Reasoning about Programs

Murat Demirbas

[2019-08-28 Wed]

Outline

Trust me on this detour

http://www.youtube.com/watch?v=3PycZtfns_U
http://www.youtube.com/watch?v=eWMtUDJQfYs

Computation model

Programs

Programs consist of two things:

1. A set of typed variables.
2. A finite set of assignments (also called "actions").

By convention, all programs implicitly contain skip, which leaves the state unchanged

Programs may also contain some initial conditions, under section initially. The initially section is a predicate (on the program variables).

```
A predicate is function whose range is boolean.
```

Program Trivial

Program Trivial
var x, y : int
assign
x := 4
[] y:=f.7

Guarded actions

The execution of an assignment can be conditional on a predicate (called the "guard") being true. For example:

$x > 0 \longrightarrow x := 4$

The guard is a predicate on the state space of the program. If the guard is true in some state, the action is said to be "enabled" in that state.

Program execution

- A program can begin in any state satisfying the initially predicate
- An action is nondeterministically selected & executed atomically

Once this action has completed, an action is again non-deterministically selected and executed. This process is repeated infinitely.

If the selected action is guarded and the guard is false, the action is simply a skip

Distributed execution concerns

Atomic execution of an action is a simplifying assumption. In reality, we have to ensure that.

- E.g., we can use locking on the program variables, and try to acquire the lock on those variables in the guard, before executing the body of the action.

If we write our program in the read-write model (a node j can read from its neighbors but write only to its own variables), it is easy to transform this to message-passing model with some care (proper synchronization)

Termination

The nondeterministic selection of actions continues without end. There is no "last" action picked and executed.

We declare termination when the arrival of the program execution to a fixed point (FP). A fixed point is a state in which the execution of any action leaves the state unchanged.

assign
x := y
[] y:=f.7

FP is $y = f.7 \wedge x = y$

FP for guarded actions

assign
$x \geq 0 \longrightarrow x := 4$
$[\!] \; y := f.7$

FP is ???

FP for guarded actions

assign
$x \geq 0 \longrightarrow x := 4$
$[\!] \; y := f.7$

FP is $y = f.7 \wedge (x < 0 \vee x = 4)$

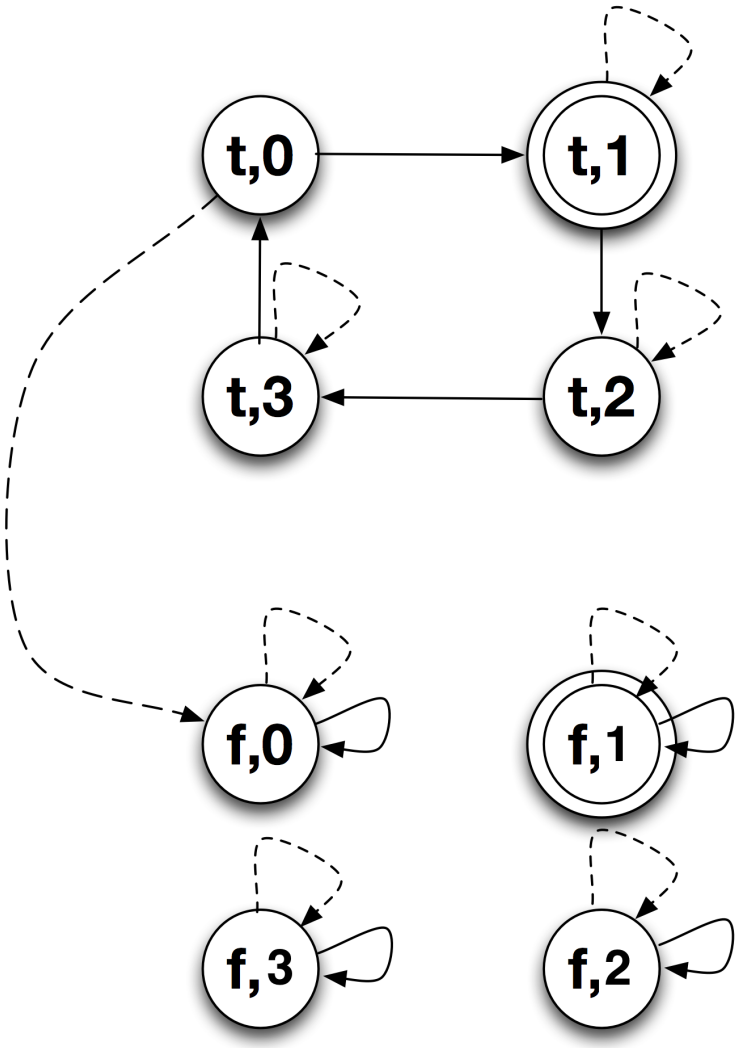
To calculate FP, we require each action to have no effect. So, for a guarded action of the form $g \longrightarrow x := E$, we have FP= $\neg g \vee (x = E)$.

Visualizing a program

Program Example1
var
b : boolean,
n : $\{0, 1, 2, 3\}$
initially n $= 1$
assign
$b \longrightarrow n := n +_4 1$
$[]\, n = 0 \longrightarrow b := \mathit{false}$

Draw the state diagram

Visualizing a program



Weak Fairness

Under weak fairness, every action is guaranteed to be selected infinitely often.

This means that between any two selections of some particular action (A), there are a finite (but unbounded) number of selections of other (i.e., not A) actions.

In Program Example1 above, if the execution starts at $\langle t, 1 \rangle$, can we say that the program reaches to FP under weak fairness?

Strong Fairness

Strong fairness requires that

- each action be selected infinitely often (like weak fairness), and
- if an action is enabled infinitely often, it is selected infinitely often when enabled.

Under strong fairness Example1 reaches FP.

If we prove some property, P under weak fairness, then P is also a property of the program under strong fairness. Converse is not true. We will assume weak fairness model, unless otherwise noted.

Reasoning about programs

Specification

Reasoning about a program involves showing that the program meets its specification. A specification is a high-level description of program behavior. E.g. Invariant

We will use program properties to specify our programs
A program property is a predicate on an execution

We say that a program property R holds for a program P exactly when R holds for every possible execution of P

Safety & Progress properties

There are two fundamental categories of program properties that we will use to describe program behavior:

- safety properties
- progress properties

Before considering these properties on computations (consisting of an infinite sequence of actions), we first examine how to reason about the behavior of a single action, executed once.

Hoare triples

A Hoare triple is an expression of the form
$\{P\}S\{Q\}$

where S is a program action and P and Q are predicates. The predicate P is often called the "precondition" and Q the "postcondition".

Informally, this triple says that if S begins execution in a state satisfying predicate P, it is guaranteed to terminate in a state satisfying Q.

Assignment axiom

Saying that a triple $\{P\}$ x := y + 1 {even.x} holds is the same as saying:
$[P \Rightarrow even.(y+1)]$

To prove $\{P\}$ x := E $\{Q\}$, we must show $[P \Rightarrow Q^x_E]$.

(Q^x_E is used to indicate writing expression Q with all occurrences of x replaced by E).

Example

Does this triple hold?

$\{x \geq -2\}$ x:=x-y+3 $\{x + y \geq 0\}$

Guarded actions

To prove the triple $\{P\}$ g $\longrightarrow$ x := E $\{Q\}$, we need to show:

$$[(P \wedge g \Rightarrow Q_E^x) \wedge (P \wedge \neg g \Rightarrow Q)]$$

Example

$$\{P\}\ x > y \quad \longrightarrow \quad x, y := y, x\ \{x > 3\}$$

What is P?

Safety properties

Safety properties

A safety property is a property that can be violated by a finite computation

- Invariant property is a safety property
- Two processes are not in critical section concurrently is a safety property
- The program will terminate eventually is NOT a safety property

Next

A next property (i.e., a predicate on programs) is written:
P next Q
where P and Q are predicates on states in the program

P next Q means that if a program is in a state satisfying P, its very next state (i.e., after choosing and executing exactly one action) must satisfy Q

Since any action could be chosen as the next one to be executed, we must show that for every action, if it begins in P, must terminate in Q

Next (proof rule)

To prove
$(P \text{ next } Q).\mathrm{G}$
we must show
$(\forall a : a \in G : \{P\}a\{Q\})$

Since *skip* is always part of any program, we have $P \Rightarrow Q$

Next theorems that hold for any program G

- false next Q
- P next true
- $(P1 \text{ next } Q1) \wedge (P2 \text{ next } Q2) \Rightarrow (P1 \wedge P2) \text{ next } (Q1 \wedge Q2)$
- $(P1 \text{ next } Q1) \wedge (P2 \text{ next } Q2) \Rightarrow (P1 \vee P2) \text{ next } (Q1 \vee Q2)$
- $(P \text{ next } Q) \wedge [Q \Rightarrow Q'] \Rightarrow (P \text{ next } Q')$
- $(P \text{ next } Q) \wedge [P' \Rightarrow P] \Rightarrow (P' \text{ next } Q)$

Stable

stable.P means that once P becomes true, it remains true.
stable.$P \equiv P$ next P

- stable.true
- stable.false
- stable.$P \wedge$ stable.$Q \quad \Rightarrow \quad$ stable.$(P \wedge Q)$
- stable.$P \wedge$ stable.$Q \quad \Rightarrow \quad$ stable.$(P \vee Q)$
- ??? stable.$P \wedge [P \Rightarrow P'] \quad \Rightarrow \quad$ stable.P'
- ??? stable.$P \wedge [P' \Rightarrow P] \quad \Rightarrow \quad$ stable.P'

Invariant

$\text{invariant}.P \equiv \text{initially}.P \wedge \text{stable}.P$

Invariant property is very important for reasoning about safety of your program.

Progress properties

Unlike safety, a progress (liveness) property can not be violated by a finite execution.

Progress is a predicate on possible computation suffixes.

All program properties of interest can be expressed as a conjunction of safety and progress.

Transient

$$\text{transient}.P.G \equiv$$
$$(\exists a : a \in G : \{P\}a\{\neg P\})$$

$$\text{transient}.P \wedge [P' \Rightarrow P] \Rightarrow \text{transient}.P'$$
$$\text{transient}.P \wedge [P \Rightarrow P'] \Rightarrow \text{transient}.P' \text{ ???}$$

Transient (example)

$even.x \longrightarrow x := x + 1$

transient.$(x = 2)$?

$n \leq 2 \longrightarrow n := n + 1$

transient.$(n = 0 \vee n = 1)$???

Ensures

P ensures Q means that if P holds, it will continue to hold so long as Q does not hold, and eventually Q does hold.

$$P \text{ ensures } Q \equiv ((P \wedge \neg Q) \text{ next } (P \vee Q)) \wedge \text{ transient}.(P \wedge \neg Q)$$

Ensures (example)

$even.x \longrightarrow x := x + 1$

$(x = 2 \vee x = 6)$ ensures $(x = 3 \vee x = 7)$?

$n \leq 2 \longrightarrow n := n + 1$

$n = 1$ ensures $n = 3$???

Leadsto

$P \leadsto Q$ means that if P is true at some point, Q will be true (at that same or a later point) in the computation.

P ensures $Q \Rightarrow P \leadsto Q$

$(P \leadsto Q) \wedge (Q \leadsto R) \Rightarrow P \leadsto R$

Leadsto

What is the relation between:

transient.P

$P \leadsto \neg P$

Leadsto

- $P \leadsto \mathit{true}$
- $\mathit{false} \leadsto P$
- $P \leadsto P$
- $(P \leadsto Q) \wedge [Q \Rightarrow Q'] \Rightarrow P \leadsto Q'$
- $(P \leadsto Q) \wedge [P' \Rightarrow P] \Rightarrow P' \leadsto Q$
- stable.$P \wedge$ trans.$(P \wedge \neg Q) \Rightarrow P \leadsto (P \wedge Q)$
- ??? $(P \leadsto Q) \wedge (P' \leadsto Q') \Rightarrow (P \wedge P') \leadsto (Q \wedge Q')$

Induction

A metric (or "variant function") is a function from the state space to a well-founded set (e.g., the natural numbers).
The well-foundedness of the range means that the value of the function is bounded below (i.e., can only decrease a finite number of times).
Theorem 10 (Induction). For a metric M,
$(\forall m :: P \wedge M = m \ \leadsto \ (P \wedge M < m) \vee Q) \ \Rightarrow \ P \ \leadsto \ Q$

Induction

Theorem 11. For a metric M,

$$(\forall m :: P \wedge M = m \text{ next } (P \wedge M \leq m) \vee Q)$$
$$\wedge(\forall m :: \text{transient}.(P \wedge M = m))$$
$$\Rightarrow \; P \rightsquigarrow Q$$

$$(\forall i, m :: \{P \wedge M = m \wedge g_i\} g_i \longrightarrow a_i\{(P \wedge M < m) \vee FP\})$$
$$\Rightarrow \; P \rightsquigarrow FP$$