Karlsruhe Institute of Technology

Dynamic Frames in Java Dynamic Logic

Peter H. Schmitt, Mattias Ulbrich, Benjamin Weiß

Overview

Setting

- Object-oriented programming (Java)

Overview

Setting

- Object-oriented programming (Java)
- Design by contract (JML)

Overview

Setting

- Object-oriented programming (Java)
- Design by contract (JML)
- Deductive verification (KeY)

Overview

Setting

- Object-oriented programming (Java)
- Design by contract (JML)
- Deductive verification (KeY)
  -  +  → JavaDL formula

Overview

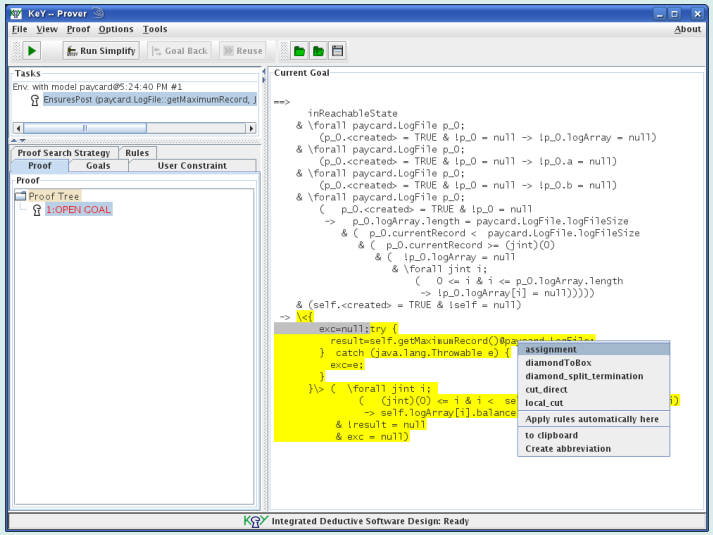
Setting

- Object-oriented programming (Java)
- Design by contract (JML)
- Deductive verification (KeY)
  -  +  → JavaDL formula
  - Theorem proving (interactive & automatic)

Overview

Setting

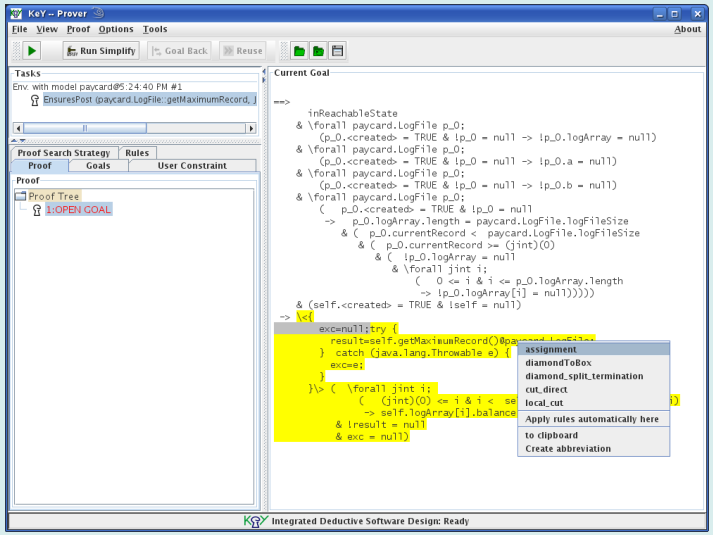
- Object-oriented programming (Java)
- Design by contract (JML)
- Deductive verification (KeY)
  -  +  → JavaDL formula
  - Theorem proving (interactive & automatic)



Overview

Setting

- Object-oriented programming (Java)
- Design by contract (JML)
- Deductive verification (KeY)
  -  +  → JavaDL formula
  - Theorem proving (interactive & automatic)



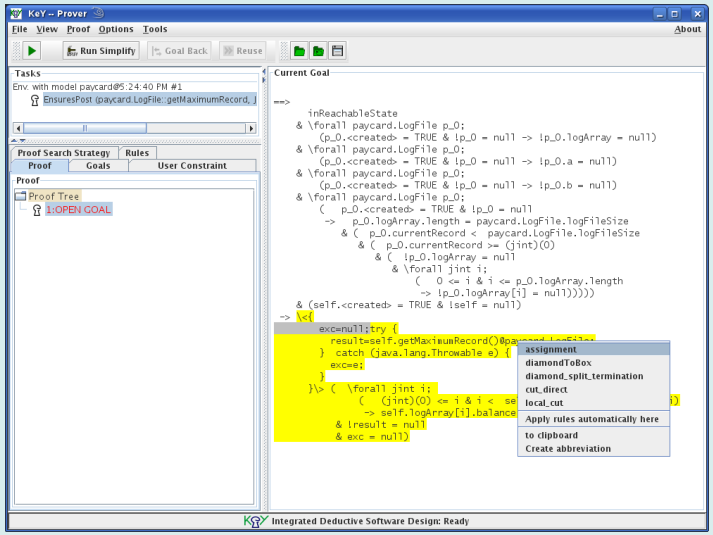
This talk

JavaDL meets dynamic frames (Kassios, 2006)

Overview

Setting

- Object-oriented programming (Java)
- Design by contract (JML)
- Deductive verification (KeY)
  -  +  → JavaDL formula
  - Theorem proving (interactive & automatic)



This talk

JavaDL meets dynamic frames (Kassios, 2006)

⇝ modularity

Example

```
interface List {
  void add(Object o);
  int size();

  Object get(int i);

}
```

Example

```
interface List {
  void add(Object o);
  int size();

  pre:

  post:

  Object get(int i);



}
```

Example

```
interface List {
  void add(Object o);
  int size();

  pre:

  post:

  Object get(int i);



}
```

- pure methods: `size`, `get`

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()

  post:

  Object get(int i);



}
```

- pure methods: `size`, `get`

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()

  post:

  Object get(int i);



}
```

- pure methods: `size`, `get`
- abstract fields: *inv*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post:

  Object get(int i);



}
```

- pure methods: `size`, `get`
- abstract fields: *inv*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≠ null

  Object get(int i);



}
```

- pure methods: `size`, `get`
- abstract fields: *inv*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≠̇ null
  mod:  ∅̇
  Object get(int i);



}
```

- pure methods: `size`, `get`
- abstract fields: *inv*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≠̇ null
  mod:  ∅̇
  Object get(int i);



}
```

```
class Client {
  int x;



}
```

- pure methods: `size`, `get`
- abstract fields: *inv*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≠̇ null
  mod:  ∅̇
  Object get(int i);



}
```

```
class Client {
  int x;

  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- pure methods: size, get
- abstract fields: *inv*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≠̇ null
  mod:  ∅̇
  Object get(int i);



}
```

```
class Client {
  int x;

  pre:



  post:
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- pure methods: `size`, `get`
- abstract fields: *inv*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≠̇ null
  mod:  ∅̇
  Object get(int i);



}
```

```
class Client {
  int x;

  pre:  l ≠̇ null ∧ l.inv
        ∧ 0 < l.size()

  post:
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- pure methods: `size`, `get`
- abstract fields: *inv*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);



}
```

```
class Client {
  int x;

  pre:  l ≠̇ null ∧ l.inv
        ∧ 0 < l.size()

  post: res ≠̇ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- pure methods: `size`, `get`
- abstract fields: *inv*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);



}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()

  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- pure methods: size, get
- abstract fields: *inv*
- dependency contracts

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≐≠ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):
  dep(this.size):
}
```

```
class Client {
  int x;

  pre:  l ≐≠ null ∧ l.inv
        ∧ 0 < l.size()

  post: res ≐≠ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- pure methods: size, get
- abstract fields: *inv*
- dependency contracts

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):
  dep(this.size):
}
```

```
class Client {
  int x;

  pre:  l ≠̇ null ∧ l.inv
        ∧ 0 < l.size()

  post: res ≠̇ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- pure methods: `size`, `get`
- abstract fields: *inv*, *locs* (a "dynamic frame")
- dependency contracts

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()

  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- pure methods: `size`, `get`
- abstract fields: *inv*, *locs* (a "dynamic frame")
- dependency contracts

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≠ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≠ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≠ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- pure methods: `size`, `get`
- abstract fields: *inv*, *locs* (a "dynamic frame")
- dependency contracts

Java Dynamic Logic

Dynamic logic (Harel, 1984)

- First-order logic + modal operators [p]

Java Dynamic Logic

Dynamic logic (Harel, 1984)

- First-order logic + modal operators [p]
- $[\mathrm{p}]\varphi$

Java Dynamic Logic

Dynamic logic (Harel, 1984)

- First-order logic + modal operators [p]
- [p]φ holds in a state iff φ holds in all states reachable via p

Java Dynamic Logic

Dynamic logic (Harel, 1984)

- First-order logic + modal operators [p]
- [p]φ holds in a state iff φ holds in all states reachable via p
- *pre* $\rightarrow$ [p]*post*

Java Dynamic Logic

Dynamic logic (Harel, 1984)

- First-order logic + modal operators [p]
- [p]φ holds in a state iff φ holds in all states reachable via p
- $pre \rightarrow [p]post \quad \hat{=} \quad$ Hoare triple $\{pre\}p\{post\}$

Java Dynamic Logic

Dynamic logic (Harel, 1984)

- First-order logic + modal operators [p]
- [p]φ holds in a state iff φ holds in all states reachable via p
- $pre \rightarrow [p]post \quad \hat{=} \quad$ Hoare triple $\{pre\}p\{post\}$

Updates

- $x := t$

Java Dynamic Logic

Dynamic logic (Harel, 1984)

- First-order logic + modal operators [p]
- [p]φ holds in a state iff φ holds in all states reachable via p
- $pre \rightarrow [\mathrm{p}]post \quad \hat{=} \quad$ Hoare triple $\{pre\}\mathrm{p}\{post\}$

Updates

- $\underbrace{\mathrm{x} := t}_{u}$

Java Dynamic Logic

Dynamic logic (Harel, 1984)

- First-order logic + modal operators [p]
- [p]φ holds in a state iff φ holds in all states reachable via p
- $pre \rightarrow [\mathrm{p}]post \quad \hat{=} \quad$ Hoare triple $\{pre\}\mathrm{p}\{post\}$

Updates

- $\underbrace{\mathrm{x} := t}_{u}$
- $\{u\}\varphi$

Java Dynamic Logic

Dynamic logic (Harel, 1984)

- First-order logic + modal operators [p]
- [p]φ holds in a state iff φ holds in all states reachable via p
- $pre \rightarrow [\mathrm{p}]post \quad \hat{=} \quad$ Hoare triple $\{pre\}\mathrm{p}\{post\}$

Updates

- $\underbrace{\mathrm{x} := t}_{u}$
- $\{u\}\varphi$ holds in a state iff φ holds in the state produced by u

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- `o.f`

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- $\mathtt{o.f} \;\hat{=}\; \mathtt{f(o)}$

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- $\mathtt{o.f} \;\hat{=}\; \mathtt{f(o)}$
- $[\mathtt{o.f}{=}t\mathtt{;}]$

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- $\mathtt{o.f} \;\hat{=}\; \mathtt{f(o)}$
- $[\mathtt{o.f}{=}t\mathtt{;}] \;\hat{=}\; \{\mathtt{f(o)} := t\}$

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- $\mathtt{o.f} \;\hat{=}\; \mathtt{f(o)}$
- $[\mathtt{o.f}{=}t\mathtt{;}] \;\hat{=}\; \{\mathtt{f(o)} := t\}$

Heap as a program variable

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

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- $\mathtt{o.f} \;\hat{=}\; \mathtt{f(o)}$
- $[\mathtt{o.f}{=}t\mathtt{;}] \;\hat{=}\; \{\mathtt{f(o)} := t\}$

Heap as a program variable

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

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- $\mathtt{o.f} \;\hat{=}\; \mathtt{f(o)}$
- $[\mathtt{o.f}{=}t\mathtt{;}] \;\hat{=}\; \{\mathtt{f(o)} := t\}$

Heap as a program variable

- constant symbols $\mathtt{f} : \mathit{Field}$
- program variable $\mathtt{H} : \mathit{Heap}$
- $\mathtt{o.f} \;\hat{=}\; \mathit{select}(\mathtt{H}, \mathtt{o}, \mathtt{f})$

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- $\mathtt{o.f} \;\hat{=}\; \mathtt{f(o)}$
- $[\mathtt{o.f}{=}t\mathtt{;}] \;\hat{=}\; \{\mathtt{f(o)} := t\}$

Heap as a program variable

- constant symbols $\mathtt{f} : \mathit{Field}$
- program variable $\mathtt{H} : \mathit{Heap}$
- $\mathtt{o.f} \;\hat{=}\; \mathit{select}(\mathtt{H}, \mathtt{o}, \mathtt{f})$
- $[\mathtt{o.f}{=}t\mathtt{;}] \;\hat{=}\; \{\mathtt{H} := \mathit{store}(\mathtt{H}, \mathtt{o}, \mathtt{f}, t)\}$

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- $\mathtt{o.f} \;\hat{=}\; \mathtt{f(o)}$
- $[\mathtt{o.f}{=}t\mathtt{;}] \;\hat{=}\; \{\mathtt{f(o)} := t\}$

Heap as a program variable

- constant symbols $\mathtt{f} : \mathit{Field}$
- program variable $\mathtt{H} : \mathit{Heap}$
- $\mathtt{o.f} \;\hat{=}\; \mathit{select}(\mathtt{H}, \mathtt{o}, \mathtt{f})$
- $[\mathtt{o.f}{=}t\mathtt{;}] \;\hat{=}\; \{\mathtt{H} := \mathit{store}(\mathtt{H}, \mathtt{o}, \mathtt{f}, t)\}$
- $\mathit{select}(\mathit{store}(h, o, f, x), o', f')$

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- $\mathtt{o.f} \;\hat{=}\; \mathtt{f(o)}$
- $[\mathtt{o.f}{=}t\mathtt{;}] \;\hat{=}\; \{\mathtt{f(o)} := t\}$

Heap as a program variable

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

Heap Modelling

Heap as non-rigid functions

- function symbols $\mathtt{f} : C \to A$
- $\mathtt{o.f} \;\hat{=}\; \mathtt{f(o)}$
- $[\mathtt{o.f}{=}t\mathtt{;}] \;\hat{=}\; \{\mathtt{f(o)} := t\}$

Heap as a program variable

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

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- size : *Heap*, List → **int**

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < this.size()
        ∧ this.inv
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- size : *Heap*, List → **int**

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < size(H,this)
        ∧ this.inv
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- size : *Heap*, List → **int**

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < size(H,this)
        ∧ this.inv
  post: res ≠̇ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≠̇ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≠̇ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- size : *Heap*, List → **int**
- *inv* : *Heap*, List

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < size(H,this)
        ∧ this.inv
  post: res ≠̇ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≠̇ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≠̇ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- $\texttt{size} : \mathit{Heap}, \texttt{List} \to \textbf{int}$
- $\mathit{inv} : \mathit{Heap}, \texttt{List}$

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < size(H,this)
        ∧ inv(H,this)
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- size : *Heap*, List → **int**
- *inv* : *Heap*, List

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < size(H,this)
        ∧ inv(H,this)
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- size : *Heap*, List → **int**
- *inv* : *Heap*, List
- *locs* : *Heap*, List → *LocSet*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < size(H,this)
        ∧ inv(H,this)
  post: res ≠̇ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  this.locs
  dep(this.size): this.locs
}
```

```
class Client {
  int x;

  pre:  l ≠̇ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≠̇ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- size : *Heap*, List → **int**
- *inv* : *Heap*, List
- *locs* : *Heap*, List → *LocSet*

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < size(H,this)
        ∧ inv(H,this)
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  locs(H,this)
  dep(this.size): locs(H,this)
}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

- size : *Heap*, List → **int**
- *inv* : *Heap*, List
- *locs* : *Heap*, List → *LocSet*

Proof Obligations

For all **method contracts** $(\mathtt{m}, \mathit{pre}, \mathit{post}, \mathit{mod})$, classes C:

Proof Obligations

For all **method contracts** $(\mathtt{m}, \mathit{pre}, \mathit{post}, \mathit{mod})$, classes C:

$$\mathit{pre} \wedge \mathit{exactInstance}_C(\mathtt{self})$$
$$\Rightarrow [\mathtt{self.m();}](\mathit{post} \wedge \mathit{frame})$$

Proof Obligations

For all **method contracts** $(\mathtt{m}, pre, post, mod)$, classes C:

$$pre \wedge exactInstance_C(\mathtt{self})$$
$$\Rightarrow [\mathtt{self.m();}](post \wedge frame)$$

For all **dependency contracts** (f, pre, dep), classes C:

Proof Obligations

For all method contracts $(\mathtt{m}, pre, post, mod)$, classes C:

$$pre \land exactInstance_C(\mathtt{self})$$
$$\Rightarrow [\mathtt{self.m();}](post \land frame)$$

For all dependency contracts (f, pre, dep), classes C:

$$pre \land exactInstance_C(\mathtt{self})$$
$$\Rightarrow f(\mathtt{H}, \mathtt{self})$$
$$\doteq \{\mathtt{H} := anon(\mathtt{H}, allLocs \setminus dep, h)\} f(\mathtt{H}, \mathtt{self})$$

Proof Obligations

For all method contracts $(\mathtt{m}, pre, post, mod)$, classes C:

$$pre \land exactInstance_C(\mathtt{self})$$
$$\Rightarrow [\mathtt{self.m();}](post \land frame)$$

For all dependency contracts (f, pre, dep), classes C:

$$pre \land exactInstance_C(\mathtt{self})$$
$$\Rightarrow f(\mathtt{H}, \mathtt{self})$$
$$\doteq \{\mathtt{H} := anon(\mathtt{H}, allLocs \setminus dep, h)\} f(\mathtt{H}, \mathtt{self})$$

Proof Obligations

For all method contracts $(\mathtt{m}, pre, post, mod)$, classes C:

$$pre \land exactInstance_C(\mathtt{self})$$
$$\Rightarrow [\mathtt{self.m();}](post \land frame)$$

For all dependency contracts (f, pre, dep), classes C:

$$pre \land exactInstance_C(\mathtt{self})$$
$$\Rightarrow f(\mathtt{H}, \mathtt{self})$$
$$\doteq \{\mathtt{H} := anon(\mathtt{H}, allLocs \setminus dep, h)\} f(\mathtt{H}, \mathtt{self})$$

$anon : Heap, LocSet, Heap \to Heap$

Proof Obligations

For all method contracts $(\mathtt{m}, pre, post, mod)$, classes C:

$$pre \wedge exactInstance_C(\mathtt{self}) \Rightarrow [\mathtt{self.m();}](post \wedge frame)$$

For all dependency contracts (f, pre, dep), classes C:

$$pre \wedge exactInstance_C(\mathtt{self}) \Rightarrow f(\mathtt{H}, \mathtt{self}) \doteq \{\mathtt{H} := anon(\mathtt{H}, allLocs \setminus dep, h)\} f(\mathtt{H}, \mathtt{self})$$

$anon : Heap, LocSet, Heap \to Heap$

$$select(anon(h_1, s, h_2), o, f) = \begin{cases} \\ \\ \end{cases}$$

Proof Obligations

For all method contracts $(\mathtt{m}, pre, post, mod)$, classes C:

$$pre \land exactInstance_C(\mathtt{self}) \Rightarrow [\mathtt{self.m();}](post \land frame)$$

For all dependency contracts (f, pre, dep), classes C:

$$pre \land exactInstance_C(\mathtt{self}) \Rightarrow f(\mathtt{H}, \mathtt{self}) \doteq \{\mathtt{H} := anon(\mathtt{H}, allLocs \setminus dep, h)\} f(\mathtt{H}, \mathtt{self})$$

$anon : Heap, LocSet, Heap \to Heap$

$$select(anon(h_1, s, h_2), o, f) = \begin{cases} select(h_1, o, f) & \text{if } (o, f) \doteq\in s \\ \end{cases}$$

Proof Obligations

For all method contracts $(\mathtt{m}, pre, post, mod)$, classes C:

$$pre \land exactInstance_C(\mathtt{self}) \Rightarrow [\mathtt{self.m();}](post \land frame)$$

For all dependency contracts (f, pre, dep), classes C:

$$pre \land exactInstance_C(\mathtt{self}) \Rightarrow f(\mathtt{H}, \mathtt{self}) \doteq \{\mathtt{H} := anon(\mathtt{H}, allLocs \setminus dep, h)\} f(\mathtt{H}, \mathtt{self})$$

$anon : Heap, LocSet, Heap \to Heap$

$$select(anon(h_1, s, h_2), o, f) = \begin{cases} select(h_1, o, f) & \text{if } (o, f) \doteq\in s \\ select(h_2, o, f) & \text{otherwise} \end{cases}$$

Proof Obligations

For all method contracts $(\mathtt{m}, pre, post, mod)$, classes C:

$$pre \land exactInstance_C(\mathtt{self})$$
$$\Rightarrow [\mathtt{self.m();}](post \land frame)$$

For all dependency contracts (f, pre, dep), classes C:

$$pre \land exactInstance_C(\mathtt{self})$$
$$\Rightarrow f(\mathtt{H}, \mathtt{self})$$
$$\doteq \{\mathtt{H} := anon(\mathtt{H}, allLocs \setminus dep, h)\} f(\mathtt{H}, \mathtt{self})$$

$anon : Heap, LocSet, Heap \to Heap$

$$select(anon(h_1, s, h_2), o, f) = \begin{cases} select(h_1, o, f) & \text{if } (o, f) \doteq\in s,\ f \doteq\neq created \\ select(h_2, o, f) & \text{otherwise} \end{cases}$$

Proof Obligations

For all method contracts $(\mathtt{m}, pre, post, mod)$, classes C:

$$pre \land exactInstance_C(\mathtt{self}) \Rightarrow [\mathtt{self.m();}](post \land frame)$$

For all dependency contracts (f, pre, dep), classes C:

$$pre \land exactInstance_C(\mathtt{self}) \Rightarrow f(\mathtt{H}, \mathtt{self}) \doteq \{\mathtt{H} := anon(\mathtt{H}, allLocs \setminus dep, h)\} f(\mathtt{H}, \mathtt{self})$$

$anon : Heap, LocSet, Heap \to Heap$

$$select(anon(h_1, s, h_2), o, f) = \begin{cases} select(h_1, o, f) & \text{if } (o, f) \doteq\in s,\ f \doteq\neq created \\ & \text{or } \neg select(h_1, o, created) \\ select(h_2, o, f) & \text{otherwise} \end{cases}$$

Example

```
interface List {
  void add(Object o);
  int size();

  pre:  0 ≤ i ∧ i < size(H,this)
        ∧ inv(H,this)
  post: res ≐ null
  mod:  ∅̇
  Object get(int i);

  dep(this.inv):  locs(H,this)
  dep(this.size): locs(H,this)
}
```

```
class Client {
  int x;

  pre:  l ≐ null ∧ l.inv
        ∧ 0 < l.size()
        ∧ (this, x) ∉̇ l.locs
  post: res ≐ null
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

Summary

- Goal: modular deductive Java verification

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style
  - Method contracts, dependency contracts

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style
  - Method contracts, dependency contracts
  - Pure methods, abstract fields

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style
  - Method contracts, dependency contracts
  - Pure methods, abstract fields
  - Dynamic frames: abstract fields of type *LocSet*

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style
  - Method contracts, dependency contracts
  - Pure methods, abstract fields
  - Dynamic frames: abstract fields of type *LocSet*
- Verification in dynamic logic with updates

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style
  - Method contracts, dependency contracts
  - Pure methods, abstract fields
  - Dynamic frames: abstract fields of type *LocSet*
- Verification in dynamic logic with updates
  - Heap model: single program variable (vs. per-field non-rigid functions)

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style
  - Method contracts, dependency contracts
  - Pure methods, abstract fields
  - Dynamic frames: abstract fields of type *LocSet*
- Verification in dynamic logic with updates
  - Heap model: single program variable (vs. per-field non-rigid functions)
  - $anon : Heap, LocSet, Heap \to Heap$

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style
  - Method contracts, dependency contracts
  - Pure methods, abstract fields
  - Dynamic frames: abstract fields of type *LocSet*
- Verification in dynamic logic with updates
  - Heap model: single program variable (vs. per-field non-rigid functions)
  - $anon : Heap, LocSet, Heap \to Heap$
  - Proof obligations for contracts

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style
  - Method contracts, dependency contracts
  - Pure methods, abstract fields
  - Dynamic frames: abstract fields of type *LocSet*
- Verification in dynamic logic with updates
  - Heap model: single program variable (vs. per-field non-rigid functions)
  - $anon : Heap, LocSet, Heap \to Heap$
  - Proof obligations for contracts
  - Rules for using contracts

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style
  - Method contracts, dependency contracts
  - Pure methods, abstract fields
  - Dynamic frames: abstract fields of type *LocSet*
- Verification in dynamic logic with updates
  - Heap model: single program variable (vs. per-field non-rigid functions)
  - $anon : Heap, LocSet, Heap \to Heap$
  - Proof obligations for contracts
  - Rules for using contracts
- Related: VeriCool, Dafny, data groups, ownership, separation logic

Summary

- Goal: modular deductive Java verification
- Specification in design-by-contract style
  - Method contracts, dependency contracts
  - Pure methods, abstract fields
  - Dynamic frames: abstract fields of type *LocSet*
- Verification in dynamic logic with updates
  - Heap model: single program variable (vs. per-field non-rigid functions)
  - $anon : Heap, LocSet, Heap \to Heap$
  - Proof obligations for contracts
  - Rules for using contracts
- Related: VeriCool, Dafny, data groups, ownership, separation logic
- Implemented in a version of KeY

Axioms

Pure methods

$$\forall \texttt{ArrayList}\ l;\ \big(\mathit{exactInstance}_{\texttt{ArrayList}}(l) \rightarrow \forall \mathit{Heap}\ h;\ \forall \mathbf{int}\ i;\ \big(\texttt{size}(h, l) \doteq i \leftrightarrow \{\texttt{H} := h\}[\texttt{p}](\texttt{res} \doteq i)\big)\big)$$

Abstract variables

$$\forall \texttt{ArrayList}\ l;\ \big(\mathit{exactInstance}_{\texttt{ArrayList}}(l) \rightarrow \forall \mathit{Heap}\ h;\ \big(\mathit{locs}(h, l) \doteq \{(h, \texttt{a})\} \dot{\cup}\ \mathit{allFields}(\mathit{select}(h, l, \texttt{a}))\big)\big)$$

Example: Proving a Proof Obligation