CS720

Logical Foundations of Computer Science

Lecture 20: How to verify?

Tiago Cogumbreiro

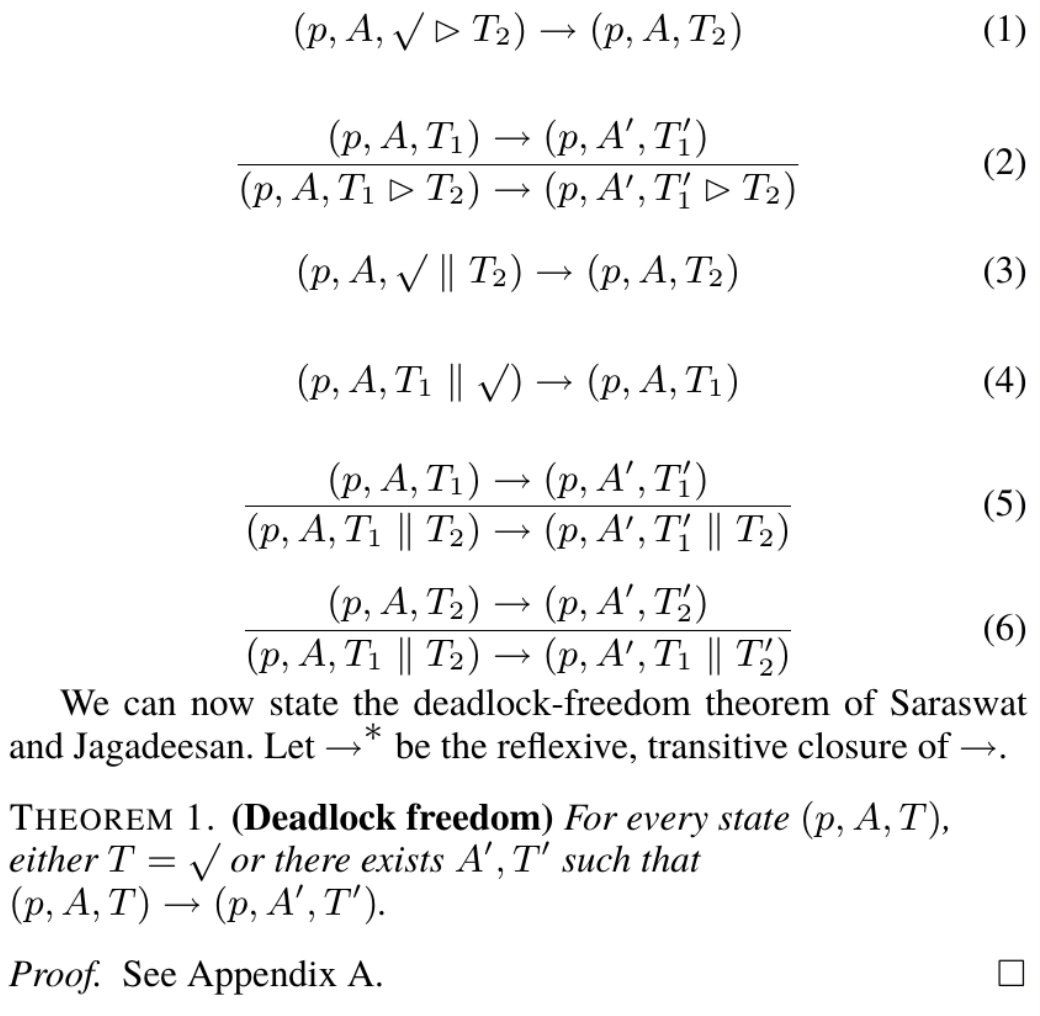
HW9/HW10 recap

HW9/HW10

> Our goal (homework) is to formalize and prove Theorem 1, for an ***abstract expression language*** that enjoys strong progress. We will also introduce a type system to identify sequential programs.

Featherweight X10: A Core Calculus for Async-Finish Parallelism. Jonathan K. Lee, Jens Palsberg. In PPoPP'10. DOI: 10.1145/1693453.1693459.

- Our language does not have arrays, nor function calls, nor imperative features



$$(p, A, \surd \rhd T_2) \to (p, A, T_2) \tag{1}$$

$$\frac{(p, A, T_1) \to (p, A', T_1')}{(p, A, T_1 \rhd T_2) \to (p, A', T_1' \rhd T_2)} \tag{2}$$

$$(p, A, \surd \parallel T_2) \to (p, A, T_2) \tag{3}$$

$$(p, A, T_1 \parallel \surd) \to (p, A, T_1) \tag{4}$$

$$\frac{(p, A, T_1) \to (p, A', T_1')}{(p, A, T_1 \parallel T_2) \to (p, A', T_1' \parallel T_2)} \tag{5}$$

$$\frac{(p, A, T_2) \to (p, A', T_2')}{(p, A, T_1 \parallel T_2) \to (p, A', T_1 \parallel T_2')} \tag{6}$$

We can now state the deadlock-freedom theorem of Saraswat and Jagadeesan. Let $\to^*$ be the reflexive, transitive closure of $\to$.

THEOREM 1. **(Deadlock freedom)** *For every state* (p, A, T)*, either* $T = \surd$ *or there exists* A', T' *such that* $(p, A, T) \to (p, A', T')$.

Proof. See Appendix A. □

Language

See Figure 1

A statement:

$$s ::= \mathbf{skip} \mid e; s \mid \mathbf{async}\{s\}; s \mid \mathbf{finish}\{s\}; s$$

A task tree:

$$T ::= T \rhd T \mid T \,||\, T \mid \langle s \rangle \mid \surd$$

Small-step semantics for commands

See Figure 2

$$\frac{e \Rightarrow e'}{e; c \Rightarrow \langle e'; c \rangle}$$

$$\frac{\mathrm{value}(e)}{e; c \Rightarrow \langle c \rangle} \qquad \frac{}{\mathtt{skip} \Rightarrow \surd}$$

$$\frac{}{\mathtt{async}\{c_1\}; c_2 \Rightarrow \langle c_1 \rangle \mid\mid \langle c_2 \rangle} \qquad \frac{}{\mathtt{finish}\{c_1\}; c_2 \Rightarrow \langle c_1 \rangle \rhd \langle c_2 \rangle}$$

Small-step semantics for trees

See rules (1) to (6) in page 28

$$\frac{}{\surd \triangleright T \Rightarrow T} \qquad \frac{T_1 \Rightarrow T_1'}{T_1 \triangleright T_2 \Rightarrow T_1' \triangleright T_2}$$

$$\frac{}{\surd \,||\, T \Rightarrow T} \qquad \frac{}{T \,||\, \surd \Rightarrow T}$$

$$\frac{T_1 \Rightarrow T_1'}{T_1 \,||\, T_2 \Rightarrow T_1' \,||\, T_2} \qquad \frac{T_2 \Rightarrow T_2'}{T_1 \,||\, T_2 \Rightarrow T_1 \,||\, T_2'}$$

$$\frac{c \Rightarrow T}{\langle c \rangle \Rightarrow T}$$

How to verify?

What can I use?

Road map

- What kind of problem do you have?
- How much do you know of the code?
- Let me guide you through various verification techniques

> **Disclaimer:** This is not a comprehensive list. Many of the techniques covered may be useful in different contexts.

Black-box testing

- **Context:** No access to the source code
- **Goal:** Does the program behave unexpectedly?

Black-box testing

- **Context:** No access to the source code
- **Goal:** Does the program behave unexpectedly?

Try **fuzzing:** randomized testing to search for bugs

- generate random inputs, check if the tool's behaviors
- generate random inputs, compare multiple tool's outputs (languages are starting to include fuzzing, eg go)
- Research questions:
  - how to generate interesting inputs?
  - can we use the source code to guide code generation?
  - compiler fuzzing [OOPSLA19]

White-box testing

- **Context:** Have access to source code, small domain knowledge
- **Goal:** Does the program behave unexpectedly?

White-box testing

- **Context:** Have access to source code, small domain knowledge
- **Goal:** Does the program behave unexpectedly?

Try **property testing**

- Define "theorems" as test cases
- Has the notion of ∀ binders through sampling

```
from hypothesis import given
from hypothesis.strategies import text

@given(text())
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

White-box testing

- **Context:** Have access to source code, small domain knowledge
- **Goal:** Does the program behave unexpectedly?

White-box testing

- **Context:** Have access to source code, small domain knowledge
- **Goal:** Does the program behave unexpectedly?

Try **symbolic execution**

- runs program with "symbolic variables"
- tries to iterate over all possible executions
- groups executions and reports input/output pairs
- we can include asserts to test some conditions
- we can test outputs

Symbolic execution

Klee tutorial

See Symbolic Execution for Software Testing

```
int get_sign(int x) {
  if (x == 0) return 0;
  if (x < 0) return -1;
  else return 1;
}
```

- generates a test-case **per output**
- will try to exercise **all paths** of the code
- analysis may not terminate, relies on SAT solvers which may give up
- reports errors (memory safety, exit codes, etc)
- even with partial results, may be useful (like fuzzing is)

Hoare logic

- Add pre-/post- conditions to regular languages
- Tool will prove that they are met for **all** inputs
- Dafny, F*, Why3, Frama-C
- Challenging when the tool ***cannot*** prove the results

```
let malloc_copy_free (len:uint32 { 0ul < len })
                     (src:lbuffer len uint8)
  : ST (lbuffer len uint8)
       (requires fun h ->
         live h src /\
         freeable src)
       (ensures fun h0 dest h1 ->
         live h1 dest /\
         (forall (j:uint32). j < len ==> get h0 src j == get h1 dest j))
  = let dest = malloc 0uy len in
    memcpy len 0ul src dest;
    free src;
    dest
```

Model checking

- **Context:** Have access to source code and understand the code
- **Goal:** Can we assert something for every possible execution?

Model checking

- **Context:** Have access to source code and understand the code
- **Goal:** Can we assert something for every possible execution?
- Symbolic execution allows us to search for one possible bad execution ($\exists$)
- Model checking lets us brute force **all** execution paths ($\forall$)
- Limited to small problem sizes
- Usually a domain-specific language
- Write an algorithm in a model checking language, prove that a certain assertion is always met
- Struggles with unbounded data
- Success stories: locking algorithms, distributed systems, hardware circuits

Model checking

TLA+: Arbitrage example

```
while actions < MaxActions do
  either
    Buy:
      with v \in V, i \in Items \ backpack do
      profit := profit - market[≪v, i≫].sell;
      backpack := backpack \union {i};
      end with;
  or
    Sell:
      with v \in V, i \in backpack do
        profit := profit + market[≪v, i≫].buy;
        backpack := backpack \ {i};
      end with;
  end either;
  Loop:
    actions := actions + 1;
end while;
\*  Is there a potential for arbitrage?
NoArbitrage == profit ≤ 0
```

SAT solvers

- When you can reduce your problem into a formula
- SMTLIB2/Z3
- Rosette: a solver-aided programming language that extends Racket
- Many verification tools use SAT solvers behind the scenes (eg, symbex)

```
x = Int('x')
y = Int('y')

s = Solver()
s.add(x > 2)
s.add(y < 10)
s.add(x + 2 *y == 7)

print(s.check())
print(s.model())
# sat
# [y = 0, x = 7]
```

Datalog

- Graph-based problems
- Queries of interesting relations
- Souffle; Formulog is datalog+SMT solver

```
.decl alias( a:var, b:var ) output
alias(X,X) :- assign(X,_).
alias(X,X) :- assign(_,X).
alias(X,Y) :- assign(X,Y).
alias(X,Y) :- ld(X,A,F), alias(A,B), st(B,F,Y).

.decl pointsTo( a:var, o:obj )
.output pointsTo
pointsTo(X,Y) :- new(X,Y).
pointsTo(X,Y) :- alias(X,Z), pointsTo(Z,Y).
```

Proof assistants

- Full control of the theory
- Limited support to generating executable code

```
Theorem Rice
  (P : input -> Prop)
  (nt: Nontrivial P):
  (forall M M',
    (forall i,
      Run (Call M i) true <-> Run (Call M' i) true
    ) ->
    P (encode_mach M) <-> P (encode_mach M')
  ) ->
  ~ Decidable P.
```

```
q: {push empty;
    spawn q1;}
q1: {
  if (read a)
    {push a; loop;}
  spawn q2;
}
q2: {
  if (read b)
    {push b; loop;}
  if (pop empty)
    {spawn q3;}
}
q3: {return true;}
```