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

Outline

Outline

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

Outline

Outline

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

$\dot{\ddot{\alpha}}[\![P]\!](\text{Post}[\![\texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}]\!])$

$=$ $\wr$def. (110) of $\dot{\ddot{\alpha}}[\![P]\!]\wr$

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

$=$ $\wr$def. (103) of Post$\wr$

$\ddot{\alpha}[\![P]\!] \circ \text{post}[\tau^\star[\![\texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}]\!]] \circ \ddot{\gamma}[\![P]\!]$

$=$ $\wr$big step operational semantics (93)$\wr$

$\ddot{\alpha}[\![P]\!] \circ \text{post}\,[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t) \cup (1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)] \circ \ddot{\gamma}[\![P]\!]$

$=$ $\wr$Galois connection (98) so that post preserves joins$\wr$

$\ddot{\alpha}[\![P]\!] \circ (\text{post}\,[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t)] \mathbin{\dot{\cup}} \text{post}[(1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)]) \circ \ddot{\gamma}[\![P]\!]$

$=$ $\wr$Galois connection (106) so that $\ddot{\alpha}[\![P]\!]$ preserves joins$\wr$

$(\ddot{\alpha}[\![P]\!] \circ \text{post}[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t)] \circ \ddot{\gamma}[\![P]\!]) \mathbin{\dot{\ddot{\cup}}} (\ddot{\alpha}[\![P]\!] \circ \text{post}[(1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)] \circ \ddot{\gamma}[\![P]\!])$

$\dot{\ddot{\sqsubseteq}}$ $\wr$lemma (5.3) and similar one for the **else** branch$\wr$

$$\lambda J \cdot \text{let } J^{t'} = \lambda l \in \text{in}_P[\![P]\!] \cdot (l = \text{at}_P[\![S_t]\!] \;?\; J_{\text{at}_P[\![S_t]\!]} \mathbin{\dot{\cup}} \text{Abexp}[\![B]\!](J_\ell) \;¿\; J_l) \text{ in} \tag{120}$$

$\quad \text{let } J^{t''} = \text{APost}[\![S_t]\!](J^{t'}) \text{ in}$

$\quad\quad \lambda l \in \text{in}_P[\![P]\!] \cdot (l = \ell' \;?\; J^{t''}_{\ell'} \mathbin{\dot{\cup}} J^{t''}_{\text{after}_P[\![S_t]\!]} \;¿\; J^{t''}_l)$

$\quad \dot{\ddot{\cup}}$

$\quad \text{let } J^{f'} = \lambda l \in \text{in}_P[\![P]\!] \cdot (l = \text{at}_P[\![S_f]\!] \;?\; J_{\text{at}_P[\![S_f]\!]} \mathbin{\dot{\cup}} \text{Abexp}[\![T(\neg B)]\!](J_\ell) \;¿\; J_l) \text{ in}$

$\quad \text{let } J^{f''} = \text{APost}[\![S_f]\!](J^{f'}) \text{ in}$

$\quad\quad \lambda l \in \text{in}_P[\![P]\!] \cdot (l = \ell' \;?\; J^{f''}_{\ell'} \mathbin{\dot{\cup}} J^{f''}_{\text{after}_P[\![S_f]\!]} \;¿\; J^{f''}_l)$

$=$ $\wr$by grouping similar terms$\wr$

$\lambda J \cdot \text{let } J^{t'} = \lambda l \in \text{in}_P[\![P]\!] \cdot (l = \text{at}_P[\![S_t]\!] \;?\; J_{\text{at}_P[\![S_t]\!]} \mathbin{\dot{\cup}} \text{Abexp}[\![B]\!](J_\ell) \;¿\; J_l)$

$\quad \text{and } J^{f'} = \lambda l \in \text{in}_P[\![P]\!] \cdot (l = \text{at}_P[\![S_f]\!] \;?\; J_{\text{at}_P[\![S_f]\!]} \mathbin{\dot{\cup}} \text{Abexp}[\![T(\neg B)]\!](J_\ell) \;¿\; J_l) \text{ in}$

$\quad \text{let } J^{t''} = \text{APost}[\![S_t]\!](J^{t'})$

$\quad \text{and } J^{f''} = \text{APost}[\![S_f]\!](J^{f'}) \text{ in}$

$\quad\quad \lambda l \in \text{in}_P[\![P]\!] \cdot (l = \ell' \;?\; J^{t''}_{\ell'} \mathbin{\dot{\cup}} J^{t''}_{\text{after}_P[\![S_t]\!]} \mathbin{\dot{\cup}} J^{f''}_{\ell'} \mathbin{\dot{\cup}} J^{f''}_{\text{after}_P[\![S_f]\!]} \;¿\; J^{t''}_l \mathbin{\dot{\cup}} J^{f''}_l)$

$=$ $\wr$by locality (113) and labelling scheme (59) so that in particular $J^{t''}_{\ell'} = J^{t'}_{\ell'} = J^t_{\ell'} = J^f_{\ell'} = J^{f'}_{\ell'} = J^{f''}_{\ell'}$ and APost$[\![S_t]\!]$ and APost$[\![S_f]\!]$ do not interfere$\wr$

Proof

$\dot{\ddot{\alpha}}[\![P]\!](\mathrm{Post}[\![\texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}]\!])$

$=$ $\wr$def. (110) of $\dot{\ddot{\alpha}}[\![P]\!]$$\wr$

$\ddot{\alpha}[\![P]\!] \circ \mathrm{Post}[\![\texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}]\!] \circ \ddot{\gamma}[\![P]\!]$

$=$ $\wr$def. (103) of Post$\wr$

$\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[\tau^\star[\![\texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}]\!]] \circ \ddot{\gamma}[\![P]\!]$

$=$ $\wr$big step operational semantics (93)$\wr$

$\ddot{\alpha}[\![P]\!] \circ \mathrm{post}\,[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t) \cup (1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)] \circ \ddot{\gamma}[\![P]\!]$

$=$ $\wr$Galois connection (98) so that post preserves joins$\wr$

$\ddot{\alpha}[\![P]\!] \circ (\mathrm{post}\,[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t)] \,\dot{\cup}\, \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)]) \circ \ddot{\gamma}[\![P]\!]$

$=$ $\wr$Galois connection (106) so that $\ddot{\alpha}[\![P]\!]$ preserves joins$\wr$

$(\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t)] \circ \ddot{\gamma}[\![P]\!]) \;\dot{\ddot{\cup}}\; (\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)] \circ \ddot{\gamma}[\![P]\!])$

$\dot{\ddot{\subseteq}}$ $\wr$lemma (5.3) and similar one for the **else** branch$\wr$

$\lambda J \cdot \mathrm{let}\ J^{t'} = \lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \mathrm{at}_P[\![S_t]\!]\ ?\ J_{\mathrm{at}_P[\![S_t]\!]} \,\dot{\cup}\, \mathrm{Abexp}[\![B]\!](J_\ell) \;¿\; J_l)$ in (120)

$\quad \mathrm{let}\ J^{t''} = \mathrm{APost}[\![S_t]\!](J^{t'})$ in

$\quad\quad \lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \ell'\ ?\ J^{t''}_{\ell'} \,\dot{\cup}\, J^{t''}_{\mathrm{after}_P[\![S_t]\!]} \;¿\; J^{t''}_l)$

$\quad \ddot{\dot{\cup}}$

$\quad \mathrm{let}\ J^{f'} = \lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \mathrm{at}_P[\![S_f]\!]\ ?\ J_{\mathrm{at}_P[\![S_f]\!]} \,\dot{\cup}\, \mathrm{Abexp}[\![T(\neg B)]\!](J_\ell) \;¿\; J_l)$ in

$\quad\quad \mathrm{let}\ J^{f''} = \mathrm{APost}[\![S_f]\!](J^{f'})$ in

$\quad\quad\quad \lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \ell'\ ?\ J^{f''}_{\ell'} \,\dot{\cup}\, J^{f''}_{\mathrm{after}_P[\![S_f]\!]} \;¿\; J^{f''}_l)$

$=$ $\wr$by grouping similar terms$\wr$

$\lambda J \cdot \mathrm{let}\ \ J^{t'} = \lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \mathrm{at}_P[\![S_t]\!]\ ?\ J_{\mathrm{at}_P[\![S_t]\!]} \,\dot{\cup}\, \mathrm{Abexp}[\![B]\!](J_\ell) \;¿\; J_l)$

$\quad \mathrm{and}\ J^{f'} = \lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \mathrm{at}_P[\![S_f]\!]\ ?\ J_{\mathrm{at}_P[\![S_f]\!]} \,\dot{\cup}\, \mathrm{Abexp}[\![T(\neg B)]\!](J_\ell) \;¿\; J_l)$ in

$\quad\quad \mathrm{let}\ \ J^{t''} = \mathrm{APost}[\![S_t]\!](J^{t'})$

$\quad\quad \mathrm{and}\ J^{f''} = \mathrm{APost}[\![S_f]\!](J^{f'})$ in

$\quad\quad\quad \lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \ell'\ ?\ J^{t''}_{\ell'} \,\dot{\cup}\, J^{t''}_{\mathrm{after}_P[\![S_t]\!]} \,\dot{\cup}\, J^{f''}_{\ell'} \,\dot{\cup}\, J^{f''}_{\mathrm{after}_P[\![S_f]\!]} \;¿\; J^{t''}_l \,\dot{\cup}\, J^{f''}_l)$

$=$ $\wr$by locality (113) and labelling scheme (59) so that in particular $J^{t''}_{\ell'} = J^{t'}_{\ell'} = J^t_{\ell'} = J^f_{\ell'} = J^{f'}_{\ell'} = J^{f''}_{\ell'}$ and APost$[\![S_t]\!]$ and APost$[\![S_f]\!]$ do not interfere$\wr$

Implementation

```
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->nbcolumns = 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);
    }}
}
```

Proof

$\dot{\ddot{\alpha}}[\![P]\!](\mathrm{Post}[\![\texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}]\!])$

$=$ ⦇def. (110) of $\dot{\ddot{\alpha}}[\![P]\!]$⦈

$\ddot{\alpha}[\![P]\!] \circ \mathrm{Post}[\![\texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}]\!] \circ \ddot{\gamma}[\![P]\!]$

$=$ ⦇def. (103) of Post⦈

$\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[\tau^\star[\![\texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}]\!]] \circ \ddot{\gamma}[\![P]\!]$

$=$ ⦇big step operational semantics (93)⦈

$\ddot{\alpha}[\![P]\!] \circ \mathrm{post}\,[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t) \cup (1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)] \circ \ddot{\gamma}[\![P]\!]$

$=$ ⦇Galois connection (98) so that post preserves joins⦈

$\ddot{\alpha}[\![P]\!] \circ (\mathrm{post}\,[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t)] \mathbin{\dot{\cup}} \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)]) \circ \ddot{\gamma}[\![P]\!]$

$=$ ⦇Galois connection (106) so that $\ddot{\alpha}[\![P]\!]$ preserves joins⦈

$(\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t)] \circ \ddot{\gamma}[\![P]\!]) \mathbin{\dot{\ddot{\cup}}} (\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)] \circ \ddot{\gamma}[\![P]\!])$

$\dot{\ddot{\sqsubseteq}}$ ⦇lem…

$\lambda J \cdot$ let …

let …

$\lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \ell' \mathbin{?} J^{t''}_{\ell'} \mathbin{\dot{\cup}} J^{t''}_{\mathrm{after}_P[\![S_t]\!]} \mathbin{¿} J^{t''}_l)$

$\dot{\ddot{\cup}}$

let $J^{f'} = \lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \mathrm{at}_P[\![S_f]\!] \mathbin{?} J_{\mathrm{at}_P[\![S_f]\!]} \mathbin{\dot{\cup}} \mathrm{Abexp}[\![T(\neg B)]\!](J_\ell) \mathbin{¿} J_l)$ in

let $J^{f''} = \mathrm{APost}[\![S_f]\!](J^{f'})$ in

$\lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \ell' \mathbin{?} J^{f''}_{\ell'} \mathbin{\dot{\cup}} J^{f''}_{\mathrm{after}_P[\![S_f]\!]} \mathbin{¿} J^{f''}_l)$

$=$ ⦇by grouping similar terms⦈

$\lambda J \cdot$ let $J^{t'} = \lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \mathrm{at}_P[\![S_t]\!] \mathbin{?} J_{\mathrm{at}_P[\![S_t]\!]} \mathbin{\dot{\cup}} \mathrm{Abexp}[\![B]\!](J_\ell) \mathbin{¿} J_l)$

and $J^{f'} = \lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \mathrm{at}_P[\![S_f]\!] \mathbin{?} J_{\mathrm{at}_P[\![S_f]\!]} \mathbin{\dot{\cup}} \mathrm{Abexp}[\![T(\neg B)]\!](J_\ell) \mathbin{¿} J_l)$ in

let $J^{t''} = \mathrm{APost}[\![S_t]\!](J^{t'})$

and $J^{f''} = \mathrm{APost}[\![S_f]\!](J^{f'})$ in

$\lambda l \in \mathrm{in}_P[\![P]\!] \cdot (l = \ell' \mathbin{?} J^{t''}_{\ell'} \mathbin{\dot{\cup}} J^{t''}_{\mathrm{after}_P[\![S_t]\!]} \mathbin{\dot{\cup}} J^{f''}_{\ell'} \mathbin{\dot{\cup}} J^{f''}_{\mathrm{after}_P[\![S_f]\!]} \mathbin{¿} J^{t''}_l \mathbin{\dot{\cup}} J^{f''}_l)$

$=$ ⦇by locality (113) and labelling scheme (59) so that in particular $J^{t''}_{\ell'} = J^{t'}_{\ell'} = J^{t}_{\ell'} = J^{f}_{\ell'} = J^{f'}_{\ell'} = J^{f''}_{\ell'}$ and $\mathrm{APost}[\![S_t]\!]$ and $\mathrm{APost}[\![S_f]\!]$ do not interfere⦈

Implementation

```
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->nbcolumns = 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;
```

```
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);
    }}
}
```


Do the two parts talk about the same ?

Certified static analyses

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

Kernel ⟶ Proof assistant ⟶ Static analysis ⟶

- 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

[Cousot&Cousot 75, 76, 77, 79, 80, 81, 82, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 00, 01, 02, 03, 04, 05, 06, 07,. . .][1]

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.

[1] See `http://www.di.ens.fr/~cousot/`

Abstract Interpretation

[Cousot&Cousot 75, 76, 77, 79, 80, 81, 82, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 00, 01, 02, 03, 04, 05, 06, 07,. . .][1]

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.

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

[1] See `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 $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.

```
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 $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.

```
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 $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.

```
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 $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.

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

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of values.

Collecting semantics

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

```
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 $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.

```
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.

```
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.

```
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 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.

```
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 values.

Collecting semantics

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

```
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 $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.

```
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 $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.

```
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 $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.

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

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

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

    y = y+2;

  };

  x = x+1;

}
```

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 $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.

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

  };

  x = x+1;

}
```

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

}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of 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 = 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.

```
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 $(\mathtt{x}, \mathtt{y})$ 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 ⩾ 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 $(\mathtt{x}, \mathtt{y})$ 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 ⩾ 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.

```
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 $(\mathtt{x}, \mathtt{y})$ 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 ⩾ 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 $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.

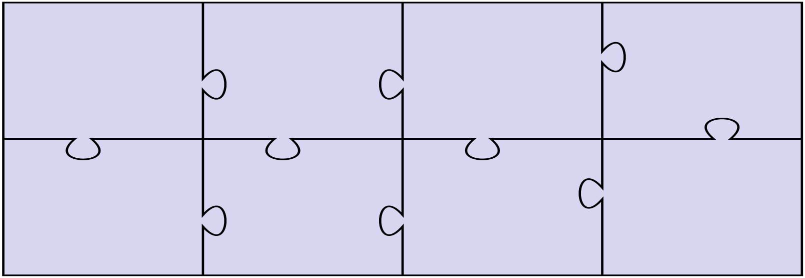
Approximation

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

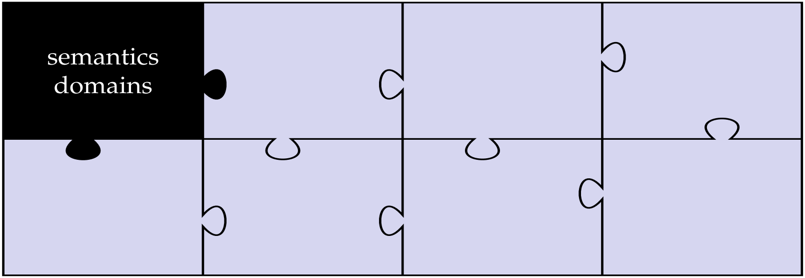
Outline

Building a certified static analyser

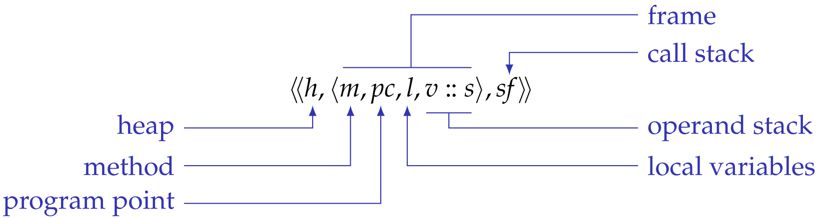


- A puzzle with 8 pieces,
- Each piece interacts with its neighbors

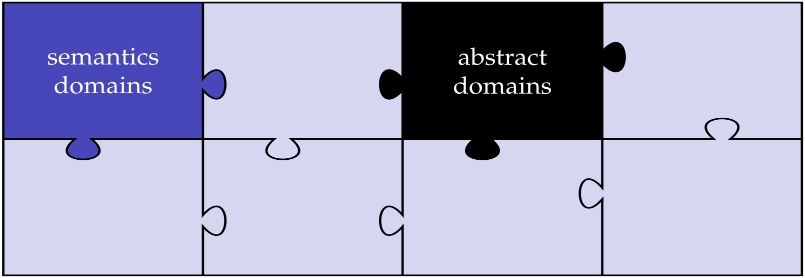
Building a certified static analyser



Example : JVM states

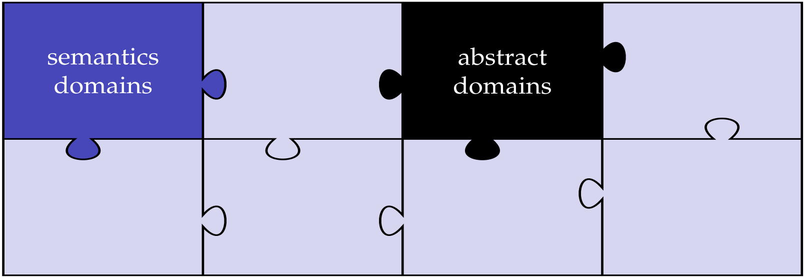


Building a certified static analyser



- Each semantic sub-domain has its abstract counterpart
- An abstract domain is a lattice $(\mathcal{D}^\sharp, =, \sqsubseteq, \bot, \sqcup, \sqcap)$ without infinite strictly increasing chains $x_0 \sqsubset x_1 \sqsubset \cdots \sqsubset \cdots$
- 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 $\left(\mathcal{D}^\sharp, =, \sqsubseteq, \bot, \sqcup, \sqcap\right)$ without infinite strictly increasing chains $x_0 \sqsubset x_1 \sqsubset \cdots \sqsubset \cdots$
- 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 (⊑) 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 *)




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

```
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) ∧ P2.eq (snd x) (snd y)
   with Definition order := fun x y : (P1.t * P2.t) =>
    P1.order (fst x) (fst y) ∧ 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 :

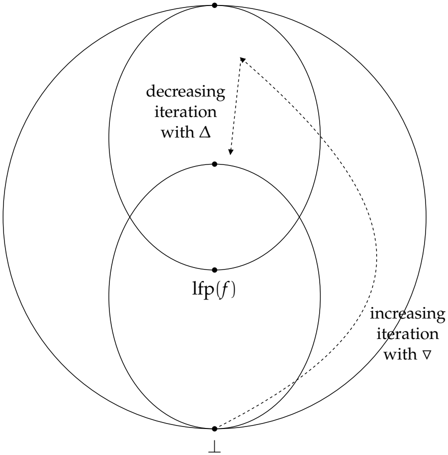
```
Module AbSt  :=
  Product (Array (Array (List (Sum FiniteSet Constant))))
  (Product (Array (Array (Array (List (Sum FiniteSet Constant)))))
           (Array (Array (List (Sum FiniteSet Constant)))))
```

post-fixpoint computation by widening/narrowing ∇/Δ

[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

$$\mathrm{lfp}(f) \sqsubseteq a' \sqsubseteq a$$



Building lattices in Coq

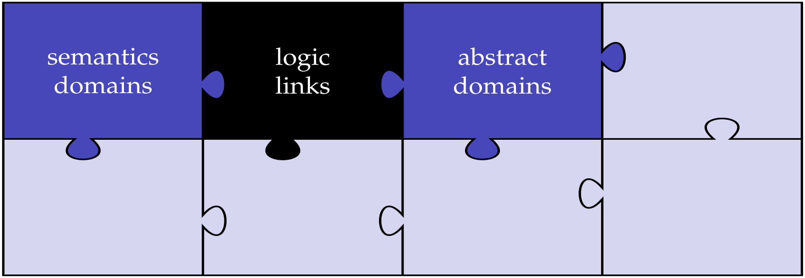
We propose a technique based on the new Coq module system (inspired by the ML module system)

- 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 :

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

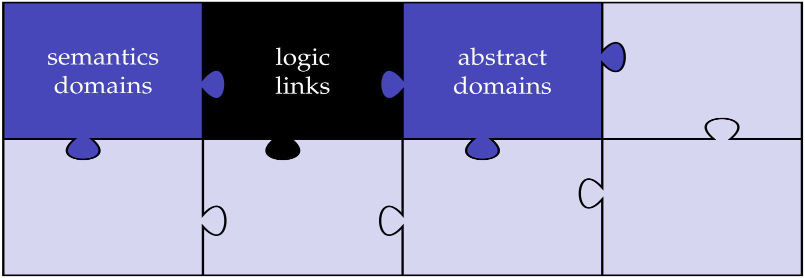
Building a certified static analyser



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

$$\gamma : (\mathcal{D}^\sharp, \sqsubseteq) \longrightarrow_m (\wp(\mathcal{D}), \subseteq)$$

Building a certified static analyser

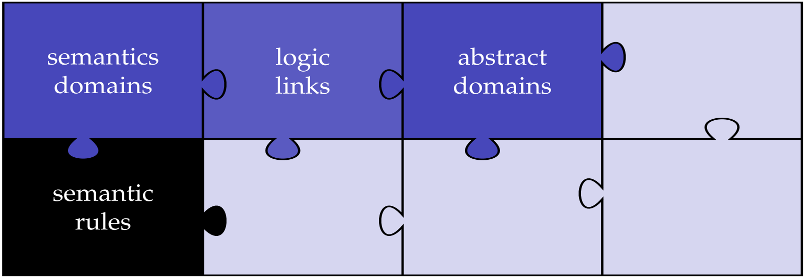


- Each abstract value represents a property on concrete values
- This correspondence is formalised by a monotone concretisation function

$$\gamma : (\mathcal{D}^\sharp, \sqsubseteq) \longrightarrow_m (\wp(\mathcal{D}), \subseteq)$$

$x \subseteq \gamma(x^\sharp)$ means "$x^\sharp$ is a correct approximation of x"

Building a certified static analyser



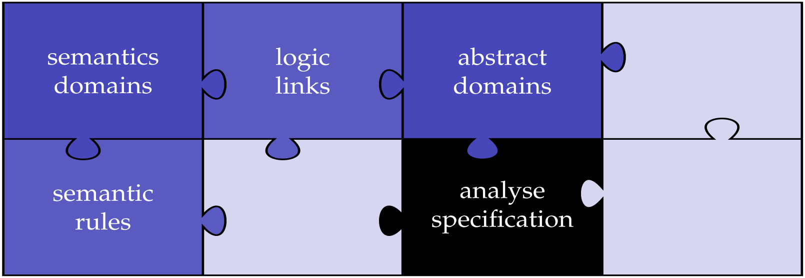
- operational semantics $\cdot \rightarrow_P \cdot$ between states
- collecting semantics : $[\![P]\!] = \{ s \mid \exists s_0 \in S_{\mathrm{init}},\ s_0 \rightarrow_P^* s \}$
- we want to compute a correct approximation of $[\![P]\!]$
  - a sound invariant $s^\sharp$ on the reachable states : $[\![P]\!]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

Example : JVM operational semantics

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

$$\frac{\begin{array}{c}\text{instructionAt}_P(m,pc) = \texttt{invokevirtual}\ m_{id} \\ \begin{array}{rcl} m' & = & \text{methodLookup}(m_{id}, h(loc)) \\ V & = & v_1 :: \cdots :: v_{\text{nbArguments}(m_{id})}\end{array}\end{array}}{\langle\langle h, \langle m, pc, l, loc :: V :: s\rangle, sf\rangle\rangle \rightarrow \langle\langle h, \langle m', 1, V, \varepsilon\rangle, \langle m, pc, l, s\rangle :: sf\rangle\rangle}$$

Building a certified static analyser



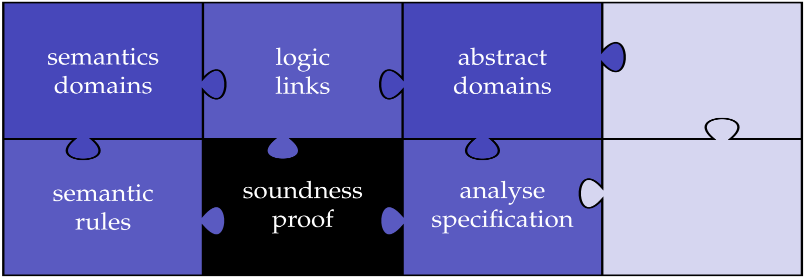
- the analysis is specified as a solution of a post fixpoint problem

$$F_P^\sharp(s^\sharp) \sqsubseteq^\sharp s^\sharp$$

- after partitioning : constraint system

$$\left\{ \begin{array}{ccc} f_1^\sharp(s_1^\sharp, \ldots, s_n^\sharp) & \sqsubseteq^\sharp & s_{i_1}^\sharp \\ \ldots & & \\ f_n^\sharp(s_1^\sharp, \ldots, s_n^\sharp) & \sqsubseteq^\sharp & s_{i_n}^\sharp \end{array} \right.$$

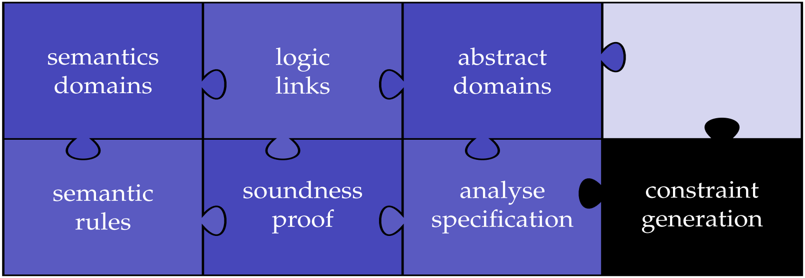
Building a certified static analyser



$$\forall P, \forall s^\sharp, \quad F_P^\sharp(s^\sharp) \sqsubseteq^\sharp s^\sharp \Rightarrow [\![P]\!] \subseteq \gamma(s^\sharp)$$

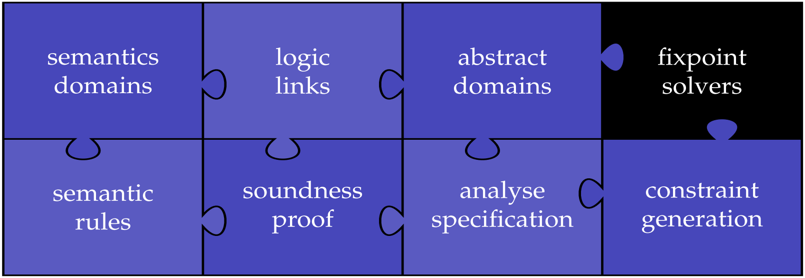
- easy proof, but tedious
- one proof by instruction : a long work for real langages

Building a certified static analyser



- collects all constraints in a program
- generic tool

Building a certified static analyser



$$\forall P,\ \exists s^\sharp,\quad F^\sharp_P(s^\sharp) \sqsubseteq s^\sharp$$

Two techniques of iterative computation

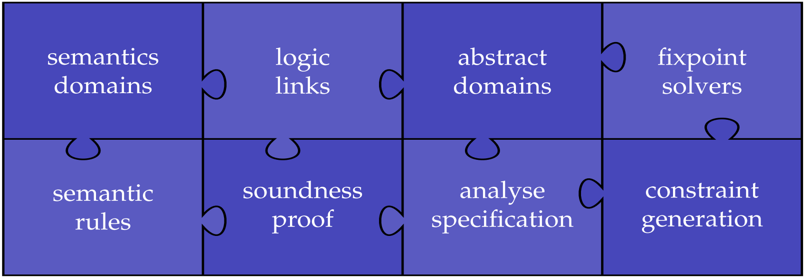
- traditional least (post)-fixpoint computation

$$\bot \rightarrow F^\sharp_P(\bot) \rightarrow {F^\sharp_P}^2(\bot) \rightarrow \cdots \mathrm{lfp}(F^\sharp_P)$$

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

In the two cases, a generic tool

Building a certified static analyser



Final result

$$\left.\begin{array}{l}\forall P,\ \forall s^\sharp,\ F^\sharp_P(s^\sharp) \sqsubseteq s^\sharp \Rightarrow [\![P]\!] \subseteq \gamma(s^\sharp) \\ \forall P,\ \exists s^\sharp,\ F^\sharp_P(s^\sharp) \sqsubseteq s^\sharp\end{array}\right\}\quad \forall P,\ \exists s^\sharp,\ [\![P]\!] \subseteq \gamma(s^\sharp)$$

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

Checking a property with abstract interpretation

If we want to ensure that a program P satisfies a property ϕ

$$[\![P]\!] \subseteq \phi \ ?$$

1. we compute a post-fixpoint of $F_P^\sharp$ (over-approximation of $[\![P]\!]$)

$$\forall s^\sharp,\ F_P^\sharp(s^\sharp) \sqsubseteq s^\sharp \Rightarrow [\![P]\!] \subseteq \gamma(s^\sharp)$$

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

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

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

$$s^\sharp \sqsubseteq^\sharp \phi^\sharp$$

4. by transitivity we conclude P satisfies ϕ

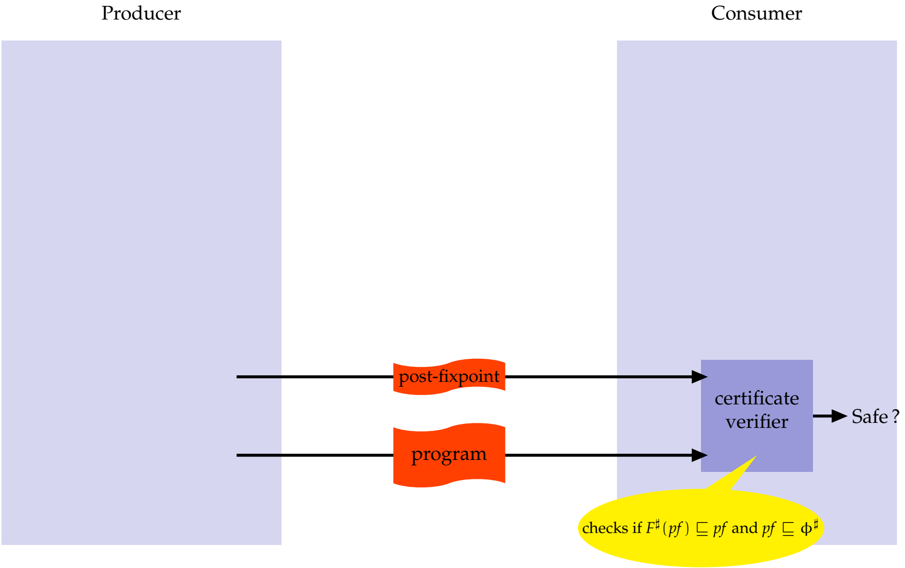
$$[\![P]\!] \subseteq \gamma(s^\sharp) \subseteq \gamma(\phi^\sharp) \subseteq \phi$$

PCC by abstract interpretation

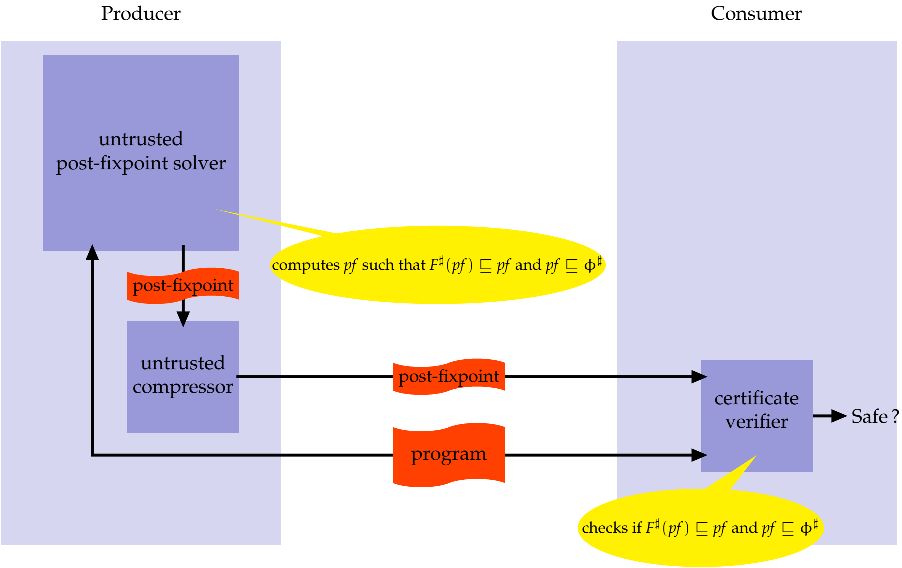
Producer

Consumer

PCC by abstract interpretation



PCC by abstract interpretation

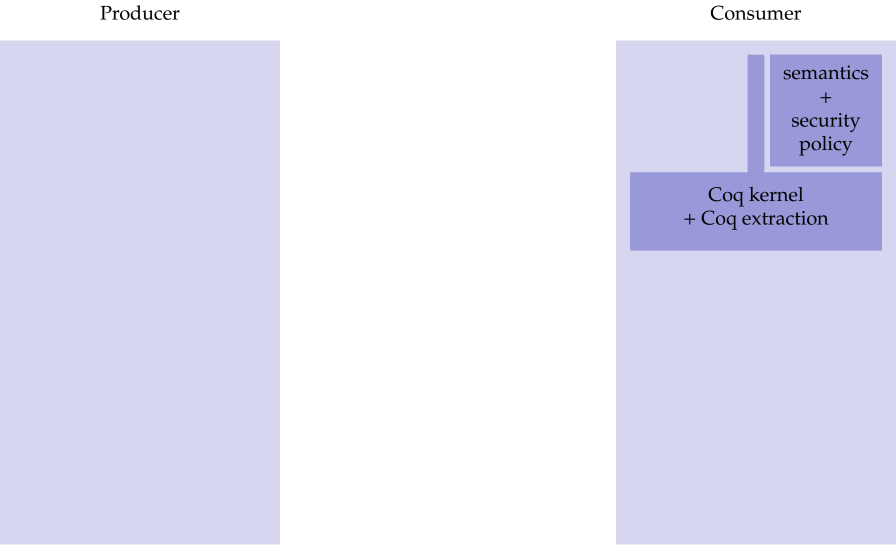


Certified PCC by abstract interpretation

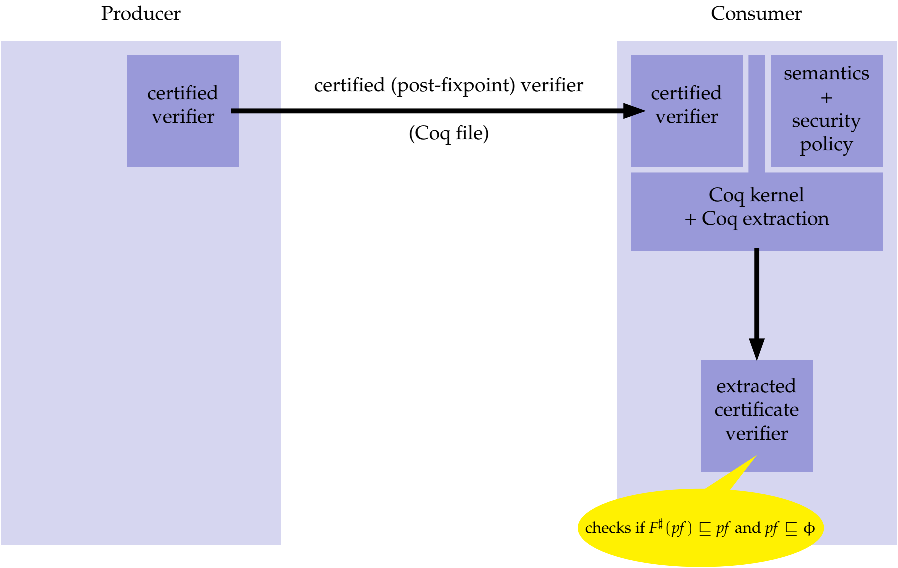
Producer

Consumer

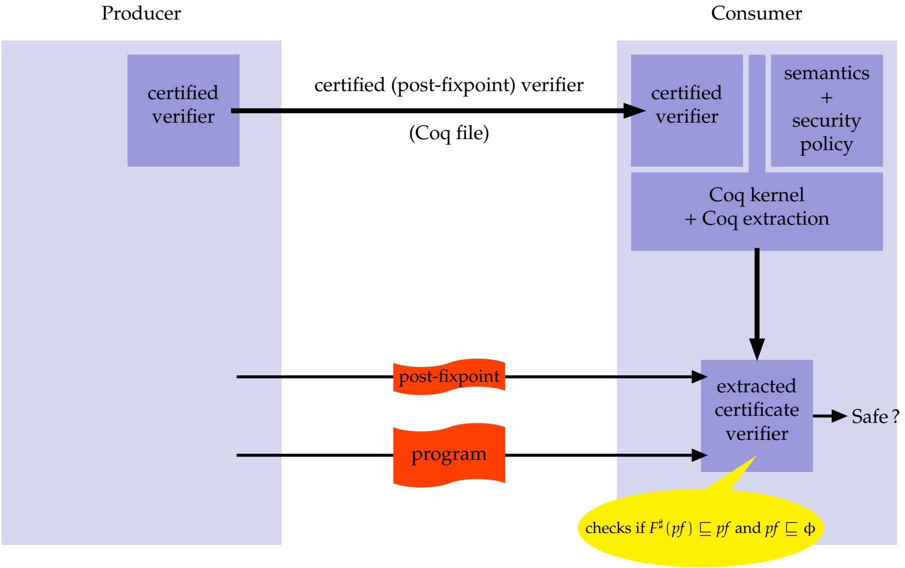
Certified PCC by abstract interpretation



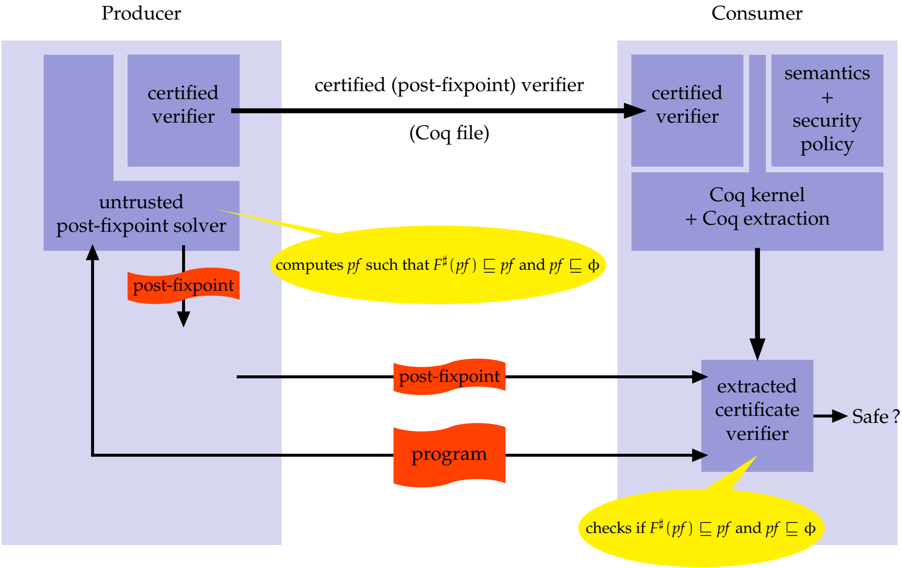
Certified PCC by abstract interpretation



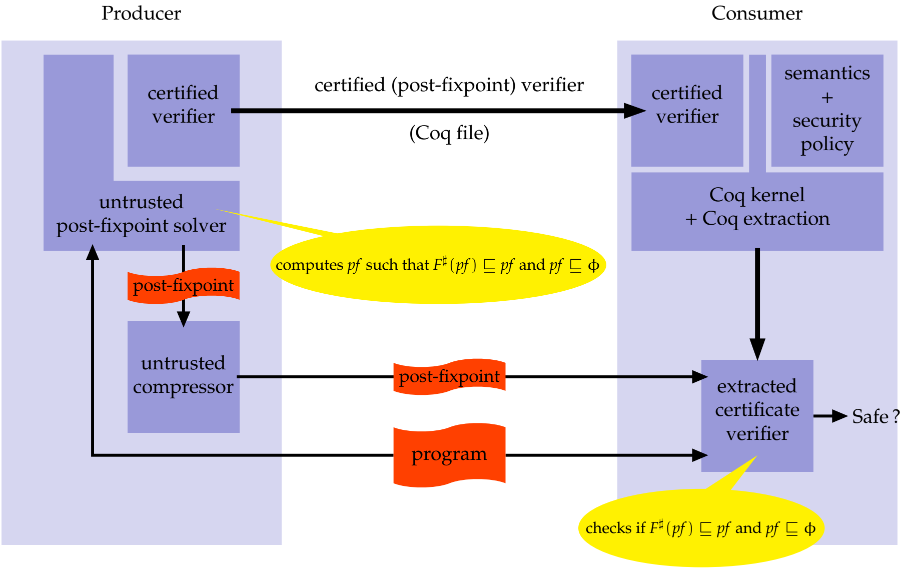
Certified PCC by abstract interpretation



Certified PCC by abstract interpretation



Certified PCC by abstract interpretation



Outline

Polyhedral abstract interpretation

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

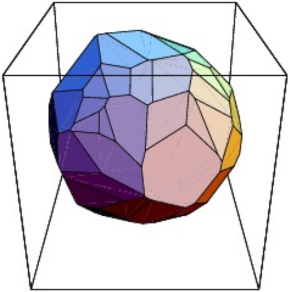
Patrick Cousot

Nicolas Halbwachs

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

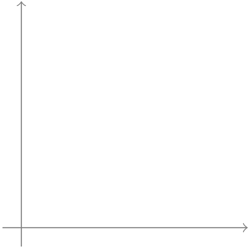
Convex polyhedra

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



Polyhedral analysis

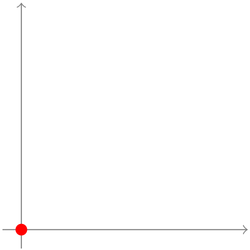
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;

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

Polyhedral analysis

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

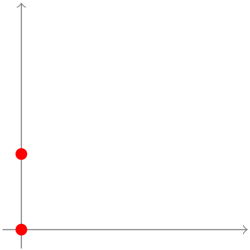


```
x = 0; y = 0;
```
$\{x = 0 \land y = 0\}$

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

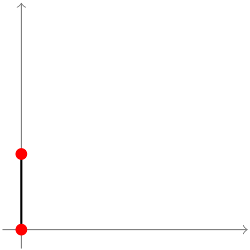


At junction point, we over approximate union by a convex union.

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

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

  x = x+1;

}
```

Polyhedral analysis

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

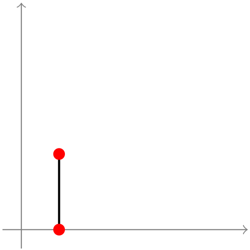


At junction point, we over approximate union by a convex union.

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

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

  x = x+1;

}
```

Polyhedral analysis

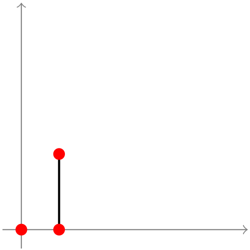
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



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

while (x<6) {
  if (?) {
        {x = 0 ∧ y = 0}
    y = y+2;
        {x = 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 $\mathbb{Q}^2$.



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

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

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

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

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

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

```
  };
```

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

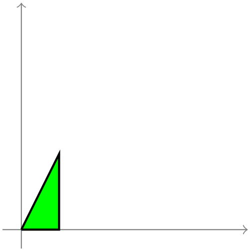
```
  x = x+1;
```

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

```
}
```

Polyhedral analysis

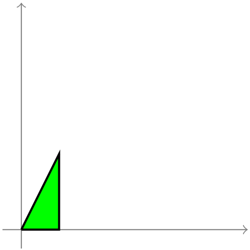
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;
      {x ⩽ 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

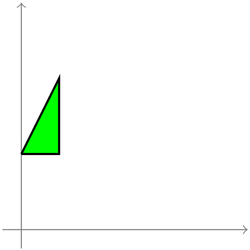
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
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

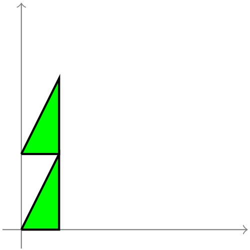
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



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

while (x<6) {
  if (?) {
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
        {x ⩽ 1 ∧ 2 ⩽ y ⩽ 2x + 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 $\mathbb{Q}^2$.

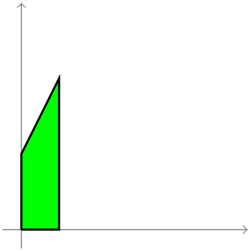


```
x = 0; y = 0;
```
$\{x \leqslant 1 \land 0 \leqslant y \leqslant 2x\}$

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

Polyhedral analysis

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



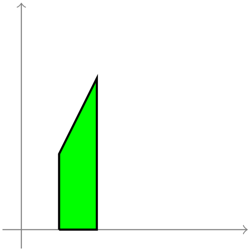
```
x = 0; y = 0;
```
$\{x \leqslant 1 \land 0 \leqslant y \leqslant 2x\}$

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

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

Polyhedral analysis

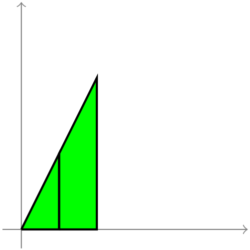
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



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

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

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

Polyhedral analysis

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

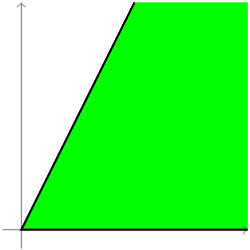


At loop headers, we use heuristics (widening) to ensure finite convergence.

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

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

Polyhedral analysis

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



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

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

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

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

Polyhedral analysis

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

By propagation we obtain a post-fixpoint

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

while (x<6) {
  if (?) {
        {0 ⩽ y ⩽ 2x ∧ x ⩽ 5}
    y = y+2;
        {2 ⩽ y ⩽ 2x + 2 ∧ x ⩽ 5}
  };
        {0 ⩽ y ⩽ 2x + 2 ∧ 0 ⩽ x ⩽ 5}

  x = x+1;
        {0 ⩽ y ⩽ 2x ∧ 1 ⩽ x ⩽ 6}
}
        {0 ⩽ y ⩽ 2x ∧ 6 ⩽ x}
```

Polyhedral analysis

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

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

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

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

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

Polyhedral analysis

A more complex example.

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

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

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

```
  };
```
$$\{A \leqslant y \leqslant 2x + A + 2 \ \wedge \ 0 \leqslant x \leqslant N - 1\}$$

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

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

The analysis accepts to replace some constants by parameters.

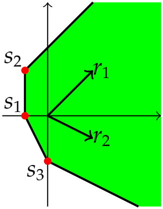
The four polyhedra operations

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

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

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

- by set of generators

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

  - 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/

Outline

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, . . .)

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 ;

$$[\![x := e]\!](P) = \exists x',\ P[x'/x]\ \wedge\ x = e[x'/x]$$

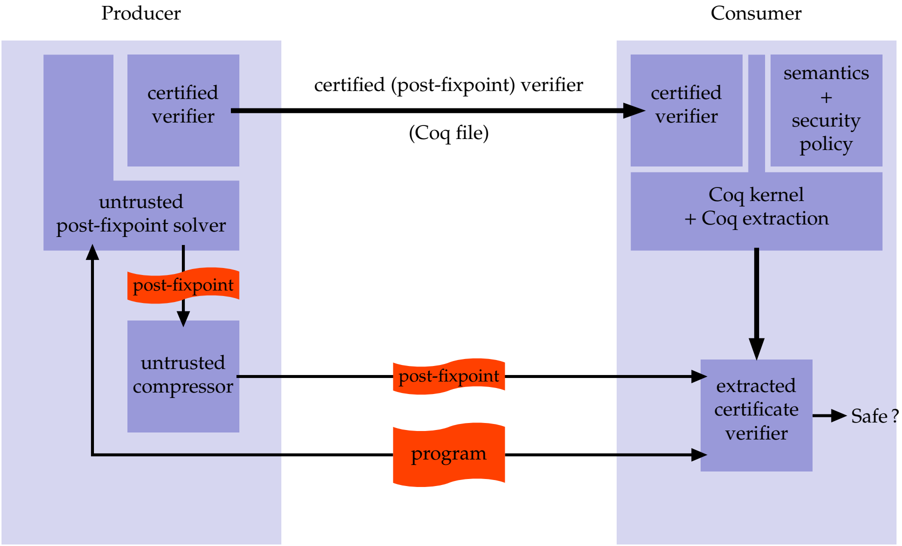
- Intersection is just syntactic union of constraints ;
- (Over-approximations) of Convex Hulls are given as untrusted invariants ;

$$isUpperBound(P, Q, UB) \equiv P \sqsubseteq UB \wedge Q \sqsubseteq UB$$

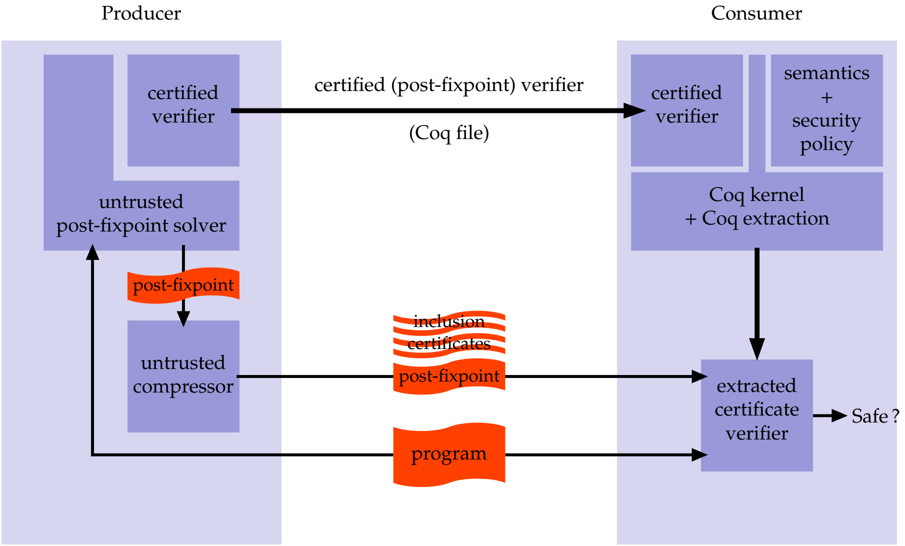
- Polyhedra inclusion is guided by a certificate ;

$$isIncluded(P, Q, Cert) \Rightarrow P \sqsubseteq Q$$

Certified PCC by abstract interpretation



Certified PCC by abstract interpretation



Checking polyhedra inclusion using certificates

- Inclusion reduces to a conjunction of emptiness problems

$$P \sqsubseteq \{q_1 \geqslant c_1, \ldots q_m \geqslant c_m\}$$

if and only if

$$P \cup \{-q_1 \geqslant -c_1 + 1\} = \emptyset \wedge \ldots \wedge P \cup \{-q_m \geqslant -c_m + 1\} = \emptyset$$

- Each emptiness reduces to unsatisfiability of linear constraints

$$\forall x_1, \ldots, x_n,\ \neg \begin{pmatrix} a_{1,1}, \ldots, a_{1,n} \\ \vdots \\ a_{m,1}, \ldots, a_{m,n} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \geqslant \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

Unsatisfiability certificates

Lemma (Farkas's Lemma (Variant))

Let $A \in \mathbb{Q}^{m \times n}$ *and* $b \in \mathbb{Q}^n$.

$$\forall x \in \mathbb{Q}^n, \neg(A \cdot x \geqslant b)$$

if and only if

$$\exists(cert \in \mathbb{Q}^m), cert \geqslant \bar{0}, \textit{ such that } \left\{ \begin{array}{l} A^t \cdot cert = \bar{0} \\ b^t \cdot cert > 0 \end{array} \right.$$

Soundness of certificates is easy ($\Leftarrow$)

Démonstration.

Suppose	$A \cdot x \geqslant b$.
Since $cert \geqslant \bar{0}$ we have	$(A \cdot x)^t \cdot cert \geqslant b^t \cdot cert$.
Now	$x^t \cdot (A^t \cdot cert) = (x^t \cdot A^t) \cdot cert = (A \cdot x)^t \cdot cert$.
Hence	$x^t \cdot (A^t \cdot cert) \geqslant b^t \cdot cert$.
Therefore	$x^t.\bar{0} = 0 \geqslant b^t \cdot cert > 0 \ \rightarrow$ contradiction.

□

Certificate checking

Example

Using the certificate $cert = (1;1;5)$, check that

$$\begin{pmatrix} 1 & 1 \\ -1 & 4 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \geqslant \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$$ 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 $\begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}^t \cdot \begin{pmatrix} 1 \\ 1 \\ 5 \end{pmatrix} > 0.$

□

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

$$Cert = \left\{ c \;\middle|\; \begin{array}{l} c \geqslant 0 \\ b^t \cdot c > 0 \\ A^t \cdot c = 0 \end{array} \right\}$$

Finding an *extremal* certificate is a linear programming problem

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

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

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

```
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

// PRE: $0 \leq |\mathtt{vec}_0|$

```
static int bsearch(int key, int[] vec) {
```

// (I_1) $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge 0 \leq |\mathtt{vec}_0|$

```
  int low = 0, high = vec.length - 1;
```

// (I_2) $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge 0 \leq \mathtt{low} \leq \mathtt{high} + 1 \leq |\mathtt{vec}_0|$

```
  while (0 < high-low) {
```

// (I_3) $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge 0 \leq \mathtt{low} < \mathtt{high} < |\mathtt{vec}_0|$

```
    int mid = (low + high) / 2;
```

// (I_4) $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge 0 \leq \mathtt{low} < \mathtt{high} < |\mathtt{vec}_0| \wedge \mathtt{low} + \mathtt{high} - 1 \leq 2 \cdot \mathtt{mid} \leq \mathtt{lo}$

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

// (I_5) $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge -2 + 3 \cdot \mathtt{low} \leq 2 \cdot \mathtt{high} + \mathtt{mid} \wedge -1 + 2 \cdot \mathtt{low} \leq \mathtt{hi}$
$\mathtt{mid} \wedge -1 + \mathtt{low} \leq \mathtt{mid} \leq 1 + \mathtt{high} \wedge \mathtt{high} \leq \mathtt{low} + \mathtt{mid} \wedge 1 + \mathtt{high} \leq 2 \cdot \mathtt{low} + \mathtt{mid} \wedge 1 + \mathtt{low} +$
$|\mathtt{vec}_0| + \mathtt{high} \wedge 2 \leq |\mathtt{vec}_0| \wedge 2 + \mathtt{high} + \mathtt{mid} \leq |\mathtt{vec}_0| + \mathtt{low}$

```
  }
```

// (I_6) $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge \mathtt{low} - 1 \leq \mathtt{high} \leq \mathtt{low} \wedge 0 \leq \mathtt{low} \wedge \mathtt{high} < |\mathtt{vec}_0|$

```
  return -1;
}
```

// POST: $-1 \leq \mathtt{res} < |\mathtt{vec}_0|$

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

Example : binary search

```
  //      PRE: True
static int bsearch(int key, int[] vec) {
  // (I'_1) |vec_0| = |vec| ∧ 0 ≤ |vec_0|
   int low = 0, high = vec.length - 1;
  // (I'_2) |vec_0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec_0|
   while (0 < high-low) {
  // (I'_3) |vec_0| = |vec| ∧ 0 ≤ low < high < |vec_0|
     int mid = (low + high) / 2;
  // (I'_4) |vec| − |vec_0| = 0 ∧ low ≥ 0 ∧ mid − low ≥ 0∧
  //        2 · high − 2 · mid − 1 ≥ 0 ∧ |vec_0| − high − 1 ≥ 0
     if (key == vec[mid]) return mid;
     else if (key < vec[mid]) high = mid - 1;
     else low = mid + 1;
  // (I'_5) |vec_0| = |vec| ∧ −1 + low ≤ high ∧ 0 ≤ low ∧ 5 + 2 · high ≤ 2 · |vec|
   }
  // (I'_6) 0 ≤ |vec_0|
   return -1;
} //    POST: −1 ≤ res < |vec_0|
```

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

Some preliminary benchmarks

	.class	certificates		checking time	
Program		before	after	before	after
BSearch	515	22	12	0.005	0.007
BubbleSort	528	15	14	0.0005	0.0003
HeapSort	858	72	32	0.053	0.025
QuickSort	833	87	44	0.54	0.25

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:
 ∀ (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)
```

demo