Dynamic Frames in KeY

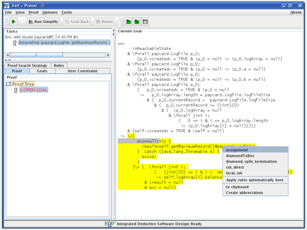
Benjamin Weiß

Karlsruhe, January 29, 2010

Overview

Context

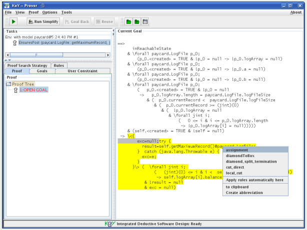
- Object-oriented programming (Java)
- Design by contract (JML)
- Deductive verification (KeY)
- Goal: **Modularity**
  "Proofs remain valid if program is changed"



Overview

Context

- Object-oriented programming (Java)
- Design by contract (JML)
- Deductive verification (KeY)
- Goal: **Modularity**
  "Proofs remain valid if program is changed"



This talk

- JML*: JML meets *dynamic frames* (Kassios, 2006)
- KeYHeap: KeY version which verifies JML*

Motivating Example

```
interface Cell {
  void setX(int value);
  int getX();
}
```

Example adapted from Smans et al. (2008)

Motivating Example

```
interface Cell {
  void setX(int value);
  int getX();
}
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);




}
```

Example adapted from Smans et al. (2008)

Motivating Example

```
interface Cell {
  void setX(int value);
  int getX();
}
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);

  Cell c2 = new CellImpl();
  c2.setX(10);


}
```

Example adapted from Smans et al. (2008)

Motivating Example

```
interface Cell {
  void setX(int value);
  int getX();
}
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);

  Cell c2 = new CellImpl();
  c2.setX(10);

  //@ assert c1.getX() == 5;
}
```

Example adapted from Smans et al. (2008)

Motivating Example

```
interface Cell {
  void setX(int value);
  int getX();
}
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);

  Cell c2 = new CellImpl();
  c2.setX(10);

  //@ assert c1.getX() == 5;
}
```

How to verify the assertion *without* looking into `CellImpl`?

Example adapted from Smans et al. (2008)

Specifying Cell

```
interface Cell {



  void setX(int value);


  int getX();
}
```

Specifying Cell

```
interface Cell {
  //@ model int m;


  void setX(int value);


  int getX();
}
```

Specifying Cell

```
interface Cell {
  //@ model int m;


  void setX(int value);


  int getX();
}
```

Model fields

- abstractions of actual program state
- related to Hoare's *abstract variables* (1972), notions of *refinement*

Specifying Cell

```
interface Cell {
  //@ model int m;

  //@ ensures m == value;
  void setX(int value);


  int getX();
}
```

Model fields

- abstractions of actual program state
- related to Hoare's *abstract variables* (1972), notions of *refinement*

Specifying Cell

```
interface Cell {
  //@ model int m;

  //@ ensures m == value;
  void setX(int value);

  //@ ensures \result == m;
  int getX();
}
```

Model fields

- abstractions of actual program state
- related to Hoare's *abstract variables* (1972), notions of *refinement*

Specifying Cell

```
interface Cell {
  //@ model int m;

  //@ ensures m == value;
  void setX(int value);

  //@ ensures \result == m;
  int getX();
}

class CellImpl implements Cell {
  private int x;
}
```

Model fields

- abstractions of actual program state
- related to Hoare's *abstract variables* (1972), notions of *refinement*

Specifying Cell

```
interface Cell {
  //@ model int m;

  //@ ensures m == value;
  void setX(int value);

  //@ ensures \result == m;
  int getX();
}

class CellImpl implements Cell {
  private int x; //@ represents m = x;
}
```

Model fields

- abstractions of actual program state
- related to Hoare's *abstract variables* (1972), notions of *refinement*

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);

  Cell c2 = new CellImpl();
  c2.setX(10);

  //@ assert c1.getX() == 5;
}
```

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                 //c1.m==5

  Cell c2 = new CellImpl();
  c2.setX(10);

  //@ assert c1.getX() == 5;
}
```

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                  //c1.m==5

  Cell c2 = new CellImpl();
  c2.setX(10);

  //@ assert c1.getX() == 5;  //still c1.m==5 ?
}
```

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                 //c1.m==5

  Cell c2 = new CellImpl();   //unknown state change
  c2.setX(10);                //unknown state change

  //@ assert c1.getX() == 5;  //still c1.m==5 ?
}
```

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                  //c1.m==5

  Cell c2 = new CellImpl();    //unknown state change
  c2.setX(10);                 //unknown state change

  //@ assert c1.getX() == 5;   //still c1.m==5 ?
}
```

$\rightsquigarrow$ Need to limit which locations may be changed by the methods

```
interface Cell {
  //@ model int m;



  //@ ensures m == value;
  void setX(int value);

  //@ ensures \result == m;
  int getX();
}
```

Modifies clauses for Cell

```
interface Cell {
  //@ model int m;



  //@ ensures m == value;
  void setX(int value);

  //@ ensures \result == m;
  /*@pure@*/ int getX();
}
```

Modifies clauses for Cell

```
interface Cell {
  //@ model int m;
  //@ model \locset footprint;  //a "dynamic frame"


  //@ ensures m == value;
  void setX(int value);

  //@ ensures \result == m;
  /*@pure@*/ int getX();
}
```

Modifies clauses for Cell

```
interface Cell {
  //@ model int m;
  //@ model \locset footprint;  //a "dynamic frame"

  //@ assignable footprint;
  //@ ensures m == value;
  void setX(int value);

  //@ ensures \result == m;
  /*@pure@*/ int getX();
}
```

Modifies clauses for Cell

```
interface Cell {
  //@ model int m;
  //@ model \locset footprint;  //a "dynamic frame"

  //@ assignable footprint;
  //@ ensures m == value;
  void setX(int value);

  //@ ensures \result == m;
  /*@pure@*/ int getX();
}

class CellImpl implements Cell {
  private int x; //@ represents m = x;

  public CellImpl() {}
}
```

```
interface Cell {
  //@ model int m;
  //@ model \locset footprint;  //a "dynamic frame"

  //@ assignable footprint;
  //@ ensures m == value;
  void setX(int value);

  //@ ensures \result == m;
  /*@pure@*/ int getX();
}

class CellImpl implements Cell {
  private int x; //@ represents m = x;
                 //@ represents footprint = \singleton(x);
  public CellImpl() {}
}
```

Modifies clauses for `Cell`

```
interface Cell {
  //@ model int m;
  //@ model \locset footprint;  //a "dynamic frame"

  //@ assignable footprint;
  //@ ensures m == value;
  void setX(int value);

  //@ ensures \result == m;
  /*@pure@*/ int getX();
}

class CellImpl implements Cell {
  private int x; //@ represents m = x;
                 //@ represents footprint = \singleton(x);
  public /*@pure@*/ CellImpl() {}
}
```

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                    //c1.m==5

  Cell c2 = new CellImpl();
  c2.setX(10);


  //@ assert c1.getX() == 5;   //still c1.m==5 ?
}
```

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                   //c1.m==5

  Cell c2 = new CellImpl();     //changes "nothing"
  c2.setX(10);


  //@ assert c1.getX() == 5;    //still c1.m==5 ?
}
```

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                  //c1.m==5

  Cell c2 = new CellImpl();    //changes "nothing"
  c2.setX(10);                 /*changes elements of
                                 c2.footprint*/

  //@ assert c1.getX() == 5;   //still c1.m==5 ?
}
```

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                  //c1.m==5

  Cell c2 = new CellImpl();    //changes "nothing"
  c2.setX(10);                 /*changes elements of
                                 c2.footprint*/

  //@ assert c1.getX() == 5;   //still c1.m==5 ?
}
```

⇝ Need knowledge about *dependencies* of `c1.m`

```
interface Cell {
  //@ model int m;
  //@ model \locset footprint;


}
```

Depends clauses for Cell

```
interface Cell {
  //@ model int m;
  //@ model \locset footprint;
  //@ accessible m: footprint;

}
```

Depends clauses for `Cell`

```
interface Cell {
  //@ model int m;
  //@ model \locset footprint;
  //@ accessible m: footprint;

}
```

Depends clause "`accessible m: s`"

If the values of the locations in `s` do not change,
then `m` does not change

Depends clauses for Cell

```
interface Cell {
  //@ model int m;
  //@ model \locset footprint;
  //@ accessible m: footprint;

}

class CellImpl implements Cell {
  private int x; //@ represents m = x;
                 //@ represents footprint = \singleton(x);
}
```

Depends clause "`accessible m: s`"

If the values of the locations in s do not change,
then m does not change

Depends clauses for `Cell`

```
interface Cell {
  //@ model int m;
  //@ model \locset footprint;
  //@ accessible m: footprint;
  //@ accessible footprint: footprint;
}

class CellImpl implements Cell {
  private int x; //@ represents m = x;
                 //@ represents footprint = \singleton(x);
}
```

Depends clause "`accessible m: s`"

If the values of the locations in s do not change,
then m does not change

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                 //c1.m==5

  Cell c2 = new CellImpl();   //changes "nothing"
  c2.setX(10);                /*changes elements of
                                c2.footprint*/

  //@ assert c1.getX() == 5;  //still c1.m==5 ?
}
```

How to verify the assertion?

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                  //c1.m==5

  Cell c2 = new CellImpl();    //changes "nothing"
  c2.setX(10);                 /*changes elements of
                                 c2.footprint*/

  //@ assert c1.getX() == 5;   //still c1.m==5 ?
}
```

$\leadsto$ Need to know: $\texttt{c1.footprint} \cap \texttt{c2.footprint} = \emptyset$

Specifying footprint

```
class CellImpl implements Cell {
  private int x; //@ represents m = x;
                 //@ represents footprint = \singleton(x);


  public /*@pure@*/ CellImpl() {}
}
```

Specifying footprint

```
class CellImpl implements Cell {
  private int x; //@ represents m = x;
                 //@ represents footprint = \singleton(x);

  //@ ensures \fresh(footprint);
  public /*@pure@*/ CellImpl() {}
}
```

Specifying footprint

```
class CellImpl implements Cell {
  private int x; //@ represents m = x;
                 //@ represents footprint = \singleton(x);

  //@ ensures \fresh(footprint);
  public /*@pure@*/ CellImpl() {}
}
```

\fresh(s)

- usable in postconditions
- true iff all members of s belong to freshly created objects

Specifying footprint (cont.)

```
interface Cell {
  //@ assignable footprint;
  //@ ensures m == value;

  void setX(int value);
}
```

Specifying footprint (cont.)

```
interface Cell {
  //@ assignable footprint;
  //@ ensures m == value;
  //@ ensures \new_elems_fresh(footprint);
  void setX(int value);
}
```

Specifying footprint (cont.)

```
interface Cell {
  //@ assignable footprint;
  //@ ensures m == value;
  //@ ensures \new_elems_fresh(footprint);
  void setX(int value);
}
```

\new_elems_fresh(s)

- usable in postconditions
- true iff all members of s either
  - belong to freshly created objects, or
  - have already been in s before

We can verify the assertion!

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                   //c1.m==5

  Cell c2 = new CellImpl();     //creates fresh c2.footprint
  c2.setX(10);



  //@ assert c1.getX() == 5;    //still c1.m==5 ?


}
```

We can verify the assertion!

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                    //c1.m==5

  Cell c2 = new CellImpl();      //creates fresh c2.footprint
  c2.setX(10);                   /*changes elements of
                                   c2.footprint, but never
                                   elements of c1.footprint*/

  //@ assert c1.getX() == 5;     //still c1.m==5 ?


}
```

We can verify the assertion!

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                   //c1.m==5

  Cell c2 = new CellImpl();     //creates fresh c2.footprint
  c2.setX(10);                  /*changes elements of
                                  c2.footprint, but never
                                  elements of c1.footprint*/

  //@ assert c1.getX() == 5;    //still c1.m==5 ?
                                //Yes.

}
```

We can verify the assertion!

```
void m() {
  Cell c1 = new CellImpl();
  c1.setX(5);                   //c1.m==5

  Cell c2 = new CellImpl();     //creates fresh c2.footprint
  c2.setX(10);                  /*changes elements of
                                  c2.footprint, but never
                                  elements of c1.footprint*/

  //@ assert c1.getX() == 5;    //still c1.m==5 ?
                                //Yes.

}
```

independent of private data, method bodies & represents clauses

JML*: Summary

- Proper type **`\locset`** instead of *data groups*

JML*: Summary

- Proper type `\locset` instead of *data groups*
- "Dynamic frames": model fields of type `\locset`

JML*: Summary

- Proper type **`\locset`** instead of *data groups*
- "Dynamic frames": model fields of type **`\locset`**
- Explicit specifications for dynamic frames (**`\fresh`**, **`\new_elems_fresh`**, **`\subset`**, **`\disjoint`**, ...)

JML*: Summary

- Proper type **`\locset`** instead of *data groups*
- "Dynamic frames": model fields of type **`\locset`**
- Explicit specifications for dynamic frames (**`\fresh`**, **`\new_elems_fresh`**, **`\subset`**, **`\disjoint`**, ...)
- Depends clauses for model fields

JML*: Summary

- Proper type **`\locset`** instead of *data groups*
- "Dynamic frames": model fields of type **`\locset`**
- Explicit specifications for dynamic frames (**`\fresh`**, **`\new_elems_fresh`**, **`\subset`**, **`\disjoint`**, ...)
- Depends clauses for model fields
- Different handling of object invariants:
  - No *visible state semantics* (complicated, non-modular)

JML*: Summary

- Proper type **`\locset`** instead of *data groups*
- "Dynamic frames": model fields of type **`\locset`**
- Explicit specifications for dynamic frames (**`\fresh`**, **`\new_elems_fresh`**, **`\subset`**, **`\disjoint`**, ...)
- Depends clauses for model fields
- Different handling of object invariants:
  - No *visible state semantics* (complicated, non-modular)
  - Instead reduce to model fields and pre-/postconditions:
    - built-in model field **`\inv`**
    - use `o.`**`\inv`** as needed
    - give depends clauses for **`\inv`**

JML*: Summary

- Proper type **`\locset`** instead of *data groups*
- "Dynamic frames": model fields of type **`\locset`**
- Explicit specifications for dynamic frames (**`\fresh`**, **`\new_elems_fresh`**, **`\subset`**, **`\disjoint`**, ...)
- Depends clauses for model fields
- Different handling of object invariants:
  - No *visible state semantics* (complicated, non-modular)
  - Instead reduce to model fields and pre-/postconditions:
    - built-in model field **`\inv`**
    - use `o.`**`\inv`** as needed
    - give depends clauses for **`\inv`**
- Nesting of dynamic frames $\hat{=}$ ownership hierarchy

JML*: Summary

- Proper type `\locset` instead of *data groups*
- "Dynamic frames": model fields of type `\locset`
- Explicit specifications for dynamic frames (`\fresh`, `\new_elems_fresh`, `\subset`, `\disjoint`, ...)
- Depends clauses for model fields
- Different handling of object invariants:
  - No *visible state semantics* (complicated, non-modular)
  - Instead reduce to model fields and pre-/postconditions:
    - built-in model field `\inv`
    - use `o.\inv` as needed
    - give depends clauses for `\inv`
- Nesting of dynamic frames $\hat{=}$ ownership hierarchy
- Dynamic frames can handle cases that ownership cannot

JML*: Summary

- Proper type **`\locset`** instead of *data groups*
- "Dynamic frames": model fields of type **`\locset`**
- Explicit specifications for dynamic frames (**`\fresh`**, **`\new_elems_fresh`**, **`\subset`**, **`\disjoint`**, ...)
- Depends clauses for model fields
- Different handling of object invariants:
  - No *visible state semantics* (complicated, non-modular)
  - Instead reduce to model fields and pre-/postconditions:
    - built-in model field **`\inv`**
    - use `o.`**`\inv`** as needed
    - give depends clauses for **`\inv`**
- Nesting of dynamic frames $\hat{=}$ ownership hierarchy
- Dynamic frames can handle cases that ownership cannot
- Price: verbose specifications

KeY

Verification process

Java + JML $\longrightarrow$ JAVA DL sequent $\longrightarrow$ proof

Verification process

Java + JML $\longrightarrow$ JAVA DL sequent $\longrightarrow$ proof

Example proof

$$\Rightarrow [\texttt{if (x<y) max=y; }\textbf{else}\texttt{ max=x;}](\texttt{max} \geq \texttt{x})$$

Verification process

Java + JML $\longrightarrow$ Java DL sequent $\longrightarrow$ proof

Example proof

$$\Rightarrow [\texttt{if (x<y) max=y; else max=x;}](\texttt{max} \geq \texttt{x})$$

conditional

$$\texttt{x} < \texttt{y} \Rightarrow [\texttt{max=y;}](\texttt{max} \geq \texttt{x}) \qquad \texttt{x} \geq \texttt{y} \Rightarrow [\texttt{max=x;}](\texttt{max} \geq \texttt{x})$$

Verification process

Java + JML → Java DL sequent → proof

Example proof

$$\Rightarrow [\texttt{if (x<y) max=y; }\textbf{\texttt{else}}\texttt{ max=x;}](\texttt{max} \geq \texttt{x})$$

conditional

$$\texttt{x} < \texttt{y} \Rightarrow [\texttt{max=y;}](\texttt{max} \geq \texttt{x}) \qquad \texttt{x} \geq \texttt{y} \Rightarrow [\texttt{max=x;}](\texttt{max} \geq \texttt{x})$$

assignment

$$\texttt{x} < \texttt{y} \Rightarrow \{\texttt{max} := \texttt{y}\}(\texttt{max} \geq \texttt{x})$$

Verification process

Java + JML → JAVA DL sequent → proof

Example proof

$$\Rightarrow [\texttt{if (x<y) max=y; else max=x;}](\texttt{max} \geq \texttt{x})$$

conditional

$$\texttt{x} < \texttt{y} \Rightarrow [\texttt{max=y;}](\texttt{max} \geq \texttt{x}) \qquad \texttt{x} \geq \texttt{y} \Rightarrow [\texttt{max=x;}](\texttt{max} \geq \texttt{x})$$

assignment

$$\texttt{x} < \texttt{y} \Rightarrow \{\texttt{max} := \texttt{y}\}(\texttt{max} \geq \texttt{x})$$

update application

$$\texttt{x} < \texttt{y} \Rightarrow \texttt{y} \geq \texttt{x}$$

Verification process

Java + JML $\longrightarrow$ JAVA DL sequent $\longrightarrow$ proof

Example proof

$$\Rightarrow [\texttt{if (x<y) max=y; else max=x;}](\texttt{max} \geq \texttt{x})$$

conditional

$$\texttt{x} < \texttt{y} \Rightarrow [\texttt{max=y;}](\texttt{max} \geq \texttt{x}) \qquad \texttt{x} \geq \texttt{y} \Rightarrow [\texttt{max=x;}](\texttt{max} \geq \texttt{x})$$

assignment

$$\texttt{x} < \texttt{y} \Rightarrow \{\texttt{max} := \texttt{y}\}(\texttt{max} \geq \texttt{x})$$

update application

$$\texttt{x} < \texttt{y} \Rightarrow \texttt{y} \geq \texttt{x}$$

arithmetic

$$*$$

Verification process

Java + JML $\rightarrow$ Java DL sequent $\rightarrow$ proof

Example proof

$$\Rightarrow [\mathtt{if\,(x<y)\ max=y;\ else\ max=x;}](\mathtt{max} \geq \mathtt{x})$$

conditional

$$\mathtt{x} < \mathtt{y} \Rightarrow [\mathtt{max=y;}](\mathtt{max} \geq \mathtt{x}) \qquad \mathtt{x} \geq \mathtt{y} \Rightarrow [\mathtt{max=x;}](\mathtt{max} \geq \mathtt{x})$$

assignment / assignment

$$\mathtt{x} < \mathtt{y} \Rightarrow \{\mathtt{max} := \mathtt{y}\}(\mathtt{max} \geq \mathtt{x}) \qquad \mathtt{x} \geq \mathtt{y} \Rightarrow \{\mathtt{max} := \mathtt{x}\}(\mathtt{max} \geq \mathtt{x})$$

update application

$$\mathtt{x} < \mathtt{y} \Rightarrow \mathtt{y} \geq \mathtt{x}$$

arithmetic

$$*$$

Verification process

Java + JML $\longrightarrow$ Java DL sequent $\longrightarrow$ proof

Example proof

$$\Rightarrow [\texttt{if (x<y) max=y; else max=x;}](\texttt{max} \geq \texttt{x})$$

conditional

$$\texttt{x} < \texttt{y} \Rightarrow [\texttt{max=y;}](\texttt{max} \geq \texttt{x}) \qquad \texttt{x} \geq \texttt{y} \Rightarrow [\texttt{max=x;}](\texttt{max} \geq \texttt{x})$$

assignment / assignment

$$\texttt{x} < \texttt{y} \Rightarrow \{\texttt{max} := \texttt{y}\}(\texttt{max} \geq \texttt{x}) \qquad \texttt{x} \geq \texttt{y} \Rightarrow \{\texttt{max} := \texttt{x}\}(\texttt{max} \geq \texttt{x})$$

update application / update application

$$\texttt{x} < \texttt{y} \Rightarrow \texttt{y} \geq \texttt{x} \qquad \texttt{x} \geq \texttt{y} \Rightarrow \texttt{x} \geq \texttt{x}$$

arithmetic

$$*$$

KeY

Verification process

Java + JML $\rightarrow$ JAVA DL sequent $\rightarrow$ proof

Example proof

$$\Rightarrow [\texttt{if (x<y) max=y; else max=x;}](\texttt{max} \geq \texttt{x})$$

conditional

$$\texttt{x} < \texttt{y} \Rightarrow [\texttt{max=y;}](\texttt{max} \geq \texttt{x}) \qquad \texttt{x} \geq \texttt{y} \Rightarrow [\texttt{max=x;}](\texttt{max} \geq \texttt{x})$$

assignment / assignment

$$\texttt{x} < \texttt{y} \Rightarrow \{\texttt{max} := \texttt{y}\}(\texttt{max} \geq \texttt{x}) \qquad \texttt{x} \geq \texttt{y} \Rightarrow \{\texttt{max} := \texttt{x}\}(\texttt{max} \geq \texttt{x})$$

update application / update application

$$\texttt{x} < \texttt{y} \Rightarrow \texttt{y} \geq \texttt{x} \qquad \texttt{x} \geq \texttt{y} \Rightarrow \texttt{x} \geq \texttt{x}$$

arithmetic / arithmetic

$$* \qquad *$$

Modelling the Java Heap

Heaps as non-rigid functions

- Fields are non-rigid function symbols `f` : $\mathit{Object} \longrightarrow A$

Modelling the Java Heap

Heaps as non-rigid functions

- Fields are non-rigid function symbols `f` : $\mathit{Object} \longrightarrow A$
- `o.f` means `f(o)`

Modelling the Java Heap

Heaps as non-rigid functions

- Fields are non-rigid function symbols $\mathtt{f} : \mathit{Object} \rightarrow A$
- `o.f` means `f(o)`
- $\{\mathtt{f}(\mathtt{o}) := t\}$

Modelling the Java Heap

Heaps as non-rigid functions

- Fields are non-rigid function symbols $\mathtt{f} : \mathit{Object} \to A$
- `o.f` means $\mathtt{f}(\mathtt{o})$
- $\{\mathtt{f}(\mathtt{o}) := t\}$

"Explicit" heaps (theory of arrays)

- Fields are constant symbols $\mathtt{f} : \mathit{Field}$

Modelling the Java Heap

Heaps as non-rigid functions

- Fields are non-rigid function symbols $\mathtt{f} : \mathit{Object} \rightarrow A$
- $\mathtt{o.f}$ means $\mathtt{f(o)}$
- $\{\mathtt{f(o)} := t\}$

"Explicit" heaps (theory of arrays)

- Fields are constant symbols $\mathtt{f} : \mathit{Field}$
- Global program variable $\mathtt{H} : \mathit{Heap}$

Modelling the Java Heap

Heaps as non-rigid functions

- Fields are non-rigid function symbols $\mathtt{f} : \mathit{Object} \to A$
- $\mathtt{o.f}$ means $\mathtt{f}(\mathtt{o})$
- $\{\mathtt{f}(\mathtt{o}) := t\}$

"Explicit" heaps (theory of arrays)

- Fields are constant symbols $\mathtt{f} : \mathit{Field}$
- Global program variable $\mathtt{H} : \mathit{Heap}$
- $\mathtt{o.f}$ means $\mathit{select}(\mathtt{H}, \mathtt{o}, \mathtt{f})$

Modelling the Java Heap

Heaps as non-rigid functions

- Fields are non-rigid function symbols $\mathtt{f} : \mathit{Object} \to A$
- $\mathtt{o.f}$ means $\mathtt{f}(\mathtt{o})$
- $\{\mathtt{f}(\mathtt{o}) := t\}$

"Explicit" heaps (theory of arrays)

- Fields are constant symbols $\mathtt{f} : \mathit{Field}$
- Global program variable $\mathtt{H} : \mathit{Heap}$
- $\mathtt{o.f}$ means $\mathit{select}(\mathtt{H}, \mathtt{o}, \mathtt{f})$
- $\{\mathtt{H} := \mathit{store}(\mathtt{H}, \mathtt{o}, \mathtt{f}, t)\}$

Modelling the Java Heap

Heaps as non-rigid functions

- Fields are non-rigid function symbols $\mathtt{f} : \mathit{Object} \to A$
- $\mathtt{o.f}$ means $\mathtt{f}(\mathtt{o})$
- $\{\mathtt{f}(\mathtt{o}) := t\}$

"Explicit" heaps (theory of arrays)

- Fields are constant symbols $\mathtt{f} : \mathit{Field}$
- Global program variable $\mathtt{H} : \mathit{Heap}$
- $\mathtt{o.f}$ means $\mathit{select}(\mathtt{H}, \mathtt{o}, \mathtt{f})$
- $\{\mathtt{H} := \mathit{store}(\mathtt{H}, \mathtt{o}, \mathtt{f}, t)\}$
- $\mathit{select}(\mathit{store}(h, o, f, x), o', f') = \begin{cases} x & \text{if } o = o', f = f' \\ \mathit{select}(h, o', f') & \text{otherwise} \end{cases}$

Modelling the Java Heap

Heaps as non-rigid functions

- Fields are non-rigid function symbols $\mathtt{f} : \mathit{Object} \to A$
- $\mathtt{o.f}$ means $\mathtt{f(o)}$
- $\{\mathtt{f(o)} := t\}$

"Explicit" heaps (theory of arrays)

- Fields are constant symbols $\mathtt{f} : \mathit{Field}$
- Global program variable $\mathtt{H} : \mathit{Heap}$
- $\mathtt{o.f}$ means $\mathit{select}(\mathtt{H}, \mathtt{o}, \mathtt{f})$
- $\{\mathtt{H} := \mathit{store}(\mathtt{H}, \mathtt{o}, \mathtt{f}, t)\}$
- $\mathit{select}(\mathit{store}(h, o, f, x), o', f') = \begin{cases} x & \text{if } o = o', f = f' \\ \mathit{select}(h, o', f') & \text{otherwise} \end{cases}$

For dynamic frames, an explicit modelling has clear advantages

Dynamic frames in JAVA DL

Sorts

- *Heap*, *Object*, *Field*

Dynamic frames in Java DL

Sorts

- *Heap*, *Object*, *Field*
- $\mathit{LocSet} \triangleq 2^{\mathit{Object} \times \mathit{Field}}$

Dynamic frames in JAVA DL

Sorts

- *Heap*, *Object*, *Field*
- $LocSet \triangleq 2^{Object \times Field}$

Symbols

- *select*, *store*

Dynamic frames in Java DL

Sorts

- $Heap$, $Object$, $Field$
- $LocSet \mathrel{\hat{=}} 2^{Object \times Field}$

Symbols

- $select$, $store$
- $\dot{\in}$, $\dot{\subseteq}$, $\dot{\cap}$, $\dot{\emptyset}$, $everything$

Dynamic frames in JAVA DL

Sorts

- $\mathit{Heap}, \mathit{Object}, \mathit{Field}$
- $\mathit{LocSet} \triangleq 2^{\mathit{Object} \times \mathit{Field}}$

Symbols

- $\mathit{select}, \mathit{store}$
- $\dot{\in}, \dot{\subseteq}, \dot{\cap}, \dot{\emptyset}, \mathit{everything}$
- $\mathit{created} : \mathit{Field}$

Dynamic frames in JAVA DL

Sorts

- $Heap$, $Object$, $Field$
- $LocSet \mathrel{\hat{=}} 2^{Object \times Field}$

Symbols

- $select$, $store$
- $\dot{\in}$, $\dot{\subseteq}$, $\dot{\cap}$, $\dot{\emptyset}$, $everything$
- $created : Field$
- $anon : Heap \times LocSet \times Heap \rightarrow Heap$

Dynamic frames in JAVA DL

Sorts

- Heap, Object, Field
- $\mathit{LocSet} \mathrel{\hat{=}} 2^{\mathit{Object} \times \mathit{Field}}$

Symbols

- $\mathit{select}, \mathit{store}$
- $\dot{\in}, \dot{\subseteq}, \dot{\cap}, \dot{\emptyset}, \mathit{everything}$
- $\mathit{created} : \mathit{Field}$
- $\mathit{anon} : \mathit{Heap} \times \mathit{LocSet} \times \mathit{Heap} \to \mathit{Heap}$

Rules

- $\mathit{select}/\mathit{store}$, set theory

Dynamic frames in JAVA DL

Sorts

- *Heap*, *Object*, *Field*
- $LocSet \mathrel{\hat{=}} 2^{Object \times Field}$

Symbols

- *select*, *store*
- $\dot{\in}, \dot{\subseteq}, \dot{\cap}, \dot{\emptyset}$, *everything*
- $created : Field$
- $anon : Heap \times LocSet \times Heap \rightarrow Heap$

Rules

- *select*/*store*, set theory
- $select(anon(h, s, h'), o, f) = \begin{cases} select(h', o, f) & \text{if } (o, f) \dot{\in} s \\ select(h, o, f) & \text{otherwise} \end{cases}$

Dynamic frames in Java DL

Sorts

- $Heap$, $Object$, $Field$
- $LocSet \triangleq 2^{Object \times Field}$

Symbols

- $select$, $store$
- $\dot{\in}, \dot{\subseteq}, \dot{\cap}, \dot{\emptyset}$, $everything$
- $created : Field$
- $anon : Heap \times LocSet \times Heap \to Heap$

Rules

- $select/store$, set theory
- $select(anon(h, s, h'), o, f) = \begin{cases} select(h', o, f) & \text{if } (o, f) \dot{\in} s, f \dot{\neq} created \\ select(h, o, f) & \text{otherwise} \end{cases}$

Dynamic frames in JAVA DL

Sorts

- $Heap$, $Object$, $Field$
- $LocSet \triangleq 2^{Object \times Field}$

Symbols

- $select$, $store$
- $\dot{\in}$, $\dot{\subseteq}$, $\dot{\cap}$, $\dot{\emptyset}$, $everything$
- $created : Field$
- $anon : Heap \times LocSet \times Heap \to Heap$

Rules

- $select / store$, set theory
- $select(anon(h, s, h'), o, f) = \begin{cases} select(h', o, f) & \text{if } (o, f) \dot{\in} s, f \dot{\neq} created \\ & \text{or } \neg select(h, o, created) \\ select(h, o, f) & \text{otherwise} \end{cases}$

\locset	
o.f	
o.m	
\fresh(s)	
\new_elems_fresh(s)	

\locset	*LocSet*
o.f	
o.m	
\fresh(s)	
\new_elems_fresh(s)	

JML* $\rightsquigarrow$ Java DL

\locset	*LocSet*
o.f	$\mathit{select}(\mathtt{H}, \mathtt{o}, \mathtt{f})$
o.m	
\fresh(s)	
\new_elems_fresh(s)	

JML* $\rightsquigarrow$ Java DL

\locset	*LocSet*
o.f	$\mathit{select}(\mathtt{H}, \mathtt{o}, \mathtt{f})$
o.m	$\mathtt{m}(\mathtt{H}, \mathtt{o})$
\fresh(s)	
\new_elems_fresh(s)	

\locset	*LocSet*
`o.f`	$select(\mathtt{H}, \mathtt{o}, \mathtt{f})$
`o.m`	$\mathtt{m}(\mathtt{H}, \mathtt{o})$
\fresh(s)	$\forall Object\ o; \forall Field\ f;$ $((o, f) \dot{\in} \mathtt{s} \rightarrow \neg select(\mathtt{H}^{old}, o, created))$
\new_elems_fresh(s)	

JML* $\rightsquigarrow$ Java DL

`\locset`	LocSet
`o.f`	$\mathit{select}(\mathtt{H}, \mathtt{o}, \mathtt{f})$
`o.m`	$\mathtt{m}(\mathtt{H}, \mathtt{o})$
`\fresh(s)`	$\forall \mathit{Object}\ o; \forall \mathit{Field}\ f;$ $((o, f) \dot{\in} \mathtt{s} \rightarrow \neg \mathit{select}(\mathtt{H}^{old}, o, \mathit{created}))$
`\new_elems_fresh(s)`	$\forall \mathit{Object}\ o; \forall \mathit{Field}\ f;$ $(o, f) \dot{\in} \mathtt{s} \rightarrow \neg \mathit{select}(\mathtt{H}^{old}, o, \mathit{created})$ $\lor\ (o, f)\ \dot{\in}\ \{\mathtt{H} := \mathtt{H}^{old}\}\mathtt{s}$

Method Contracts

Method contract $(pre, post, mod)$

Method Contracts

Method contract $(pre, post, mod)$

Proof obligation

$$pre \wedge GeneralAssumptions$$
$$\rightarrow [\texttt{self.m();}](post \wedge frame)$$

Method Contracts

Method contract $(pre, post, mod)$

Proof obligation

$$pre \land GeneralAssumptions$$
$$\rightarrow [\texttt{self.m();}](post \land frame)$$

$$\begin{aligned} frame \;=\; & \forall Object\, o; \forall Field\, f; \big((o, f) \,\dot{\in}\, \{\mathtt{H} := \mathtt{H}^{old}\} mod \\ & \qquad \lor \neg select(\mathtt{H}^{old}, o, created) \\ & \qquad \lor select(\mathtt{H}, o, f) \doteq select(\mathtt{H}^{old}, o, f)\big) \end{aligned}$$

Method Contracts

Method contract $(pre, post, mod)$

Proof obligation

$$pre \land GeneralAssumptions$$
$$\to [\texttt{self.m();}](post \land frame)$$

$$frame \;=\; \forall Object\, o; \forall Field\, f; \big((o, f) \,\dot{\in}\, \{\mathtt{H} := \mathtt{H}^{old}\} mod$$
$$\lor \neg select(\mathtt{H}^{old}, o, created)$$
$$\lor\, select(\mathtt{H}, o, f) \doteq select(\mathtt{H}^{old}, o, f)\big)$$

Rule

$$\Rightarrow \{\mathcal{U}\}[\texttt{o.m();}\ \ldots]\varphi$$

Method Contracts

Method contract $(\mathit{pre}, \mathit{post}, \mathit{mod})$

Proof obligation

$$\begin{aligned} & \mathit{pre} \land \mathit{GeneralAssumptions} \\ & \to [\texttt{self.m();}](\mathit{post} \land \mathit{frame}) \end{aligned}$$

$$\begin{aligned} \mathit{frame} \;=\; & \forall \mathit{Object}\, o; \forall \mathit{Field}\, f; \big((o, f) \,\dot{\in}\, \{\mathtt{H} := \mathtt{H}^{old}\}\mathit{mod} \\ & \qquad \lor \neg \mathit{select}(\mathtt{H}^{old}, o, \mathit{created}) \\ & \qquad \lor \mathit{select}(\mathtt{H}, o, f) \doteq \mathit{select}(\mathtt{H}^{old}, o, f)\big) \end{aligned}$$

Rule

$$\Rightarrow \{\mathcal{U}\}[\texttt{o.m(); ...}]\varphi$$

$$\Rightarrow \{\mathcal{U}\}\mathit{pre}$$

Method Contracts

Method contract $(pre, post, mod)$

Proof obligation

$$pre \land GeneralAssumptions$$
$$\rightarrow [\texttt{self.m();}](post \land frame)$$

$$frame \;=\; \forall Object\ o; \forall Field\ f; \big((o, f) \dot{\in} \{\texttt{H} := \texttt{H}^{old}\} mod$$
$$\lor \neg select(\texttt{H}^{old}, o, created)$$
$$\lor\ select(\texttt{H}, o, f) \doteq select(\texttt{H}^{old}, o, f)\big)$$

Rule

$$\Rightarrow \{\mathcal{U}\}[\texttt{o.m();}\ \ldots]\varphi$$

$$\Rightarrow \{\mathcal{U}\} pre \qquad\qquad \Rightarrow \{\mathcal{U}\}\{\texttt{H} := anon(\texttt{H}, mod, h')\}(post \rightarrow [\ldots]\varphi)$$

Dependency Contracts

Dependency contract (pre, dep)

Dependency Contracts

Dependency contract (pre, dep)

Proof obligation

$$
\begin{aligned}
& pre \land GeneralAssumptions \\
& \rightarrow m(\mathtt{H}, \mathtt{self}) \\
& \quad \doteq \{\mathtt{H} := anon(\mathtt{H}, everything \mathbin{\dot\setminus} dep, h)\} \\
& \qquad m(\mathtt{H}, \mathtt{self})
\end{aligned}
$$

Dependency Contracts

Dependency contract (pre, dep)

Proof obligation

$$\begin{aligned} & pre \land GeneralAssumptions \\ & \rightarrow m(\mathtt{H}, \mathtt{self}) \\ & \quad \doteq \{\mathtt{H} := anon(\mathtt{H}, everything \mathbin{\dot{\setminus}} dep, h)\} \\ & \qquad m(\mathtt{H}, \mathtt{self}) \end{aligned}$$

Rule (example application)

$$m(\mathtt{H}, \mathtt{c1}) \doteq 5 \Rightarrow m(anon(\mathtt{H}, s, h'), \mathtt{c1}) \doteq 5$$

Dependency contract (pre, dep)

Proof obligation

$$
\begin{aligned}
& pre \wedge GeneralAssumptions \\
& \rightarrow m(\mathtt{H}, \mathtt{self}) \\
& \quad \doteq \{\mathtt{H} := anon(\mathtt{H}, everything \mathbin{\dot{\setminus}} dep, h)\} \\
& \qquad m(\mathtt{H}, \mathtt{self})
\end{aligned}
$$

Rule (example application)

$$m(\mathtt{H}, \mathtt{c1}) \doteq 5 \Rightarrow m(anon(\mathtt{H}, s, h'), \mathtt{c1}) \doteq 5$$

add "$\Rightarrow pre \wedge s \mathbin{\dot{\cap}} dep \doteq \dot{\emptyset}$"

Dependency Contracts

Dependency contract (pre, dep)

Proof obligation

$$\begin{aligned} & pre \land GeneralAssumptions \\ & \rightarrow m(\mathtt{H}, \mathtt{self}) \\ & \quad \doteq \{\mathtt{H} := anon(\mathtt{H}, everything \mathbin{\dot{\setminus}} dep, h)\} \\ & \qquad m(\mathtt{H}, \mathtt{self}) \end{aligned}$$

Rule (example application)

$$m(\mathtt{H}, \mathtt{c1}) \doteq 5 \Rightarrow m(anon(\mathtt{H}, s, h'), \mathtt{c1}) \doteq 5$$

add "$\Rightarrow pre \land s \mathbin{\dot{\cap}} dep \doteq \dot{\emptyset}$"

add "$pre \land s \mathbin{\dot{\cap}} dep \doteq \dot{\emptyset} \Rightarrow$"

Dependency contract (pre, dep)

Proof obligation

$$
\begin{aligned}
& pre \land GeneralAssumptions \\
& \rightarrow m(\mathtt{H}, \mathtt{self}) \\
& \quad \doteq \{\mathtt{H} := anon(\mathtt{H}, everything \mathbin{\dot{\setminus}} dep, h)\} \\
& \quad\quad m(\mathtt{H}, \mathtt{self})
\end{aligned}
$$

Rule (example application)

$$m(\mathtt{H}, \mathtt{c1}) \doteq 5 \Rightarrow m(anon(\mathtt{H}, s, h'), \mathtt{c1}) \doteq 5$$

add "$\Rightarrow pre \land s \mathbin{\dot{\cap}} dep \doteq \dot{\emptyset}$"

add "$pre \land s \mathbin{\dot{\cap}} dep \doteq \dot{\emptyset} \Rightarrow$"
add "$m(\mathtt{H}, \mathtt{c1}) \doteq m(anon(\mathtt{H}, s, h'), \mathtt{c1}) \Rightarrow$"

Summary

- Goal: modular verification

Summary

- Goal: modular verification
- Specification with JML & dynamic frames
  - Type **`\locset`**
  - Depends clauses for model fields
  - Model field **`\inv`**

Summary

- Goal: modular verification
- Specification with JML & dynamic frames
  - Type `\locset`
  - Depends clauses for model fields
  - Model field `\inv`
- Verification with KeY & explicit heap modelling
  - Quantifying over locations and heaps
  - Reasoning with sets of locations

Summary

- Goal: modular verification
- Specification with JML & dynamic frames
  - Type **`\locset`**
  - Depends clauses for model fields
  - Model field **`\inv`**
- Verification with KeY & explicit heap modelling
  - Quantifying over locations and heaps
  - Reasoning with sets of locations
- Method contracts, dependency contracts

Summary

- Goal: modular verification
- Specification with JML & dynamic frames
  - Type `\locset`
  - Depends clauses for model fields
  - Model field `\inv`
- Verification with KeY & explicit heap modelling
  - Quantifying over locations and heaps
  - Reasoning with sets of locations
- Method contracts, dependency contracts
- Represents clauses $\rightsquigarrow$ assumptions

Summary

- Goal: modular verification
- Specification with JML & dynamic frames
  - Type `\locset`
  - Depends clauses for model fields
  - Model field `\inv`
- Verification with KeY & explicit heap modelling
  - Quantifying over locations and heaps
  - Reasoning with sets of locations
- Method contracts, dependency contracts
- Represents clauses $\leadsto$ assumptions
- Experience with implementation:
  - Examples by Smans et al. (cell, list, stack)
  - List with iterator
  - Observer pattern