Formal Methods and Specification (SS 2021) Modeling of Data Structures 2 and Object Oriented Programming

Stefan Ratschan

Katedra číslicového návrhu
Fakulta informačních technologií
České vysoké učení technické v Praze

Evropský sociální fond Praha & EU: Investujeme do vaší budoucnosti

Abstract Data Type

A data type with certain operations that is
independent of any implementation.

Description of abstract data types:
signature + axioms

Example:

Signature:
init: $\mathcal{N} \rightarrow$ counter
val: counter $\rightarrow \mathcal{N}$
dec: counter $\rightarrow$ counter

Axioms:

$$\forall n \in \mathcal{N} \ . \ val(init(n)) = n$$

$$\forall c \in \text{counter}, n \in \mathcal{N} \ . \ \left[\begin{array}{l} val(c) = 0 \Rightarrow val(dec(c)) = 0 \wedge \\ val(c) \neq 0 \Rightarrow val(dec(c)) = val(c) - 1 \end{array} \right]$$

This is nothing else than a logical theory! C++ header files, Java interfaces

Object Oriented Programming

$\approx$ abstract data types + inheritance + dynamic binding + strange notation

Example: notation for signature

init: $\mathcal{N} \to$ counter
val: counter $\to \mathcal{N}$
dec: counter $\to$ counter

class counter
constructor init(n)
method val(): $\mathcal{N}$
method dec()

Usage example:

$$i \leftarrow c.val()$$

meaning

$$i \leftarrow val(c)$$

Writing Axioms as Assertions

$$\forall n \in \mathcal{N} \,.\, val(init(n)) = n$$

$$\forall c \in \text{counter} \,.\, \left[\begin{array}{l} val(c) = 0 \Rightarrow val(dec(c)) = 0 \wedge \\ val(c) \neq 0 \Rightarrow val(dec(c)) = val(c) - 1 \end{array} \right]$$

class counter
constructor init(n)
    assume $n \in \mathcal{N}$
    ...
    @ val() $= n$
method val(): $\mathcal{N}$
method dec()
    $v \leftarrow val()$
    ...
    @ $[v = 0 \Rightarrow \text{val}() = 0] \wedge [v \neq 0 \Rightarrow \text{val}() = v - 1]$

Verification Conditions in Natural Notation

Assumptions from specification:

$$\forall n \in \mathcal{N} \, . \, val(init(n)) = n$$

$$\forall c \in \text{counter} \, . \, \left[\begin{array}{l} val(c) = 0 \Rightarrow val(dec(c)) = 0 \wedge \\ val(c) \neq 0 \Rightarrow val(dec(c)) = val(c) - 1 \end{array} \right]$$

$c \leftarrow \text{init}(2)$	// $c \leftarrow \text{init}(2)$
@ $c.\text{val}() = 2$	// @ $\text{val}(c) = 2$
$v \leftarrow c.\text{val}()$	// $v \leftarrow \text{val}(c)$
$c.\text{dec}()$	// $c \leftarrow \text{dec}(c)$
@ $c.\text{val}() = v - 1$	// @ $\text{val}(c) = v - 1$

Verification conditions:

- $c = \text{init}(2) \Rightarrow \text{val}(c) = 2$
- $[\text{val}(c) = 2 \wedge v = \text{val}(c) \wedge c_1 = \text{dec}(c)] \Rightarrow \text{val}(c_1) = v - 1$

Verification Conditions in Strange Notation

Assumptions from the specification (dots in the formulas are only notation)

$$\forall n \in \mathcal{N} \ . \ init(n).val() = n$$

$$\forall c \ . \ \left[\begin{array}{l} c.val() = 0 \Rightarrow c.dec().val() = 0 \ \wedge \\ c.val() \neq 0 \Rightarrow c.dec().val() = c.val() - 1 \end{array} \right]$$

$c \leftarrow \mathrm{init}(2)$	
@ $c.\mathrm{val}() = 2$	// @ $\mathrm{val}(c) = 2$
$v \leftarrow c.\mathrm{val}()$	// $v \leftarrow \mathrm{val}(c)$
$c.\mathrm{dec}()$	// $c \leftarrow \mathrm{dec}(c)$
@ $c.\mathrm{val}() = v - 1$	// @ $\mathrm{val}(c) = v - 1$

Verification conditions:

- $c = \mathrm{init}(2) \Rightarrow c.val() = 2$
- $[c.\mathrm{val}() = 2 \wedge v = c.\mathrm{val}() \wedge c_1 = c.\mathrm{dec}()] \Rightarrow c_1.\mathrm{val}() = v - 1$

Internal Consistency of Data Types

Example:
The internal variable x should always fulfill $x \geq 0$.

This would amount to an assertion at
the beginning and end of every method.

Some object oriented languages have explicit support:

Eiffel: **invariant** $x \geq 0$,
is checked at the beginning and end of every method

Everything we have seen up to now,
can also be expressed (maybe even better) without OO constructions.

see also: design by contract
(Eiffel, Microsoft Code Contracts, Java Modeling Language)

Abstract Data Types

Rest of lecture:

- example
- subtyping

Example: Axiomatic Specification of FIFO Queue

Signature:

emptyq: $\rightarrow \mathcal{Q}$
enqueue: $\mathcal{N} \times \mathcal{Q} \rightarrow \mathcal{Q}$
get: $\mathcal{Q} \rightarrow \mathcal{N}$
dequeue: $\mathcal{Q} \rightarrow \mathcal{Q}$

Axioms:

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \; . \; \mathrm{enqueue}(x, q) \neq \mathrm{emptyq}()$$

$$\forall x \in \mathcal{N} \; . \; \mathrm{get}(\mathrm{enqueue}(x, \mathrm{emptyq}())) = x$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \; . \; q \neq \mathrm{emptyq}() \Rightarrow \mathrm{get}(\mathrm{enqueue}(x, q)) = \mathrm{get}(q)$$

$$\forall x \in \mathcal{N} \; . \; \mathrm{dequeue}(\mathrm{enqueue}(x, \mathrm{emptyq}())) = \mathrm{emptyq}()$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \; . \; q \neq \mathrm{emptyq}() \Rightarrow \\ \mathrm{dequeue}(\mathrm{enqueue}(x, q)) = \mathrm{enqueue}(x, \mathrm{dequeue}(q))$$

This leaves the behavior of dequeue/get on an empty queue open.

Application

Axioms:

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \;.\; \mathrm{enqueue}(x, q) \neq \mathrm{emptyq}()$$

$$\forall x \in \mathcal{N} \;.\; \mathrm{get}(\mathrm{enqueue}(x, \mathrm{emptyq}())) = x$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \;.\; q \neq \mathrm{emptyq}() \Rightarrow \mathrm{get}(\mathrm{enqueue}(x, q)) = \mathrm{get}(q)$$

$$\forall x \in \mathcal{N} \;.\; \mathrm{dequeue}(\mathrm{enqueue}(x, \mathrm{emptyq}())) = \mathrm{emptyq}()$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \;.\; q \neq \mathrm{emptyq}() \Rightarrow \mathrm{dequeue}(\mathrm{enqueue}(x, q)) = \mathrm{enqueue}(x, \mathrm{dequeue}(q))$$

Expression simplification:

$$\mathrm{get}(\mathrm{enqueue}(7, \mathrm{enqueue}(3, \mathrm{emptyq}()))))) = \mathrm{get}(\mathrm{enqueue}(3, \mathrm{emptyq}())) = 3$$

and further proofs of behavior of data type, independent of implementation

Implementing Abstract Data Types

Let Σ be a signature over types $T_1, \dots, T_n$.

A Σ-*structure* S is a tuple $(\Omega_1, \dots, \Omega_n, \mathcal{I})$ where

- $\Omega_1, \dots, \Omega_n$ are sets (the *carrier sets* of S)
- $\mathcal{I}$ assigns to
  - every function symbol $f : T_{i_1} \times \cdots \times T_{i_a} \to T_j$ from Σ
    a function $f : \Omega_{i_1} \times \cdots \times \Omega_{i_a} \to \Omega_j$
  - every predicate symbol $p : T_{i_1} \times \cdots \times T_{i_a}$ from Σ
    a relation $r \subseteq \Omega_{i_1} \times \cdots \times \Omega_{i_a}$

Intuition (Java, C++):

- structure: concrete class
- elements of $\Omega_1, \dots, \Omega_n$: objects

We will concentrate on structures without predicates (except for $=$)

Notation: $(\Omega_1, \dots, \Omega_n, \mathcal{I}) \models \phi$: Σ-formula ϕ holds in $(\Omega_1, \dots, \Omega_n, \mathcal{I})$

Example: $1 + 1 = 0$ holds in the structure $\mathbb{Z}_2$

Semantics of ADT

A Σ-structure $(\Omega_1, \ldots, \Omega_n, \mathcal{I})$ is a *model* of a data type T iff

- T has the signature Σ, and
- for every axiom ϕ of T, $(\Omega_1, \ldots, \Omega_n, \mathcal{I}) \models \phi$.

Example: the rational numbers are a model of group theory

Intuition: ADT: interface, model: implementation

Attention: the term "model" usually means something different!

Extensions of Data Types

Sometimes we want to add properties to an existing specification, e.g.:

- add axiom dequeue(emptyQ()) = emptyQ()
- restrict groups to commutative groups

Given two data types $(\check{\Sigma}, \check{A})$ and $(\hat{\Sigma}, \hat{A})$,
$(\check{\Sigma}, \check{A})$ is an *extension* of $(\hat{\Sigma}, \hat{A})$ iff $\check{\Sigma} \supseteq \hat{\Sigma}$ and $\check{A} \supseteq \hat{A}$

Notation: $(\check{\Sigma}, \check{A}) \preceq (\hat{\Sigma}, \hat{A})$

For two data types $(\check{\Sigma}, \check{A}) \preceq (\hat{\Sigma}, \hat{A})$
for every model $\check{S}$ of $(\check{\Sigma}, \check{A})$,
the restriction of $\check{S}$ to $\hat{\Sigma}$ is a model of $(\hat{\Sigma}, \hat{A})$

Intuition:
every implementation of $(\check{\Sigma}, \check{A})$ is also an implementation of $(\hat{\Sigma}, \hat{A})$

something similar in object-oriented programming?

Subtyping vs. Theory Extension

Subtypes should ensure the same properties as the supertype
when used in the same way

Liskov substitution principle:

> *"Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T". [Liskov and Wing, 1994]*

In OO programming languages only partially ensured,
syntactically (e.g., rules of co/contravariance)

Theory extensions guarantee this, but
do not take into account usual phenomena of OO languages

Definition of Abstract Data Types by Extensions

Up to now ADT defined purely intensionally
(based on their properties)

But we already know many data types (i.e., logical theories)

Why start from scratch?

Build new data type based on existing ones?

Example: FIFO Queue by Extension

We define queues $\mathcal{Q}$ by extending lists $\mathcal{L}$

Signature: $\mathcal{L}+$

emptyq: $\rightarrow \mathcal{Q}$

enqueue: $\mathcal{N} \times \mathcal{Q} \rightarrow \mathcal{Q}$

get: $\mathcal{Q} \rightarrow \mathcal{N}$

dequeue: $\mathcal{Q} \rightarrow \mathcal{Q}$

Axioms: $\mathcal{L}+$

$$\mathrm{emptyq}() = \langle\rangle$$

$$\forall x \in \mathcal{N}, \forall q \in \mathcal{Q} \;.\; \mathrm{enqueue}(x, q) = \mathrm{cons}(x, q)$$

$$\forall q \in \mathcal{Q} \;.\; q \neq \langle\rangle \Rightarrow \mathrm{get}(q) = \mathrm{last}(q)$$

$$\forall q \in \mathcal{Q} \;.\; q \neq \langle\rangle \Rightarrow \mathrm{dequeue}(q) = \mathrm{dropLast}(q)$$

where last, dropLast are defined in $\mathcal{L}$, e.g. as follows:

$$\mathrm{last}(< x >) = x, l \neq \langle\rangle \Rightarrow \mathrm{last}(\mathrm{cons}(x, l)) = \mathrm{last}(l)$$

$$\mathrm{dropLast}(< x >) = <>, l \neq \langle\rangle \Rightarrow \mathrm{dropLast}(\mathrm{cons}(x, l)) = \mathrm{dropLast}(l)$$

Application: Expression Simplification

$$\text{emptyq}() = \langle\rangle$$

$$\forall x \in \mathcal{N}, \forall q \in \mathcal{Q}\ .\ \text{enqueue}(x, q) = \text{cons}(x, q)$$

$$\forall q \in \mathcal{Q}\ .\ q \neq \langle\rangle \Rightarrow \text{get}(q) = \text{last}(q)$$

$$\forall q \in \mathcal{Q}\ .\ q \neq \langle\rangle \Rightarrow \text{dequeue}(q) = \text{dropLast}(q)$$

Expression simplification:

$$\begin{aligned} \text{get}(\text{enqueue}(7, \text{enqueue}(3, \text{emptyq}()))) &= \\ \text{get}(\text{enqueue}(7, \text{enqueue}(3, <>))) &= \\ \text{get}(\text{enqueue}(7, < 3 >)) &= \\ \text{get}(< 7, 3 >) &= 3 \end{aligned}$$

Comparison

Intensional:

$$\begin{aligned} \mathrm{get}(\mathrm{enqueue}(7, \mathrm{enqueue}(3, \mathrm{emptyq}()))))) &= \\ \mathrm{get}(\mathrm{enqueue}(3, \mathrm{emptyq}())) &= 3 \end{aligned}$$

Extension Based:

$$\begin{aligned} \mathrm{get}(\mathrm{enqueue}(7, \mathrm{enqueue}(3, \mathrm{emptyq}()))) &= \\ \mathrm{get}(\mathrm{enqueue}(7, \mathrm{enqueue}(3, <>))) &= \\ \mathrm{get}(\mathrm{enqueue}(7, < 3 >)) &= \\ \mathrm{get}(< 7, 3 >) &= 3 \end{aligned}$$

Intensional vs. Extending Definition of ADT

Intensional axioms:

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \, . \, \mathsf{enqueue}(x, q) \neq \mathsf{emptyq}()$$

$$\forall x \in \mathcal{N} \, . \, \mathsf{get}(\mathsf{enqueue}(x, \mathsf{emptyq}())) = x$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \, . \, q \neq \mathsf{emptyq}() \Rightarrow \mathsf{get}(\mathsf{enqueue}(x, q)) = \mathsf{get}(q)$$

$$\forall x \in \mathcal{N} \, . \, \mathsf{dequeue}(\mathsf{enqueue}(x, \mathsf{emptyq}())) = \mathsf{emptyq}()$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \, . \, q \neq \mathsf{emptyq}() \Rightarrow \\ \mathsf{dequeue}(\mathsf{enqueue}(x, q)) = \mathsf{enqueue}(x, \mathsf{dequeue}(q))$$

Extending axioms:

$$\mathsf{emptyq}() = \langle\rangle$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \, . \, \mathsf{enqueue}(x, q) = \mathsf{cons}(x, q)$$

$$\forall q \in \mathcal{Q} \, . \, q \neq \langle\rangle \Rightarrow \mathsf{get}(q) = \mathsf{last}(q)$$

$$\forall q \in \mathcal{Q} \, . \, q \neq \langle\rangle \Rightarrow \mathsf{dequeue}(q) = \mathsf{dropLast}(q)$$

Does the extending definition fulfill the intensional axioms?

Correctness Proof

Axiom:

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \;.\; \mathrm{enqueue}(x, q) \neq \mathrm{emptyq}()$$

List-based spec:

$$\mathrm{emptyq}() = \langle\rangle$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \;.\; \mathrm{enqueue}(x, q) = \mathrm{cons}(x, q)$$

$$\forall q \in \mathcal{Q} \;.\; q \neq \langle\rangle \Rightarrow \mathrm{get}(q) = \mathrm{last}(q)$$

$$\forall q \in \mathcal{Q} \;.\; q \neq \langle\rangle \Rightarrow \mathrm{dequeue}(q) = \mathrm{dropLast}(q)$$

Proof:

$$\mathrm{enqueue}(x, q) = \mathrm{cons}(x, q) \neq \langle\rangle = \mathrm{emptyq}()$$

Correctness Proof

Axiom:

$$\forall x \in \mathcal{N} \,.\, \mathrm{get}(\mathrm{enqueue}(x, \mathrm{emptyq}())) = x$$

List-based spec:

$$\mathrm{emptyq}() = \langle\rangle$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \,.\, \mathrm{enqueue}(x, q) = \mathrm{cons}(x, q)$$

$$\forall q \in \mathcal{Q} \,.\, q \neq \langle\rangle \Rightarrow \mathrm{get}(q) = \mathrm{last}(q)$$

$$\forall q \in \mathcal{Q} \,.\, q \neq \langle\rangle \Rightarrow \mathrm{dequeue}(q) = \mathrm{dropLast}(q)$$

Proof:

$$\mathrm{get}(\mathrm{enqueue}(x, \mathrm{emptyq}())) =$$
$$\mathrm{get}(\mathrm{enqueue}(x, <>) = \mathrm{get}(< x >) = \mathrm{last}(< x >) = x$$

Correctness Proof

Axiom:

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \,.\, q \neq \text{emptyq}() \Rightarrow \text{get}(\text{enqueue}(x, q)) = \text{get}(q)$$

List-based spec:

$$\text{emptyq}() = \langle\rangle$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \,.\, \text{enqueue}(x, q) = \text{cons}(x, q)$$

$$\forall q \in \mathcal{Q} \,.\, q \neq \langle\rangle \Rightarrow \text{get}(q) = \text{last}(q)$$

$$\forall q \in \mathcal{Q} \,.\, q \neq \langle\rangle \Rightarrow \text{dequeue}(q) = \text{dropLast}(q)$$

Proof:
Assume $q \neq \text{emptyq}()$
Prove: $\text{get}(\text{enqueue}(x, q)) = \text{last}(\text{cons}(x, q)) = \text{last}(q) = \text{get}(q)$

Implementation

We defined $\mathcal{Q}$ by extending $\mathcal{L}$

Must implementations of this ADT use lists?

$$\text{emptyq}() = \langle\rangle$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \;.\; \text{enqueue}(x, q) = \text{cons}(x, q)$$

$$\forall q \in \mathcal{Q} \;.\; q \neq \langle\rangle \Rightarrow \text{get}(q) = \text{last}(q)$$

$$\forall q \in \mathcal{Q} \;.\; q \neq \langle\rangle \Rightarrow \text{dequeue}(q) = \text{dropLast}(q)$$

We implicitely assumed $\mathcal{Q}$ and $\mathcal{L}$ to be identical.

Now: Drop this assumption.

Demonstration based on example ...

Implementation Based on Arrays

Intuition: Represent the list by an array, e.g. $\langle 1, 2, 3, 4, 5 \rangle$ by

4	5				1	2	3
		n			m		

More concretely: $\mathcal{Q} := \mathcal{A} \times \mathcal{N} \times \mathcal{N}$, where

(a, m, n) represents $\langle a[m], a[(m+1) \bmod l], \dots, a[n-1] \rangle$,

for some constant l.

Formally: $$\rho(a, m, n) := \begin{cases} \langle\rangle, & \text{if } m = n, \text{ and} \\ cons(a[m], \rho(a, (m+1) \bmod l, n)), & \text{otherwise.} \end{cases}$$

Implementation:

- emptyq() $:= (a, 0, 0)$
- enqueue(x, (a, m, n)) $:= (write(a, m^-, x), m^-, n)$
- get(a, m, n) $:= a[n^-]$
- dequeue(a, m, n) $:= (a, m, n^-)$

where $i^- := (i-1) \bmod l$

Correct?

Checking Correctness

List-based spec:

$$\text{emptyq}() = \langle\rangle$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \;.\; \text{enqueue}(x, q) = \text{cons}(x, q)$$

$$\forall q \in \mathcal{Q} \;.\; q \neq \langle\rangle \Rightarrow \text{get}(q) = \text{last}(q)$$

$$\forall q \in \mathcal{Q} \;.\; q \neq \langle\rangle \Rightarrow \text{dequeue}(q) = \text{dropLast}(q)$$

Partially lists, partially array representation

Conversion possible! representation function ρ

Checking Correctness

List-based spec:

$$\rho(\mathsf{emptyq}()) = \langle\rangle$$

$$\forall x \in \mathcal{N}, q \in \mathcal{Q} \;.\; \rho(\mathsf{enqueue}(x, q)) = \mathsf{cons}(x, \rho(q))$$

$$\forall q \in \mathcal{Q} \;.\; \rho(q) \neq \langle\rangle \Rightarrow \mathsf{get}(q) = \mathsf{last}(\rho(q))$$

$$\forall q \in \mathcal{Q} \;.\; \rho(q) \neq \langle\rangle \Rightarrow \rho(\mathsf{dequeue}(q)) = \mathsf{dropLast}(\rho(q))$$

Now: $\mathcal{Q}$:our array based implementation

Checking Correctness

List-based spec:

$$\rho(\mathrm{emptyq}()) = \langle\rangle$$

$$\forall x \in \mathcal{N}, (a, m, n) \in \mathcal{Q} \;.\; \rho(\mathrm{enqueue}(x, (a, m, n))) = \mathrm{cons}(x, \rho(a, m, n))$$

$$\forall (a, m, n) \in \mathcal{Q} \;.\; \rho(a, m, n) \neq \langle\rangle \Rightarrow \mathrm{get}(a, m, n) = \mathrm{last}(\rho(a, m, n))$$

$$\forall (a, m, n) \in \mathcal{Q} \;.\; \rho(a, m, n) \neq \langle\rangle \Rightarrow \rho(\mathrm{dequeue}(a, m, n)) = \mathrm{dropLast}(\rho(a, m, n)$$

Let us check this.

$\rho(\mathrm{emptyq}()) = \langle\rangle$

$$\rho(\mathrm{emptyq}()) = \rho(a, 0, 0) = \langle\rangle$$

$\rho(\text{enqueue}(x, (a, m, n))) = \text{cons}(x, \rho(a, m, n))$

By definition,

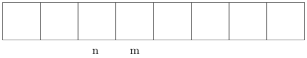
$$\rho(\text{enqueue}(x, (a, m, n))) = \rho(\mathit{write}(a, m^-, x), m^-, n).$$

According to the definition of ρ, if $m^- \neq n$, this is

$$\mathit{cons}(\mathit{write}(a, m^-, x)[m^-], \rho(a, (m^- + 1) \bmod l, n)),$$

which, due to the array axioms and modular arithmetic is

$$\text{cons}(x, \rho(a, m, n)).$$

If $m^- = n$? $\langle\rangle$? 

bug (overflow)

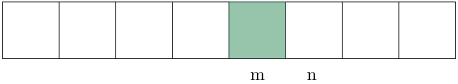
$$\rho(a, m, n) := \begin{cases} \langle\rangle, & \text{if } m = n, \text{ and} \\ \mathit{cons}(a[m], \rho(a, (m+1) \bmod l, n)), & \text{otherwise.} \end{cases}$$

$\rho(a, m, n) \neq \langle\rangle \Rightarrow \mathrm{get}(a, m, n) = \mathrm{last}(\rho(a, m, n))$

We assume $(a, m, n) \neq \langle\rangle$, and prove $get(a, m, n) = \mathrm{last}(\rho(a, m, n))$

By definition, $\mathrm{get}(\mathrm{a}, \mathrm{m}, \mathrm{n}) = a[n^-]$.

We prove that this is equal to $\mathrm{last}(\rho(a, m, n))$.



If $m = n^-$,

$$\mathrm{last}(\rho(a, m, n)) = \mathrm{last}(cons(a[m], \rho(a, n, n)) = \\ = \mathrm{last}(cons(a[m], \langle\rangle) = a[m] = a[n^-].$$

Now we proceed by induction: We assume

$$\mathrm{last}(\rho(a, m, n)) = a[n^-],$$

and prove

$$\mathrm{last}(\rho(a, m^-, n)) = a[n^-].$$

Induction Step

We assume

$$\mathrm{last}(\rho(a, m, n)) = a[n^-],$$

and prove

$$\mathrm{last}(\rho(a, m^-, n)) = a[n^-].$$

We have:

$$\mathrm{last}(\rho(a, m^-, n)) = \mathrm{last}(\mathrm{cons}(a[m^-], \rho(a, m, n))) =$$
$$= last(\rho(a, m, n)) = a[n^-],$$

which is, what we wanted to prove.

Wait! $m^- = n$: overflow!

$\rho(a, m, n) \neq \langle\rangle \Rightarrow \rho(\text{dequeue}(a, m, n)) = \text{dropLast}(\rho(a, m, n))$

similar

Methods for Specification of Data Types

- Property based:
  - axiomatic
  - algebraic (axioms restricted to equalities)

  Example: Alloy (`https://alloytools.org`)
- Model based: list possible behavior instead of describing properties
  Examples:
  - Z notation
  - Vienna Development Method (VDM)
  - TLA+ (Amazon, Microsoft)

Conclusion

OO $\approx$ abstract data types $=$ logical theories

Literature I

Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.