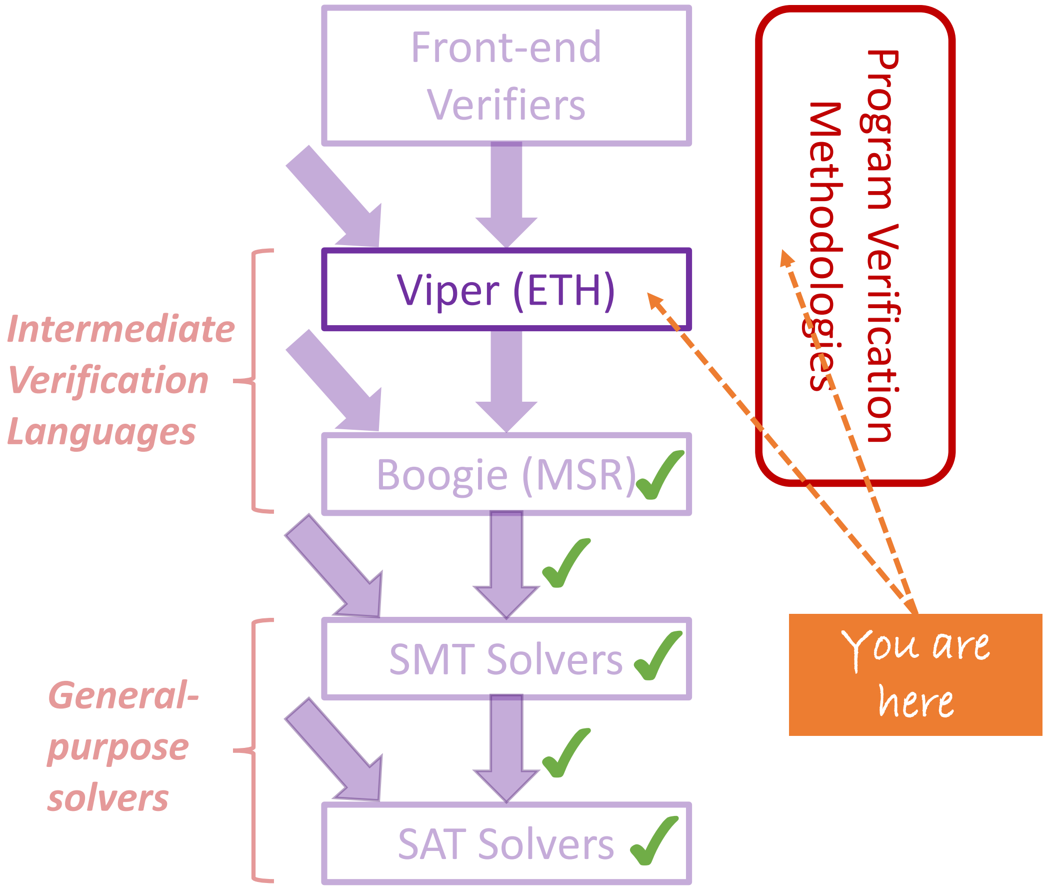
10. Unbounded Heap Data

Program Verification

ETH Zurich, Spring Semester 2017

Alexander J. Summers

Next up: Unbounded Heap Data



- We've seen the basics of the *Implicit Dynamic Frames (IDF)* logic
  - Provides constructs for *framing heap-dependent information* (per field location)
- Real programs use *unbounded* data
  - *recursively-defined* structures (lists, trees)
  - *random-access data* (arrays, maps)
  - *mutual references* (doubly-linked lists)
  - *complex aliasing* (reference-based graphs)
- We'll look at extensions of basic IDF to enable *verification and framing* for *unbounded heap-based data structures*
  - We'll use *Viper* to illustrate these features

Recursive Data Structures

- The following Viper example appends two linked lists:

```
method append(l1: Ref, l2: Ref)
{
  if(l1.next == null) {
    l1.next := l2
  } else {
    append(l1.next,l2)
  }
}
```

- How can we write a specification for this (recursive) method?
  - The pre-condition must require permission for *unboundedly many* locations
  - The post-condition should guarantee `l1` to be the appended linked-list
- Two main techniques for handling unbounded data structures in IDF:
  - *recursive definitions (predicates and functions), and quantified permissions*

Predicates

- We add support for *user-defined predicates* to the assertion logic
- A *predicate declaration* consists of
  - a *predicate name*, used to identify *instances* of the predicate
  - any number of *formal arguments*, which can be *expressions* of any type
  - a *predicate body*, which is a *self-framing assertion* (in terms of the arguments)
- e.g. `positive_node` below defines a predicate consisting of permission to two fields, with a non-negative value for `val`

```
field next : Ref
field val : Int

predicate positive_node(n:Ref) {
  acc(n.val) && acc(n.next) && n.val >= 0
}
```

  - an instance of this predicate for parameter `x` is written `positive_node(x)`
  - there are no permissions to fields of null; `acc(n.val)` entails `n!=null`

Recursive Predicates

- Predicate declarations can be *recursive:* the predicate's body can include instances of the predicate being declared, e.g.

```
predicate list(start : Ref)
{
  acc(start.val) && acc(start.next) &&
    (start.next != null ==> list(start.next))
}
```

- Recursive predicate definitions are interpreted as *least-fixed points*
  - all *instances* of the predicate have *finite unfoldings* (in a *particular state*)
  - e.g. for `list(x)` to be true in a state, `x` and `x.next` *cannot be equal*
- An instance of the above predicate describes a *finite linked list*
  - includes permission to the `next` and `val` fields of all nodes in the list
  - Assertions `list(x)` denote a *statically-unbounded* number of permissions

Static Verification with Recursive Predicates?

- A program verifier cannot know statically how far to unfold predicates
  - `list(x)` ⊨ `acc(x.next) && (x.next!=null ==> list(x.next))`
    ⊨ `acc(x.next) && (x.next!=null ==> acc(x.next.next) &&`
    `(x.next.next!=null ==> list(x.next.next)) )` …
- The cause is *recursion* combined with *statically-unknown* information
  - the same reason that a verifier cannot simply unroll while loops etc.
- *Inductive reasoning* is frequently required, e.g. for *list-segments*:

```
predicate lseg(start : Ref, end:Ref)
{
  acc(start.val) && acc(start.next) &&
    (start.next != end ==> lseg(start.next,end))
}
```

  - Are `list(x)` and `lseg(x, null)` equivalent?
  - Does `lseg(x, y) && lseg(y, z)` entail `lseg(x, z)`?

Fold and Unfold Operations

- *Automatic* reasoning about recursive definitions is *undecidable*
- One common design is to treat predicate definitions *iso-recursively*
  - terminology originally comes from type systems with recursive types
  - A predicate instance is *not* treated as identical to the corresponding body
  - But, the two can be *explicitly exchanged*, via extra statements in the program
- An *unfold statement* exchanges a predicate instance for its body
  - e.g. `unfold list(x)` exchanges `list(x)` for `acc(x.val) && acc(x.next) && (x.next!=null ==> list(x.next))`
  - after such a statement, the field location `x.val` can be accessed
- A *fold statement* exchanges a predicate body for a predicate instance
  - e.g. `fold list(x)` exchanges `acc(x.val) && acc(x.next) && (x.next!=null ==> list(x.next))` for `list(x)`
  - note that after such a statement, the field location `x.val` cannot be accessed

Specifying Simple List Append

- We can now specify a simple list append:
- The `unfold` statement allows us to access the field `l1.next`
  - it also provides the necessary precondition for the recursive call to append
- The `fold` statement assembles the predicate instance for the postcondition

```
method append(l1: Ref, l2: Ref)
  requires list(l1) && list(l2)
  ensures list(l1)
{
  unfold list(l1)
  if(l1.next == null) {
    l1.next := l2
  } else {
    append(l1.next,l2)
  }
  fold list(l1)
}
```

- Is this code correct? What would happen if `l1` and `l2` were aliases (or the lists overlapped in some other way)?
  - In Viper, the && operator is actually *separating conjunction* (see last lecture)
  - This precondition implicitly requires that the lists are *completely disjoint* (*separating conjunction* along with *full permissions* in predicate definition)
  - Viper (as many similar tools) *doesn't support logical conjunction* as primitive

Adding Functional Specification

- So far, our append specification says nothing about the resulting list
  - e.g. the sequence of *values* should be the concatenation of the initial two
- We can introduce *abstractions* of the underlying heap data structure
  - e.g. abstract the list values as a *mathematical sequence* (of integers)
  - We can then write functional specifications *in terms of these abstractions*
- One way to introduce abstractions is *additional predicate arguments*
  - e.g. we add an `elems` parameter to our list, for the sequence of values stored
- Viper builds-in `Seq[T]`
  - `|s|` sequence length
  - `s[i]` sequence indexing
  - `Seq(v,…)` fixed-length sequence constructor
  - `++` sequence append, etc.

```
predicate listelems(start : Ref, elems: Seq[Int])
{
  acc(start.val) && acc(start.next) &&
    |elems| > 0 && elems[0]==start.val &&
    (start.next == null ?
      elems == Seq(start.val) :
      listelems(start.next, elems[1..]))
}
```

Specifying List Append with Predicate Parameters

- To use this new predicate, we must specify the sequence parameters
  - e.g. in precondition, we need a way to describe the *elements of list* `l1`
  - One way to model this is using extra method parameters:

```
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 `assert` is needed due to unreliable extensionality for built-in sequences

Heap-Dependent Functions

- The use of extra predicate parameters complicates the specifications
  - the parameter `elems` is *conceptually redundant*, since the relevant sequence could be *computed from the underlying heap data structure*
- Viper also supports *heap-dependent functions*: declarations consist of
  - A *function name*, declared *formal parameters* and a *return-type*
  - A *function body*, which is an *expression*
  - A *precondition*, which must be a *self-framing assertion*; it must provide *sufficient permissions to frame the body*
  - Optionally, a *postcondition*, which must be a *boolean expression*

```
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)
  ))
}
```

- Viper provides *unfolding expressions*
  - an expression `unfolding p(x) in e` instructs the verifier to temporarily *unfold predicate instance* `p(x)` during evaluation of `e`

Using Heap-Dependent Functions

- Function invocations are *expressions*, and can occur in any expression
  - However, their *preconditions must be true* wherever they are used
- In particular, functions can be used in *specifications*:
  - the `elems` function can be used to complement a `list` predicate

```
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)
}
```

- Note the additional precondition `l2 != null`
  - needed for verification of the if-branch, to know `elems(l1)= Seq(l1.val)++elems(l2)`
  - note: `l2 != null` is implied by the existing conjunct `list(l2)` *but only after unfolding it*
  - adding a "dummy" unfold/fold of `list(l2)` to the if-branch is an alternative

Recursive Definitions and Iterative Code

```
method appendit(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) {  // easy case
    l1.next := l2; fold list(l1)
  } else {
    var tmp : Ref := l1.next
    while(unfolding list(tmp) in tmp.next != null)
      invariant list(tmp) && ???
      invariant old(elems(l1)) == ??? ++ elems(tmp)
    {
      unfold list(tmp)
      tmp := tmp.next
    }
    unfold list(tmp)
    tmp.next := l2 … restore the list from l1?
  }
}
```

- Recursive predicates and functions can work well for recursive implementations
  - particularly when the recursion matches a *top-down data-structure traversal*
- They work less well for iterative implementations (i.e. loop invariants)
  - On the previous slide, the *fold statement* comes *after* the recursive calls: *not tail-recursive*
  - Loop invariants must also describe the *already-traversed partial data structure* (how?)

Partial Recursive Definitions and Loop Invariants

```
……
} else {
  var tmp : Ref := l1.next
  fold lseg(l1,l1.next)
  while(unfolding list(tmp) in tmp.next != null)
    invariant list(tmp) && lseg(l1,tmp)
    invariant old(elems(l1)) ==
      lsegelems(l1,tmp) ++ elems(tmp)
  {
    unfold list(tmp)
    var prev : Ref := tmp
    tmp := tmp.next
    addAtEnd(l1,prev) // extend lseg at end
  }
  unfold list(tmp)
  tmp.next := l2
  addAtEnd(l1,tmp) // extend lseg at end
  prependLseg(l1,l2) // lseg+list --> list
}
……
```

- Use list segment (`lseg`, slide 225) predicates for partial lists
  - also add an `lsegelems` function
- Loop invariant: use *remaining* `list` and the `lseg` *seen so far*
  - *permissions* to the data structure parts and *values* stored there
  - *re-establishing* the invariant is a problem: `lseg` must be extended at the *wrong end* (not a `fold`)
  - We *add a method* `addAtEnd` to perform this operation
  - Similarly, a new `prependLseg` method "glues" `lseg(l1,l2)` and `list(l2)` into `list(l1)`

Recursive Definitions: Summary

- Recursive predicates and functions can specify unbounded heap data
  - natural for implementations which perform *top-down, recursive traversals*
  - predicates can provide built-in *acyclicity* and *tree-like* guarantees easily
  - functions can *augment existing predicates* with additional information
  - functions can also *relate multiple data structures* (e.g. an `equals` function)
- The features have some disadvantages
  - Need for extra `fold` and `unfold` statements in the program text
  - Need for *extra methods* to handle more-complex predicate rearrangements
  - Iterative specifications tend to require extra specification machinery/effort
  - Not useful for random-access data (arrays) or structures with complex sharing
- We will look at one alternative mechanism: *quantified permissions*
  - describe properties over an unbounded set of individuals *point-wise*
  - e.g. an assertion for *each node in a graph*, or *each location in an array*

Cycle-detection in Linked Lists

- Consider the code opposite, which works on a *possibly-cyclic* list
  - specifying such data structures via recursive definitions is hard, especially since when *may or may not be cyclic*
- The code uses Viper's built-in sets
  - mathematical sets – similar to using an *immutable data-structure* in e.g. Scala
- The idea: add a set to the code to represent the data structure itself
  - *permission* to each node in the set
  - *knowledge* that the set is closed under `.next` dereferences (for the loop)

```
method isCyclic(root:Ref)
  returns (cyclic : Bool)
{
  var seen : Set[Ref] := Set(root)
// built-in Set[T] type
  var current : Ref := root

  while(current.next != null &&
        !(current.next in seen))
  {
    seen := seen union Set(current.next)
    current := current.next
  }
  cyclic := (current.next != null)
}
```

Quantified Permissions

- The key new feature we make use of is *quantified permissions*
  - we allow, e.g. `acc(x.f)` under a `forall x:Ref ::` quantifier
- Recall: first-order quantifiers can be thought of as a *possibly-infinite iterated conjunction*: $\forall x{:}T.A \;\equiv\; A[t_1/x] \wedge A[t_2/x] \wedge \ldots$
- Analogously, Viper's `forall` quantifiers can be thought of as a *possibly-infinite iterated separating conjunction*
  - `forall x:Ref ::` $A \;\equiv\; A[t_1/x] * A[t_2/x] * \ldots$
- For example, `forall x:Ref :: x in nodes ==> acc(x.next)` can be thought of as `acc(`n_1`.next) && acc(`n_2`.next) && …`
  - for each $n_1, n_2, \ldots$ in the set `nodes`
  - this allows us to access *some instantiations* of the quantifier, make use of the *permissions held*, and *frame information* about the all other instantiations

Cycle-detection with Quantified Permissions

```
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)
}
```

- The specification opposite uses *quantified permissions*
  - we focus here on expressing sufficient *permissions* for the field-accesses to be allowed
  - functional specifications *could be added* (exercise session)
- The extra `nodes` parameter represents the set of nodes making up the data structure
  - the `forall` assertions (which are each the same) specify permission to each such node
  - they also specify that the set is closed under `.next`

Generalised Quantified Permissions

- In fact, quantified permissions in Viper are more general than shown
  - when accessibility predicates `acc(e.f)` are used under `forall x` quantifiers, the receiver `e` does not necessary have to be the variable `x`
- In general, `acc(e.f)` can occur under a `forall x:T` quantifier if the mapping from instantiations of `x` to instantiations of `e` is *injective*
  - i.e., for all `y,z:T`, it must be true that $y \neq z \Rightarrow e[y/x] \neq e[z/x]$
  - this is a *well-definedness condition*, to be checked by the verifier
  - the motivation for the condition is technical; here we focus on examples
- Some examples of well-defined quantified permissions:
  - `forall x:Ref :: x in nodes ==> acc(x.next)`
    (using `x` as the receiver expression trivially satisfies injectivity)
  - `forall i:Int :: 0<=i<n ==> acc(f(i).val)`
    (if we can prove for any $0 \leq j \neq k < n$, that $f(j) \neq f(k)$, e.g. from an axiom)

Encoding Arrays in Viper

```
/* Encoding of integer arrays */

field val: Int // for integer arrays

domain Array {
  function loc(a: Array, i: Int): Ref
  function len(a: Array): Int
  function first(r: Ref): Array
  function second(r: Ref): Int

  axiom injectivity {
    forall a: Array, i: Int :: {loc(a, i)}
    first(loc(a, i)) == a && second(loc(a, i)) == i
  }

  axiom length_nonneg {
    forall a: Array :: len(a) >= 0
  }
}
```

- We can verify array programs using *quantified permissions*
  - we use a domain type `Array`
  - A function `loc` maps arrays and indices to a value of type `Ref`
  - we use fields of these `Ref`s to represent *array slots (locations)*
  - `loc(a,i).val` models `a[i]`
  - axiom guarantees injectivity of `loc`
- We can then represent permission to the array slots with an assertion:
  - `forall i:Int :: 0<=i && i<len(a) ==> acc(loc(a,i).val)`
  - similarly for sub-ranges of the array

Increment All Example

```
method incrementAll(a:Array)
  requires forall i:Int :: 0 <= i && i< len(a) ==> acc(loc(a,i).val)
  ensures forall i:Int :: 0 <= i && i< len(a) ==> acc(loc(a,i).val)
    && loc(a,i).val == old(loc(a,i).val) + 1
{
  var j:Int := 0
  while(j < len(a))
    invariant forall i:Int :: 0 <= i && i< len(a) ==>
      acc(loc(a,i).val) &&
      loc(a,i).val == old(loc(a,i).val) + (i<j ? 1 : 0)
    invariant 0 <= j && j <= len(a)
  {
    loc(a,j).val := loc(a,j).val + 1
    j := j + 1
  }
}
```

Aside: Triggering and Quantified Permissions

```
method triggeringProblem(a:Array)
 requires forall i:Int ::
 -2 <= i && i< len(a)-2 ==> acc(loc(a,i+2).val)
{
  var j:Int := 0
  if(j < len(a)) {
    loc(a,j).val := loc(a,j).val + 1 // fails:
    // Viper cannot prove we have permission
  }
}
```

- Just as for regular (pure) quantifiers, quantified permissions must be *appropriately instantiated*
  - e.g. when checking that we have permission to allow a field lookup
- Example opposite fails to verify:
  - precondition is, in principle equivalent to that used on the previous slide (expressed an an offset of 2)
  - The usage of *arithmetic expressions* (e.g. the extra addition term) here, however, means that Viper *doesn't find suitable triggers* for the quantifier
  - On the previous slide, `loc(a,i)` was a potential suitable trigger
  - Another possible option was `loc(a,i).val` : Viper allows *field lookups in triggers*
- Rewriting the quantifier to the form on the previous slide fixes the problem
  - Just as in Boogie, expressing quantifiers different ways can help/harm triggering

Unbounded Heap Data - Summary

- We've covered two main approaches to specifying unbounded data
- *Recursive definitions* work well for recursive implementations
  - Predicates enable *recursively-defined assertions*, including permissions
  - Explicit `fold` and `unfold` statements manage the recursion
  - Functions allow *recursively-defined expressions* to be used in specifications
  - These can be *heap-dependent*, using permissions from e.g. existing predicates
- *Quantified permissions* work well for point-wise specification of data
  - allow an *iterated form of separating conjunction* to be expressed
  - good for *random-access situations*, or structures with *cycles / sharing*
  - *more automatic* than recursive definitions in Viper (no fold/unfold analogue)
  - point-wise *functional specifications* fit well (e.g. "increment all" example)
  - functional specifications which *summarise multiple locations* are more cumbersome (e.g. specifying that cycle-detection really detects cycles)

Unbounded Heap Data – References

- Implicit Dynamic Frames:
  - *Specification and Automatic Verification of Frame Properties for Java-like Programs.* J. Smans. (PhD thesis) (2009)
  - *Implicit dynamic frames: Combining dynamic frames and separation logic.* J. Smans, B. Jacobs, and F. Piessens (2009)
  - *The Relationship Between Separation Logic and Implicit Dynamic Frames.* M. J. Parkinson and A. J. Summers (2012)
- 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)
  - *Verification condition generation for permission logics with abstract predicates and abstraction functions.* S. Heule, I. T. Kassios, P. Müller, A. J. Summers (2013)
- Quantified Permissions and Viper:
  - *Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution.* P. Müller, M. Schwerhoff, A. J. Summers. (2016)
  - *Viper: A Verification Infrastructure for Permission-Based Reasoning.* P. Müller, M. Schwerhoff, A. J. Summers. (2016)