Handling first-order proofs

Michael Raskin, raskin@mccme.ru
Christoph Welzel

Dept. of CS, TU Munich

April 1, 2019

The project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 787367

Overview

- Direction: algorithm and program verification
  Local goal: formal proofs for distributed algorithms
- Approach: first-order logic and help from automated systems
- Example: mutual exclusion
- Our tooling and workflow

General context

Some programs have bugs

More complex tasks; hardware rewards more complex approaches
Bugs harder to detect, reproduce, eliminate

Testing? Strict in-language restrictions (types, lifetimes, …)? Static analysis? Proofs?

All the approaches are useful — let developers choose

We want to expand the options for **proofs**

General context

Some programs have bugs

More complex tasks; hardware rewards more complex approaches
Bugs harder to detect, reproduce, eliminate

Testing? Strict in-language restrictions (types, lifetimes, …)? Static analysis? Proofs?

All the approaches are useful — let developers choose

We want to expand the options for **proofs**

Proofs: what kind of proofs?

We currently work on verifying distributed algorithm design via formal proofs

What kind of proofs and tools can we use?

- First-order logic proofs (with support from Automated Theorem Provers)

  More compatible tools, more automation available, simpler logic, cannot use theory-specific knowledge
- Proofs in specific first-order theories (via SMT)

  Reliance on complicated tools for a more complex problem
- Higher-order logic proofs (probably with interactive provers)

  More expressive logics — different for different tools

We use first-order logic and ATP systems

First-order logic and higher-order logic: illustration

Second-order logic puts function variables into language

```
(setf f (lambda (x) …))
(funcall f arg)
```

First-order logic: function variables not supported directly;
… so interface-passing style is needed

```
(defmethod call-f ((f (eql :something)) x) …)
(defmethod call-f ((f (eql :something-else)) x) …)
(setf f :something)
(call-f f arg)
```

Our current experiments don't use that (yet)

First-order logic and higher-order logic: illustration

Second-order logic puts function variables into language

```
        (setf f (lambda (x) …))
        (funcall f arg)
```

First-order logic: function variables not supported directly;
… so interface-passing style is needed

```
        (defmethod call-f ((f (eql :something)) x) …)
        (defmethod call-f ((f (eql :something-else)) x) …)
        (setf f :something)
        (call-f f arg)
```

Our current experiments don't use that (yet)

First-order logic — our view

Closest to what is used in mathematics and theoretical CS textbooks for proofs

… if everything goes well, reusing published proofs might become less work after some time

A lot of compatible tools — even with the same format
(Thousands of Problems for Theorem Provers — TPTP)

… hope of cross-verification

Most advanced proof search tools

Hard to do fully formal proofs — but can ask automated provers to finish proof details

Example: mutual exclusion — Dijkstra's algorithm

Our current experiments: around Dijkstra's mutex
(single CPU core, multi-threading)

Algorithm idea:
(Some code to resolve conflicts faster — who gets priority)
Process declares intent to enter critical section
Verifies no other process has declared same intent
Executes critical section
Cleans up declarations

Logical encoding:
Some theory of discrete time, and discrete list of agents
Variables — functions from time to values
Local variables — time and agent to values
State of the program (instruction pointer) — local variable
Single-threaded execution: active agent is a global variable

Example: mutual exclusion — Dijkstra's algorithm

Our current experiments: around Dijkstra's mutex
(single CPU core, multi-threading)

Algorithm idea:
(Some code to resolve conflicts faster — who gets priority)
Process declares intent to enter critical section
Verifies no other process has declared same intent
Executes critical section
Cleans up declarations

Logical encoding:
Some theory of discrete time, and discrete list of agents
Variables — functions from time to values
Local variables — time and agent to values
State of the program (instruction pointer) — local variable
Single-threaded execution: active agent is a global variable

Example: Dijkstra's mutex (cont.)

Process declares intent to enter critical section
Verifies no other process has declared same intent

Some theory of discrete time, and discrete list of agents
Variables — functions from time to values
Local variables — time and agent to values
State of the program (instruction pointer) — local variable
Single-threaded execution: active agent is a global variable

Want to prove:
Safety — two different agents cannot enter critical section simultaneously

Actually prove:
Define invariant; if it holds, it holds at the next moment
Invariant as defined implies safety

Proving induction step

Want to prove:
Safety — two different agents cannot enter critical section simultaneously

Actually prove:
Define invariant; if it holds now, it holds at the next moment
Invariant as defined implies safety

The base case tends to be obvious

Why prove only inductive step?
Simpler... and already hard enough for now

Easy to also prove the base case — still exploring the options for proving the (harder) step

Proving induction step

Want to prove:
Safety — two different agents cannot enter critical section simultaneously

Actually prove:
Define invariant; if it holds now, it holds at the next moment
Invariant as defined implies safety

The base case tends to be obvious

Why prove only inductive step?
Simpler... and already hard enough for now

Easy to also prove the base case — still exploring the options for proving the (harder) step

Why prove only inductive step?

Easy encoding of induction needs infinitely many axioms
Rich theories harder for proof search

Clear strategies for encoding first-order theories with induction
(for example, in an interface-passing style)
This is not our current priority (yet)

Working with a proof

Defining axioms (model + algorithm)

Defining the safety condition

Defining invariant

Claiming invariant is inductive
Claiming invariant implies safety

Adding/generating lemmas

Verifying the proof

Axioms

Must have manual part
(if we care what we prove…)

Basic theory + system behaviour

Basic theory: «order is transitive»
Execution model: «we can iterate over threads in order»
Behaviour: «this is how this variable is updated by that assignment»

Behaviour specification partially generated
has to be tuned if (when…) model changes

Axioms: (relatively) generic

«order is transitive»

```
fof(leq_transitive, axiom,
  ![X,Y,Z]: ((leq(X,Y)&leq(Y,Z))=>leq(X,Z))).
```

$\forall X, Y, Z : X \leq Y \wedge Y \leq Z \Rightarrow X \leq Z$

«we can iterate over threads in order»

```
fof(next_agent_exhaustive, axiom,
  ![X,Y]: ((is_agent_or_initial_or_failure(X)
            & is_agent_or_initial_or_failure(Y)
            & leq(X,Y) & leq(Y,next_agent(X)))
          =>(X=Y | next_agent(X)=Y))).
```

$\approx \forall X, Y : X \leq Y \wedge Y \leq X + 1 \Rightarrow X + 1 = Y \vee X = Y$

Axioms: algorithm

«this is how this variable is updated by that assignment»

```
fof(counter_structure_at_next_moment_3, axiom,
  ![T,A]: (((~(active_agent(T) != A))
             & (~((~is_moment(next_moment(T)))
                    | (~is_agent(A))))
             & (active_state(T,A) = step))
            => counter(next_moment(T),A) =
                             next_agent(counter(T,A)))).
```

$$\approx \forall moment\, T, agent\, A :$$
$$active\_agent(T) = A \wedge active\_state(T,A) = step$$
$$\Rightarrow counter(T+1,A) = counter(T,A)+1$$

Safety condition

Must be hand-picked — this is what we care about!

Usually simple
«Two different agents cannot be in critical section at once»

```
fof(define_safety_for, checked_definition,
  ![T,A1,A2]: (safe_for(T,A1,A2)<=>(
        (active_state(T,A1)=criticalSection
         & active_state(T,A2)=criticalSection)
        => A1=A2))).
fof(define_safety, checked_definition,
  ![T]: (safe(T)<=>(![A1,A2]: safe_for(T,A1,A2)))).
```

$$\forall T, A_1, A_2 : (\mathit{safe\_for}(T, A_1, A_2) \Leftrightarrow$$
$$((\mathit{state}(T, A_1) = \mathit{state}(T, A_2) = \mathit{criticalSection}) \Rightarrow A_1 = A_2))$$

Invariant

Some technical additions to safety…

```
fof(define_inside, checked_definition,
  ![T,A]: (inside(T,A) <=> (
    is_moment(T) & is_agent(A) &
    (active_state(T,A) = startCheck
     | active_state(T,A) = selfCheck
     | …)))).
fof(define_inside_correct_for, checked_definition,
  ![T, A]: (
    inside_correct_for(T, A) <=>
      (inside(T,A) => outside(T,A) = false))).
```

(per-thread variable *outside* is consistent with state)

Invariant

… and some key proof ideas

```
fof(define_passed, checked_definition,
  ![T,A,B]: (passed(T,A,B)<=>(instant_agent_pair(T,A,B)
    & counter_scope(T, A) & (leq(B,counter(T,A)))
    & (B=counter(T,A) => active_state(T,A)=step)))).

fof(define_passed_exclusive_for, checked_definition,
  ![T, A, B]: (passed_exclusive_for(T, A, B) <=>
     ~(passed(T,A,B) & passed(T,B,A)))).
```

instant_agent_pair: typing/non-aliasing conditions
counter_scope: program part where *counter* is used

If two agents check each other, one sees the other having started check

Invariants

Currently added manually
Exploring ways to generate some of them

Some classes of algorithms known to be verifiable automatically (under some assumptions)

Lemmas

If we have infinite computing power, lemmas are not needed

... or if we had perfect proof search algorithms

Currently added manually or semi-manually

Lemmas: manual addition

```
fof(invariant_safety_local, checked_lemma,
  ![T,A,B]: ((passed_exclusive_for(T,A,B)
             & passed_in_critical(T))
       => safe_for(T,A,B))).
```

(unfolding quantifier, specifying the dependencies)

Yes, this makes the proof much faster

Lemmas: guided generation

```
fof(invariant_preserved, checked_lemma,
  ![T]: (invariant(T) => invariant(next_moment(T)))).
tpi(ed_invariant_preserved, expand_specific_definitions,
  invariant_preserved(invariant)).
…
tpi(sc_all, split_all_conjunctions, '-').
```

Expand specific definition before proving

… which makes the expansion eligible for conjunction-splitting

```
fof(sc_ed_invariant_preserved_expanded_…_…,
  ![T]:((inside_correct(T) & passed_exclusive(T)
        & passed_in_critical(T))
     =>passed_in_critical(next_moment(T)))).
```

Lemmas: guided generation

Another example — too big to show — case analysis

Prove «A or B»,
then «if A then C» and «if B then C»,
then «C»

Lemmas: automatic generation

Would be nice

Some facts easy to generate

May be useful even if insufficient

(Future direction)

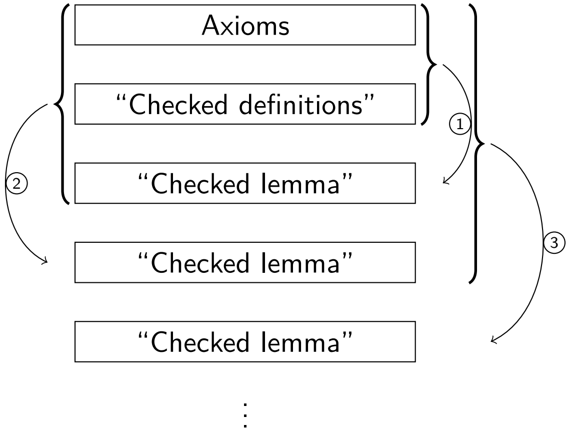
Verifying the proof

Running the proof script — as a script
(automated, of course)

Read entire input, apply instructions for lemma generation

For each lemma, make ATP prove it from axioms and previous lemmas
    ... each request is completely self-sufficient and independent
Use that lemma as an axiom in the future

Lemma verification loop



Arrows are ATP invocations

Further guidance

User can specify which axioms (and previous lemmas) some lemma needs

Guided lemma generation generates such hints

Can make proof search much faster

Once the proof is verified…

If ATP prints proofs (not all do…), we can:
Export dependency graph of steps taken when proving a single lemma
… and a dependency graph between axioms/lemmas in the entire proof

Everything is complicated

- Different provers have different power
  and different functionality around proof search
- Global rankings do not reflect fitness for specific purpose
- Proof search is a dark art based on heuristics
  useless lemmas can be useful for guiding search...

Thanks for your attention!

Questions?

https://gitlab.common-lisp.net/mraskin/gen-fof-proof