Program Equivalence From Trace Equivalence

Tim Wood[1] Sophia Drossopoulou[1]

[1]Imperial College London

Context

- Program maintenance is a common and important activity
- Programmers need to change or extend behaviour
- Intent is often to preserve most of the behaviour
- When changing one behaviour, easy and common to unintentionally damage some other behaviour

Context

- Program maintenance is a common and important activity
- Programmers need to change or extend behaviour
- Intent is often to preserve most of the behaviour
- When changing one behaviour, easy and common to unintentionally damage some other behaviour
- Programmers often don't want to
  - specify
  - write tests for
  - read or understand

  the whole program before they can change it

Question

- What does it mean "preserve most of the behaviour"?
- Can we automatically check that most of the behaviour is preserved?

Answer

We propose a precise criterion to judge preservation of most of the behaviour

- Programmer tells us which objects should be affected by the modification
- If any other objects are affected, then the behaviour was not preserved

Answer

We propose a sufficient condition for the criteria that is useful for our tool

- Condition depends only on the trace of method calls between the affected and unaffected objects
- Condition doesn't require the unaffected objects to be precisely tracked

We are developing a tool which uses symbolic execution and approximation (not described here)

Contributions

- Formal criterion for "preserve most of the behaviour"
- Useful sufficient condition for automated checking
- Automated proof of sufficiency of condition in Dafny Verifier

Example

```
1  StudentDb db = new StudentDb();
2  Logger logger = new Logger();
3  Prizes prizes = new Prizes();
4  void main() {
5   List<Student> students = db.orderedByGrade();
6   logger.considered(students);
7   prizes.awardTo(students); }
8
9  class Prizes {
10  void awardTo(List<Student> students) {
11   award(students.get(0));
12   award(students.get(1));
13   award(students.get(2));}
14  void award(final Student student) { /* ... */ }}
15
16 class Logger {
17  void considered(List<Student> students) {
18
19   for (Student s : students) {
20    println("considered: " + s.name()); }}}
```

Example

```
1  StudentDb db = new StudentDb();
2  Logger logger = new Logger();
3  Prizes prizes = new Prizes();
4  void main() {
5   List<Student> students = db.orderedByGrade();
6   logger.considered(students);
7   prizes.awardTo(students); }
8
9  class Prizes {
10  void awardTo(List<Student> students) {
11   award(students.get(0));
12   award(students.get(1));
13   award(students.get(2));}
14  void award(final Student student) { /* ... */ }}
15
16 class Logger {
17  void considered(List<Student> students) {
18   sort(students, compareStudentsByName());
19   for (Student s : students) {
20    println("considered: " + s.name()); }}}
```

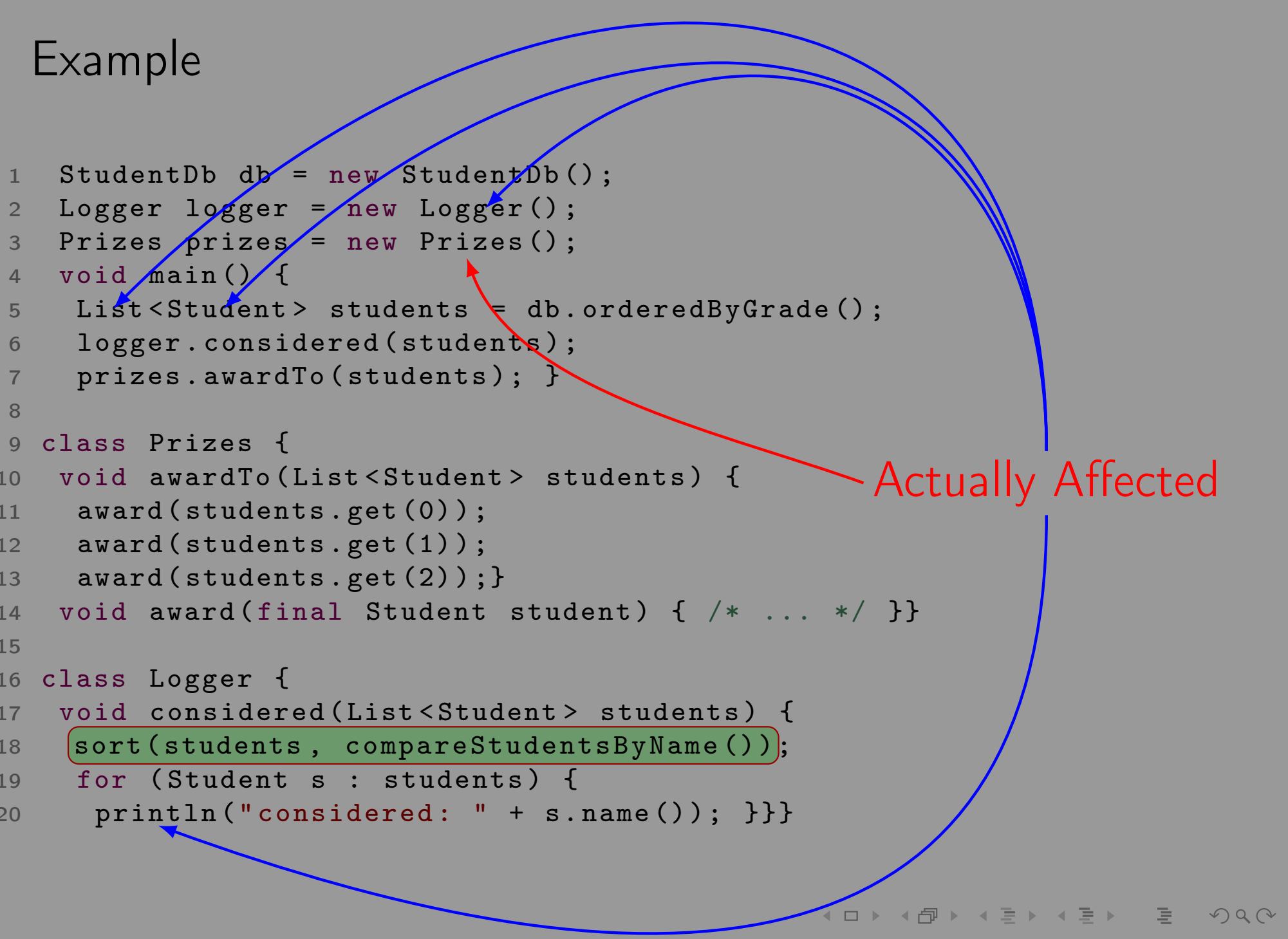
Example

```
1  StudentDb db = new StudentDb();
2  Logger logger = new Logger();
3  Prizes prizes = new Prizes();
4  void main() {
5   List<Student> students = db.orderedByGrade();
6   logger.considered(students);
7   prizes.awardTo(students); }
8
9  class Prizes {
10  void awardTo(List<Student> students) {
11   award(students.get(0));
12   award(students.get(1));
13   award(students.get(2));}
14  void award(final Student student) { /* ... */ }}
15
16 class Logger {
17  void considered(List<Student> students) {
18   sort(students, compareStudentsByName());
19   for (Student s : students) {
20    println("considered: " + s.name()); }}}
```

Allegedly Affected

Example

```
1  StudentDb db = new StudentDb();
2  Logger logger = new Logger();
3  Prizes prizes = new Prizes();
4  void main() {
5   List<Student> students = db.orderedByGrade();
6   logger.considered(students);
7   prizes.awardTo(students); }
8
9  class Prizes {
10  void awardTo(List<Student> students) {
11   award(students.get(0));
12   award(students.get(1));
13   award(students.get(2));}
14  void award(final Student student) { /* ... */ }}
15
16 class Logger {
17  void considered(List<Student> students) {
18   sort(students, compareStudentsByName());
19   for (Student s : students) {
20    println("considered: " + s.name()); }}}
```



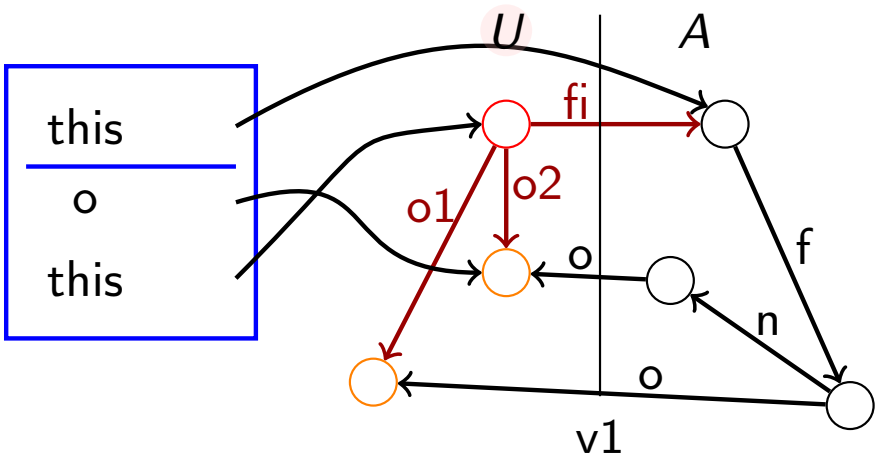
Equivalence Criteria

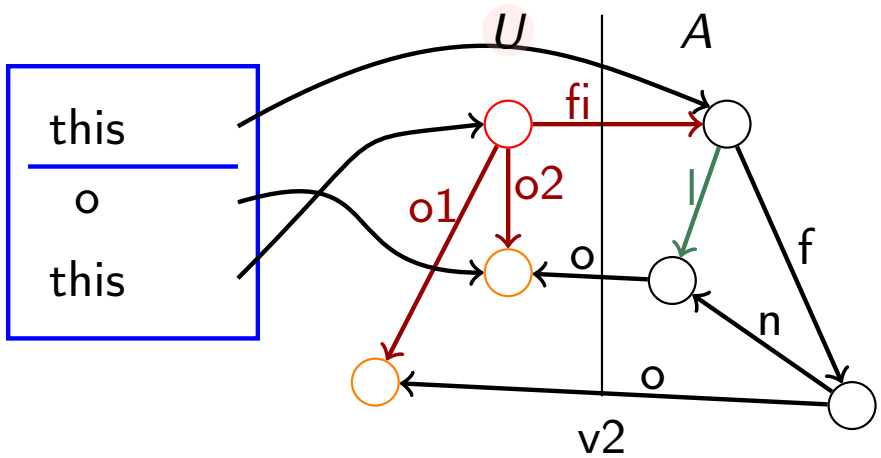
- Equivalence criteria are expressed only in terms of the objects involved in the behaviour of interest to the programmer
- Programmer can be ignorant of the rest of the objects and their behaviour
  - Programmer doesn't have to read or understand those parts, our criteria takes care that they are not affected

Equivalence Criteria

Stack and Heap have the same shape when only the unaffected objects are considered.

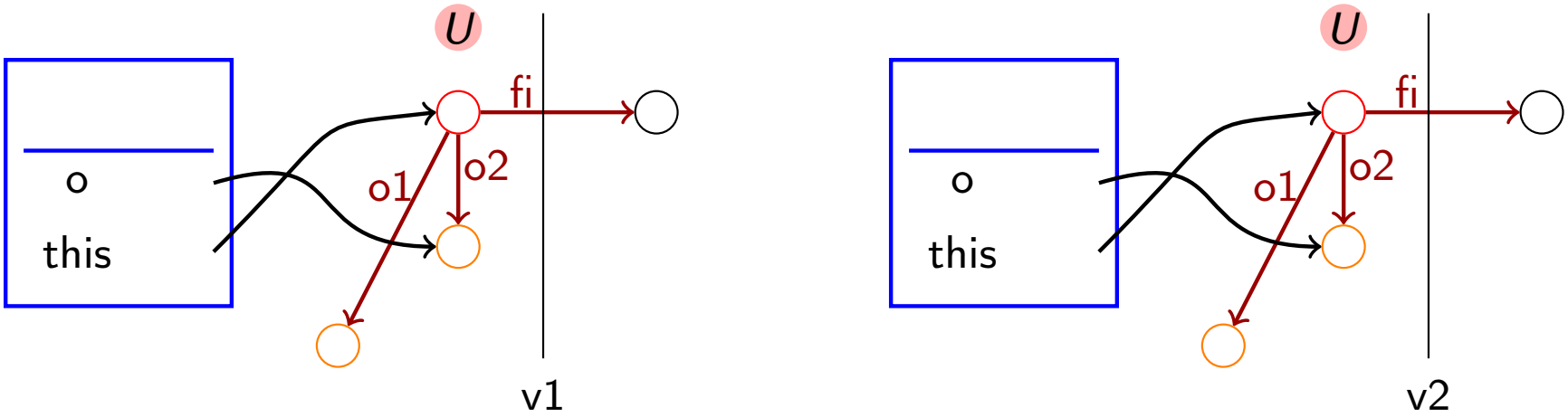
State Equivalence - Example





Stack and heap at a point in execution of a different example program.

State Equivalence - Example



Unaffected objects and stack frames only

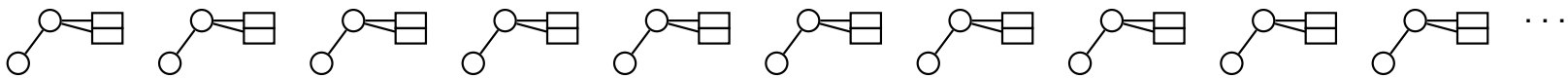
Sufficient Condition

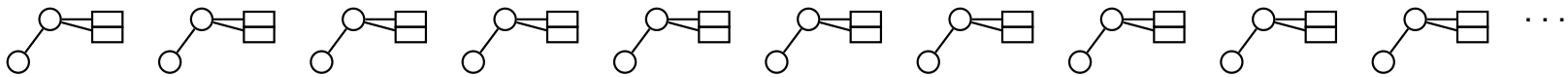
- Objective: Avoid comparing the whole heap at each execution step
- Solution: Compare only methods calls/returns between affected and unaffected objects

Also allows us to over-approximate some or all of the unaffected objects

Sufficient Condition - Illustration

We want to establish that the unaffected objects correspond at each execution step

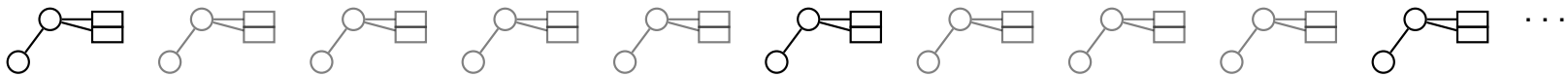
V1 

V2 

Sufficient Condition - Illustration

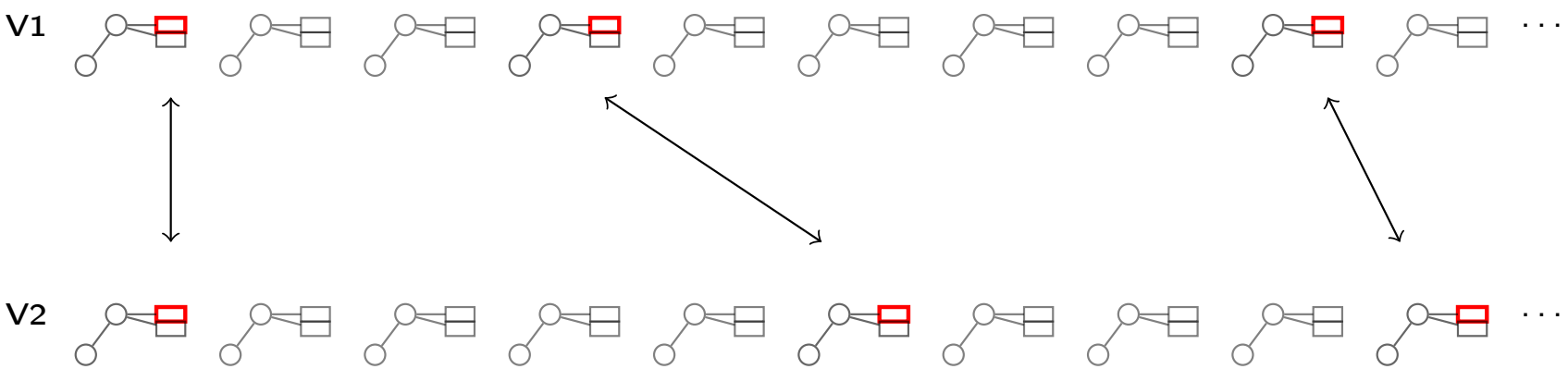
Only need to consider states which are method calls or returns between affected and unaffected objects

V1 

V2 

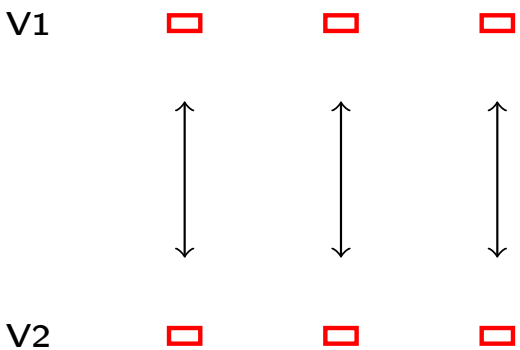
Sufficient Condition - Illustration

Only need to consider the topmost stack frame



Sufficient Condition - Illustration

Intuition: the method calls and returns (and method parameters), between the affected and unaffected objects are enough information to uniquely determine the unaffected objects



Trace Equivalence - Example

t_1 (from v1)

Method	**Params**	
Fifo	this: 3	
return	3	
add	this: 3	o: 1
return		
add	this: 3	o: 2
return		
remove	this: 3	
return	1	
remove	this: 3	
return	2	

Trace Equivalence - Example

t_1 (from v1)				t_2 (from v2)			
Method	**Params**			**Method**	**Params**		
Fifo	this: 3			Fifo	this: 10		
return	3			*return*	10		
add	this: 3	o:	1	add	this: 10	o:	8
return				*return*			
add	this: 3	o:	2	add	this: 10	o:	9
return				*return*			
remove	this: 3			remove	this: 10		
return	1			*return*	8		
remove	this: 3			remove	this: 10		
return	2			*return*	9		

$t_1 \cong t_2$ using mapping (1,8),(3,10),(2,9)

Trace Equivalence - Example

t_1 (from v1)

Method	**Params**		
Fifo	this: 3		
return	3		
add	this: 3	o:	1
return			
add	this: 3	o:	2
return			
remove	this: 3		
return	1		
remove	this: 3		
return	2		

t_3 (imaginary)

Method	**Params**		
Fifo	this: 7		
return	7		
add	this: 7	o:	4
return			
add	this: 7	o:	6
return			
remove	this: 7		
return	6		
remove	this: 7		
return	4		

$t_2 \not\cong t_3$

References

Many languages have features that leak information about the actual values of addresses, or order of allocations.

- Java/C♯/etc: `hashcode()`, python: `id()`, ruby: `object_id`, etc.
- A tool using our sufficient condition would additionally need to check any uses of such features

Proof of Sufficiency

Use *Microsoft Dafny* program verifier to prove that trace equivalence implies state equivalence

- How to represent the question in Dafny?
- How to represent and prove Theorems?
- Some useful techniques.

Operational Semantics

implement operational semantics as executable pure function

```
1 datatype Expr = Nop | Call(meth:Method) | ... | Exception
2
3 function method eval(s:SC) : Result {
4   if(!hasSf(s)) then Stuck else
5   ...
6   match expr
7    case Exception => Stuck
8    case Nop => result(Tau, s1)
9    ...
```

Operational Semantics

```
1 datatype Expr = Nop | Call(meth:Method) | ... | Exception
2
3 function method eval(s:SC) : Result {
4   if(!hasSf(s)) then Stuck else
5   ...
6   match expr
7    case Exception => Stuck
8    case Nop => result(Tau, s1)
9    ...
```

instructions, algebraic data type

Operational Semantics

```
1 datatype Expr = Nop | Call(meth:Method) | ... | Exception
2
3 function method eval(s:SC) : Result {
4   if(!hasSf(s)) then Stuck else
5   ...
6   match expr
7    case Exception => Stuck
8    case Nop => result(Tau, s1)
9    ...
```

pure, non-ghost

Operational Semantics

```
1 datatype Expr = Nop | Call(meth:Method) | ... | Exception
2
3 function method eval(s:SC) : Result {
4   if(!hasSf(s)) then Stuck else
5   ...
6   match expr
7    case Exception => Stuck
8    case Nop => result(Tau, s1)
9    ...
```

terminate if out of stack

Operational Semantics

```
1 datatype Expr = Nop | Call(meth:Method) | ... | Exception
2
3 function method eval(s:SC) : Result {
4   if(!hasSf(s)) then Stuck else
5   ...
6   match expr
7    case Exception => Stuck
8    case Nop => result(Tau, s1)
9    ...
```

case split on instruction, and so on...

Theorems/Lemmas

proof by induction over a sequence

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  // Imperative ghost code that proves lemma
8  ...
9  if(ls1 = []) { /* induction base case */ }
10 else {
11 ...
12 /* call induction hypothesis (recursive call) */
13 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
14 ...
15 }}
```

Theorems/Lemmas

ghost procedure - imperative code proves declarative lemma

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  // Imperative ghost code that proves lemma
8  ...
9  if(ls1 = []) { /* induction base case */ }
10 else {
11 ...
12 /* call induction hypothesis (recursive call) */
13 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
14 ...
15 }}
```

Theorems/Lemmas

parameters universally quantified
returns existentially quantified

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  // Imperative ghost code that proves lemma
8  ...
9  if(ls1 = []) { /* induction base case */ }
10 else {
11 ...
12 /* call induction hypothesis (recursive call) */
13 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
14 ...
15 }}
```

Theorems/Lemmas

prove precondition implies post condition

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  // Imperative ghost code that proves lemma
8  ...
9  if(ls1 = []) { /* induction base case */ }
10 else {
11 ...
12 /* call induction hypothesis (recursive call) */
13 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
14 ...
15 }}
```

Theorems/Lemmas

Dafny proves base case automatically

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  // Imperative ghost code that proves lemma
8  ...
9  if(ls1 = []) { /* induction base case */ }
10 else {
11 ...
12 /* call induction hypothesis (recursive call) */
13 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
14 ...
15 }}
```

Theorems/Lemmas

applying induction hypothesis is a recursive procedure call

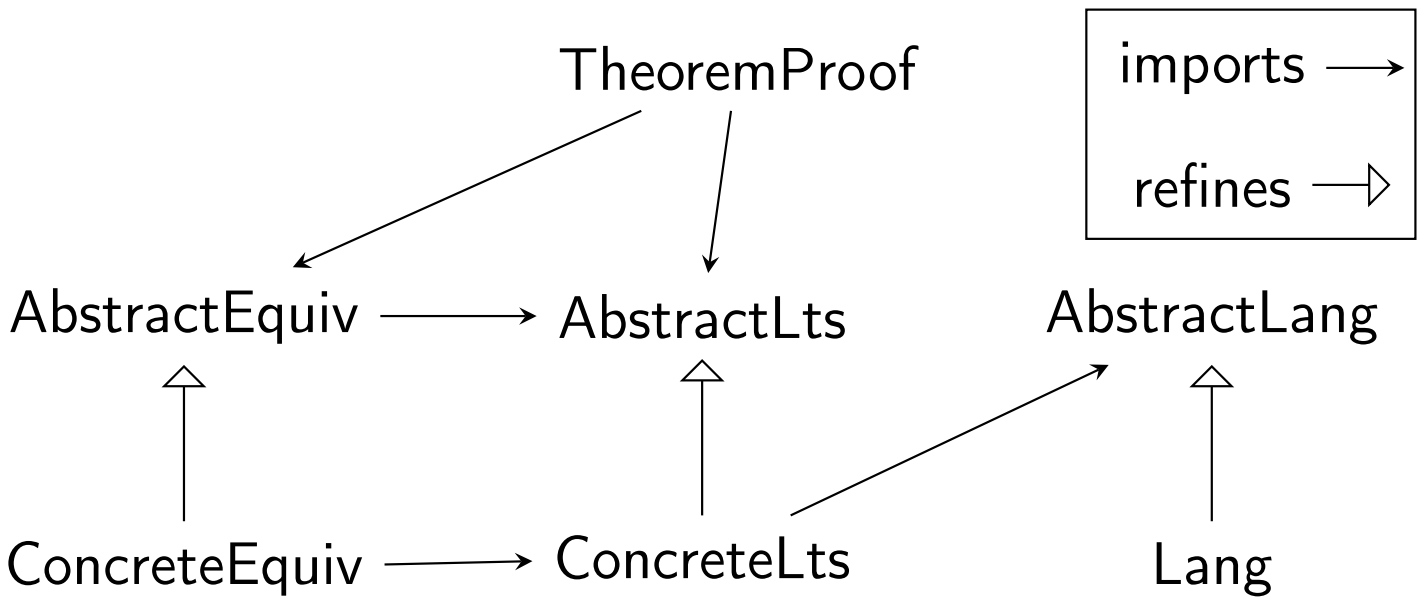
```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  // Imperative ghost code that proves lemma
8  ...
9  if(ls1 = []) { /* induction base case */ }
10 else {
11 ...
12 /* call induction hypothesis (recursive call) */
13 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
14 ...
15 }}
```

Performance Issues

- Proof is fairly big ($\approx$ 6000 lines)
- Our operational semantics function and equivalence predicates are large
- Dafny automatically exposes function definitions at the call site
- Seems to cause the proof search to get lost

Refinement

- Dafny supports modules and module refinement
- Use abstract evaluation and equivalence functions in most of the proof
- Refine the abstractions to show the proof applies to the concrete evaluation and equivalence



Refinement

- Resolved performance issues
  - along with some other techniques - see paper
- Led us to prove more general result
  - proof applies to any equivalences and language semantics provided you can construct the refinements

End

End

Extras

Code for Example 2

State Equivalence - Another Example

```
1  class FiFo {
2   Node f;
3   void add(final Object o) {
4    if(f == null)
5     { f=new Node(o); }
6    else { f.add(o); }}
7   Object remove() {
8    Object r=f.value();
         f=f.next(); return r;}}
9
10 class Node {
11  Object o; Node n;
12  Node(Object o) { this.o=o; }
13  Node next() { return n; }
14  Object value() { return o; }
15  void add(final Object o) {
16   if(n == null)
17    { n=new Node(o); }
18   else { n.add(o); }}}
```

```
1  class FiFo {
2   Node f, l;
3   void add(final Object o) {
4    if(f == null)
5     { l=f=new Node(o); }
6    else { l=l.add(o); }}
7   Object remove() {
8    Object r=f.value();
         f=f.next(); return r;}}
9
10 class Node {
11  Object o; Node n;
12  Node(Object o) { this.o=o; }
13  Node next() { return n; }
14  Object value() { return o; }
15  Node add(Object o) {
16  return n=new Node(o); }}
17
18
```

State Equivalence - Example

```
1 class Test {
2  FiFo fi = new FiFo();
3  Object o1 = new Object(),
4         o2 = new Object();
5  void test() {
6    fi.add(o1); fi.add(o2);
7    o1=fi.remove(); o2=fi.remove();}}
```

Extras

Operational Semantics More Detail

Operational Semantics

```
1  datatype Expr = Nop | Call(meth:Method) | ... | Exception
2  datatype Result = result(l:LC, s:SC) | Stuck
3  function method eval(s:SC) : Result {
4    if(!hasSf(s)) then Stuck else
5    ...
6    var expr := topCont(s).o[0];
7    var s1 := advance(s);
8    ...
9    match expr
10    case Exception => Stuck
11    case Nop => result(Tau, s1)
12    ...
13    case Call(meth) =>
14     if |topOperands(s).o| < 2 then Stuck
15     else
16       var receiver := topOperands(s).o[0];
17       var param := topOperands(s).o[1];
18       var klass := KlassOf(s1, receiver).o;
19       var code := codeForMeth(version(s1), klass, meth);
20       var sr := pushNewFrameForMethod
21                   (s1, receiver, IsMethod(klass, meth), param, code);
22    ...
```

Operational Semantics

```
1  datatype Expr = Nop | Call(meth:Method) | ... | Exception
2  datatype Result = result(l:LC, s:SC) | Stuck
3  function method eval(s:SC) : Result {
4    if(!hasSf(s)) then Stuck else
5    ...
6    var expr := topCont(s).o[0];
7    var s1 := advance(s);
8    ...
9    match expr
10    case Exception => Stuck
11    case Nop => result(Tau, s1)
12    ...
13    case Call(meth) =>
14     if |topOperands(s).o| < 2 then Stuck
15     else
16       var receiver := topOperands(s).o[0];
17       var param := topOperands(s).o[1];
18       var klass := KlassOf(s1, receiver).o;
19       var code := codeForMeth(version(s1), klass, meth);
20       var sr := pushNewFrameForMethod
21                 (s1, receiver, IsMethod(klass, meth), param, code);
22    ...
```

Operational Semantics

```
1  datatype Expr = Nop | Call(meth:Method) | ... | Exception
2  datatype Result = result(l:LC, s:SC) | Stuck
3  function method eval(s:SC) : Result {
4    if(!hasSf(s)) then Stuck else
5    ...
6    var expr := topCont(s).o[0];
7    var s1 := advance(s);
8    ...
9    match expr
10    case Exception => Stuck
11    case Nop => result(Tau, s1)
12    ...
13    case Call(meth) =>
14     if |topOperands(s).o| < 2 then Stuck
15     else
16       var receiver := topOperands(s).o[0];
17       var param := topOperands(s).o[1];
18       var klass := KlassOf(s1, receiver).o;
19       var code := codeForMeth(version(s1), klass, meth);
20       var sr := pushNewFrameForMethod
21                 (s1, receiver, IsMethod(klass, meth), param, code);
22    ...
```

pure, non-ghost

Operational Semantics

```
1  datatype Expr = Nop | Call(meth:Method) | ... | Exception
2  datatype Result = result(l:LC, s:SC) | Stuck
3  function method eval(s:SC) : Result {
4    if(!hasSf(s)) then Stuck else
5    ...
6    var expr := topCont(s).o[0];
7    var s1 := advance(s);
8    ...
9    match expr
10    case Exception => Stuck
11    case Nop => result(Tau, s1)
12    ...
13    case Call(meth) =>
14     if |topOperands(s).o| < 2 then Stuck
15     else
16       var receiver := topOperands(s).o[0];
17       var param := topOperands(s).o[1];
18       var klass := KlassOf(s1, receiver).o;
19       var code := codeForMeth(version(s1), klass, meth);
20       var sr := pushNewFrameForMethod
21                 (s1, receiver, IsMethod(klass, meth), param, code);
22    ...
```

terminate if out of stack

Operational Semantics

```
1  datatype Expr = Nop | Call(meth:Method) | ... | Exception
2  datatype Result = result(l:LC, s:SC) | Stuck
3  function method eval(s:SC) : Result {
4    if(!hasSf(s)) then Stuck else
5    ...
6    var expr := topCont(s).o[0];
7    var s1 := advance(s);
8    ...
9    match expr
10    case Exception => Stuck
11    case Nop => result(Tau, s1)
12    ...
13    case Call(meth) =>
14     if |topOperands(s).o| < 2 then Stuck
15     else
16       var receiver := topOperands(s).o[0];
17       var param := topOperands(s).o[1];
18       var klass := KlassOf(s1, receiver).o;
19       var code := codeForMeth(version(s1), klass, meth);
20       var sr := pushNewFrameForMethod
21                   (s1, receiver, IsMethod(klass, meth), param, code);
22    ...
```

case split on instruction

Operational Semantics

```
1  datatype Expr = Nop | Call(meth:Method) | ... | Exception
2  datatype Result = result(l:LC, s:SC) | Stuck
3  function method eval(s:SC) : Result {
4    if(!hasSf(s)) then Stuck else
5    ...
6    var expr := topCont(s).o[0];
7    var s1 := advance(s);
8    ...
9    match expr
10    case Exception => Stuck
11    case Nop => result(Tau, s1)
12    ...
13    case Call(meth) =>
14     if |topOperands(s).o| < 2 then Stuck
15     else
16       var receiver := topOperands(s).o[0];
17       var param := topOperands(s).o[1];
18       var klass := KlassOf(s1, receiver).o;
19       var code := codeForMeth(version(s1), klass, meth);
20       var sr := pushNewFrameForMethod
21                 (s1, receiver, IsMethod(klass, meth), param, code);
22    ...
```

call instruction

Operational Semantics

```
1  datatype Expr = Nop | Call(meth:Method) | ... | Exception
2  datatype Result = result(l:LC, s:SC) | Stuck
3  function method eval(s:SC) : Result {
4    if(!hasSf(s)) then Stuck else
5    ...
6    var expr := topCont(s).o[0];
7    var s1 := advance(s);
8    ...
9    match expr
10    case Exception => Stuck
11    case Nop => result(Tau, s1)
12    ...
13    case Call(meth) =>
14     if |topOperands(s).o| < 2 then Stuck
15     else
16       var receiver := topOperands(s).o[0];
17       var param := topOperands(s).o[1];
18       var klass := KlassOf(s1, receiver).o;
19       var code := codeForMeth(version(s1), klass, meth);
20       var sr := pushNewFrameForMethod
21                 (s1, receiver, IsMethod(klass, meth), param, code);
22    ...
```

lookup code by version

Operational Semantics

```
1  datatype Expr = Nop | Call(meth:Method) | ... | Exception
2  datatype Result = result(l:LC, s:SC) | Stuck
3  function method eval(s:SC) : Result {
4    if(!hasSf(s)) then Stuck else
5    ...
6    var expr := topCont(s).o[0];
7    var s1 := advance(s);
8    ...
9    match expr
10    case Exception => Stuck
11    case Nop => result(Tau, s1)
12    ...
13    case Call(meth) =>
14     if |topOperands(s).o| < 2 then Stuck
15     else
16       var receiver := topOperands(s).o[0];
17       var param := topOperands(s).o[1];
18       var klass := KlassOf(s1, receiver).o;
19       var code := codeForMeth(version(s1), klass, meth);
20       var sr := pushNewFrameForMethod
21                   (s1, receiver, IsMethod(klass, meth), param, code);
22    ...
```

calculate next state

Extras

Theorems In Dafny

Theorems/Lemmas

proof by induction over a sequence

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  ls2,s4 :| Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
8  var al :| StateComp(s1,s2,al) ∧ TraceComp(ls1,ls2,al);
9
10 if(ls1 = []) { } else {
11 var l1,l2,ls5,ls6 := ls1[0], ls2[0], ls1[1..], ls2[1..];
12 ...
13 assert [l1]+ls5 = ls1; assert [l2]+ls6 = ls2;
14
15 var s5 :| Step(s1,l1,s5) ∧ Steps(s5,ls5,s3);
16 var s6 :| Step(s2,l2,s6) ∧ Steps(s6,ls6,s4);
17
18 var al' := StepPreservesTraceComp(s1,l1,s5,ls5, s2,l2,s6,ls6, al);
19
20 StepPreservesTraceEquiv(s1,l1,s5,ls5,s3, s2,l2,s6,ls6,s4);
21
22 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
23 DeterminismSeq(s6,ls6,s4,ls6',s4');
24 }}
```

Theorems/Lemmas

ghost procedure - imperative

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  ls2,s4 :| Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
8  var al :| StateComp(s1,s2,al) ∧ TraceComp(ls1,ls2,al);
9
10 if(ls1 = []) { } else {
11 var l1,l2,ls5,ls6 := ls1[0], ls2[0], ls1[1..], ls2[1..];
12 ...
13 assert [l1]+ls5 = ls1; assert [l2]+ls6 = ls2;
14
15 var s5 :| Step(s1,l1,s5) ∧ Steps(s5,ls5,s3);
16 var s6 :| Step(s2,l2,s6) ∧ Steps(s6,ls6,s4);
17
18 var al' := StepPreservesTraceComp(s1,l1,s5,ls5, s2,l2,s6,ls6, al);
19
20 StepPreservesTraceEquiv(s1,l1,s5,ls5,s3, s2,l2,s6,ls6,s4);
21
22 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
23 DeterminismSeq(s6,ls6,s4,ls6',s4');
24 }}
```

Theorems/Lemmas

parameters universally quantified
returns existentially quantified

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  ls2,s4 :| Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
8  var al :| StateComp(s1,s2,al) ∧ TraceComp(ls1,ls2,al);
9
10 if(ls1 = []) { } else {
11 var l1,l2,ls5,ls6 := ls1[0], ls2[0], ls1[1..], ls2[1..];
12 ...
13 assert [l1]+ls5 = ls1; assert [l2]+ls6 = ls2;
14
15 var s5 :| Step(s1,l1,s5) ∧ Steps(s5,ls5,s3);
16 var s6 :| Step(s2,l2,s6) ∧ Steps(s6,ls6,s4);
17
18 var al' := StepPreservesTraceComp(s1,l1,s5,ls5, s2,l2,s6,ls6, al);
19
20 StepPreservesTraceEquiv(s1,l1,s5,ls5,s3, s2,l2,s6,ls6,s4);
21
22 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
23 DeterminismSeq(s6,ls6,s4,ls6',s4');
24 }}
```

Theorems/Lemmas prove precondition implies post condition

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  ls2,s4 :| Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
8  var al :| StateComp(s1,s2,al) ∧ TraceComp(ls1,ls2,al);
9
10 if(ls1 = []) { } else {
11 var l1,l2,ls5,ls6 := ls1[0], ls2[0], ls1[1..], ls2[1..];
12 ...
13 assert [l1]+ls5 = ls1; assert [l2]+ls6 = ls2;
14
15 var s5 :| Step(s1,l1,s5) ∧ Steps(s5,ls5,s3);
16 var s6 :| Step(s2,l2,s6) ∧ Steps(s6,ls6,s4);
17
18 var al' := StepPreservesTraceComp(s1,l1,s5,ls5, s2,l2,s6,ls6, al);
19
20 StepPreservesTraceEquiv(s1,l1,s5,ls5,s3, s2,l2,s6,ls6,s4);
21
22 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
23 DeterminismSeq(s6,ls6,s4,ls6',s4');
24 }}
```

Theorems/Lemmas

assign such-that predicate is satisfied

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  ls2,s4 :| Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
8  var al :| StateComp(s1,s2,al) ∧ TraceComp(ls1,ls2,al);
9
10 if(ls1 = []) { } else {
11 var l1,l2,ls5,ls6 := ls1[0], ls2[0], ls1[1..], ls2[1..];
12 ...
13 assert [l1]+ls5 = ls1; assert [l2]+ls6 = ls2;
14
15 var s5 :| Step(s1,l1,s5) ∧ Steps(s5,ls5,s3);
16 var s6 :| Step(s2,l2,s6) ∧ Steps(s6,ls6,s4);
17
18 var al' := StepPreservesTraceComp(s1,l1,s5,ls5, s2,l2,s6,ls6, al);
19
20 StepPreservesTraceEquiv(s1,l1,s5,ls5,s3, s2,l2,s6,ls6,s4);
21
22 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
23 DeterminismSeq(s6,ls6,s4,ls6',s4');
24 }}
```

Theorems/Lemmas

Dafny proves base case automatically

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  ls2,s4 :| Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
8  var al :| StateComp(s1,s2,al) ∧ TraceComp(ls1,ls2,al);
9
10 if(ls1 = []) { } else {
11 var l1,l2,ls5,ls6 := ls1[0], ls2[0], ls1[1..], ls2[1..];
12 ...
13 assert [l1]+ls5 = ls1; assert [l2]+ls6 = ls2;
14
15 var s5 :| Step(s1,l1,s5) ∧ Steps(s5,ls5,s3);
16 var s6 :| Step(s2,l2,s6) ∧ Steps(s6,ls6,s4);
17
18 var al' := StepPreservesTraceComp(s1,l1,s5,ls5, s2,l2,s6,ls6, al);
19
20 StepPreservesTraceEquiv(s1,l1,s5,ls5,s3, s2,l2,s6,ls6,s4);
21
22 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
23 DeterminismSeq(s6,ls6,s4,ls6',s4');
24 }}
```

Theorems/Lemmas asserts are in-line lemmas

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  ls2,s4 :| Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
8  var al :| StateComp(s1,s2,al) ∧ TraceComp(ls1,ls2,al);
9
10 if(ls1 = []) { } else {
11 var l1,l2,ls5,ls6 := ls1[0], ls2[0], ls1[1..], ls2[1..];
12 ...
13 assert [l1]+ls5 = ls1; assert [l2]+ls6 = ls2;
14
15 var s5 :| Step(s1,l1,s5) ∧ Steps(s5,ls5,s3);
16 var s6 :| Step(s2,l2,s6) ∧ Steps(s6,ls6,s4);
17
18 var al' := StepPreservesTraceComp(s1,l1,s5,ls5, s2,l2,s6,ls6, al);
19
20 StepPreservesTraceEquiv(s1,l1,s5,ls5,s3, s2,l2,s6,ls6,s4);
21
22 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
23 DeterminismSeq(s6,ls6,s4,ls6',s4');
24 }}
```

Theorems/Lemmas — apply induction hypothesis

```
1  lemma TraceEquivStepsPreservesComp
2    (s1:S,ls1:seq<L>,s3:S,s2:S) returns (ls2:seq<L>,s4:S)
3   requires Steps(s1,ls1,s3) ∧ TraceEquiv(s1,ls1,s3,s2);
4   ensures Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
5   ensures ∃al :: StateComp(s3,s4,al);
6  {
7  ls2,s4 :| Steps(s2,ls2,s4) ∧ StepsComp(s1,ls1,s3, s2,ls2,s4);
8  var al :| StateComp(s1,s2,al) ∧ TraceComp(ls1,ls2,al);
9
10 if(ls1 = []) { } else {
11 var l1,l2,ls5,ls6 := ls1[0], ls2[0], ls1[1..], ls2[1..];
12 ...
13 assert [l1]+ls5 = ls1; assert [l2]+ls6 = ls2;
14
15 var s5 :| Step(s1,l1,s5) ∧ Steps(s5,ls5,s3);
16 var s6 :| Step(s2,l2,s6) ∧ Steps(s6,ls6,s4);
17
18 var al' := StepPreservesTraceComp(s1,l1,s5,ls5, s2,l2,s6,ls6, al);
19
20 StepPreservesTraceEquiv(s1,l1,s5,ls5,s3, s2,l2,s6,ls6,s4);
21
22 var ls6',s4' := TraceEquivStepsPreservesComp(s5,ls5,s3,s6);
23 DeterminismSeq(s6,ls6,s4,ls6',s4');
24 }}
```