Smart & ProvenTools

Proof Techniques That Scale

Stéphane Lescuyer

Prove & Run

Entropy 2019, Stockholm, 16/06/2019
ProvenCore is a very large verification project:

- 17,000 lines of actual code
- 380,000 lines of specs and lemmas across 720+ modules and 4 refinement levels
- 180,000 hints to prove 29,000 VCs
• ProvenCore is very large verification project
  → 17000 lines of actual code
  → 380000 lines of specs and lemmas across 720+ modules and 4 refinement levels
  → 180000 hints to prove 29000 VCs

• in an interactive proof system, with limited manpower
ProvenCore

- ProvenCore is very large verification project
  - 17000 lines of actual code
  - 380000 lines of specs and lemmas across 720+ modules and 4 refinement levels
  - 180000 hints to prove 29000 VCs

- in an interactive proof system, with limited manpower

- how do we achieve and maintain such a large-scale effort?
WINTER AIN'T COMING YET...

MUST FINISH PROOF FIRST...
Scalable approach

- proof by refinement allows parallel work
Scalable approach

- proof by refinement allows parallel work
- we designed our own language and IDE ProvenTools
Scalable approach

- proof by refinement allows parallel work
- we designed our own language and IDE ProvenTools
  - Smart is a unique language for code & specs
Scalable approach

- proof by refinement allows parallel work
- we designed our own language and IDE ProvenTools
  - Smart is a unique language for code & specs
  - C generator supports ghost code, and the linear discipline it enforces is light and natural
Scalable approach

- proof by refinement allows *parallel* work
- we designed our own language and IDE ProvenTools
  - Smart is a unique language for code & specs
  - C generator supports ghost code, and the linear discipline it enforces is light and natural
  - automated and assisted maintenance of proofs
Scalable approach

- proof by refinement allows *parallel* work
- we designed our own language and IDE ProvenTools
  - Smart is a unique language for code & specs
  - C generator supports ghost code, and the linear discipline it enforces is light and natural
  - automated and assisted maintenance of proofs
  - static analyses for the *framing problem*
Scalable approach

- proof by refinement allows parallel work
- we designed our own language and IDE ProvenTools
  - Smart is a unique language for code & specs
  - C generator supports ghost code, and the linear discipline it enforces is light and natural
  - automated and assisted maintenance of proofs
  - static analyses for the framing problem
- makes strict separation of code and specs/proofs possible
Obfuscating code with specs (ADA/Spark2014)
Obfuscating code with specs (Java/VeriFast)
Obfuscating code with specs (Why3)
Separation of code and specifications

→ better readability
→ simpler dependencies (important for CC evaluation)
→ separation of concerns
Separation of code and specifications

→ better readability
→ simpler dependencies (important for CC evaluation)
→ separation of concerns

How to achieve separation?
→ do not use Hoare-style contracts

\[ \{P\} f \{Q\} \]

becomes a single separate lemma

\[ P \rightarrow f \rightarrow Q \]
Separation of code and specifications

→ better readability
→ simpler dependencies (important for CC evaluation)
→ separation of concerns

How to achieve separation?
→ do not use Hoare-style contracts

\[
\{P_1 \land P_2\} f \{Q_1 \land Q_2\}
\]

becomes a single separate lemma

\[
P_1 \rightarrow P_2 \rightarrow f \rightarrow Q_1 \land Q_2
\]
Separation of code and specifications

→ better readability
→ simpler dependencies (important for CC evaluation)
→ separation of concerns

How to achieve separation?
→ do not use Hoare-style contracts

\[ \{ P1 \land P2 \} f \{ Q1 \land Q2 \} \]

becomes two separate lemmas

\[ P1 \rightarrow P2 \rightarrow f \rightarrow Q1 \]
\[ P1 \rightarrow P2 \rightarrow f \rightarrow Q2 \]
Separation of code and specifications

→ better readability
→ simpler dependencies (important for CC evaluation)
→ separation of concerns

How to achieve separation?
→ do not use Hoare-style contracts

\[
\{P_1 \land P_2\} f \{Q_1 \land Q_2\}
\]

becomes two separate lemmas

\[
P_1 \rightarrow f \rightarrow Q_1
\]

\[
P_2 \rightarrow f \rightarrow Q_2
\]
Separation of code and specifications

→ better readability
→ simpler dependencies (important for CC evaluation)
→ separation of concerns

How to achieve separation?
→ do not use Hoare-style contracts

\[
\{ P1 \land P2 \} f \{ Q1 \land Q2 \}
\]

becomes two separate lemmas

\[
P1 \rightarrow f \rightarrow Q1
\]
\[
P2 \rightarrow f \rightarrow Q2
\]

→ how to get rid of loop invariants? (without getting rid of loops)
Inductive loop invariants

- Loop invariants hold at every iteration.
- Inductive loop properties are preserved by the loop.
- Reasoning about a loop means finding indutive loop invariants.
Inductive loop invariants

- loop invariants hold at every iteration
- inductive loop properties are preserved by the loop
  → reasoning about a loop means finding inductive loop invariants

Let $\mathcal{I}$ be the set of inductive loop invariants

- the conjunction of two inductive invariants is an inductive invariant
- $(\mathcal{I}, \supseteq)$ form a lattice, its join operation is the conjunction operator $\wedge$
- its bottom element is $\text{True}$, and its maximum element $\bigwedge_{I \in \mathcal{I}} I$ is what we call the most general inductive invariant (MGI)
Generating the MGI

MGI can be defined as the inductive closure of the relation which contains the loop initialization and which is closed by applying an iteration of the loop body.
Generating the MGI

Start

Initialization

Condition
- If condition is TRUE:
  Body of Loop
  Update
- If condition is FALSE:
  Update

Stop
Generating the MGI

Start
  ↓
Initialization
  ↓
Condition
    ↓
Body of loop
      ↓
Update
      ↖
If condition is TRUE
     ↓
Body of loop
     ↓
Update
     ↖
If condition is FALSE
  ↖
Stop

...
→ MGI can be defined as the inductive closure of the relation which contains the loop initialization and which is closed by applying an iteration of the loop body
MGI in practice

- this works with loops in sequence or even *nested loops*
MGIs in practice

- this works with loops in sequence or even *nested loops*
- one can specify the *frame* of the MGI, i.e. the variables that it should track
  - → tracking less variables means the MGI is not so general anymore, but applies to more loops
MGIs in practice

- this works with loops in sequence or even *nested loops*
- one can specify the *frame* of the MGI, i.e., the variables that it should track
  - → tracking less variables means the MGI is not so general anymore, but applies to more loops
- MGIs allow sharing proofs between “similar” loops
  - → if the MGI of some loop $\mathcal{L}$ is an invariant of some loop $\mathcal{L}'$, all invariants of $\mathcal{L}$ are invariants of $\mathcal{L}'$
MGIs in practice

- this works with loops in sequence or even *nested loops*
- one can specify the *frame* of the MGI, i.e. the variables that it should track
  - → tracking less variables means the MGI is not so general anymore, but applies to more loops
- MGIs allow sharing proofs between “similar” loops
  - → if the MGI of some loop $\mathcal{L}$ is an invariant of some loop $\mathcal{L}'$, all invariants of $\mathcal{L}$ are invariants of $\mathcal{L}'$
- MGI generation need not be trusted
MGIs in practice

- this works with loops in sequence or even *nested loops*
- one can specify the *frame* of the MGI, i.e. the variables that it should track
  - tracking less variables means the MGI is not so general anymore, but applies to more loops
- MGIs allow sharing proofs between “similar” loops
  - if the MGI of some loop $\mathcal{L}$ is an invariant of some loop $\mathcal{L}'$, all invariants of $\mathcal{L}$ are invariants of $\mathcal{L}'$
- MGI generation need not be trusted
- we use a similar trick to delay *termination proofs* for recursive predicates or internal loops
Conclusion

- good tooling is **key** to large verification project like ProvenCore
- ProvenTools is designed to meet our ends and make the project manageable
- we value separation of code and specs
  - → original way of dealing with loop reasoning
Conclusion

- good tooling is **key** to large verification project like ProvenCore
- ProvenTools is designed to meet our ends and make the project manageable
- we value separation of code and specs
  → original way of dealing with loop reasoning