Formal Specification and Verification

Bernhard Beckert

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

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

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

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

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

A) Expressing Sortedness

A `SortedIntegers` object is sorted if:

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

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

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

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[1]

[1] 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 $f(t_1, \dots, t_n)$,
  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[2] take over the role of models.

[2] 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