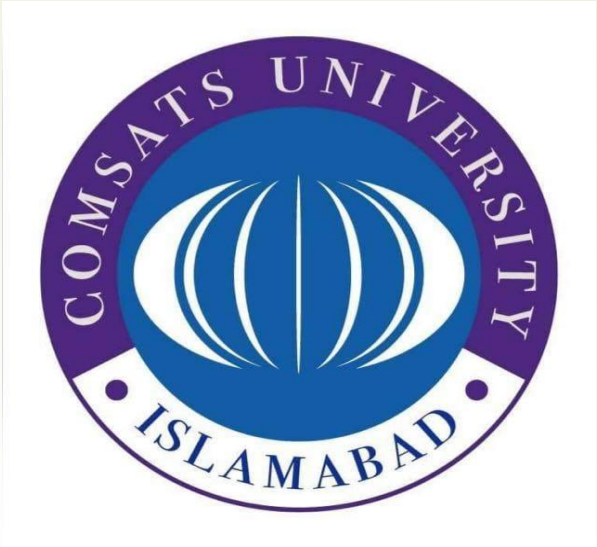


Department of Computer Science, CUI Lahore Campus

Formal Methods

By

Farooq Ahmad

Case study: orders in stocks

Sets, partial functions and bags as system state variables

Topics covered

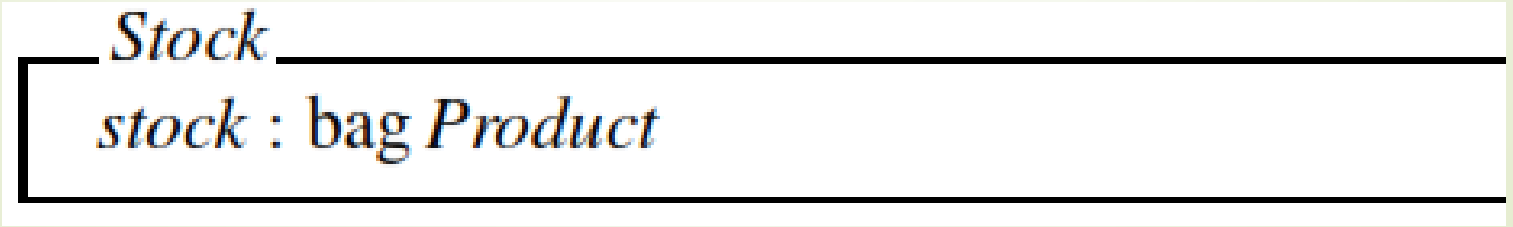
- Case study: Orders in Stock
- Partial function as state variable
- Bags as system state variable
- Free type definition
- Error report schemas

System's Requirements

- A user may wish to order several types of product at once and this should be discussed with the customer. Here, we assume that the customer decides to allow orders of one *or more products for extra flexibility, but not* empty orders (i.e., an order for no products).
- Since *stock is defined as a bag of products, it is convenient to define an order as a* bag of products too
- However, whereas the stock may be completely empty, an order must consist of one (or more) products.

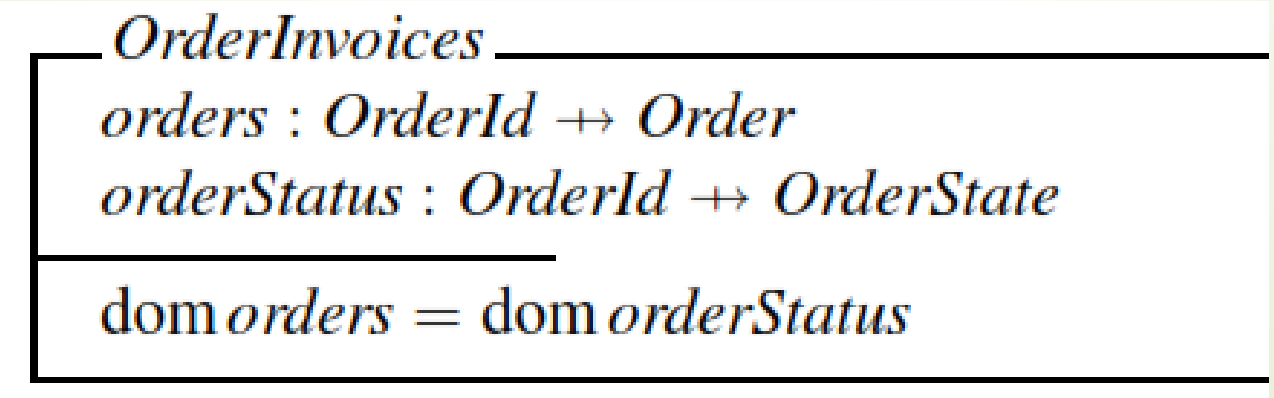
Basic types

- Here, we define sets of order identifiers and products which can potentially be held in stock:
- **[*OrderId*, *Product*]**
- *Order == {order : bag Product | order ≠ Ø}*
- The orders have two states, i.e. Pending and invoiced.
- ***OrderState* ::= *pending* | *invoiced***
- *We need to model the state products in stock and orders including* their invoicing status.



System state variables

- Continuing with the definition of the abstract model, the **status** of orders can be modeled as a function from an identifying *OrderId to their state (pending or invoiced)*.
- State components *orderStatus* and orders are packaged into an *OrderInvoices schema* with appropriate *type information:*



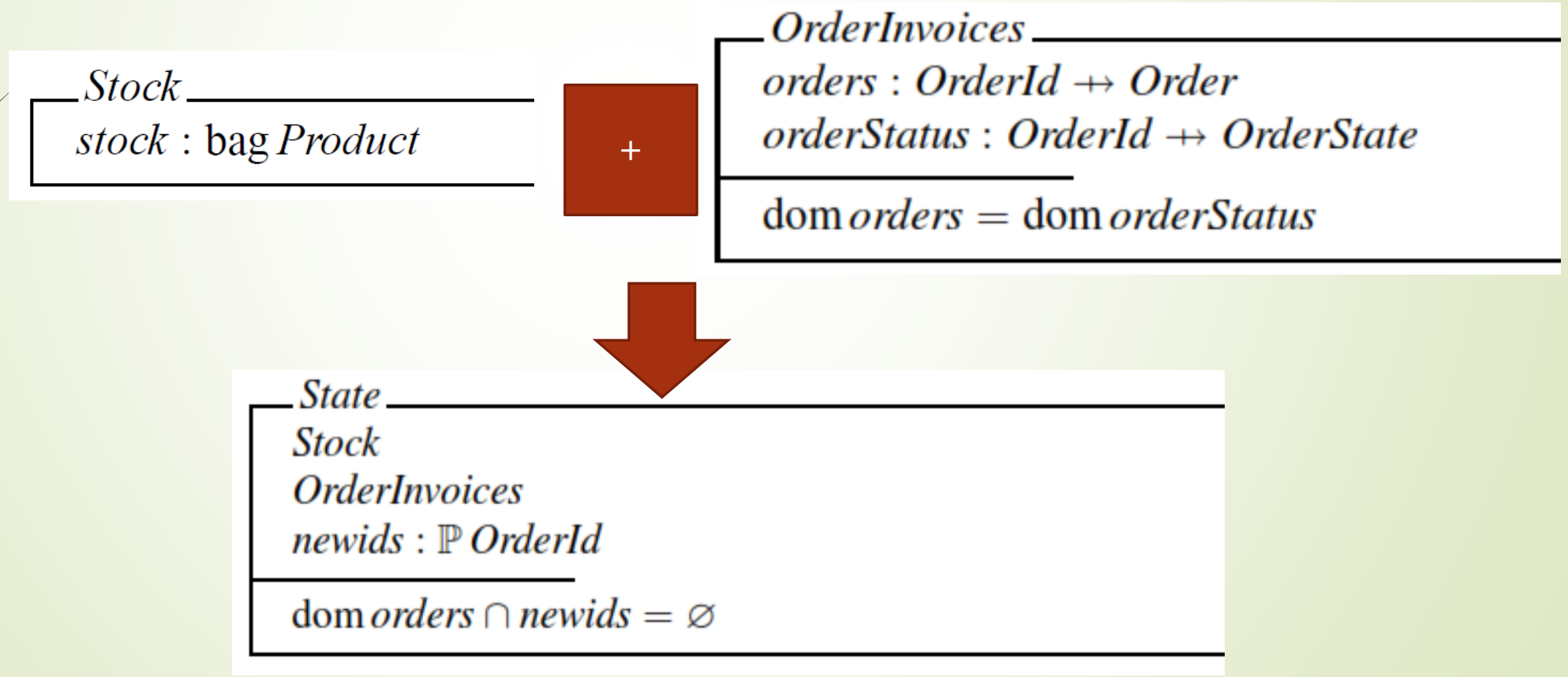
Requirement for order identifier

- Should order identifiers be unique for the entire lifetime of the system?
- **We could decide that new identifiers must never have been used previously** or that they just need to be unique at any given time.
- The state specification so far assumes the former, which is easiest.

System's state

Order: bag *Product*

Order == {order : bag Product | order ≠ ∅}



System state schema

State

$stock$: bag $Product$

$orders$: $OrderId \pfun Order$

$orderStatus$: $OrderId \pfun OrderState$

$newids$: $\mathbb{P} OderId$

$\mathrm{dom}\ orders = \mathrm{dom}\ orderStatus$

$\mathrm{dom}\ orders \cap newids = \varnothing$

InitState

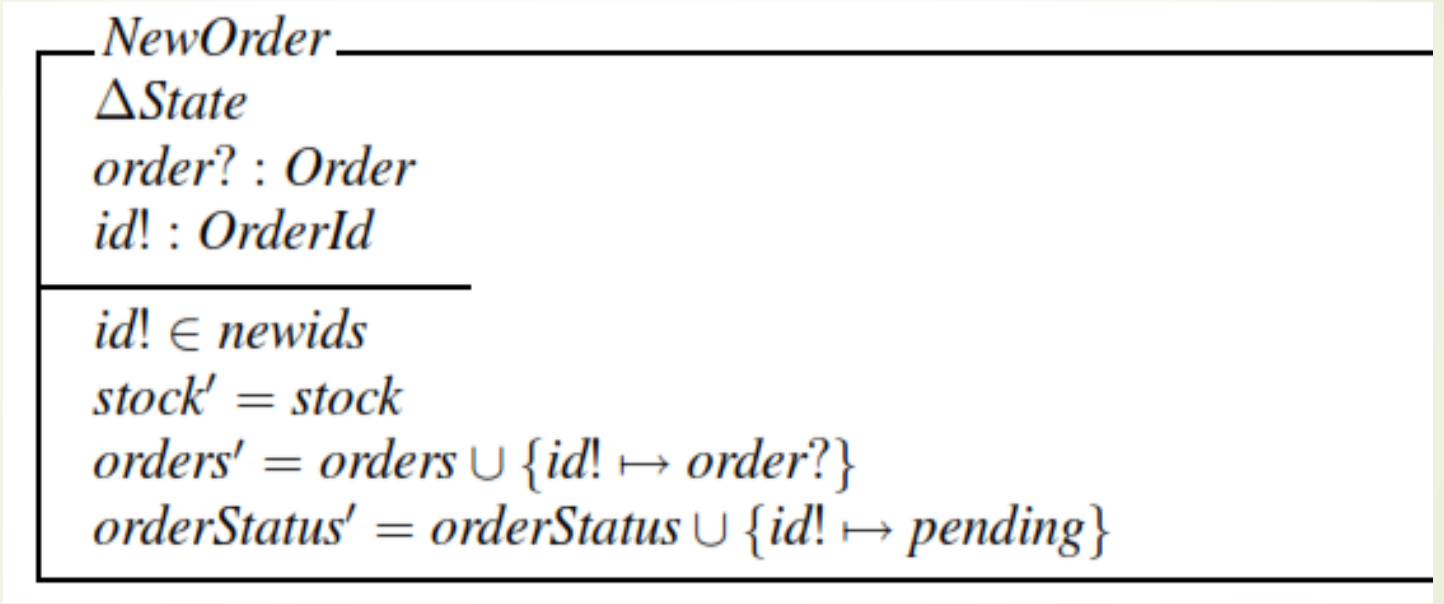
$State'$

$stock' = \varnothing$

$orders' = \varnothing$

$newids' = OrderId$

Operation schema: New order



Operation schema: Invoice Order

InvoiceOrder

$\Delta State$
$id? : OrderId$

$orders(id?) \sqsubseteq stock$
$orderStatus(id?) = pending$
$stock' = stock \uplus orders(id?)$
$orders' = orders$
$orderStatus' = orderStatus \oplus \{id? \mapsto invoiced\}$

$\sqsubseteq$ is the sub-bag relational operator from the Z toolkit. As used in the schema above, this ensures a precondition that there are enough quantities of the required product(s) in stock. For example, $\{nuts \mapsto 3\} \sqsubseteq \{nuts \mapsto 5, bolts \mapsto 6\}$ is *true*.

Another precondition is that the status of the order must be *pending*. If the preconditions are satisfied, the required product quantities are removed from the available stock using the bag difference operator ('$\uplus$', cf. the set difference operator '$\setminus$' for sets). Here, for example, $\{nuts \mapsto 5, bolts \mapsto 6\} \uplus \{nuts \mapsto 3\}$ would result in $\{nuts \mapsto 2, bolts \mapsto 6\}$.

Operation schemas

CancelOrder

$\Delta State$

$id? : OrderId$

$orderStatus(id?) = pending$

$stock' = stock$

$orders' = \{id?\} \lhd orders$

$orderStatus' = \{id?\} \lhd orderStatus$

EnterStock

$\Delta State$

$newstock? : \text{bag } Product$

$stock' = stock \uplus newstock?$

$orders' = orders$

$orderStatus' = orderStatus$

Error scenario: Invoice Error

$$Report ::= OK \mid order\_not\_pending \mid not\_enough\_stock \mid no\_more\_ids$$

InvoiceError

$\Xi State$

$id? : OrderId$

$rep! : Report$

$orderStatus(id?) \neq pending$

$rep! = order\_not\_pending$

Error scenario: Stock error

StockError

$\Xi State$

$id? : OrderId$

$rep! : Report$

$\neg\ orders(id?) \sqsubseteq stock$

$rep! = not\_enough\_stock$

Summary of the lecture: conclusion

- Case study using partial functions and bags as state variables
- Software specification using different state variables

Reference and reading material

- Chapter 1 of the book "Software Specification Methods" published by ISTE Ltd
- Chapter 5: Section 5.3 of the book "Software Development with Z"