The Implementation of Object Propositions: the Oprop Verification Tool

Ligia Nistor and Jonathan Aldrich

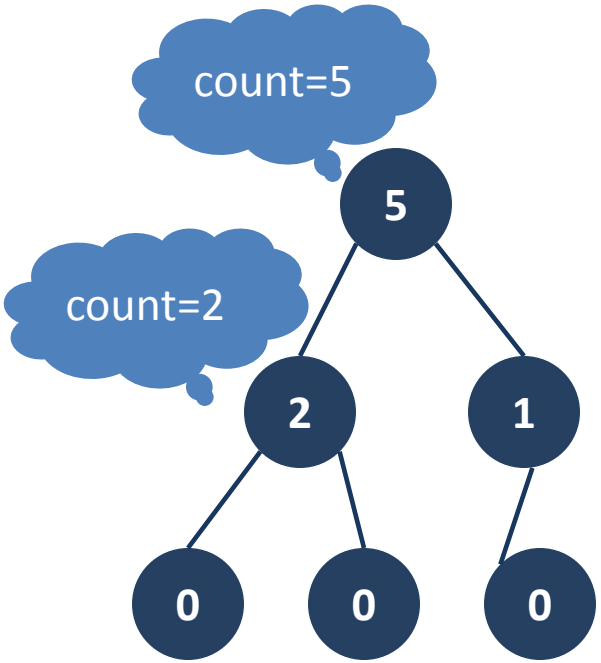
Carnegie Mellon University
Pittsburgh, PA, USA

Motivation

- Verify different components **independently**
- Components = **classes** or **methods** written in an object oriented language
- Efficient **static formal verification** of object oriented programs still an open problem

Formal verification

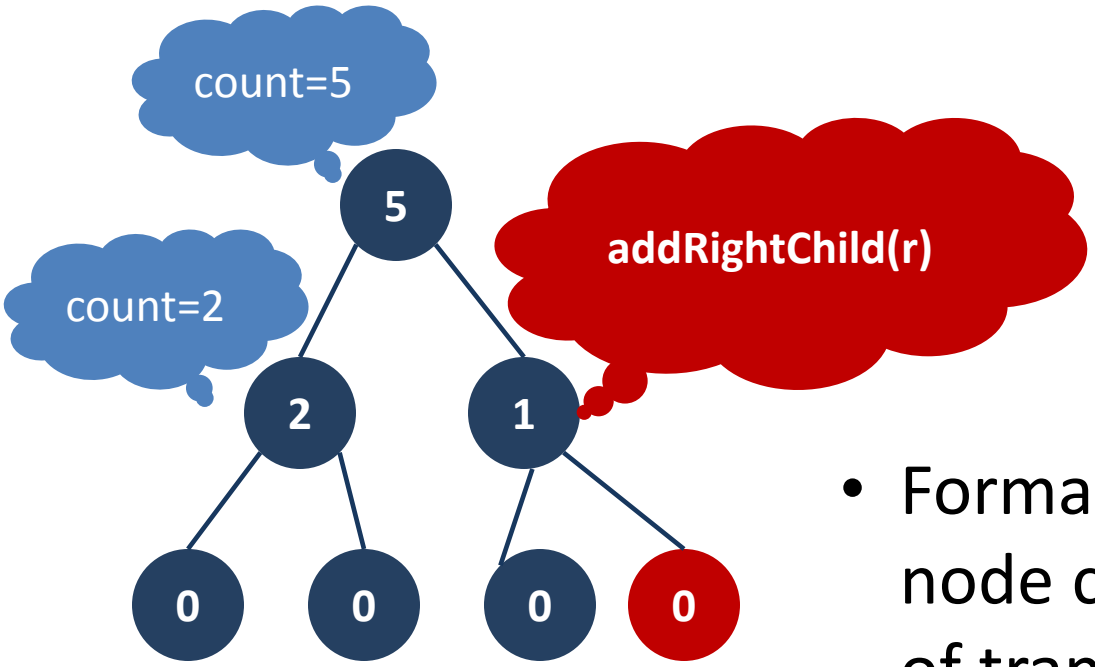
- Use formal rules to reason about correctness of programs



- Formally check that each node contains the number of transitive children

Formal verification

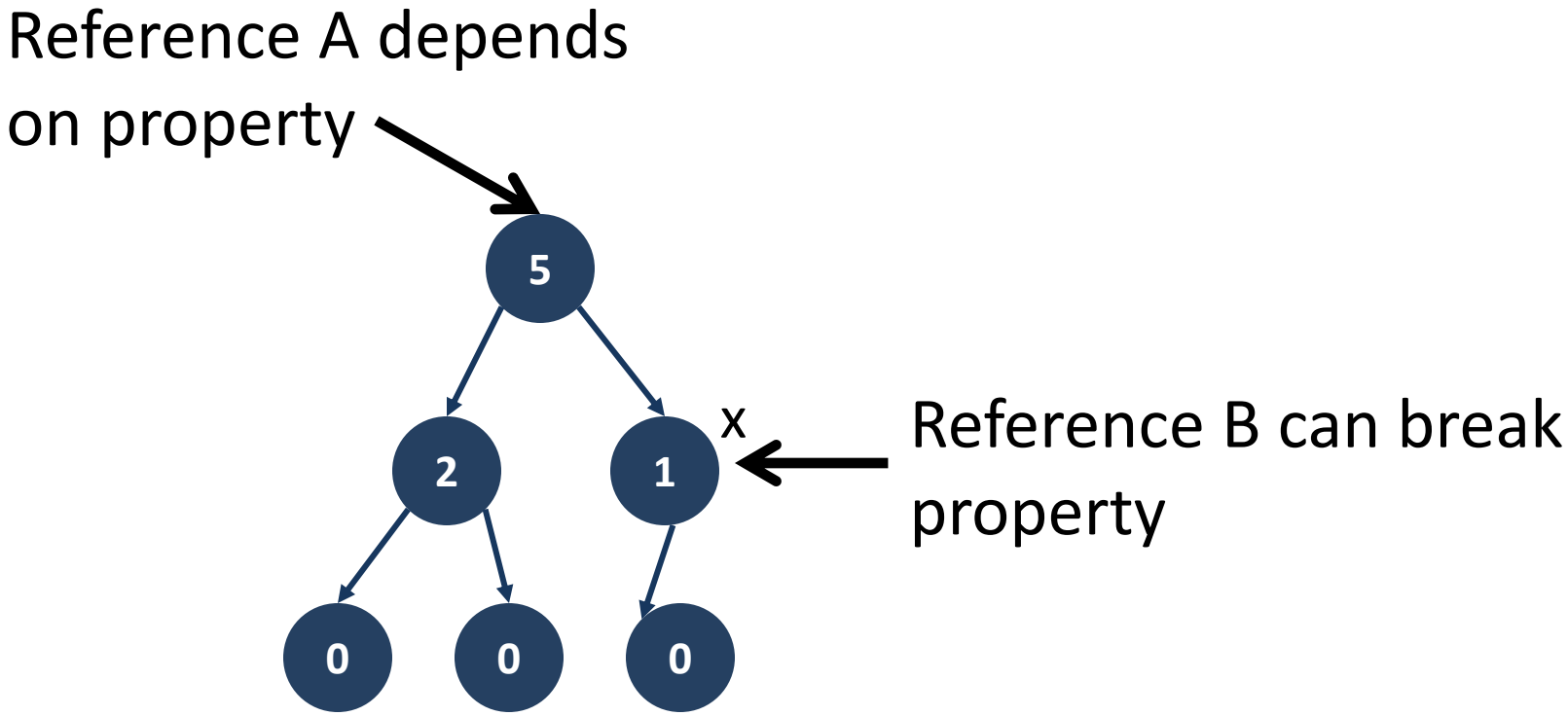
- Use formal rules to reason about correctness of programs



- Formally check that each node contains the number of transitive children

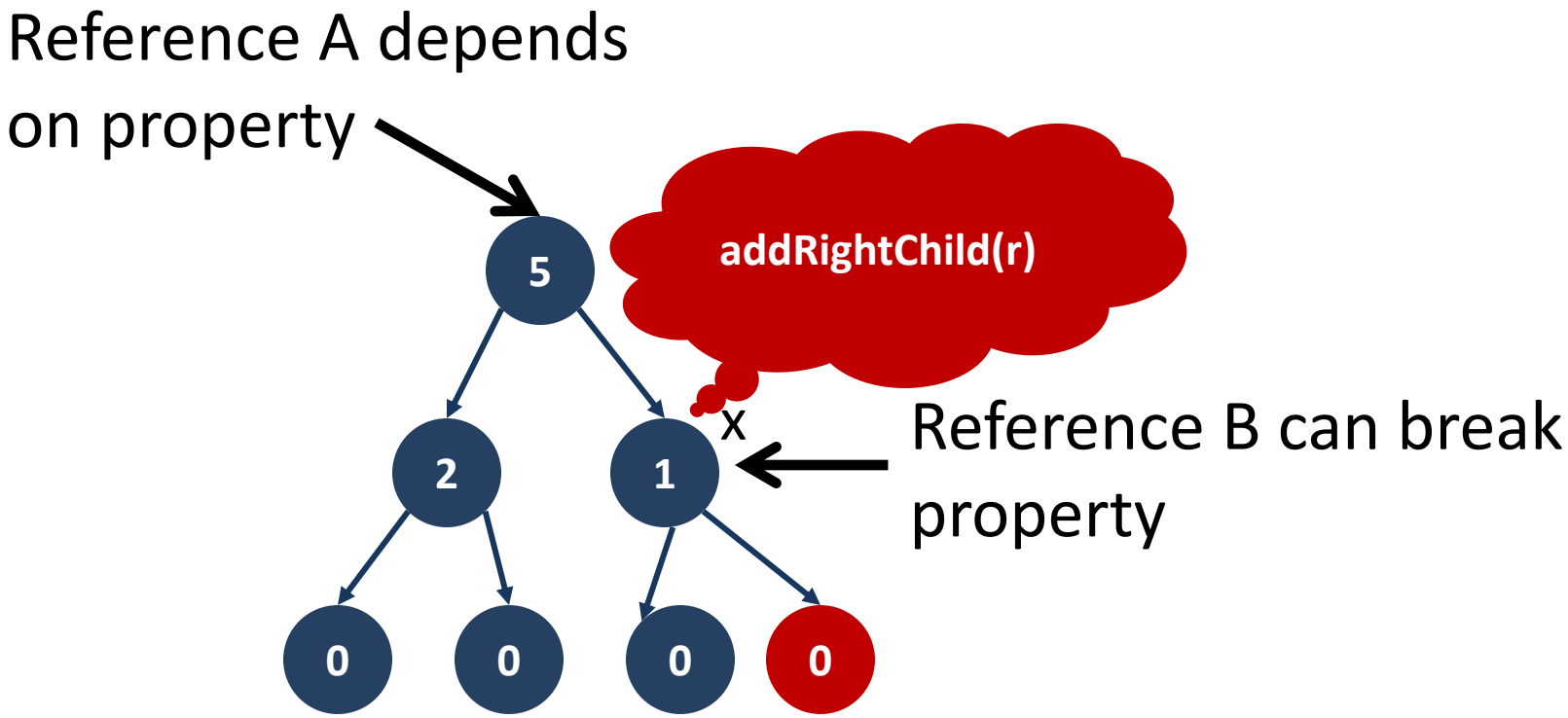
Verifying Object Oriented Code

- Why is it difficult?
  - aliasing



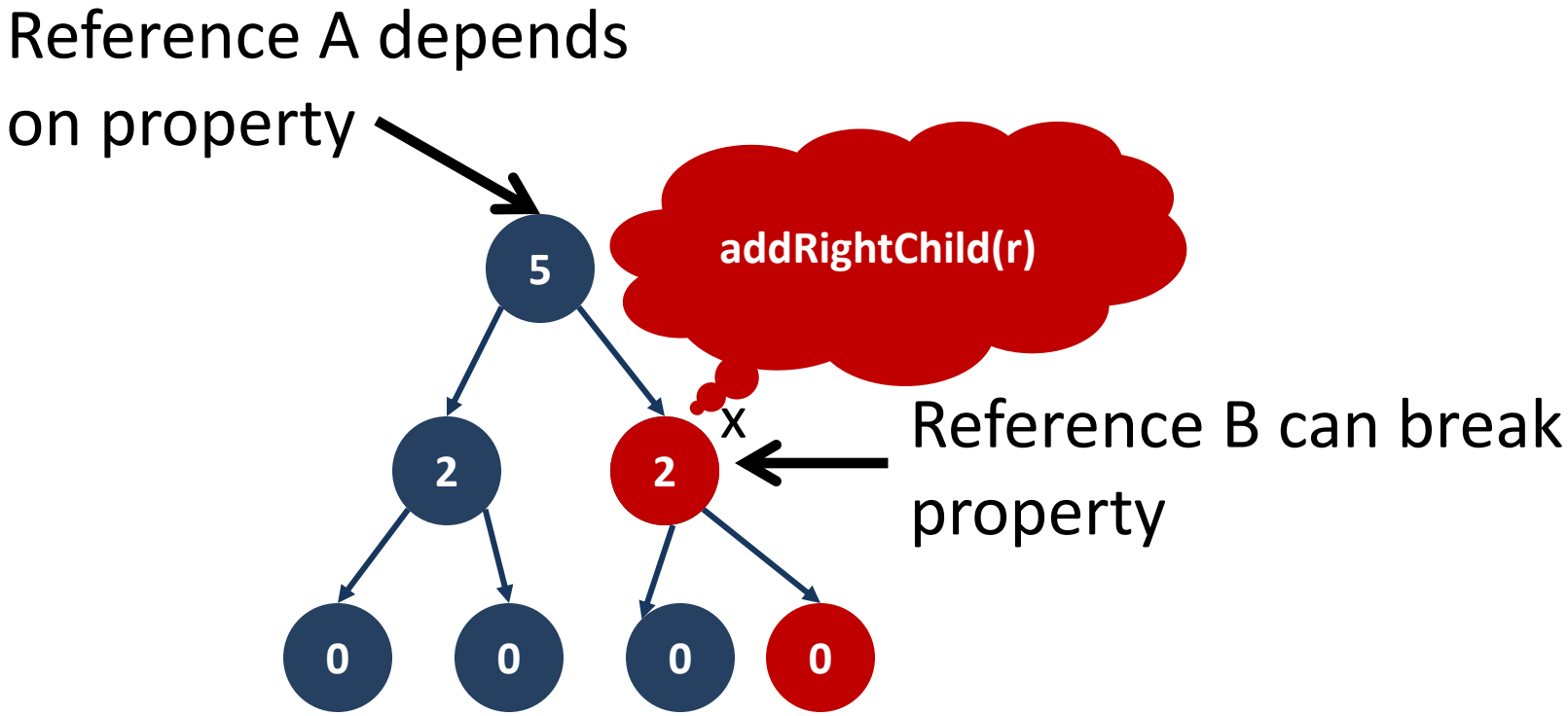
Verifying Object Oriented Code

- Why is it difficult?
  - aliasing



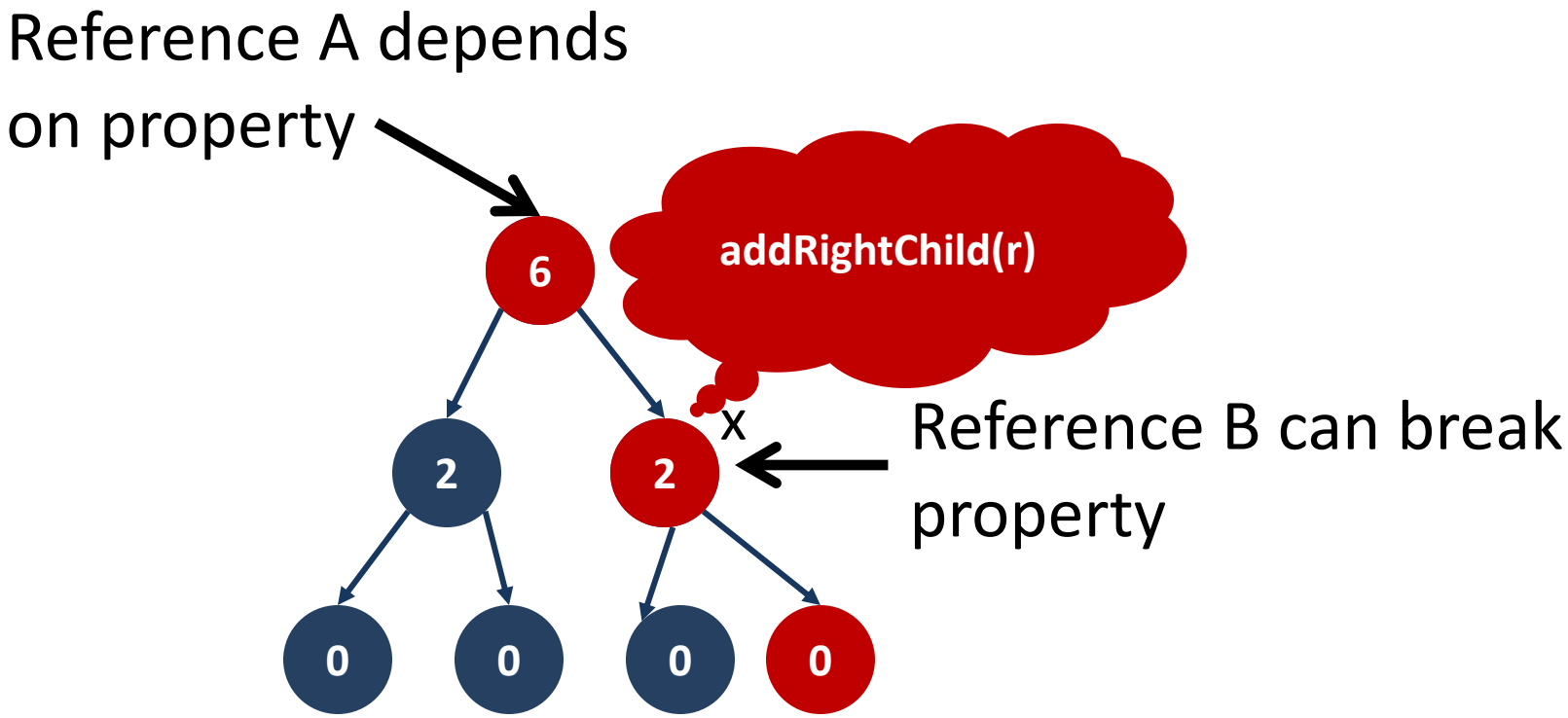
Verifying Object Oriented Code

- Why is it difficult?
  - aliasing



Verifying Object Oriented Code

- Why is it difficult?
  - aliasing



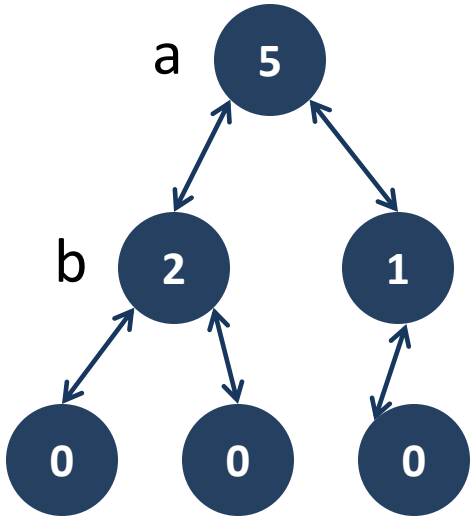
Object Propositions

- New verification methodology 

- Express specifications about objects ➔ object propositions
- Modularity ➔ verify each component independently
- Single-thread

Abstract Predicates

- Predicate **count(int c)** = the **count** of the current Composite object is number of its transitive children



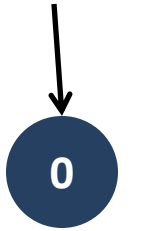
object a satisfies **count(5)**

object b satisfies **count(2)**

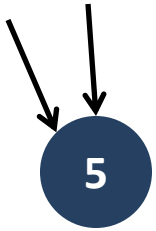
[M. Parkinson]

Fractional permissions

- dealing with aliases



permission of 1
read/write access



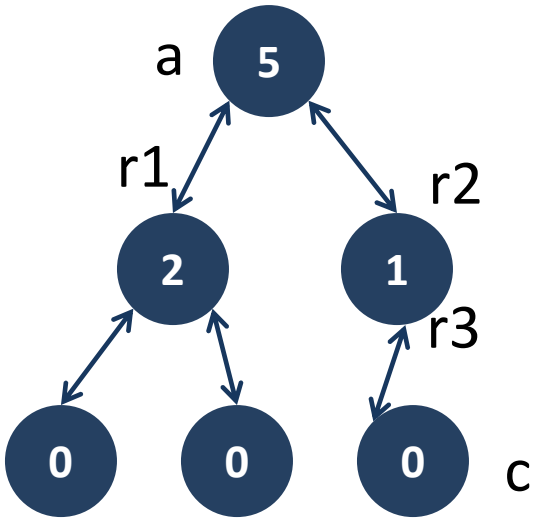
permission of 1/2
read/write access,
as long as the initial
predicate is maintained

Contribution: The state referred to by a fraction < 1 is not immutable.
That state satisfies an invariant that can be relied on by other objects.

[Boyland]

Putting it together

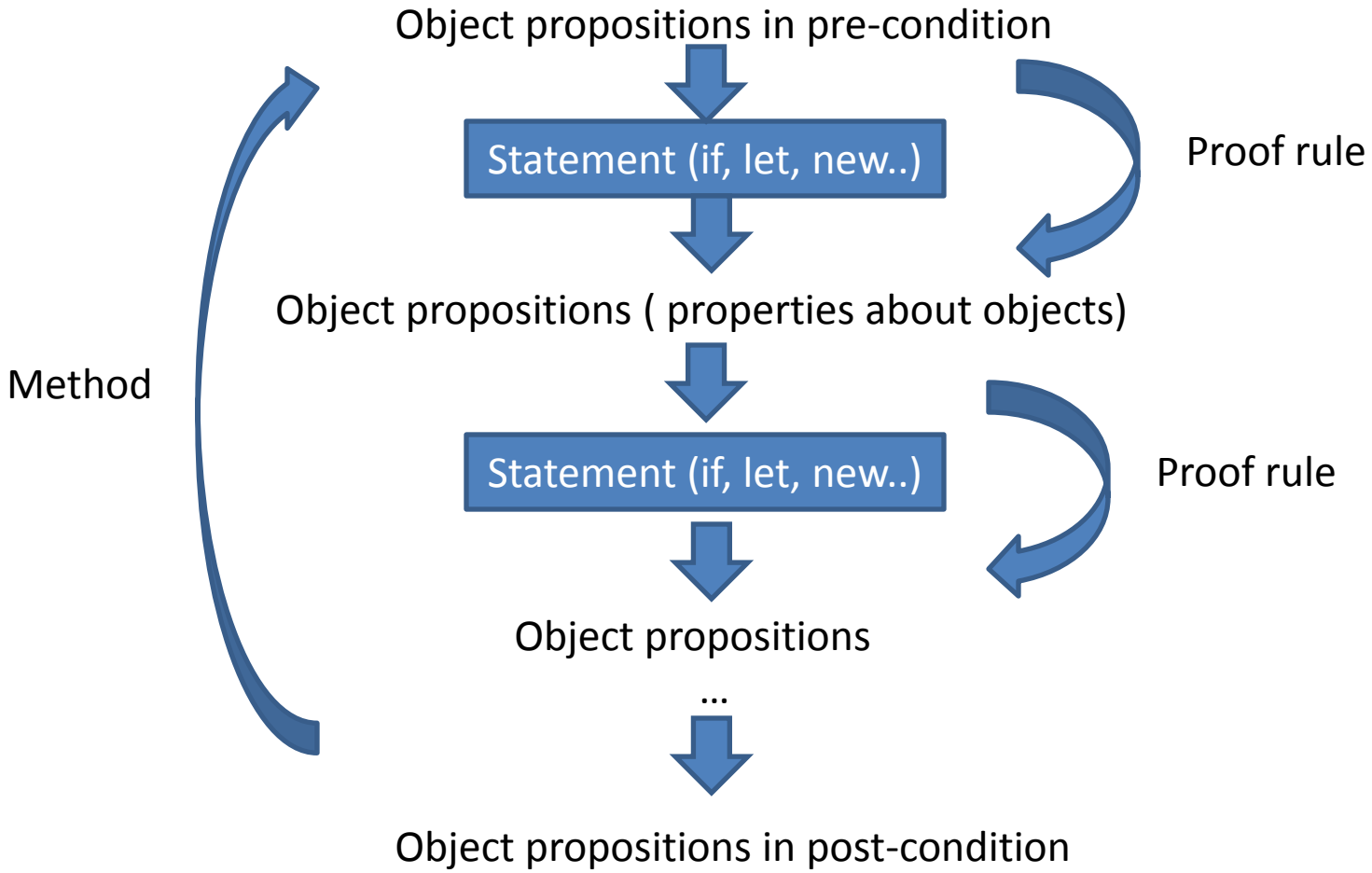
- Object proposition =
  abstract predicate + fractional permission



- r1#1/2 count(5)
- r3#1 count(0)

The Verification of a Method

- Using proof rules



Linear logic

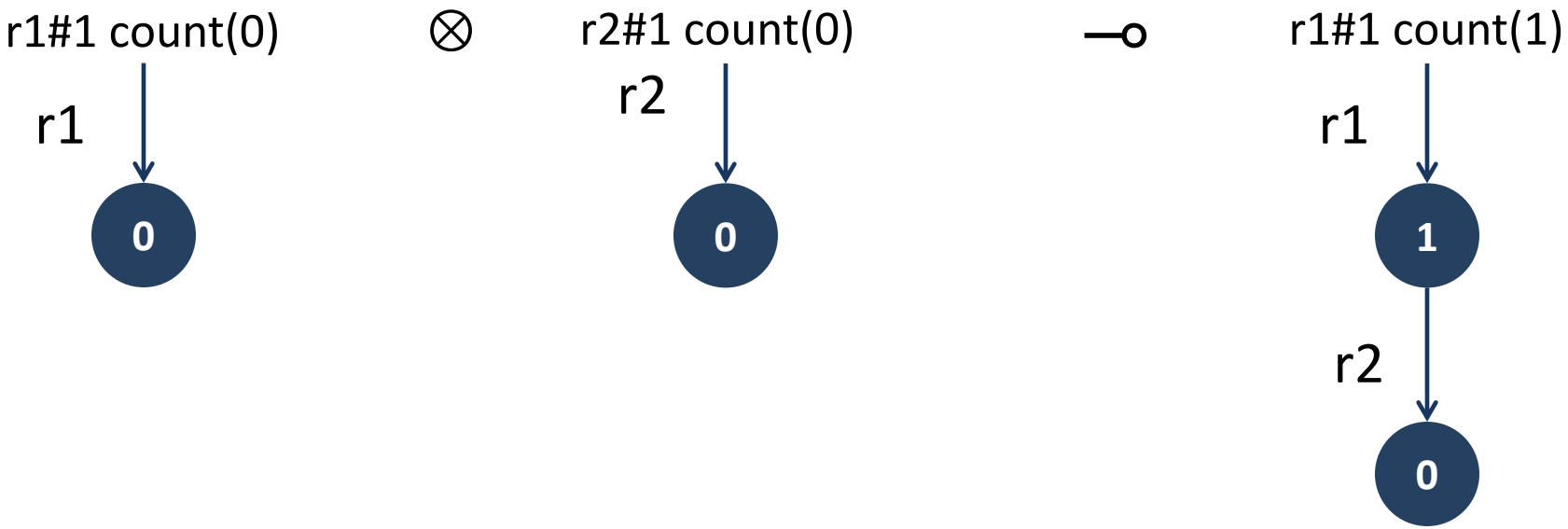
- Classical logic: from A and (A ⇒ B) get (A ∧ B)

A   B

- Linear logic: from A and (A ⊸ B) get B (**transform**)
  - Logic of resources
  - ⊗ **Simultaneous** occurrence of resources
  - ⊕ **Alternative** occurrence of resources

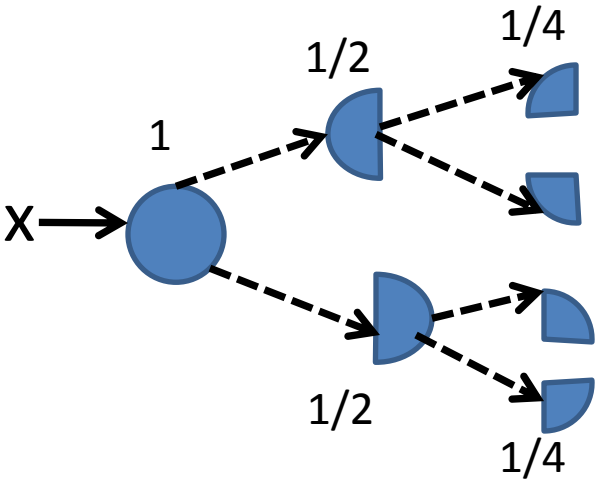
Connect

- Object propositions = resources consumed upon usage
- buildSimpleTree(r1,r2):



Formal system

- Rules for **splitting/adding** fractions



- x#1 ⇔ x#1/2 ⊗ x#1/2
- x#k ⇔ x#k/2 ⊗ x#k/2

[Boyland]

Pack, unpack

- Abstraction:

  Predicate: from outside → Count(c)

  from inside → get to the fields

- **pack** to a predicate

- **unpack** a predicate: gain access to fields of object

Consistency

- packed predicate → consistent
- unpacked predicate → inconsistent
- In a method, after the first assignment to a field, the unpacked predicate is inconsistent
- We have aliasing and fractions, how come unpacking is still sound?
- As long as we have a fraction to an object, we know that the invariant of that object will not be broken. When we pack back the predicate, the invariant is restored.

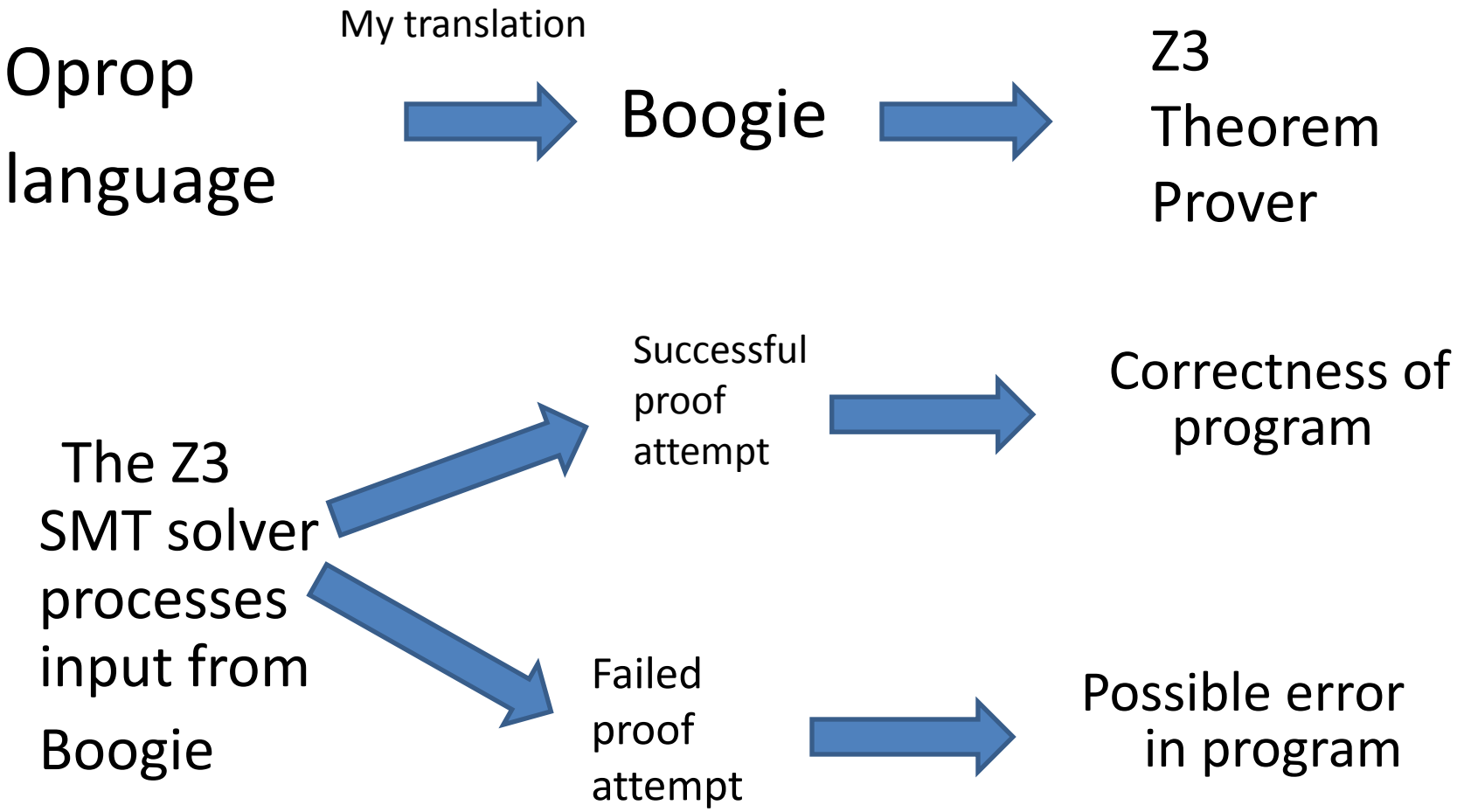
- We assume termination, so at end of program all objects are packed

Oprop Grammar – Featherweight Java

$$
\begin{array}{rcl}
\text{Prog} & ::= & \overline{\text{ClDecl}}\ \text{e} \\
\text{ClDecl} & ::= & \texttt{class}\ C\ \{\ \overline{\text{FldDecl}}\ \overline{\text{PredDecl}}\ \overline{\text{MthDecl}}\ \} \\
\text{FldDecl} & ::= & \text{T}\ f \\
\text{PredDecl} & ::= & \texttt{predicate}\ Q(\overline{\text{T x}}) \equiv \text{R} \\
\text{MthDecl} & ::= & \text{T}\ m(\overline{\text{T x}})\ \text{MthSpec}\ \{\ \bar{\text{e}};\ \texttt{return e}\ \} \\
\text{MthSpec} & ::= & \text{R} \multimap \text{R} \\
\text{R} & ::= & \text{P} \mid \text{R} \otimes \text{R} \mid \text{R} \oplus \text{R} \mid \\
 & & \exists \text{x:T.R} \mid \exists \text{z:double.R} \mid \exists \text{z:double.z binop t} \Rightarrow \text{R} \mid \\
 & & \forall \text{x:T.R} \mid \forall \text{z:double.R} \mid \forall \text{z:double.z binop t} \Rightarrow \text{R} \mid \\
 & & \text{t binop t} \Rightarrow \text{R} \\
\text{P} & ::= & r\#\text{k}\ Q(\bar{\text{t}}) \mid \texttt{unpacked}(r\#\text{k}\ Q(\bar{\text{t}})) \mid \\
 & & r.f \rightarrow \text{x} \mid \text{t binop t} \\
\text{k} & ::= & \frac{n_1}{n_2}\ (\text{where } n_1, n_2 \in \mathbb{N} \text{ and } 0 < n_1 \leq n_2) \mid \text{z} \\
\text{e} & ::= & \text{t} \mid r.f \mid r.f = \text{t} \mid r.m(\bar{\text{t}}) \mid \\
 & & \texttt{new}\ C(Q(\bar{\text{t}})\,[\bar{\text{t}}])(\bar{\text{t}}) \mid \\
 & & \texttt{if (t) \{ e \} else \{ e \}} \mid \texttt{let x = e in e} \mid \\
 & & \text{t binop t} \mid \text{t \&\& t} \mid \text{t} \,\|\, \text{t} \mid \texttt{!}\,\text{t} \mid \\
 & & \texttt{pack}\ r\#\text{k}\ Q(\bar{\text{t}})\,[\bar{\text{t}}]\,\texttt{in e} \mid \texttt{unpack}\ r\#\text{k}\ Q(\bar{\text{t}})\,[\bar{\text{t}}]\,\texttt{in e} \\
\text{t} & ::= & \text{x} \mid n \mid \texttt{null} \mid \texttt{true} \mid \texttt{false} \\
\text{x} & ::= & r \mid i \\
\text{binop} & ::= & + \mid - \mid \% \mid = \mid != \mid \leq \mid < \mid \geq \mid > \\
\text{T} & ::= & C \mid \texttt{int} \mid \texttt{boolean} \mid \texttt{double}
\end{array}
$$

Boogie: a language and a tool

- Intermediate verification language



Boogie: 6 kinds of declarations

1). **type** Car;

2). **const** renault : Car;

3). **function** age(Car) **returns** (**int**);

4). **axiom** age(renault)==7;

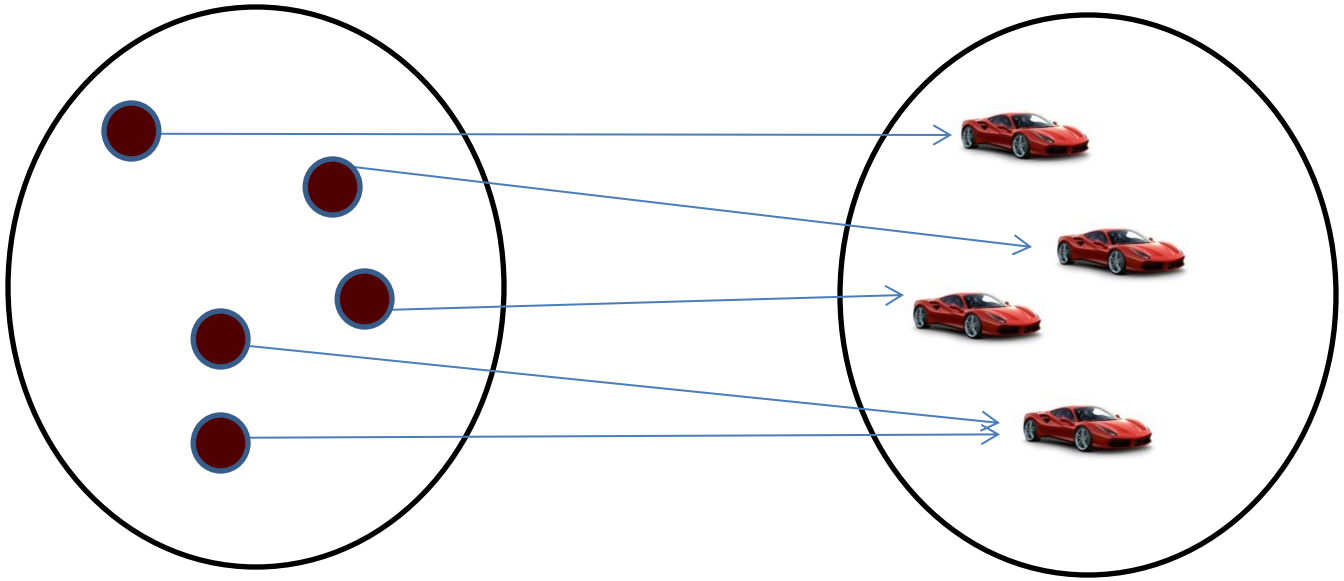
5). **var** favorite: Car;

6). procedure NewFavorite(ferari:Car);
modifies favorite;
requires favorite == renault;
ensures favorite == ferari;
{ favorite := ferari;}

Maps in Boogie

class Family { Car fieldCar; Address fieldA; …}

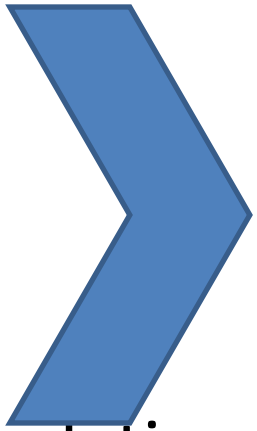
- We represent class fields using a map per field. Each map goes from objects to the field of that object.



var fieldCar : [Ref]Car;
var fieldA : [Ref]Address;

Architecture of Translation part of Oprop Tool

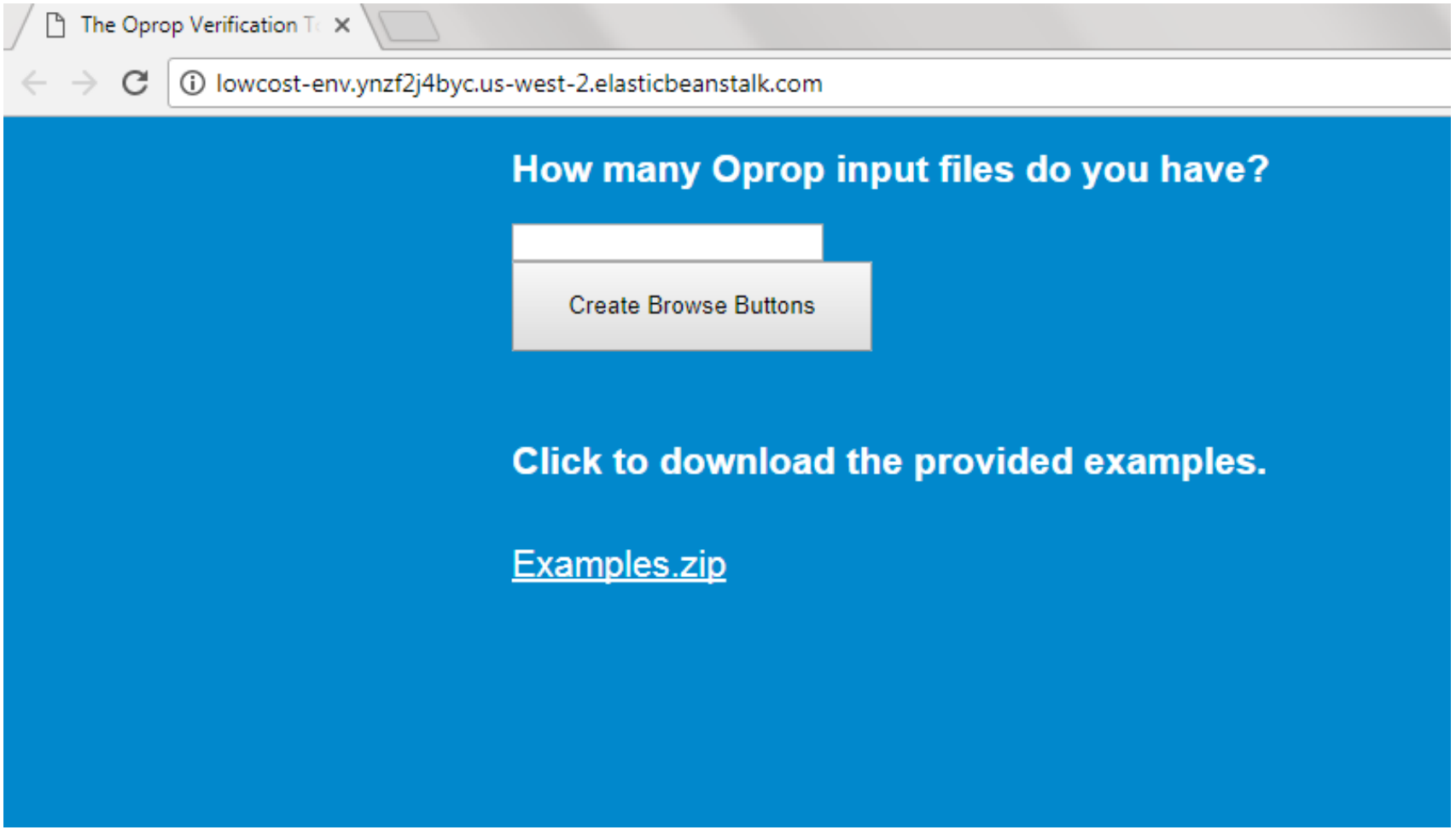
- JavaCC parser – Java subset
- Abstract Syntax Tree classes
- Contextual Analysis Visitor
- Type Resolution Class



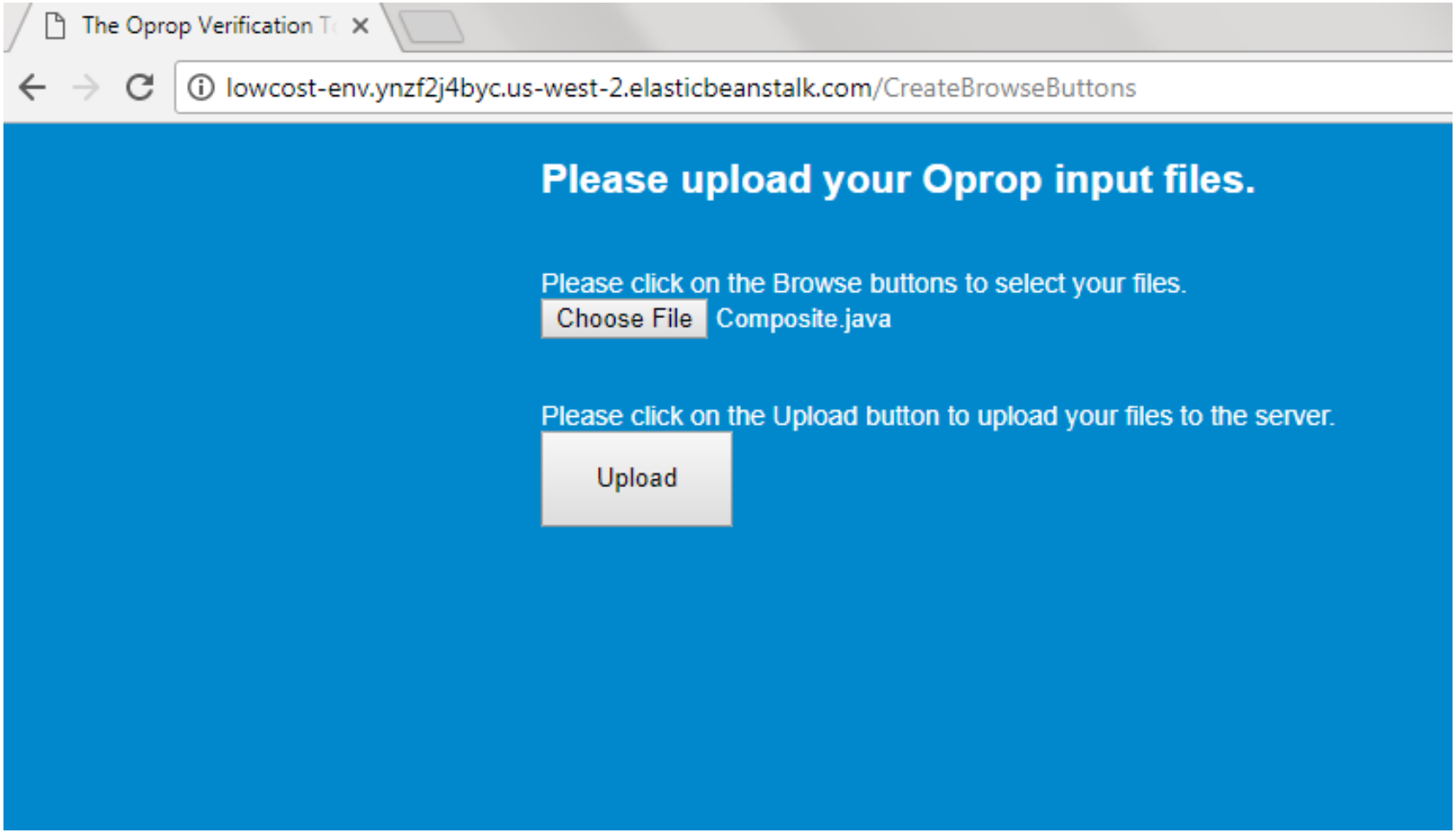
JExpr package

- Boogie Visitor: implements translation rules; Abstract Syntax Tree that visits FieldDeclaration, PredicateDeclaration, MethodDeclaration, Object Proposition, Binary Expression
- Translation sent to the Boogie tool, then to Z3

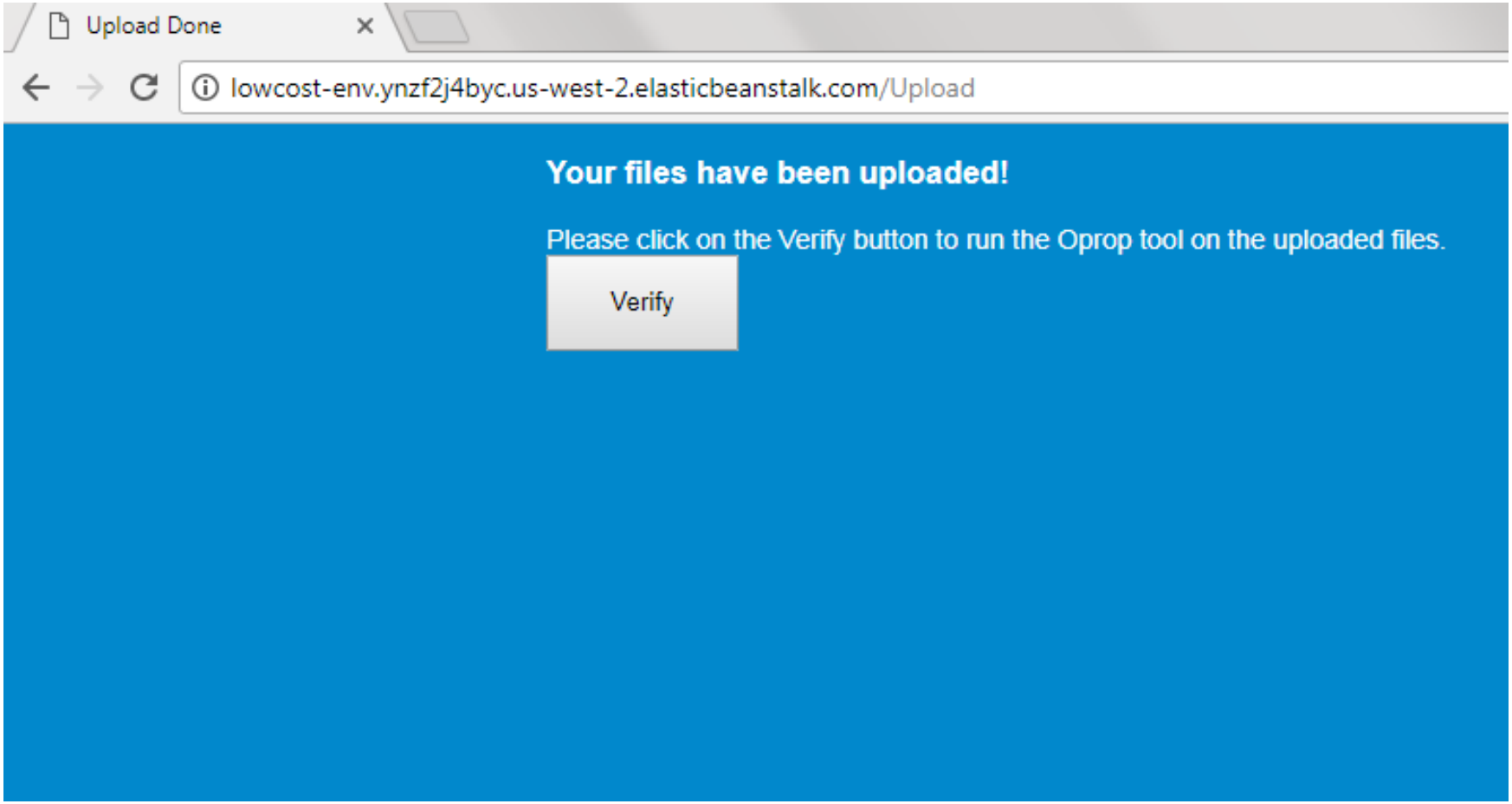
Oprop Online Tool – 1st webpage



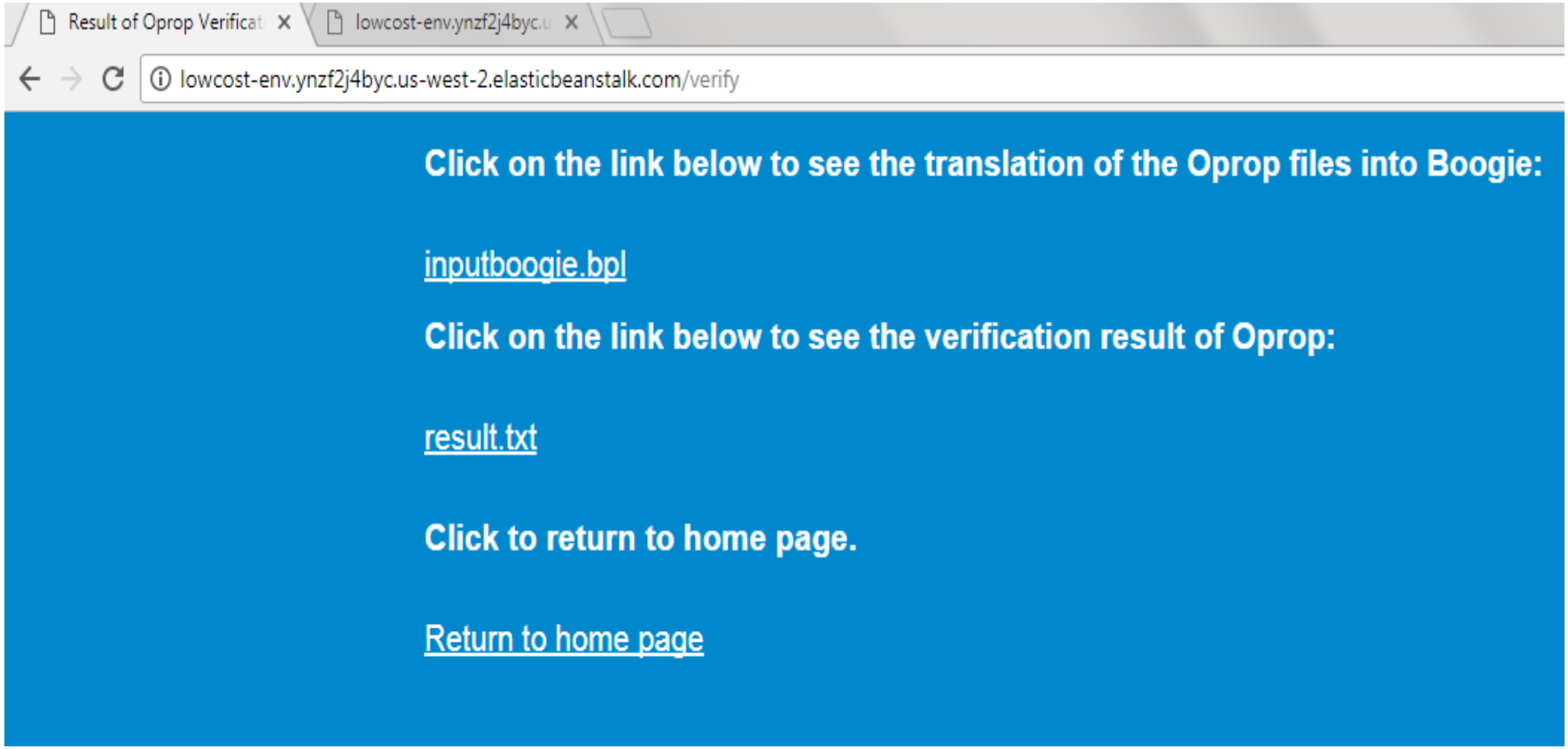
Oprop Online Tool – 2nd webpage



Oprop Online Tool – 3rd webpage



Oprop Online Tool – 4th webpage



SimpleCell.java

```
package x;
class SimpleCell {
 int val;
 SimpleCell next;
 predicate PredVal() = exists int v : this.val -> v && v<15
 predicate PredNext() = exists SimpleCell obj :
  this.next -> obj && (obj#0.34 PredVal())
 void changeVal(int r)
 ~double k : requires (this#k PredVal()) && (r<15)
 ensures this#k PredVal()
 {
  unpack(this#k PredVal())[this.val];
  this.val = r;
  pack(this#k PredVal())[r];
 }
 void main() {
  SimpleCell c = new SimpleCell(PredVal()[2])(2, null);
  SimpleCell a = new SimpleCell(PredNext()[c])(2, c);
  SimpleCell b = new SimpleCell(PredNext()[c])(3, c);
  c.changeVal(4);
 }
}
```

Translation of Fields

```
1  type Ref;
2  const null: Ref;
3  var val: [Ref]int;
4  var next: [Ref]Ref;
5  var packedPredNext: [Ref] bool;
6  var fracPredNext: [Ref] real;
7  var packedPredVal: [Ref] bool;
8  var fracPredVal: [Ref] real;
```

Translation of Predicates

```
19  procedure PackPredVal(v:int, this:Ref);
20    requires (packedPredVal[this]==false) &&
21      ((v<15)) && (val[this]==v);
22  procedure UnpackPredVal(v:int, this:Ref);
23    requires packedPredVal[this] &&
24      (fracPredVal[this] > 0.0);
25    requires (val[this]==v);
26    ensures ((v<15)) && (val[this]==v);
```

Translation of Method Specifications

```
36  procedure changeVal(r:int, this:Ref)
37    modifies packedPredVal,val;
38    requires (this != null) && (((packedPredVal[this] ) &&
39      (fracPredVal[this] > 0.0))&&(r<15));
40    ensures ((packedPredVal[this] ) &&
41      (fracPredVal[this] > 0.0));
42    requires (forall x:Ref :: packedPredVal[x]);
43    ensures (forall x:Ref :: packedPredVal[x]);
```

Translation of Method Bodies

```
65    var c:Ref;
66    var a:Ref;
67    var b:Ref;
68    assume (c!=a) && (c!=b) && (a!=b) ;
69    assume (forall y:Ref :: (fracPredNext[y] >= 0.0) );
70    call SimpleCell(2,null,c);
71    packedPredVal[c] := false;
72    call PackPredVal(2,c);
73    packedPredVal[c] := true;
74    fracPredVal[c] := 1.0;
```

Semantics for Linear Logic Formulas

$$\frac{\Gamma; \Pi_0; \Pi_1 \vdash r@\mathsf{k}/2\ Q\ (\bar{\mathsf{t}}) \otimes r@\mathsf{k}/2\ Q\ (\bar{\mathsf{t}})}{\Gamma; \Pi_0; \Pi_1 \vdash r@\mathsf{k}\ Q\ (\bar{\mathsf{t}})}\ (OPack_1)$$

$$\frac{\Gamma; \Pi_0; \Pi_1 \vdash r@\mathsf{k}*2\ Q\ (\bar{\mathsf{t}})}{\Gamma; \Pi_0; \Pi_1 \vdash r@\mathsf{k}\ Q\ (\bar{\mathsf{t}}) \otimes r@\mathsf{k}\ Q\ (\bar{\mathsf{t}})}\ (OPack_2)$$

$$\frac{\Gamma; \Pi_0; \Pi_1 \vdash r@\mathsf{k}/2\ Q\ (\bar{\mathsf{t}}) \otimes \mathtt{unpacked}(r@\mathsf{k}/2\ Q\ (\bar{\mathsf{t}}))}{\Gamma; \Pi_0; \Pi_1 \vdash \mathtt{unpacked}(r@\mathsf{k}\ Q\ (\bar{\mathsf{t}}))}\ (OUnpack_1)$$

$$\frac{\Gamma; \Pi_0; \Pi_1 \vdash \mathtt{unpacked}(r@\mathsf{k}/2\ Q\ (\bar{\mathsf{t}})) \otimes \mathtt{unpacked}(r@\mathsf{k}/2\ Q\ (\bar{\mathsf{t}}))}{\Gamma; \Pi_0; \Pi_1 \vdash \mathtt{unpacked}(r@\mathsf{k}\ Q\ (\bar{\mathsf{t}}))}\ (OUnpack_2)$$

$$\frac{\Gamma; \Pi_0; \Pi_1 \vdash \mathtt{unpacked}(r@\mathsf{k}*2\ Q\ (\bar{\mathsf{t}}))}{\Gamma; \Pi_0; \Pi_1 \vdash \mathtt{unpacked}(r@\mathsf{k}\ Q\ (\bar{\mathsf{t}})) \otimes \mathtt{unpacked}(r@\mathsf{k}\ Q\ (\bar{\mathsf{t}}))}\ (OUnpack_3)$$

Soundness Theorem

- For a formula R that is written in linear logic and parses according to our grammar, if trans(R) holds in first order logic then R holds in linear logic.
- Induction on complexity of R, on the depth of logical connectives in the formula

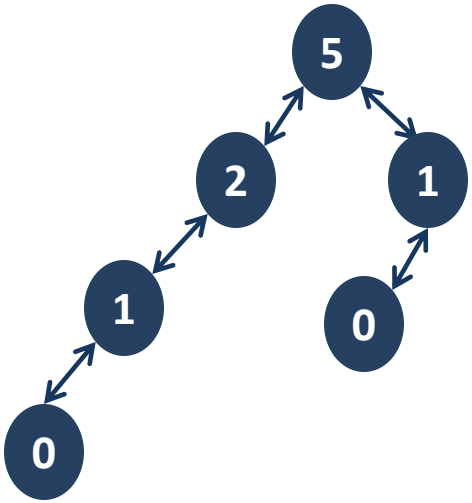
Completeness Theorem

- For a formula R that is written in linear logic and parses according to our grammar, if R holds in linear logic then trans(R) holds in first order logic
- Induction on linear logic rules on previous slide

Verification of Composite Example

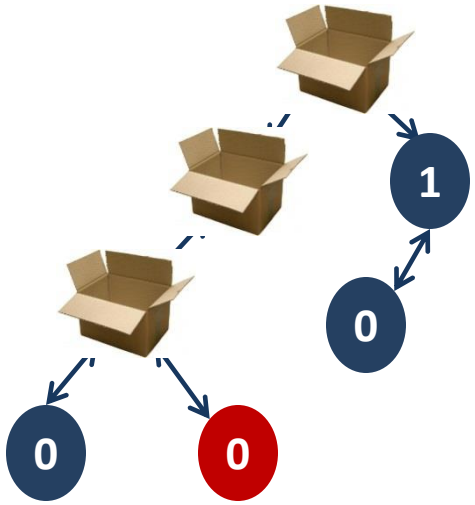
- My thesis verifies a number of design patterns
- This example was manually verified in 2014 when I presented the work during the Formal Methods symposium
- Now I am verifying it automatically using my tool

Solution to Composite



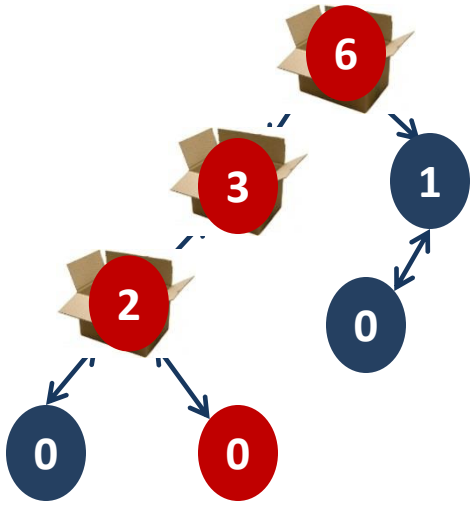
- Notification algorithm goes up the tree→ need to unpack the **count** predicate of the **parent**
- Unpack all nodes on path to root
- To pack the predicates gain, **count** fields must be updated and consistent

Solution to Composite



- Notification algorithm goes up the tree→ need to unpack the **count** predicate of the **parent**
- Unpack all nodes on path to root
- To pack the predicates gain, **count** fields must be updated and consistent

Solution to Composite



- Notification algorithm goes up the tree→ need to unpack the **count** predicate of the **parent**
- Unpack all nodes on path to root
- To pack the predicates gain, **count** fields must be updated and consistent

Implementation and code on GitHub

- https://github.com/ligianistor/Oprop
- https://github.com/ligianistor/Oprop/blob/master/src/testcases/cager/jexpr/Composite.java
- https://github.com/ligianistor/boogie/blob/master/composite2noPackedv7.bpl

Related work

- Bierhoff and Aldrich: access permissions
- Boyland: fractional permissions
- Parkinson: abstract predicates

- Barnett & Leino: Boogie verifier
- Müller : Viper toolchain
- Heule: Chalice verifier

Future Work

- Augment features of Oprop language so that more examples can be verified using Oprop
- Extend for multi-threaded programs

Conclusions

- **Object proposition** = abstract predicate + fractional permission
- **Translation rules** from Oprop to Boogie
- **Soundness and completeness** of the translation from Oprop into the Boogie language
- Automatically verified instance of **Composite Design Pattern** using Oprop