Formal Specification of Software

Bernhard Beckert

Adaptation of slides by
Wolfgang Ahrendt
Chalmers University, Gothenburg, Sweden

Unit Specifications

in the object-oriented setting:

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
-

But what do we mean by state?

Unit Specifications

in the object-oriented setting:

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
-

But what do we mean by state?

Unit Specifications

in the object-oriented setting:

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,

But what do we mean by state?

Unit Specifications

in the object-oriented setting:

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
- the overall state

But what do we mean by state?

Unit Specifications

in the object-oriented setting:

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
- the locally visible part of the overall state

But what do we mean by state?

Unit Specifications

in the object-oriented setting:

The units to be specified are interfaces, classes, and their methods.

We first focus on specifying methods.

Methods are specified by *potentially* referring to:

- the result value,
- the initial values of formal parameters,
- the locally visible part of the overall state

But what do we mean by state?

Prerequisite: Object-oriented States

By state, we mean a 'snapshot' of the system, at any point during the the computation, described in terms of the *programmer's model*.

An *object oriented* state consists of:

- the set $\mathcal{C}$ of all loaded classes
- the values of the static fields of classes in $\mathcal{C}$
- the set $\mathcal{O}$ of references to all created objects
- the values of the instance fields of objects in $\mathcal{O}$

Here, values of *local variables* and *formal parameters* are *not* considered part of the state.

Prerequisite: Object-oriented States

By state, we mean a 'snapshot' of the system, at any point during the the computation, described in terms of the *programmer's model*.

An *object oriented* state consists of:

- the set $\mathcal{C}$ of all loaded classes
- the values of the static fields of classes in $\mathcal{C}$
- the set $\mathcal{O}$ of references to all created objects
- the values of the instance fields of objects in $\mathcal{O}$

Here, values of *local variables* and *formal parameters* are *not* considered part of the state.

Prerequisite: Object-oriented States

By state, we mean a 'snapshot' of the system, at any point during the the computation, described in terms of the *programmer's model*.

An *object oriented* state consists of:

- the set $\mathcal{C}$ of all loaded classes
- the values of the static fields of classes in $\mathcal{C}$
- the set $\mathcal{O}$ of references to all created objects
- the values of the instance fields of objects in $\mathcal{O}$

Here, values of *local variables* and *formal parameters* are *not* considered part of the state.

Prerequisite: Visible State

Like implementations, specifications can only refer to the locally visible part of the state (e.g., not to `private` fields of other classes).

Prerequisite: Visible State

In our context, we stick to the following principle:

Same Visible State for Specifications and Implementations:

In some local context, specifications and implementations can access the same part of the overall state.[a]

[a] Later, we'll refine this principle, and introduce well defined exceptions.

Thus, specifications talk only about those parts of the state which are accessible by:

- respecting JAVA's visibility rules (`public`, `protected`, `private`),
- following (visible) references, starting from local fields.

Prerequisite: Visible State

In our context, we stick to the following principle:

Same Visible State for Specifications and Implementations:

In some local context, specifications and implementations can access the same part of the overall state.[a]

[a] Later, we'll refine this principle, and introduce well defined exceptions.

Thus, specifications talk only about those parts of the state which are accessible by:

- respecting JAVA's visibility rules (`public`, `protected`, `private`),
- following (visible) references, starting from local fields.

Prerequisite: Visible State

In our context, we stick to the following principle:

Same Visible State for Specifications and Implementations:

In some local context, specifications and implementations can access the same part of the overall state.[a]

[a] Later, we'll refine this principle, and introduce well defined exceptions.

Thus, specifications talk only about those parts of the state which are accessible by:

- respecting JAVA's visibility rules (`public`, `protected`, `private`),
- following (visible) references, starting from local fields.

Purely Functional Specification

A purely functional specification of a (non-`void`) method talks

- only about
  - the result of a call
  - the initial value of input parameters
- but not about
  - (any part of) the state

examples:

interface/class:	*method:*
`Math`	`static int` **`abs`**`(int a)`
`Math`	`static double` **`sqrt`**`(double a)`

Purely Functional Specification

A purely functional specification of a (non-`void`) method talks

- only about
  - the result of a call
  - the initial value of input parameters
- but not about
  - (any part of) the state

examples:

interface/class:	*method:*
`Math`	`static int` **`abs`**`(int a)`
`Math`	`static double` **`sqrt`**`(double a)`

Purely Functional Specification: `Math::abs()`

from the JAVA API:

Specification of `static int abs(int a)`

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Green: Intuitive description rather than a specification.
Red: Precise specification by case distinction, given we know what 'negative' and 'negation' mean exactly.
Blue: A consequence of the specification, i.e. a *redundant part* of it.

Red and Blue are candidates for formalisation.

Purely Functional Specification: `Math::abs()`

from the JAVA API:

Specification of `static int abs(int a)`

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Green: Intuitive description rather than a specification.
Red: Precise specification by case distinction, given we know what 'negative' and 'negation' mean exactly.
Blue: A consequence of the specification, i.e. a *redundant part* of it.

Red and Blue are candidates for formalisation.

Purely Functional Specification: `Math::abs()`

from the JAVA API:

Specification of `static int abs(int a)`

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Green: Intuitive description rather than a specification.
Red: Precise specification by case distinction, given we know what 'negative' and 'negation' mean exactly.
Blue: A consequence of the specification, i.e. a *redundant part* of it.

Red and Blue are candidates for formalisation.

Purely Functional Specification: `Math::abs()`

from the JAVA API:

Specification of `static int abs(int a)`

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Green: Intuitive description rather than a specification.
Red: Precise specification by case distinction, given we know what 'negative' and 'negation' mean exactly.
Blue: A consequence of the specification, i.e. a *redundant part* of it.

Red and Blue are candidates for formalisation.

Going a bit more formal

```
static int abs(int a)
```

Informal spec:
If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Semi formal:

- Under the precondition ‘$a \in [0 ... 2147483647]$’, **abs** ensures the postcondition ‘$\mathrm{result} = a$’.
- Under the precondition ‘$a \in [-2147483648 ... -1]$’, **abs** ensures the postcondition ‘$\mathrm{result} = -a$’.

Going a bit more formal

```
static int abs(int a)
```

Informal spec:
If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Semi formal:

- Under the precondition '$\mathtt{a} \in [0...2147483647]$', **abs** ensures the postcondition '$\mathrm{result} = \mathtt{a}$'.
- Under the precondition '$\mathtt{a} \in [-2147483648... -1]$', **abs** ensures the postcondition '$\mathrm{result} = -\mathtt{a}$'.

Going a bit more formal

`static int abs(int a)`

Redundant informal spec:
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Semi formal:

- Under the precondition '$a = -2147483648$', **`abs`** ensures the postcondition '$\text{result} = -2147483648$'.

Or simply:[1]

- **`abs`**$(-2147483648) = -2147483648$

[1] But be careful when using a method call in a formula, see below.

Going a bit more formal

```
static int abs(int a)
```

Redundant informal spec:
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Semi formal:

- Under the precondition '$\texttt{a} = -2147483648$', **abs** ensures the postcondition '$\text{result} = -2147483648$'.

Or simply:[1]

- $\texttt{abs}(-2147483648) = -2147483648$

[1] But be careful when using a method call in a formula, see below.

State Aware Specification

A state aware specification of a (`void` or non-`void`) method talks about

- the result of a call (if non-`void`)
- the initial value of input parameters
- *two* states:
  - the 'pre-state' of the method call
  - the 'post-state' of the method call

examples:

interface/class:	*method:*
`List`	`Object` **`set`**`(int index, Object element)`
`Collections`	`static void` **`sort`**`(List list)`

State Aware Specification

A state aware specification of a (`void` or non-`void`) method talks about

- the result of a call (if non-`void`)
- the initial value of input parameters
- *two* states:
  - the 'pre-state' of the method call
  - the 'post-state' of the method call

examples:

interface/class:	*method:*
`List`	`Object` **`set`**`(int index, Object element)`
`Collections`	`static void` **`sort`**`(List list)`

State Aware Specification

A state aware specification of a (`void` or non-`void`) method talks about

- the result of a call (if non-`void`)
- the initial value of input parameters
- *two* states:
  - the 'pre-state' of the method call
  - the 'post-state' of the method call

examples:

interface/class:	*method:*
`List`	`Object` **`set`**`(int index, Object element)`
`Collections`	`static void` **`sort`**`(List list)`

State Aware Specification

A state aware specification of a (`void` or non-`void`) method talks about

- the result of a call (if non-`void`)
- the initial value of input parameters
- *two* states:
  - the 'pre-state' of the method call
  - the 'post-state' of the method call

examples:

interface/class:	*method:*
`List`	`Object` **`set`**`(int index, Object element)`
`Collections`	`static void` **`sort`**`(List list)`

State Aware Specification: `List::set(i,e)`

from the Java API of `List::set` *(simplified):*

```
public Object set(int index, Object element)
```

Replaces the element at the specified position in this list with the specified element.

Parameters:

`index` - index of element to replace.

`element` - element to be stored at the specified position.

Returns:

the element previously at the specified position.

Throws:

`IndexOutOfBoundsException`

- if the index is out of range (index $<$ 0 || index $>=$ size()).

Why is the spec state aware?

It talks about the state, in particular about the state change.

State Aware Specification: `List::set(i,e)`

from the Java API of `List::set` *(simplified):*

`public Object` **`set`**`(int index, Object element)`

Replaces the element at the specified position in this list with the specified element.

Parameters:
`index` - index of element to replace.
`element` - element to be stored at the specified position.
Returns:
the element previously at the specified position.
Throws:
`IndexOutOfBoundsException`
- if the index is out of range (index $<$ 0 || index $>=$ size()).

Why is the spec state aware?

It talks about the state, in particular about the state change.

State Aware Specification: `List::set(i,e)`

from the Java API of `List::set` *(simplified):*

`public Object` **`set`**`(int index, Object element)`

Replaces the element at the specified position in this list with the specified element.

Parameters:

`index` - index of element to replace.

`element` - element to be stored at the specified position.

Returns:

the element previously at the specified position.

Throws:

`IndexOutOfBoundsException`

- if the index is out of range (index $<$ 0 $||$ index $>=$ size()).

Why is the spec state aware?

It talks about the state, in particular about the state change.

Going a bit more formal

`public Object set(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element.

Semi formal:
`set` ensures the following postcondition:

- `element` = '`get(index)` *evaluated in the post-state*'

Does this capture the meaning of the word 'replace'?

Going a bit more formal

`public Object set(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element.

Semi formal:
`set` ensures the following postcondition:

- `element` = '`get(index)` *evaluated in the post-state*'

Does this capture the meaning of the word 'replace'?

Going a bit more formal

`public Object set(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element.

Semi formal:
`set` ensures the following postcondition:

- `element` = ‘`get(index)` *evaluated in the post-state*’

Does this capture the meaning of the word ‘replace’?

Going a bit more formal

`public Object set(int index, Object element)`

Informal spec:

Replaces the element at the specified position in this list with the specified element.

Semi formal:

`set` ensures the following postconditions:

- `element` = '`get(index)` *evaluated in the post-state*', and
- for all j $\in$ [0...`size()` $-$ 1] with j $\neq$ `index`:
  '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'

Going a bit more formal

`public Object set(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element.

Semi formal:
`set` ensures the following postconditions:

- `element` = '`get(index)` *evaluated in the post-state*', and
- for all `j` $\in$ [0...`size()` − 1] with `j` $\neq$ `index`:
  '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'

Going a bit more formal

`public Object set(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position.

Semi formal:
`set` ensures the following postconditions:

- result = '`get(index)` *evaluated in the pre-state*', and
- `element` = '`get(index)` *evaluated in the post-state*', and
- for all j $\in$ [0...size() $-$ 1] with j $\neq$ `index`:
  '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'

Going a bit more formal

```
public Object set(int index, Object element)
```

Informal spec:

Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position.

Semi formal:

`set` ensures the following postconditions:

- result = '`get(index)` *evaluated in the pre-state*', and
- `element` = '`get(index)` *evaluated in the post-state*', and
- for all $\texttt{j} \in [0 \ldots \texttt{size}() - 1]$ with $\texttt{j} \neq \texttt{index}$:
  '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'

Going a bit more formal

`public Object set(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position ... Throws `IndexOutOfBoundsException` if the index is out of range (index $<$ 0 || index $>=$ size()).

Semi formal:

- Under the precondition '`index` $\in$ [0...`size`() $-$ 1]', **`set`** ensures the following postconditions:
  - result $=$ '`get(index)` *evaluated in the pre-state*', and
  - `element` $=$ '`get(index)` *evaluated in the post-state*', and
  - for all j $\in$ [0...`size`() $-$ 1] with j $\neq$ `index`:
    '`get(j)` *in post-state*' $=$ '`get(j)` *in pre-state*'
- Under the precondition '`index` $\notin$ [0...`size`() $-$ 1]', **`set`** throws `IndexOutOfBoundsException`.

Going a bit more formal

`public Object set(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position ... Throws `IndexOutOfBoundsException` if the index is out of range (index $<$ 0 || index $>=$ size()).

Semi formal:

- Under the precondition '`index` $\in$ [0...`size`() $-$ 1]', **`set`** ensures the following postconditions:
    - result $=$ '`get(index)` *evaluated in the pre-state*', and
    - `element` $=$ '`get(index)` *evaluated in the post-state*', and
    - for all j $\in$ [0...`size`() $-$ 1] with j $\neq$ `index`:
      '`get(j)` *in post-state*' $=$ '`get(j)` *in pre-state*'
- Under the precondition '`index` $\notin$ [0...`size`() $-$ 1]', **`set`** throws `IndexOutOfBoundsException`.

Altogether:

```
public Object set(int index, Object element)
```

Informal spec:
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position ... Throws `IndexOutOfBoundsException` if the index is out of range (index $<$ 0 || index $>=$ size()).

Semi formal:

- Under the precondition '$\texttt{index} \in [0...\texttt{size}() - 1]$', **`set`** ensures the following postconditions:
  - result = '`get(index)` *evaluated in the pre-state*', and
  - `element` = '`get(index)` *evaluated in the post-state*', and
  - for all $\texttt{j} \in [0...\texttt{size}() - 1]$ with $\texttt{j} \neq \texttt{index}$:
    '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'
- Under the precondition '$\texttt{index} \notin [0...\texttt{size}() - 1]$', **`set`** throws `IndexOutOfBoundsException`.

Reflection

We identify elements of a framework for *Formal Specification*

- pairs of
  - preconditions
  - corresponding postconditions
- a language to express these conditions, capturing:
  - relations, equality, logical connectives
  - *quantification*
- constructs to refer to:
  - values in the *new* and in the *old* state
  - the throwing of exceptions

To identify one more element, we consider another example.

Reflection

We identify elements of a framework for *Formal Specification*

- pairs of
  - preconditions
  - corresponding postconditions
- a language to express these conditions, capturing:
  - relations, equality, logical connectives
  - *quantification*
- constructs to refer to:
  - values in the *new* and in the *old* state
  - the throwing of exceptions

To identify one more element, we consider another example.

Consider Class SortedIntegers

```java
public class SortedIntegers {

    private int arr[];
    private int capacity, size = 0;

    public SortedIntegers(int capacity) {
        this.capacity = capacity;
        this.arr = new int[capacity];
    }

    public void add(int elem) { /*...*/ }

    public boolean remove(int elem) { /*...*/ }

    public int max() { /*...*/ }
}
```

Which methods have purely functional / state aware specifications?

Consider Class `SortedIntegers`

```
public class SortedIntegers {

    private int arr[];
    private int capacity, size = 0;

    public SortedIntegers(int capacity) {
        this.capacity = capacity;
        this.arr = new int[capacity];
    }

    public void add(int elem) { /*...*/ }

    public boolean remove(int elem) { /*...*/ }

    public int max() { /*...*/ }
}
```

Which methods have purely functional / state aware specifications?

Specifying `SortedIntegers::max()`

Specification of `int max()`

`max()` returns the maximum of the elements in the array `arr`.

But that is not what we wanted.

`max()` should return the maximum of the elements which were already added, and not removed thereafter.

Specifying `SortedIntegers::max()`

Specification of `int max()`

`max()` returns the maximum of the elements in the array `arr`.

But that is not what we wanted.

`max()` should return the maximum of the elements which were already added, and not removed thereafter.

Specifying `SortedIntegers::max()`

Specification of `int max()`

`max()` returns the maximum of the elements in the array `arr`.

But that is not what we wanted.

`max()` should return the maximum of the elements which were already added, and not removed thereafter.

Specifying `SortedIntegers::max()`

Specification of `int max()`

`max()` returns the maximum of those elements in the array `arr` which were already added, and not removed thereafter.

How can we state this without referring to the history of the object?

We can use the fact that the integers are (supposed to be) sorted.

Specifying `SortedIntegers::max()`

Specification of `int max()`

`max()` returns the maximum of those elements in the array `arr` which were already added, and not removed thereafter.

How can we state this without referring to the history of the object?

We can use the fact that the integers are (supposed to be) sorted.

Specifying `SortedIntegers::max()`

Specification of `int max()`

`max()` returns the maximum of those elements in the array `arr` which were already added, and not removed thereafter.

How can we state this without referring to the history of the object?

We can use the fact that the integers are (supposed to be) sorted.

Specifying `SortedIntegers::max()`

Specification of `int max()` now much simpler

`max()` returns `arr(size-1)`.

Sufficient if we assume sortedness.

Questions:

A) how to express the sortedness property?

B) how to specify that an instance of `SortedIntegers` always has this property?

Specifying `SortedIntegers::max()`

Specification of `int max()` now much simpler

`max()` returns `arr(size-1)`.

Sufficient if we assume sortedness.

Questions:

A) how to express the sortedness property?

B) how to specify that an instance of `SortedIntegers` always has this property?

Specifying `SortedIntegers::max()`

Specification of `int max()` now much simpler

`max()` returns `arr(size-1)`.

Sufficient if we assume sortedness.

Questions:

A) how to express the sortedness property?

B) how to specify that an instance of `SortedIntegers` always has this property?

A) Expressing Sortedness

A `SortedIntegers` object is sorted if:

for all $\texttt{i} \in [0 \ldots \texttt{size}() - 2]$: $\quad \texttt{arr(i)} \leq \texttt{arr(i+1)}$

Below, we abbreviate this condition by '$\mathcal{SORTED}$'.

Note:
Even `SortedIntegers` objects with with $\texttt{size()} \leq 1$ satisfy $\mathcal{SORTED}$.

A) Expressing Sortedness

A `SortedIntegers` object is sorted if:

for all $\texttt{i} \in [0 \ldots \texttt{size()} - 2]$: $\texttt{arr(i)} \leq \texttt{arr(i+1)}$

Below, we abbreviate this condition by '$\mathcal{SORTED}$'.

Note:
Even `SortedIntegers` objects with with $\texttt{size()} \leq 1$ satisfy $\mathcal{SORTED}$.

A) Expressing Sortedness

A `SortedIntegers` object is sorted if:

for all $\texttt{i} \in [0 ... \texttt{size}() - 2]$: $\quad \texttt{arr(i)} \leq \texttt{arr(i+1)}$

Below, we abbreviate this condition by '$\mathcal{SORTED}$'.

Note:
Even `SortedIntegers` objects with with `size()` ≤ 1 satisfy $\mathcal{SORTED}$.

B) Specifying Sortedness

How to specify that sortedness is a property of a `SortedIntegers` object *at any time*?

State that $\mathcal{SORTED}$ is *invariant* w.r.t. actions on `SortedIntegers`.

i.e., $\mathcal{SORTED}$ is:

- established by all constructors
- maintained by all methods

B) Specifying Sortedness

How to specify that sortedness is a property of a `SortedIntegers` object *at any time*?

State that $\mathcal{SORTED}$ is *invariant* w.r.t. actions on `SortedIntegers`.

i.e., $\mathcal{SORTED}$ is:

- established by all constructors
- maintained by all methods

B) Specifying Sortedness

How to specify that sortedness is a property of a `SortedIntegers` object *at any time*?

State that $\mathcal{SORTED}$ is *invariant* w.r.t. actions on `SortedIntegers`.

i.e., $\mathcal{SORTED}$ is:

- established by all constructors
- maintained by all methods

B) Specifying Sortedness

add $\mathcal{SORTED}$ to

- postcondition of all constructors
- precondition and postcondition of all methods

Problem: This way,

- invariant conditions bloat the specification,
- invariant conditions are difficult to maintain.

B) Specifying Sortedness

add $\mathcal{SORTED}$ to

- postcondition of all constructors
- precondition and postcondition of all methods

Problem: This way,

- invariant conditions bloat the specification,
- invariant conditions are difficult to maintain.

Solution: Class Invariants

Invariant conditions belong to the *object*, not to the actions on object.

Attach invariant conditions to the class, not to methods/constructors.

We call these conditions 'class invariants'.

Constructors/methods of a class are *implicitly* (but firmly!) obliged to establish/maintain invariant conditions of their class.

Solution: Class Invariants

Invariant conditions belong to the *object*, not to the actions on object.

Attach invariant conditions to the class, not to methods/constructors.

We call these conditions 'class invariants'.

Constructors/methods of a class are *implicitly* (but firmly!) obliged to establish/maintain invariant conditions of their class.

Solution: Class Invariants

Invariant conditions belong to the *object*, not to the actions on object.

Attach invariant conditions to the class, not to methods/constructors.

We call these conditions 'class invariants'.

Constructors/methods of a class are *implicitly* (but firmly!) obliged to establish/maintain invariant conditions of their class.

Specification Conditions

in summary: three types of conditions in specifications

- preconditions of methods
- postconditions of methods and constructors
- class invariants[2]

[2] not to be confused with loop invariants, see last part of course

Formal Language for Conditions

We will use the 'Java Modelling Language' (JML) to specify JAVA programs.

JML combines

- JAVA
- First-Order Logic (FOL)

We first introduce First-Order Logic, and JML afterwards.

Formal Language for Conditions

We will use the 'Java Modelling Language' (JML) to specify JAVA programs.

JML combines

- JAVA
- First-Order Logic (FOL)

We first introduce First-Order Logic, and JML afterwards.

First-Order Logic

Signature

A first-order signature Σ consists of

- a set T_Σ of types
- a set F_Σ of function symbols, each with fixed typing
- a set P_Σ of predicate symbols, each with fixed typing
- a typing α_Σ

The *typing* α_Σ assigns

- to each function and predicate symbol:
  - its number of arguments (≥ 0)
  - its argument types
- to each function symbol its result type.

We assume set V of variables ($V \cap (F_\Sigma \cup P_\Sigma) = \emptyset$), each having a unique type.

First-Order Logic

Signature

A first-order signature Σ consists of

- a set T_Σ of types
- a set F_Σ of function symbols, each with fixed typing
- a set P_Σ of predicate symbols, each with fixed typing
- a typing α_Σ

The *typing* α_Σ assigns

- to each function and predicate symbol:
  - its number of arguments (≥ 0)
  - its argument types
- to each function symbol its result type.

We assume set V of variables ($V \cap (F_\Sigma \cup P_\Sigma) = \emptyset$), each having a unique type.

First-Order Terms

terms are defined recursively:

Terms

A first-order term of type $\tau \in T_\Sigma$

- is either a variable of type τ, or
- has the form $f(t_1, \ldots, t_n)$,
  where $f \in F_\Sigma$ has result type τ, and each t_i is term of the correct type, following the typing α_Σ of f.

Atomic Formulae

Logical Atoms

A logical atom has either of the forms

- *true*
- *false*
- $t_1 = t_n$ *("equality")*
- $p(t_1, \ldots, t_n)$ *("predicate")*,
  where $p \in P_\Sigma$, and each t_i is term of the correct type, following the typing α_Σ of p.

General Formulae

first-order formulae are defined recursively:

Formulae

- each atomic formula is a formula
- if ϕ and ψ are formulae, and *x* is a variable, then the following are also formulae:
  - $\neg\phi$ (*"not ϕ"*)
  - $\phi \wedge \psi$ (*"ϕ and ψ"*)
  - $\phi \vee \psi$ (*"ϕ or ψ"*)
  - $\phi \rightarrow \psi$ (*"ϕ implies ψ"*)
  - $\phi \leftrightarrow \psi$ (*"ϕ is equivalent to ψ"*)
  - $\forall\ t\ x.\ \phi$ (*"for all x of type t holds ϕ"*)
  - $\exists\ t\ x.\ \phi$ (*"there exists an x of type t such that ϕ"*)

In a real Logic Course ....

... we now would rigorously define:

- validity of formulae
- provability of formulae (in various calculi)

$\Rightarrow$ see course ‘Logic in Computer Science’

In *our* course, we stick to the intuitive meaning of formulae.

But we mention ‘models’.

Models vs. States

Model

A model assigns *meaning* to the symbols in $F_\Sigma \cup P_\Sigma$
(assigning functions to function symbols, relations to predicate symbols).

In a given model M, a formula is either valid or not valid.

Tautologies

A formula is a tautology if it is valid in all models.

In the context of formal specification of imperative programs:
states take over the role of models.

Models vs. States

Model

A model assigns *meaning* to the symbols in $F_\Sigma \cup P_\Sigma$
(assigning functions to function symbols, relations to predicate symbols).

In a given model M, a formula is either valid or not valid.

Tautologies

A formula is a tautology if it is valid in all models.

In the context of formal specification of imperative programs:
states take over the role of models.

Models vs. States

Model

A model assigns *meaning* to the symbols in $F_\Sigma \cup P_\Sigma$
(assigning functions to function symbols, relations to predicate symbols).

In a given model M, a formula is either valid or not valid.

Tautologies

A formula is a tautology if it is valid in all models.

In the context of formal specification of imperative programs:
states take over the role of models.

Models vs. States

Model

A model assigns *meaning* to the symbols in $F_\Sigma \cup P_\Sigma$
(assigning functions to function symbols, relations to predicate symbols).

In a given model M, a formula is either valid or not valid.

Tautologies

A formula is a tautology if it is valid in all models.

In the context of formal specification of imperative programs:
states[3] take over the role of models.

[3] together with input values and results, and possibly paired with an old states

Good to Remember

useful tautologies: whiteboard

Next Lecture

We will use the 'Java Modelling Language' (JML) to specify JAVA programs.

JML combines

- First-Order Logic (FOL)
- JAVA