(star) (no star)

Chalice to Boogie

Program Verification for Object-Oriented Programs

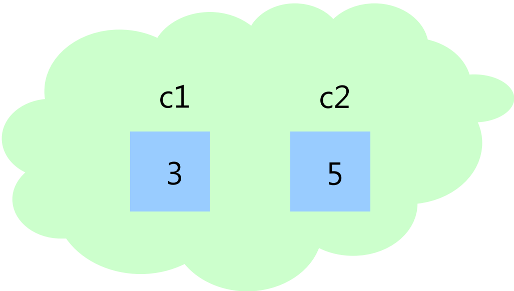
Chinmay Kakatkar

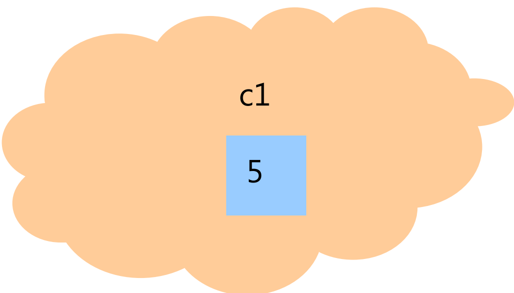
```
class Car
{
    int fuel;

    void refuel ( int amount )
    {
        this.fuel := amount;
    }

    void main ( )
    {
        ...
    }
}
```

```
class Car
{
    int fuel;

    void refuel ( int amount )
    {
        this.fuel := amount;
    }

    void main ( )
    {
        ...
    }
}
```

```
Car c1 := new Car ( );
Car c2 := new Car ( );
c1.refuel ( 3 );
c2.refuel ( 5 );
assert ( c1.fuel == 3 );
```

```
class Car
{
    int fuel;

    void refuel ( int amount )
    {
        this.fuel := amount;
    }

    void main ( )
    {
        ...
    }
}
```



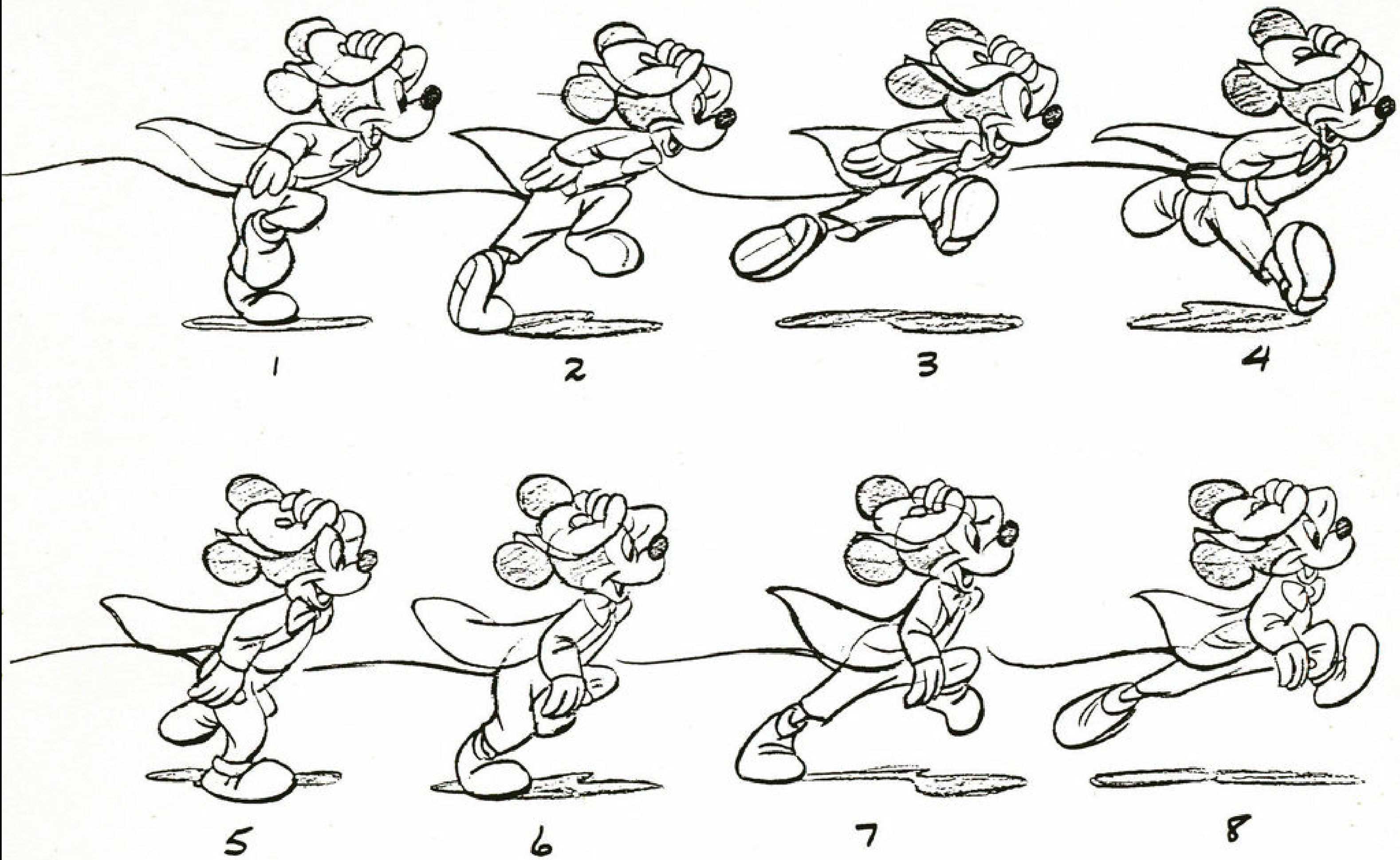



```
Car c1 := new Car ( );
Car c2 := new Car ( );
c1.refuel ( 3 );
c2.refuel ( 5 );
assert ( c1.fuel == 3 );
```

The Problem is

Framing

MICKEY MOUSE / *run cycle*



(IN THIS ACTION — DRAWING NO 1 WOULD FOLLOW NO 8)

"We wish to ***logically represent how*** the execution of ***a command changes the state*** ***without having to explicitly say how the command does not change the state***."

One Solution is

Implicit Dynamic Frames *(IDF)*

Theory: **Kassios (2006), Smans (2008)**

Implementation: **ETH, Microsoft Research, Leuven**

Chalice

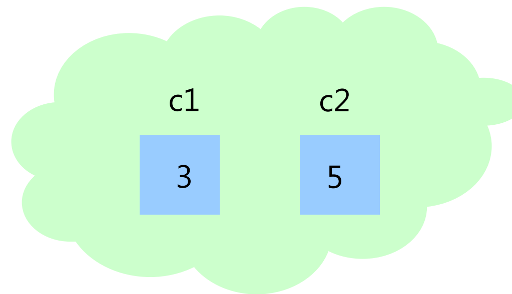
Car program in Chalice

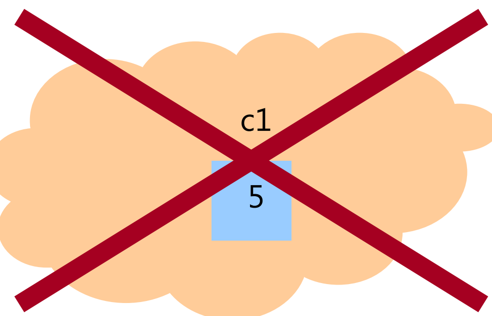
```
class Car
{
   var fuel : int;

   void refuel (amount :  int)
   requires acc (this.fuel );
   ensures acc (this.fuel ) &*& this.fuel == amount;
   {
      this.fuel := amount;
   }

   void main ( )
   {
      ...
   }
}
```

```
var c1 := new Car;
var c2 := new Car;
call c1.refuel ( 3 );
call c2.refuel ( 5 );
assert  c1.fuel == 3;
```

Car program in Chalice

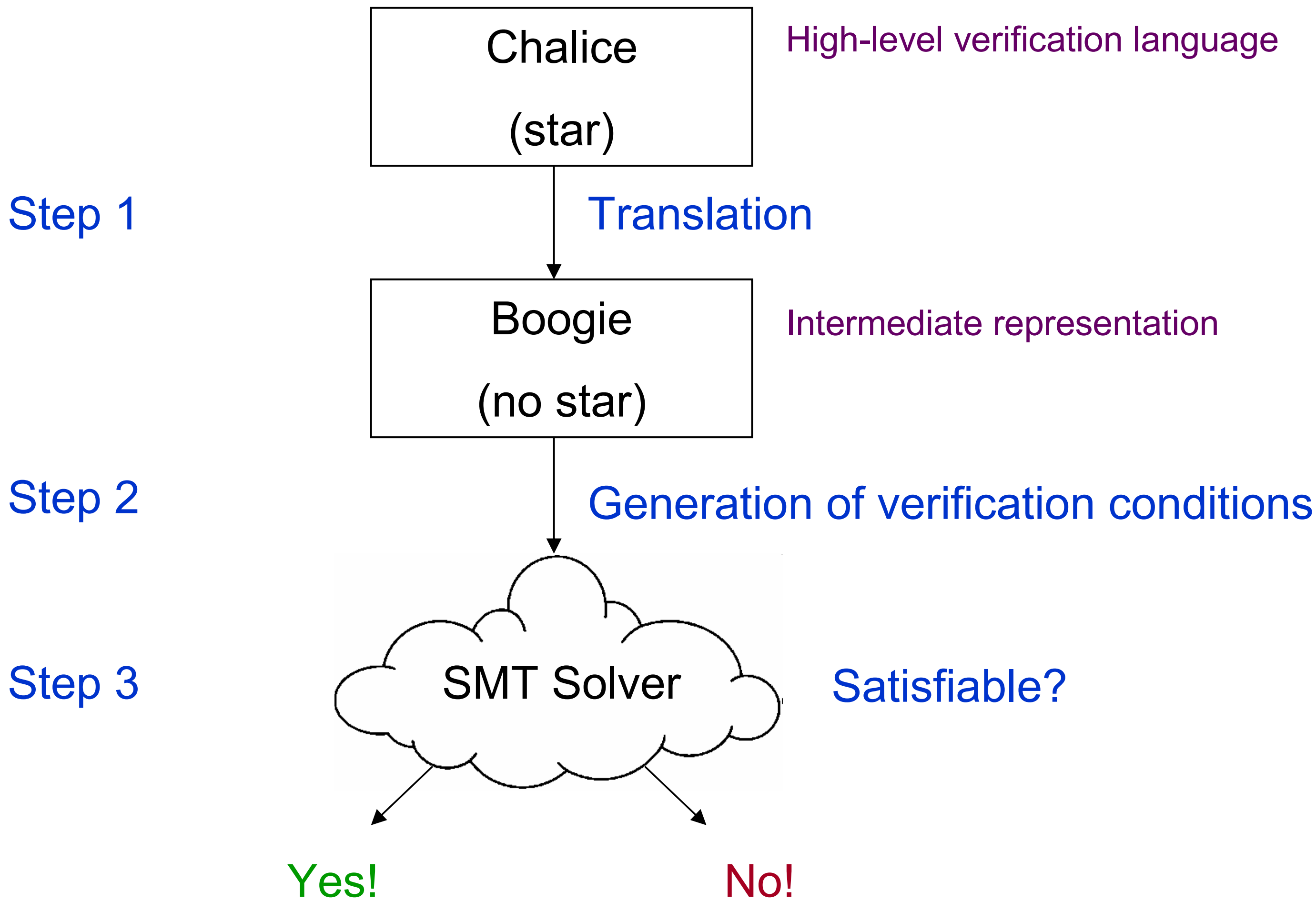
```
class Car
{
   var fuel : int;

   void refuel (amount :  int)
   requires acc (this.fuel );
   ensures acc (this.fuel ) &*& this.fuel == amount;
   {
      this.fuel := amount;
   }

   void main ( )
   {
      ...
   }
}
```



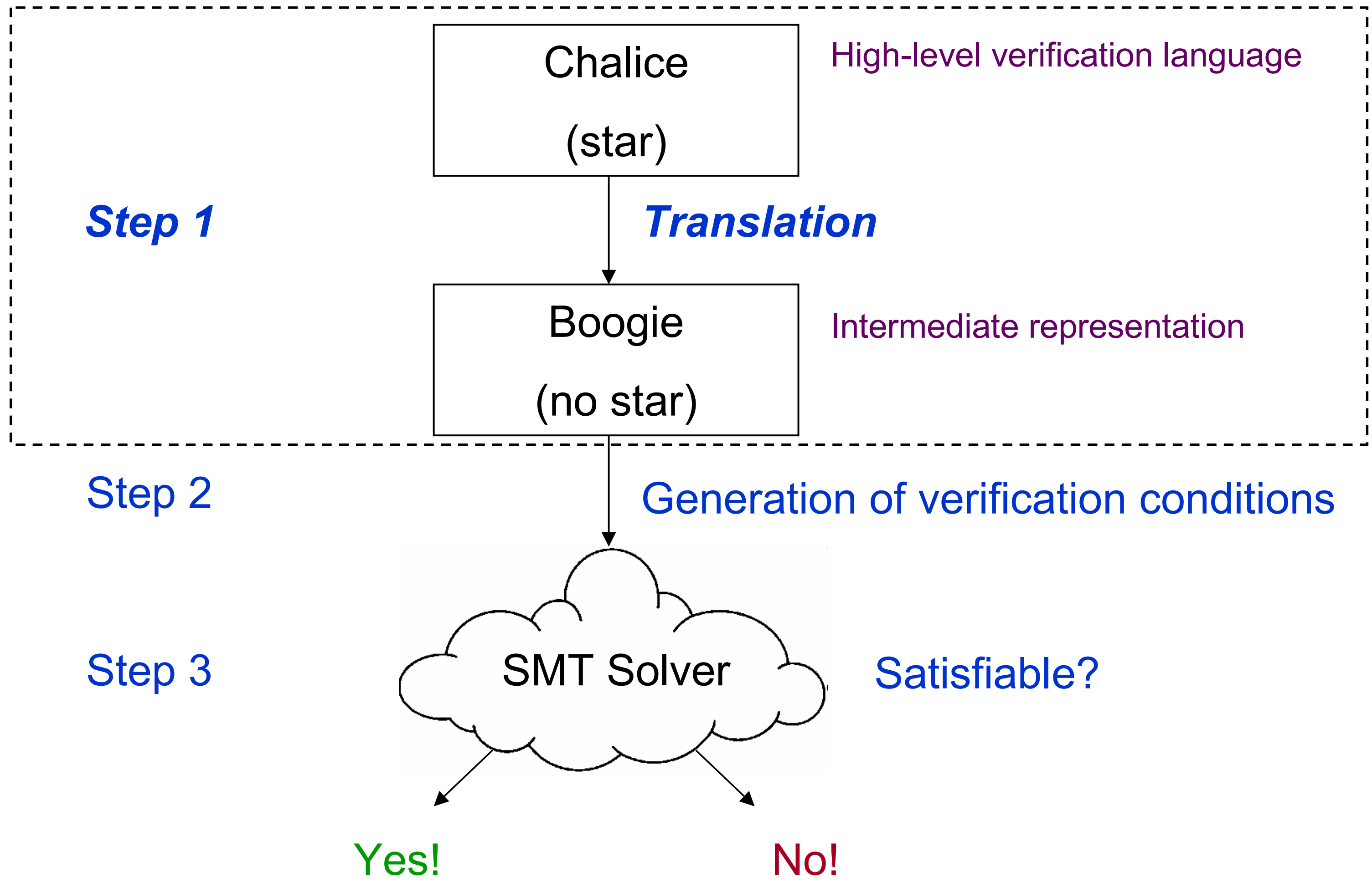



```
var c1 := new Car;
var c2 := new Car;
call c1.refuel ( 3 );
call c2.refuel ( 5 );
assert  c1.fuel == 3;
```

Verification Pipeline



Verification Pipeline



Contributions

1. Formalization of a Chalice subset
2. Formalization of a Boogie subset
3. Formalization of a subset translation from Chalice to Boogie
4. Proof of Soundness of Translation

Contributions

1. Formalization of a Chalice subset
2. Formalization of a Boogie subset
3. Formalization of a subset translation from Chalice to Boogie
4. Proof of Soundness of Translation

Chalice
(has star)

Our subset ignores concurrency features

We give:

- ***Operational semantics***
- ***Hoare logic*** { Pre } C { Post }
- ***Soundness proof of Hoare logic w.r.t. operational semantics***

"A Sip of the Chalice" (Drossopoulou & Raad, 2011)

Self-framing

Badly framed Car program in Chalice (1)

```
class Car
{
    var fuel : int;

    void refuel ( amount : int )
    requires true;
    ensures this.fuel == amount;
    {
        this.fuel := amount;  //ERROR: update is not framed
    }

    void main ( )
    {
        ...
    }
}
```

Badly framed Car program in Chalice (2)

```
class Car
{
    var fuel : int;

    void refuel ( amount : int )
    requires this.fuel == 0;  //ERROR: assertion not self-framing
    ensures acc ( this.fuel ) &*& this.fuel == amount;
    {
        this.fuel := amount;
    }

    void main ( )
    {
        ...
    }
}
```

Our approach to self-framing

An assertion *A* is ***self-framing***

if and only if

All heap references in *A* are sufficiently framed by the access predicate.

$$\mathit{A} = \underbrace{\text{acc (x.f)}}_{\text{Rights}} \;\&{*}\&\; \ldots \;\&{*}\&\; \underbrace{\text{x.f == 100}}_{\text{Access}}$$

A is self-framing

if and only if

Access (*A*) $\subseteq$ Rights (*A*)

To be or not to be self-framing

A	Access (*A*)	Rights (*A*)	
acc(x.f) &*& x.f == 100	{ x.f }	{ x.f }	✓
acc(x.f) &*& y.f == 90	{ y.f }	{ x.f }	✗
acc(x.f) &*& x.g == 101	{ x.g }	{ x.f }	✗
acc(x.f)	{ }	{ x.f }	✓

Our approach to self-framing

is intuitive,

provides an operational angle,

and simplifies proof of soundness.

Challenges

- Scoping & Simplification
- Design choices
- Formalization of method calls

Contributions

1. Formalization of a Chalice subset
2. Formalization of a Boogie subset
3. Formalization of a subset translation from Chalice to Boogie
4. Proof of Soundness of Translation

Boogie
(no star)

Our Boogie subset mirrors our Chalice subset

We give an operational semantics for our Boogie subset

Motivated in Rustan Leino's talk at Imperial College London (2012)

$$\frac{\varphi(mask)(r, C.f) = v}{[\![\texttt{CanAccess}(mask, r, f)]\!]_\varphi = v}$$

$$\frac{\forall r : ObjectReference, f : FieldId.(\varphi(mask)(r, C.f) \in \{0, 1\})}{[\![\texttt{IsGoodMask}(mask)]\!]_\varphi = 1}$$

$$\frac{\begin{array}{l}\forall r : ObjectReference, f : FieldId.(\texttt{CanAccess}(mask, r, f) \\ \Rightarrow \varphi(h_i)(r, C.f) = \varphi(h)(r, C.f))\end{array}}{[\![\texttt{IsGoodInhaleState}(h_i, h, mask)]\!]_\varphi = 1}$$

$$\frac{}{[\![mask[r, C.f]]\!]_\varphi = \varphi(mask)(r, C.f)} \qquad \frac{}{[\![h[r, C.f]]\!]_\varphi = \varphi(h)(r, C.f)}$$

$$\frac{\varphi \models B}{\texttt{assume}\ \ B, \varphi \rightsquigarrow \varphi} \qquad \frac{\varphi \models B}{\texttt{assert}\ \ B, \varphi \rightsquigarrow \varphi}$$

$$\frac{}{\texttt{havoc}(x), \varphi \rightsquigarrow \varphi[x \mapsto v]} \qquad \frac{\varphi \not\models B}{\texttt{assert}\ \ B, \varphi \rightsquigarrow ERROR}$$

$$\frac{\varphi(mask)(r, C.f) = v}{[\![\texttt{CanAccess}(mask, r, f)]\!]_\varphi = v}$$

$$\frac{\forall r : ObjectReference, f : FieldId.(\varphi(mask)(r, C.f) \in \{0, 1\})}{[\![\texttt{IsGoodMask}(mask)]\!]_\varphi = 1}$$

$$\frac{\begin{array}{l}\forall r : ObjectReference, f : FieldId.(\texttt{CanAccess}(mask, r, f) \\ \Rightarrow \varphi(h_i)(r, C.f) = \varphi(h)(r, C.f))\end{array}}{[\![\texttt{IsGoodInhaleState}(h_i, h, mask)]\!]_\varphi = 1}$$

$$\frac{}{[\![mask[r, C.f]]\!]_\varphi = \varphi(mask)(r, C.f)} \qquad \frac{}{[\![h[r, C.f]]\!]_\varphi = \varphi(h)(r, C.f)}$$

$$\frac{\varphi \models B}{\texttt{assume}\ \ B, \varphi \rightsquigarrow \varphi} \qquad \frac{\varphi \models B}{\texttt{assert}\ \ B, \varphi \rightsquigarrow \varphi}$$

$$\frac{}{\texttt{havoc}(x), \varphi \rightsquigarrow \varphi[x \mapsto v]} \qquad \frac{\varphi \not\models B}{\texttt{assert}\ \ B, \varphi \rightsquigarrow ERROR}$$

Challenges

- Scoping, Simplification, Design
- Formalization of Boogie-specific commands

Without the benefit of existing literature!

Contributions

1. Formalization of a Chalice subset
2. Formalization of a Boogie subset
3. Formalization of a subset translation from Chalice to Boogie
4. Proof of Soundness of Translation

Boogie

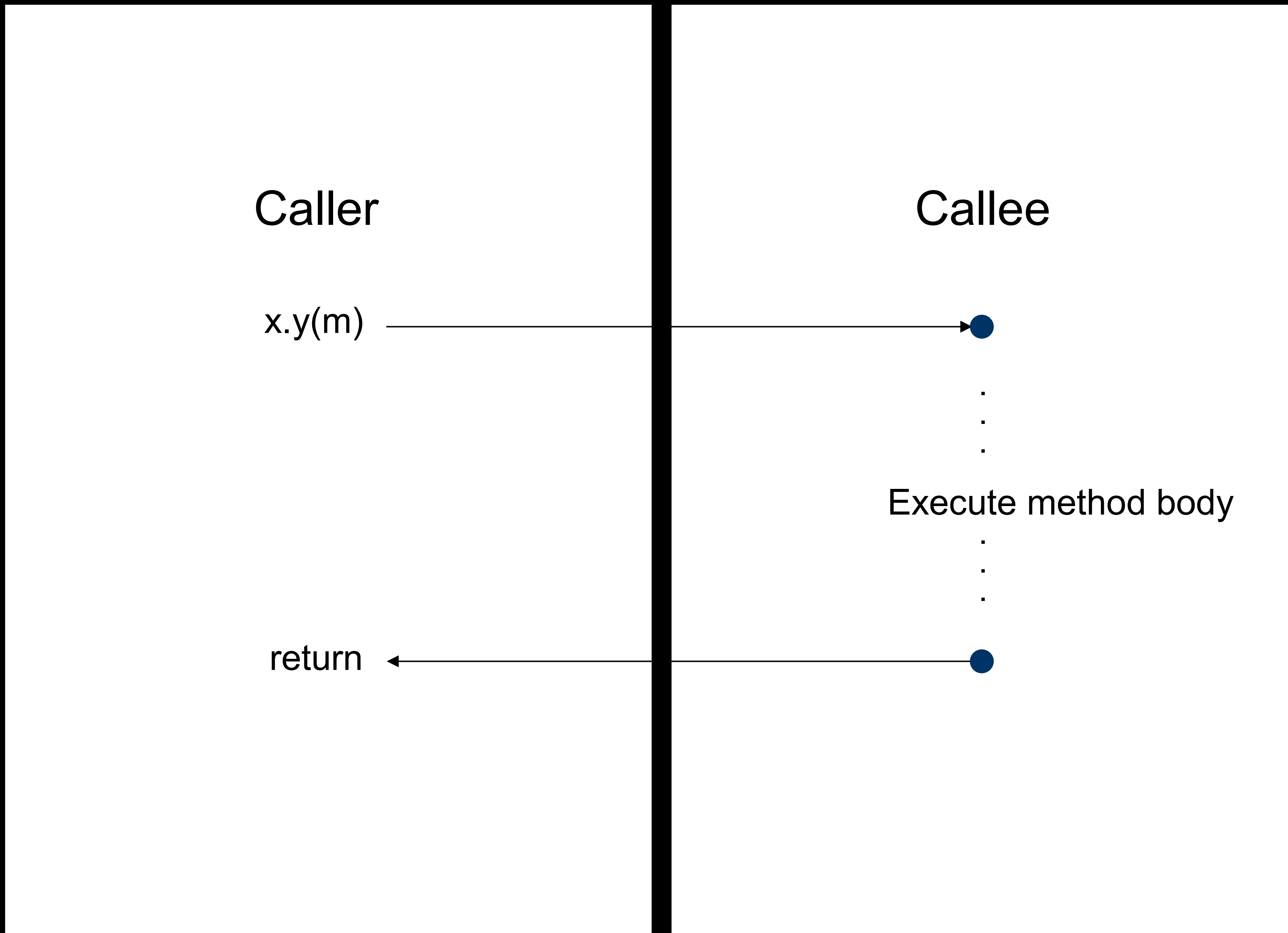
Has special commands and predicates

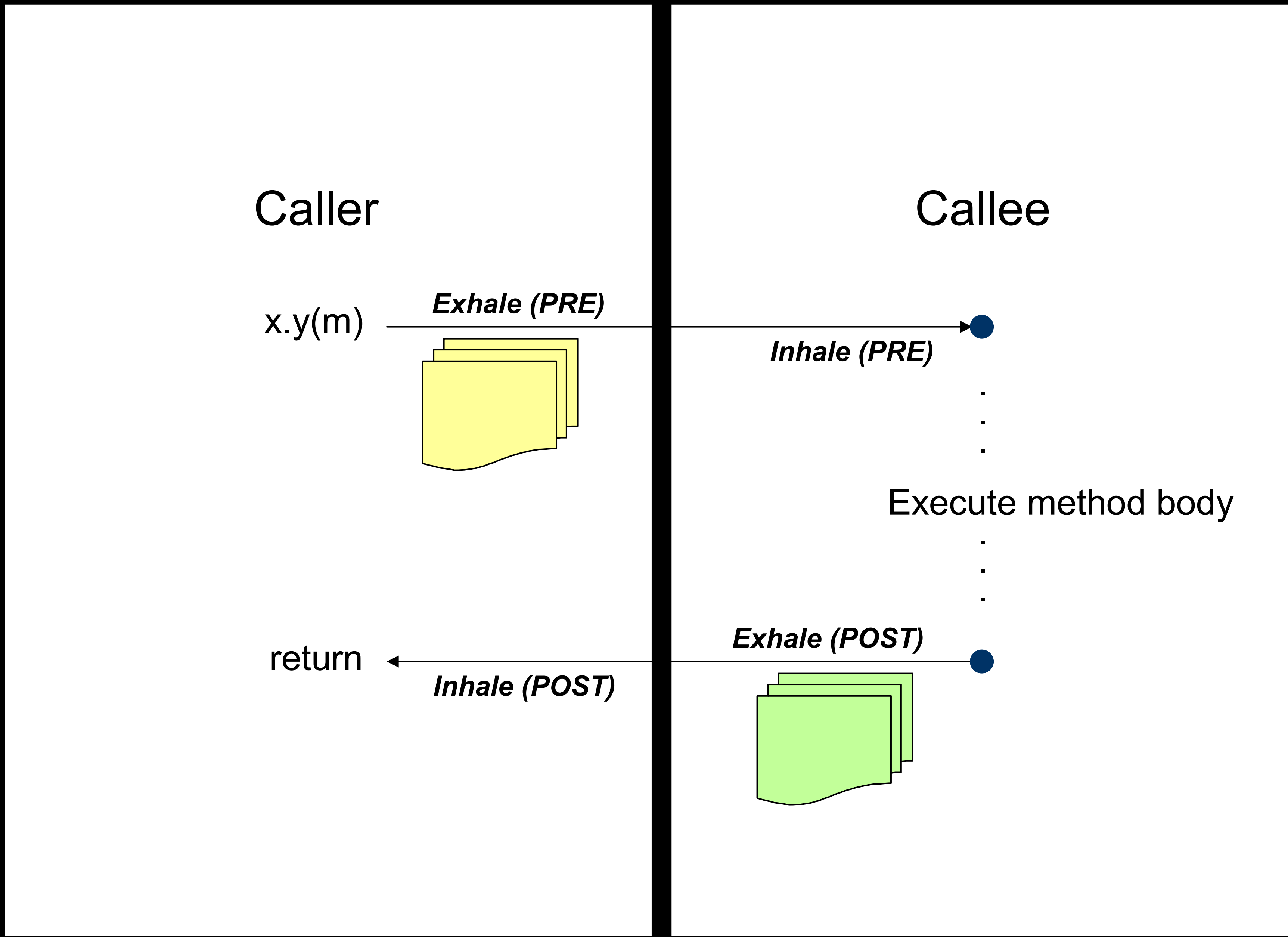
< $frame_B$ >

Chalice	to	Boogie
Has notion of star operator	$\Leftrightarrow$	Has special commands and predicates
$< \text{mask}_C , \text{heap}_C , \text{frame}_C >$	$\Leftrightarrow$	$< \text{frame}_B >$

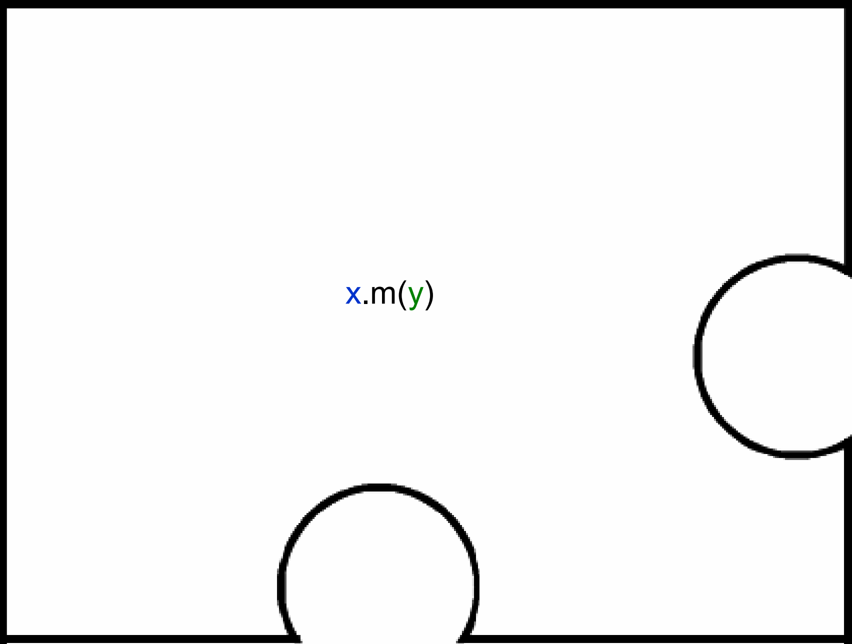
Translation needs to be meaning-preserving

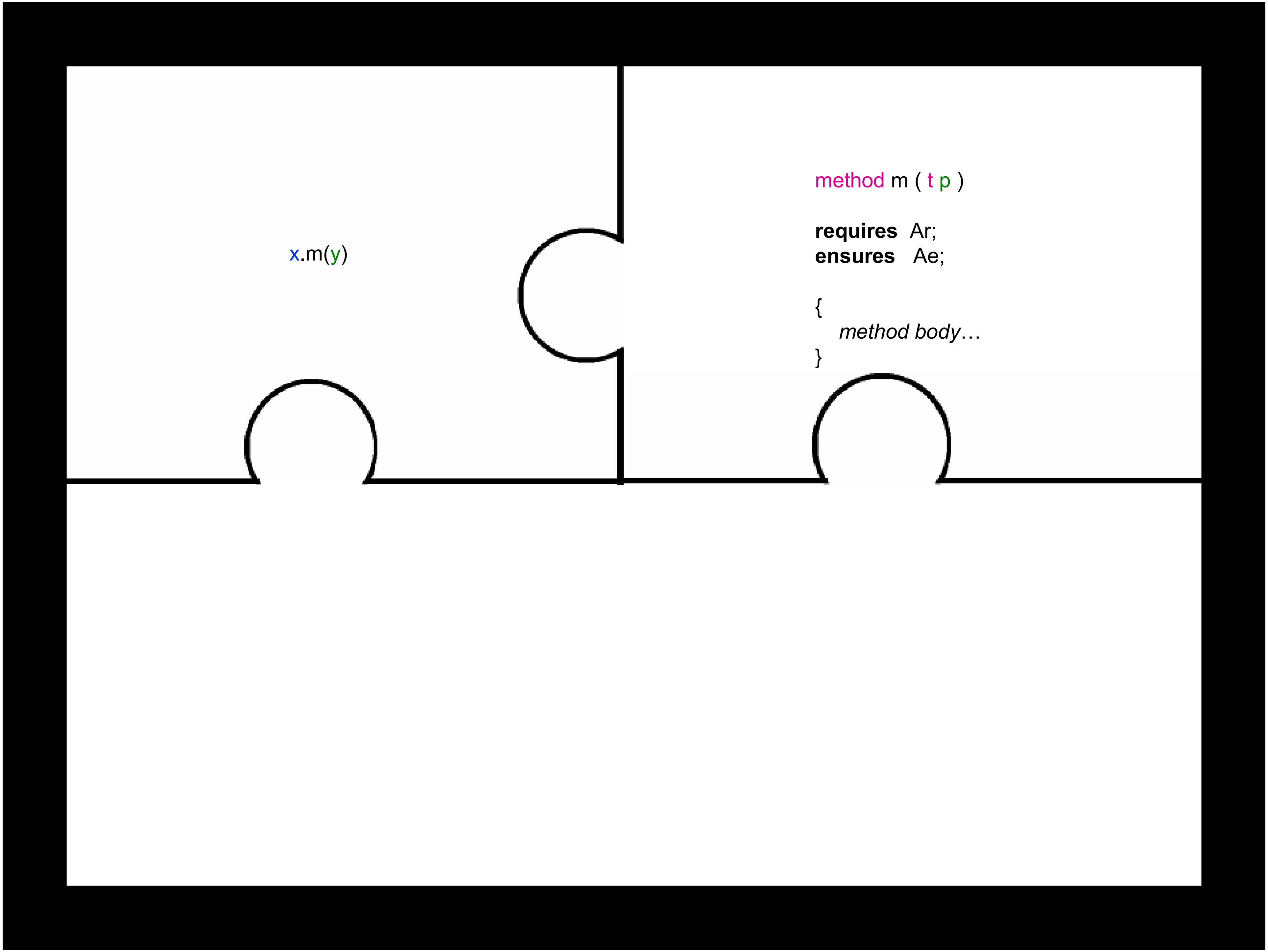
Method Call


Caller
Callee
x.y(m)
Execute method body
return



Caller
Callee
x.y(m)
Exhale (PRE)
Inhale (PRE)
Execute method body
Exhale (POST)
return
Inhale (POST)


x.m(y)


x.m(y)
method m (t p)
requires Ar;
ensures Ae;
{
method body…
}

x.m(y)

```
method m ( t p )

requires  Ar;
ensures   Ae;

{
    method body…
}
```

```
var mask_e : Mask; var heap_i : Heap;
var y1 : Expression;
y1 := y;

mask_e := mask;
exhale( Ar [ this / x ] [ y1 / y ] , mask_e );
mask := mask_e;

havoc heap_i;
assume IsGoodInhaleState( heap_i , heap , mask );
inhale( Ae [ this / x ] [ y1 / y ] , mask , heap_i );
```

```
x.m(y)
```

```
method m ( t p )

requires  Ar;
ensures   Ae;

{
    method body…
}
```

```
var mask_e : Mask; var heap_i : Heap;
var y1 : Expression;
y1 := y;

mask_e := mask;
exhale( Ar [ this / x ] [ y1 / y ] , mask_e );
mask := mask_e;

havoc heap_i;
assume IsGoodInhaleState( heap_i , heap , mask );
inhale( Ae [ this / x ] [ y1 / y ] , mask , heap_i );
```

```
procedure C.m ( this : C , p : t )
{
 var mask_e : Mask;    var heap_i : Heap;

 havoc heap_i;
 assume IsGoodInhaleState ( heap_i , heap , mask );
 inhale ( Ar , mask , heap_i );

 translate ( method body );

 mask_e := mask;
 exhale ( Ae , mask_e );
 mask := mask_e;
}
```

```
exhale( acc( x.f ) , mask ) = assert 0 < mask[ x , C.f ];
                              mask[x , C.f] := mask[ x , C.f ] – 1;
                              assume IsGoodMask ( mask );

exhale( a1 &*& a2 , mask ) = exhale( a1 , mask );
                             exhale( a2 , mask );

exhale( b , mask ) = assert translate ( b );
```

```
inhale( b , mask , heap_i ) = assume translate ( b );

inhale( a1 &*& a2 , mask , heap_i ) = inhale( a1 , mask , heap_i );
                                      inhale( a2 , mask , heap_i );

inhale( acc(x.f) , mask , heap_i ) = heap[ x , C.f ] := heap_i[ x , C.f ];
                                     mask[ x , C.f ] := mask[ x , C.f ] + 1;
                                     assume IsGoodMask ( mask );
```

```
exhale( acc( x.f ) , mask ) = assert 0 < mask[ x , C.f ];
                              mask[x , C.f] := mask[ x , C.f ] – 1;
                              assume IsGoodMask ( mask );

exhale( a1 &*& a2 , mask ) = exhale( a1 , mask );
                              exhale( a2 , mask );

exhale( b , mask ) = assert translate ( b );
```

```
inhale( b , mask , heap_i ) = assume translate ( b );

inhale( a1 &*& a2 , mask , heap_i ) = inhale( a1 , mask , heap_i );
                                      inhale( a2 , mask , heap_i );

inhale( acc(x.f) , mask , heap_i ) = heap[ x , C.f ] := heap_i[ x , C.f ];
                                     mask[ x , C.f ] := mask[ x , C.f ] + 1;
                                     assume IsGoodMask ( mask );
```

Checking self-framedness of assertions in Boogie (1)

Not self-framing:
A = x.f == 3 &*& y.f == 3;

exhale(x.f == 3 &*& y.f == 3, mask) =

assert heap[x , C.f] == 3; //ERROR

assert heap[y , C.f] == 3;

Checking self-framedness of assertions in Boogie (2)

Self-framing:
A = acc(x.f) &*& acc(y.f) &*& x.f == 3 &*& y.f == 3;   ***where x and y not aliases of each other!***

exhale(acc(x.f) &*& acc(y.f) &*& x.f == 3 &*& y.f == 3, mask) =

```
assert 0 < mask[ x , C.f ];
mask[ x , C.f ] := mask[ x , C.f ] – 1;
assume IsGoodMask ( mask );

assert 0 < mask[ y , C.f ];
mask[ y , C.f ] := mask[ y , C.f ] – 1;
assume IsGoodMask ( mask );

assert heap[ x , C.f ] == 3;

assert heap[ y , C.f ] == 3;
```

Car program still running ...

```
class Car
{
   var fuel : int;

   void refuel ( amount : int )
   requires acc ( this.fuel );
   ensures acc ( this.fuel ) &*& this.fuel == amount;
   {
      this.fuel := amount;
   }

   void main ( )
   {
      var c1 := new Car;
      var c2 := new Car;

      call c1.refuel ( 3 );

      call c2.refuel ( 5 );
      assert ( c1.fuel == 3 );
   }
}
```

```
call c1.refuel ( 3 );
```

```
//set up variables
var mask_e : Mask;    var heap_i : Heap;
var y : Expression;
y := 3;

mask_e := mask;

//exhale precondition
assert 0 < mask_e [ this , Car.fuel ];
mask_e [ this , Car.fuel ] := mask_e [ this , Car.fuel ] – 1;
assume IsGoodMask ( mask_e );

mask := mask_e;

havoc heap_i;

assume IsGoodInhaleState ( heap_i , heap, mask );

//inhale postcondition
heap [ this , Car.fuel ] := heap_i [ this , Car.fuel ];
mask [ this , Car.fuel ] := mask [ this , Car.fuel ] + 1;
assume IsGoodMask ( mask );
assume heap [ this , Car.fuel ] == y;
```

```
void refuel ( amount : int )
  requires acc ( this.fuel );
  ensures acc ( this.fuel ) &*& this.fuel == amount;
  {
    this.fuel := amount;
  }
```

```
procedure Car.refuel ( this : Car , amount : int )
{
 var mask_e : Mask;    var heap_i : Heap;

 havoc heap_i;
 assume IsGoodInhaleState ( heap_i , heap , mask );

 //inhale precondition
 heap [ this , Car.fuel ] := heap_i [ this , Car.fuel ];
 mask [ this , Car.fuel ] := mask [ this , Car.fuel ] + 1;
 assume IsGoodMask ( mask );

 //translate method body
 CanAccess ( mask , this , Car.fuel );
 heap [ this , Car.fuel ] := amount;

 mask_e := mask;

 //exhale postcondition
 assert 0 < mask_e [ this , Car.fuel ];
 mask_e [ this , Car.fuel ] := mask_e [ this , Car.fuel ] – 1;
 assume IsGoodMask ( mask_e );
 assert heap [ this , Car.fuel ] == amount;
 mask := mask_e;
}
```

Contributions

1. Formalization of a Chalice subset
2. Formalization of a Boogie subset
3. Formalization of a subset translation from Chalice to Boogie
4. Proof of Soundness of Translation

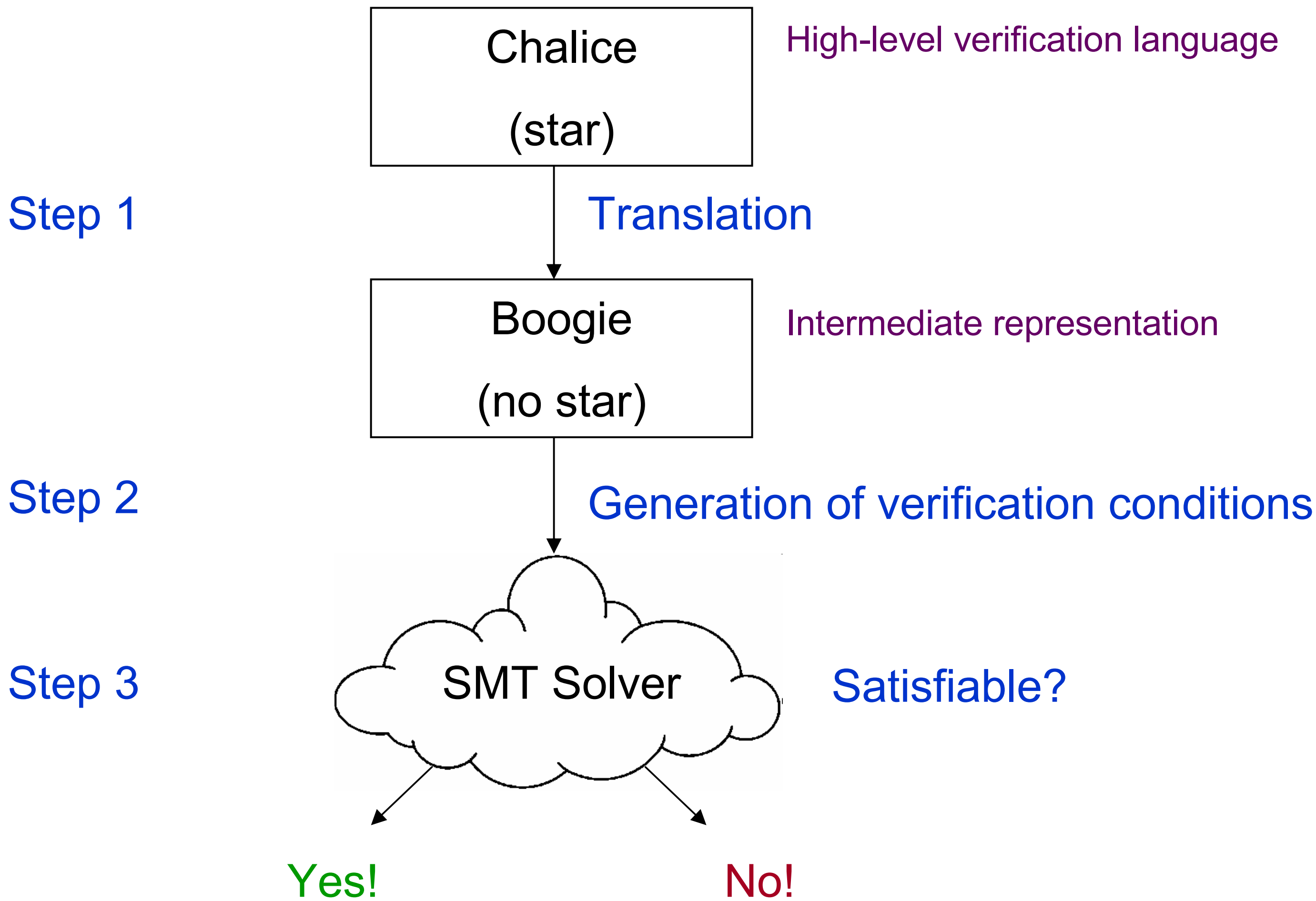
Soundness Argument

The translation of a Chalice program ***P*** is sound

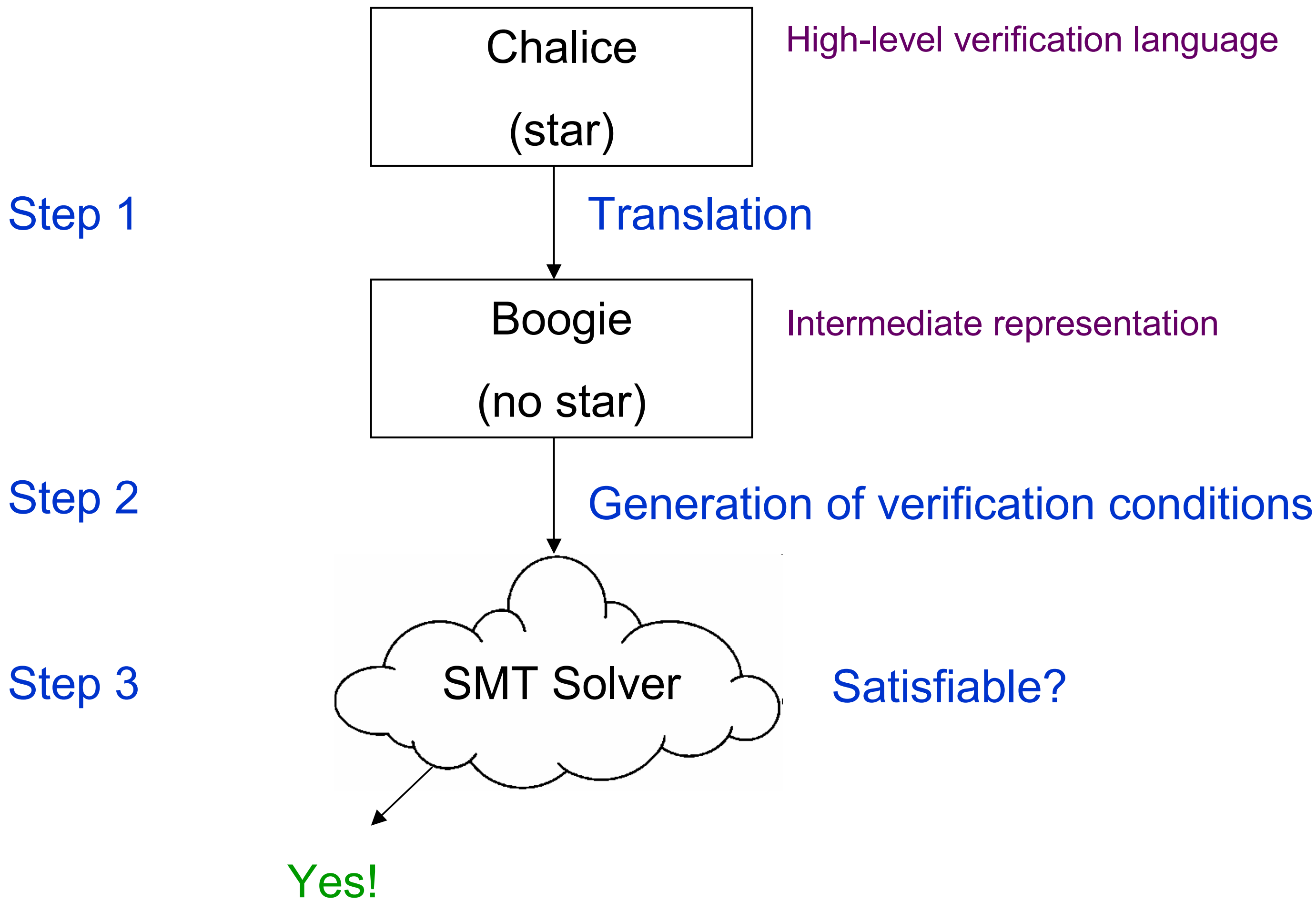
if and only if

given that ***P*** verifies in the Boogie environment, it also verifies in the Chalice environment.

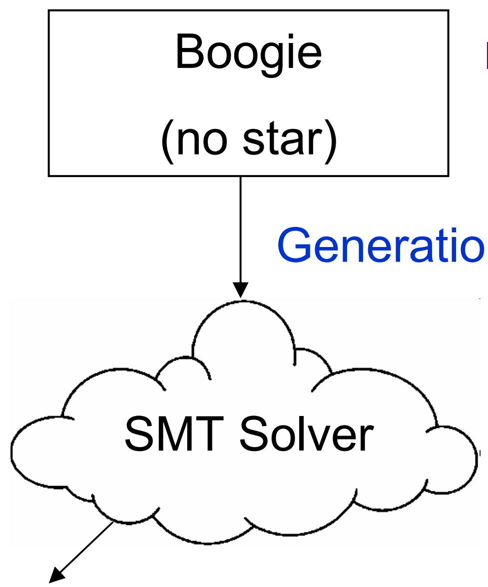
Verification Pipeline Revisited



If program verifies in Chalice environment …



If program verifies in Boogie environment …



Intermediate representation

Step 2

Generation of verification conditions

Step 3

Satisfiable?

Yes!

Lemma 6.4.4. $\forall C : ClassId, m : MethId :$

If:

1. $Prog_C(C, m) = \texttt{requires}\ \ A; \texttt{ensures}\ \ A'; \{C_C\} \wedge$
2. $(inhale(A, mask, h_i); translate(C_C); exhale(A, mask_e)), \varphi_\varepsilon \not\rightarrow ABORT \wedge$
3. $\Pi, \varphi_C, h_C \models A \wedge$
4. $C_C, \Pi, \varphi_C, h_C \rightsquigarrow \Pi', \varphi'_C, h'_C$

Then $\Pi', \varphi'_C, h'_C \models A'$

Other Machinery

Auxiliary Definitions, Lemmas …

Challenges

- Design of translation function
- Formulating & justifying soundness argument
- Lemmas and proofs

Highlights

1. Formalization of a Chalice subset
   - Approach to self-framing

2. Formalization of a Boogie subset
   - Operational semantics

3. Formalization of a sound translation from Chalice to Boogie
   - Translation function, soundness argument & proofs

Applications

1. Boogie-based verification

2. Pedagogic uses

Applications

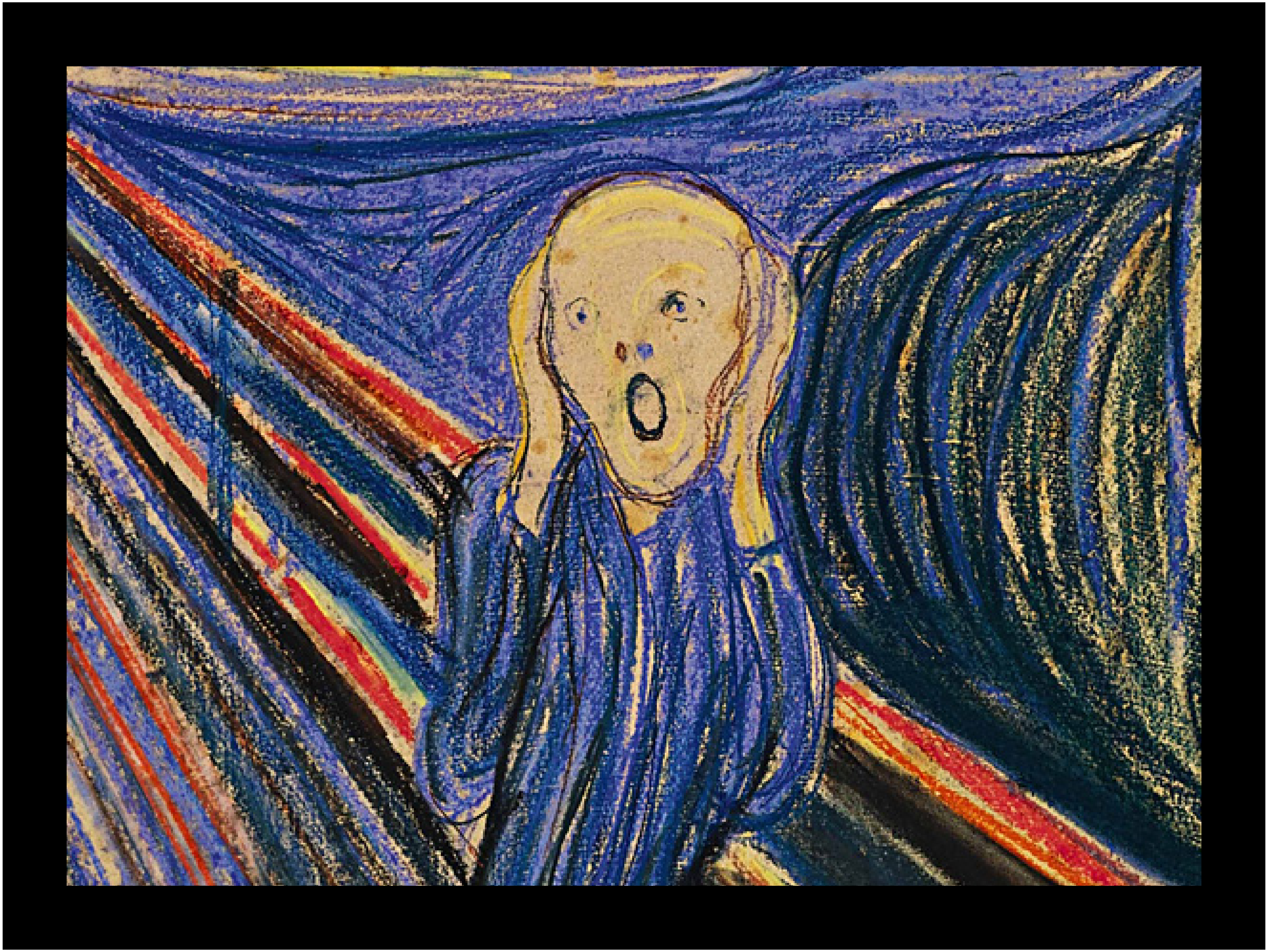
1. Boogie-based verification

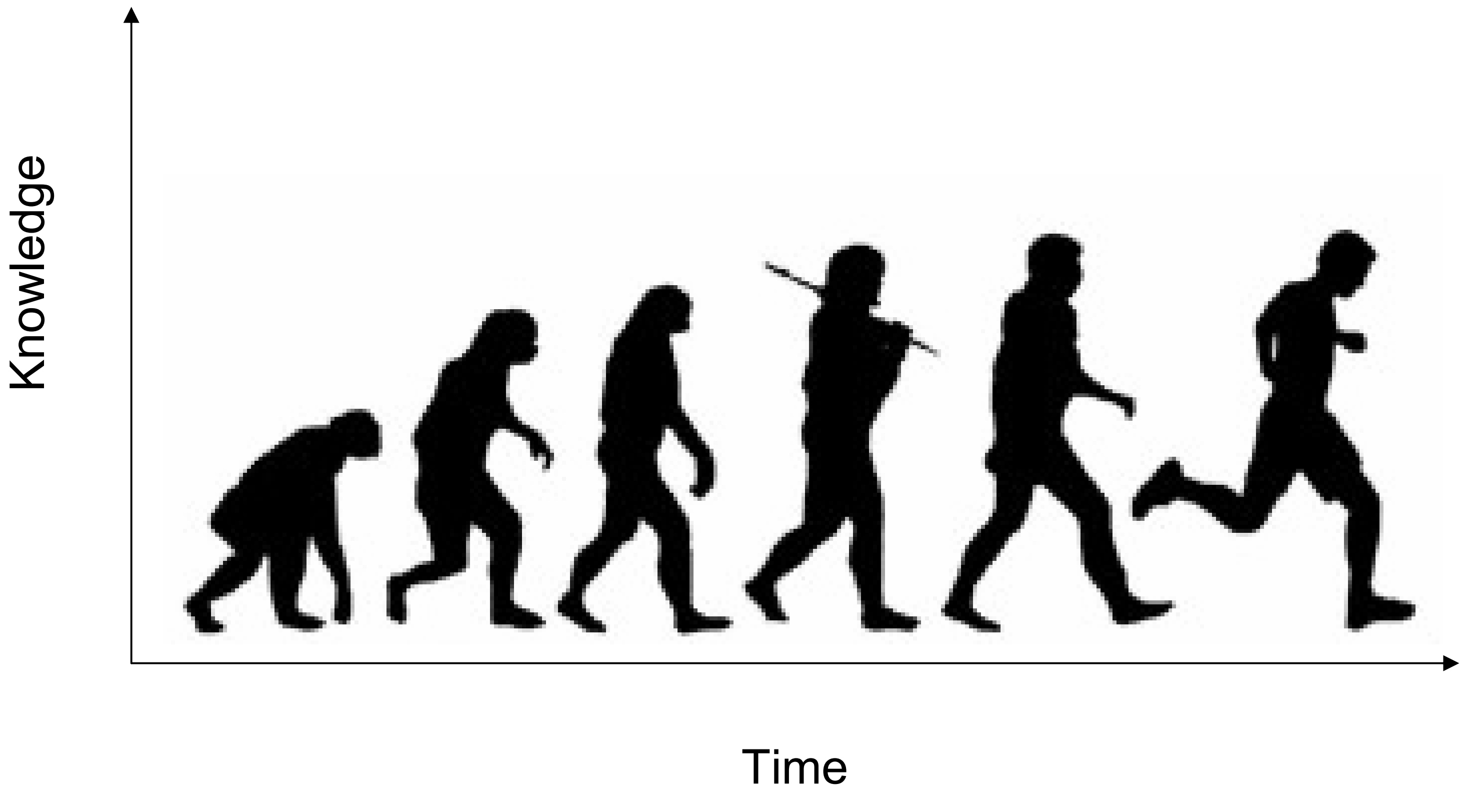
2. Pedagogic uses

http://rise4fun.com/

A Challenging Project

- Lots of background reading
- An open-ended project
- Balancing breadth and depth of investigation
- Experimenting with tools
- Formalizing approaches and arguments
- Making original contributions
- I am not JMC or MEng (but I am passionate about research!)




Knowledge
Time

Future Work

1. **Formalize** translation of while loop, and proof of method call
2. **Extend** language subsets to include concurrency
3. **Consider** translation from VeriFast to Chalice

Q&A

Extra Slides

Usual approach to self-framing

An assertion *A* is ***self-framing***

if and only if

The validity of *A* is preserved in all heaps which agree in the locations mentioned in the permissions. *(Parkinson & Summers, 2011)*

While loop

```
while ( condition )
{

    // check invariant holds upon loop entry (assert …)

    loop body …

    // check invariant holds after arbitrary loop iteration (havoc…)

}
```

Method call

If:

- *x.m(y)*
- translation of *x.m(y)* gives a Boogie encoding C_B
- C_B verifies in Boogie
- Chalice and Boogie starting configurations are congruent
- and given preconditions of operational semantics for *x.m(y)*…

Then:

(esp. using Lemmas for Inhale / Exhale …)

Show that there exists a terminal Boogie configuration φ_B s. t.

- C_B execution in Boogie leads to φ_B
- Terminal configurations in Chalice and Boogie match

Lemma 6.4.1. *Given* $C : Command_C, \forall \Pi : Mask_C, \varphi_C : Store_C, h_C : Heap_C, \varphi_B : Store_B,$

If:

1. $translate(C), \varphi_B \not\rightsquigarrow ABORT$
2. $\Pi, \varphi_C, h_C \cong \varphi_B$
3. $C, \Pi, \varphi_C, h_C \rightsquigarrow \Pi', \varphi'_C, h'_C$

Then there exists a $\varphi'_B : Store_B$ *such that:*

4. $translate(C), \varphi_B \rightsquigarrow \varphi'_B$
5. $\Pi', \varphi'_C, h'_C \cong \varphi'_B$

Lemma 6.4.2. *Given $A : Assertion_C, mask : Mask_B, h_B : Heap$ (where A is self-framing, $mask$ is the current mask, and h_B is the current heap),*

If:

1. $\texttt{inhale}(A, mask, h_B) = C_i$ *where* $C_i : Command_B$
2. $C_i, \varphi_B \rightsquigarrow \varphi'_B$ *where* $\varphi'_B : Store_B$ *such that* $\varphi'_B \neq ABORT$
3. $\Pi', \varphi'_c, h'_c \cong \varphi'_B$, *where* $\Pi' : Mask_C, \varphi'_C : Store_C, h'_c : Heap_C$

Then:

4. $\Pi', \varphi'_c, h'_c \models A$
5. *There exists a* $mask' : Mask_B$ *such that* $\varphi'_B \equiv \varphi_B[mask \mapsto mask']$, *and* $mask \leq_\Pi mask'$.

Lemma 6.4.3. *Given* $A : Assertion_C, mask : Mask_B$, *(where* A *is self-framing, and* $mask$ *is the current mask)*

If:

1. $\texttt{exhale}(A, mask) = C_e$ *where* $C_e : Command_B$
2. $C_e, \varphi_B \rightsquigarrow \varphi'_B$ *where* $\varphi'_B : Store_B$ *such that* $\varphi'_B \neq ABORT$
3. $\Pi, \varphi_c, h_c \cong \varphi_B$

Then:

4. $\Pi, \varphi_c, h_c \models A$, *where* $\Pi : Mask_C, \varphi_C : Store_C, h_c : Heap_C$
5. *There exists a* $mask' : Mask_B$ *such that* $\varphi'_B \equiv \varphi_B[mask \mapsto mask']$, *and* $mask' \leq_\Pi mask$.

Definition 6.3.1. *combine* $: Mask_C \times Store_C \times Heap_C \rightarrow Store_B$ *such that, for* $\Pi : Mask_C, \varphi_C : Store_C, h_C : Heap_C$ *we:*

1. *Create a fresh Boogie variable* $mask_B : Mask_B$ *and populate it with the access permissions found in* Π*, such that* $\forall v : \{0, 1\}, r : ObjectReference, f : FieldId.(\Pi[r, f] = v \Rightarrow mask[(r, C.f) \mapsto v])$ *where* f *is a field of class* C*.*

2. *Create a fresh Boogie variable* $h_B : Heap_B$ *and populate it with the values found in* h_C*, such that* $\forall r : ObjectReference, f : FieldId.(h_B[(r, C.f) \mapsto [\![r.f]\!]_{\varphi, h_C}])$ *where* f *is a field of class* C*.*

3. *Now take an empty store* $\varphi_\varepsilon : Store_B$*, such that* $\varphi_\varepsilon = \{mask \mapsto \emptyset, heap \mapsto \emptyset\}$*.*

4. *Construct a* $\varphi_B : Store_B$*, such that* $\varphi_B = \varphi_\varepsilon[mask \mapsto mask_B, h \mapsto h_B] \cup \varphi_C$

Definition 6.3.2. $\cong : Mask_C \times Store_C \times Heap_C \times Store_B$ *such that, for* $\Pi : Mask_C, \varphi_C : Store_C, h_C : Heap_C, \varphi_B : Store_B$*, we have*

$$\Pi, \varphi_C, h_C \cong \varphi_B \iff combine(\Pi, \varphi_C, h_C) = \varphi_B$$

$$Program : ClassId \rightharpoonup FieldId \times (MethId \times MethDef)$$

$$B \in Boolean ::= \texttt{true} \mid \texttt{false} \mid \neg B \mid E == E$$

$$E \in Expression ::= n \mid x \mid x.f \mid E + E \mid x.m_{pure}(E)$$

$$\begin{aligned} C \in Command ::= {} & x := y.f \mid x.f := y \mid \texttt{if } B \texttt{ then } C \texttt{ else } C \mid \\ & \texttt{while } B \texttt{ do } C \mid C;C \mid \texttt{skip} \mid x := y.m(E) \mid \\ & x := \texttt{new } C \mid \texttt{assert } A \end{aligned}$$

$$A \in Assertion ::= B \mid \texttt{acc}(x.f) \mid A * A$$

$$ClassID, FieldID ::= (a - zA - Z)+$$

$$Term ::= Boolean | Expression | Command | Assertion$$

Figure 3.1: **Our Subset of the full Chalice Syntax**

$$[\![\cdot]\!] : (Assertion \cup Expression) \times Mask \times Store \times Heap \rightharpoonup \mathbb{Z}$$

$$\rightsquigarrow : Command \times Mask \times Store \times Heap \rightharpoonup Mask \times Store \times Heap$$

$$\Pi \in Mask : ObjectReference \times (ClassId \times (FieldId \rightharpoonup \{0, 1\}))$$

$$\varphi \in Store : Variable \rightharpoonup \mathbb{Z}$$

$$h \in Heap : ObjectReference \rightharpoonup (ClassId \times (FieldId \rightharpoonup \mathbb{Z}))$$

$$ObjectReference ::= \mathbb{Z}^+$$

$$Variable, ClassID, FieldID ::= (a - zA - Z)+$$

Figure 3.2: **Runtime Configuration for Our Chalice Subset**

$$
\begin{aligned}
B \in \mathit{Boolean} ::= {} & \texttt{true} \mid \texttt{false} \mid \neg B \mid E == E \mid \\
& \texttt{CanAccess(}\mathit{mask}\texttt{,}\, r\texttt{,}\, f\texttt{)} \mid \\
& \texttt{IsGoodInhaleState(}h\texttt{,}\, h\texttt{,}\, \mathit{mask}\texttt{)} \mid \\
& \texttt{IsGoodMask(}\mathit{mask}\texttt{)}
\end{aligned}
$$

$$
E \in \mathit{Expression} ::= n \mid x \mid h[r, C.f] \mid \mathit{mask}[r, C.f] \mid E + E
$$

$$
\begin{aligned}
C \in \mathit{Command} ::= {} & \mathit{var}\ x\ :\ t \mid x := E \mid h[r, C.f] := E \mid \mathit{mask}[r, C.f] := E \mid \\
& \texttt{havoc(}x\texttt{)} \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \mid C; C \mid \\
& \texttt{assume}\ B \mid \texttt{assert}\ B
\end{aligned}
$$

$$
\mathit{Term} ::= \mathit{Boolean} | \mathit{Expression} | \mathit{Command}
$$

Figure 4.1: **Our Subset of the full Boogie Syntax**

$$[\![\cdot]\!] : (Boolean \cup Expression) \times Store \rightharpoonup \mathbb{Z}$$

$$\rightsquigarrow : Command \times Store \rightharpoonup Store$$

$$\varphi \in Store : Variable \rightharpoonup Value$$

$$mask \in Mask : ObjectReference \times (ClassId \times (FieldId \rightharpoonup \{0, 1\}))$$

$$h \in Heap : ObjectReference \times (ClassId \times (FieldId \rightharpoonup Value))$$

$$Variable, ClassId, FieldId ::= (a - zA - Z)+$$

$$ObjectReference ::= \mathbb{Z}^+$$

$$Value ::= \mathbb{Z}|Heap|Mask$$

Figure 4.2: **Runtime Configuration for Our Boogie Subset**