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 ( )
    {
        ...  
        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*
“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)*


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

1. **Translation**
   - **Chalice** (star)
   - **Boogie** (no star)

2. **Generation of verification conditions**

3. **Satisfiable?**
   - **SMT Solver**

   - **Yes!**
   - **No!**
Verification Pipeline

Step 1: Translation
Chalice (star) → Boogie (no star)  
High-level verification language  
Intermediate representation

Step 2: Generation of verification conditions
Boogie  
Generation of verification conditions

Step 3: SMT Solver
Satisfiable?

Yes!  
No!
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
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 ( )
    {
    ...
    }
}
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.
\[ A = \text{acc} ( x.f ) \land \ldots \land x.f = 100 \]

\( A \) is self-framing if and only if

\[ \text{Access} ( A ) \subseteq \text{Rights} ( A ) \]
To be or not to be self-framing

<table>
<thead>
<tr>
<th>A</th>
<th>Access (A)</th>
<th>Rights (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc(x.f) &amp;*&amp; x.f == 100</td>
<td>{ x.f }</td>
<td>{ x.f }</td>
</tr>
<tr>
<td>acc(x.f) &amp;*&amp; y.f == 90</td>
<td>{ y.f }</td>
<td>{ x.f }</td>
</tr>
<tr>
<td>acc(x.f) &amp;*&amp; x.g == 101</td>
<td>{ x.g }</td>
<td>{ x.f }</td>
</tr>
<tr>
<td>acc(x.f)</td>
<td>{}</td>
<td>{ x.f }</td>
</tr>
</tbody>
</table>
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)
\[ \varphi(mask)(r, C.f) = v \]

\[
\frac{}{[\text{CanAccess}(mask, r, f)]_\varphi = v}
\]

\[ \forall r : \text{ObjectReference}, f : \text{FieldId}. (\varphi(mask)(r, C.f) \in \{0, 1\}) \]

\[
[\text{IsGoodMask}(mask)]_\varphi = 1
\]

\[ \forall r : \text{ObjectReference}, f : \text{FieldId}. (\text{CanAccess}(mask, r, f) \Rightarrow \varphi(h_i)(r, C.f) = \varphi(h)(r, C.f)) \]

\[
[\text{IsGoodInhaleState}(h_i, h, mask)]_\varphi = 1
\]

\[
[\text{mask}[r, C.f]]_\varphi = \varphi(mask)(r, C.f)
\]

\[
[h[r, C.f]]_\varphi = \varphi(h)(r, C.f)
\]

\[
\varphi \models B \\
\text{assume } B, \varphi \vdash \varphi
\]

\[
\text{havoc}(x), \varphi \vdash \varphi[x \mapsto v]
\]

\[
\varphi \not\models B \\
\text{assert } B, \varphi \vdash \text{ERROR}
\]
\[
\varphi(\text{mask})(r, C.f) = v \\
\quad \therefore \quad \Box \varphi = v
\]

\forall r : \text{ObjectReference}, f : \text{FieldId}. (\varphi(\text{mask})(r, C.f) \in \{0, 1\})
\quad \boxed{\varphi(\text{mask})} = 1

\forall r : \text{ObjectReference}, f : \text{FieldId}. (\varphi(\text{CanAccess}(\text{mask}, r, f))
\quad \Rightarrow \varphi(h_i)(r, C.f) = \varphi(h)(r, C.f))
\quad \boxed{\varphi(\text{IsGoodInhaleState}(h_i, h, \text{mask}))} = 1

\boxed{\varphi(\text{mask}[r, C.f])} = \varphi(\text{mask})(r, C.f)
\quad \boxed{\varphi(h[r, C.f])} = \varphi(h)(r, C.f)

\varphi \models B \quad \text{assume } B, \varphi \triangleright \varphi

\boxed{\varphi \models B \quad \text{assert } B, \varphi \triangleright \varphi}

\text{havoc}(x), \varphi \triangleright \varphi[x \mapsto v]

\varphi \not\models B \quad \text{assert } B, \varphi \triangleright \text{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 $< \text{mask}_C, \text{heap}_C, \text{frame}_C >$

Has special commands and predicates $< \text{frame}_B >$

Translation needs to be meaning-preserving
Method Call
x.y(m) → . . . Execute method body . . .

return ←
x.y(m) \rightarrow Exhale (PRE) \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
Exhale (POST) \\
\rightarrow return \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
Inhale (POST) \\
\rightarrow Inhale (PRE) \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
\ldots \\
Execute method body
x.m(y)
method m ( t p )
requires Ar;
ensures Ae;
{
  method body...
}
method m ( tp )
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 );
method m( t p )
requires Ar;
ensures Ae;
{
    method body...
}

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

```cpp
procedure Car.refuel ( this : Car , amount : int )
{
  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;

  //inhale precondition
  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 ] == amount;
  mask := mask_e;

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

  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 );
  assert heap [ this , Car.fuel ] == amount;
  mask := mask_e;

  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;
```
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

1. **Step 1**: Translation
   - **Chalice** (star)
     - High-level verification language
   - **Boogie** (no star)
     - Intermediate representation

2. **Step 2**: Generation of verification conditions

3. **Step 3**: Satisfiable?
   - Yes!
   - No!

- SMT Solver
If program verifies in Chalice environment ...

Chalice (star) → Translation → Boogie (no star) → Generation of verification conditions → SMT Solver → Satisfiable?

Yes!

- High-level verification language
- Intermediate representation
If program verifies in Boogie environment ...

- Step 2: Generation of verification conditions
- Step 3: Satisfiable?
  - Yes!

Boogie (no star)

Intermediate representation

SMT Solver
**Lemma 6.4.4.** \( \forall C : ClassId, m : MethId : \)

*If:*

1. \( \text{Prog}_C(C, m) = \text{requires } A; \text{ensures } A';\{C_C}\wedge \)
2. \( (\text{inhale}(A, \text{mask}, h_i); \text{translate}(C_C); \text{exhale}(A, \text{mask}_e)), \varphi_{\varepsilon} \not\vdash \text{ABORT}\wedge \)
3. \( \Pi, \varphi_C, h_C \models A\wedge \)
4. \( C_C, \Pi, \varphi_C, h_C \sim \Pi', \varphi'_C, h'_C \)

*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*
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. \(\text{\textit{(Parkinson \& Summers, 2011)}}\)
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 \( \varphi_B \) s. t.
• \( C_B \) execution in Boogie leads to \( \varphi_B \)
• Terminal configurations in Chalice and Boogie match
Lemma 6.4.1. Given $C : \text{Command}_C$, $\forall \Pi : \text{Mask}_C, \varphi_C : \text{Store}_C, h_C : \text{Heap}_C, \varphi_B : \text{Store}_B$,

If:

1. $\text{translate}(C), \varphi_B \not\triangleright \text{ABORT}$
2. $\Pi, \varphi_C, h_C \cong \varphi_B$
3. $C, \Pi, \varphi_C, h_C \leadsto \Pi', \varphi'_C, h'_C$

Then there exists a $\varphi'_B : \text{Store}_B$ such that:

4. $\text{translate}(C), \varphi_B \leadsto \varphi'_B$
5. $\Pi', \varphi'_C, h'_C \cong \varphi'_B$
Lemma 6.4.2. Given $A : \text{Assertion}_C, \text{mask} : \text{Mask}_B, h_B : \text{Heap}$ (where $A$ is self-framing, mask is the current mask, and $h_B$ is the current heap),

If:

1. $\text{inhale}(A, \text{mask}, h_B) = C_i$ where $C_i : \text{Command}_B$

2. $C_i, \varphi_B \rightsquigarrow \varphi'_B$ where $\varphi'_B : \text{Store}_B$ such that $\varphi'_B \neq \text{ABORT}$

3. $\Pi', \varphi'_c, h'_c \cong \varphi'_B$, where $\Pi' : \text{Mask}_C, \varphi'_C : \text{Store}_C, h'_c : \text{Heap}_C$

Then:

4. $\Pi', \varphi'_c, h'_c \models A$

5. There exists a $\text{mask}' : \text{Mask}_B$ such that $\varphi'_B \equiv \varphi'_B[\text{mask} \mapsto \text{mask}']$, and $\text{mask} \leq_{\Pi} \text{mask}'$. 
Lemma 6.4.3. Given $A : \text{Assertion}_{C}, mask : \text{Mask}_{B}$, (where $A$ is self-framing, and $mask$ is the current mask)

If:

1. $\text{exhale}(A, mask) = C_{e} \text{ where } C_{e} : \text{Command}_{B}$
2. $C_{e}, \varphi_{B} \rightsquigarrow \varphi'_{B} \text{ where } \varphi'_{B} : \text{Store}_{B} \text{ such that } \varphi'_{B} \neq \text{ABORT}$
3. $\Pi, \varphi_{e}, h_{e} \equiv \varphi_{B}$

Then:

4. $\Pi, \varphi_{e}, h_{e} \models A, \text{ where } \Pi : \text{Mask}_{C}, \varphi_{C} : \text{Store}_{C}, h_{e} : \text{Heap}_{C}$
5. There exists a $\text{mask}' : \text{Mask}_{B}$ such that $\varphi'_{B} \equiv \varphi_{B}[\text{mask} \mapsto \text{mask}'], \text{ and } \text{mask}' \leq_{\Pi} \text{mask}$.
Definition 6.3.1. \( \text{combine} : \text{Mask}_C \times \text{Store}_C \times \text{Heap}_C \rightarrow \text{Store}_B \) such that, for \( \Pi : \text{Mask}_C, \varphi_C : \text{Store}_C, h_C : \text{Heap}_C \) we:

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

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

3. Now take an empty store \( \varphi_\epsilon : \text{Store}_B \), such that \( \varphi_\epsilon = \{\text{mask} \mapsto \emptyset, \text{heap} \mapsto \emptyset\} \).

4. Construct a \( \varphi_B : \text{Store}_B \), such that \( \varphi_B = \varphi_\epsilon[\text{mask} \mapsto \text{mask}_B, h \mapsto h_B] \cup \varphi_C \)

Definition 6.3.2. \( \equiv : \text{Mask}_C \times \text{Store}_C \times \text{Heap}_C \times \text{Store}_B \) such that, for \( \Pi : \text{Mask}_C, \varphi_C : \text{Store}_C, h_C : \text{Heap}_C, \varphi_B : \text{Store}_B \), we have

\( \Pi, \varphi_C, h_C \equiv \varphi_B \iff \text{combine}(\Pi, \varphi_C, h_C) = \varphi_B \)
\[
\begin{align*}
\text{Program : } & \text{ClassId} \rightarrow \text{FieldId} \times (\text{MethId} \times \text{MethDef}) \\
B \in \text{Boolean} : &= \text{true} \mid \text{false} \mid \neg B \mid E == E \\
E \in \text{Expression} : &= n \mid x \mid x.f \mid E + E \mid x.m_{\text{pure}}(E) \\
C \in \text{Command} : &= x := y.f \mid x.f := y \mid \text{if } B \text{ then } C \text{ else } C \mid \\
&\quad \text{while } B \text{ do } C \mid C; C \mid \text{skip} \mid x := y.m(E) \mid \\
&\quad x := \text{new } C \mid \text{assert } A \\
A \in \text{Assertion} : &= B \mid \text{acc}(x.f) \mid A \ast A \\
\text{ClassID, FieldID} : &= (a - zA - Z) + \\
\text{Term} : &= \text{Boolean} \mid \text{Expression} \mid \text{Command} \mid \text{Assertion}
\end{align*}
\]

Figure 3.1: Our Subset of the full Chalice Syntax
$$\llbracket \cdot \rrbracket : (Assertion \cup Expression) \times Mask \times Store \times Heap \rightarrow \mathbb{Z}$$

$$\leadsto: Command \times Mask \times Store \times Heap \rightarrow Mask \times Store \times Heap$$

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

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

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

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

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

---

**Figure 3.2:** Runtime Configuration for Our Chalice Subset
\( B \in \text{Boolean} ::= \text{true} | \text{false} | \neg B | E == E | \text{CanAccess}(\text{mask}, r, f) | \text{IsGoodInhaleState}(h, h, \text{mask}) | \text{IsGoodMask}(\text{mask}) \)

\( E \in \text{Expression} ::= n | x | h[r, C.f] | \text{mask}[r, C.f] | E + E \)

\( C \in \text{Command} ::= \text{var } x : t \ \mid x := E \ \mid h[r, C.f] := E \ \mid \text{mask}[r, C.f] := E \ \mid \text{havoc}(x) \ \mid \text{if } B \ \text{then } C \ \text{else } C \ \mid C; C \ \mid \text{assume } B \ \mid \text{assert } B \)

\( \text{Term} ::= \text{Boolean} | \text{Expression} | \text{Command} \)

---

**Figure 4.1:** Our Subset of the full Boogie Syntax
\[
\left[ \cdot \right] : (\text{Boolean} \cup \text{Expression}) \times \text{Store} \rightarrow \mathbb{Z}
\]

\[\sim : \text{Command} \times \text{Store} \rightarrow \text{Store}\]

\[\varphi \in \text{Store} : \text{Variable} \rightarrow \text{Value}\]

\[\text{mask} \in \text{Mask} : \text{ObjectReference} \times (\text{ClassId} \times (\text{FieldId} \rightarrow \{0, 1\}))\]

\[h \in \text{Heap} : \text{ObjectReference} \times (\text{ClassId} \times (\text{FieldId} \rightarrow \text{Value}))\]

\begin{align*}
\text{Variable, ClassId, FieldId} &::= (a - zA - Z) + \\
\text{ObjectReference} &::= \mathbb{Z}^+ \\
\text{Value} &::= \mathbb{Z} | \text{Heap} | \text{Mask}
\end{align*}

Figure 4.2: Runtime Configuration for Our Boogie Subset