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
A flavor of Abstract Interpretation

David Pichardie
Static analysis computes approximations \(^1\)

- \(P\) is safe w.r.t. \(\phi_1\) and the analyser proves it
  \[\lceil P \rceil \cap \phi_1 = \emptyset \quad \lceil P \rceil^\text{approx} \cap \phi_1 = \emptyset\]

- \(P\) is unsafe w.r.t. \(\phi_2\) and the analyser warns about it
  \[\lceil P \rceil \cap \phi_2 \neq \emptyset \quad \lceil P \rceil^\text{approx} \cap \phi_2 \neq \emptyset\]

- but \(P\) is safe w.r.t. \(\phi_3\) and the analyser can’t prove it (this is called a false alarm)
  \[\lceil P \rceil \cap \phi_3 = \emptyset \quad \lceil P \rceil^\text{approx} \cap \phi_3 \neq \emptyset\]

\[\lceil P \rceil: \text{concrete semantics (e.g. set of reachable states)} \quad \text{(not computable)}\]
\[\phi_1, \phi_2, \phi_3: \text{erroneous/dangerous set of states} \quad \text{(computable)}\]
\[\lceil P \rceil^\text{approx}: \text{analyser result (here over-approximation)} \quad \text{(computable)}\]

1. see [http://www.astree.ens.fr/IntroAbsInt.html](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 $(x, y)$ 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)

```plaintext
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 \((x, y)\) 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)

```plaintext
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 $(x, y)$ 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)

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

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

```plaintext
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 $(x, y)$ 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)

```plaintext
x = 0; y = 0;
{(0,0)}

while (x<6) {
  if (?) {
    
    y = y+2;
    
    
  } else {
    
    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 \((x, y)\) 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)

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

```plaintext
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 $(x, y)$ 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)

```plaintext
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 $(x, y)$ 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)

```plaintext
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 \((x, y)\) 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)

```plaintext
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 $(x, y)$ 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)

```plaintext
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

\[ P ::= x \in C 0 \land y \in C 0 \]
\[ C ::= < | \leq | = | > | \geq \]

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

\[ x = 0; \ y = 0; \]
\[ x = 0 \land y = 0 \]
\[ \textbf{while} \ (x < 6) \{ \]
\[ \textbf{if} \ (?) \{ \]
\[ y = y + 2; \]
\[ \} \]
\[ x = x + 1; \]
\[ \} \]
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

\[
P ::= x \in C 0 \land y \in C 0
\]

\[
C ::= < \mid \leq \mid = \mid > \mid \geq
\]

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

\[
x = 0; \ y = 0;
\]

\[
x = 0 \land y = 0
\]

\[
\text{while (x<6)} \{
\]

\[
\text{if (?) \{}
\]

\[
x = 0 \land y = 0
\]

\[
y = y+2;
\]

\[
}\;
\]

\[
}\;
\]

\[
x = x+1;
\]

\[
\}
\]
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

\[ P ::= x \leq 0 \land y \leq 0 \]
\[ C ::= \leq | = | > | \geq \]

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

\[ x = 0; y = 0; \]
\[ x = 0 \land y = 0 \]
\[ \textbf{while} (x<6) \{ \]
\[ \quad \textbf{if} (?) \{ \]
\[ \quad \quad x = 0 \land y = 0 \]
\[ \quad \quad y = y+2; \]
\[ \quad \quad x = 0 \land y > 0 \quad \text{over-approximation!} \]
\[ \} \]
\[ x = x+1; \]
\[ \} \]
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

\[
P ::= x C 0 \land y C 0
\]
\[
C ::= < | \leq | = | > | \geq
\]

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

```plaintext
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 > 0
  };
    x = 0 \land y \geq 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

\[
P ::= x \in \mathbb{C} \land y \in \mathbb{C}
\]

\[
C ::= < | \leq | = | > | \geq
\]

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

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

\begin{align*}
x &= 0 \land y = 0 \\
\text{while} \ (x < 6) \{ \\
\text{if} \ (?) \{ \\
\ x &= 0 \land y = 0 \\
\ y &= y + 2; \\
\ x &= 0 \land y > 0 \\
\}; \\
\ x &= 0 \land y \geq 0 \\
\ x &= x + 1; \\
\ x &= x > 0 \land y \geq 0 \text{ over-approximation!}
\end{align*}
```
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**

\[
P ::= x C 0 \land y C 0
\]

\[
C ::= \prec | \preceq | \preceq | \succ | \succeq
\]

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

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

x \geq 0 \land y \geq 0

while (x<6) {
    if (?) {
        x = 0 \land y = 0
        y = y+2;
        x = 0 \land y > 0
    }
    x = x+1;
    x > 0 \land y \geq 0
}
```
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**

\[
P ::= x \in [0] \land y \in [0]
\]

\[
C ::= < | \leq | = | > | \geq
\]

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

```plaintext
x = 0; y = 0;
   x \geq 0 \land y \geq 0
while (x<6) {
   if (?) {
      x \geq 0 \land y \geq 0
      y = y+2;
      x = 0 \land y > 0
   };
      x = 0 \land y \geq 0
   x = x+1;
      x > 0 \land y \geq 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

\[ P ::= x \in C 0 \land y \in C 0 \]
\[ C ::= < | \leq | = | > | \geq \]

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

\[
x = 0; \ y = 0; \\
x \geq 0 \land y \geq 0 \\
\text{while} \ (x < 6) \{ \\
\text{if} \ (?) \{ \\
x \geq 0 \land y \geq 0 \\
y = y + 2; \\
x \geq 0 \land y > 0 \\
\} \\
x = 0 \land y \geq 0 \\
x = x + 1; \\
x > 0 \land y \geq 0 \\
\}
\]
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

\[ P ::= x C 0 \land y C 0 \]
\[ C ::= < | \leq | = | > | \geq \]

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

```plaintext
x = 0; y = 0;
\quad x \geq 0 \land y \geq 0

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

x \geq 0 \land y \geq 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**

```plaintext
P ::= x \in C \{0\} \land y \in C \{0\}
C ::= < | \leq | = | > | \geq
```

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

```plaintext
x = 0; y = 0;
x \geq 0 \land y \geq 0
while (x<6) {
  if (?) {
    x \geq 0 \land y \geq 0
    y = y+2;
    x \geq 0 \land y > 0
  }
  x = x+1;
  x > 0 \land y \geq 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

\[ P ::= x \in \{0\} \land y \in \{0\} \]

\[ C ::= \lt | \leq | = | > | \geq \]

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

```plaintext
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 ≤ i ≤ T.length}
{1 ≤ i ≤ T.length - 1}
{1 ≤ i ≤ T.length - 1 ∧ -1 ≤ j ≤ i - 1}
{1 ≤ i ≤ T.length - 1 ∧ 0 ≤ j ≤ i - 1}
{1 ≤ i ≤ T.length - 1 ∧ -1 ≤ j ≤ i - 2}
{1 ≤ i ≤ T.length - 1 ∧ -1 ≤ j ≤ i - 1}
{2 ≤ i ≤ T.length + 1 ∧ -1 ≤ j ≤ 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.

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 $Q^2$.

\[
x = 0; \ y = 0;
\]

\[
\textbf{while} \ (x<6) \ { \ \\
\quad \textbf{if} \ (\ ?) \ { \ \\
\quad \quad \ y = y+2; \ \\
\quad \}; \ \\
\}
\]

\[
x = x+1; \ \\
\]
Polyhedral analysis

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

\[
\begin{align*}
x &= 0; \\ y &= 0; \\ \{x = 0 & \land y = 0\}
\end{align*}
\]

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

A flavor of Abstract Interpretation
Polyhedral analysis

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

\[
x = 0; \quad y = 0;
\{x = 0 \land y = 0\}
\]

\[\text{while } (x<6) \{\]
\[\text{if } (?) \{\]
\[
{x = 0 \land y = 0}\]
\[
y = y+2; \quad \{x = 0 \land y = 2\} \]
\[
\}; \quad \{x = 0 \land y = 0\} \cup \{x = 0 \land y = 2\}
\]
\[x = x+1; \]
\[\}

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

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

$$x = 0; \quad y = 0;$$

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

\[
\textbf{while} \ (x<6) \ {\{}
\quad \textbf{if} \ (?) \ {\{}
\quad \quad \{x = 0 \land y = 0\}
\quad \quad y = y+2;
\quad \quad \{x = 0 \land y = 2\}
\quad \};
\quad \{x = 0 \land 0 \leq y \leq 2\}
\}\]

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

$$x = x+1;$$

$$\}$$
Polyhedral analysis

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

\[
x = 0; \ y = 0;
\{ x = 0 \land y = 0 \}
\]

\[
\text{while} \ (x<6) \ {\{ \}
\text{if} \ (?) \ {\{ \}
\{ x = 0 \land y = 0 \}
\ y = y+2;
\{ x = 0 \land y = 2 \}
\}
\{ x = 0 \land 0 \leq y \leq 2 \}
\]

\[
x = x+1;
\{ x = 1 \land 0 \leq y \leq 2 \}
\}
\]
Polyhedral analysis

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

$$x = 0; \ y = 0;$$
$$\{x = 0 \land y = 0\} \uplus \{x = 1 \land 0 \leq y \leq 2\}$$

```c
while (x<6) {
    if (?) {
        \{x = 0 \land y = 0\}
        y = y+2;
        \{x = 0 \land y = 2\}
    };
    \{x = 0 \land 0 \leq y \leq 2\}
    x = x+1;
    \{x = 1 \land 0 \leq y \leq 2\}
}
```
Polyhedral analysis

State properties are over-approximated by convex polyhedra in $Q^2$.

$x = 0; y = 0; \{x \leq 1 \land 0 \leq y \leq 2x\}$

while (x<6) {
    if (?) {
        $y = y+2; \{x = 0 \land y = 2\}$
    }
    $x = x+1; \{x = 1 \land 0 \leq y \leq 2\}$
}
Polyhedral analysis

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

$x = 0; y = 0;
\{x \leq 1 \land 0 \leq y \leq 2x\}$

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

x = x+1;
\{x = 1 \land 0 \leq y \leq 2\}
}
Polyhedral analysis

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

\[
x = 0; \quad y = 0;
\{x \leq 1 \land 0 \leq y \leq 2x\}
\]

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

x = x+1;
\{x = 1 \land 0 \leq y \leq 2\}
}

A flavor of Abstract Interpretation
Polyhedral analysis

State properties are over-approximated by convex polyhedra in $Q^2$.

$$x = 0; \ y = 0; \ \{x \leq 1 \land 0 \leq y \leq 2x\}$$

**while** (x<6) {
  **if** (?) {
    $$\{x \leq 1 \land 0 \leq y \leq 2x\}$$
    y = y+2;
    $$\{x \leq 1 \land 2 \leq y \leq 2x + 2\}$$
  };
  $$\{x \leq 1 \land 0 \leq y \leq 2x\} \uplus \{x \leq 1 \land 2 \leq y \leq 2x + 2\}$$
  x = x+1;
  $$\{x = 1 \land 0 \leq y \leq 2\}$$
}
Polyhedral analysis

State properties are over-approximated by convex polyhedra in $Q^2$.

\[
x = 0; \quad y = 0;
\{x \leq 1 \land 0 \leq y \leq 2x\}
\]

\[
\textbf{while} \ (x < 6) \{ \\
\quad \textbf{if} \ (?) \{ \\
\quad \quad \{x \leq 1 \land 0 \leq y \leq 2x\}
\quad \quad y = y + 2;
\quad \quad \{x \leq 1 \land 2 \leq y \leq 2x + 2\}
\quad \};
\quad \{0 \leq x \leq 1 \land 0 \leq y \leq 2x + 2\}
\]

\[
x = x + 1;
\quad \{x = 1 \land 0 \leq y \leq 2\}
\]

A flavor of Abstract Interpretation
Polyhedral analysis

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

\[
x = 0; \ y = 0;
\{x \leq 1 \land 0 \leq y \leq 2x\}
\]

while (x<6) {
    if (?) {
        \{x \leq 1 \land 0 \leq y \leq 2x\}
        y = y+2;
        \{x \leq 1 \land 2 \leq y \leq 2x + 2\}
    };
    \{0 \leq x \leq 1 \land 0 \leq y \leq 2x + 2\}
    x = x+1;
    \{1 \leq x \leq 2 \land 0 \leq y \leq 2x\}
}
Polyhedral analysis

State properties are over-approximated by convex polyhedra in $Q^2$.

$x = 0; y = 0;
\{x \leq 1 \land 0 \leq y \leq 2x\}
\n\{x \leq 2 \land 0 \leq y \leq 2x\}

while (x<6) {
  if (?) {
    \{x \leq 1 \land 0 \leq y \leq 2x\}
    y = y+2;
    \{x \leq 1 \land 2 \leq y \leq 2x + 2\}
  };
  \{0 \leq x \leq 1 \land 0 \leq y \leq 2x + 2\}
  x = x+1;
  \{1 \leq x \leq 2 \land 0 \leq y \leq 2x\}
}
Polyhedral analysis

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

\[
x = 0; \ y = 0;
\{0 \leq y \leq 2x\}
\]

\[
\text{while} \ (x<6) \ {\{ \begin{align*}
\text{if} \ (?), & \{ x \leq 1 \ \& \ 0 \leq y \leq 2x \} \\
y = y+2; & \{ x \leq 1 \ \& \ 2 \leq y \leq 2x + 2 \}
\end{align*} \}};
\{0 \leq x \leq 1 \ \& \ 0 \leq y \leq 2x + 2 \}
\]

\[
x = x+1;
\{1 \leq x \leq 2 \ \& \ 0 \leq y \leq 2x\}
\]

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

State properties are over-approximated by convex polyhedra in $Q^2$.

$x = 0; \ y = 0; \ {0 \leq y \leq 2x}$

while (x<6) {
  if (?) {
    {0 \leq y \leq 2x \land x \leq 5}
    y = y+2;
    {2 \leq y \leq 2x + 2 \land x \leq 5}
  }
  {0 \leq y \leq 2x + 2 \land 0 \leq x \leq 5}
  x = x+1;
  {0 \leq y \leq 2x \land 1 \leq x \leq 6}
}
{0 \leq y \leq 2x \land 6 \leq x}$

By propagation we obtain a post-fixpoint
Polyhedral analysis

State properties are over-approximated by convex polyhedra in $Q^2$.

$x = 0; y = 0;$

$\{0 \leq y \leq 2x \land x \leq 6\}$

**while** $(x<6)$ {

**if** (?) {

$\{0 \leq y \leq 2x \land x \leq 5\}$

$y = y+2;$

$\{2 \leq y \leq 2x + 2 \land x \leq 5\}$

$\}$;

$\{0 \leq y \leq 2x + 2 \land 0 \leq x \leq 5\}$

$x = x+1;$

$\{0 \leq y \leq 2x \land 1 \leq x \leq 6\}$

$\}$

$\{0 \leq y \leq 2x \land 6 = x\}$

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

A more complex example.

\[
x = 0;\ y = A;
\{A \leq y \leq 2x + A \land x \leq N\}
\]

\begin{verbatim}
while (x<N) {
  if (?) {
  \{
  \{A \leq y \leq 2x + A \land x \leq N - 1\}
  y = y+2;
  \{A + 2 \leq y \leq 2x + A + 2 \land x \leq N - 1\}
  \};
  \{A \leq y \leq 2x + A + 2 \land 0 \leq x \leq N - 1\}
}
x = x+1;
\{A \leq y \leq 2x + A \land 1 \leq x \leq N\}
\}
\{A \leq y \leq 2x + A \land N = x\}
\end{verbatim}

The analysis accepts to replace some constants by parameters.
The four polyhedra operations

- $\cup \in \mathbb{P}_n \times \mathbb{P}_n \rightarrow \mathbb{P}_n$ : convex union
  - over-approximates the concrete union at junction points

- $\cap \in \mathbb{P}_n \times \mathbb{P}_n \rightarrow \mathbb{P}_n$ : intersection
  - over-approximates the concrete intersection after a conditional instruction

- $\llbracket x := e \rrbracket \in \mathbb{P}_n \rightarrow \mathbb{P}_n$ : affine transformation
  - over-approximates the assignment of a variable by a linear expression

- $\triangledown \in \mathbb{P}_n \times \mathbb{P}_n \rightarrow \mathbb{P}_n$ : widening
  - ensures (and accelerates) convergence of (post-)fixpoint iteration
  - includes heuristics to infer loop invariants

x = 0; y = 0;
$P_0 = [y := 0] [x := 0] (Q^2) \triangledown P_4$

while (x<6) {
  if (?) {
    $P_1 = P_0 \cap \{x < 6\}$
    y = y+2;
    $P_2 = [y := y + 2] (P_1)$
  }
  $P_3 = P_1 \cup P_2$
  x = x+1;
  $P_4 = [x := x + 1] (P_3)$
}

$P_5 = P_0 \cap \{x \geq 6\}$
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

\[ P = \begin{cases} (x, y) \in \mathbb{Q}^2 & \begin{align*} x &\geq -1 \\ x - y &\geq -3 \\ 2x + y &\geq -2 \\ x + 2y &\geq -4 \end{align*} \end{cases} \]

- by set of inequalities

\[ P = \left\{ \lambda_1 s_1 + \lambda_2 s_2 + \lambda_3 s_3 + \mu_1 r_1 + \mu_2 r_2 \in \mathbb{Q}^2 \mid \begin{align*} \lambda_1, \lambda_2, \lambda_3, \mu_1, \mu_2 &\in \mathbb{R}^+ \\ \lambda_1 + \lambda_2 + \lambda_3 &= 1 \end{align*} \right\} \]

- by set of generators

- 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
}.
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

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** \( env := \text{var} \rightarrow \mathbb{Z} \).

**Inductive** \( \text{config} := \text{Final} (\rho:env) | \text{Inter} (i:\text{instr}) (\rho:env) \).
Language Semantics

Semantic Domains

**Definition** $env := \text{var} \rightarrow \mathbb{Z}$.  

**Inductive** $config := \text{Final}\ (\rho : \text{env})\ |\ \text{Inter}\ (i : \text{instr})\ (\rho : \text{env})$.

Structural Operational Semantics

**Inductive** $sos\ (p : \text{program}) : (\text{instr} \times \text{env}) \rightarrow config \rightarrow \text{Prop} :=$

- $sos_{\text{assign}} : \forall l x e n\ \rho_1\ \rho_2,$
  - $sem_{\text{expr}}\ p\ \rho_1\ e\ n\ \rightarrow\ subst\ \rho_1\ x\ n\ \rho_2\ \rightarrow\ In\ x\ (\text{vars}\ p)\ \rightarrow$
  - $sos\ p\ (\text{Assign}\ l\ x\ e,\ \rho_1)\ (\text{Final}\ \rho_2)$

[...]
Language Semantics

Semantic Domains

**Definition** \( \text{env} := \text{var} \rightarrow \mathbb{Z}. \)

**Inductive** \( \text{config} := \text{Final } (\rho:\text{env}) \mid \text{Inter } (i:\text{instr}) (\rho:\text{env}). \)

Structural Operational Semantics

**Inductive** \( \text{sos } (p: \text{program}) : (\text{instr}\times\text{env}) \rightarrow \text{config} \rightarrow \text{Prop} := \)

\[
| \text{sos\_assign} : \forall \ l \ x \ e \ n \ \rho_1 \ \rho_2, \\
\text{sem\_expr } p \ \rho_1 \ e \ n \rightarrow \text{subst } \rho_1 \ x \ n \ \rho_2 \rightarrow \text{In } x \ (\text{vars } p) \rightarrow \\
\text{sos } p \ (\text{Assign } l \ x \ e, \rho_1) \ (\text{Final } \rho_2)
\]

\[\ldots\]

\[
\text{sem\_expr } p \ \rho_1 \ e \ n \ \rho_2 = \rho_1[x \mapsto n] \quad x \in (\text{vars } p)
\]

\[
\text{sos } p \ (\text{Assign } l \ x \ e, \rho_1) \ (\text{Final } \rho_2)
\]
Language Semantics

Semantic Domains

Definition \( \text{env} := \text{var} \rightarrow \mathbb{Z}. \)

Inductive \( \text{config} := \text{Final} \ (\rho: \text{env}) \ | \ \text{Inter} \ (i: \text{instr}) \ (\rho: \text{env}). \)

Structural Operational Semantics

Inductive \( \text{sos} \ (p: \text{program}) : \ (\text{instr} \times \text{env}) \rightarrow \text{config} \rightarrow \text{Prop} := \)

\( | \ \text{sos}\_\text{assign} : \ \forall \ l \ x \ e \ n \ \rho_1 \ \rho_2, \)
\( \ \text{sem}\_\text{expr} \ p \ \rho_1 \ e \ n \rightarrow \ \text{subst} \ \rho_1 \ x \ n \ \rho_2 \rightarrow \ \text{In} \ x \ (\text{vars} \ p) \rightarrow \)
\( \ \text{sos} \ p \ (\text{Assign} \ l \ x \ e, \rho_1) \ (\text{Final} \ \rho_2) \)

[...]

\[
\text{sem}\_\text{expr} \ p \ \rho_1 \ e \ n \ \rho_2 = \rho_1[x \mapsto n] \quad x \in (\text{vars} \ p)
\]

\( \ \text{sos} \ p \ (\text{Assign} \ l \ x \ e, \rho_1) (\text{Final} \ \rho_2) \)
Language Semantics

Semantic Domains

**Definition** \( \text{env} := \text{var} \rightarrow \mathbb{Z} \).

**Inductive** \( \text{config} := \text{Final} (\rho:\text{env}) \mid \text{Inter} (i:\text{instr}) (\rho:\text{env}). \)

Structural Operational Semantics

**Inductive** \( \text{sos} (p:\text{program}) : (\text{instr*env}) \rightarrow \text{config} \rightarrow \text{Prop} := \)

\( \mid \text{sos_assign} : \forall \ l \ x \ e \ n \ \rho_1 \ \rho_2, \)

\( \text{sem_expr} \ p \ \rho_1 \ e \ n \rightarrow \text{subst} \ \rho_1 \ x \ n \ \rho_2 \rightarrow \text{In} \ x \ (\text{vars} \ p) \rightarrow \)

\( \text{sos} \ p \ (\text{Assign} \ l \ x \ e, \rho_1) \ (\text{Final} \ \rho_2) \)

\(...\)

\( \overline{\text{sem_expr} \ p \ \rho_1 \ e \ n} \rho_2 = \rho_1[x \mapsto n] \quad x \in (\text{vars} \ p) \)

\( \text{sos} \ p \ (\text{Assign} \ l \ x \ e, \rho_1) \ (\text{Final} \ \rho_2) \)
Language Semantics

Semantic Domains

**Definition** \( \text{env} := \text{var} \rightarrow \mathbb{Z} \).

**Inductive** \( \text{config} := \text{Final} \ (\rho:\text{env}) \mid \text{Inter} \ (i:\text{instr}) \ (\rho:\text{env}) \).

Structural Operational Semantics

**Inductive** \( \text{sos} \ (p:\text{program}) : \ (\text{instr}*\text{env}) \rightarrow \text{config} \rightarrow \text{Prop} := \)

| sos_assign : \( \forall \ l \ x \ e \ n \ \rho_1 \ \rho_2, \)  
| \( \text{sem_expr} \ p \ \rho_1 \ e \ n \rightarrow \text{subst} \ \rho_1 \ x \ n \ \rho_2 \rightarrow \text{In} \ x \ (\text{vars} \ p) \rightarrow \)  
| \( \text{sos} \ p \ (\text{Assign} \ l \ x \ e, \rho_1) \ (\text{Final} \ \rho_2) \)  
[...]

\[
\text{sem_expr} \ p \ \rho_1 \ e \ n \quad \rho_2 = \rho_1[x \mapsto n] \quad x \in (\text{vars} \ p) \\
\text{sos} \ p \ (\text{Assign} \ l \ x \ e, \rho_1) \ (\text{Final} \ \rho_2)
\]
Language Semantics

Semantic Domains

**Definition** $\text{env} := \text{var} \rightarrow \mathbb{Z}$.

**Inductive** $\text{config} := \text{Final } (\rho : \text{env}) \mid \text{Inter } (i : \text{instr}) (\rho : \text{env})$.

Structural Operational Semantics

**Inductive** $\text{sos } (p : \text{program}) : (\text{instr}\ast \text{env}) \rightarrow \text{config} \rightarrow \text{Prop} :=$

$| \text{sos\_assign} : \forall l \ x \ e \ n \ \rho_1 \ \rho_2,$

$\text{sem\_expr } p \ \rho_1 \ e \ n \rightarrow \text{subst } \rho_1 \ x \ n \ \rho_2 \rightarrow \text{In } x (\text{vars } p) \rightarrow$

$\text{sos } p \ (\text{Assign } l \ x \ e, \rho_1) (\text{Final } \rho_2)$

[...]

$$\text{sem\_expr } p \ \rho_1 \ e \ n \ \rho_2 = \rho_1[x \mapsto n] \quad x \in (\text{vars } p)$$

$$\text{sos } p \ (\text{Assign } l \ x \ e, \rho_1) (\text{Final } \rho_2)$$
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

\[
\begin{align*}
\text{sem\_test } p \quad \rho \quad t \quad \text{true} \rightarrow \\
sos p \quad (\text{KAssert } t) \quad (\text{Assert } l \quad t, \rho) \quad (\text{Final } \rho) \\
| \text{sos\_if\_true} : \forall \ l \ t \ b1 \ b2 \ \rho, \\
\text{sem\_test } p \quad \rho \quad t \quad \text{true} \rightarrow \\
sos p \quad (\text{KAssert } t) \quad (\text{If } l \ t \ b1 \ b2, \rho) \quad (\text{Inter } b1 \ \rho) \\
| \text{sos\_if\_false} : \forall \ l \ t \ b1 \ b2 \ \rho, \\
\text{sem\_test } p \quad \rho \quad t \quad \text{false} \rightarrow \\
sos p \quad (\text{KAssert } (\text{Not } t)) \quad (\text{If } l \ t \ b1 \ b2, \rho) \quad (\text{Inter } b2 \ \rho) \\
| \text{sos\_while\_true} : \forall \ l \ t \ b \ \rho, \\
\text{sem\_test } p \quad \rho \quad t \quad \text{true} \rightarrow \\
sos p \quad (\text{KAssert } t) \quad (\text{While } l \ t \ b, \rho) \quad (\text{Inter } (\text{Seq } b \ (\text{While } l \ t \ b)) \ \rho) \\
| \text{sos\_while\_false} : \forall \ l \ t \ b \ \rho, \\
\text{sem\_test } p \quad \rho \quad t \quad \text{false} \rightarrow \\
sos p \quad (\text{KAssert } (\text{Not } t)) \quad (\text{While } l \ t \ b, \rho) \quad (\text{Final } \rho) \\
| \text{sos\_seq1} : \forall \ k \ i1 \ i2 \ \rho \ \rho', \\
sos p \quad k \quad (i1, \rho) \quad (\text{Final } \rho') \rightarrow \\
sos p \quad (\text{KSeq1 } i1 \ (\text{first } i2)) \quad (\text{Seq } i1 \ i2, \rho) \quad (\text{Inter } i2 \ \rho') \\
| \text{sos\_seq2} : \forall \ k \ i1 \ i1' \ i2 \ \rho \ \rho', \\
sos p \quad k \quad (i1, \rho) \quad (\text{Inter } i1' \ \rho') \rightarrow \\
sos p \quad (\text{KSeq2 } k) \quad (\text{Seq } i1 \ i2, \rho) \quad (\text{Inter } (\text{Seq } i1' \ i2) \ \rho').
\end{align*}
\]
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). |
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).

Language Semantics

Reachable states from any initial environment
Final Objective for today
The analyzer computes an abstract representation of the program semantics

\textbf{Definition} \ analyse : program \to\ abdom := [...]

Final Objective for today
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 $\mathcal{P}(\text{pp} \times \text{env})$

Definition $\gamma : \text{abdom} \rightarrow (\text{pp} \times \text{env} \rightarrow \text{Prop}) := [...]$
Final Objective for today

The analyzer computes an abstract representation of the program semantics

Definition analyse : program → abdomen := [...] 

Each abstract element is given a concretization in $\mathcal{P}(\mathbb{P} \times \mathbb{E})$

Definition $\gamma : \text{abdom} \rightarrow (\mathbb{P} \ast \mathbb{E} \rightarrow \text{Prop}) := [...]$

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

Theorem analyse_correct : $\forall \text{prog:program}, \text{reachable_sos prog \subseteq } \gamma (\text{analyse prog})$. 
Roadmap

Standard Semantics

Abstract Semantics
Roadmap

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
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

Collecting Semantics

Lattice Theory Intermezzo

Lattice Library

Abstract Semantics
A Few Coq Complements
Programming in Coq

Coq allows to mix

- data types
- programs
- predicates
- proofs

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

Record $t := \{$

$\quad A : Type;$

$\quad f1 : A \rightarrow A;$

$\quad f2 : A \rightarrow A;$

$\quad P : A \rightarrow A \rightarrow \text{Prop};$

$\quad \text{prop} : \forall a : A, P \ (f1 \ a) \ (f2 \ a)$

$\}$. 
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)
}.

\[\text{type } \_ = \text{Obj.t}\]

\[\text{type } t = \{\]
\[\text{  f1 : (}_\_\_\_ → _\_\_\_);\]
\[\text{  f2 : (}_\_\_\_ → _\_\_\_)}\]
\[\}\]

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 useful 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 instantiations
Definition sign_lattice : lattice sign := [...]
Given $s_1, \ldots, s_4$ of type sign, how to write $(s_1, s_2) \sqsubset (s_3, s_4)$?
Coq records for algebraic structures

Given $s_1, ..., s_4$ of type `sign`, how to write $(s_1, s_2) \sqsubseteq (s_3, s_4)$?

```coq
order (sign \times sign) _ (s_1, s_2) (s_3, s_4)
```

Coq records for algebraic structures

Given \( s_1, \ldots, s_4 \) of type \( \text{sign} \), how to write \( (s_1, s_2) \sqsubseteq (s_3, s_4) \) ?

\[
\text{order } (\text{sign} \times \text{sign}) \; _\sqsubseteq \; (s_1, s_2) \; (s_3, s_4)
\]

We need to fill the hole with a term of type \( \text{lattice} \; (\text{sign} \times \text{sign}) \)

\textbf{Definition} \( L : \; \text{lattice} \; (\text{sign} \times \text{sign}) \) :=
\[
\text{prod_lattice} \; \text{sign} \; \text{sign_lattice} \; \text{sign} \; \text{sign_lattice}.
\]
Coq records for algebraic structures

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

order \((\text{sign} \times \text{sign})\) \_ \_ (s1,s2) (s3,s4)

We need to fill the hole with a term of type lattice \((\text{sign} \times \text{sign})\)

\textbf{Definition} \textit{L} : lattice (sign\times 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 $s_1, ..., s_4$ of type `sign`, how to write $(s_1,s_2) \sqsubseteq (s_3,s_4)$?

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

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

**Definition**

```coq
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.

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

Notation overloading!

```coq
Notation "x \sqsubseteq 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**

**Definition** \( \text{lfp} \{ L \} \{ \text{CompleteLattice.t} \text{.t} L \} \ (f: \text{monotone} \ L \ L) : \text{CompleteLattice.meet} \ (\text{PostFix} \ f) \).
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)
}.
Knaster-Tarski Theorem

Definition $\text{lfp} \{L\} \{\text{CompleteLattice}.t \ L\}$ (f:monotone L L) :
CompleteLattice.meet (PostFix f).

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

Complete lattices on elements of type A
Monotone functions from L to L
Knaster-Tarski Theorem

Definition lfp \{L\} \{CompleteLattice.t L\} (f:\text{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 : \forall x, f x == x \rightarrow lfp f \sqsubseteq x. [...]
Lemma lfp_postfixpoint : f (lfp f) \sqsubseteq lfp f. [...]
Lemma lfp_least_postfixpoint : \forall x, f x \sqsubseteq x \rightarrow lfp f \sqsubseteq 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 : \forall x, f x == x \rightarrow lfp f \sqsubseteq x. [...]
  Lemma lfp_postfixpoint : f (lfp f) \sqsubseteq lfp f. [...]
  Lemma lfp_least_postfixpoint : \forall x, f x \sqsubseteq x \rightarrow lfp f \sqsubseteq x. [...]
End KnasterTarski.

Coq Type Classes = Record + Inference (super) capabilities
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 : \forall x, f x == x -> lfp f. [...] 
  Lemma lfp_postfixpoint : f (lfp f) == lfp f. [...] 
  Lemma lfp_least_postfixpoint : \forall x, f x == x -> lfp f. [...] 
End KnasterTarski.

Coq Type Classes = Record + Inference (super) capabilities

We declare this argument as implicit

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

Instance PowerSetCL A : CompleteLattice.t \mathcal{P}(A) := [...]

Instance PointwiseCL A L \{CompleteLattice.t L\} :
    CompleteLattice.t (A \to L) := [...]

Canonical Complete Lattices

Instance PowerSetCL A : CompleteLattice.t \( \mathcal{P}(A) := [...] \)

Instance PointwiseCL A L \{CompleteLattice.t L\} :
    CompleteLattice.t (A \rightarrow L) := [...]
Canonical Complete Lattices

Instance PowerSetCL A : CompleteLattice.t \( \mathcal{P}(A) := [...] \)

Notation for \((A \rightarrow \text{Prop})\)

Instance PointwiseCL A L \{CompleteLattice.t L\} :
CompleteLattice.t \((A \rightarrow L) := [...]\)

Set inclusion ordering
Canonical Complete Lattices

Instance PowerSetCL A : CompleteLattice.t \( \mathcal{P}(A) := [...] \)

Set inclusion ordering

Instance PointwiseCL A L \{CompleteLattice.t L\} :
  CompleteLattice.t (A \rightarrow L) := [...]
Canonical Complete Lattices

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

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

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

The right complete lattice is automatically inferred
Roadmap

Standard Semantics

A Few Coq complements

Collecting Semantics

Lattice Theory Intermezzo

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 $\wp(\text{State})$ instead of $\text{State}_\bot$
i = 0; k = 0;

while k < 10 {
    i = 0;
    while i < 9 {
        i = i + 2
    }
    k = k + 1
}
Collecting Semantics: Example

\[ \begin{align*}
  i &= 0; \quad k = 0; \\
  \text{while } [k < 10] &\{ \\
    [i = 0]; \\
    \text{while } [i < 9] &\{ \\
    [i = i + 2]; \\
    [k = k + 1] \}
  \}
\end{align*} \]
Collecting Semantics: Example

\[ i = 0; \quad k = 0; \]
\[ \text{while } [k < 10] \{ \]
\[ \quad [i = 0]; \quad l_1 \mapsto [0, 10] \times ([0, 10] \cap \text{Even}) \]
\[ \quad \text{while } [i < 9] \{ \]
\[ \quad \quad [i = i + 2] \quad l_2 \mapsto [0, 9] \times ([0, 10] \cap \text{Even}) \]
\[ \quad \quad [k = k + 1] \quad l_3 \mapsto [0, 9] \times ([0, 10] \cap \text{Even}) \]
\[ \quad \}; \quad l_4 \mapsto [0, 9] \times ([0, 8] \cap \text{Even}) \]
\[ \}; \quad l_5 \mapsto [0, 9] \times ([0, 10] \cap \text{Even}) \]
\[ l_6 \mapsto \{(10, 10)\} \]
Collecting Semantics

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

- **precondition**
- **label after i**
- **invariants on each reachable states during execution of i**
Collecting Semantics

Collect \((i: \text{stmt}) \ (l: \text{pp}) : \text{monotone} \ (P(\text{env})) \ (\text{pp} \rightarrow P(\text{env}))\)

We generate only monotone operators
Collecting Semantics

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

Final instanciation:

Collect \(p.(p\_stmt)\) \(p.(p\_end) \supset\) : \((pp \rightarrow \mathcal{P}(\text{env}))\)

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 \to \P(env)) :=
    match i with
      | Assign p x e =>
        Mono (fun Env => ⊥ +[p \mapsto Env] +[l \mapsto assign x e Env])
      | While p t i =>
          [...]
      | [...]
    end.
Collecting Semantics

Program Fixpoint Collect (i:stmt) (l:pp):

\[
\text{monotone} \quad \left( \mathcal{P}(\text{env}) \right) \quad (\text{pp} \to \mathcal{P}(\text{env})) := \\
\text{match} \quad i \quad \text{with} \\
| \text{Assign} \ p \ x \ e \Rightarrow \\
\quad \text{Mono} \quad \left( \text{fun} \quad \text{Env} \Rightarrow \bot + [p \mapsto \text{Env}] + [l \mapsto \text{assign} \ x \ e \ \text{Env}] \right) \\
| \text{While} \ p \ t \ i \Rightarrow \\
\quad [...]
\]

end.

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 \(\rightarrow \) \(P(env)\)) :=
match i with
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p \(\mapsto\) Env] +[l \(\mapsto\) assign x e Env]) _
  | While p t i =>
    [...]_
  | [...]_
end.
Collecting Semantics

Program Fixpoint Collect (i:stmt) (l:pp):

  monotone (\(P(env)\)) (pp \(\rightarrow\) \(P(env)\)) :=

match i with

  | Assign p x e =>
    Mono (fun Env => \bot +[p \rightarrow Env] +[l \rightarrow assign x e Env])

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

  | [...] end.
Program Fixpoint Collect (i:stmt) (l:pp):
    monotone (P(env)) (pp \rightarrow P(env)) :=
    match i with
    | Assign p x e =>
        Mono (fun Env =>
            let I := P(env) := lfp (iter Env (Collect i p) t p)
            in
            (Collect i p (assert t I))
        )
    | While p t i =>
        Mono (fun Env =>
            let I := P(env) := lfp (? in
                (Collect i p (assert t I))
            )
            in
           +[p \mapsto I] +[l \mapsto 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 + l] + [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.
Collecting Semantics

Program Fixpoint Collect (i:stmt) (l:pp):

\[
\text{monotone } (\mathcal{P}(\text{env})) \ (\text{pp} \rightarrow \mathcal{P}(\text{env})) :=
\]

match i with

| Assign p x e =>
    Mono (fun Env => \bot +[p \mapsto Env] +[l \mapsto \text{assign x e Env}])
    end

| While p t i =>
    Mono (fun Env =>
        let I:\mathcal{P}(\text{env}) := \text{lfp \ (iter Env (Collect i p) t p)\ in}
        (Collect i p (assert t I))
        +[p \mapsto I] +[l \mapsto \text{assert (Not t) I}])
    end.

| [...]

end.
Collecting Semantics

Program Fixpoint Collect (i:stmt) (l:pp):

\[
\text{monotone } (P(\text{env})) \ (pp \rightarrow P(\text{env})) :=
\]

match i with

| Assign p x e =>
  Mono (fun Env => \bot +[p \mapsto \text{Env}] +[l \mapsto \text{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 \mapsto I] +[l \mapsto \text{assert (Not t) I}]) _

| [...] end.
Collecting Semantics

Program Fixpoint Collect (i:stmt) (l:pp):

\[
\text{monotone } (\mathcal{P}(\text{env})) \quad (\text{pp} \rightarrow \mathcal{P}(\text{env})) := \\
\text{match } i \text{ with} \\
| \text{Assign } p \ x \ e \ => \\
\quad \text{Mono } (\text{fun } \text{Env} \Rightarrow \bot + [p \mapsto \text{Env}] + [l \mapsto \text{assign } x \ e \ \text{Env}]) \\
| \text{While } p \ t \ i \ => \\
\quad \text{Mono } (\text{fun } \text{Env} \Rightarrow \\
\quad \quad \text{let } I : \mathcal{P}(\text{env}) := \text{lfp } (\text{iter } \text{Env } (\text{Collect } i \ p) \ t \ p) \ \text{in} \\
\quad \quad (\text{Collect } i \ p (\text{assert } t \ I)) \\
\quad \quad + [p \mapsto I] + [l \mapsto \text{assert } (\text{Not } t) \ I]) \\
| \text{[...]} \\
\text{end}. \\
\]

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) (\top) k env.

**Theorem** reachable\_sos\_implies\_reachable\_collect :
   \forall p, reachable\_sos p \subseteq 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) (\top) k env.

Theorem reachable_sos_implies_reachable_collect : 
\forall p, \text{reachable_sos}\ p \subseteq \text{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

Collecting Semantics

Lattice Theory Intermezzo

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 compute on \((\mathbf{P} \to \mathcal{P}(\text{env}))\) but on an abstract lattice \(\mathbb{A}\)
Abstract Lattice

Abstract lattices are formalized with type classes

\[
\text{AbLattice}.t : \sqsubseteq, \sqcap, \sqcup, \bot + \text{widening/narrowing}
\]

Each abstract lattice is equipped with a post-fixpoint solver

**Definition** approx_lfp :

\[
\forall \{t\} \{L:\text{AbLattice}.t t\}, (t \to t) \to t := [...]
\]

**Lemma** approx_lfp_is_postfixpoint :

\[
\forall t (L:\text{AbLattice}.t t) (f:t \to t),
    f (\text{approx_lfp} f) \sqsubseteq (\text{approx_lfp} f).
\]
Fixpoint approximation with widening/narrowing

\[ f(x) \subseteq x \]

\[ f(x) = x \]

\[ \text{lfp}(f) \]
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:

\begin{verbatim}
Instance ProdLattice
t1 t2 {L1:AbLattice.t t1} {L2:AbLattice.t t2}:
AbLattice.t (t1*t2) := [...]
\end{verbatim}

\begin{verbatim}
Instance ArrayLattice t {L:AbLattice.t t}:
AbLattice.t (array t) := [...]
\end{verbatim}

\(^1\)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:

```plaintext
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 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 {
  \gamma : A \to a;
  _\text{monotone} : \forall N1 N2 : A, N1 \sqsupseteq N2 \to \gamma N1 \sqsubseteq \gamma N2;
  _\text{meet_morph} : \forall N1 N2 : A, \gamma N1 \sqcap \gamma N2 \sqsubseteq \gamma (N1 \sqcap \# 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
- Collecting Semantics
- A Few Coq complements
- Lattice Theory Intermezzo
- 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.

```plaintext
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
```
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
    k ≥ 0  i ≥ 0
    i = i + 2
}
    k ≥ 0  i ≥ 0

sign
The analyzer is parameterized wrt. to an environment abstraction.

The development provides several non-relational instantiations.

\[
\begin{align*}
  i &= 0; k = 0; \\
  k &\in [0,10] \quad i \in [0,10] \\
  \text{while } k < 10 \{ \\
  k &\in [0,9] \quad i \in [0,10] \\
  i &= 0; \\
  k &\in [0,9] \quad i \in [0,10] \\
  \text{while } i < 9 \{ \\
  k &\in [0,9] \quad i \in [0,8] \\
  i &= i + 2 \\
  } \\
  k &= k + 1 \\
\end{align*}
\]

interval

\[
\begin{align*}
  i &= 0; k = 0; \\
  k &\geq 0 \quad i \geq 0 \\
  \text{while } k < 10 \{ \\
  k &\geq 0 \quad i \geq 0 \\
  i &= 0; \\
  k &\geq 0 \quad i \geq 0 \\
  \text{while } i < 9 \{ \\
  k &\geq 0 \quad i \geq 0 \\
  i &= i + 2 \\
  } \\
  k &= k + 1 \\
\end{align*}
\]

sign

\[
\begin{align*}
  i &= 0; k = 0; \\
  i &\equiv 0 \mod 2 \\
  \text{while } k < 10 \{ \\
  i &\equiv 0 \mod 2 \\
  i &= 0; \\
  i &\equiv 0 \mod 2 \\
  \text{while } i < 9 \{ \\
  i &\equiv 0 \mod 2 \\
  i &= i + 2 \\
  } \\
  k &= k + 1 \mod 2 \\
\end{align*}
\]

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:pp) :=
match i with
| Assign p x e =>
  fun Env =>
    +[p \mapsto Env] +[l \mapsto Ab.assign Env x e]
| While p t i =>
  fun Env =>
    let I := approx_lfp
      (fun X => Env \mapsto
        (get (AbSem i p (Ab.assert t X)) p)) in
      (AbSem i p (Ab.assert t I))
    +[p \mapsto I] +[l \mapsto 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 $(i : instr) (l : pp)$ :=

match $i$ with

| Assign $p \ x \ e$ =>
  fun $Env$ =>
    $[p \mapsto Env] + [l \mapsto Ab.assign Env x e]$

| 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 $t$ $I$))
    $[p \mapsto I] + [l \mapsto Ab.assert (Not t) I]$
| [...] end.
Connecting Concrete and Abstract Semantics

Theorem AbSem_correct : \( \forall i \ l \_end \ Env, \)
\( \text{Collect prog i l\_end (} \gamma \ \text{Env) } \subseteq \gamma \ (\text{AbSem i l\_end Env}) \).
Connecting Concrete and Abstract Semantics

Theorem AbSem_correct : \( \forall \ i \ l\_end \ Env, \)
\( \text{Collect \ prog \ i \ l\_end \ (\gamma \ Env) \ \subseteq \ \gamma \ (\text{AbSem \ i \ l\_end \ Env}) \).
Type Classes to the rescue

**Theorem** \( \text{AbSem\_correct} : \forall i \ l\_end \ \text{Env}, \)
\( \text{Collect} \ \text{prog} \ i \ l\_end \ (\gamma \ \text{Env}) \subseteq \gamma \ (\text{AbSem} \ i \ l\_end \ \text{Env}). \)
Type Classes to the rescue

**Theorem** \( \text{AbSem\_correct} : \forall i \text{ l\_end Env,} \)
\[
\text{Collect prog i l\_end } (\gamma \text{ Env}) \sqsubseteq \gamma \ (\text{AbSem i l\_end Env}) .
\]

canonical order on \( pp \rightarrow \mathcal{P} (\text{env}) \)
Type Classes to the rescue

**Theorem** \( \text{AbSem\_correct} : \forall i \; \text{l\_end} \; \text{Env}, \)
\[ \text{Collect} \; \text{prog} \; i \; \text{l\_end} \; (\gamma \; \text{Env}) \sqsubseteq \gamma \; (\text{AbSem} \; i \; \text{l\_end} \; \text{Env}) \]
Type Classes to the rescue

Theorem AbSem_correct : \forall i l_end Env, Collect prog i l_end (\gamma Env) \subseteq \gamma (AbSem i l_end Env).

concretization on \( \mathbb{P}(\text{env}) \)

concretization on \( \mathbb{P}(\text{env}) \)

canonical order on \( \mathbb{P}(\text{env}) \rightarrow \mathbb{P}(\text{env}) \)

Need 4 minutes after...
Type Classes to the rescue

Theorem `AbSem_correct` : \( \forall \ i \ l\_end \ Env, \)
\( \text{Collect prog i l\_end} (\gamma \ Env) \subseteq \gamma \ (AbSem \ i \ l\_end \ Env). \)

Without Type Classes

\( \text{Need 4 minutes after} \)
Theorem AbSem_correct : \( \forall i \, l_{\text{end}} \, \text{Env}, \)
\[ \text{Collect \ prog \ i \ l_{\text{end}} \ (\gamma \ \text{Env}) \subseteq \gamma \ (\text{AbSem \ i \ l_{\text{end}} \ Env}) \].

Without Type Classes

Theorem AbSem_correct : \( \forall i \, l_{\text{end}} \, \text{Env}, \)
\[ (\text{PointwisePoset} \ (\text{PowerSetPoset} \ \text{env})) \cdot (\text{Poset}.c) \]
\[ (\text{Collect \ prog \ i \ l_{\text{end}} \ (\text{AbEnv}.(\text{AbEnv}.\gamma) \ \text{Env})) \]
\[ (\text{FuncLattice}.\Gamma \ \text{AbEnv}.(\text{AbEnv}.\gamma) \ (\text{AbSem \ i \ l_{\text{end}} \ Env})) \].
Type Classes to the rescue

**Theorem** AbSem_correct : \( \forall i \ l \_end \ Env, \) 
\( \text{Collect prog i l\_end (}\gamma \ Env) \subseteq \gamma \ (\text{AbSem i l\_end Env}). \)

Without Type Classes

**Theorem** AbSem_correct : \( \forall i \ l \_end \ Env, \) 
\( \text{(PointwisePoset (PowerSetPoset env)).(Poset.} \) 
\( \text{order)} \) 
\( \text{(Collect prog i l\_end (AbEnv.(AbEnv.gamma) Env))} \) 
\( \text{(FuncLattice.Gamma AbEnv.(AbEnv.gamma) (AbSem i l\_end Env))}. \)
Type Classes to the rescue

Theorem AbSem_correct : \( \forall \ i \ l\_end \ Env, \)
\[
\text{Collect prog i l_end (\( \gamma \ Env \))} \subseteq \gamma \ (\text{AbSem i l_end Env}) .
\]

Without Type Classes

Theorem AbSem_correct : \( \forall \ i \ l\_end \ Env, \)
\[
\text{(PointwisePoset (PowerSetPoset env)).(Poset.order)} \)
\[
\text{(Collect prog i l_end (AbEnv.(AbEnv.gamma) Env))} \)
\[
\text{(FuncLattice.Gamma AbEnv.(AbEnv.gamma) (AbSem i l_end Env))} .
\]

Need 4 minutes after
Connecting Concrete and Abstract Semantics

**Theorem** AbSem_correct : \( \forall \ i \ l\_end \ Env,\)
Collect prog i l_end (\( \gamma \ Env \)) \( \subseteq \gamma \) (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 \(\rightarrow\) \(P(env)\)) :=
match i with
| Assign p x e =>
    Mono (fun Env => ⊥ +[p \rightarrow Env] +[l \rightarrow assign x e Env]) _
| While p t i =>
    Mono (fun Env =>
        let I: \(\mathcal{P}(env)\) := lfp (iter Env (Collect i p) t p) in
        (Collect i p (assert t I)) +[p \leftrightarrow I] +[l \leftrightarrow assert (Not t) I]) _
        [...]
end.

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

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.

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 ⊥ +[p → Env] +[l → Ab.assign Env x e]) in
    (AbSem i p (Ab.assert t I)) +[p → I] +[l → Ab.assert (Not t) I]
[...]
end.
Definition \text{analyse} : \text{array t} := \\
\text{AbSem} \ prog. (p\_instr) \ prog. (p\_end) \ (\text{Ab} \ . \ top) .

Theorem \text{analyse\_correct} : \forall \ k \ \text{env}, \\
\text{reachable\_sos} \ prog \ (k, \text{env}) \rightarrow \gamma \ (\text{get} \ \text{analyse} \ k) \ \text{env} .

The function \text{analyse} can be extracted to real OCaml code

You can type-check, extract and run the analyser yourself!

\url{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
  • 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?

See next lecture