Certified Proof Carrying Code by abstract interpretation
Types Summer School 2007 - Bertinoro - Italy

David Pichardie

INRIA Rennes - Bretagne Atlantique
Outline

1 Certified static analysis
Outline

1 Certified static analysis
   - Introduction
1 Certified static analysis
   • Introduction
   • Building a certified static analyser
Outline

1 Certified static analysis
   - Introduction
   - Building a certified static analyser

2 From certified static analysis to certified PCC
Outline

1. Certified static analysis
   - Introduction
   - Building a certified static analyser

2. From certified static analysis to certified PCC

3. A case study: array-bound checks polyhedral analysis
Outline

1. Certified static analysis
   - Introduction
   - Building a certified static analyser

2. From certified static analysis to certified PCC

3. A case study: array-bound checks polyhedral analysis
   - Polyhedral abstract interpretation
Outline

1. Certified static analysis
   - Introduction
   - Building a certified static analyser

2. From certified static analysis to certified PCC

3. A case study: array-bound checks polyhedral analysis
   - Polyhedral abstract interpretation
   - Certified polyhedral abstract interpretation
Outline

1. Certified static analysis
   - Introduction
   - Building a certified static analyser

2. From certified static analysis to certified PCC

3. A case study: array-bound checks polyhedral analysis
   - Polyhedral abstract interpretation
   - Certified polyhedral abstract interpretation
   - Application: a polyhedral bytecode analyser
Static program analysis

The goals of static program analysis

- To prove properties about the run-time behaviour of a program
- In a fully automatic way
- Without actually executing this program
Static program analysis

The goals of static program analysis
▶ To prove properties about the run-time behaviour of a program
▶ In a fully automatic way
▶ Without actually executing this program

Solid foundations for designing an analyser
▶ Abstract Interpretation gives a guideline
  ▶ to formalise analyses
  ▶ to prove their soundness with respect to the semantics of the programming language
▶ Resolution of constraints on lattices by iteration and symbolic computation
So what’s the problem?
Proof

$$\hat{\alpha}[P](\text{Post[if } B \text{ then } S_t \text{ else } S_f \text{ fi]}$$

$$= \text{def. (110) of } \hat{\alpha}[P]$$

$$\hat{\alpha}[P] \circ \text{Post[if } B \text{ then } S_t \text{ else } S_f \text{ fi]} \circ \hat{\gamma}[P]$$

$$= \text{def. (103) of Post}$$

$$\hat{\alpha}[P] \circ \text{post}[\tau^*\text{if } B \text{ then } S_t \text{ else } S_f \text{ fi}] \circ \hat{\gamma}[P]$$

$$= \text{big step operational semantics (93)}$$

$$\hat{\alpha}[P] \circ \text{post}[\{1_{\Sigma}[P] \cup \tau^B \} \cup \tau^*[S_t] \circ \{1_{\Sigma}[P] \cup \tau^f \} \cup \tau^*[S_f] \circ \{1_{\Sigma}[P] \cup \tau^f \}] \circ \hat{\gamma}[P]$$

$$= \text{Galois connection (98) so that post preserves joins}$$

$$\hat{\alpha}[P] \circ \text{post}[\{1_{\Sigma}[P] \cup \tau^B \} \circ \tau^*[S_t] \circ \{1_{\Sigma}[P] \cup \tau^f \}] \circ \hat{\gamma}[P]$$

$$= \text{Galois connection (106) so that } \hat{\alpha}[P] \text{ preserves joins}$$

$$(\hat{\alpha}[P] \circ \text{post}[\{1_{\Sigma}[P] \cup \tau^B \} \circ \tau^*[S_t] \circ \{1_{\Sigma}[P] \cup \tau^f \}] \circ \hat{\gamma}[P]) \circ (\hat{\alpha}[P] \circ \text{post}[\{1_{\Sigma}[P] \cup \tau^B \} \circ \tau^*[S_t] \circ \{1_{\Sigma}[P] \cup \tau^f \}] \circ \hat{\gamma}[P])$$

$$= \text{lemma (5.3) and similar one for the else branch}$$

$$\lambda J \cdot \text{let } J'' = \lambda l \in \text{in}_P[l = \text{at}_P[S_t] ? J_{\text{at}_P[S_t]} \cup \text{Abexp}[B](J_t) \& J_l) \text{ in}$$

$$\text{let } J'''' = \text{APost}[S_t](J'') \text{ in}$$

$$\lambda l \in \text{in}_P[l = \text{at}_P[S_t] ? J'''' \cup J_{\text{after}_P[S_t]} \& J'''' \text{ in}$$

$$\lambda l \in \text{in}_P[l = \text{at}_P[S_t] ? J'''' \cup J_{\text{after}_P[S_t]} \& J'''' \text{ in}$$

$$= \text{by grouping similar terms}$$

$$\lambda J \cdot \text{let } J'' = \lambda l \in \text{in}_P[l = \text{at}_P[S_t] ? J_{\text{at}_P[S_t]} \cup \text{Abexp}[B](J_t) \& J_l) \text{ and}$$

$$J'''' = \lambda l \in \text{in}_P[l = \text{at}_P[S_t] ? J_{\text{at}_P[S_t]} \cup \text{Abexp}[T(\neg B)](J_t) \& J_l) \text{ in}$$

$$\text{let } J'''' = \text{APost}[S_t](J'') \text{ in}$$

$$\text{and } J'''' = \text{APost}[S_t](J'') \text{ in}$$

$$\lambda l \in \text{in}_P[l = \text{at}_P[S_t] ? J'''' \cup J_{\text{after}_P[S_t]} \& J'''' \cup J_{\text{after}_P[S_t]} \& J'''' \cup J_{\text{after}_P[S_t]} \& J'''' \text{ in}$$

$$\lambda l \in \text{in}_P[l = \text{at}_P[S_t] ? J'''' \cup J_{\text{after}_P[S_t]} \& J'''' \cup J_{\text{after}_P[S_t]} \& J'''' \text{ in}$$

$$= \text{by locality (113) and labelling scheme (59) so that in particular } J'''' = J'''' = J'''' = J'''' \text{ and } \text{APost}[S_t] \text{ and APost}[S_t] \text{ do not interfere}$$

©P.Cousot
Proof

\[ \hat{\alpha}(P) \circ \text{Post}[\text{if } B \text{ then } S_t \text{ else } S_f \text{ fi}] = \hat{\alpha}(P) \circ (\text{Post}[\text{if } B \text{ then } S_t \text{ else } S_f \text{ fi}] \circ \hat{\gamma}(P)) \]

\[ = \hat{\alpha}(P) \circ \text{Post}[\text{if } B \text{ then } S_t \text{ else } S_f \text{ fi}] \circ \hat{\gamma}(P) \]

\[ = \text{big step operational semantics (93)} \]

\[ \hat{\alpha}(P) \circ \text{post}(\{(1\Sigma)[P] \cup \tau^B\} \circ \tau^*[S_t] \circ (1\Sigma)[P] \cup \tau^I \} \circ (1\Sigma)[P] \cup \tau^I \} \circ \hat{\gamma}(P)) \]

\[ = \hat{\gamma}(P) \circ \text{post}(\{(1\Sigma)[P] \cup \tau^B\} \circ \tau^*[S_t] \circ (1\Sigma)[P] \cup \tau^I \} \circ \hat{\gamma}(P)) \]

\[ \triangleq \text{Galois connection (98) so that post preserves joins' \}

\[ \lambda J \cdot \text{let } J^i = \lambda l \in \text{in}_P[P].(l = \text{at}_P[S_t] \cup \text{Abexp}(B)(J_i) \otimes J_i) \text{ in } \]

\[ \text{let } J^r = \text{APost}[S_t](J^i) \text{ in } \]

\[ \lambda l \in \text{in}_P[P].(l = l' \otimes J^r \cup J^r_{\text{after}_P[S_t]} \otimes J^r_{i}) \]

\[ \triangleq \text{lemma (5.3) and similar one for the else branch'} \]

\[ \lambda J \cdot \text{let } J^i = \lambda l \in \text{in}_P[P].(l = \text{at}_P[S_t] \cup \text{Abexp}(B)(J_i) \otimes J_i) \text{ in } \]

\[ \text{let } J^r = \text{APost}[S_t](J^i) \text{ in } \]

\[ \lambda l \in \text{in}_P[P].(l = l' \otimes J^r \cup J^r_{\text{after}_P[S_t]} \otimes J^r_{i}) \]

\[ \triangleq \text{by grouping similar terms'} \]

\[ \lambda J \cdot \text{let } J^i = \lambda l \in \text{in}_P[P].(l = \text{at}_P[S_t] \cup \text{Abexp}(B)(J_i) \otimes J_i) \text{ and } J^r = \lambda l \in \text{in}_P[P].(l = \text{at}_P[S_t] \cup \text{Abexp}(\neg B)(J_i) \otimes J_i) \text{ in } \]

\[ \text{let } J^r = \text{APost}[S_t](J^i) \text{ and } J^r = \text{APost}[S_t](J^i) \text{ in } \]

\[ \lambda l \in \text{in}_P[P].(l = l' \otimes J^r \cup J^r_{\text{after}_P[S_t]} \otimes J^r_{i} \cup J^r_{i}) \]

\[ \text{by locality (113) and labelling scheme (59) so that in particular } J^r_{i} = J^r_{i} = J^r_{i} = J^r_{i} \]

\[ = J^r_{i} = J^r_{i} \text{ and APost}[S_t] \text{ and APost}[S_t] \text{ do not interfere'} \]

©P.Cousot

Implementation

```c
matrix_t* _matrix_alloc_int(const int mr, const int nc) {
  matrix_t* mat = (matrix_t*)malloc(sizeof(matrix_t));
  mat->nbrows = mat->_maxrows = mr;
  mat->ncolumns = nc;
  mat->_sorted = s;
  if (mr*nc>0) {
    int i;
    pkint_t* q;
    mat->_pinit = _vector_alloc_int(mr*nc);
    mat->p = (pkint_t**)malloc(mr * sizeof(pkint_t*));
    q = mat->_pinit;
    for (i=0;i<mr;i++) {
      mat->p[i]=q;
      q=q+nc;
    }
    return mat;
  }
}
void backsubstitute(matrix_t* con, int rank) {
  int i,j,k;
  for (k=rank-1; k>=0; k--) {
    j = pk_cherni_intp[k];
    for (i=0; i<k; i++) {
      if (pkint_sgn(con->p[i][j]))
        matrix_combine_rows(con,i,k,i,j);
    }
  }
  for (i=k+1; i<con->nbrows; i++) {
    if (pkint_sgn(con->p[i][j]))
      matrix_combine_rows(con,i,k,i,j);
  }
}
```

©B.Jeannet
Proof

\[
\begin{align*}
\hat{\alpha}(P) \circ \text{Post}[\text{if } B \text{ then } S_f \text{ else } S_f \text{ fi}] \\
= \text{def. (110) of } \hat{\alpha}(P) \\
= \text{def. (103) of Post} \\
= \text{big step operational semantics (93)} \\
= \text{Galois connection (98) so that Post preserves joins} \\
= \text{Galois connection (106) so that } \hat{\alpha}(P) \text{ preserves joins} \\
= \text{by grouping similar terms} \\
\end{align*}
\]

\[
\lambda \in \text{in}_P(P) \cdot (l = l' ? J_c^{l'} \cup J_f^{l'} \gamma_{after}^{l'} & J_f^l) \\
\text{let } J_f^{l'} = \lambda \in \text{in}_P(P) \cdot (l = at_P[S_f] \wedge J_{at}\gamma_{after}^{l'} \cup \text{Abexp}[T(-B)](J_f) \wedge J_f^{l'}) \text{ in} \\
\text{let } J_f^{l'} = \text{Post}(\text{Abexp}[T(-B)](J_f) \wedge J_f^{l'}) \text{ in} \\
\text{let } J_f^{l'} = \text{Abexp}[T(-B)](J_f) \wedge J_f^{l'}) \text{ in} \\
\text{by locality (113) and labelling scheme (59) so that in particular } J_c^{l'} = J_f^{l'} = J_f^{l'} \text{ and } J_f^{l'} = J_f^{l'} \text{ and } J_f^{l'} \text{ and } J_f^{l'} \text{ do not interfere} \\
\]

Do the two parts talk about the same?

@P.Cousot

Implementation

```c
matrix_t* _matrix_alloc_int(const int mr, const int nc) {
    matrix_t* mat = (matrix_t*)malloc(sizeof(matrix_t));
    mat->nbrows = mat->maxrows = mr;
    mat->ncolumns = nc;
    mat->_sorted = s;
    if (mr*nc>0){
        int i;
        pkint_t* q;
        mat->_pinit = _vector_alloc_int(mr*nc);
        mat->p = (pkint_t*)malloc(mr * sizeof(pkint_t*));
        q = mat->_pinit;
        for (i=0;i<mr;i++){
            mat->p[i]=q;
            q = q+nc;
        }
    }
    return mat;
}
```

void backsubstitute(matrix_t* con, int rank) {
    int i,j,k;
    for (k=rank-1; k>=0; k--) {
        j = pk_int_i(k);
        for (i=0; i<nc; i++) {
            if (pkint_sgn(con->p[i][j]))
                matrix_combine_rows(con,i,k,i,j);
        }
    }
}
```

©B.Jeannet
Certified static analyses

A certified static analysis is an analysis whose implementation has been formally proved correct using a proof assistant.

- proof assistant: Coq
  - we benefit from the extraction mechanism to prove executable analyser
- proof technique: abstract interpretation
  - general enough to handle a broad range of static analysis
- applications to static analysis of bytecode programs
  - to go beyond the state of the art about Sun’s bytecode verifier
Abstract Interpretation

Abstract Interpretation is a method for designing approximate semantics of programs.

- An approximate semantics mimics the concrete one, considering only a fragment of the properties
- Application to static analysis: static analysers are computable approximate semantics of programs
- A method to prove soundness of static analysis with respects to a semantics
- A method to formally design static analysis by systematic abstraction of the semantics of programs
- A method to compare precision between different analyses.

1See http://www.di.ens.fr/~cousot/
Abstract Interpretation

Abstract Interpretation is a method for designing approximate semantics of programs.

- An approximate semantics mimics the concrete one, considering only a fragment of the properties
- Application to static analysis: static analysers are computable approximate semantics of programs
- A method to prove soundness of static analysis with respect to a semantics
- A method to formally design static analysis by systematic abstraction of the semantics of programs
- A method to compare precision between different analyses.

We focus here on a fragment of the theory because we only prove soundness

^1See [http://www.di.ens.fr/~cousot/](http://www.di.ens.fr/~cousot/)
A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of values.

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.

```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 values.

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.

```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 values.

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.

```plaintext
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 values.

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.

```plaintext
x = 0; y = 0;
{{(0,0)}}
while (x<6) {
  if (?) {
    {{(0,0)}}
    y = y+2;
    {{(0,2)}}
  }
  { }
  x = x+1;
  {{(1,0)}}
}
```
A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of values.

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.

```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 values.

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.

```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;
    {(1,0),(1,2)}
}
```
A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of values.

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.

```plaintext
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 values.

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.

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

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.

```plaintext
x = 0; y = 0;
{(0,0),(1,0),(1,2)  }

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

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.

```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 values.

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.

```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 values.

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.

```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 values.

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.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
- Example: sign of variables
- To stay in the domain of selected properties, we over-approximate the concrete properties.
A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of values.

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.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.

Example: sign of variables

- To stay in the domain of selected properties, we over-approximate the concrete properties.
A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of values.

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.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example: sign of variables
- 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 = x+1;
}
```
A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of values.

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.

Approximation

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

- 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 values.

Collecting semantics

- A state property is a subset in \( P(\mathbb{Z}^2) \) of \((x, y)\) values.
- When a point is reached for a second time we make an union with the previous property.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
- Example: sign of variables
- 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;
    x > 0 \land y \geq 0
}
```
A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of values.

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.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.

Example: sign of variables

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

```
x = 0; y = 0;
\textbf{x} \geq 0 \land y \geq 0

\textbf{while} (x<6) {
\textbf{if} (?) {
    x = 0 \land y = 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 values.

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.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.

Example: sign of variables

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

```plaintext
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 values.

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.

Approximation

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

**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.

**Approximation**

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

- 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
```

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

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.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.

Example: sign of variables

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 \geq 0
  }
  \ x \geq 0 \land y \geq 0
  x = x+1;
  \ x \geq 0 \land y \geq 0
}
```
A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of values.

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.

Approximation

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

- 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 \geq 0
    y = y+2;
    x \geq 0 \land y \geq 0
  }
}

x \geq 0 \land y \geq 0
x = x+1;
\quad x \geq 0 \land y \geq 0
```
Outline

1 Certified static analysis
   - Introduction
   - Building a certified static analyser

2 From certified static analysis to certified PCC

3 A case study: array-bound checks polyhedral analysis
   - Polyhedral abstract interpretation
   - Certified polyhedral abstract interpretation
   - Application: a polyhedral bytecode analyser
Building a certified static analyser

- A puzzle with 8 pieces,
- Each piece interacts with its neighbors
Building a certified static analyser

Example: JVM states

\[
\langle \langle h, \langle m, pc, l, v :: s \rangle, sf \rangle \rangle
\]

- heap
- method
- program point
- frame
- call stack
- operand stack
- local variables
Each semantic sub-domain has its abstract counterpart

- An abstract domain is a lattice \((D^\#, =, \sqsubseteq, \perp, \sqcup, \sqcap)\) without infinite strictly increasing chains \(x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq \cdots\)
- First difficult point: how can we quickly develop big lattice structures in Coq?
Building a certified static analyser

- Each semantic sub-domain has its abstract counterpart
- An abstract domain is a lattice \((\mathcal{D}^\#, =, \sqsubseteq, \bot, \sqcup, \sqcap)\) without infinite strictly increasing chains \(x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq \cdots\)
- First difficult point: how can we quickly develop big lattice structures in Coq?
  - generic lattice library
Building lattices in Coq

We propose a technique based on the new Coq module system (inspired by the ML module system)

▶ Lattice requirements are collected in a module contract
Lattice contract

Module Type LatticeWf.

End Lattice.
Lattice contract

Module Type LatticeWf.
  Parameter t : Set.

End Lattice.
Lattice contract

Module Type LatticeWf.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_prop : ...
    (* eq (\(=\)) is a computable equivalence relation *)

End Lattice.
Lattice contract

Module Type LatticeWf.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_prop : ...
    (* eq (=) is a computable equivalence relation *)
  Parameter order : t → t → Prop.
  Parameter order_prop : ...
    (* order (⊆) is a computable order relation *)

End Lattice.
Lattice contract

Module Type LatticeWf.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_prop : ...
    (* eq (≡) is a computable equivalence relation *)
  Parameter order : t → t → Prop.
  Parameter order_prop : ...
    (* order (⊑) is a computable order relation *)
  Parameter join : t → t → t.
  Parameter join_prop : ...
    (* join (⊔) is a binary least upper bound *)

End Lattice.
Lattice contract

Module Type LatticeWf.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_prop : ... (* eq (\(=\)) is a computable equivalence relation *)
  Parameter order : t → t → Prop.
  Parameter order_prop : ... (* order (\(\sqsubseteq\)) is a computable order relation *)
  Parameter join : t → t → t.
  Parameter join_prop : ... (* join (\(\sqcup\)) is a binary least upper bound *)
  Parameter meet : t → t → t.
  Parameter meet_prop : ... (* meet (\(\sqcap\)) is a binary greatest lower bound *)

End Lattice.
Lattice contract

Module Type LatticeWf.

Parameter t : Set.
Parameter eq : t → t → Prop.
Parameter eq_prop : ...
  (* eq (≡) is a computable equivalence relation *)
Parameter order : t → t → Prop.
Parameter order_prop : ...
  (* order (⊑) is a computable order relation *)
Parameter join : t → t → t.
Parameter join_prop : ...
  (* join (⊔) is a binary least upper bound *)
Parameter meet : t → t → t.
Parameter meet_prop : ...
  (* meet (⊓) is a binary greatest lower bound *)
Parameter bottom : t.
  (* bottom element to start iteration *)
Parameter bottom_is_bottom : ∀ x : t, order bottom x.

End Lattice.
Lattice contract

Module Type LatticeWf.
Parameter t : Set.
Parameter eq : t → t → Prop.
Parameter eq_prop : ...
  (* eq (=) is a computable equivalence relation *)
Parameter order : t → t → Prop.
Parameter order_prop : ...
  (* order (⊑) is a computable order relation *)
Parameter join : t → t → t.
Parameter join_prop : ...
  (* join (⊔) is a binary least upper bound *)
Parameter meet : t → t → t.
Parameter meet_prop : ...
  (* meet (⊓) is a binary greatest lower bound *)
Parameter bottom : t.
  (* bottom element to start iteration *)
Parameter bottom_is_bottom : ∀ x : t, order bottom x.
Parameter termination_property : well_founded □
End Lattice.
Building lattices in Coq

We propose a technique based on the new Coq module system (inspired by the ML module system)

- Lattice requirements are collected in a module contract

- Various functors are proposed in order to build lattices by composition of others
Lattice functors

- **Disjoint sum, linear sum, product**

  ```coq
  Module ProdLatWf (P1 : LatticeWf) (P2 : LatticeWf) : LatticeWf
    with Definition t := P1.t * P2.t
    with Definition eq := fun x y : (P1.t * P2.t) =>
      P1.eq (fst x) (fst y) \land P2.eq (snd x) (snd y)
    with Definition order := fun x y : (P1.t * P2.t) =>
      P1.order (fst x) (fst y) \land P2.order (snd x) (snd y).
  ...
  End ProdLatWf.
  ```

- **List of elements from a lattice**
- **Map from a finite set of keys to a lattice (using efficient data-structures)**

For each functor the most challenging proofs deals with the preservation of the termination criterion.
Building lattices in Coq

We propose a technique based on the new Coq module system (inspired by the ML module system)

- Lattice requirements are collected in a module contract

- Various functors are proposed in order to build lattices by composition of others

- The library deals as well with widening/narrowing
Building lattices in Coq

We propose a technique based on the new Coq module system (inspired by the ML module system)

- Lattice requirements are collected in a module contract

- Various functors are proposed in order to build lattices by composition of others

- The library deals as well with widening/narrowing

Example:

```
Module AbSt :=
  Product (Array (Array (List (Sum FiniteSet Constant))))
  (Product (Array (Array (List (Sum FiniteSet Constant))))))
  (Array (Array (List (Sum FiniteSet Constant)))))
```
post-fixpoint computation by widening/narrowing ∇/∆

[Cousot & Cousot 77]

1. we compute the limit of
   \[ x_0 = \bot, x_{n+1} = x_n \nabla f(x_n) \]
2. we reach a post-fixpoint \( a \) of \( f \)
3. we compute the limit of
   \[ x_0 = \bot, x_{n+1} = x_n \Delta f(x_n) \]
4. we reach a post-fixpoint \( a' \) of \( f \)

\[ \text{lfp}(f) \subseteq a' \subseteq a \]
Building lattices in Coq

We propose a technique based on the new Coq module system (inspired by the ML module system)

- Lattice requirements are collected in a module contract

- Various functors are proposed in order to build lattices by composition of others

- The library deals as well with widening/narrowing

Example:

```coq
Module AbSt :=
  Product (Array (Array (List (Sum FiniteSet Constant))))
  (Product (Array (Array (Array (List (Sum FiniteSet Constant))))))
  (Array (Array (List (Sum FiniteSet Constant)))))
```

David Pichardie
Certified Proof Carrying Code by abstract interpretation
Building a certified static analyser

- Each abstract value represents a property on concrete values
- This correspondence is formalised by a monotone concretisation function

\[ \gamma : (\mathcal{D}^\#, \sqsubseteq) \rightarrow_m (\wp(\mathcal{D}), \subseteq) \]
Building a certified static analyser

- Each abstract value represents a property on concrete values
- This correspondence is formalised by a monotone concretisation function

$$\gamma : (D^\#, \sqsubseteq) \xrightarrow{m} (\wp(D), \subseteq)$$

$$x \subseteq \gamma(x^\#)$$ means “$$x^\#$$ is a correct approximation of $$x$$”
Building a certified static analyser

- operational semantics $\rightarrow_P$ between states
- collecting semantics: $[[P]] = \{ s \mid \exists s_0 \in S_{init}, s_0 \rightarrow^*_P s \}$
- we want to compute a correct approximation of $[[P]]$
  - a sound invariant $s^\#$ on the reachable states: $[[P]] \subseteq \gamma(s^\#)$
Example: JVM operational semantics

\[
\text{instructionAt}_P(m, pc) = \text{push } c
\]
\[
\langle \langle h, \langle m, pc, l, s \rangle, sf \rangle \rangle \rightarrow \langle \langle h, \langle m, pc + 1, l, c :: s \rangle, sf \rangle \rangle
\]

\[
\text{instructionAt}_P(m, pc) = \text{invokevirtual } m_{id}
\]
\[
m' = \text{methodLookup}(m_{id}, h(loc))
\]
\[
V = v_1 :: \cdots :: v_{\text{nbArguments}(m_{id})}
\]
\[
\langle \langle h, \langle m, pc, l, \text{loc} :: V :: s \rangle, sf \rangle \rangle \rightarrow \langle \langle h, \langle m', 1, V, \varepsilon \rangle, \langle m, pc, l, s \rangle :: sf \rangle \rangle
\]
the analysis is specified as a solution of a post fixpoint problem

\[ F_P^\#(s^\#) \sqsubseteq^\# s^\# \]

after partitioning: constraint system

\[
\begin{aligned}
& f_1^\#(s_1^\#, \ldots, s_n^\#) \sqsubseteq^\# s_{i_1}^\# \\
& \ldots \\
& f_n^\#(s_1^\#, \ldots, s_n^\#) \sqsubseteq^\# s_{i_n}^\#
\end{aligned}
\]
Building a certified static analyser

∀P, ∀s#, F_p(s#) ⊆ s# ⇒ [P] ⊆ γ(s#)

- easy proof, but tedious
- one proof by instruction: a long work for real languages
Building a certified static analyser

<table>
<thead>
<tr>
<th>semantics domains</th>
<th>logic links</th>
<th>abstract domains</th>
<th>constraint generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>semantic rules</td>
<td>soundness proof</td>
<td>analyse specification</td>
<td></td>
</tr>
</tbody>
</table>

- collects all constraints in a program
- generic tool
### Building a certified static analyser

<table>
<thead>
<tr>
<th>semantics domains</th>
<th>logic links</th>
<th>abstract domains</th>
<th>fixpoint solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>semantic rules</td>
<td>soundness proof</td>
<td>analyse specification</td>
<td>constraint generation</td>
</tr>
</tbody>
</table>

\[ \forall P, \exists s^#, \quad F^\#_P(s^#) \sqsubseteq s^# \]

Two techniques of iterative computation

- **traditional least (post)-fixpoint computation**
  \[ \bot \rightarrow F^\#_P(\bot) \rightarrow F^\#_P^2(\bot) \rightarrow \cdots \text{lfp}(F^\#_P) \]

- **post-fixpoint computation by widening/narrowing with chaotic iterations**

In the two cases, a generic tool
Building a certified static analyser

Final result

∀P, ∀s#, F_p#(s#) ⊆ s# ⇒ \(\llbracket P \rrbracket \subseteq \gamma(s#)\)
∀P, ∃s#, F_p#(s#) ⊆ s#

In Coq:  analyse : ∀ p:program, { s:abstate | sem(P) ⊆ gamma(P,s) }
In Caml: analyse : program → abstate
Case studies

The previous framework has been used to develop several analyses

1. A class analysis for a representative subset of bytecode Java [ESOP’04, TCS’04]
2. A memory usage analysis for a representative subset of bytecode Java [FM’05]
3. An interval analysis for the imperative fragment of bytecode Java [TCS’06]

But we are here a little too brave: termination is not mandatory to establish the soundness of an analysis.
Outline

1. Certified static analysis
   - Introduction
   - Building a certified static analyser

2. From certified static analysis to certified PCC

3. A case study: array-bound checks polyhedral analysis
   - Polyhedral abstract interpretation
   - Certified polyhedral abstract interpretation
   - Application: a polyhedral bytecode analyser
Checking a property with abstract interpretation

If we want to ensure that a program $P$ satisfies a property $\phi$

$$\llbracket P \rrbracket \subseteq \phi ?$$

1. we compute a post-fixpoint of $F^\#_P$ (over-approximation of $\llbracket P \rrbracket$)

$$\forall s^\#, F^\#_P(s^\#) \subseteq s^\# \Rightarrow \llbracket P \rrbracket \subseteq \gamma(s^\#)$$

2. we compute an under-approximation $\phi^\#$ of $\phi$

$$\gamma(\phi^\#) \subseteq \phi$$

3. we check that $\gamma(s^\#)$ implies $\gamma(\phi^\#)$ using an abstract order check

$$s^\# \sqsubseteq^\# \phi^\#$$

4. by transitivity we conclude $P$ satisfies $\phi$

$$\llbracket P \rrbracket \subseteq \gamma(s^\#) \subseteq \gamma(\phi^\#) \subseteq \phi$$
PCC by abstract interpretation

Producer

Consumer
PCC by abstract interpretation

From certified static analysis to certified PCC

Producer

Consumer

post-fixpoint

program

certificate verifier

Safe?

checks if $F^\sharp (pf) \subseteq pf$ and $pf \subseteq \phi^\sharp$
PCC by abstract interpretation

Producer

untrusted post-fixpoint solver

post-fixpoint

untrusted compressor

Consumer

certificate verifier

Safe?

computes $pf$ such that $F^\sharp (pf) \subseteq pf$ and $pf \subseteq \phi^\sharp$

checks if $F^\sharp (pf) \subseteq pf$ and $pf \subseteq \phi^\sharp$
Certified PCC by abstract interpretation

Producer

Consumer
Certified PCC by abstract interpretation

Producer

Consumer

- semantics + security policy
- Coq kernel + Coq extraction
Certified PCC by abstract interpretation

**Producer**
- certified verifier
  
**Consumer**
- certified verifier
  - (Coq file)
  - semantics + security policy
  - Coq kernel + Coq extraction
  - extracted certificate verifier
    
MARKER: checks if $F^\#(pf) \subseteq pf$ and $pf \subseteq \Phi$

Certified static analysis to certified PCC
Certified PCC by abstract interpretation

Producer

certified verifier

certified (post-fixpoint) verifier  
(Coq file)

Consumer

certified verifier

semantics + security policy

Coq kernel + Coq extraction

extracted certificate verifier

Safe ?

post-fixpoint

program

checks if $F^\sharp(pf) \subseteq pf$ and $pf \subseteq \Phi$
Certified PCC by abstract interpretation

Producer

- certified verifier
- untrusted post-fixpoint solver

Consumer

- certified verifier
- semantics + security policy
- Coq kernel + Coq extraction

(certified (post-fixpoint) verifier)

(cooked (Coq file))

(post-fixpoint)

computes \( p_f \) such that \( F^\sharp(p_f) \subseteq p_f \) and \( p_f \subseteq \phi \)

(post-fixpoint)

program

(post-fixpoint)

extracted certificate verifier

Safe?

checks if \( F^\sharp(p_f) \subseteq p_f \) and \( p_f \subseteq \phi \)
Certified PCC by abstract interpretation

Producer

certified verifier

untrusted post-fixpoint solver

post-fixpoint

untrusted compressor

Consumer

certified (post-fixpoint) verifier

(Coq file)

certified verifier

semantics + security policy

Coq kernel + Coq extraction

extracted certificate verifier

program

Safe?

computes $p_f$ such that $F^\#(p_f) \subseteq p_f$ and $p_f \subseteq \Phi$

checks if $F^\#(p_f) \subseteq p_f$ and $p_f \subseteq \Phi$
Outline

1. Certified static analysis
   - Introduction
   - Building a certified static analyser

2. From certified static analysis to certified PCC

3. A case study: array-bound checks polyhedral analysis
   - Polyhedral abstract interpretation
   - Certified polyhedral abstract interpretation
   - Application: a polyhedral bytecode analyser
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 to 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 $\mathbb{Q}^2$.

```plaintext
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 $\mathbb{Q}^2$.

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

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

x = x+1;
```
Polyhedral analysis

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

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

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

At junction point, we over approximate union by a convex union.
Polyhedral analysis

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

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

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

\[
x = x + 1;
\]

At junction point, we over approximate union by a convex union.
Polyhedral analysis

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

```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 = 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 $\mathbb{Q}^2$.

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

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

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

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

```
x = 0; y = 0;
{x ≤ 1 ∧ 0 ≤ y ≤ 2x}

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

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

```plaintext
x = 0; y = 0;
{x ≤ 1 ∧ 0 ≤ y ≤ 2x}

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

  x = x+1;
    {x = 1 ∧ 0 ≤ y ≤ 2}
}
```
Polyhedral analysis

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

\[
\begin{align*}
    x &= 0; \\ y &= 0; \\ &\{x \leq 1 \land 0 \leq y \leq 2x\}
\end{align*}
\]

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

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

\[
\begin{align*}
x = 0; & \quad y = 0; \\
\{x \leq 1 \land 0 \leq y \leq 2x\}
\end{align*}
\]

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

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

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

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

```
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\}
    };
    \{0 \leq x \leq 1 \land 0 \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 $\mathbb{Q}^2$.

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

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

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

\[\]
Polyhedral analysis

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

\begin{align*}
\text{x} &= 0; \quad \text{y} = 0; \\
\{ & \text{x} \leq 1 \land 0 \leq \text{y} \leq 2\text{x} \} \\
\lor & \{ \text{x} \leq 2 \land 0 \leq \text{y} \leq 2\text{x} \}
\end{align*}

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

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

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

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

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

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

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

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

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

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

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

\[
x = x+1; \\
\{0 \leq y \leq 2x \land 1 \leq x \leq 6\}
\]

By propagation we obtain a post-fixpoint

\[
\{0 \leq y \leq 2x \land 6 \leq x\}
\]
Polyhedral analysis

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

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

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\}
Polyhedral analysis

A more complex example.

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

\[
\text{while } (x < N) \ { \\
\text{if } (?) \ { \\
\{ A \leq y \leq 2x + A \land x \leq N - 1 \} \\
y = y + 2; \\
\{ A + 2 \leq y \leq 2x + A + 2 \land x \leq N - 1 \} \\
} \\
\{ A \leq y \leq 2x + A + 2 \land 0 \leq x \leq N - 1 \}
\]

\[
x = x + 1; \\
\{ A \leq y \leq 2x + A \land 1 \leq x \leq N \}
\]

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

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

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

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

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

$x = 0; \ y = 0; \ P_0 = \llbracket y := 0 \rrbracket \llbracket x := 0 \rrbracket (Q^2) \cup P_4$

while $(x<6)$ {
  if (?) {
    $P_1 = P_0 \cap \{x < 6\}$
    $y = y+2;$
    $P_2 = \llbracket y := y + 2 \rrbracket (P_1)$
  };
  $P_3 = P_1 \cup P_2$
  $x = x+1; \ P_4 = \llbracket x := x + 1 \rrbracket (P_3)$
} \n
$P_5 = P_0 \cap \{x \geq 6\}$
Library for manipulating polyhedra

- Parma Polyhedra Library\(^2\) (PPL), NewPolka: complex C/C++ libraries
- They rely on the Double Description Method
  - polyhedra are managed using two representations in parallel
    - by set of inequalities
      \[
P = \left\{ (x, y) \in \mathbb{Q}^2 \middle| \begin{array}{l} x \geq -1 \\ x - y \geq -3 \\ 2x + y \geq -2 \\ x + 2y \geq -4 \end{array} \right\}
    \]
    - by set of generators
      \[
P = \left\{ \lambda_1 s_1 + \lambda_2 s_2 + \lambda_3 s_3 + \mu_1 r_1 + \mu_2 r_2 \in \mathbb{Q}^2 \middle| \begin{array}{l} \lambda_1, \lambda_2, \lambda_3, \mu_1, \mu_2 \in \mathbb{R}^2 \\ \lambda_1 + \lambda_2 + \lambda_3 = 1 \end{array} \right\}
    \]
  - operations efficiency strongly depends on the chosen representations, so they keep both
  - We really don’t want this in a Trusted Computes Base!
  - But we really don’t want to certify this C/C++ libraries neither!

\(^2\)Previous tutorial on polyhedra partially comes from [http://www.cs.unipr.it/ppl/](http://www.cs.unipr.it/ppl/)
Outline

1 Certified static analysis
   • Introduction
   • Building a certified static analyser

2 From certified static analysis to certified PCC

3 A case study: array-bound checks polyhedral analysis
   • Polyhedral abstract interpretation
   • Certified polyhedral abstract interpretation
   • Application: a polyhedral bytecode analyser
Polyhedra in a PCC framework
Join work with F. Besson, T. Jensen and T. Turpin

Develop a checker of analysis results
▶ minimize the number of operations to certify
▶ avoid (some of the most) costly operations

The checker will receive a post-fixpoint + a certificate of certain polyhedra inclusions to be verified by the checker

We develop one checker for a rich abstract domain based on Farkas lemma

Can accommodate invariants that are obtained
▶ automatically (intervals, polyhedra,...)
▶ by user-annotation (polynomials, ...)

David Pichardie
A minimal polyhedral tool-kit

For efficiency and simplicity,

- Polyhedra are represented in constraint form prefixed by existentially quantified variables
- Constraints are never normalised

Abstract operators are much simpler:

- Assignments do not trigger quantifier elimination:
  \[
  \llbracket x := e \rrbracket(P) = \exists x', P[x'/x] \land x = e[x'/x]
  \]

- Intersection is just syntactic union of constraints;
- (Over-approximations) of Convex Hulls are given as untrusted invariants:

  \[
  isUpperBound(P, Q, UB) \equiv P \subseteq UB \land Q \subseteq UB
  \]

- Polyhedra inclusion is guided by a certificate:

  \[
  isIncluded(P, Q, Cert) \Rightarrow P \subseteq Q
  \]
Certified PCC by abstract interpretation

Producer

untrusted post-fixpoint solver

untrusted compressor

post-fixpoint

certified verifier

Consumer

certified verifier

(Coq file)

certified (post-fixpoint) verifier

post-fixpoint

program

extracted certificate verifier

Safe?

Coq kernel + Coq extraction

semantics + security policy
Certified PCC by abstract interpretation

Producers

Certified verifier

Untrusted post-fixpoint solver

Post-fixpoint

Untrusted compressor

Post-fixpoint

Inclusion certificates

Program

Certified (post-fixpoint) verifier

Consumer

Certified verifier

Semantics + security policy

Coq kernel + Coq extraction

Extracted certificate verifier

Safe?
Checking polyhedra inclusion using certificates

- Inclusion reduces to a conjunction of emptiness problems

\[
P \subseteq \{ q_1 \geq c_1, \ldots, q_m \geq c_m \}
\]

if and only if

\[
P \cup \{-q_1 \geq -c_1 + 1\} = \emptyset \land \ldots \land P \cup \{-q_m \geq -c_m + 1\} = \emptyset
\]

- Each emptiness reduces to unsatisfiability of linear constraints

\[
\forall x_1, \ldots, x_n, \neg \left( \begin{array}{c}
a_{1,1}, \ldots, a_{1,n} \\
\vdots \\
a_{m,1}, \ldots, a_{m,n}
\end{array} \right) \cdot \left( \begin{array}{c}
x_1 \\
\vdots \\
x_n
\end{array} \right) \geq \left( \begin{array}{c}
b_1 \\
\vdots \\
b_m
\end{array} \right)
\]
Unsatisfiability certificates

Lemma (Farkas’s Lemma (Variant))

Let \( A \in \mathbb{Q}^{m \times n} \) and \( b \in \mathbb{Q}^n \).

\[
\forall x \in \mathbb{Q}^n, \neg(A \cdot x \geq b)
\]

if and only if

\[
\exists (cert \in \mathbb{Q}^m), cert \geq \bar{0}, \text{ such that } \begin{cases} A^t \cdot cert = \bar{0} \\ b^t \cdot cert > 0 \end{cases}
\]

Soundness of certificates is easy (\( \Leftarrow \))

Démonstration.

Suppose \( A \cdot x \geq b \).

Since \( cert \geq \bar{0} \) we have \( (A \cdot x)^t \cdot cert \geq b^t \cdot cert \).

Now \( x^t \cdot (A^t \cdot cert) = (x^t \cdot A^t) \cdot cert = (A \cdot x)^t \cdot cert \).

Hence \( x^t \cdot (A^t \cdot cert) \geq b^t \cdot cert \).

Therefore \( x^t\bar{0} = 0 \geq b^t \cdot cert > 0 \rightarrow \text{ contradiction.} \)
Certificate checking

**Example**

Using the certificate $cert = (1; 1; 5)$, check that
\[
\begin{pmatrix}
1 & 1 \\
-1 & 4 \\
0 & -1
\end{pmatrix}
\cdot
\begin{pmatrix}
x \\
y
\end{pmatrix}
\geq
\begin{pmatrix}
2 \\
1 \\
1
\end{pmatrix}
\]
has no solutions.

**Checking algorithm.**

- Check
  \[
  \begin{pmatrix}
  1 & 1 \\
  -1 & 4 \\
  0 & 1
  \end{pmatrix}^t
  \cdot
  \begin{pmatrix}
  1 \\
  1 \\
  5
  \end{pmatrix}
  =
  \begin{pmatrix}
  0 \\
  0
  \end{pmatrix}
  \]

- Check
  \[
  \begin{pmatrix}
  2 \\
  1
  \end{pmatrix}^t
  \cdot
  \begin{pmatrix}
  1 \\
  1 \\
  5
  \end{pmatrix}
  > 0.
  \]

Checking time complexity is quadratic (matrix-vector product).
Certificate generation by linear programming

Let $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^n$, the set of unsatisfiability certificates is defined as

$$\text{Cert} = \left\{ c \mid \begin{array}{c} c \geq 0 \\ b^t \cdot c > 0 \\ A^t \cdot c = 0 \end{array} \right\}$$

Finding an extremal certificate is a linear programming problem

$$\min \{ c^t \cdot \vec{1} \mid c \in \text{Cert} \}$$

that can be solved

- Over $\mathbb{N}$, by linear integer programming algorithms (Bad complexity, smallest certificate)
- Over $\mathbb{Q}$, by the Simplex (or interior point methods) (Good complexity and small certificate – in practise)
Outline

1. Certified static analysis
   - Introduction
   - Building a certified static analyser

2. From certified static analysis to certified PCC

3. A case study: array-bound checks polyhedral analysis
   - Polyhedral abstract interpretation
   - Certified polyhedral abstract interpretation
   - Application: a polyhedral bytecode analyser
Application: a polyhedral bytecode analyser

We have applied this technique for a Java-like bytecode language with

- (unbounded) integers,
- dynamically created (unidimensional) array of integers,
- static methods (procedures),
- static fields (global variables).

Linear invariant are used to statically checks that all array accesses are within bounds.

It allows to remove the dynamic check used by standard JVM without risk of buffer overflow attack.

In practice we could only try to detect statically some valid array accesses and keep dynamic checks for the other accesses.
Example: binary search

```java
static int bsearch(int key, int[] vec) {
    int low = 0, high = vec.length - 1;

    while (0 < high-low) {
        int mid = (low + high) / 2;

        if (key == vec[mid]) return mid;
        else if (key < vec[mid]) high = mid - 1;
        else low = mid + 1;
    }

    return -1;
}
```
Example: binary search

```java
// PRE: 0 ≤ |vec0|
static int bsearch(int key, int[] vec) {
    // (I1) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ |vec0|
    int low = 0, high = vec.length - 1;
    // (I2) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec0|
    while (0 < high-low) {
        // (I3) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low < high < |vec0|
        int mid = (low + high) / 2;
        // (I4) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low < high < |vec0| ∧ low+high-1 ≤ 2·mid ≤ low+high
        if (key == vec[mid]) return mid;
        else if (key < vec[mid]) high = mid - 1;
        else low = mid + 1;
        // (I5) key0 = key ∧ |vec0| = |vec| ∧ −2+3·low ≤ 2·high + mid ∧ −1+2·low ≤ high ∧ mid ∧ −1+low ≤ mid ≤ 1+high ∧ high ≤ low + mid ∧ 1+high ≤ 2·low + mid ∧ 1+low + |vec0| + high ∧ 2 ≤ |vec0| ∧ 2 + high + mid ≤ |vec0| + low
    }
    // (I6) key0 = key ∧ |vec0| = |vec| ∧ low − 1 ≤ high ∧ 0 ≤ low ∧ high < |vec0|
    return -1;
} // POST: −1 ≤ res < |vec0|
```

This is a correct post-fixpoint but there is too many informations (too precise)!
Example: binary search

```java
// PRE: True
static int bsearch(int key, int[] vec) {
    // (I_1) |vec0| = |vec| ∧ 0 ≤ |vec0|
    int low = 0, high = vec.length - 1;
    // (I_2) |vec0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec0|
    while (0 < high-low) {
        // (I_3) |vec0| = |vec| ∧ 0 ≤ low < high < |vec0|
        int mid = (low + high) / 2;
        // (I_4) |vec| − |vec0| = 0 ∧ low ≥ 0 ∧ mid − low ≥ 0∧
        // 2 · high − 2 · mid − 1 ≥ 0 ∧ |vec0| − high − 1 ≥ 0
        if (key == vec[mid]) return mid;
        else if (key < vec[mid]) high = mid - 1;
        else low = mid + 1;
        // (I_5) |vec0| = |vec| ∧ −1 + low ≤ high ∧ 0 ≤ low ∧ 5 + 2 · high ≤ 2 · |vec|
    }
    // (I_6) 0 ≤ |vec0|
    return -1;
} // POST: −1 ≤ res < |vec0|
```

This one is less precise but sufficient to ensure the security policy.
Some preliminary benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>.class</th>
<th>certificates before</th>
<th>certificates after</th>
<th>checking time before</th>
<th>checking time after</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSearch</td>
<td>515</td>
<td>22</td>
<td>12</td>
<td>0.005</td>
<td>0.007</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>528</td>
<td>15</td>
<td>14</td>
<td>0.0005</td>
<td>0.0003</td>
</tr>
<tr>
<td>HeapSort</td>
<td>858</td>
<td>72</td>
<td>32</td>
<td>0.053</td>
<td>0.025</td>
</tr>
<tr>
<td>QuickSort</td>
<td>833</td>
<td>87</td>
<td>44</td>
<td>0.54</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Class files are given in bytes, certificates in number of constraints, time in seconds. The two checking times in the last column give the checking time with and without fixpoint pruning.
Foudational PCC by reflection

The generated certificate is a compressed post-fixpoint
  ▶ small certificate,
  ▶ but very adhoc checker.

In Foudational PCC, you want to obtain a machine-checked proof of Safe(p)
  ▶ general checker (as Coq)
  ▶ but the proof λ-term may be bigger than the adhoc certificate.

This can be done using reflection because we prove

```
checker_correct :
  \forall (p : program) (cert : positive), checker p cert = true → safe p
```

With a foudational proof of the same size as the adhoc certificate!

```
checker_correct prog cert (refl_equal true)
```