key-project.org

KeY-JML Cheat Sheet

- EXPRESSIONS -

Operators

JML extends the Java operators: `+ - * / % <= < > >= <: <:= == != ~ & ^ & | && || ==> <== <==> <=!=> ?: << >>` JML expressions must be side-effect free! New: `a <: b` (subtyping), `a ==> b` (implication), and `a <==> b` (equivalence).

Arithmetic Semantics

RED: Please write 3-4 lines about the current handling of arithmetic `\bigint` vs. `code_math` and `spec_math`

Functions

- `\dl_name(e)` — direct access to JavaDL functions
- `\invariant_for(o)` — class invariant of `o`
- `\old(x)` — value of `x` in the state before the current block
- `\pre(x)` — value of `x` in the state before the current method
- `\fresh(x)` — holds iff `x` was not allocated in the current method's prestate

Location Sets (Type: `\locset`)

Type `\locset` describes positions as *Object* × *Field* pairs on the heap. Useable in `assignable` clauses and ghost variables.

- `\locset(o.f, a[l..u], b[*])` — the set consisting of (o, f) and all entries in a between l and u (exclusive).
- `\intersect(x,y)` — $x \cap y$
- `\set_union(x,y)` — $x \cup y$
- `\set_minus(x,y)` — $x \setminus y$

Sequence (Type: `\seq`)

Mathematical data type of finite sequences.

- `\seq_empty` — empty sequence
- `\seq_concat(a, b)` — concatenation
- `(T)s[i]` — element access with cast to type `T`
- `s.length` — length of the sequence
- `\seq(e1,...,en)` — seq. constructor
- `s[i.. j]` — subsequence
- `\seq_def \bigint x; i; j; t` — binder form

Binders

Syntax: `(` *Q T v* `;` *guard* `;` *value* `)`

Quantifiers:

`\forall` — $\forall v : T.\ guard(v) \rightarrow value(v)$

`\exists` — $\exists v : T.\ guard(v) \wedge value(v)$

`\sum` — $\sum_{v:T \wedge guard(v)} value(v)$

`\product` — $\prod_{v:T \wedge guard(v)} value(v)$

`\num_of` — $\sum_{v:T \wedge guard(v)} 1$ Number of valid entries.

- CONSTRUCTS -

Modifiers

- `ghost` — declaration of spec-only fields assigned by `set` statements (see `JML Statements`)
- `model` — declaration of model fields and methods; these have no state of their own, but are coupled to a state by via a `represents`
- `nullable` — declaration of a type as nullable (the default being non-null)
- `helper` — helper methods neither require nor ensure the invariant
- `pure` — pure methods modify no existing objects

Class-level

- `invariant` — object invariant adhered to in every method's initial and terminal state, except `helper` methods
- `represents` — model field definition

Contracts

`behavior` = (`normal_behavior` + `exceptional_behavior`) defines the allowed clauses:

- `requires` — precondition
- `ensures` — postcondition; access return value using `\result`
- `assignable` — frame condition
- `measured_by` — termination witness
- `signals` — abnormal postcondition; access exception using `\exception`
- `signals_only` — allowed exceptions

Loop Invariants

Appear in JML comments before loops and have the following clauses:

- `loop_invariant` — inductive invariant formula
- `assignable` — frame condition (for whole loop, not single iteration)
- `decreases` — strong monotonic decreasing expression as a witness for termination

JML statements

- `//@ assert e;` or `//@ assume e;` adds a proof goal or assumption on computation path.
- `//@ set v = e;` assignment to a ghost variable

JML explained on Binary Search

```
/*@ private normal_behavior
  @   requires   (\exists int idx;
  @                 0 <= idx < a.length;
  @                 a[idx] == v);
  @   requires   (\forall int x, y;
  @                  0 <= x < y < a.length; a[x] <= a[y]);
  @   ensures  0 <= \result < a.length
  @     &&    a[\result] == v;
  @   assignable \nothing;
  @ also private exceptional_behavior
  @   requires    ! (\exists int idx;
  @                    0 <= idx < a.length; a[idx] == v);
  @   assignable \nothing;
  @   signals_only NoSuchElementException;
  @*/
private int binSearch(int[] a, int v) {
  int low = 0;
  int up = a.length;

  /*@ loop_invariant 0 <= low <= up <= a.length
    @    &&  (\forall int x; 0 <= x < low; a[x] != v);
    @    &&  (\forall int x; up <= x < a.length; a[x] != v);
    @ assignable \nothing;
    @ decreases up - low;
    @*/
  while (low < up) {
    int mid = low + ((up - low) / 2);
    if (v == a[mid])          { return mid; }
    else if (v < a[mid])      { up = mid; }
    else                      { low = mid + 1; }
  }
  throw new NoSuchElementException();
}
```

Contracts have an optional visibility modifier and behavior

`requires` introduces a pre-condition.

Support for multi-way comparison.

`ensures` introduces a post-condition.

`also` introduces a second contract.

`assignable` specifies that no heap locations are modified (`\nothing`).

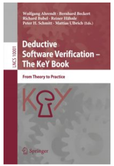
`signals_only` lists the allowed exceptions.

`loop_invariant` defines an inductive formula that holds in every iteration.

Termination witness for loop:
$0 \leq (up - low) < \texttt{\textbackslash old}(up - low)$

Reached with assume $low \geq up$ and invariant

Latest KeY Book



key-project.org/thebook2
support@key-project.org