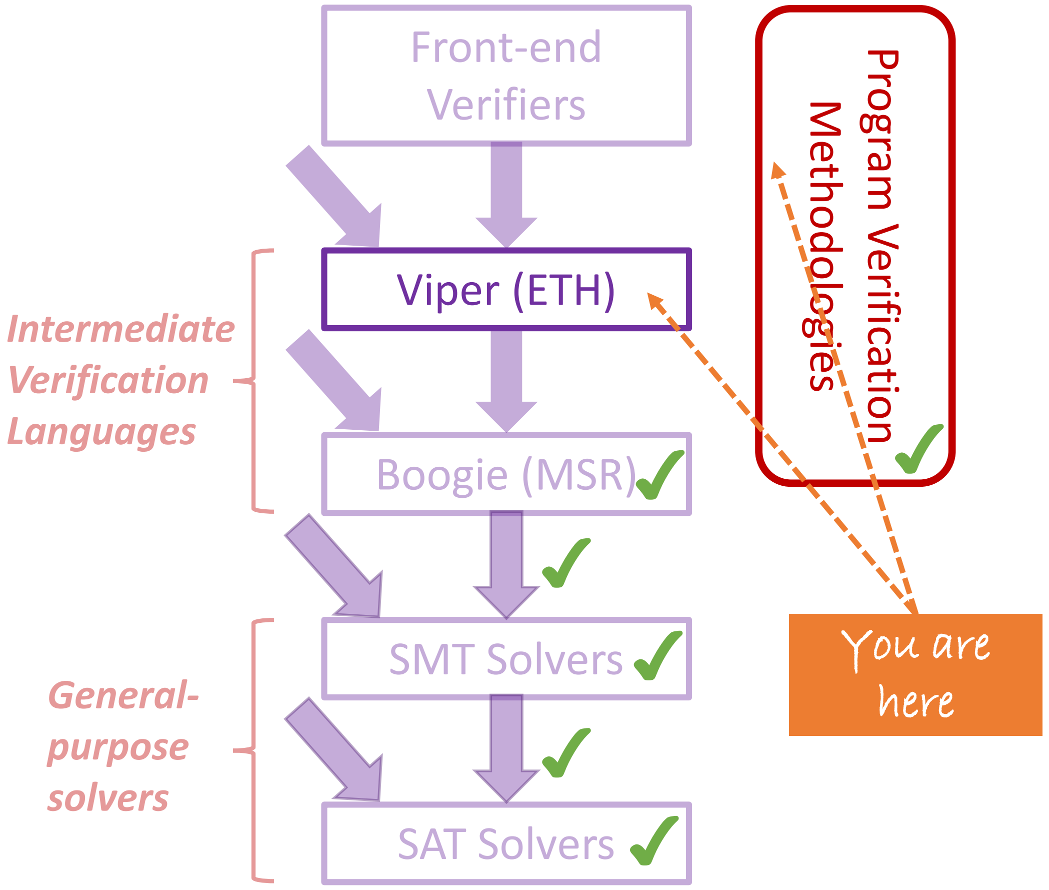
11. Advanced Specification and Verification

Program Verification

ETH Zurich, Spring Semester 2017
Alexander J. Summers

Next up: Advanced Specification and Verification



- We now have the basic tools for specifying and verifying *unbounded, state-dependent program properties*
- We'll next tackle some frequent and challenging specification scenarios
  - *degenerate specification cases* (weak specifications allow too many behaviours)
  - expressing properties which are *implicit* in the program state (using *auxiliary state*)
  - modelling *non-deterministic* behaviour
  - *abstracting* over concrete implementation decisions (e.g. how memory is allocated)
  - Appealing to *external reasoning* / *proofs*
- We'll again use *Viper* to illustrate

Procedure Modularity Revisited

- *Modular reasoning* (e.g. procedure-modularity) brings the challenge of *how general* to make a specification
  - Making *preconditions weaker* and *postconditions stronger* is good for callers
  - *But* this will make implementing of the procedure harder, and *less flexible*
  - How much *knowledge of current call-sites* should be used?
  - How much information about the implementation should be *exposed*?
  - Recall: procedure implementations can be changed without re-verifying callers, provided that a new implementation *still satisfies the specification*
- As well as logical strength, the issue of *information hiding* is relevant
  - e.g. *field names* in specifications (e.g. `acc(l.next)`) may not be desirable
  - client verification becomes *sensitive to these names*; bad for code evolution
  - (less problematic for internal methods, where field names are visible anyway)

Abstract Predicates and Functions

- Recall our specification for list append (slide 231):

```
method appendfunc(l1 : Ref, l2: Ref)
  requires list(l1) && list(l2) && l2 != null
  ensures list(l1) && elems(l1) == old(elems(l1) ++ elems(l2))
```

- The specification does not leak implementation details (field names)
  - client code (i.e. callers) can be verified *independently of these details*
- An *abstract predicate* consists of a name and signature, but no body
  - an *abstract function* consists of a function signature/specification but no body
  - an *abstract method* consists of a method signature/specification but no body
- Used to abstract over concrete definitions in specifications

```
predicate list(start : Ref)

function elems(start: Ref) : Seq[Int]
  requires list(start)
```

  - enables *information hiding*, even for *bounded data structures*; clients verified against such specifications *need not be re-verified* when the data structure implementation is exchanged

Degenerate Specification Cases I

- Writing procedure specifications *without implementing them* runs the risk that implementation turns out to be awkward/impossible
  - e.g. a method can't be implemented:
  - this could introduce *inconsistency* in client verification; even if not, fixing the specification will entail re-verification

```
method veryLargeInt() returns (n:Int)
  ensures forall i:Int :: n > i
```

- A procedure might not be implementable for more-subtle reasons
  - e.g. the precondition allows for unintended (perhaps not useful) *initial states*
  - A verified implementation will have to cover such *degenerate cases*, too
- e.g. what's wrong with this specification for `addAtEnd` (slide 233)?

```
method addAtEnd(l1:Ref, l2:Ref)
  requires lseg(l1,l2) && acc(l2.val) && acc(l2.next)
  ensures lseg(l1, old(l2.next)) &&
    lsegelems(l1, old(l2.next)) == old(lsegelems(l1, l2)) ++ Seq(old(l2.val))
```

Degenerate Specification Cases II

```
method addAtEnd(l1:Ref, l2:Ref)
  requires lseg(l1,l2) && acc(l2.val) && acc(l2.next)
  ensures lseg(l1, old(l2.next)) &&
    lsegelems(l1, old(l2.next)) == old(lsegelems(l1, l2)) ++ Seq(old(l2.val))
```

- The precondition permits the *possibility* that `l1.next == l2.next`
  - the permissions included (even inside the predicate) do not prevent this
- In such an initial state, the *postcondition is unsatisfiable*
  - `lseg(l1, old(l2.next))` is satisfiable, but the instance will not recurse
  - `lsegelems(l1, old(l2.next))` is a singleton sequence, in this case
  - since `old(lsegelems(l1, l2))` cannot be an empty sequence, the last conjunct of the postcondition cannot be satisfied for such an initial state
- Similar if `l2.next` aliased any other node making up `lseg(l1,l2)`
  - also not ruled out by the specification above

Degenerate Specification Cases III

- We can eliminate the possibility of location aliasing *any* node in `lseg(l1,l2)` by requiring *permission to a field of* `l2.next`
  - the full permissions to the `next` and `val` fields of the nodes making up the `lseg` are already in the predicate instance (or in the precondition, for `l2`)
- For example, we can use the following specification:

```
method addAtEnd(l1:Ref, l2:Ref)
  requires lseg(l1,l2) && acc(l2.val) && acc(l2.next) && acc(l2.next.next, 1/2)
  ensures lseg(l1, old(l2.next)) &&
    lsegelems(l1, old(l2.next)) == old(lsegelems(l1, l2)) ++ Seq(old(l2.val))
  ensures acc(old(l2.next).next, 1/2)
```

  - note the use of `old`: in the postcondition we can't evaluate `l2.next`
  - fractional permission lets callers frame value of location `l2.next.next`
  - this specification requires an extra unfold/fold around each call to `addAtEnd` in method `appendit` (slide 233); the file `list-examples.vpr` on the course website contains an alternative specification which avoids this

Auxiliary State

- Specifications often describe concepts which are not explicit in code
  - e.g. the complete sequence of *values* stored in a linked-list data structure
- For verification, we often need to *make these concepts explicit*
- It is common to add *additional (auxiliary) state* to the program
  - e.g. state which is not needed at runtime, only for specification purposes
- Two main variants of auxiliary state: *ghost state* and *model state*
  - difference is in how the state is *updated* when the program's state changes
- *Ghost state* is auxiliary program state which is *manually updated*
  - i.e. additional code is added to the program, to give ghost state correct values
  - this *ghost code*, along with all ghost state can be *erased at runtime*
- *Model state* is auxiliary program state which is *automatically updated*
  - verifier must provide *native support*: e.g. heap-dependent functions in Viper

Ghost State: List Append with Predicate Parameters

- Recall our list append with extra predicate parameters (slide 229):

```
method appendelems(l1 : Ref, l2: Ref, l1elems : Seq[Int], l2elems : Seq[Int])
  requires listelems(l1,l1elems) && listelems(l2,l2elems) && l2 != null
  ensures listelems(l1,l1elems ++ l2elems)
{
  unfold listelems(l1,l1elems)
  if(l1.next == null) {
    l1.next := l2
  } else {
    appendelems(l1.next,l2,l1elems[1..],l2elems)
  }
  assert (l1elems ++ l2elems)[1..] == (l1elems[1..] ++ l2elems)
  fold listelems(l1,l1elems ++ l2elems)
}
```

- The extra method parameters are *ghost state*, here
  - we have to manually provide the correct values, e.g. for recursive call

Model State: Heap Dependent Functions

- The `elems` function (slide 230) is an example of *model state*
  - The function *precondition* tells the verifier when the function's value might potentially change
  - The function *definition* allows reasoning about *how* it changes
- This reasoning is automatic for *local updates* to the state:
  - when data structure is traversed in the *same recursive fashion*
- Method specifications must include information on *how these functions change value*

```
function elems(start: Ref) : Seq[Int]
  requires list(start)
{ unfolding list(start) in (
    (start.next == null ? Seq(start.val) :
     Seq(start.val) ++ elems(start.next)))
}

method appendfunc(l1 : Ref, l2: Ref)
  requires list(l1) && list(l2) && l2 != null
  ensures list(l1) &&
    elems(l1) == old(elems(l1) ++ elems(l2))
{
  unfold list(l1)
  if(l1.next == null) {
    l1.next := l2
  } else {
    appendfunc(l1.next,l2)
  }
  fold list(l1)
}
```

Cycle-detection in Linked Lists, Revisited

```
method isCyclic(nodes : Set[Ref], root:Ref)
  returns (cyclic : Bool)
  requires root != null && root in nodes
  requires forall n:Ref :: n in nodes ==> acc(n.next)
    && (n.next != null ==> n.next in nodes)
  ensures forall n:Ref :: n in nodes ==> acc(n.next)
    && (n.next != null ==> n.next in nodes)
{
  var seen : Set[Ref] := Set(root) // built-in Set[T]
  var current : Ref := root

  while(current.next != null && !(current.next in seen))
    invariant current != null && current in nodes
    invariant forall n:Ref :: n in nodes ==> acc(n.next)
      && (n.next != null ==> n.next in nodes)
  {
    seen := seen union Set(current.next)
    current := current.next
  }
  cyclic := (current.next != null)
}
```

- Recall: example from slide 237
- To specify this using quantified permissions, the parameter `nodes` is used
  - can be considered ghost state
- No *functional specification* so far – i.e. when should the value of `cyclic` be true?
  - how can we characterise the list (not) having a cycle?
  - can we do this in a way which avoids recursive definitions / inductive reasoning?

Functional Specification of Cyclicity

```
method isCyclic(nodes : Set[Ref], root:Ref)
  returns (cyclic : Bool , path:Seq[Ref])
  requires root != null && root in nodes
  requires forall n:Ref :: n in nodes ==> acc(n.next)
    && (n.next != null ==> n.next in nodes)
  ensures forall n:Ref :: n in nodes ==> acc(n.next)
    && (n.next != null ==> n.next in nodes)
  // it is a path:
  ensures forall i:Int :: 0<= i && i < |path|-1 ==>
    path[i] in nodes && path[i].next == path[i+1]
  ensures |path| > 0 &&
    (!cyclic ==> path[|path|-1] == null)
  ensures cyclic ==> path[|path|-1] in path[..|path|-1]
{
  var seen : Set[Ref] := Set(root) // built-in Set[T]
  var current : Ref := root
  path := Seq(root)

  while(current.next != null && !(current.next in seen))
    …… // see quantified-permission-examples.vpr online
```

- We add as further *ghost state*, the *sequence* of nodes visited
  - this sequence is returned as an extra *out parameter* `path`
  - additional post-conditions connect sequence to `cyclic`
- Note that this specification doesn't directly say that there are no cycles when `!cyclic`
  - this is implied by the given specification; we could justify this via *external reasoning*
  - we express *sufficient properties* in a way which can be simply handled by the verifier

Writing out a Set to an Array

- Suppose we want to write a `Set` of integers into an array (any order)
  - we'll model arrays in Viper as shown in the last lecture (slide 239)
- The following specification is one way to express our requirements
  - the method will have to *allocate the array*; we don't pass permissions in

```
method setToArray(vals:Set[Int]) returns (a:Array)
ensures len(a) == |vals|
ensures forall i:Int :: 0 <= i && i< len(a) ==> acc(loc(a,i).val)
ensures forall i:Int :: i in vals ==>
          exists k: Int :: 0 <= k && k < len(a) && loc(a,k).val == i
```

- This usage of existential quantification under the `forall` (*quantifier alternation*) can be challenging for the underlying SMT solver
  - indirectly specifies that a suitable *function* from the set to the array exists
  - Direct *quantification over functions* is not supported (SMT is first-order)

Modelling Array Allocation

- We don't want to concretely specify how a memory allocator works
  - instead, abstract over how it works, via *non-determinism and specifications*
- We model allocation by ***assigning*** **a** ***an arbitrary value*** (havocing)
  - we then add the *assumption* that the result has the right size
  - this *non-determinism abstracts over* what a real memory allocator will do

```
method setToArray(vals:Set[Int]) returns (a:Array)
ensures …
{
  // model allocating an array of size |vals|
  a := havocArray()
  assume len(a) == |vals|
```

- We use an abstract method `havocArray()` for the havoc
  - Viper doesn't have a havoc statement `method havocArray() returns (a:Array)`
  - method lets us generate arbitrary values of a type (*why not use a function?*)

Inhale and Exhale Statements

- Viper supports a statement `inhale A` (where A is an assertion)
  - this has the effect of *adding* the permissions required by A (via accessibility predicates), and *assuming* pure assertions made in A
  - In our example, we can model allocation using an `inhale` statement:

```
method setToArray(vals:Set[Int]) returns (a:Array)
ensures …
{
  // model allocating an array of size |vals|
  a := havocArray() // use an abstract method to
  inhale len(a) == |vals| // same as assume for pure assertions
  inhale forall i:Int :: 0 <= i && i< len(a) ==> acc(loc(a,i).val)
  // now we have the permissions to the array slots
```

- The dual statement is called `exhale A`
  - Checks that A is true and *removes* the permissions required by A
- We'll see more of `inhale` and `exhale` in the next lecture

Writing out the Set elements

```
…
var s : Set[Int] := vals
var element : Int; var j : Int := 0;

while (|s| > 0)
  invariant forall i:Int :: 0 <= i && i< len(a) ==>
              acc(loc(a,i).val)
  invariant s subset vals && j == |vals setminus s|
  // forall-exists here is difficult for the tools
  invariant forall i:Int :: {i in vals} // note trigger!
    i in (vals setminus s) ==>
  exists k: Int :: 0 <= k && k < j && loc(a,k).val == i
{
  element := havocInt() // simulate a havoced Int value
  assume element in s // we have *some* element of s
  loc(a,j).val := element
  s := s setminus Set(element)
  j := j + 1
}
… // see set-to-array.vpr for full code
```

- We can iterate through the set by *non-deterministically choosing* elements
  - again, achieved via a *havoc + assumption*
  - each element is then added to the next available array slot
- The loop invariants still use forall-exists pattern
  - *difficult for the verifiers*
  - full example works *only in Carbon* specified this way

Eliminating the Existentials

- We can remove the need for the existentials in our specification, by employing *additional ghost state*
  - the idea is to turn the *witness* for the existential quantifier into *explicit state*
  - we add ghost state to represent the *map* from set elements to array indices (we can use the `Map` domain from exercise sheet 5, question 3)

```
method setToArrayGhostState(vals:Set[Int]) returns (a:Array, map:IntMap)
  ensures len(a) == |vals|
  ensures forall i:Int :: 0 <= i && i< len(a) ==> acc(loc(a,i).val)
  ensures forall i:Int :: {i in vals} i in vals ==>
    let k == (select(map,i)) in
      0 <= k && k < len(a) && loc(a,k).val == i
{ …
```

- Note that the *existential is no longer necessary*: we can now write the specification by looking up values in the ghost state `map`
  - see `set-to-array.vpr` online (works automatically in either verifier)

Advanced Specification and Verification - Summary

- We've seen techniques for specifying *rich function properties*, helping *automate their verification*, and *abstracting implementation details*
- *Abstract predicates, functions and methods* allow specifications to be agnostic as to the underlying representation of data
  - *Abstract methods* can also be used to model other behaviours, e.g. havocing
- *Auxiliary state* is a powerful specification mechanism
  - Turn concepts which are implicit in the code into *explicit program state*
  - *Ghost state* must be manually updated with additional program code
  - *Model state* - automatically kept up-to-date by verifier (e.g. Viper functions)
- *Non-determinism* plus *assumptions* can be used to abstract over concrete program behaviours (or to model real non-determinism)
- In the next lecture, we'll see how to use and extend the techniques presented so far to also encode and verify *concurrent programs*

Advanced Specification and Verification – References

- Model state
  - *Data abstraction and information hiding.* K.R.M. Leino, G. Nelson. (2000)
  - *A verification methodology for model fields.* K.R.M. Leino, P. Müller. (2006)
- Implicit Dynamic Frames:
  - *Specification and Automatic Verification of Frame Properties for Java-like Programs.* J. Smans. (PhD thesis) (2009)
- Recursive Predicates and Functions:
  - *Separation logic and abstraction.* M. J. Parkinson and G. Bierman. (2005)
  - *A Formal Semantics for Isorecursive and Equirecursive State Abstractions.* A.J. Summers and S. Drossopoulou (2013)
  - *Viper: A Verification Infrastructure for Permission-Based Reasoning.* P. Müller, M. Schwerhoff, A. J. Summers. (2016)
  - *Verification condition generation for permission logics with abstract predicates and abstraction functions.* S. Heule, I. T. Kassios, P. Müller, A. J. Summers (2013)