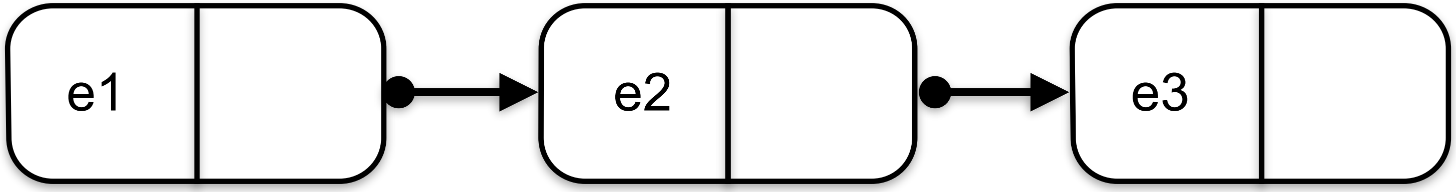
Selected Topics

Specification and Verification using ADT — Linked Data Structures

TECHNISCHE UNIVERSITÄT DARMSTADT

Software Engineering Group

Linked Data Structures




```
public class LinkedList {
  private int element;
  private LinkedList next;
  public int head() { … }
  public LinkedList tail() { … }
  public int get(int i) { … }
}
```

How to specify end-of-list?

- null: add /*@ **nullable** @*/ to next
- static unique dummy element referencing itself
- …

How to specify method

- head()?
  - e.g., **\result** == element
- tail()?
  - e.g., **\result** == next
- get(int)?
  - recursive specification (see: SimpleLinkedListRec.java)
  - need to express reachability

Expressing Reachability in Dynamic Logic

Predicate Symbol

reach: Heap × LocSet × Object × Object × int

Predefined (i.e., interpreted same by any interpretation function I)

I(reach)(h, $locs$, o, u, n) = tt iff.

there exist $o=o_0 \ldots o_n=u$ such that

$h(o_i,f) = o_{i+1}$, $o_i \notin D^{Null}$ (i=0..n-1) and $(o_i,f) \in locs$

with $h \in D^{Heap}$, $locs \in D^{LocSet}$, $n \in D^{int}$ and $o,o_i,u \in D^{Object}$

Reachable: Axiom

Logic Characterization of reach

$$\mathrm{reach}(h, locs, o, u, n) \leftrightarrow$$
$$(\ n \geq 0 \land \neg(o \doteq null) \land \neg(u \doteq null) \land ($$
$$(n \doteq 0 \land o \doteq u) \lor \exists \mathrm{Object}\ s;(\mathrm{reach}(h, locs, s, u, n\text{-}1) \land \mathrm{acc}(h, locs, s, o))\ )$$

where $\mathrm{acc}(h,locs,s,o) \leftrightarrow \exists \mathrm{Field}\ f;(\mathrm{singleton}(s,f) \subseteq locs \land \mathrm{select}(h,s,f) \doteq o)$

Reachability in Dynamic Logic

Examples

TECHNISCHE UNIVERSITÄT DARMSTADT

Are the following sequents valid?

- heap = store(store(store(heap, l1, next, l2), l3, next, l1) ==>
  reach(heap, allObjects(next), l3, l2, 2)

  Yes, in any state satisfying the antecedent, l2 is reachable from l3 in exactly two steps via locations (l3,f) and (l1,f) which are in the location set allObjects(next).

- heap = store(store(store(heap, l1, next, l2), l3, next, l1) ==>
  reach(heap, allObjects(next), l1, l2, 2)

  No. l2 is not reachable from l1 in exactly 2 steps using the specified locations in some (here actually in any) structure, which satisfies the antecedent.

- heap = store(store(store(heap, l1, next, l2), l3, next, l1) ==>
  reach(heap, singleton(l3, next), l1, l2, 1)

  No, although in any state satisfying the antecedent, l2 is reachable from l1 in exactly one step that is only the case via location (l1,next) which is not in the provided set of locations.

Software Engineering Group

Example: Specification of Acyclicity of a List

How to specify that a LinkedList l is acyclic in heap h?

$$
\begin{aligned}
&!\backslash\mathbf{exists}\ \mathrm{LinkedList}\ l2;\\
&\qquad (\backslash\mathbf{exists}\ \mathrm{int}\ d; \mathrm{reach}(\mathrm{h}, \mathrm{allObject}(\mathrm{next}), \mathrm{l}, l2, d)\\
&\qquad\qquad \wedge\ \backslash\mathbf{exists}\ \mathrm{int}\ n;\ (n{>}0 \wedge \mathrm{reach}(\mathrm{h}, \mathrm{allObject}(\mathrm{next}), l2, l2, n)))
\end{aligned}
$$

Exercise: Specify that two lists are disjoint.

Reachability in JML

\reach: \locSet × Object × Object × int

and

\reach: \locSet × Object × Object

where

\reach(l,o,u) := (**\exists** int n; n >= 0; **\reach**(l,o,u,n))

Both JML **\reach** variants have some oddities/hacks to specify location sets

\reach(first.next, o, u, i)

means (in DL)

reach(heap, allObjects(next), o, u, i)

[[On the slides we will most of the time use our DL syntax for location sets or short forms like {(o,f)} for singleton(o,f) etc.]]

Reachability in JML
Specification of a Single Linked List

See file:

SimpleLinkedList.java

Using Abstract Data Types as Abstractions
By Example "Finite Sequences"

TECHNISCHE UNIVERSITÄT DARMSTADT

```
interface List {
 //@ public model instance Object[] content;
 …
}
```

Avoid problems by using **abstract data types!**

What is the problem using an array to represent the content of a list?

- Java type: We have to deal with all the OO-problems like aliasing
- Already expressing that two lists have same contents is convoluted:
  - other.content == this.content
    (only expresses that array objects are the same and not their content)
  - instead:
    (this.content.length == other.content.length && (**\forall** int i; i>=0 &&
    i<this.content.length; this.content[i] == other.content[i]));

Software Engineering Group

Predefined Type: Seq

The basic constructor of Sequence ADT in DL is

$$\text{seqDef}\{\text{int } i;\}: \text{int} \times \text{int} \times \text{any}$$

(logic variable binding symbol)

$$val_{S,\beta}(\text{seqDef}\{\text{int } i;\}(le,ri,e)) = \begin{cases} \langle a_0, \ldots, a_{n-1} \rangle & \text{if } n = val_{S,\beta}(ri) - val_{S,\beta}(le) > 0 \\ & \text{and with} \\ & a_k = val_{S,\beta'}(e),\ \beta' = \beta[i / val_{S,\beta}(le) + k] \\ & \textit{(i.e., variable assignment } \beta' \textit{ identical to } \beta \textit{ except for } i \\ & \quad \textit{which has the specified value)} \\ \langle\rangle & \text{otherwise} \end{cases}$$

Examples: Comprehension

- seqDef{int x;}(0, 5, 1)
  - evaluates to the sequence <1,1,1,1,1>

- seqDef{int x;}(-4, 0, x)
  - evaluates to the sequence <-4,-3,-2,-1>

- seqDef{int x;}(a, b, null)
  - evaluates to <null, …, null> if $b > a$ holds

    <> otherwise

The Sequence Data Type Seq
Core Theory — Length and Getter

- seqLen: Seq → int *(the length of the sequence)*

  Axioms*: Let t be an arbitrary term.*

  - $\forall$Seq $s;$ seqLen$(s) \geq 0$
  - $\forall$int $le, ri;$ $((ri > le \;\rightarrow \text{seqLen}(\text{seqDef}\{\text{int } i;\}(le,ri,t)) \doteq ri - le)$
    $\wedge\ (ri \leq le \rightarrow \text{seqLen}(\text{seqDef}\{\text{int } i;\}(le,ri,t)) \doteq 0))$

- A::seqGet: Seq × int→A for any type A≤any
  (retrieves the n-th element of the given sequence and casts it to type A)

  - $\forall$int $le, ri, k;$ $(\ ((le \leq k \wedge k<ri) \rightarrow A::\text{seqGet}(\text{seqDef}\{\text{int } i;\}(le,ri,t), k) \doteq (A)\ t[i/le+k]\ )$
    $(\neg(le \leq k \wedge k<ri) \rightarrow A::\text{seqGet}(\text{seqDef}\{\text{int } i;\}(le,ri,t), k) \doteq (\text{A})\text{seqGetOutside}))$

- seqGetOutside: any
  (element retrieved by A::seqGet *if index was negative or greater-or-equal than* seqLen *of the sequence)*

Two sequences are equal iff they have equal arguments in the same order:

$$
\begin{array}{l}
\forall \mathrm{Seq}\ s1, s2;\ (\\
\qquad s1 \doteq s2 \leftrightarrow (\mathrm{seqLen}(s1) \doteq \mathrm{seqLen}(s2) \wedge \\
\qquad\qquad \forall \mathrm{int}\ k;\ (k \geq 0 \wedge k < \mathrm{seqLen}(s1) \rightarrow \mathrm{any::seqGet}(s1, k) \doteq \mathrm{any::seqGet}(s2, k))\ ) \\
)
\end{array}
$$

The Sequence Data Type
Definitional Extensions

- seqEmpty: Seq (*the empty sequence*)
- seqSingleton: any → Seq
  (sequence of length 1 with the given element as content)

  Example: seqSingleton(*1*) *(describes sequence <1>)*
- seqConcat: Seq × Seq → Seq *(concatenation of two sequences)*

  Example:
  seqConcat(seqSingleton(1), seqConcat(s, seqSingleton(1)))
- seqSub: Seq × int × int → Seq *(subsequence of the given sequence between indices given as 2nd argument and 3rd argument)*

Functions for the Sequence Data Type

- seqReverse *:* Seq → Seq
  (returns a sequence that is the reverse of the sequence given as 1st argument)
- seqIndexOf *:* Seq × any → int
  (returns 1st occurrence of the 2nd argument in the sequence specified as 1st argument otherwise -1)
- seqSwap *:* Seq × int × int → Seq
  (returns a sequence equal to the one given as 1st argument with the two elements whose indices are given as 2nd and 3rd argument swapped)
- seqRemove *:* Seq × int → Seq
  (returns a sequence equal to the given one except that the element at the specified index has been returned)
- *predicates for permutation properties etc.*

Using Sequences in JML

Ghost and model fields can be declared of type Seq

//@ **model \seq** content; //@ **ghost \seq** content;

JML	DL
\seq_get(*s*, *idx*) or *s*[*idx*]	any::seqGet($\mathcal{E}$(*s1*), $\mathcal{E}$(*idx*))
s.**length**	seqLen($\mathcal{E}$(s))
\seq_concat(*s1*,*s2*)	seqConcat($\mathcal{E}$(*s1*),$\mathcal{E}$(*s2*))
\dl_arr2seq(*array*)	array2seq(heap, $\mathcal{E}$(*array*))
\seq_def, \seq_empty, \seq_get, \seq_reverse, \seq_singleton, \seq_sub	…

Maps array to a finite sequence of same length, content, and order of elements

*(**s, s1, s2** JML expressions of type* \seq*; **idx** JML expression of type* int*; **array** a JML expression of array type;* heap *the global program variable referring to the current program heap)*

Specification of Lists Using Sequences

See file:

SimplifiedListSeq.java

Specifying an interface for lists using sequences:

See file:

List.java

Modular Specifications

Modularity

- Local reasoning
- Verification without involving whole program state

Did we achieve modularity?

- Method contracts instead of inlining to abstract from implementation
- Specification inheritance to ensure behavioural subtyping and thus to provide sound supertype abstraction

Modular Specifications

Open Problems — Specification of Assignable Clauses

TECHNISCHE UNIVERSITÄT DARMSTADT

But what about assignable clauses?

Classes must adhere to assignable clauses of their superclasses/interfaces

```
public interface List {
  //@ public instance model \seq theList;

  /*@ public normal_behavior
    @ ensures theList ==\seq_concat(\seq_singleton(elem), \old(theList));
    @ assignable ?
    @*/
  public void add (int elem);
```

Which locations are allowed to be changed?

- Omitting (same as \everything)
  - Flexible for implementing classes, but practically prevents use of contract in clients
- Union of assignable of all implementing classes
  - Need to know all implementing classes (contradicts open world assumption)
  - Work-around flavour

Software Engineering Group

Modular Specifications
Open Problems — Abstract Aliasing

TECHNISCHE UNIVERSITÄT DARMSTADT

Clients using the List interface

```
class Client {
  //@ public invariant \invariant_for(a) && \invariant_for(b);
  List a, b;
  /*@ normal_behavior
    @ requires a != b;
    @ ensures b.size() == \old(b.size());
    @*/
  void m() { a.add(23); }
}
```

- Not provable, if method add might change whole heap
- Still not provable when using more specific assignable
  - need to express that lists a and b do not share list elements
  - and that adding an element does not introduce sharing (!)

Dynamic Frames: Abstract Sets of Locations

Specify an **abstract set** Acc **of locations** called *dynamic frame* on which an

- invariant
- model field or
- method

might depend (or in other words might access) using

- model fields of type \locset (to specify the set of locations)
- **accessible** clauses (to frame an invariant, model field or method)

Dynamic Frames
Model Fields and Invariant

```
public interface List {
  //@ public model instance \locset footprint;
  //@ public accessible footprint: footprint;

  //@ public model instance \seq theList;
  //@ public accessible theList: footprint;

  //@ public invariant size() >= 0;
  //@ public accessible \inv: footprint;
  …
}
```

Model field of type \locset representing an abstract set of locations

Self framing of model field footprint, otherwise set of locations might depend on external location

Frame model field theList, i.e., all representations of the model field might only depend on the locations contained in footprint

Frame for the invariant

Dynamic Frames
Methods

```
public interface List {
  //@ public model instance \locset footprint;
  //@ …

  /*@ public normal_behavior
    @ ensures theList ==\seq_concat(\old(theList), \seq_singleton(elem));
    @ ensures size() == \old(size()) + 1;
    @ assignable footprint;
    @*/
  public void add (int elem);

  /*@ public normal_behavior
    @ ensures \result == theList.length;
    @ accessible footprint;
    @*/
  public /*@ pure @*/ int size ();
```

Might change only locations in footprint

Might access/depend only on locations in footprint

Dynamic Frames
Specifying Location Sets in JML*

JML expression of type \locset are translated to DL terms of type LocSet

JML	DL
\singleton(o.f), \singleton(a[i])	singleton($\mathcal{E}$(o),f), singleton($\mathcal{E}$(a),arr($\mathcal{E}$(i)))
o.*, a[*], a[i..j]	allFields($\mathcal{E}$(o)), allFields($\mathcal{E}$(a)), arrayRange($\mathcal{E}$(a),$\mathcal{E}$(i),$\mathcal{E}$(j))
\set_union(s1,s2), \set_minus(s1,s2), \intersect(s1,s2), \subset(s1,s2), \disjoint(s1,s2)	…
\reachLocs(s1, o, n) (resp. \reachLocs(s1, o))	*set of all locations reachable (in exactly* n *steps) from* o *using locations in location set* s1

*(**s1, s2** JML expressions of type \locset; **i,j** JML expression of type* int*;* **a** *JML expression of array type)*

Example: LinkedList

```
public interface List {
  //@ public model instance \locset footprint;
  //@ ...
}

public class LinkedList implements List {
  private /*@ spec_public @*/ int elem;
  private /*@ spec_public nullable @*/ LinkedList tail;
  //@ represents footprint = elem, \reachLocs(tail, this);

  ...
}
```

same as
\set_union(\singleton(this, elem),
\reachLocs(tail, this))

not really necessary as \reachLocs is reflexive

Note: Set \reachLocs(…) contains all locations of reachable objects.
Here: If u is reachable from this then the set contains the locations $\mathrm{allFields}(u)$

Modular Specifications

Abstract Aliasing — Revisited

TECHNISCHE UNIVERSITÄT DARMSTADT

Clients using the List interface

```
class Client {
   //@ public invariant \invariant_for(a) && \invariant_for(b);
   List a, b;
   /*@ normal_behavior
     @ requires a != b;
     @ requires \disjoint(a.footprint, b.footprint)
     @ ensures b.size() == \old(b.size());
     @*/
   void m() { a.add(23); }
}
```

Added that lists do not share locations

Can we now prove the specification of m() ?

No, add changes the footprint of a and might introduce sharing.

Software Engineering Group

Modular Specifications

Abstract Aliasing — Revisited

Clients using the List interface

```
class Client {
  //@ public invariant \invariant_for(a) && \invariant_for(b);
  List a, b;
  /*@ normal_behavior
    @ requires a != b;
    @ requires \disjoint(a.footprint, b.footprint)
    @ ensures b.size() == \old(b.size());
    @*/
  void m() { a.add(23); }
}
```

Can we now prove the specification of m() ?

```
public interface List {
  //@ public model instance \locset footprint;
  //@ ...

  /*@ public normal_behavior
    @ ensures theList ==\seq_concat(\seq_singleton(elem), \old(theList));
    @ ensures size() == \old(size()) + 1;
    @ ensures \new_elems_fresh(footprint);
    @ assignable footprint;
    @*/
  public void add (int elem);
```

ensures that any locations added to the footprint did not exist before method invocation

Modular Specifications

Abstract Aliasing — Revisited

TECHNISCHE UNIVERSITÄT DARMSTADT

Clients using the List interface

```
class Client {
   //@ public invariant \invariant_for(a) && \invariant_for(b);
   List a, b;
   /*@ normal_behavior
     @ requires a != b;
     @ requires \disjoint(a.footprint, b.footprint)
     @ ensures b.size() == \old(b.size());
     @*/
   void m() { a.add(23); }
}
```

Can we now prove the specification of m() ?

With enhanced specification of List. Now provable as disjointness is maintained.

Software Engineering Group

Fresh Elements

\new_elems_fresh(s) : Swinging Pivots Operator

- Only allowed in ensures
- s is a JML expression of type **\locset**
- **Meaning**: All locations in s which did where not contained in s in the pre-state must be fresh elements.

\fresh(s):

- Only allowed in ensures
- **s** is a JML expression of type **\locset**
- **Meaning:** All elements in s are fresh (did not exist in pre-state)