Logical Foundations of Computer Science

Lecture 20: How to verify?

Tiago Cogumbreiro
HW9/HW10 recap
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.


- Our language does not have arrays, nor function calls, nor imperative features
\[(p, A, \sqrt \triangleright T_2) \rightarrow (p, A, T_2)\] (1)

\[(p, A, T_1) \rightarrow (p, A', T'_1)\]
\[(p, A, T_1 \triangleright T_2) \rightarrow (p, A', T'_1 \triangleright T_2)\] (2)

\[(p, A, \sqrt \parallel T_2) \rightarrow (p, A, T_2)\] (3)

\[(p, A, T_1 \parallel \sqrt) \rightarrow (p, A, T_1)\] (4)

\[(p, A, T_1) \rightarrow (p, A', T'_1)\]
\[(p, A, T_1 \parallel T_2) \rightarrow (p, A', T'_1 \parallel T_2)\] (5)

\[(p, A, T_2) \rightarrow (p, A', T'_2)\]
\[(p, A, T_1 \parallel T_2) \rightarrow (p, A', T_1 \parallel T'_2)\] (6)

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

**Theorem 1. (Deadlock freedom)** For every state \((p, A, T)\), either \(T = \sqrt\) or there exists \(A', T'\) such that \((p, A, T) \rightarrow (p, A', T')\).

*Proof.* See Appendix A. \(\square\)
Language

See Figure 1

A statement:

\[ s ::= \text{skip} \mid e; s \mid \text{async}\{s\}; s \mid \text{finish}\{s\}; s \]

A task tree:

\[ T ::= T \triangleright T \mid T \parallel T \mid \langle s \rangle \mid \sqrt{ } \]
Small-step semantics for commands

See Figure 2

\[
\frac{e \Rightarrow e'}{e; c \Rightarrow \langle e'; c \rangle}
\]

\[
\frac{\text{value}(e)}{e; c \Rightarrow \langle c \rangle}
\]

\[
\frac{\text{skip} \Rightarrow \sqrt{}}{e; c \Rightarrow \langle c \rangle}
\]

\[
\frac{\text{async}\{c_1\}; c_2 \Rightarrow \langle c_1 \rangle \| \langle c_2 \rangle}{e; c \Rightarrow \langle c \rangle}
\]

\[
\frac{\text{finish}\{c_1\}; c_2 \Rightarrow \langle c_1 \rangle \triangleright \langle c_2 \rangle}{e; c \Rightarrow \langle c \rangle}
\]
Small-step semantics for trees

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

\[
\begin{align*}
\sqrt \triangleright T & \Rightarrow T \\
T_1 \triangleright T_2 & \Rightarrow T_1' \triangleright T_2 \\
\sqrt \parallel T & \Rightarrow T \\
T \parallel \sqrt & \Rightarrow T \\
T_1 \Rightarrow T_1' \\
T_1 \parallel T_2 & \Rightarrow T_1' \parallel T_2 \\
T_2 \Rightarrow T_2' \\
T_1 \parallel T_2 & \Rightarrow T_1 \parallel T_2' \\
c \Rightarrow T \\
\langle c \rangle \Rightarrow T
\end{align*}
\]
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 $\forall$ binders through sampling

```python
from hypothesis import given
from hypothesis.strategies import 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

```c
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

```plaintext
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**

```plaintext
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?\n
NoArbitrage == profit \leq 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)

```python
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

```plaintext
.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