Software Verification

Matthew Parkinson
Lent 2010 (16 Lectures)

Part III: Concurrency

I recommend you read Peter O'Hearn's paper on concurrent separation logic:

Resources, Concurrency and Local Reasoning
http://www.eecs.qmul.ac.uk/~ohearn/papers/concurrency.pdf

O'Hearn has some tutorial slides that accompany the paper
http://www.eecs.qmul.ac.uk/~ohearn/papers/etapstalk.pdf

Francesco Zappa-Nardelli has some very nice slides on concurrency verification:
http://moscova.inria.fr/~zappa/teaching/mpri/2009/
Many of the slides in this section of the notes have been adapted from this course.

Overview

Part 3: Concurrency
- Concurrency Examples
- Disjoint Concurrency
- Concurrent Separation Logic
- Owicki/Gries method
- Rely-guarantee
- Current/Future research

Examples of concurrency

Concurrency

Concurrent:

> "Running together in space, as parallel lines; going on side by side, as proceedings; occurring together, as events or circumstances; existing or arising together; conjoint"
>
> - Oxford English Dictionary

Motivation

- Concurrency is hard:

  > "If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug."
  >
  > Java Sun Tutorial "Threads and Swing"

- Multi-core means concurrency everywhere!

Testing is hard

"Testing concurrent software is hard. Even simple tests require invoking methods from multiple threads and worrying about issues such as timeouts and deadlock. Unlike in sequential programs, many failures are rare, probabilistic events and numerous factors can mask potential errors."

JavaOne Technical session

Verification to the rescue?

Types of concurrency

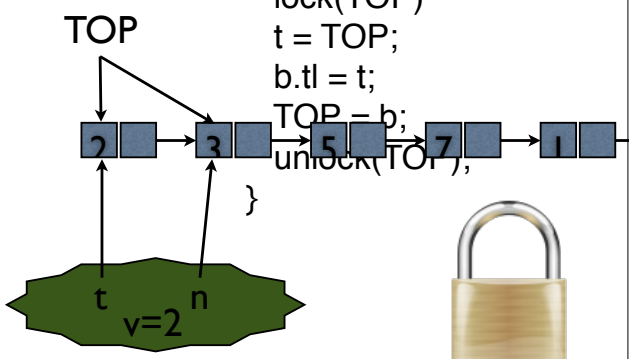
- Disjoint concurrency
- Coarse-grain concurrency
- Fine-grained concurrency
- Non-blocking concurrency

Blocking stack

(Coarse Grain)

```
pop() {
   Node t,n; int v;
   lock(TOP)
   t = TOP;
   if (t == null)
     return -1;
   n = t.tl;
   TOP = n;
   unlock(TOP)
   v = t.value;
   delete(t);
   return v;
}
```

```
push (v) {
   Node t,n;
   b = new Node(v);
   lock(TOP)
   t = TOP;
   b.tl = t;
   TOP = b;
   unlock(TOP);
}
```



Implementing Locks

Compare and Swap:

```
CAS(l, t, n)
means
if ( *l == t) {
   *l = n; return true
}
```

Problems with locks

- Pre-emption
- Bad in high contention
- Priority inversion

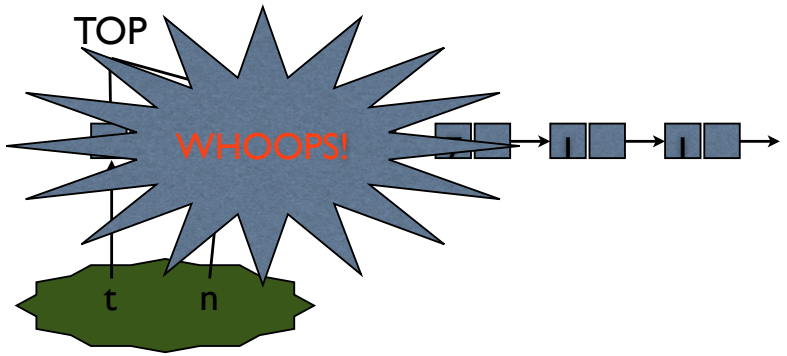
Try without locks.

Nonblocking stack

```
pop() {
  Node t,n; int v;
  while(true) {
    t = TOP;
    if (t == null) return -1;
    n = t.tl;
    if (CAS(&TOP,t,n)) {
        v = t.value;
        delete(t); return v;
    }
  }
}
```

```
push (v) {
  Node t,n;
  b = new Node(v);
  while(true) {
    t = TOP;
    b.tl = t;
    if (CAS(&TOP, t, b))
        break;
  }
}
```

The "non-obvious" bug



Non-blocking difficult

Several fixes to this algorithm

- Timestamps
- Garbage collector
- DCAS
- Hazard pointers

Disjoint Concurrency

Programming language

C ::= … | C || C | …

Now we extend the language with parallel composition. That is the ability for two threads/processes to be interleaved.

C_1 ll C_2 ,s,h → C_1' ll C_2',s',h'
if either C_1,s,h → C_1',s',h' and C_2=C_2'
or C_2,s,h → C_2',s',h' and C_1=C_1'

Parallel Rule

$$\frac{\{ P_1 \}\ C_1\ \{ Q_1 \} \qquad \{ P_2 \}\ C_2\ \{ Q_2 \}}{\{ P_1 * P_2 \}\ C_1 \,||\, C_2\ \{ Q_1 * Q_2 \}}$$

Provided variables used in C_1 not modified by C_2

and variables used in C_2 not modified by C_1

The variable side condition prevents races on the stack variables.

The * prevents races on the heap.

Example

$$\frac{\{ x \mapsto \_ \} \qquad \{ y \mapsto \_ \}}{\{ x \mapsto \_ * y \mapsto \_ \}}$$

$$\frac{\{ x \mapsto 3 \} \qquad \{ y \mapsto 4 \}}{\{ x \mapsto 3 * y \mapsto 4 \}}$$

[x] := 3 || [y] := 4

The variable side condition is trivially satisfied as neither side modifies any variables.

The * is used to ensure that neither thread modifies the same heap.

Parallel Dispose tree

{ tree(x) }disposetree(x) { emp } ⊢
{ tree(x) }
if x != 0 then
i := [x+1];
j := [x];
disposetree(i) || disposetree(j) || dispose x
{ empty }

Example

{ tree(x) ∧ x != 0}
{ ∃i,j. tree(i) * tree(j) * x ↦ i,j }
i := [x+1];
{ ∃j. tree(i) * tree(j) * x ↦ i,j }
j := [x];
{ tree(i) * tree(j) * x ↦ i,j }
disposetree(i) || disposetree(j) || dispose x
{ empty }

Exercise: Prove that the rule
{ ∃x. P * E ↦ x } x := [E] { P * E ↦ x }
provided E does not mention x.
is sound.

Example

{ tree(i) * tree(j) * x ↦ i,j }

{ tree(i) } { tree(j) } { x ↦ i,j }

disposetree(i) || disposetree(j) || dispose x .

{ empty } { empty } { empty }

{ empty * empty * empty }

{ empty }

Can we verify these?

{ empty }

x := cons(3);

z := cons(3);

[x]:=4 || [z]:=5;

{x⟼4 * z⟼5}

{ empty }

x := cons(3);

[x]:=4 || [x]:=5;

{x⟼_}

{ empty }

x:=4 || x:=5;

{ empty }

{ y = x+1 }

x:=4 || y:=y+1;

{ y = x+2 }

Merge sort

```
mergesort(x, n)
  if n >1 then
    local m in
    m := n/2;
    mergesort(x,m) || mergesort(x+m,n-m);
    merge(x,m,n-m)
```

Merge sort takes a pointer to an array an a length.
If the array has 1 or 0 elements then it is trivial to sort it. Otherwise, we must recurse. We divide the size of the array into two, and sort the two parts in parallel. Finally, we must merge the arrays.

Merge sort

{ array(x,n) }
  mergesort(x, n)
{ sorted_array(x,n) }

{ sorted_array(x,m) * sorted_array(x+m,n) }
  merge(x,m,n)
{ sorted_array(x,m+n) }

We can define array as
$\text{array}(x,n) \Leftrightarrow \circledast_{0 \le i < n}.\ x+i \mapsto \_$

where
$\circledast_{0 \le i < 0}.P(i) \Leftrightarrow \text{empty}$

and
$\circledast_{0 \le i < n+1}.P(i) \Leftrightarrow (\circledast_{0 \le i < n}.P(i)) * P(n)$

That is,
$\circledast_{0 \le i < n}.P(i) \Leftrightarrow P(0) * \ldots * P(n-1)$

We define a
$\text{data\_array}(x,n,f) \Leftrightarrow \circledast_{0 \le i < n}.\ x+i \mapsto f(i)$

Here f is used as a function to represent the data contained in the array.
We can then define a sorted array as
$\text{sorted\_array}(x,n) \Leftrightarrow \exists f.\ \text{data\_array}(x,n,f) \wedge \text{sorted}(f)$

$\text{sorted}(f) \Leftrightarrow \forall i.\ f(i) \le f(i+1)$

Exercise

Given the previous specifications verify the body of mergesort.

The specification does not deal with the resulting array being a permutation, how could you extend the specification?

Concurrent Separation Logic

Multiple access

How do we verify a program where several threads want access to the same memory? e.g.

[x] := 43 || [x] := 47

Programming language

C ::= … | resource r in C | with r when B in C | …

We add two connectives to deal with communication
resource r in C
Allocates a lock, r, that can be used inside the command C.
The second
with r when B in C
acquires the lock for the execution of C provided the condition B holds on entry, otherwise it blocks until that condition holds before acquiring the lock. This is also called a critical region.

Resource Rule

$$\frac{\Delta, r : I \vdash \{ P \}\ C\ \{ Q \}}{\Delta \vdash \{ P * I \}\ \text{resource } r \text{ in } C\ \{ Q * I \}}$$

We use a context to specify the resource invariant associate to each lock.

$\Delta ::= r : P \mid \Delta, \Delta$

We can treat this as a map from resource/lock name r, to its invariant P.

This rule can be read as removing some local state and making it shared by the lock r.

Lock Rule

$$\frac{\Delta \vdash \{ (P * I) \wedge B \}\ C\ \{ Q * I \}}{\Delta, r : I \vdash \{ P \}\ \text{with } r \text{ when } B \text{ in } C\ \{ Q \}}$$

In this rule, we acquire the lock r, and get the resource's invariant, I, added to our pre-condition. On exit from the lock, we must re-establish the resource's invariant, hence the "* I" in the post-condition.

This is similar to the while rule, where we must reestablish the loop invariant after executing the body. Here it is a little different, reestablishing the resource invariant it required, so the next thread to acquire the lock will correctly be able to assume the resource invariant holds.

Question: How do we know the resource invariant will not be modified when the lock isn't acquired?

Caveat: side-conditions

There are subtle variable side-conditions used to allow locks to refer to global variables.

Each variable is either associate to

- a single thread; or
- a single lock.

It can then only be modified and used in assertions by the thread, or while the thread holds the associate lock.

Alternatively, we could encode all variable access into the heap, but this leads to complications in the presentation.

Another alternative is presented in
Variables as resource for Hoare Logic
Parkinson, Bornat, and Calcagno, LICS 2006
where a logic of variables is developed in a similar way to separation logic deals with the heap.

Binary Semaphore

We can encode a semaphore as a critical region

$P(s)$ = with r_s when $s=1$ do $s := 0$
$V(s)$ = with r_s when $s=0$ do $s := 1$

Resource invariant

$(s=0 \wedge \text{empty}) \quad \vee \quad (s=1 \wedge Q)$

Initially,

$s=0$

The variable s is associate with the critical region/lock r_s.

Q is the state protected by the semaphore.

We can specify
$\{ \text{emp} \}\ P(s)\ \{ Q \}$
and
$\{ Q \}\ V(s)\ \{ \text{emp} \}$

Exercise: Prove the body meets these specifications.

Example

{ emp }	‖	{ emp }
P(s)		P(s)
[x] := 43		[x] := 47
V(s)		V(s)
{ emp }		{ emp }

Example

{ emp }
P(s)
{ x ↦ _ }
[x] := 43
{ x ↦ _ }
V(s)
{ emp }

$$\frac{\{\ \text{emp} * (l_s \wedge s{=}1)\}\ s := 0\ \{\ x \mapsto \_ * l_s\ \}}{\{\text{emp}\}\ \text{with}\ r_s\ \text{when}\ s{=}1\ \text{do}\ s{:=}0\ \{\ x \mapsto \_\ \}}$$

$$\frac{\{\ x \mapsto \_ * (l_s \wedge s{=}0)\}\ s := 0\ \{\ \text{emp} * l_s\ \}}{\{x \mapsto \_\}\ \text{with}\ r_s\ \text{when}\ s{=}0\ \text{do}\ s{:=}1\ \{\ \text{emp}\ \}}$$

Let Q be x ↦ _ in the resource invariant of the semaphore.

One place buffer

full := false

```
with buff when full do
    full := false
     y := c
dispose y
```

‖

```
x := new
with buff when ¬full do
    full := true;
    c := x;
```

One place buffer

full := false

{ (full ∧ c ↦ _) ∨ (¬full ∧ empty) }

Resource Invariant

```
with buff when full do
    full := false
    y := c
dispose y
```

‖

```
x := new
with buff when ¬full do
    full := true;
    c := x;
```

The variable full is associated with the lock, buff.

One place buffer

{ (full ∧ c ↦ _) ∨ (¬full ∧ empty) }

```
with buff when full do          ||  x := new
  { full ∧ c ↦ _ }              ||  { x ↦ _ }
  full := false                 ||  with buff when ¬full do
   y := c                       ||    { (¬full∧ empty)
  { (¬full∧ empty)              ||      * x ↦ _ }
    * y ↦ _ }                   ||    full := true;
{ y ↦ _ }                       ||    c := x;
dispose y                       ||    { full ∧ c ↦ _ }
```

Upon entry to the critical region, we get the resource invariant added to our current precondition. When we leave the region we must reestablish the resource invariant.

Ownership is in the eye of the assertor

full := false

```
with buff when full do          ||  x := new
  full := false                 ||  with buff when ¬full do
  y := c                        ||    full := true;
                                ||    c := x;
                                ||  dispose y
```

Can we verify the following?

Caveat: precise invariants

The resource invariant associate with a particular lock must be precise.

A formula, P, is precise, iff, for any heap, there is at most one subheap satisfying the formula

$\forall h_1, h_2, h.\quad h_1 \preceq h \wedge h_2 \preceq h \wedge h_1 \models P \wedge h_2 \models P \Rightarrow h_1 = h_2$

We can define an order on heaps as
$h_1 \preceq h \Leftrightarrow \exists h'.\ h_1 * h = h'$

Rule of conjunction

$$\frac{\{ P_1 \}\ C\ \{ Q_1 \} \qquad \{ P_2 \}\ C\ \{ Q_2 \}}{\{ P_1 \wedge P_2 \}\ C\ \{ Q_1 \wedge Q_2 \}}$$

Subtle soundness

Without rule of conjunction logic is sounds with arbitrary resource invariants.

With precise invariants and the rule of conjunction the logic is sound.

Reynolds' counter example

We can prover the following holds with the resource invariant: r: true.

$$\frac{\{true\}\ skip\ \{true\} \qquad \{(emp \vee x \mapsto \_) * true\}\ skip\ \{emp * true\}}{\{\ emp \vee x \mapsto \_\ \}\ \text{with r when true do skip}\ \{\ emp\ \}}$$

Reynolds counter example

From the previous proof we can derive both

{ emp * x ↦ _ } with… { x ↦ _ }

{ emp * x ↦ _ } with… { emp }

which with the rule of conjunction leads to a contradiction.

What about racy programs?

Do we want to forbid all races?

Does concurrent separation logic forbid all races?

Owicki/Gries method

Example

If we assume assignment is atomic, then is the following true:

{ x = 0 }
x := x + 1 || x := x + 2
{ x = 3 }

For this lecture we will ignore the heap.

We will focus on how to remove the draconian restriction on variables from the previous section. As we had the heap these restrictions didn't seem so bad. But without the heap they are very restrictive.

This lecture is based very heavily on Francesco Zappa-Nardelli's presentation.

Parallel rule (attempt)

$$\frac{\{ P_1 \}\ C_1\ \{ Q_1 \} \qquad \{ P_2 \}\ C_2\ \{ Q_2 \}}{\{ P_1 \wedge P_2 \}\ C_1 \parallel C_2\ \{ Q_1 \wedge Q_2 \}}$$

This rule is unsound. Consider

{ y=1 } x := 0 { y=1 }
{ true } y := 2 { true }

both hold in Hoare logic, but
{ y=1 ∧ true } x :=0 ll y := 2 { y=1 ∧ true }
certainly does not hold.

Parallel rule (attempt 2)

$$\frac{\{ P_1 \}\ C_1\ \{ Q_1 \} \qquad \{ P_2 \}\ C_2\ \{ Q_2 \}}{\{ P_1 \wedge P_2 \}\ C_1 \parallel C_2\ \{ Q_1 \wedge Q_2 \}}$$

Provided $FV(P_1,Q_1)$ not modified by C_1

and $FV(P_2,Q_2)$ not modified by C_2

This rule is unsound. Consider

{ y=1 } x := y ; z := x { z=1 }
{ true } x := 2 { true }

both hold in Hoare logic, but
{ y=1 ∧ true } (x := y ; y := x) ll x := 2 { z=1 ∧ true }
certainly does not hold.

The issue is that the intermediate assertion in the proof
{ y=1 }
x := y ;
{ x=1 } ← This is affected by the other thread.
z := x
{ z=1 }

The assignment interferes with the assertion.

Interference

Do the following commands affect the assertions

	x:=x+1	y:=y+1	x:=x+2	x:=y
x>100				
y=4				
even(x)				
x<30				
x=y				
even(x)∧even(y)				

Interference

C does not interfere with P:

- mod(C) is disjoint from FV(P) ←too restrictive
- { P } C { P }

The second says the command does not affect the validity of the assertion

Exercise: prove the questions on the previous slide with both definitions.

Interference freedom

We define the critical formula of a proof outline Δ proving {P}C{Q}, as Q and every pre-condition of a command.

Given two proof trees Δ_1 Δ_2, they are interference free if for every critical formula in one, R, and every triple in the other {P}C{Q}, then the triple preserves the formula, $\{P \wedge R\}\ C\ \{ R \}$.

Example

$\{ x = 0 \}$

$\{ x = 0 \vee x=2 \}$ $\{ x = 0 \vee x = 1\}$

x := x +1 || x := x + 2

$\{ x = 1 \vee x = 3\}$ $\{ x = 2 \vee x = 3\}$

$\{ x = 3 \}$

Interference freedom requires

$\{ x = 0 \}$
$\{ x = 0 \vee x=2 \}$ $\{ x = 0 \vee x = 1 \}$
$x := x + 1$ || $x := x + 2$
$\{ x = 1 \vee x = 3 \}$ $\{ x = 2 \vee x = 3 \}$
$\{ x = 3 \}$

$\{(x=0 \vee x=2) \wedge (x=0 \vee x=1)\}\ x := x + 1\ \{ x=0 \vee x=1 \}$

$\{(x=0 \vee x=2) \wedge (x=2 \vee x=3)\}\ x := x + 1\ \{ x=2 \vee x=3 \}$

$\{(x=0 \vee x=1) \wedge (x=0 \vee x=2)\}\ x := x + 2\ \{ x=0 \vee x=2 \}$

$\{(x=0 \vee x=1) \wedge (x=1 \vee x=3)\}\ x := x + 2\ \{ x=1 \vee x=3 \}$

Example: Bank

$\{ dep > 0 \}$
if credit > 1000 then flag := 1 else flag :=0
||
credit := credit + dep
$\{ flag = 1 \Rightarrow credit > 1000 \}$

Example: Bank

if credit > 1000 then
{credit > 1000}
{1=1 ⇒ credit>1000}
flag := 1
{flag=1 ⇒ credit>1000}
else
{credit<1000}
{1=0 ⇒ credit>1000}
flag :=0
{flag=1 ⇒ credit>1000}
{flag=1 ⇒ credit>1000}

{ dep > 0 }
credit := credit + dep
{ dep > 0 }

We have four unique critical assertions, and three commands that update the state.

Exercise: Prove each of the critical assertions is preserved by the commands.

Example: Bank

What goes wrong if we strengthen the specification?

{ dep > 0 }
if credit > 1000 then flag := 1 else flag :=0
||
credit := credit + dep
{ flag = 1 ⇔ credit > 1000 }

Exercise: Attempt this proof, and illustrate which of the conditions does not hold.

Completeness

Can you prove the following:

{ x=0 }
x:=x+1 || x := x+1
{ x=2 }

?

Attempt

{ x=0 ∨ x=1 ∨ x=2 ∨ …}
x:=x+1
||
{ x=0 ∨ x=1 ∨ x=2 ∨ …}
x:=x+1

Auxiliary Variables

Sometimes we need to instrument program to account for interference more precisely

$\{ x=0 \wedge b_1=0 \wedge b_2 = 0\}$

$\langle x:=x+1; b_1:=1\rangle \;||\; \langle x := x+1; b_2:=1\rangle$

$\{ x=2 \}$

Angle brackets mean the program executes all the operation in one indivisible unit of time. That is, intermediate states are not observable. It is atomic.

Attempt II

$\{ x=b_1=b_2=0 \}$

$\{ x=b_2 \wedge b_1=0 \}$		$\{ x=b_1 \wedge b_2=0\}$
$\langle x:=x+1; b_1:=1\rangle$	$\|\|$	$\langle x:=x+1; b_2:=1\rangle$
$\{ x=b_2+1 \wedge b_1=1\}$		$\{ x=b_1+1 \wedge b_2=1\}$

$\{ x=b_2+1=b_1+1 \wedge b_1=1 \wedge b_2 =1 \}$

$\{ x=2 \}$

Auxiliary Variables

A set of variables, A, is consider auxiliary, if

- the only expressions they appear in, are on the right of an assignment where the target of the assignment is in the set of auxiliary.

We define erase(C,A) as replacing all assignment to an auxiliary by skip.

That is they cannot appear in the guards of loops or if-then-else commands, and their value cannot be assigned to a normal variables.

Alternatively, it can be seen as erase(C,A) no longer mentions A.

Auxiliary variable elimination

$$\frac{\{ P \}\ C\ \{ Q \}}{\{ P \}\ C'\ \{ Q \}}$$

where C' = erase(C,A)

and A not in FV(P,Q)

This rule allows us to introduce auxiliary variables in the justification of a program. As we did for the double increment program.

Conclusions

If we have n commands in a thread, and m critical assertions, and t threads how many interference checks would we have to perform?

Is there a more scalable/compositional way?

Rely-guarantee method

A good introduction to rely-guarantee can be found in Vafeiadis's award winning dissertation:

Modular fine-grained concurrency verification
http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-726.html

Interference

In Owicki/Gries method each pre-condition needed checking against each command in another thread.

Can we abstract the interference?

Rely-guarantee

Judgements are extended with a concurrent context

$R, G \vdash \{ P \}\ C\ \{ Q \}$

- the rely, R, is an abstraction of what the other threads can do; and
- the guarantee, R, is an abstraction of what this thread does.

Both the rely and guarantee are relations.

Example

We specify relations by describing the current and the previous state: for example,

$old(x=0) \wedge x=1$

$\forall X.\ old(x=X) \wedge x=X+1$

$(old(x=0) \wedge x=1) \ \vee (old(x=2) \wedge x=3)$

Stability

We define an assertion as stable with respect to a relation as

P stable under R

$\Leftrightarrow \quad \sigma \models P \wedge (\sigma,\sigma') \models R \quad \Rightarrow \quad \sigma' \models P$

Stability

Do the following relations preserve the assertions

	x=old(x)+1	x=old(x)+2	id(x) ∧ y=old(y)+1	id(x) ∧ y=old(y)+2
x>100				
y=4				
even(x)				
x<30				
x=y				
even(x)∧even(y)				

Parallel Rule

$G_1 \subseteq R_2$

$G_2 \subseteq R_1$

$R_1, G_1 \vdash \{ P_1 \} C_1 \{ Q_1 \}$

$\underline{R_2, G_2 \vdash \{ P_2 \} C_2 \{ Q_2 \}}$.

$R_1 \cap R_2, G_1 \cup G_2 \vdash \{ P_1 \wedge P_2 \} C_1 \,||\, C_2 \{Q_1 \wedge Q_2 \}$

Parallel Rule

$$\frac{G_1 \Rightarrow R_2 \quad G_2 \Rightarrow R_1 \quad R_1, G_1 \vdash \{ P_1 \}\ C_1\ \{ Q_1 \} \quad R_2, G_2 \vdash \{ P_2 \}\ C_2\ \{ Q_2 \}}{R_1 \wedge R_2, G_1 \vee G_2 \vdash \{ P_1 \wedge P_2 \}\ C_1 \parallel C_2\ \{Q_1 \wedge Q_2 \}}$$

Assignment

$$\frac{P \text{ stable under } R \quad Q \text{ stable under } R \quad P \wedge x=\text{old}(E) \Rightarrow G \quad P \Rightarrow Q\ [x:=E]}{R, G \vdash \{ P \}\ x := E\ \{ Q \}}$$

If we consider a logic over relations, we can view it with the logical connectives.

Skip

P stable under R .

R, G ⊢ { P } skip { P }

Surprisingly, we have to modify this rule slightly to maintain soundness of the system.

Example

Let us return to our simple example:

x := x + 1 || x := x + 2

We can verify this using the following relations

R_1= (old(x)=0 ∧ x=2) ∨ (old(x)=1 ∧ x=3) = G_2

R_2= (old(x)=0 ∧ x=1) ∨ (old(x)=2 ∧ x=3) = G_1

First Thread

To show

$R_1, G_1 \vdash \{ x=0 \vee x=2 \}\ x := x + 1\ \{ x=1 \vee x=3 \}$

We need to prove

- $x=0 \vee x=2$ stable under R_1; and
- $x=1 \vee x=3$ stable under R_1; and
- $old(x=0 \vee x=2) \wedge x=old(x+1) \Rightarrow G_1$; and
- $(x=0 \vee x=2)\ [x:=x+1] \Rightarrow (x=1 \vee x=3)$

Comparison

How does this compare to Owicki/Gries?

Exercise: Prove the second thread meets the specification.

Current/Future Research