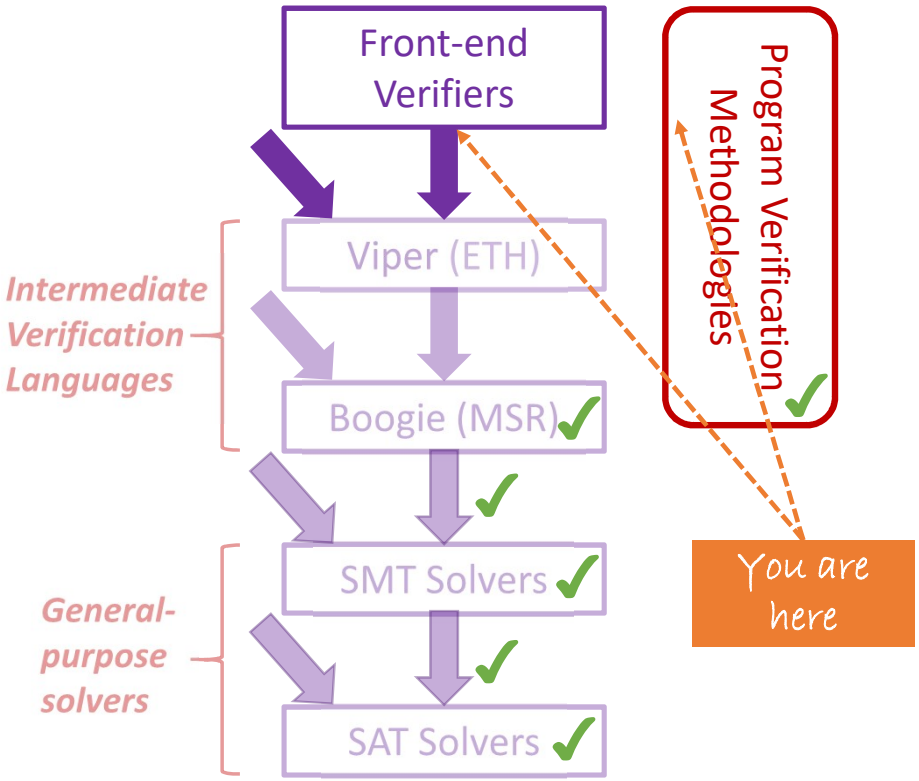
12. Permissions and Concurrent Programs

Program Verification

ETH Zurich, Spring Semester 2017
Alexander J. Summers

Next up: Permissions and Concurrent Programs



- We've seen how to use permissions to verify *heap-based sequential programs*
- How about *concurrent programs?*
  - we can use permissions to model which *threads* can access which heap locations
  - Verification for various language features can be *reduced* to a few core statements (notably *inhale and exhale* statements)
  - We will look first at some such examples for *sequential* language features
  - Then, we'll examine verification for *multithreaded programs and locks*
- We'll again use *Viper* to illustrate

Recall: Inhale and Exhale Statements

- We've briefly seen the Viper *inhale statements*: `inhale A`
  - this has the effect of *adding* the permissions required by `A` (via accessibility predicates), and *assuming* pure assertions made in `A`
  - An `inhale` *extends* the part of the heap we can access (+ adds information)
- Viper *exhale statements* perform the dual operation: `exhale A`
  - Checks that `A` is true and *removes* the permissions required by `A`
  - This indirectly affects which heap value information we can frame (next slide)
- `inhale` and `exhale` are different from `assume` and `assert`
  - A statement `assert A` checks that `A` is true *without removing permissions*
  - In other words, it checks that we *could* exhale `A` without actually doing so
  - Similarly `assume A` doesn't change the permissions we currently hold
  - Viper only supports `assume A` if `A` is *pure*; then equivalent to `inhale A`
  - Note: `assert A` is also the same as `exhale A` if `A` is *pure*

Framing and Havoc-ing Heap Information

- Recall: a field location's value can only be read when *some permission* to the location is held (*well-definedness condition* for all field reads)
- Same criterion governs when a field location's value can be *framed*
  - value can be *assumed unchanged* (only) while some permission is *still held*
  - if *all* permission to a location is lost, information about that location's value is also lost (value is *implicitly havoced*):
  - Framing *heap-dependent function values* is analogous; value preserved while *some* permission is still held to *each* field location and predicate required in function's precondition

```
method test(x:Ref) {
  inhale acc(x.val)
  x.val := 4
  exhale acc(x.val)
  inhale acc(x.val)
  assert x.val == 4 // not guaranteed
  assert x.val != 4 // also not known
}
```

- e.g. we could desugar *field assignments* using *inhales* and *exhales*
  - a write `x.val := y` can be modelled by

```
exhale acc(x.val)
inhale acc(x.val) && x.val == y
```

Encoding Method Calls with Exhale/Inhale I

- Recall that we were able to reduce our small language from slide deck 7 to just a few statements for computing weakest preconditions
  - *branching* and *sequential composition*, plus `assume` *and* `assert` *statements*
- Similar ideas work for Viper/IDF, particularly using *inhales* and *exhales*
  - We can model `havoc` and non-deterministic choice using these (*how?*)
- e.g. can we desugar Viper *method call statements* $\vec{z} := m(\vec{e})$?
  - recall 1st attempt at desugaring *Boogie procedure calls* (slide 180); we rewrote $\text{call } \vec{z} := p(\vec{e}) \rightsquigarrow \text{assert } pre[\vec{e}/\vec{x}]; \text{ havoc } \vec{g}; \text{ havoc } \vec{z}; \text{ assume } post[\vec{e}/\vec{x}][\vec{z}/\vec{y}]$
- No *global variables* $\vec{g}$ in Viper, but the heap is analogous *global state*
  - A method's specification prescribes the permissions to lose/gain across a call
- Using `exhale` + `inhale` takes care of *both permissions and havocing:*
  - $\vec{z} := m(\vec{e}) \rightsquigarrow \texttt{exhale } pre[\vec{e}/\vec{x}]; \text{ havoc } \vec{z}; \texttt{inhale } post[\vec{e}/\vec{x}][\vec{z}/\vec{y}]$

Encoding Method Calls with Exhale/Inhale II

- Desugaring on the previous slide doesn't take care of old expressions
  - Recall (slide 184) that additional work was needed to support these in Boogie
  - An *old-expression in Viper* needs to be evaluated in the *heap before the call*
- Viper supports a mechanism for supporting this simply:
  - A *label statement* `label l` can be used in normal statement position
  - A *labelled-old expression* `old[l](e)` can be used in subsequent expressions
  - Meaning: evaluate `e` in the heap *as it was* at the `label l` statement
- We can now write an improved rule, supporting old expressions:
- $\vec{z} := m(\vec{e}) \rightsquigarrow$ **label l**; **exhale** $pre[\vec{e}/\vec{x}]$; havoc $\vec{z}$;
  inhale $post[\vec{e}/\vec{x}][\vec{z}/\vec{y}]$[**old[l]**/**old**]
  - here, the "substitution" [**old[l]**/**old**] should replace all expressions of the form `old(e)` with corresponding expressions `old[l](e)`

Eliminating Viper Loops

- Recall our old rule for *eliminating loops with invariants* (slide 148):
  - $\text{while(b)invariant } \text{A}_\text{I}\{\text{s}\} \rightsquigarrow$
    $\text{assert } \text{A}_\text{I}\text{; havoc } \vec{\text{x}}\text{;}$
    $((\text{assume } \text{A}_\text{I} \wedge \text{b}\text{; s; assert } \text{A}_\text{I}\text{; assume false}) \ [] \ (\text{assume } \text{A}_\text{I} \wedge \neg\text{b}))$
  - where $\vec{\text{x}}$ are the (perhaps zero) variables modified by s
- This rule works for first-order logic assertions over program variables
  - For Viper/IDF we again need to take care of *permissions* and *heap values*
- We obtain an analogous rule for *eliminating Viper loops with invariants* by employing **`inhale`** and **`exhale`** statements:
  - $\text{while(b)invariant } \text{A}_\text{I}\{\text{s}\} \rightsquigarrow$
    $\textbf{exhale } \text{A}_\text{I}\text{ ; havoc } \vec{\text{x}}\text{;}$
    $((\textbf{inhale } \text{A}_\text{I} \wedge \text{b}\text{; s; } \textbf{exhale } \text{A}_\text{I}\text{; assume false}) \ [] \ (\textbf{inhale } \text{A}_\text{I} \wedge \neg\text{b}))$
- The exhales havoc all heap locations *potentially modified* by the loop

Summary: Encoding with Inhales and Exhales

- We use inhale and exhale statements to model *transferring assertions* from/to the current verification scope (method body / loop body)
  - when these assertions are *self-framing*, any constraints about heap locations will be provided along with *sufficient permissions to frame the information*
  - this makes it conceptually justified to *transfer assertions from one scope to another* (e.g. a method precondition, from caller to callee)
- Viper doesn't know the *justification* for inhale/exhale statements
  - e.g. after desugaring a method call, no-longer explicit *why* the inhale happens
- We use the inhales and exhales to *encode the intended methodology*
  - when must assertions be checked, and permissions given up?
  - when can assertions be assumed, and permissions added?
- Can we apply this general recipe to other programming constructs?

Structured Parallelism

- Suppose we want to verify certain classes of *concurrent programs*
  - in *some* high-level programming language; we'll verify by encoding to Viper
- For example, consider support for *forking and joining threads*
  - consider *fork statements* `fork tid := m(e⃗)` , which create a *new thread* executing method call `m(e⃗)`; thread can be subsequently identified by `tid`
  - corresponding *join statements* `join z⃗ := tid` cause the *current thread to block* until thread `tid` terminates (any out parameters are assigned to `z⃗`)
- For simplicity, we'll assume we use only *structured parallelism:*
  - `fork` and `join` statements come in pairs, which are "scoped" (i.e. an extra scope from any `fork` to the `join` would fit the existing program structure)
  - We also assume that thread identifiers are uniquely used in one such pair
  - This can model the structured parallelism support present in many languages

Structured Parallelism Example

- We can e.g. parallelise the `incrementAll` example (slide 240):
  - in some imaginary high-level language (with arrays, IDF-like specifications)

```
method incrementAllParallel(a:Int[], start:Int, end:Int)
  requires forall i:Int :: start <= i && i< end ==> acc(a[i])
  ensures forall i:Int :: start <= i && i< end ==> acc(a[i])
    && a[i] == old(a[i]) + 1
{
  if(end-start > 10) // some threshold for parallelism
  {
    var mid : Int := start + (end - start) / 2
    fork t1 := incrementAllParallel(a, start, mid)
    fork t2 := incrementAllParallel(a, mid, end)
    join t1 // join order is not important
    join t2
  } else {
    // do the increments directly in a loop; e.g. as on slide 240...
  }
}
```

Verification of Structured Parallelism

- How could we verify such examples with `fork`/`join` statements?
  - We already know how to encode arrays, control flow etc. in Viper
- Observation: a `fork` statement entails the *transfer of an assertion*
  - permissions and properties from the *forked method's precondition* should be conceptually *transferred from the forking thread to the forked thread*
  - in a Viper setting, we can model a `fork` with an `exhale` of this precondition
- Dually, a `join` statement entails the *transfer of an assertion*
  - permissions and properties from the *forked method's postcondition* should be conceptually *transferred from the joined thread to the joining thread*
  - in a Viper setting, we can model a `join` with an `inhale` of this postcondition
- We obtain a suitable encoding into Viper (cf. method calls: slide 269):
  - `fork tid` $:=$ `m(`$\vec{\mathrm{e}}$`)` $\rightsquigarrow$ `label l;` `exhale` $\mathrm{pre}[\vec{\mathrm{e}}/\vec{\mathrm{x}}]$
  - `join` $\vec{\mathrm{z}}$ $:=$ `tid` $\rightsquigarrow$ $\mathrm{havoc}\ \vec{\mathrm{z}};$ `inhale` $\mathrm{post}[\vec{\mathrm{e}}/\vec{\mathrm{x}}][\vec{\mathrm{z}}/\vec{\mathrm{y}}]$`[old[l]/old]`

Structured Parallelism Example Encoded in Viper

```
method incrementAllParallel(a:Array, start:Int, end:Int)
  requires forall i:Int :: start <= i && i< end ==> acc(loc(a,i).val)
  ensures forall i:Int :: start <= i && i< end ==> acc(loc(a,i).val)
    && loc(a,i).val == old(loc(a,i).val) + 1
{
  if(end-start > 10) // some threshold for parallelism
  {
    var mid : Int := start + (end - start) \ 2
    label l1; exhale forall i:Int :: start <= i && i< mid ==> acc(loc(a,i).val)
    label l2; exhale forall i:Int :: mid <= i && i< end ==> acc(loc(a,i).val)
    inhale forall i:Int :: start <= i && i< mid ==> acc(loc(a,i).val)
      && loc(a,i).val == old[l1](loc(a,i).val) + 1
    inhale forall i:Int :: mid <= i && i< end ==> acc(loc(a,i).val)
      && loc(a,i).val == old[l2](loc(a,i).val) + 1
  } else {
    // do the increments directly; cf. slide 240 ...
  }
}
```

Supporting Locks

- Let's consider now the usage of *locks in concurrent programs*
  - Typically used to *guard access* to some data/resource in the program
- Let's consider a high-level language with a notion of locks
  - *lock statements* `acquire l`: block current thread until lock `l` acquired
  - corresponding *unlock statements* `release l`: release currently held lock `l`
- We need some way of specifying the *role* of a lock in the program
  - which data is protected by the lock? Are properties guaranteed of the data?
- A common methodology is to declare a *lock invariant* per lock
  - An assertion which describes the state *guarded by the lock*
  - In an IDF-like setting, this would typically be a *self-framing assertion*
  - Permissions in the lock invariant define the data guarded by the lock

Example: Lock-Guarded Counter

- Here's a simple pseudocode example using locks and lock invariants

```
class LockableCounter {
  var count: int;
  // lock invariant for all instances of this class
  lock invariant acc(this.count) && this.count >= 0
}

class Test {
  method Increment(c : LockableCounter)
    requires c != null
  {
    acquire c // this provides access to c.count
    c.count := c.count + 1 // - 1 would not work!
    release c
  }
}
```

- How should such programs be verified?

Verification with Lock Invariants

- `acquire` statements *transfer the lock invariant* to the current thread
  - operationally, these block until we hold the lock; if we consider only partial correctness, we can simply consider how the state should be *after* the acquire
  - We can model an `acquire` using an `inhale` of the lock invariant
- `release` statements *transfer the lock invariant* from current thread
  - We can model a `release` using an `exhale` of the lock invariant
- We obtain a suitable encoding into Viper
  - `acquire e` $\rightsquigarrow$ `inhale` $\mathrm{inv}[\mathrm{e}/\mathrm{this}]$ (inv is the lock invariant from `e`'s class)
  - `release e` $\rightsquigarrow$ `exhale` $\mathrm{inv}[\mathrm{e}/\mathrm{this}]$
- We might also want a verifier to guarantee *additional properties*
  - e.g. we *only release* locks we hold
  - we don't *forget* to release locks (?)

```
method Increment(c : Ref)
  requires c != null
{
  inhale acc(c.count) && c.count >= 0
  c.count := c.count + 1
  exhale acc(c.count) && c.count >= 0
}
```

Modelling Held Locks I

- Suppose we want to extend our methodology for lock verification, to guarantee that threads only release locks that are *currently held*
  - attempting to release a lock that is not held would likely cause a *runtime error*
  - when designing a verifier, we must *decide which runtime errors to rule out*
- We need a way of tracking which locks a thread holds, in the verifier
  - note that the existing exhale on `release` is not sufficient; e.g. consider:

```
method Wrong()
{
  var c : LockableCounter := new LockableCounter()
  c.count := 0
  release(c) // runtime error
}
```

- Idea: we add *auxiliary permissions* to our Viper encoding
  - An extra field `holds`; we use `acc(l.holds)` to represent holding the lock `l`

Modelling Held Locks II

- Let's extend our Viper encoding, with the extra `holds` field in mind:
  - `acquire e` ⇝ `inhale acc(e.holds) && inv[e/this]`
  - `release e` ⇝ `exhale acc(e.holds) && inv[e/this]`
- The new requirement enforces that *only held locks can be released*
  - a verified program will therefore be *free of associated runtime errors*
- Note that the `holds` field did not exist in our original source program
  - This is *auxiliary state* added by our encoding (we don't use the field value)
  - The manipulation of this state is *generated automatically* via above encoding
- Can we also prevent *forgetting to release held locks*? e.g. this code:

```
method AlsoWrong(c : LockableCounter)
  requires c != null
{
  acquire c
  c.count := c.count + 1
}
```

Enforcing Lock Release I

- Our encoding already provides a way of checking which locks are held
- Viper *allows permissions to be leaked*: e.g. when checking a postcondition, it is allowed to have *more* permission than specified
  - unfortunately, for our `holds` field, this amounts to *forgetting holding a lock*
- We would like a way of enforcing that we *don't* hold such permissions
  - recall that *negating* accessibility predicates is not allowed in IDF (slide 211)
- Instead, Viper provides a feature to inspect currently held permissions
  - A *perm expression* `perm(e.f)` denotes the amount of permission held for the field `f` of receiver `e`
  - For example, the assert statements in the following example will both verify:

```
method TestPerm(x : Ref)
  requires acc(x.count, 1/2)
{
  assert perm(x.count) == 1/2
  exhale acc(x.count, 1/2)
  assert perm(x.count) == none // no permission
}
```

Enforcing Lock Release II

- We can use *perm expressions* to enforce that all locks are released
- At the end of every translated method body, we add an assertion:

```
assert forall r:Ref :: perm(r.holds) == none
```

- This assertion will fail if we *still hold any locks* at this program point
  - Verified program is guaranteed to *never hold locks across method boundaries*
- If we *want* to support holding *specified* locks across method boundaries (e.g. acquiring a lock in a method, releasing in its caller), we'd need *specifications at the source level to support this*
  - introduce a `holds(e)` assertion at the source level (e.g. in postconditions), which would be translated to `acc(e.holds)` at the Viper level
  - Enforce the check above *after exhaling a method's postcondition*; e.g. as follows:

```
exhale post // normal postcondition first
assert forall r:Ref :: perm(r.holds) == none
assume false // don't exhale postcondition again
```

Example of the Full Lock Encoding

- *Source below, Viper opposite:*

```
class LockableCounter {
  var count: int;
  lock invariant acc(this.count) &&
    this.count >= 0
}

class Test {
  method getLock(c:LockableCounter)
    requires c != null
    ensures holds(c) &&
      acc(c.count) && c.count >= 0
  {
    acquire c
  }

  method Increment(c:LockableCounter)
    requires c != null
  {
    getLock(c)
    c.count := c.count + 1
    release c
  }
}
```

```
field holds:Bool // could have any type

method getLock(c : Ref)
  requires c != null
  ensures acc(c.holds) && acc(c.count) && c.count >= 0
{
  // acquire c:
  inhale acc(c.holds) && acc(c.count) && c.count >= 0
  // exhale postcondition
  exhale acc(c.holds) && acc(c.count) && c.count >= 0
  assert forall r:Ref :: perm(r.holds) == none
  assume false
}

method Increment (c : Ref)
  requires c != null
{
  getLock(c)
  c.count := c.count + 1
  // release c:
  exhale acc(c.holds) && acc(c.count) && c.count >= 0
  assert forall r:Ref :: perm(r.holds) == none
  assume false
}
```

Permissions and Concurrent Programs - Summary

- In a permission-based setting, verification of many language features can be mostly reduced to appropriate *inhale and exhale statements*
  - For sequential constructs, such as *method calls* and *loops*
  - For concurrency features, such as *threads* and *locks*
- In general, we can identify the *assertions to be transferred* between program entities, and model their *addition or removal* appropriately
- Encoding depends on the *properties* that verification tool guarantees
  - for locks, we saw three *different encodings*, with *different guarantees*
  - used *auxiliary state (permissions, in this case)* to explicitly track held locks
- These ideas have been used in other concurrency verification projects
  - e.g. *Chalice* is a research language supporting threads, locks, channels etc.
  - *VerCors* is a project (@University of Twente) for verification of concurrent Java and GPU code –verifiers are implemented via such *translations to Viper*

Permissions and Concurrent Programs – References

- Permission-based Verification for Fork/Join and Locks:
  - *Resources, Concurrency and Local Reasoning.* Peter O'Hearn (2004)
  - *A Basis for Verifying Multi-Threaded Programs.* K.R.M. Leino, P. Müller. (2009)
  - *Deadlock-free Channels and Locks.* K.R.M. Leino, P. Müller, Jan Smans (2010)
  - *A Formal Semantics for Isorecursive and Equirecursive State Abstractions.* A.J. Summers and S. Drossopoulou (2013)
- Implicit Dynamic Frames:
  - *Specification and Automatic Verification of Frame Properties for Java-like Programs.* J. Smans. (PhD thesis) (2009)
- Viper:
  - *Viper: A Verification Infrastructure for Permission-Based Reasoning.* P. Müller, M. Schwerhoff, A. J. Summers. (2016)