# Modeling Complex Semantic Properties

Viktor Kunčak

## Two Procedures and Two Meanings (for Integer Variable)

```
def f =
  if (x > 0) {
    x = x − 1
    f
  }
```

$R_f = \{(x, x') \mid (x \leq 0 \wedge x' = x) \vee (x > 0 \wedge x' = 0)\}$ (makes $x$ zero)

```
def g =
  if (x > 0) {
    x = x − 2
    g
  }
```

$R_g = \{(x, x') \mid (x \leq 0 \wedge x' = x) \vee (x > 0 \wedge -1 \leq x' \leq 0 \wedge 2 \mid (x' - x))\}$
(makes $x$ zero or $-1$, preserving its parity)

## Concurrency: Need to Model Interleaving

## If we execute $f, g$ concurrently, can we go from 3 to $-2$?

```
def f =
  if (x > 0) {
    x = x − 1
    f
  }
```

$R_f = \{(x, x') \mid (x \le 0 \land x' = x) \lor (x > 0 \land x' = 0)\}$ (makes $x$ zero)

```
def g =
  if (x > 0) {
    x = x − 2
    g
  }
```

$R_g = \{(x, x') \mid (x \le 0 \land x' = x) \lor (x > 0 \land -1 \le x' \le 0 \land 2 \mid (x' - x))\}$
(makes $x$ zero or $-1$, preserving its parity)

## From 3 to −2

```
def f =
  if (x > 0) {
    x = x − 1
    f
  }
```

```
def g =
  if (x > 0) {
    x = x − 2
    g
  }
```

| step | thread | x | command | x' |
|---|---|---|---|---|
| 1 | g | 3 | if (x>0) | 3 |
| 2 | g | 3 | x = x - 2 | 1 |
| 3 | f | 1 | if (x>0) | 1 |
| 4 | g | 1 | if (x>0) | 1 |
| 5 | f | 1 | x = x - 1 | 0 |
| 6 | g | 0 | x = x - 2 | -2 |

$R_f = \{(x, x') \mid (x \le 0 \wedge x' = x) \vee (x > 0 \wedge x' = 0)\}$
$R_g = \{(x, x') \mid (x \le 0 \wedge x' = x) \vee (x > 0 \wedge -1 \le x' \le 0 \wedge 2 \mid (x' - x))\}$
No good way to define $R$ where $(3, -2) \in R$ from $R_f$ and $R_g$. Operators $\circ, \cup, \cap$ no good

## Interleaving Trace Semantics under Sequential Consistency Model

Introduce a letter for each each assignment, test, calls and returns in program:

```
def f =
  if (x > 0) {
    x = x − 1
    f
  }
```

```
def g =
  if (x > 0) {
    x = x − 2
    g
  }
```

Meaning of each procedure as the context-free language; each word describes a trace:

$L_f = \{f_1 f_2 f_5, f_1 f_2 f_3 f_4 f_1 f_2 f_5 f_5, \ldots\}$ $\qquad$ $L_g = \{g_1 g_2 g_5, g_1 g_2 g_3 g_4 g_1 g_2 g_5 g_5, \ldots\}$

$L_f ::= f_1 f_2 (f_3 f_4 L_f | \varepsilon) f_5$ $\qquad$ $L_g ::= g_1 g_2 (g_3 g_4 L_g | \varepsilon) g_5$

$L = L_f \,||\, L_g$ (parallel composition)

$X \,||\, Y = \bigcup_{x \in X, y \in Y} x \,||\, y$ where:

- $\varepsilon \,||\, y = \{y\}$
- $x \,||\, \varepsilon = \{x\}$
- $ax \,||\, by = \{aw \mid w \in x \,||\, by\} \cup \{bw \mid w \in ax \,||\, y\}$ (either $a$ or $b$ is first)

## Difficulties

Semantics is complicated. Some traces infeasible

Inductive invariants complicated, refer to program points

Few decision procedures exist when traces have no bound on length

Actual semantics is more difficult due to relaxed models
(C++11, TSO, Java memory model)

Often automated tools can only check properties for a finite number of states or up to a bounded depth:
GenMC: A Model Checker for Weak Memory Models

## Verifying an Interpreter for a Concurrent Program

Program schedule: a list that determines which concurrent thread executes in each step

- Write a simulator program that takes a schedule and simulates the execution of a concurrent program with this schedule.
- Prove that the simulator returns program state that satisfies a given invariant

The methodology works for other program features as well.
Reduces verification of programs in a complex programming language to verification in a simple language, the language of the interpreter.

Example successful application:
Java Program Verification via a JVM Deep Embedding in ACL2
See also talk “Machines Reasoning about Machines”

# Interpreter for An Example: Encoding

```
  def f =
1 while (x > 0) {
2 x = x − 1
3 } goto 3

  def g =
1 while (x > 0) {
2 x = x − 2
3 } goto 3

case class State(
  x: BigInt,
  pc_f: BigInt, // prog. counter of f
  pc_g: BigInt) // prog. counter of g

def valid(st:State): Boolean =
  1 <= st.pc_f && st.pc_f <= 3 &&
  1 <= st.pc_g && st.pc_g <= 3
```

```
def stepF(st: State): State = {
  require(valid(st))
  if (st.pc_f == 1) {
    if (st.x > 0) st.copy(pc_f = 2)
    else st.copy(pc_f = 3)
  } else if (st.pc_f == 2)
    st.copy(x = st.x − 1, pc_f=1)
  else st
} ensuring(valid(_))

def stepG(st: State): State = {
  require(valid(st))
  if (st.pc_g == 1) {
    if (st.x > 0) st.copy(pc_g = 2)
    else st.copy(pc_g = 3)
  } else if (st.pc_g == 2)
    st.copy(x = st.x − 2, pc_g=1)
  else st
} ensuring(valid(_))
```

# Running Interpreter on a Schedule

```
sealed abstract class ThreadChoice
case class F() extends ThreadChoice
case class G() extends ThreadChoice
type Schedule = List[ThreadChoice]

def run(st: State, sch: Schedule): State = {
  require(valid(st))
  sch match {
    case Nil() => st
    case Cons(c, rest) => run(runOne(st, c), rest)
  }
} ensuring(valid(_))

// We can get −2
val init1= State(BigInt(3), BigInt(1), BigInt(1))
val sch1: Schedule = List(G(), G(), G(), F(), F(), G(), F(), G())
def test1 = run(init1, sch1)
  // State(BigInt("−2"), BigInt("3"), BigInt("3"))
```

## An Inductive Invariant

```
def nice(st: State): Boolean = {
  (st.x > 0) ||
  (st.pc_f != 2 && st.pc_g != 2 && −2 <= st.x) ||
  (st.pc_f == 2 && st.pc_g != 2 && −1 <= st.x) ||
  (st.pc_g == 2 && st.pc_f != 2 && 0 <= st.x)
}

// being nice is inductive
def runNice(st: State, sch: Schedule): State = {
  require(valid(st) && nice(st))
  sch match {
    case Nil() => st
    case Cons(c, rest) => runNice(runOne(st, c), rest)
  }
} ensuring(res => valid(res) && nice(res) && res == run(st,sch))
```

If, for example, programs start with $x > 0$ then if both programs reach end (pc=3), the value is −2 or more.