Modular Specification and Verification of Delegation with SMT Solvers

IOANNIS T. KASSIOS, PETER MÜLLER

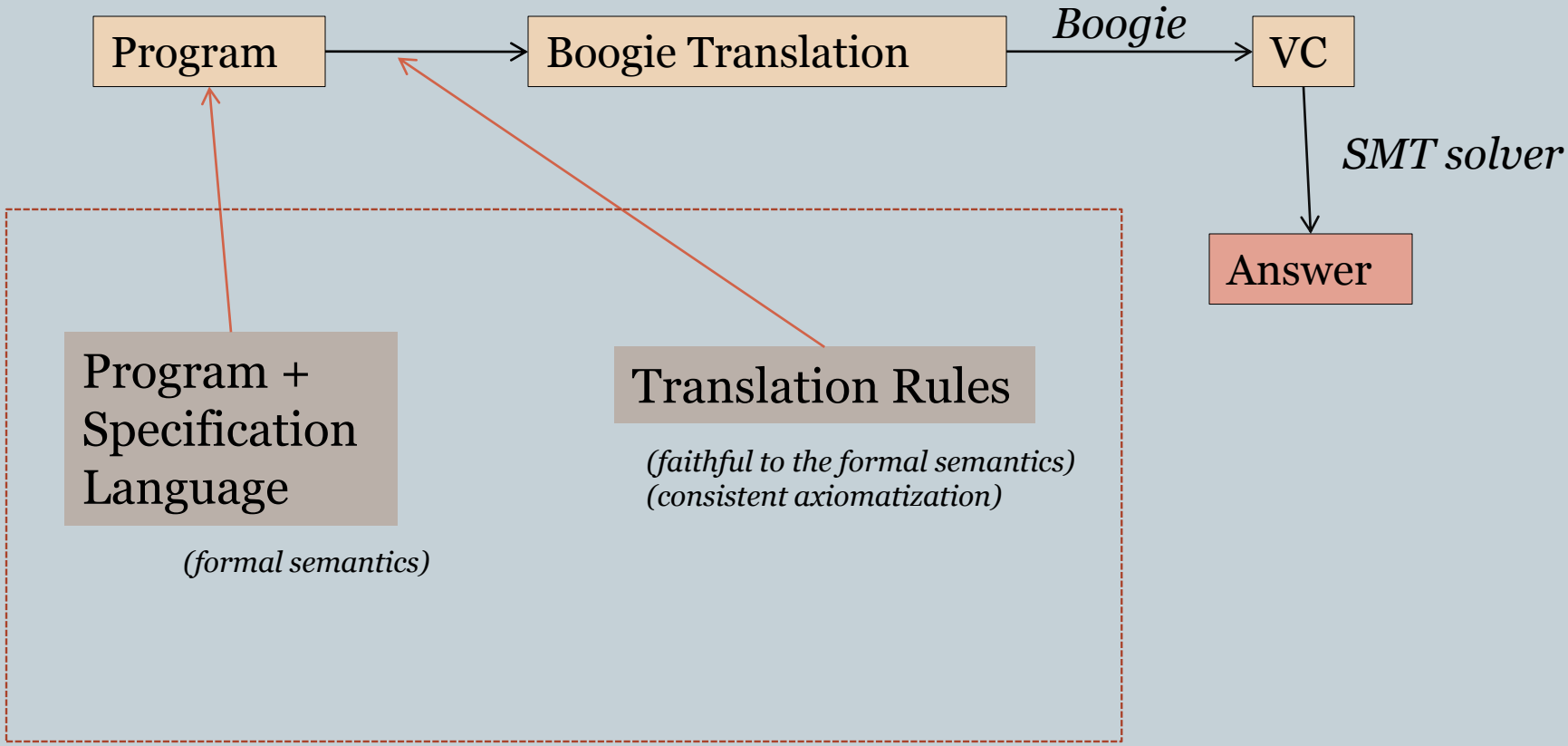
DEPT OF COMPUTER SCIENCE
CHAIR OF PROGRAMMING METHODOLOGY
ETH ZURICH

1ST HOPE WORKSHOP, 9/9 COPENHAGEN

Problem Overview

- Modular Automated Verification
  - all specification annotation comes with the program
  - no user-computer interaction during verification
  - each module verified separately
  - examples: Boogie, Chalice, Dafny, Spec#, VCC, Why,.....
  - typically uses SMT solvers → first order logic + theories
- Delegation
  - use of statically unknown code (e.g., closures)
  - essential ingredient in many design patterns
  - typically verified as a higher order logic feature

Contribution Overview



Documents and Commands


```
type Doc = { v:int };
type Command = Doc→();

val inc =
  proc(d:Doc)
    requires d≠null;
    ensures d.v=old(d.v)+1;
  { d.v:=d.v+1; };
```

```
var d:Doc; d:=new Doc; d.v:=41;
inc(d);
assert d.v=42;
```

Command Factories


```
val createAdd =
  proc(x:int) returns Command
    ensures ∀·∀d:Doc· d≠null ⇒ pre(result,d);
    ensures ∀·∀d:Doc·
        post(result,d) ⇒ d.v=old(d.v)+x;
  { result:=
      proc(d:Doc)
        requires d≠null;
        ensures d.v=old(d.v)+x;
      { d.v:=d.v+x; };
  };
```

```
var d:Doc; d:=new Doc; d.v:=40;
var f:=createAdd(2); f(d);
assert d.v=42;
```

Translation into Boogie

- A state is a heap and a lexical environment σ=(H;E)

  Env = Identifier→Location

  Heap = Location→Value

- Each static procedure definition is given a unique *key*
  - defunctionalization
- A closure is a key and a lexical environment *c*=(*k*;*e*)
- Semantical functions *pre* and *post* (specification functions) correspond to syntax **`pre`**/**`post`**
- Specifications are translated into axioms

Axiomatization in Boogie


```
val createAdd =
  proc(x:int) returns Command
    ensures ∀·∀d:Doc· d≠null ⇒ pre(result,d);
    ensures ∀·∀d:Doc·
        post(result,d) ⇒ d.v=old(d.v)+x;
  { result:=
      proc(d:Doc)
        requires d≠null;
        ensures d.v=old(d.v)+x;
      { d.v:=d.v+x; };
  };
```

k_1

k_2

$\forall c$:Closure, d:Doc, h:Heap· $key(c)=k_2 \Rightarrow (pre(h, c, d) \Leftrightarrow d \neq \mathbf{null})$

$\forall c, r$:Closure, x:$\mathbb{Z}$, h, h':Heap· $key(c)=k_1 \Rightarrow$
$(post(h', h, c, x, r) \Leftrightarrow \forall d$:Doc, h_0:Heap· $d \neq \mathbf{null} \Rightarrow pre(h_0, r, d))$

Creation and Invocation in Boogie


```
val createAdd = proc ...
```
k_1

havoc *temp*; **assume** *key*(*temp*)=k_1 ∧ *env*(*temp*)=E;
H[E[`createAdd`]] := *temp*;

```
f(d);
```

assert *pre*(H, H[E[`f`]], H[E[`d`]]);
oH:=H; **havoc** H;
assume *post*(H, oH, oH[E[`f`]], oH[E[`d`]]);

Dependency Cycles

- Specifications about specifications may be inconsistent

```
val f = proc() requires ¬pre(f); {};
```

- Cycles are not detectable statically and modularly (recursion through store)

```
var x:()→();
val g = proc() requires ¬pre(x); {};
  ......
x:=g;
```

Dependency Ordering

- We introduce a *dependency ordering* $\prec$ on closures
  - $\prec$ is strict partial ordering (irreflexive, transitive relation)
  - the specification functions of closure c may depend on those of closure d iff $c \prec d$
- Convenient default for $\prec$
  - assume closure type T_2 is a result or parameter type of closure type T_1
  - assume $type(c)=T_1$ and $type(d)=T_2$
  - then $c \prec d$
  - procedure specifications typically mention their formal parameters and results
  - note: there are no infinite types!

Custom Dependency Ordering

- Further axioms on $\prec$
  - introduced by the programmer

k_n

```
proc(...) returns...
        requires ...; ensures ...;
        below d1,d2; above c1;
{...};
```

  - checked for consistency

// consistency check
assert *init*(H[E[d1]]) ∧ *init*(H[E[d2]]) ∧ *init*(H[E[c1]]);
assert H[E[c1]]≺H[E[d1]] ∧ H[E[c1]]≺H[E[d2]];

// closure creation
havoc *temp*;
assume *key*(*temp*)=k_n ∧ *env*(*temp*)=E;
assume *temp*≺H[E[d1]] ∧ *temp*≺H[E[d2]] ∧ H[E[c1]]≺*temp*;

Dependency Cycles Revisited


```
val g = proc() requires ¬pre(x); {};
```

k_g

$\forall c$:Closure, h:Heap· $key(c)=k_g \Rightarrow$
$(pre(h, c) \Leftrightarrow \neg$ **if** $c \prec h[\mathrm{E}[\mathrm{x}]]$ **then** $pre(h, h[\mathrm{E}[\mathrm{x}]])$ **else** *undefined*$)$

A `while` Loop

ghost closure

```
val while =
  proc(cond:()→f bool, body:()→(), inv:()→g bool )
   requires ∀· pre(inv)∧(inv()⇒pre(cond));
   requires inv();
   requires body()⊑[inv()∧cond(), old(inv()∧cond())⇒inv()];
   ensures inv() ∧ ¬cond();
  { if(cond())
    { body(); while(cond, body, inv); }
  };
```

Eratosthenes' Sieve Factory


```
val eratosthenesFactory =
  proc(limit:int) returns int→fbool
    requires limit>1 ;
    ensures ∀·∀n:int· 1<n≤limit ⇒ (result(n)⇔PRIME(n));
  {
    var i:int;
    var isPrime: int→fbool;
    i:=2;
    isPrime:=func(n:int) { true };
    val inv = ...;
    val body = ...;
    while(func(){i≤limit}, body, inv);
    result:=isPrime;
  };
```

Eratosthenes' Sieve Factory: Loop Invariant


```
val inv =
  ghost() returns bool
    below isPrime;
  {
      i ≤ limit+1 ∧ inv≺isPrime
    ∧ ∀·∀j:int·
        pre(~isPrime,j)
      ∧ (1<j ⇒ (~isPrime(j)⇔
          ∀k:int· 2≤k<~i ∧ k<j ∧ PRIME(k)⇒ ¬DIVIDES(j,k)))
  };
```

Eratosthenes' Sieve Factory: Loop Body


```
val body =
  proc()
    requires inv() ∧ 1<i≤limit;
    ensures inv() ∧ i=old(i)+1;
    below inv;
  {
     val oIsPrime = isPrime;
     val oI = i;
     if(isPrime(i))
     {
        isPrime:=
          func(n:int)
              below oIsPrime; above inv;
          {oIsPrime(n) ∧ ¬DIVIDES(n,oI)};
     }
     i:=i+1;
  };
```

...AND HERE IT IS WORKING...

Conclusion

- Methodology for modular and automated verification of programs with delegation
  - first order logic specification language
  - proven consistent and sound (not shown here)
  - covers non-trivial examples
  - covers framing, abstraction, aliasing, sharing (not shown here)
- Future Work
  - implement in a verifier
  - use core ideas to design similar solution with symbolic execution and/or a permission logic (separation logic or implicit dynamic frames)