Extracting a Certified Static Analyser in Constructive Logic: Application to Proof Carrying Code

David Pichardie

CR2 INRIA - Lande project
Introduction

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

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

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

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

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

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

\[ = \text{Galos connection (98) so that post preserves joins} \]

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

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

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

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

\[ \lambda J. \text{let } \tau^\prime = \lambda l \in \text{in}_P [P].(l = \text{at}_P [S_t] \cup \text{Abexp}[B](J_l) \& J_l) \text{ in} \]

\[ \text{let } J^\prime = \text{APost}[S_t] (J_{\tau^\prime}) \text{ in} \]

\[ \lambda l \in \text{in}_P [P].(l = \tau^\prime \& J_l^\prime \& J_{l^\prime}) \]

\[ \cup \]

\[ \text{let } J^\prime = \lambda l \in \text{in}_P [P].(l = \text{at}_P [S_f] \cup \text{Abexp}[T(\neg B)](J_l) \& J_l) \text{ in} \]

\[ \text{let } J^\prime = \text{APost}[S_f] (J_{\tau^\prime}) \text{ in} \]

\[ \lambda l \in \text{in}_P [P].(l = \tau^\prime \& J_l^\prime \& J_{l^\prime}) \]

\[ = \text{by grouping similar terms} \]

\[ \lambda J. \text{let } \tau^\prime = \lambda l \in \text{in}_P [P].(l = \text{at}_P [S_t] \cup \text{Abexp}[B](J_l) \& J_l) \]

\[ \text{and } J^\prime = \lambda l \in \text{in}_P [P].(l = \text{at}_P [S_f] \cup \text{Abexp}[T(\neg B)](J_l) \& J_l) \text{ in} \]

\[ \text{let } J^\prime = \text{APost}[S_t] (J_{\tau^\prime}) \]

\[ \text{and } J^\prime = \text{APost}[S_f] (J_{\tau^\prime}) \text{ in} \]

\[ \lambda l \in \text{in}_P [P].(l = \tau^\prime \& J_l^\prime \& J_{l^\prime}) \]

\[ = \text{by locality (113) and labelling scheme (59) so that in particular } J^\prime_{l^\prime} = J^\prime_{\tau^\prime} = J_{l^\prime} \]

\[ J^\prime_{l^\prime} = J^\prime_{\tau^\prime} = J^\prime_{l^\prime} \text{ and APost}[S_t] \text{ and APost}[S_f] \text{ do not interfere} \]

©P.Cousot
Proof

\[ \hat{a}[P](\text{Post} \{ \text{if } B \text{ then } S_f \text{ else } S_f \text{ fi} \}) \]

\[ = \begin{cases} 
\hat{c} \text{ def. (110) of } \hat{a}[P] \\
\hat{a}[P] \circ \text{Post} \{ \text{if } B \text{ then } S_f \text{ else } S_f \text{ fi} \} \circ \hat{P}[P] \\
\text{def. (103) of Post} \\
\hat{a}[P] \circ \text{Post} \{ \tau^* \{ \text{if } B \text{ then } S_f \text{ else } S_f \text{ fi} \} \} \circ \hat{P}[P] \\
\text{big step operational semantics (93)} \\
\hat{a}[P] \circ \text{post}((1_{\Sigma}[P] \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma}[P] \cup \tau^I') \circ (1_{\Sigma}[P] \cup \tau^f) \circ \hat{P}[P] \\
\text{Galois connection (98)} \text{ so that post preserves joins'} \\
\hat{a}[P] \circ \text{post}((1_{\Sigma}[P] \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma}[P] \cup \tau^I') \circ \hat{P}[P] \\
\text{Galois connection (106)} \text{ so that } \hat{a}[P] \text{ preserves joins'} \\
(\hat{a}[P] \circ \text{post}((1_{\Sigma}[P] \cup \tau^B) \circ \tau^*[S_f] \circ (1_{\Sigma}[P] \cup \tau^I') \circ \hat{P}[P] \\
\text{by lemma (5.3) and similar one for the else branch'} \\
\lambda J \cdot \text{let } J' = \lambda \ell \in \text{in}_P[P], (l = \text{at}_P[S_f] \cup \text{Abexp}(B)(J_l) \cup J_l) \text{ in (120)} \\
\lambda J \cdot \text{let } J'' = \text{Post}[S_f](J') \text{ in } \\
\lambda \ell \in \text{in}_P[P], (l = \ell' ? J''_\ell \cup J''_{\ell'} \text{ after}_{P}[S_f] \cup J_{\ell'}^f) \\
\text{by grouping similar terms'} \\
\lambda J \cdot \text{let } J' = \lambda \ell \in \text{in}_P[P], (l = \text{at}_P[S_f] \cup \text{Abexp}(B)(J_l) \cup J_l) \text{ and } \\
\lambda J' = \lambda \ell \in \text{in}_P[P], (l = \text{at}_P[S_f] \cup \text{Abexp}(T(\neg B))(J_l) \cup J_l) \text{ in } \\
\lambda \ell \in \text{in}_P[P], (l = \ell' ? J'_{\ell'} \cup J'_{\ell'} \text{ after}_{P}[S_f] \cup J_{\ell'}^f) \\
\text{by locality (113) and labelling scheme (59) so that in particular } J'_{\ell'} = J'_{\ell'} = J''_{\ell'} = J''_{\ell'} = J''_{\ell'} = J''_{\ell'} = J''_{\ell'} = J''_{\ell'} \\
\text{and } \text{Post}[S_f] \text{ and } \text{Post}[S_f] \text{ do not interfere}'
\end{cases} \]

©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 = mr;
  mat->ncolumns = nc;
  mat->_pinit = matrix_t*matrix_alloc_int(const int mr, const int nc);
  mat->post = (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;
}
```

©B.Jeannët
Proof

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

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

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

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

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

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

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

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

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

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

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

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

\[\lambda \in \text{in}_P[P] \cdot (l = 0' \? J_{t'}^o \cup J_{f'}^o \text{ after}_P[S_t] \& J_{t'}^f)\]

\[\lambda \in \text{in}_P[P] \cdot (l = 0' \? J_{t'}^o \cup J_{f'}^o \text{ after}_P[S_t] \& J_{t'}^f)\]

= \{\text{by locality (113) and labelling scheme (59) so that in particular } J_{t'}^o = J_{t'}^o = J_{t'}^o\}

\[J_{t'}^o = J_{t'}^o \text{ and } \text{APost}[S_t] \text{ and } \text{APost}[S_f] \text{ do not interfere}\]

©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 = mr;
    mat->ncolumns = nc;
    mat->_pinit = 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+=nc;
        }
    }
    return mat;
}
```

```c
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);
}
```
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
Contributions

▶ We have isolated a fragment of Abstract Interpretation adequate for proving analysis soundness in the constructive logic of Coq

▶ We have programmed certified generic fixpoint solvers

▶ We have developed a lattice library allowing an easy construction of complex terminations proofs

▶ Several cases studies for bytecode Java

1. A control flow analysis for a representative subset of JavaCard
2. A memory usage analysis for a representative subset of JavaCard
3. An interval analysis for an imperative fragment of JavaCard with dynamic arrays (see PCC applications)

▶ Application to proof carrying code
Outline

1 Introduction
Outline

1. Introduction

2. Building a certified static analyser
Outline

1. Introduction
2. Building a certified static analyser
3. Application to Proof Carrying Code
Outline

1. Introduction
2. Building a certified static analyser
3. Application to Proof Carrying Code
4. Perspectives
Outline

1. Introduction
2. Building a certified static analyser
3. Application to Proof Carrying Code
4. Perspectives
Building a certified static analyser

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

Example: JVM states

\[ \langle \langle h, \langle m, pc, l, v :: s \rangle, sf \rangle \rangle \]
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?
Building a certified static analyser

- Each semantic sub-domain has its abstract counterpart
- An abstract domain is a lattice \((\mathcal{D}^#, =, \sqsubseteq, \perp, \sqcup, \sqcap)\) 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?
  - generic lattice library
Building lattices in Coq

We propose a technique based on the 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
  - Base lattices: signs, congruences, constants, intervals, finite set (implemented with trees)
  - Functors: product, disjoint and lifted sums, lists, functional arrays (implemented with trees)
  - For each functor the most challenging proofs deals with the preservation of the termination criterion.
- The library deals as well with widening/narrowing
Each abstract value represents a property on concrete values.

This correspondence is formalised by a monotone concretisation function:

\[ \gamma : (\mathcal{D}^\#, \sqsubseteq) \longrightarrow_m (\wp(\mathcal{D}), \subseteq) \]
Each abstract value represents a property on concrete values

This correspondence is formalised by a monotone concretisation function

\[ \gamma : (\mathcal{D}^\# , \sqsubseteq) \rightarrow_m (\wp(\mathcal{D}) , \subseteq) \]

\[ x \subseteq \gamma(x^\#) \text{ means } "x^\# \text{ is a correct approximation of } x" \]
Building a certified static analyser

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

\[
\begin{align*}
\text{instructionAt}_{P}(m, pc) &= \text{push } c \\
\langle \langle h, \langle m, pc, l, s \rangle, sf \rangle \rangle &\rightarrow \langle \langle h, \langle m, pc + 1, l, c :: s \rangle, sf \rangle \rangle
\end{align*}
\]

\[
\begin{align*}
\text{instructionAt}_{P}(m, pc) &= \text{invokevirtual } m_{id} \\
m' &= \text{methodLookup}(m_{id}, h(loc)) \\
V &= v_1 :: \cdots :: v_{\text{nbArguments}(m_{id})} \\
\langle \langle h, \langle m, pc, l, loc :: V :: s \rangle, sf \rangle \rangle &\rightarrow \langle \langle h, \langle m', 1, V, \epsilon \rangle, \langle m, pc, l, s \rangle :: sf \rangle \rangle
\end{align*}
\]
Building a certified static analyser

- the analysis is specified as a solution of a post fixpoint problem
  \[ F_P^\#(s^\#) \sqsubseteq^\# s^\# \]
- after partitioning: constraint system
  \[
  \begin{align*}
  f_1^\#(s_1^\#, \ldots, s_n^\#) & \sqsubseteq^\# s_{i_1}^\# \\
  \cdots \\
  f_n^\#(s_1^\#, \ldots, s_n^\#) & \sqsubseteq^\# s_{i_n}^\#
  \end{align*}
  \]
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

Two techniques of iterative computation

- traditional least (post)-fixpoint computation

\[
\downarrow \rightarrow F_P^\#(\bot) \rightarrow F_P^\#(\bot) \rightarrow \cdots \text{lfp}(F_P^\#)
\]

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

In the two cases, a generic tool

\[
\forall P, \exists s^\#, F_P^\#(s^\#) \sqsubseteq s^\#
\]
Building a certified static analyser

Final result

\[
\forall P, \forall s\#, F^\#_P(s\#) \sqsubseteq s\# \Rightarrow [P] \subseteq \gamma(s\#) \\
\forall P, \exists s\#, F^\#_P(s\#) \sqsubseteq s\#
\]

In Coq: \( \text{analyse : } \forall \, p: \text{program}, \{ \, s: \text{abstate} \mid \text{sem}(P) \sqsubseteq \gamma(P, s) \} \)

In Caml: \( \text{analyse : program } \rightarrow \text{abstate} \)
Efficiency of the extracted program

Experiments on the memory usage analysis

\[ \mathcal{D}^\# = (\mathbb{M} \to \mathbb{P}(\mathbb{M})) \times \mathbb{P}(\mathbb{M}) \times (\mathbb{M} \to \mathbb{P} \to \mathbb{P}(\mathbb{P})) \times \mathbb{P}(\mathbb{M}) \]
Outline

1. Introduction
2. Building a certified static analyser
3. Application to Proof Carrying Code
4. Perspectives
Mobile code dilemmas...

- The untrusted code may cause damages on the system
- The untrusted code may use too many resources
  - CPU, memory, SMS...
Mobile code dilemmas...

Untrusted code

Host system

- The untrusted code may cause damages on the system
  - intern structure corruption
- The untrusted code may use too many resources
  - CPU, memory, SMS...
Proof carrying code

- Code
- Certifying prover
- Certificate verifier
- CPU

Flow diagram showing the process of proof carrying code.
Proof carrying code: standard framework
Proof carrying code by abstract interpretation
Proof carrying code by abstract interpretation

PCC requirements:
- the certificate must be small
- the verifier must be efficient
- soundness of the certifying prover is not critical
- soundness of the verifier is critical

Our works on certified static analysis allows the consumer to check semantics soundness of the verifier
- certified proof carrying code = certified static analysis without fixpoint solver
Proof carrying code by certified abstract interpretation

Producer

Consumer

Safe ?

program

Certified Static Analysis
Proof carrying code by certified abstract interpretation

Producer

Consumer

certificate

program

certificate verifier

Safe?
Proof carrying code by certified abstract interpretation

Producer

Consumer

Coq kernel + Coq extraction

semantics + security policy

Safe?
Proof carrying code by certified abstract interpretation

Producer

 certified verifier

 certified (post-fixpoint) verifier (Coq file)

 Consumer

 semantics + security policy

 Coq kernel + Coq extraction

 certificate extracted verifier

 program

 certificate

 Safe ?

 [Ad hoc format]
Proof carrying code by certified abstract interpretation

Producer

- certified verifier
- untrusted post-fixpoint solver

Consumer

- certified verifier
- semantics + security policy
- Coq kernel + Coq extraction
- certificate
- program
- Safe?
Proof carrying code by certified abstract interpretation

Producer

Certificate

Program

Consumer

Semantics + security policy

Coq kernel + Coq extraction

Safe?
Proof carrying code by certified abstract interpretation

Producer

- certified verifier
- untrusted post-fixpoint solver
- post-fixpoint
- untrusted compressor

Consumer

- certified verifier
- semantics + security policy
- Coq kernel + Coq extraction
- certificate extracted verifier
- certificate (ad hoc format)
- program

Safe?
First case study

- imperative fragment of bytecode Java + dynamically allocated arrays
- we verify arrays access (no out of bounds errors) by intervals approximation
- experiments on various algorithms\(^1\):

<table>
<thead>
<tr>
<th>Program</th>
<th>.java size</th>
<th>.class size</th>
<th>certificate size</th>
<th>checking time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BubbleSort</td>
<td>440</td>
<td>528</td>
<td>32</td>
<td>0.015</td>
</tr>
<tr>
<td>HeapSort</td>
<td>1044</td>
<td>858</td>
<td>63</td>
<td>0.050</td>
</tr>
<tr>
<td>QuickSort</td>
<td>1078</td>
<td>965</td>
<td>124</td>
<td>0.060</td>
</tr>
<tr>
<td>ConvolutionProduct</td>
<td>378</td>
<td>542</td>
<td>52</td>
<td>0.010</td>
</tr>
<tr>
<td>FloydWharshall</td>
<td>417</td>
<td>596</td>
<td>134</td>
<td>0.020</td>
</tr>
<tr>
<td>PolynomProduct</td>
<td>509</td>
<td>604</td>
<td>87</td>
<td>0.010</td>
</tr>
</tbody>
</table>

certificate/programme \(\sim\) 10\%, verification time <0.1s

\(^1\) Size file in bytes, time in secs
Outline

1. Introduction
2. Building a certified static analyser
3. Application to Proof Carrying Code
4. Perspectives
Perspectives

Increasing the power of certified static analyses
  ▶ symbolic manipulation
  ▶ relational abstract domains (octagons, polyhedra)

Proof-carrying code
  ▶ fixpoint compression (see ESOP’07)
  ▶ we want to demonstrate the scalability of our architecture

Pre-analyses for others verification tools
  ▶ automatically detect some unthrowable exceptions
  ▶ alias analysis
  ▶ race detection