Modular Development of Certified Program Verifiers with a Proof Assistant

Adam Chlipala
University of California, Berkeley
ICFP 2006
Who Watches the Watcher?

Might want to ensure:
- Memory safety
- Resource usage bounds
- Total correctness

Program Verifier

- Type-checker for stylized verifier language?
- Result checker on witnesses outputted by verifiers?
- Interactive proof assistant?

Java Bytecode Verifier
- Extended Static Checking
- Typed Assembly Language
- Proof-Carrying Code
- Model Checking

Untrusted Program
But Why?

Proof-Carrying Code

- Compact proofs in a language specialized to one safety mechanism (e.g., a type system)
- Every new safety mechanism requires trusting a new body of code

Foundational Proof-Carrying Code

- Proofs about the real machine semantics, written in a very general language
- Proofs are much larger, making them expensive to check and transmit

Certified Program Verifiers

- Allow custom executable verifiers that can be reused
- Require that every verifier be proved sound
- No proofs generated or checked at runtime
The Big Picture

x86 Machine Code Semantics

Proof of Verifier Soundness

Program Extraction

OCaml Source

Compiler

Native Code

Memory Safety Verifier

Annotations

Safe

x86 executable

Unknown

Type annotations on registers, stack slots, etc., for each basic block
The Big Picture

- High-Level Type System + Soundness Proof
- Verifier Construction Library (Functor)
- Memory Safety Verifier
- Proof of Verifier Soundness
Outline

• Programming with dependent types
  – ...using a proof assistant
• A library for constructing certified verifiers
• Implementation
Classical Program Verification

Verification Condition Generator

\[ A \land B \rightarrow C \]

Verification Condition

Interactive Provers
(Coq, Isabelle, PVS, etc.)

Proof
Internal Verification

Precondition and postcondition

Proof of precondition

Proof of postcondition

Precondition and postcondition

Proof of precondition

Proof of postcondition

Proof of postcondition
Benefits vs. Classical Verification

Type Inference → Main Loop → Verification Condition Generator

Query (+ precondition proof) → Main Loop → Proof

Type Inference

Main Loop

Verification Condition Generator

“Your proof is wrong! (+ precondition proof)

Type error at line 1234!”

Thanks!

System A ≠ B

new

“This verifier is sound.”
Dealing with Proof Terms

\[
\text{fun } \text{ls1 : list } => \\
\text{list_ind} \\
\quad (\text{fun } \text{ls2 : list } => \\
\quad \quad \text{forall } \text{ls3 ls4 : list,} \\
\quad \quad \quad \text{append (append ls2 ls3) ls4 = append ls2 (append ls3 ls4)}) \\
\quad \text{(fun } \text{ls2 ls3 : list } => \text{refl_equal (append ls2 ls3))} \\
\quad \text{(fun } (n : \text{nat}) \text{ (ls2 : list)} \\
\quad \quad \quad \text{IHls1 : forall } \text{ls3 ls4 : list,} \\
\quad \quad \quad \quad \text{append (append ls2 ls3) ls4 = append ls2 (append ls3 ls4))} \\
\quad \quad \quad \quad \text{ls3 ls4 : list) } => \\
\quad \quad \quad \quad \text{eq_ind_r (fun } l : \text{list } => \text{cons n l = cons n (append ls2 (append ls3 ls4))}) \\
\quad \quad \quad \quad \quad \text{(refl_equal (cons n (append ls2 (append ls3 ls4))))} \\
\quad \quad \quad \quad \quad \quad \text{(IHls1 ls3 ls4)) ls1}
\]
Mixing Programming with Tactics

Definition isEven : forall n, [even(n)].
refine (fix isEven (n : nat) :
  [even(n)] :=
match n return [even(n)] with
  O -> Yes
| S O ->
| S S n proof
proof
Yes);
auto.
Qed.

Step 1. Declare the function

Step 3. Generate the "proof part" of the implementation using tactics.

Monadic notation: fail if the recursive call fails; otherwise, bind proof in the body.
A Verification Stack

$\tau ::= \text{int} \mid \tau \text{ptr}$

**Phantom state**: Map from addresses to types

**Phantom state**: Map from flags to correlation with registers/memory

- Proofs that inference procedure and operation checker respect that semantics

- Assumption: Code is immutable

$\mathcal{D} ::= \text{register} \rightarrow \tau$

$\text{step} : \text{state} \times \text{RISC instruction} \rightarrow \text{state}$

$\mathcal{D} ::= \text{register} \rightarrow \tau$

$\text{step} : \text{state} \times \text{x86 instruction} \rightarrow \text{state}$

Generic fixed point computation procedure

**Weak Update Type System**

**Simple Flags**

**Stack Types**

**Type System**

**Fixed Code**

**Reduction**

**Abstract Interpretation**

**x86 Semantics**

- Note: The proofs are erased by program extraction! (They are just a formal device for proving the verifier’s soundness.)
## Implementation

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification stack</td>
<td>7k (Coq)</td>
</tr>
<tr>
<td>Bitvectors and fixed-precision arithmetic</td>
<td>1k (Coq)</td>
</tr>
<tr>
<td>x86 semantics</td>
<td>1k (Coq)</td>
</tr>
<tr>
<td>Utility library</td>
<td>10k (Coq)</td>
</tr>
<tr>
<td>x86 binary parser</td>
<td></td>
</tr>
<tr>
<td>New extraction optimizations</td>
<td>1500 (OCaml)</td>
</tr>
<tr>
<td>Algebraic datatype verifier</td>
<td>600 (Coq)</td>
</tr>
<tr>
<td>Total trusted</td>
<td>~5k</td>
</tr>
</tbody>
</table>
Sample Code: Type Language

Inductive ty : Set :=
  | Constant : int32 -> ty
  | Product : product -> ty
  | Sum : ty -> ty -> ty
  | Var : var -> ty
  | Recursive : var -> ty -> ty

with product : Set :=
  | PNil : product
  | PCons : ty -> product -> product.
Sample Code: Subtype Checker

Definition subTy : forall (t1 t2 : ty),
    poption (forall ctx v,
        hasTy ctx v t1 -> hasTy ctx v t2).
refine (fix subTy (t1 t2 : ty) {struct t2}
    : poption (forall ctx v,
        hasTy ctx v t1 -> hasTy ctx v t2) :=
    match (t1, t2) with
    | (Constant n1, Constant n2) =>
        pfEq <- int32_eq n1 n2;
        Yes
    | (Product (PCons (Constant n) (PCons t PNil)),
        Sum t1 t2) =>
        if int32_eq n 0 && ty_eq t t1 then Yes
        else if int32_eq n 1 && ty_eq t t2 then Yes
        else No
    | (Recursive x body, t2) =>
        pfSub <- subTy
            (subst x (Recursive x body) body) t2;
        Yes
    | ...
end); ....
Qed.
Related Work

- CompCert certified C compiler project [Leroy et al.]
- Foundational proof checkers with small witnesses [Wu et al.]
- Lots of work on building bytecode verifiers with proof assistants
Conclusion

- Today's technology makes constructing certified verifiers with dependent types feasible
- Good mixture of soundness guarantees, ease of engineering, and runtime efficiency

Code and documentation on the web at: http://proofos.sourceforge.net/