Analyse et Conception Formelles

Lesson 6

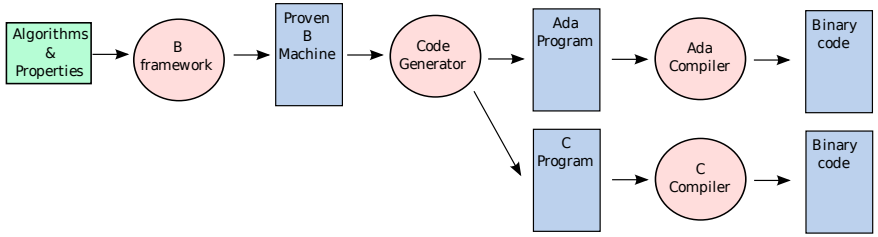
–

Certified Programming

Outline

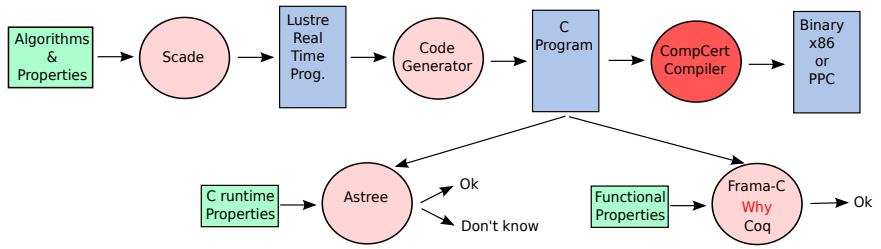
1. Certified program production lines
   - Some examples of certified code production lines
   - What are the weak links?
   - How to certify a compiler?
   - How to certify a static analyzer of code?
   - How to guarantee the correctness of proofs?
2. Methodology for formally defining programs and properties
   - Simple programs have simple proofs
   - Generalize properties when possible
   - Look for the smallest trusted base

B code production line



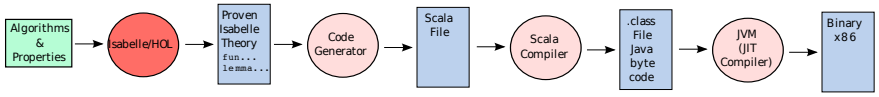
- The first certified code production line used in the industry
- For security critical code
- Used for onboard automatic train control of metro 14 (RATP)
- Several industrial users: RATP, Alstom, Siemens, Gemalto

Scade/Astree/CompCert code production line



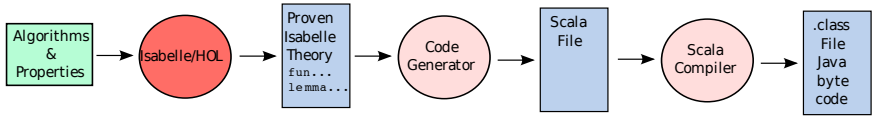
- The (next) Airbus code production line
- For security critical code (*e.g* flight control)
- Scade uses model-checking to verify programs or find counterexamples
- Astree is a static analyzer of C programs *proving* the absence of
  - division by zero, out of bound array indexing
  - arithmetic overflows
- Frama-C is a proof tool for C prog. (close to Why), automated provers like Alt-Ergo, CVC4, Z3, etc. and the Coq proof assistant
- CompCert is a certified C compiler (X. Leroy & S. Blazy, etc.)

Isabelle to Scala line



- Used for specification and verification of industrial size softwares
  e.g. Operating system kernel seL4 (C code)
- Code generation not yet used at an industrial level
- More general purpose line than previous ones
- All proofs performed in Isabelle are checked by a trusted kernel
- Formalization/Verification of other parts is ongoing research
  e.g. some research efforts for certifying a JVM

What are the weak links of such lines?



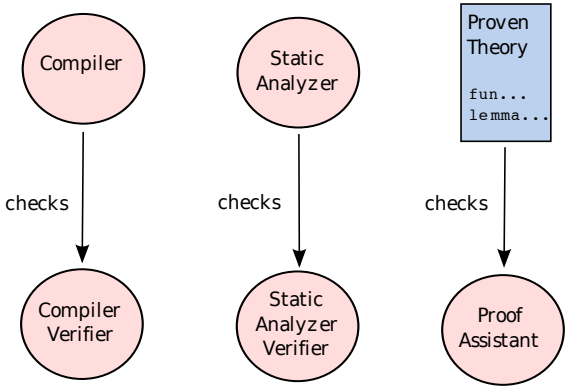
1. The initial choice of algorithms and properties
2. The verification tools (analyzers and proof assistants)
3. Code generators/compilers

$\Longrightarrow$ we need some guaranties on each link!

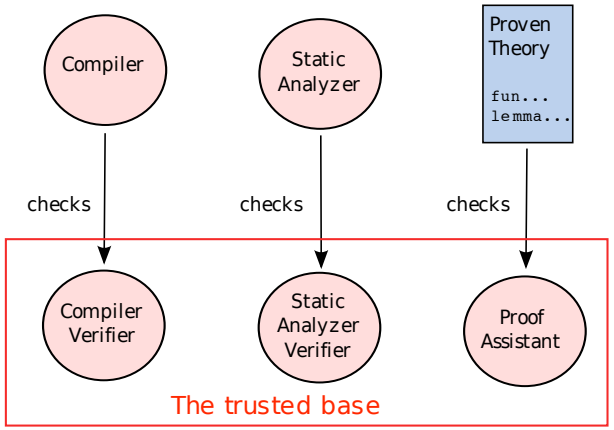
1. Certification of compilers
2. Certification of static analyzers
3. Verification of proofs in proof assistant
4. Methodology for formally defining algorithms and properties

$\Longrightarrow$ we need to limit the trusted base!

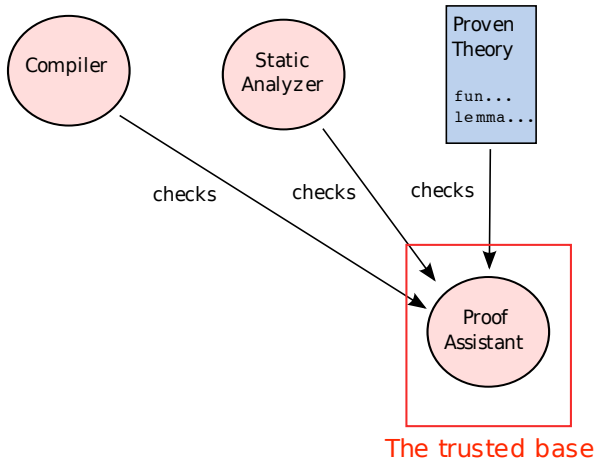
How to limit the trusted base?



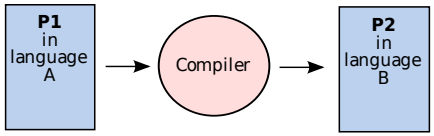
How to limit the trusted base?



How to limit the trusted base?



How to certify a compiler?

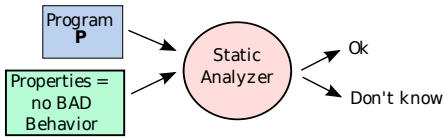


What is the property to prove? $\forall$ **P1**. **P1** «behaves» like **P2**

How can we prove this?

- Need to formally describe behaviors of programs:
  - Formal semantics for language A and language B
  - Close to defining an interpreter (using terms and functions) (≈TP4)
    i.e. define `evalA(prog,inputs)` and `evalB(prog,inputs)`
- Then, prove that $\forall$ **P1** **P2** s.t. **P2**=compil(**P1**):
  - $\forall$ inputs. `evalA`(**P1**,inputs) stops $\longleftrightarrow$ `evalB`(**P2**,inputs) stops, and
  - $\forall$ inputs. `evalA`(**P1**,inputs) = `evalB`(**P2**,inputs)
- Proving this by hand is unrealistic (recall the size of Java semantics)
- Use a proof assistant... compiler is correct if the proof assistant is!

How to certify a static analyzer (SAn)? (TP67)



What is the property to prove?

$\forall$ **P**. SAn(**P**)=True $\longrightarrow$ «nothing bad happens when executing **P**»

How can we prove this?

- Again, we need to formally describe behaviors of programs:
  - Formal semantics of language of **P**, define `eval(prog,inputs)`
- We need to formalize the analyzer and what is a «bad» behavior
  - Formalize «bad» , *i.e.* define a `BAD` predicate on program results
  - Formalize the analyser SAn
- Then, prove that the static analyzer is safe:
  $\forall$ **P**. $\forall$ inputs. (SAn(**P**)= True) $\longrightarrow$ $\neg$ `BAD`(`eval`(**P**,inputs))
- Again, proving this by hand is unrealistic
- Use a proof assistant... analyzer is correct if the proof assistant is!

Static analysis – the quiz

Quiz 1

- *What is a static analyzer good at?*

V	*Proving a property*
R	*Finding bugs*

- *Is a static analyzer running the program to analyze?*

V	*Yes*
R	*No*

- *Is a static analyzer has access to the user inputs?*

V	*Yes*
R	*No*

- *Given a program* **P**, `eval` *and* `BAD`, *can we verify by computation that for all inputs,* $\neg$ `BAD`(`eval`(**P**,*inputs*))*?*

V	*Yes*	R	*No*

- *Given a program* **P**, *and* `SAn` *can we verify by computation that* `SAn`(**P**)=`True`*?*

V	*Yes*	R	*No*

How to certify a static analyzer (SAn)? (II)

Isabelle file `cm6.thy`

Exercise 1

Define a static analyzer `san` *for such programs:*

`san:: program ⇒ bool`

Exercise 2

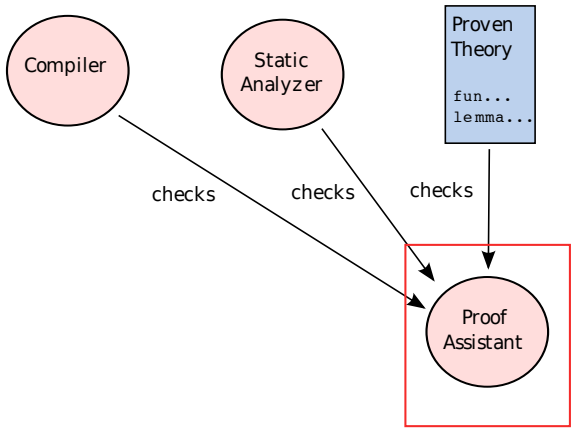
Define the `BAD` *predicate on program states:*

`BAD:: pgState ⇒ bool`

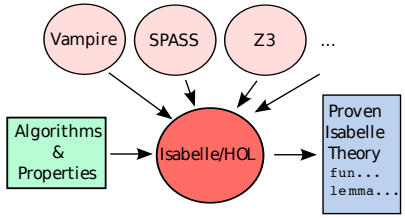
Exercise 3

Define the correctness lemma for the static analyzer `san`.

In the end, we managed to do this...



How to guarantee correctness of proofs in proof assistants?



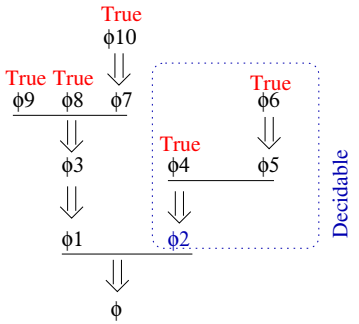
How to be convinced by the proofs done by a proof assistant?

- Relies on complex algorithms
- Relies on complex logic theories
- Relies on complex decision procedures

$\Longrightarrow$ there may be bugs everywhere!

Weak points of proof assistants

A proof in a proof assistant is a tree whose leaves are axioms



Difference with a proof on paper:

- Far more detailed
- A lot of axioms
- Shortcuts: External decision procedures

Axioms $\Longrightarrow$ fewer details

Decision Proc. $\Longrightarrow$ automatization

Axioms and decision procedures are the main weaknesses of proof assistants

Choices made in Coq, Isabelle/HOL, PVS, ACL2, etc. are very different

Proof handling : differences between proof assistants

	Coq	PVS	Isabelle	ACL2
Axioms	minimum and fixed	free	minimum and fixed	free
Decision procedures	proofs checked by Coq	trusted (no check)	proofs checked by Isabelle	trusted (no check)
Proof terms	built-in	no	additional	no
System automatization	basic	in between	in between	good
Counterexample generator	basic	basic	yes	yes

Proof checking: how is it done in Isabelle/HOL?

Isabelle/HOL have a well defined and «small » trusted base

- A kernel deduction engine (with Higher-order rewriting)
- Few axioms for each theory (see HOL.thy, HOL/Nat.thy)
- Other properties are lemmas, *i.e.* demonstrated using the axioms

All proofs are carried out using this trusted base:

- Proofs directly done in Isabelle (`auto`/`simp`/`induct`/...)
- All proofs done outside (`sledgehammer`) are re-interpreted in Isabelle using `metis` or `smt` that construct an Isabelle proof

Example 1

Prove the lemma $(x + 4) * (y + 5) \geq x * y$ using `sledgehammer`.

1. Interpret the found proof using `metis`
2. Switch on tracing: add
   `using [[simp_trace=true,simp_trace_depth_limit=5]]`
   before the `apply` command
3. Re-interpret the proof

Outline

1. Certified program production lines
   - Some examples of certified code production lines
   - What are the weak links?
   - How to certify a compiler?
   - How to certify a static analyzer of code?
   - How to guarantee the correctness of proofs?
2. Methodology for formally defining programs and properties
   1. Simple programs have simple proofs
   2. Generalize properties when possible
   3. Look for the smallest trusted base

Simple programs have simple proofs : Simple is beautiful

Example 2 (The `intersection` function of TP2/3)

An «optimized» version of `intersection` is harder to prove.

1. Program function `f(x)` as simply as possible... no optimization yet!
   - Use simple data structures for `x` and the result of `f(x)`
   - Use simple computation methods in `f`
2. Prove all the properties `lem1`, `lem2`, ... needed on `f`
3. (If necessary) program `fopt(x)` an optimized version of `f`
   - Optimize computation of `fopt`
   - Use optimized data structure if necessary
4. Prove that $\forall$ `x. f(x)=fopt(x)`
5. Using the previous lemma, prove again `lem1`, `lem2`, ... on `fopt`

Simple programs have simple proofs (II)

Exercise 4

The function `fastReverse` *is a tail-recursive version of* `reverse`*. Prove the classical lemmas on* `fastReverse` *using the same properties of* `reverse`*:*

- `fastReverse (fastReverse l)=l`
- `fastReverse (l1@l2)= (fastReverse l2)@(fastReverse l1)`

Exercise 5

Prove that the fast exponentiation function `fastPower` *enjoys the classical properties of exponentiation:*

- $x^y * x^z = x^{(y+z)}$
- $(x * y)^z = x^z * y^z$
- $x^{y^z} = x^{(y*z)}$

Generalize properties when possible

Exercise 6 (On `List.member` and `intersection` of TP2/3)

- *Prove that* `((List.member l1 e) ∧ (List.member l2 e)) ⟶ (List.member (intersection l1 l2) e)`
- *How to generalize this property?*
- *What is the problem with the given function* `intersection`*?*

Exercise 7 (On function `clean` of TP2/3)

- *Prove that* `clean [x,y,x]=[y,x]`
- *How to generalize this property of* `clean`*?*
- *What is the problem with the given definition of function* `clean`*?*

Exercise 8 (On functions `List.member` and `delete` of TP2/3)

- *Try to prove that*

`List.member l x ⟶ List.member l y ⟶ x≠y ⟶ (List.member (delete y l) x)`

Limit the trusted base in your Isabelle theories

Trusted base = functions that you cannot prove and have to trust
Basic functions on which lemmas are difficult to state

To verify a function `f`, define lemmas using `f` and:

- functions of the trusted base
- other proven functions

Example 3

In TP2/3, which functions can be a good trusted base?

Remark: There can be some interdependent functions to prove!

Example 4 (Prove a `parser` and a `prettyPrinter` on programs)

- `parser:: string ⇒ prog`
- `prettyPrinter:: prog ⇒ string`

The property to prove is: ∀ `p. parser(prettyPrinter p) = p`

`prettyPrinter` is more likely to be trusted since it is simpler