```

Coq

```
type 'a t = {
  f1 : ('a → 'a);
  f2 : ('a → 'a)
}
```

OCaml

Coq records for algebraic structures

Record types are usefull for algebraic structures

```
Record lattice (A:Type) := { [...];
      order : A → A → Prop;
      order_refl: [...];
      order_antisym: [...];
      order_trans: ∀ x y z,
        order x y → order y z → order x z; [...] }.
```

Basic instanciations

```
Definition sign_lattice : lattice sign := [...]
```

Functors

```
Definition prod_lattice
   (A1:Type) (L1:lattice A1)
   (A2:Type) (L2:Lattice A2) : Lattice (A1*A2) := [...]
```

Coq records for algebraic structures

Given s1,...,s4 of type sign, how to write (s1,s2) $\sqsubseteq$ (s3,s4) ?

Coq records for algebraic structures

Given s1,...,s4 of type sign, how to write (s1,s2) $\sqsubseteq$ (s3,s4) ?

```
order (sign*sign) _ (s1,s2) (s3,s4)
```

?

Coq records for algebraic structures

Given s1,...,s4 of type sign, how to write (s1,s2) $\sqsubseteq$ (s3,s4) ?

```
order (sign*sign) _ (s1,s2) (s3,s4)
```

?

We need to fill the hole with a term of type `lattice (sign*sign)`

```
Definition L : lattice (sign*sign) :=
  prod_lattice sign sign_lattice sign sign_lattice.
```

Coq records for algebraic structures

Given s1,...,s4 of type sign, how to write (s1,s2) $\sqsubseteq$ (s3,s4) ?

```
order (sign*sign) _ (s1,s2) (s3,s4)
```

We need to fill the hole with a term of type `lattice (sign*sign)`

```
Definition L : lattice (sign*sign) :=
  prod_lattice sign sign_lattice sign sign_lattice.
```

The recent Coq type class system (Sozeau & Oury) is able to infer itself the hole.

```
order _ _ (s1,s2) (s3,s4)
```

Coq records for algebraic structures

Given s1,...,s4 of type sign, how to write (s1,s2) $\sqsubseteq$ (s3,s4) ?

```
order (sign*sign) _ (s1,s2) (s3,s4)
```

We need to fill the hole with a term of type `lattice (sign*sign)`

```
Definition L : lattice (sign*sign) :=
  prod_lattice sign sign_lattice sign sign_lattice.
```

The recent Coq type class system (Sozeau & Oury) is able to infer itself the hole.

```
order _ _ (s1,s2) (s3,s4)
```

Notation overloading !

```
Notation "x ⊑ y" := (order _ _ x y).
```

Lattice Theory Intermezzo

A Few Lattice Theory

We need a least-fixpoint operator in Coq

- Formalization of complete lattices
- Proof of Knaster-Tarski theorem
- Construction of some useful complete lattices

Knaster-Tarski Theorem

```
Definition lfp {L} {CompleteLattice.t L} (f:monotone L L) :
  CompleteLattice.meet (PostFix f).
```

Knaster-Tarski Theorem

Complete lattices on elements of type A

Monotone functions from L to L

```
Definition lfp {L} {CompleteLattice.t L} (f:monotone L L) :
  CompleteLattice.meet (PostFix f).
```

$$\bigsqcap \{x \mid f(x) \sqsubseteq x\}$$

Monotone functions

```
Class monotone A {Poset.t A} B {Poset.t B} : Type := Mono {
  mon_func : A → B;
  mon_prop : ∀ a1 a2,
    a1 ⊑ a2 → (mon_func a1) ⊑ (mon_func a2)
}.
```

A monotone function is a term
(Mono f π)

Knaster-Tarski Theorem

Complete lattices on elements of type A

Monotone functions from L to L

```
Definition lfp {L} {CompleteLattice.t L} (f:monotone L L) :
  CompleteLattice.meet (PostFix f).
```

$$\bigsqcap \{x \mid f(x) \sqsubseteq x\}$$

Knaster-Tarski Theorem

```
Definition lfp {L} {CompleteLattice.t L} (f:monotone L L) :
  CompleteLattice.meet (PostFix f).

Section KnasterTarski.
  Variable L : Type.
  Variable CL : CompleteLattice.t L.
  Variable f : monotone L L.
  Lemma lfp_fixpoint : f (lfp f) == lfp f. [...]
  Lemma lfp_least_fixpoint : ∀ x, f x == x → lfp f ⊑ x. [..
  Lemma lfp_postfixpoint : f (lfp f) ⊑ lfp f. [...]
  Lemma lfp_least_postfixpoint : ∀x, f x ⊑ x → lfp f ⊑ x. [
End KnasterTarski.
```

Knaster-Tarski Theorem

```
Definition lfp {L} {CompleteLattice.t L} (f:monotone L L) :
  CompleteLattice.meet (PostFix f)

Section KnasterTarski.
  Variable L : Type.
  Variable CL : CompleteLattice.t L.
  Variable f : monotone L L.
  Lemma lfp_fixpoint : f (lfp f) == lfp f. [...]
  Lemma lfp_least_fixpoint : ∀ x, f x == x → lfp f ⊑ x. [..
  Lemma lfp_postfixpoint : f (lfp f) ⊑ lfp f. [...]
  Lemma lfp_least_postfixpoint : ∀x, f x ⊑ x → lfp f ⊑ x. [
End KnasterTarski.
```

Coq Type Classes = Record + Inference (super) capabilities

Knaster-Tarski Theorem

We declare this
argument as implicit

```
Definition lfp {L} {CompleteLattice.t L} (f:monotone L L) :
  CompleteLattice.meet (PostFix f)

Section KnasterTarski.
  Variable L : Type.
  Variable CL : CompleteLattice.t L.
  Variable f : monotone L L.
  Lemma lfp_fixpoint : f (lfp f) == lfp f. [...]
  Lemma lfp_least_fixpoint : ∀          f ⊑ x. [..
  Lemma lfp_postfixpoint : f (l          .]
  Lemma lfp_least_postfixpoint          f ⊑ x. [
End KnasterTarski.
```

Coq Type Classes = Record + Inference (super) capabilities

The implicit argument of type
(CompleteLattice.t L)
is automatically inferred

Canonical Complete Lattices

```
Instance PowerSetCL A : CompleteLattice.t 𝒫(A) := [...]


Instance PointwiseCL A L {CompleteLattice.t L} :
                         CompleteLattice.t (A → L) := [...]
```

Canonical Complete Lattices

Notation for `(A->Prop)`

```
Instance PowerSetCL A : CompleteLattice.t 𝒫(A) := [...]
```

```
Instance PointwiseCL A L {CompleteLattice.t L} :
                         CompleteLattice.t (A → L) := [...]
```

Canonical Complete Lattices

Notation for (A->Prop)

```
Instance PowerSetCL A : CompleteLattice.t 𝒫(A) := [...]
```

Set inclusion ordering

```
Instance PointwiseCL A L {CompleteLattice.t L} :
                        CompleteLattice.t (A → L) := [...]
```

Canonical Complete Lattices

Notation for (A->Prop)

```
Instance PowerSetCL A : CompleteLattice.t 𝒫(A) := [...]
```

Set inclusion ordering

```
Instance PointwiseCL A L {CompleteLattice.t L} :
                         CompleteLattice.t (A → L) := [...]
```

Functor

Pointwise ordering

Canonical Complete Lattices

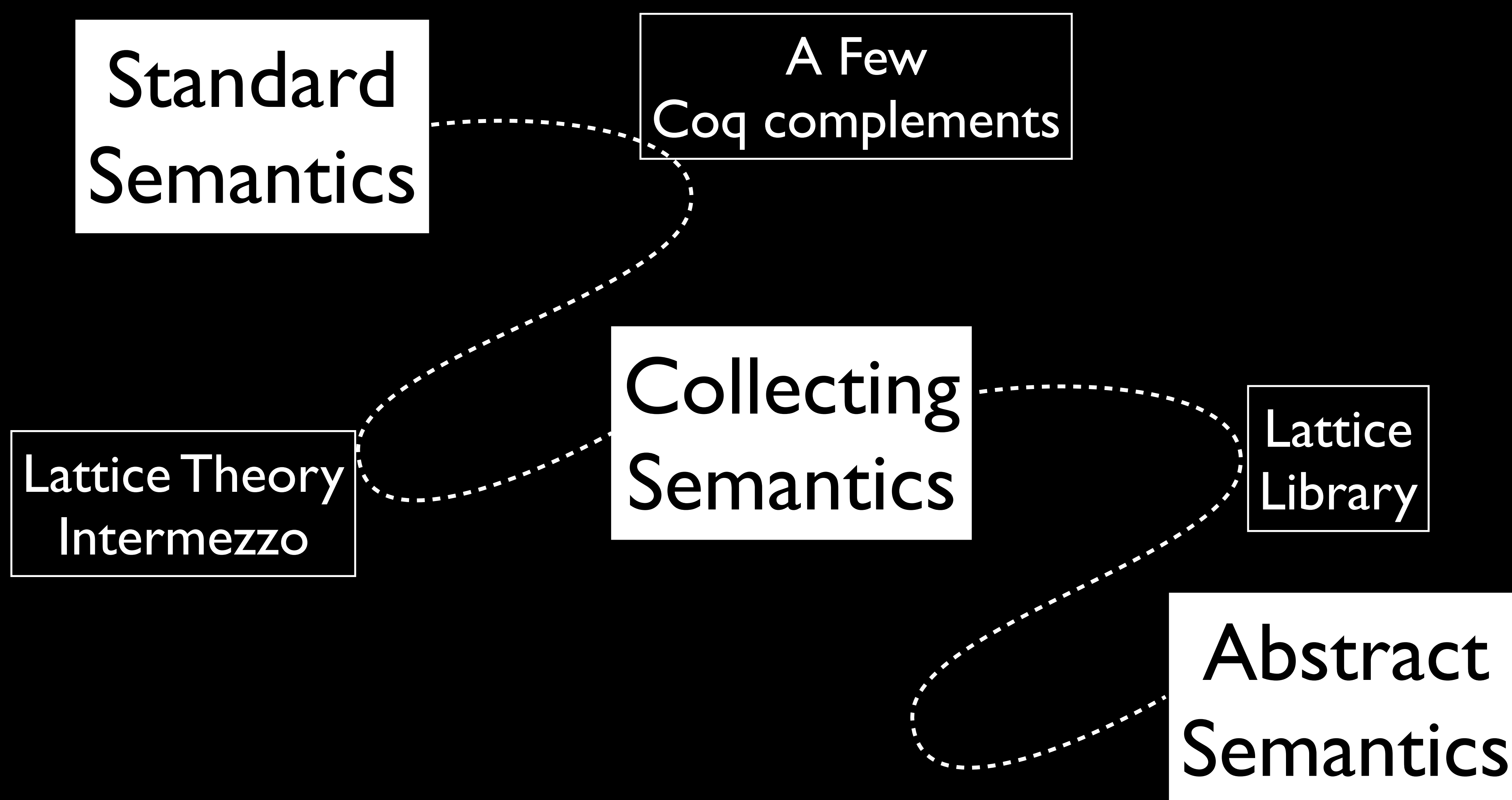
```
Instance PowerSetCL A : CompleteLattice.t 𝒫(A) := [...]
```

```
Instance PointwiseCL A L {CompleteLattice.t L} :
                         CompleteLattice.t (A → L) := [...]
```

```
Definition example (f:monotone (B → 𝒫(C)) (B → 𝒫(C))) :=
  lfp f.
```

The right complete lattice is automatically inferred

Roadmap


Standard Semantics
A Few Coq complements
Lattice Theory Intermezzo
Collecting Semantics
Lattice Library
Abstract Semantics

Collecting Semantics

- An important component in the Abstract Interpretation framework
- Mimics the behavior of the static analysis (fixpoint iteration)
- But still in the concrete domain
- Similar to a denotational semantics but operates on $\wp(State)$ instead of $State_\perp$

Collecting Semantics: Example

```
i = 0; k = 0;

while  k < 10  {

   i = 0 ;

  while  i < 9  {

     i = i + 2
  };

   k = k + 1
}
```

Collecting Semantics: Example

`i = 0; k = 0;`

`while [k < 10]`^{l_1}`{`

`   [i = 0]`^{l_2}`;`

`   while [i < 9]`^{l_3}`{`

`      [i = i + 2]`^{l_4}

`   };`

`   [k = k + 1]`^{l_5}

`}`$_{l_6}$

Collecting Semantics: Example

```
i = 0; k = 0;

while [k < 10]^l1 {

  [i = 0]^l2;

  while [i < 9]^l3 {

    [i = i + 2]^l4
  };

  [k = k + 1]^l5
}l6
```

$$l_1 \mapsto [0, 10] \times ([0, 10] \cap \text{Even})$$

$$l_2 \mapsto [0, 9] \times ([0, 10] \cap \text{Even})$$

$$l_3 \mapsto [0, 9] \times ([0, 10] \cap \text{Even})$$

$$l_4 \mapsto [0, 9] \times ([0, 8] \cap \text{Even})$$

$$l_5 \mapsto [0, 9] \times ([0, 10] \cap \text{Even})$$

$$l_6 \mapsto \{(10, 10)\}$$

Collecting Semantics

precondition

`Collect (i:stmt) (l:pp) :` $\mathcal{P}$`(env)` $\rightarrow$ `(pp` $\rightarrow \mathcal{P}$`(env))`

label after i

invariants on each reachable states during execution of i

Collecting Semantics

`Collect (i:stmt) (l:pp) : monotone` $(\mathcal{P}(\texttt{env}))$ $(\texttt{pp} \rightarrow \mathcal{P}(\texttt{env}))$

We generate only monotone operators

Collecting Semantics

$\texttt{Collect (i:stmt) (l:pp) : monotone } (\mathcal{P}(\texttt{env})) \ (\texttt{pp} \to \mathcal{P}(\texttt{env}))$

Final instanciation:

$\texttt{Collect p.(p\_stmt) p.(p\_end) } \top \texttt{ : } (\texttt{pp} \to \mathcal{P}(\texttt{env}))$

invariants on each reachable states

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                             monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i =>
    [...]




  | [...]
end.
```

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                            monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i =>
    [...]

  | [...]
end.
```

map substitution + union

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                              monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i =>
    [...]

  | [...]
end.
```

map substitution + union

Strongest post-condition
assignment transformer

Collecting Semantics

Strongest post-condition of basic instructions

```
Definition assign (x:var) (e:expr) (E:𝒫(env)) : 𝒫(env) :=
  fun ρ => ∃ρ', ∃n, E ρ' ∧ sem_expr prog ρ' e n ∧ subst ρ' x n ρ.

Definition assert (t:test) (E:𝒫(env)) : 𝒫(env) :=
  fun ρ => E ρ ∧ sem_test prog ρ t true.
```

Cumulative substitution

```
Definition Esubst {A} (f:pp → 𝒫(A)) (k:pp) (v:𝒫(A)) : pp → 𝒫(A) :=
  fun k' => if pp_eq k' k then (f k) ⊔ v else f k'.
Notation "f +[ x ↦ v ]" := (Esubst f x v) (at level 100).
```

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                              monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i =>
    [...]




  | [...]
end.
```

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                            monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
   | Assign p x e =>
     Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

   | While p t i =>
     Mono (fun Env =>
          let I:𝒫(env) := lfp    ?                          in
            (Collect i p (assert t I))
                  +[p ↦ I] +[l ↦ assert (Not t) I]) _

   | [...]
end.
```

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                            monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  |
```

Fixpoint equation: `I == Env ⊔ (Collect i p (assert t I) p)`

```
  | While p t i =>
    Mono (fun Env =>
          let I:𝒫(env) := lfp        ?                        in
            (Collect i p (assert t I))
                 +[p ↦ I] +[l ↦ assert (Not t) I]) _

  | [...]
end.
```

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                             monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  |
  | While p t i =>
    Mono (fun Env =>
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I))
                 +[p ↦ I] +[l ↦ assert (Not t) I]) _

  | [...]
end.
```

Fixpoint equation: `I == Env ⊔ (Collect i p (assert t I) p)`

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                          monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
   | Assign p x e =>
     Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

   | While p t i =>
     Mono (fun Env =>
           let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
             (Collect i p (assert t I))
                  +[p ↦ I] +[l ↦ assert (Not t) I]) _

   | [...]
end.
```

must be monotone

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                          monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
   | Assign p x e =>
     Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

   | While p t i =>
     Mono (fun Env =>
           let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
             (Collect i p (assert t I))
                  +[p ↦ I] +[l ↦ assert (Not t) I]) _

   | [...]
end.
```

proof obligation

must be monotone

proof obligation

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                           monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
   | Assign p x e =>
     Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

   | While p t i =>
     Mono (fun Env =>
           let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
             (Collect i p (assert t I))
                  +[p ↦ I] +[l ↦ assert (Not t) I]) _

   | [...]
end.
```

proof obligation

must be monotone

proof obligation

Proof obligations are generated by the `Program` mechanism and then automatically discharged by a custom tactic for monotonicity proofs

Collecting Semantics

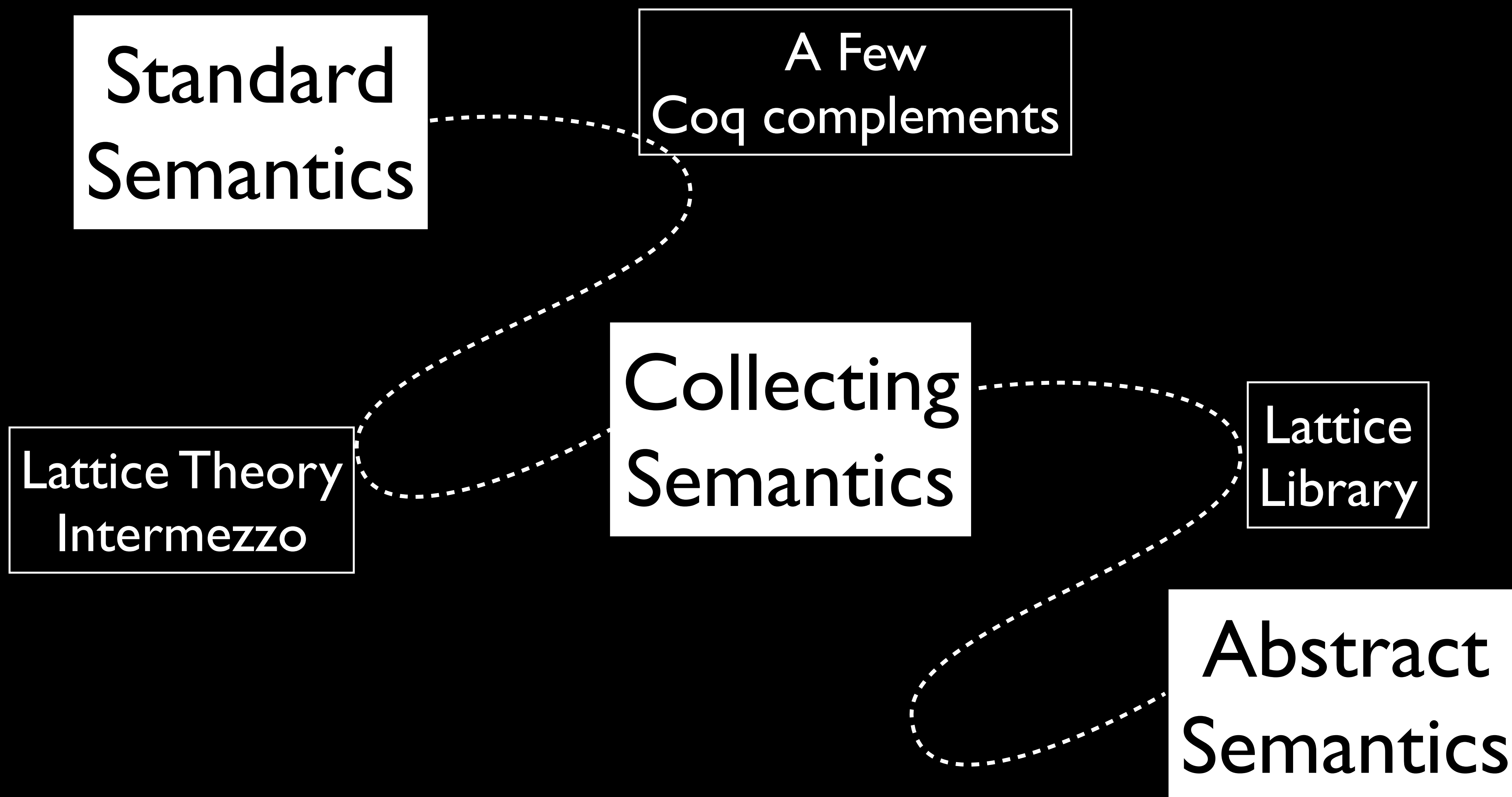
```
Definition reachable_collect (p:program) (s:pp*env) : Prop :=
  let (k,env) := s in
    Collect p p.(p_instr) p.(p_end) (⊤) k env.

Theorem reachable_sos_implies_reachable_collect :
  ∀ p, reachable_sos p ⊆ reachable_collect p.
```

Collecting Semantics

```
Definition reachable_collect (p:program) (s:pp*env) : Prop :=
  let (k,env) := s in
    Collect p p.(p_instr) p.(p_end) (⊤) k env.

Theorem reachable_sos_implies_reachable_collect :
  ∀ p, reachable_sos p ⊆ reachable_collect p.
```

This is the most difficult proof of this work. It is sometimes just skipped in the AI literature because people *start* from a collecting semantics.

Roadmap


Standard Semantics
A Few Coq complements
Lattice Theory Intermezzo
Collecting Semantics
Lattice Library
Abstract Semantics

Abstract Lattices

- Nothing can be extracted from the collecting semantics
  - it operates on Prop
  - that’s why we were able to *program* the *not-so-constructive* lfp operator in Coq
- The abstract semantics will not computes on $\texttt{(pp} \rightarrow \mathcal{P}\texttt{(env))}$ but on an *abstract lattice* $\texttt{A}^\sharp$

Abstract Lattice

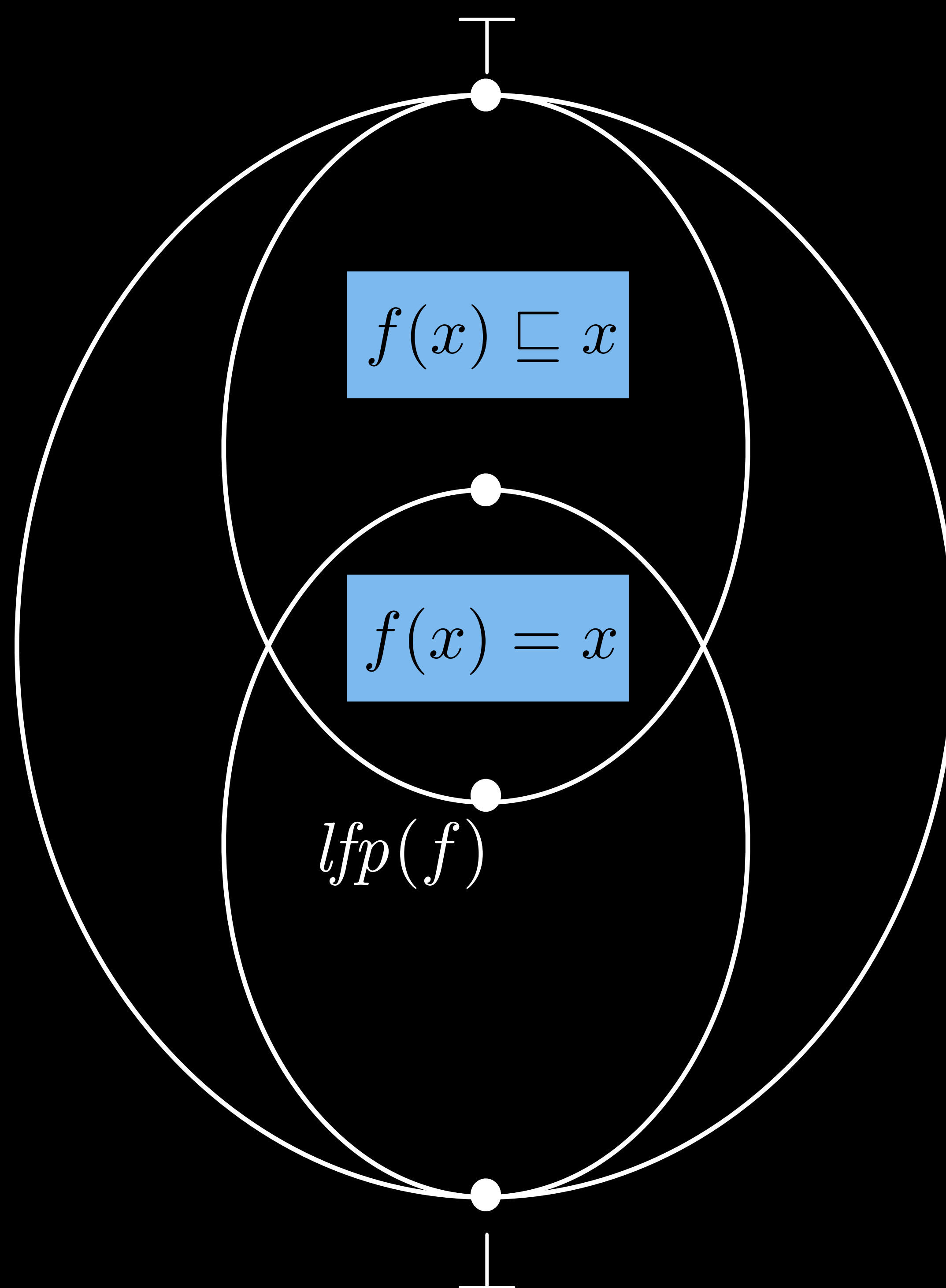
Abstract lattices are formalized with type classes

$$\texttt{AbLattice.t} : \sqsubseteq^\sharp, \sqcap^\sharp, \sqcup^\sharp, \perp^\sharp + \text{widening/narrowing}$$

Each abstract lattice is equipped with a post-fixpoint solver

```
Definition approx_lfp :
    ∀ {t} {L:AbLattice.t t}, (t → t) → t := [...]

Lemma approx_lfp_is_postfixpoint :
    ∀ t (L:AbLattice.t t) (f:t → t),
         f (approx_lfp f) ⊑♯ (approx_lfp f).
```

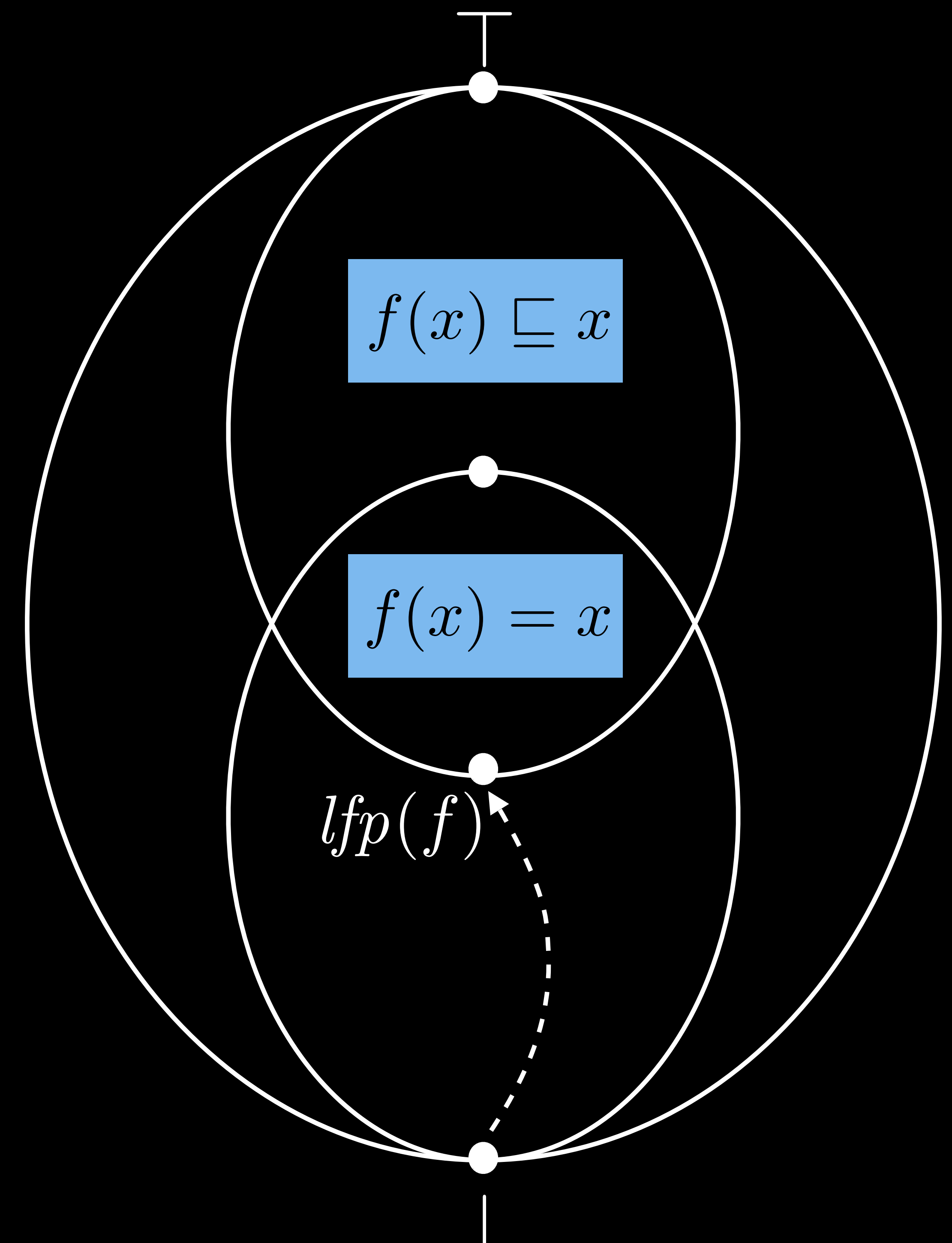
Fixpoint approximation with widening/narrowing



Fixpoint approximation with widening/narrowing

Standard Kleene fixed-point theorem

- too slow for big lattices (or just infinite)



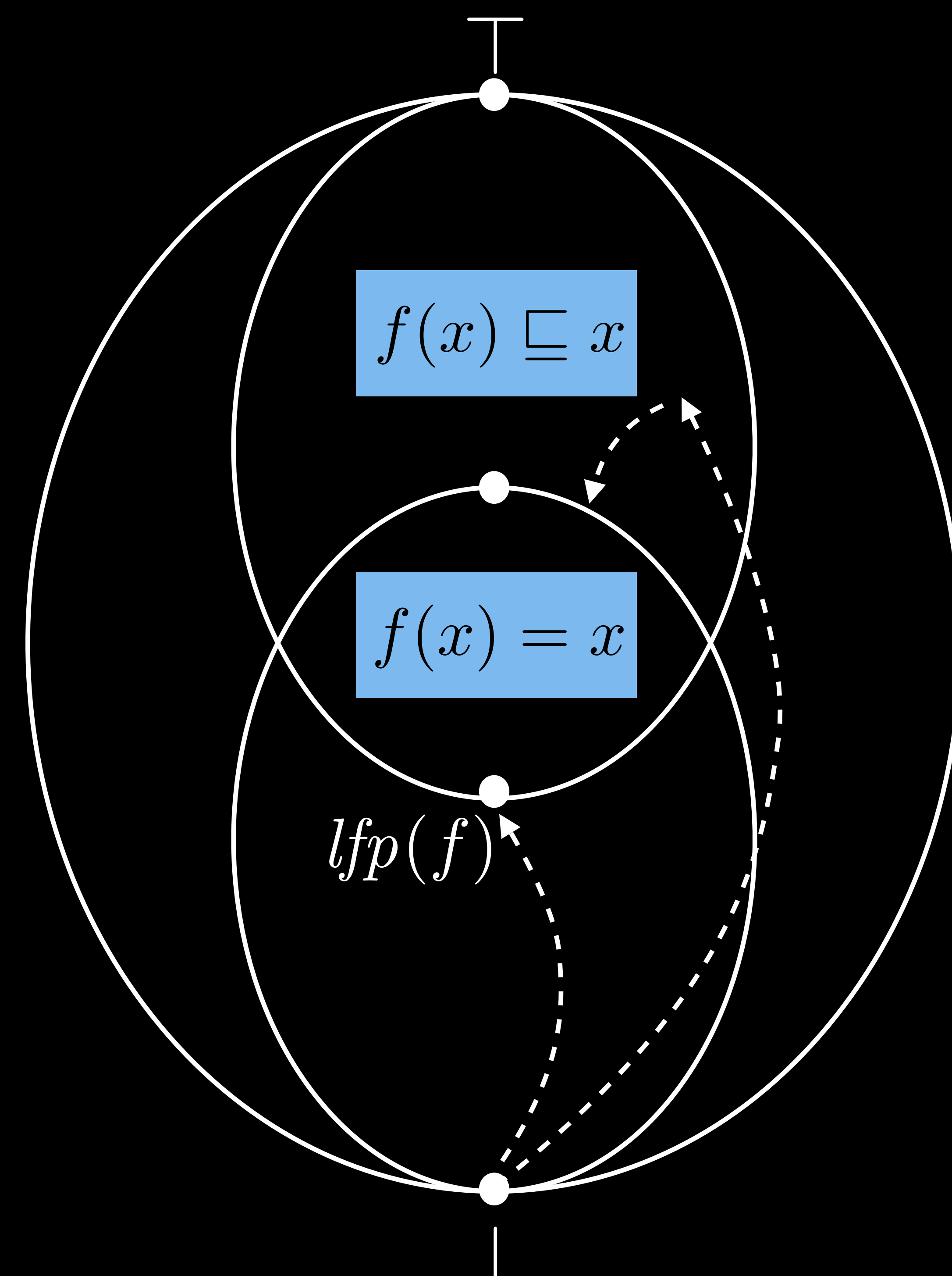
Fixpoint approximation with widening/narrowing

Standard Kleene fixed-point theorem

- too slow for big lattices (or just infinite)

Fixpoint approximation by widening/ narrowing

- over-approximates the lfp.
- requires different termination proofs than ascending chain condition
- on fixpoint equations, iteration order matters a lot !



Abstract Lattice

A library[1] is provided to build complex lattice objects with various functors for products, sums, lists and arrays.

Examples:

```
Instance ProdLattice
  t1 t2 {L1:AbLattice.t t1} {L2:AbLattice.t t2}:
  AbLattice.t (t1*t2) := [...]


Instance ArrayLattice t {L:AbLattice.t t}:
  AbLattice.t (array t) := [...]
```

[1] Adapted from our previous work: *Building certified static analysers by modular construction of well-founded lattices*. FICS'08

Abstract Lattice

A library[1] is provided to build complex lattice objects with various functors for products, sums, lists and arrays.

Examples:

```
Instance ProdLattice
  t1 t2 {L1:AbLattice.t t1} {L2:AbLattice.t t2}:
  AbLattice.t (t1*t2) := [...]
```

Contains a difficult termination proof !

```
Instance ArrayLattice t {L:AbLattice.t t}:
  AbLattice.t (array t) := [...]
```

---

[1] Adapted from our previous work: *Building certified static analysers by modular construction of well-founded lattices*. FICS'08

Abstract Lattice

A library[1] is provided to build complex lattice objects with various functors for products, sums, lists and arrays.

Examples:

```
Instance ProdLattice
  t1 t2 {L1:AbLattice.t t1} {L2:AbLattice.t t2}:
  AbLattice.t (t1*t2) := [...]
```

Contains a difficult termination proof !

```
Instance ArrayLattice t {L:AbLattice.t t}:
  AbLattice.t (array t) := [...]
```

Functional maps

[1] Adapted from our previous work: *Building certified static analysers by modular construction of well-founded lattices*. FICS'08

Concretizations

We connect concrete and abstract lattices with concretization functions (a simplified form of the Galois-connection standard)

```
Module Gamma.
Class t a A {L:Lattice.t a} {AL:AbLattice.t A} : Type := Make {
  γ : A → a;
  γ_monotone : ∀ N1 N2:A, N1 ⊑♯ N2 → γ N1 ⊑ γ N2;
  γ_meet_morph : ∀ N1 N2:A, γ N1 ⊓ γ N2 ⊑ γ (N1 ⊓♯ N2)
}.
End Gamma.
```

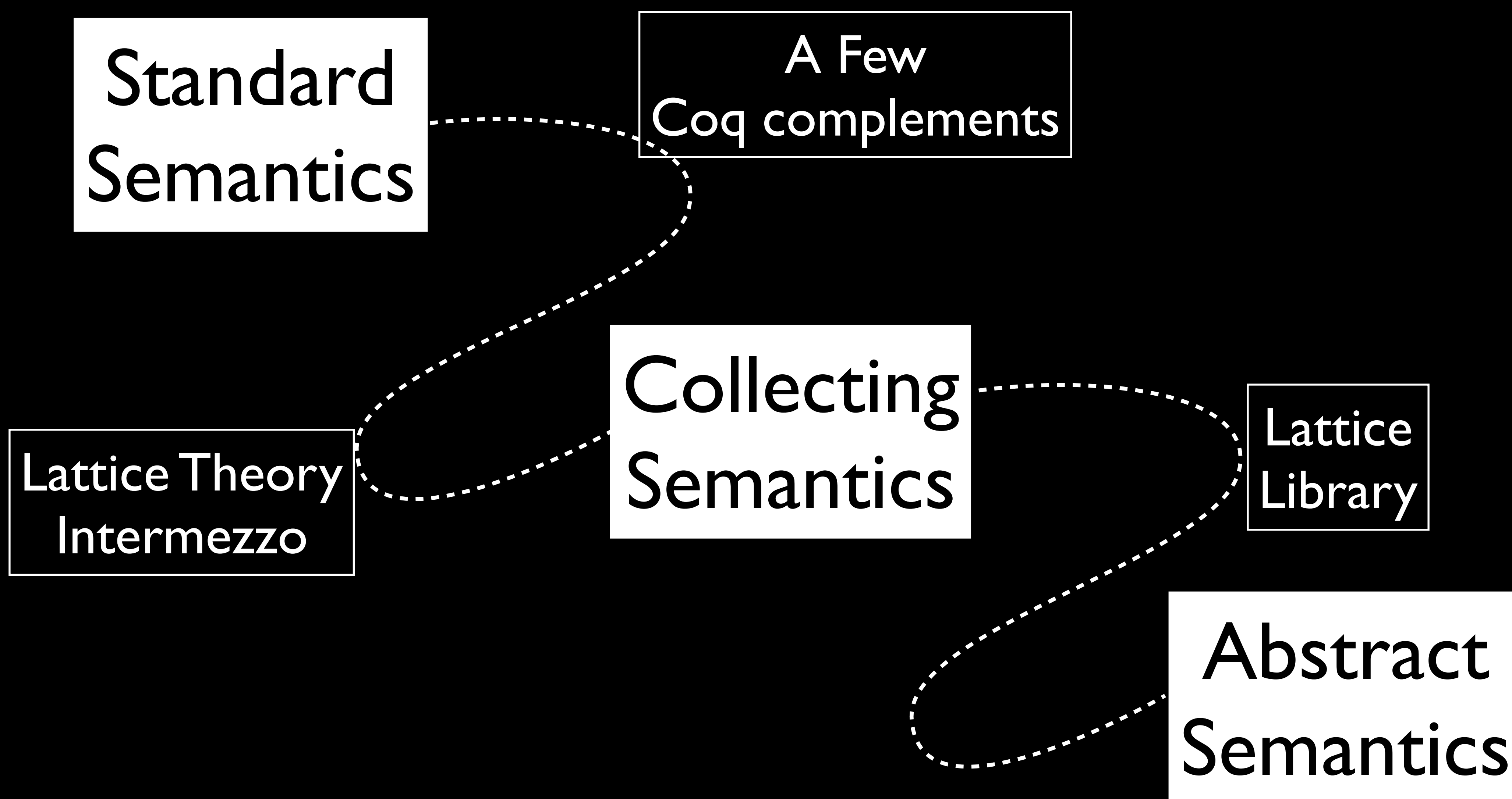
Concretization functors

The Lattice library can be lifted to a concretization library

```
Instance GammaFunc a A {G:Gamma.t a A} :
                 Gamma.t (word → a) (array A).
```

In our previous works, we relied on modules but concretizations need to be first-class citizens to be useful.

Roadmap


Standard Semantics
A Few Coq complements
Lattice Theory Intermezzo
Collecting Semantics
Lattice Library
Abstract Semantics

Abstract Algebra

The analyzer is parameterized wrt. to an environment abstraction.

The development provides several non-relational instantiations.

Abstract Algebra

The analyzer is parameterized wrt. to an environment abstraction.

The development provides several non-relational instantiations.

```
i = 0; k = 0;
```
$\mathtt{k} \in [0,10] \quad \mathtt{i} \in [0,10]$
```
while k < 10 {
```
$\mathtt{k} \in [0,9] \quad \mathtt{i} \in [0,10]$
```
  i = 0;
```
$\mathtt{k} \in [0,9] \quad \mathtt{i} \in [0,10]$
```
  while i < 9 {
```
$\mathtt{k} \in [0,9] \quad \mathtt{i} \in [0,8]$
```
    i = i + 2
  };
```
$\mathtt{k} \in [0,9] \quad \mathtt{i} \in [9,10]$
```
  k = k + 1
}
```
$\mathtt{k} \in [10,10] \quad \mathtt{i} \in [0,10]$

interval

Abstract Algebra

The analyzer is parameterized wrt. to an environment abstraction.

The development provides several non-relational instantiations.

```
i = 0; k = 0;
          k ∈ [0, 10]   i ∈ [0, 10]
while k < 10 {
          k ∈ [0, 9]   i ∈ [0, 10]
  i = 0;
          k ∈ [0, 9]   i ∈ [0, 10]
  while i < 9 {
          k ∈ [0, 9]   i ∈ [0, 8]
    i = i + 2
  };
          k ∈ [0, 9]   i ∈ [9, 10]
  k = k + 1
}
          k ∈ [10, 10]   i ∈ [0, 10]
```

interval

```
i = 0; k = 0;
          k ≥ 0   i ≥ 0
while k < 10 {
          k ≥ 0   i ≥ 0
  i = 0;
          k ≥ 0   i ≥ 0
  while i < 9 {
          k ≥ 0   i ≥ 0
    i = i + 2
  };
          k ≥ 0   i > 0
  k = k + 1
}
          k > 0   i ≥ 0
```

sign

Abstract Algebra

The analyzer is parameterized wrt. to an environment abstraction.

The development provides several non-relational instantiations.

`i = 0; k = 0;`
$\mathrm{k} \in [0, 10] \quad \mathrm{i} \in [0, 10]$
`while k < 10 {`
$\mathrm{k} \in [0, 9] \quad \mathrm{i} \in [0, 10]$
`  i = 0;`
$\mathrm{k} \in [0, 9] \quad \mathrm{i} \in [0, 10]$
`  while i < 9 {`
$\mathrm{k} \in [0, 9] \quad \mathrm{i} \in [0, 8]$
`    i = i + 2`
`  };`
$\mathrm{k} \in [0, 9] \quad \mathrm{i} \in [9, 10]$
`  k = k + 1`
`}`
$\mathrm{k} \in [10, 10] \quad \mathrm{i} \in [0, 10]$

interval

`i = 0; k = 0;`
$\mathrm{k} \geq 0 \quad \mathrm{i} \geq 0$
`while k < 10 {`
$\mathrm{k} \geq 0 \quad \mathrm{i} \geq 0$
`  i = 0;`
$\mathrm{k} \geq 0 \quad \mathrm{i} \geq 0$
`  while i < 9 {`
$\mathrm{k} \geq 0 \quad \mathrm{i} \geq 0$
`    i = i + 2`
`  };`
$\mathrm{k} \geq 0 \quad \mathrm{i} > 0$
`  k = k + 1`
`}`
$\mathrm{k} > 0 \quad \mathrm{i} \geq 0$

sign

`i = 0; k = 0;`
$\mathrm{i} \equiv 0 \bmod 2$
`while k < 10 {`
$\mathrm{i} \equiv 0 \bmod 2$
`  i = 0;`
$\mathrm{i} \equiv 0 \bmod 2$
`  while i < 9 {`
$\mathrm{i} \equiv 0 \bmod 2$
`    i = i + 2`
`  };`
$\mathrm{i} \equiv 0 \bmod 2$
`  k = k + 1`
`}`
$\mathrm{i} \equiv 0 \bmod 2$

parity

Abstract Semantics

```
Section prog.
Variable (t : Type) (L : AbLattice.t t)
         (prog : program) (Ab : AbEnv.t L prog).

Fixpoint AbSem (i:instr) (l:pp) : t → array t :=
match i with
| Assign p x e =>
  fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
| While p t i => fun Env =>
  let I := approx_lfp
             (fun X => Env ⊔♯
                        (get (AbSem i p (Ab.assert t X)) p)) in
  (AbSem i p (Ab.assert t I))
    +[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
| [...]
end.
```

Abstract Semantics

```
Section prog.
Variable (t : Type) (L : AbLattice.t t)
         (prog : program) (Ab : AbEnv.t L prog).

Fixpoint AbSem (i:instr) (l                         :=
match i with
| Assign p x e =>
  fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
| While p t i => fun Env =>
  let I := approx_lfp
             (fun X => Env ⊔♯
                         (get (AbSem i p (Ab.assert t X)) p)) in
  (AbSem i p (Ab.assert t I))
    +[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
| [...]
end.
```

Abstract counterpart of
concrete operations

Abstract Semantics

```
Section prog.
Variable (t : Type) (L : AbLattice.t t)
         (prog : program) (Ab : AbEnv.t L prog).

Fixpoint AbSem (i:instr) (l                    t :=
match i with
| Assign p x e =>
  fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
| While p t i => fun Env =>
  let I := approx_lfp
             (fun X => Env ⊔♯
                       (get (AbSem i p (Ab.assert t X)) p)) in
   (AbSem i p (Ab.assert t I))
     +[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
| [...]
end.
```

Abstract counterpart of concrete operations

Fixpoint approximation instead of least fixpoint computation

Connecting Concrete and Abstract Semantics

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

Connecting Concrete and Abstract Semantics

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

Soundness proof between abstract and collecting semantics

Type Classes to the rescue

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

Need 4 minutes after

Type Classes to the rescue

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

canonical order on pp $\rightarrow \mathcal{P}$(env)

Need 4 minutes
after

Type Classes to the rescue

concretization on $\texttt{pp} \rightarrow \mathcal{P}(\texttt{env})$

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

canonical order on $\texttt{pp} \rightarrow \mathcal{P}(\texttt{env})$

Need 4 minutes after

Type Classes to the rescue

concretization on $\texttt{pp} \to \mathcal{P}(\texttt{env})$

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

concretization on $\mathcal{P}(\texttt{env})$

canonical order on $\texttt{pp} \to \mathcal{P}(\texttt{env})$

Need 4 minutes after

Type Classes to the rescue

concretization on pp $\to \mathcal{P}$(env)

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

concretization on $\mathcal{P}$(env)

canonical order on pp $\to \mathcal{P}$(env)

Without Type Classes

Need 4 minutes after

```
Theorem AbSem_correct : ∀ i l_end Env,
  (PointwisePoset (PowerSetPoset env)).(Poset.o
  (Collect prog i l_end (AbEnv.(AbEnv.gamma) Env))
  (FuncLattice.Gamma AbEnv.(AbEnv.gamma) (AbSem i l_end Env)).
```

Type Classes to the rescue

concretization on **pp** $\to \mathcal{P}$**(env)**

Theorem AbSem_correct : ∀ i l_end Env,
Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).

concretization on $\mathcal{P}$**(env)**

canonical order on **pp** $\to \mathcal{P}$**(env)**

Without
Type
Classes

Need 4 minutes
after

canonical order on **pp** $\to \mathcal{P}$**(env)**

Theorem AbSem_correct : ∀ i l_end Env,
(PointwisePoset (PowerSetPoset env)).(Poset.o
(Collect prog i l_end (AbEnv.(AbEnv.gamma) Env))
(FuncLattice.Gamma AbEnv.(AbEnv.gamma) (AbSem i l_end Env)).

Type Classes to the rescue

concretization on **pp** $\to \mathcal{P}$**(env)**

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

concretization on $\mathcal{P}$**(env)**

canonical order on **pp** $\to \mathcal{P}$**(env)**

Without Type Classes

Need 4 minutes after

canonical order on **pp** $\to \mathcal{P}$**(env)**

```
Theorem AbSem_correct : ∀ i l_end Env,
  (PointwisePoset (PowerSetPoset env)).(Poset.o
  (Collect prog i l_end (AbEnv.(AbEnv.gamma) Env))
  (FuncLattice.Gamma AbEnv.(AbEnv.gamma) (AbSem i l_end Env)).
```

concretization on **pp** $\to \mathcal{P}$**(env)**

Type Classes to the rescue

concretization on pp $\to \mathcal{P}$(env)

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

concretization on $\mathcal{P}$(env)

canonical order on pp $\to \mathcal{P}$(env)

Without Type Classes

Need 4 minutes after

canonical order on pp $\to \mathcal{P}$(env)

```
Theorem AbSem_correct : ∀ i l_end Env,
  (PointwisePoset (PowerSetPoset env)).(Poset.o
  (Collect prog i l_end (AbEnv.(AbEnv.gamma) Env))
  (FuncLattice.Gamma AbEnv.(AbEnv.gamma) (AbSem i l_end Env)).
```

concretization on pp $\to \mathcal{P}$(env)

concretization on $\mathcal{P}$(env)

Connecting Concrete and Abstract Semantics

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

Connecting Concrete and Abstract Semantics

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

The proof is easy because the two semantics are very similar

Abstract Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp): monotone (𝒫(env)) (pp → 𝒫(env)) :=
  match i with
    | Assign p x e =>
      Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
    | While p t i =>
      Mono (fun Env =>
              let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
              (Collect i p (assert t I)) +[p ↦ I] +[l ↦ assert (Not t) I]) _
    [...]
  end.
```

The proof is easy because the two semantics are very similar

```
Fixpoint AbSem (i:instr) (l:pp) : t → array t :=
  match i with
  | Assign p x e =>
    fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
  | While p t i => fun Env =>
    let I := approx_lfp
                 (fun X => Env ⊔♯ (get (AbSem i p (Ab.assert t X)) p)) in
       (AbSem i p (Ab.assert t I)) +[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
  [...]
  end.
```

Abstract Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp): monotone (𝒫(env)) (pp → 𝒫(env)) :=
  match i with
    | Assign p x e =>
      Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
    | While p t i =>
      Mono (fun Env =>
              let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
              (Collect i p (assert t I)) +[p ↦ I] +[l ↦ assert (Not t) I]) _
    [...]
  end.
```

First Coq instance of the slogan

My abstract interpreter is correct by construction

```
Fixpoint AbSem (i:instr) (l:pp) : t → array t :=
  match i with
  | Assign p x e =>
    fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
  | While p t i => fun Env =>
    let I := approx_lfp
               (fun X => Env ⊔♯ (get (AbSem i p (Ab.assert t X)) p)) in
      (AbSem i p (Ab.assert t I)) +[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
  [...]
  end.
```

Final Theorem

```
Definition analyse : array t :=
  AbSem prog.(p_instr) prog.(p_end) (Ab.top).

Theorem analyse_correct : ∀ k env,
  reachable_sos prog (k,env) → γ (get analyse k) env.
```

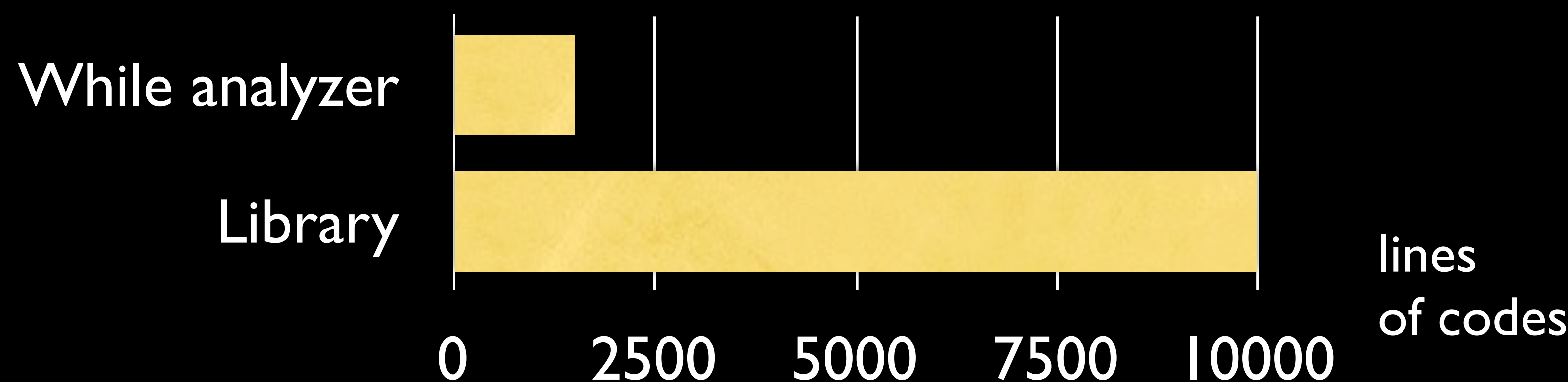
The function `analyse` can be extracted to real OCaml code

You can type-check, extract and run the analyser yourself !
http://www.irisa.fr/celtique/pichardie/teaching/digicosme13/

Conclusions

The first mechanized proof of an abstract interpreter based on a collecting semantics

- requires lattice theory components
- provides a reusable library



- the proof is more methodic and respectful with the AI theory than previous attempts

Perspectives

A first (small) step towards a certified Astrée-*like* analyser

- Ongoing project: scaling such an analyser to a C language
    - on top of the Compcert semantics
    - for a restricted C (no recursion, restricted use of pointers)

Abstraction Interpretation methodology

- would be nice to use more deeply the Galois connexion framework

- we prove soundness and termination: what about precision ?

Perspectives

A first (small) step towards a certified Astrée-*like* analyser

- Ongoing project: scaling such an analyser to a C language

  See next lecture

  - on top of the Compcert semantics
  - for a restricted C (no recursion, restricted use of pointers)

Abstraction Interpretation methodology

- would be nice to use more deeply the Galois connexion framework

- we prove soundness and termination: what about precision ?