Data-Stack Theory

syntax

stack	all stacks of items of type *X*
empty	a stack containing no items
push	a function that takes a stack and an item and gives back another stack
pop	a function that takes a stack and gives back another stack
top	a function that takes a stack and gives back an item

Data-Stack Theory

axioms

$empty$: $stack$

$push$: $stack \to X \to stack$

pop: $stack \to stack$

top: $stack \to X$

$$\begin{tikzcd}
empty \arrow[r] & s1 \arrow[r] & s2 \arrow[r] & s3 \arrow[r] & s4 \arrow[llll, bend left]
\end{tikzcd}$$

Data-Stack Theory

axioms

$empty: stack$

$push: stack \rightarrow X \rightarrow stack$

$pop: stack \rightarrow stack$

$top: stack \rightarrow X$

$push\ s\ x \neq empty$

$$\begin{tikzcd} empty \arrow[r] & s1 \arrow[r] & s2 \arrow[r] & s3 \arrow[r] & s4 \arrow[ll, bend left] \end{tikzcd}$$

Data-Stack Theory

axioms

$empty$: $stack$

$push$: $stack \to X \to stack$

pop: $stack \to stack$

top: $stack \to X$

$push\ s\ x \neq empty$

$push\ s\ x = push\ t\ y \;\; \equiv \;\; s=t \wedge x=y$

$$empty \to s1 \to s2 \to s3 \to s4 \to \ldots\ldots\ldots\ldots$$

$$\ldots\ldots\ldots\ldots \to t \to u \to v \to w \to \ldots\ldots\ldots\ldots$$

Data-Stack Theory

axioms

$empty: stack$

$push: stack \to X \to stack$

$pop: stack \to stack$

$top: stack \to X$

$push\ s\ x \neq empty$

$push\ s\ x = push\ t\ y\ \ =\ \ s{=}t \wedge x{=}y$

$empty, push\ stack\ X: stack$

$empty, push\ B\ X: B\ \ \Rightarrow\ \ stack: B$

$$empty \to s1 \to s2 \to s3 \to s4 \to \ldots\ldots\ldots$$

Data-Stack Theory

axioms

$empty: stack$

$push: stack \to X \to stack$

$pop: stack \to stack$

$top: stack \to X$

$push\ s\ x \neq empty$

$push\ s\ x = push\ t\ y \quad = \quad s=t \land x=y$

$empty, push\ stack\ X: stack$

$empty, push\ B\ X: B \quad \Rightarrow \quad stack: B$

$P\ empty \land \forall s: stack \cdot \forall x: X \cdot Ps \Rightarrow P(push\ s\ x) \quad = \quad \forall s: stack \cdot Ps$

Data-Stack Theory

axioms

$empty: stack$

$push: stack \rightarrow X \rightarrow stack$

$pop: stack \rightarrow stack$

$top: stack \rightarrow X$

$push\ s\ x \neq empty$

$push\ s\ x = push\ t\ y \quad = \quad s=t \land x=y$

$empty, push\ stack\ X: stack$

$empty, push\ B\ X: B \quad \Rightarrow \quad stack: B$

$P\ empty \land \forall s: stack\cdot\ \forall x: X\cdot\ Ps \Rightarrow P(push\ s\ x) \quad = \quad \forall s: stack\cdot\ Ps$

$pop\ (push\ s\ x) = s$

$top\ (push\ s\ x) = x$

Data-Stack Theory

implementation

stack	=	[**int*]
empty	=	[*nil*]
push	=	$\langle s\text{: } stack \rightarrow \langle x\text{: } int \rightarrow s;;[x]\rangle\rangle$
pop	=	$\langle s\text{: } stack \rightarrow$ **if** $s{=}empty$ **then** *empty* **else** $s\ [0;..\#s{-}1]$ **fi**$\rangle$
top	=	$\langle s\text{: } stack \rightarrow$ **if** $s{=}empty$ **then** 0 **else** $s\ (\#s{-}1)$ **fi**$\rangle$

Data-Stack Theory

proof

Prove that the axioms of the theory are satisfied by the definitions of the implementation.

(the axioms of the theory) $\Leftarrow$ (the definitions of the implementation)

specification $\Leftarrow$ implementation

Data-Stack Theory

proof (last axiom):

	$top\ (push\ s\ x) = x$	definition of *push*
$=$	$top\ (\langle s\colon stack \rightarrow \langle x\colon int \rightarrow s;;[x]\rangle\rangle\ s\ x) = x$	apply function
$=$	$top\ (s;;[x]) = x$	definition of *top*
$=$	$\langle s\colon stack \rightarrow \textbf{if}\ s{=}empty\ \textbf{then}\ 0\ \textbf{else}\ s\ (\#s{-}1)\ \textbf{fi}\rangle\ (s;;[x]) = x$	apply function
$=$	$\textbf{if}\ s;;[x]{=}empty\ \textbf{then}\ 0\ \textbf{else}\ (s;;[x])\ (\#(s;;[x]){-}1)\ \textbf{fi} = x$	definition of *empty*
$=$	$\textbf{if}\ s;;[x]{=}[nil]\ \textbf{then}\ 0\ \textbf{else}\ (s;;[x])\ (\#(s;;[x]){-}1)\ \textbf{fi} = x$	simplify the **if** and the index
$=$	$(s;;[x])\ (\#s) = x$	index the list
$=$	$x = x$	reflexive law
$=$	$\top$	

Data-Stack Theory

usage

var a, b: *stack*

a:= *empty*. b:= *push a* 2

consistent?

yes, we implemented it.

complete?

no, the binary expressions

pop empty = *empty*

top empty = 0

are unclassified. Proof: implement twice.

Theory as Firewall

user ensures that		implementer ensures that
only stack properties	theory	**all** stack properties
are relied upon		are provided

Simple Data-Stack Theory

axioms

~~*empty*: *stack*~~ *stack* ≠ *null*

push: *stack*→*X*→*stack*

~~*pop*: *stack*→*stack*~~

~~*top*: *stack*→*X*~~

~~*push s x* ≠ *empty*~~

~~*push s x* = *push t y* ≡ *s*=*t* ∧ *x*=*y*~~

~~*empty*, *push stack X*: *stack*~~

~~*empty*, *push B X*: *B* ⇒ *stack*: *B*~~

~~*P empty* ∧ ∀*s*: *stack*· ∀*x*: *X*· *Ps* ⇒ *P*(*push s x*) ≡ ∀*s*: *stack*· *Ps*~~

pop (*push s x*) = *s*

top (*push s x*) = *x*

Data-Queue Theory

$emptyq \colon queue$

$join\ q\ x \colon queue$

$join\ q\ x + emptyq$

$join\ q\ x = join\ r\ y\ \ =\ \ q{=}r \wedge x{=}y$

$q{+}emptyq\ \Rightarrow\ leave\ q \colon queue$

$q{+}emptyq\ \Rightarrow\ front\ q \colon X$

$emptyq, join\ B\ X \colon B\ \Rightarrow\ queue \colon B$

$leave\ (join\ emptyq\ x) = emptyq$

$q{+}emptyq\ \Rightarrow\ leave\ (join\ q\ x) = join\ (leave\ q)\ x)$

$front\ (join\ emptyq\ x) = x$

$q{+}emptyq\ \Rightarrow\ front\ (join\ q\ x) = front\ \ q$

Strong Data-Tree Theory

$emptree: tree$

$graft: tree \rightarrow X \rightarrow tree \rightarrow tree$

$emptree, graft\ B\ X\ B: B \;\Rightarrow\; tree: B$

$graft\ t\ x\ u \neq emptree$

$graft\ t\ x\ u = graft\ v\ y\ w \;=\; t{=}v \wedge x{=}y \wedge u{=}w$

$left\ (graft\ t\ x\ u) = t$

$root\ (graft\ t\ x\ u) = x$

$right\ (graft\ t\ x\ u) = u$

Weak Data-Tree Theory

$tree + null$

$graft\ t\ x\ u\colon tree$

$left\ (graft\ t\ x\ u) = t$

$root\ (graft\ t\ x\ u) = x$

$right\ (graft\ t\ x\ u) = u$

Data-Tree Implementation

$tree \;=\; emptree,\, graft\ tree\ int\ tree$

$emptree \;=\; [nil]$

$graft \;=\; \langle t\colon tree \to \langle x\colon int \to \langle u\colon tree \to [t;\, x;\, u]\rangle\rangle\rangle$

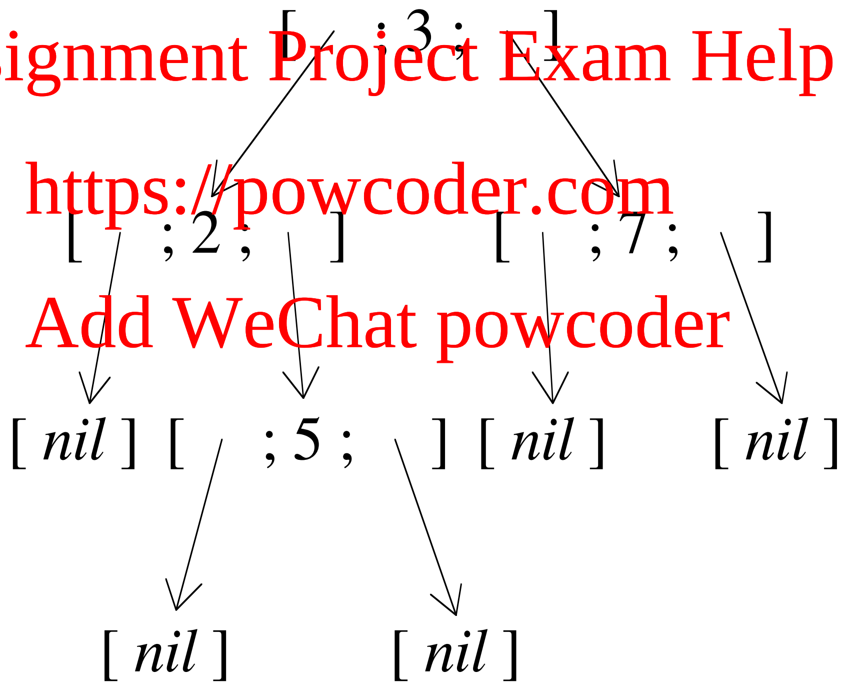
$left \;=\; \langle t\colon tree \to t\ 0\rangle$

$right \;=\; \langle t\colon tree \to t\ 2\rangle$

$root \;=\; \langle t\colon tree \to t\ 1\rangle$

Data-Tree Implementation

[[[*nil*]; 2; [[*nil*]; 5; [*nil*]]]; 3; [[*nil*]; 7; [*nil*]]]



Data-Tree Implementation

$$tree \;=\; emptree,\ graft\ tree\ int\ tree$$

$$emptree \;=\; 0$$

$$graft \;=\; \langle t\text{: } tree \to \langle x\text{: } int \to \langle u\text{: } tree \to \text{“left”}{\to}t \mid \text{“root”}{\to}x \mid \text{“right”}{\to}u\rangle\rangle\rangle$$

$$left \;=\; \langle t\text{: } tree \to t\ \text{“left”}\rangle$$

$$right \;=\; \langle t\text{: } tree \to t\ \text{“right”}\rangle$$

$$root \;=\; \langle t\text{: } tree \to t\ \text{“root”}\rangle$$

Data-Tree Implementation

$$
\begin{array}{lll}
\text{“left”} \to & (\text{“left”} \to 0 & \\
& \mid \text{“root”} \to 2 & \\
& \mid \text{“right”} \to (\text{“left”} \to 0 & \\
& \qquad\qquad \mid \text{“root”} \to 5 & \\
& \qquad\qquad \mid \text{“right”} \to 0\ )\ ) & \\
\mid \text{“root”} \to 3 & & \\
\mid \text{“right”} \to & (\text{“left”} \to 0 & \\
& \mid \text{“root”} \to 7 & \\
& \mid \text{“right”} \to 0\ ) &
\end{array}
$$

Theory Design

data theory

$s := push\ s\ x$

program theory

$push\ x$

user's variables, implementer's variables

Program-Stack Theory

syntax

push	a procedure with parameter of type X
pop	a program
top	expression of type X

axioms

$top'=x \;\Leftarrow\; push\ x$

$ok \;\Leftarrow\; push\ x.\ pop$

$$\begin{aligned} & ok \\ \Leftarrow\quad & push\ x.\ pop \\ =\quad & push\ x.\ ok.\ pop \\ \Leftarrow\quad & push\ x.\ push\ y.\ pop.\ pop \end{aligned}$$

Program-Stack Theory

syntax

push	a procedure with parameter of type X
pop	a program
top	expression of type X

axioms

$top'=x \;\Leftarrow\; push\ x$

$ok \;\Leftarrow\; push\ x.\ pop$

$top'=x$

$\Leftarrow \quad push\ x.\ ok$

$\Leftarrow \quad push\ x.\ push\ y.\ push\ z.\ pop.\ pop$

Program-Stack Implementation

var s: $[*X]$ implementer's variable

$push = \langle x{:}\ X \rightarrow s{:=}\ s;;[x] \rangle$

$pop = s{:=}\ s\ [0;..\#s-1]$

$top = s\ (\#s-1)$

Proof (first axiom):

	$(\ top'{=}x \Leftarrow push\ x)$	definitions of *push* and *top*
=	$(\ s'(\#s'-1){=}x \Leftarrow s{:=}\ s;;[x]\ )$	rewrite assignment with one variable
=	$(\ s'(\#s'-1){=}x \Leftarrow s' = s;;[x]\ )$	List Theory
=	$\top$	

consistent? yes, implemented.

complete? no, we can prove very little if we start with *pop*

Fancy Program-Stack Theory

$top'=x \land \neg isempty' \quad \Leftarrow \quad push\ x$

$ok \quad \Leftarrow \quad push\ x.\ \ pop$

$isempty' \quad \Leftarrow \quad mkempty$

Weak Program-Stack Theory

$top'=x \quad \Leftarrow \quad push\ x$

$top'=top \quad \Leftarrow \quad balance$

$balance \quad \Leftarrow \quad ok$

$balance \quad \Leftarrow \quad push\ x.\ balance.\ pop$

$count' = 0 \quad \Leftarrow \quad start$

$count' = count+1 \quad \Leftarrow \quad push\ x$

$count' = count+1 \quad \Leftarrow \quad pop$

Program-Queue Theory

$$isemptyq' \quad \Leftarrow \quad mkemptyq$$

$$isemptyq \;\Rightarrow\; front'=x \wedge \neg isemptyq' \quad \Leftarrow \quad join\ x$$

$$\neg isemptyq \;\Rightarrow\; front'=front \wedge \neg isemptyq' \quad \Leftarrow \quad join\ x$$

$$isemptyq \Rightarrow (join\ x.\ \ leave \quad = \quad mkemptyq)$$

$$\neg isemptyq \Rightarrow (join\ x.\ \ leave \quad = \quad leave.\ \ join\ x)$$

Program-Tree Theory

Variable *node* tells the value of the item where you are.

$node := 3$

Variable *aim* tells what direction you are facing.

$aim := up$ $\qquad aim := left$ $\qquad aim := right$

Program *go* moves you to the next node in the direction you are facing,

and turns you facing back the way you came.

Auxiliary specification *work* says do anything, but

do not *go* from this node (your location at the start of *work*)

in this direction (the value of variable *aim* at the start of *work*).

End where you started, facing the way you were facing at the start.

Program-Tree Theory

$(aim{=}up) = (aim'{\neq}up) \quad \Leftarrow \quad go$

$node'{=}node \land aim'{=}aim \quad \Leftarrow \quad go.\ work.\ go$

$work \quad \Leftarrow \quad ok$

$work \quad \Leftarrow \quad node{:=}\ x$

$work \quad \Leftarrow \quad a{=}aim{\neq}b \land (aim{:=}\ b.\ go.\ work.\ go.\ aim{:=}\ a)$

$work \quad \Leftarrow \quad work.\ work$

Data Transformation

user's variables u

implementer's variables v

new implementer's variables w

data transformer D relates v and w such that $\forall w\cdot \exists v\cdot D$

specification S is transformed to $\forall v\cdot D \Rightarrow \exists v'\cdot D' \wedge S$

$$\begin{tikzcd}
v \arrow[r, "S"] \arrow[d, leftrightarrow, "D"'] & v' \arrow[d, leftrightarrow, "D'"] \\
w \arrow[r, "\forall v\cdot D \Rightarrow \exists v'\cdot D' \wedge S"] & w'
\end{tikzcd}$$

Data Transformation

example

user's variable u: *bin*

implementer's variable v: *nat*

operations

$$\begin{aligned} \mathit{zero} &= v:= 0 \\ \mathit{increase} &= v:= v+1 \\ \mathit{inquire} &= u:= \mathit{even}\ v \end{aligned}$$

new implementer's variable w: *bin*

data transformer $w = \mathit{even}\ v$

Data Transformation

$$\forall v \cdot D \;\Rightarrow\; \exists v' \cdot D' \;\wedge\; zero$$

$$= \quad \forall v \cdot w = even\ v \;\Rightarrow\; \exists v' \cdot w' = even\ v' \;\wedge\; (v := 0)$$

$$= \quad \forall v \cdot w = even\ v \;\Rightarrow\; \exists v' \cdot w' = even\ v' \;\wedge\; u'=u \;\wedge\; v'=0 \qquad \text{1-pt}$$

$$= \quad \forall v \cdot w = even\ v \;\Rightarrow\; w' = even\ 0 \;\wedge\; u'=u \qquad \text{change variable}$$

$$= \quad \forall r: even\ nat \cdot w=r \;\Rightarrow\; w'=\top \;\wedge\; u'=u \qquad \text{1-pt}$$

$$= \quad w'=\top \;\wedge\; u'=u$$

$$= \quad w := \top$$

Data Transformation

$$
\begin{array}{lll}
 & \forall v\cdot D \;\Rightarrow\; \exists v'\cdot D' \;\wedge\; \mathit{increase} & \\
= & \forall v\cdot w = \mathit{even}\ v \;\Rightarrow\; \exists v'\cdot w' = \mathit{even}\ v' \;\wedge\; (v{:=}\ v{+}1) & \\
= & \forall v\cdot w = \mathit{even}\ v \;\Rightarrow\; \exists v'\cdot w' = \mathit{even}\ v' \;\wedge\; u'{=}u \;\wedge\; v'{=}v{+}1 & \text{1-pt} \\
= & \forall v\cdot w = \mathit{even}\ v \;\Rightarrow\; w' = \mathit{even}\ (v{+}1) \;\wedge\; u'{=}u & \text{change var} \\
= & \forall r{:}\ \mathit{even}\ \mathit{nat}\cdot w{=}r \;\Rightarrow\; w' = \neg r \;\wedge\; u'{=}u & \text{1-pt} \\
= & w' = \neg w \;\wedge\; u'{=}u & \\
= & w{:=}\ \neg w &
\end{array}
$$

Data Transformation

$$\begin{array}{lll}
 & \forall v\cdot\ D\ \Rightarrow\ \exists v'\cdot\ D'\ \land\ inquire & \\
= & \forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \land\ (u{:=}\ even\ v) & \\
= & \forall v\cdot\ w = even\ v\ \Rightarrow\ \exists v'\cdot\ w' = even\ v'\ \land u' = even\ v \land v'{=}v & \text{1-pt} \\
= & \forall v\cdot\ w = even\ v\ \Rightarrow\ w' = even\ v\ \land\ u' = even\ v & \text{change var} \\
= & \forall r\text{: } even\ nat\cdot\ w{=}r\ \Rightarrow\ w'{=}r\ \land\ u'{=}r & \text{1-pt} \\
= & w'{=}w\ \land\ u'{=}w & \\
= & u{:=}\ w &
\end{array}$$

Data Transformation

example

user's variable u: *bin*

implementer's variable v: *bin*

operations

$$set \;=\; v{:=}\top$$

$$flip \;=\; v{:=}\neg v$$

$$ask \;=\; u{:=}v$$

new implementer's variable w: *nat*

data transformer $v = even\ w$

Data Transformation

$$\begin{aligned}
& \forall v\cdot\ D\ \Rightarrow\ \exists v'\cdot\ D'\ \ \land\ \ set \\
=\quad & \forall v\cdot\ v = even\ w\ \Rightarrow\ \exists v'\cdot\ v' = even\ w'\ \ \land\ \ (v{:=}\ \top) \\
=\quad & even\ w'\ \ \land\ \ u'{=}u \\
\Leftarrow\quad & w{:=}\ 0
\end{aligned}$$

Data Transformation

$$\forall v\cdot\ D\ \Rightarrow\ \exists v'\cdot\ D'\ \land\ \mathit{flip}$$

$$=\quad \forall v\cdot\ v = \mathit{even}\ w\ \Rightarrow\ \exists v'\cdot\ v' = \mathit{even}\ w'\ \land\ (v{:=}\ \neg v)$$

$$=\quad \mathit{even}\ w' \ne \mathit{even}\ w\ \land\ u'{=}u$$

$$\Leftarrow\quad w{:=}\ w{+}1$$

Data Transformation

$$
\begin{array}{ll}
 & \forall v\cdot D \;\Rightarrow\; \exists v'\cdot D' \;\land\; ask \\
= & \forall v\cdot v = even\ w \;\Rightarrow\; \exists v'\cdot v' = even\ w' \;\land\; (u{:=}\,v) \\
= & even\ w' = even\ w = u' \\
\Leftarrow & u{:=}\ even\ w
\end{array}
$$

Security Switch

A security switch has three binary user's variables a , b , and c . The users assign values to a and b as input to the switch. The switch's output is assigned to c . The output changes when both inputs have changed. More precisely, the output changes when both inputs differ from what they were the previous time the output changed. The idea is that one user might flip their input indicating a desire for the output to change, but the output does not change until the other user flips their input indicating agreement that the output should change. If the first user changes back before the second user changes, the output does not change.

binary implementer's variables

A records the state of input a at last output change

B records the state of input b at last output change

Security Switch

A security switch has three binary user's variables a , b , and c . The users assign values to a and b as input to the switch. The switch's output is assigned to c . The output changes when both inputs have changed. More precisely, the output changes when both inputs differ from what they were the previous time the output changed. The idea is that one user might flip their input indicating a desire for the output to change, but the output does not change until the other user flips their input indicating agreement that the output should change. If the first user changes back before the second user changes, the output does not change.

operations

$a{:=}\ \neg a.$ **if** $a{\ne}A \wedge b{\ne}B$ **then** $c{:=}\ \neg c.\ A{:=}\ a.\ B{:=}\ b$ **else** ok **fi**

$b{:=}\ \neg b.$ **if** $a{\ne}A \wedge b{\ne}B$ **then** $c{:=}\ \neg c.\ A{:=}\ a.\ B{:=}\ b$ **else** ok **fi**

Security Switch

replace old implementer's variables A and B with nothing!

data transformer

$A=B=c$

proof

$\exists A, B\cdot\ A=B=c$ generalization, using c for both A and B

$\Leftarrow\quad \top$

operations

$a{:=}\ \neg a.$ **if** $a{\ne}A \wedge b{\ne}B$ **then** $c{:=}\ \neg c.\ A{:=}\ a.\ B{:=}\ b$ **else** ok **fi**

$b{:=}\ \neg b.$ **if** $a{\ne}A \wedge b{\ne}B$ **then** $c{:=}\ \neg c.\ A{:=}\ a.\ B{:=}\ b$ **else** ok **fi**

Security Switch

$\forall A, B\cdot\ A=B=c \Rightarrow \exists A', B'\cdot\ A'=B'=c' \wedge$ **if** $a \ne A \wedge b \ne B$ **then** $c:= \neg c.\ \ A:= a.\ \ B:= b$

else ok **fi**

expand assignments, dependent compositions, and ok

$=$ $\forall A, B\cdot\ A=B=c \Rightarrow \exists A', B'\cdot\ A'=B'=c' \wedge$ **if** $a \ne A \wedge b \ne B$

then $a'=a \wedge b'=b \wedge c'=\neg c \wedge A'=a \wedge B'=b$

else $a'=a \wedge b'=b \wedge c'=c \wedge A'=A \wedge B'=B$ **fi**

use one-point law for A and B, and for A' and B'

$=$ **if** $a \ne c \wedge b \ne c$ **then** $a'=a \wedge b'=b \wedge c'=\neg c \wedge c'=a \wedge c'=b$ use context

else $a'=a \wedge b'=b \wedge c'=c \wedge c'=c \wedge c'=c$ **fi**

$=$ **if** $a \ne c \wedge b \ne c$ **then** $a'=a \wedge b'=b \wedge c'=\neg c \wedge c'=\neg c \wedge c'=\neg c$

else $a'=a \wedge b'=b \wedge c'=c \wedge c'=c \wedge c'=c$ **fi**

$=$ **if** $a \ne c \wedge b \ne c$ **then** $c:= \neg c$ **else** ok **fi**

$=$ $c:= (a \ne c \wedge b \ne c) \ne c$

Limited Queue

user's variables: c: *bin* and x: X

old implementer's variables: Q: $[n*X]$ and p: *nat*

operations

$mkemptyq = p:= 0$

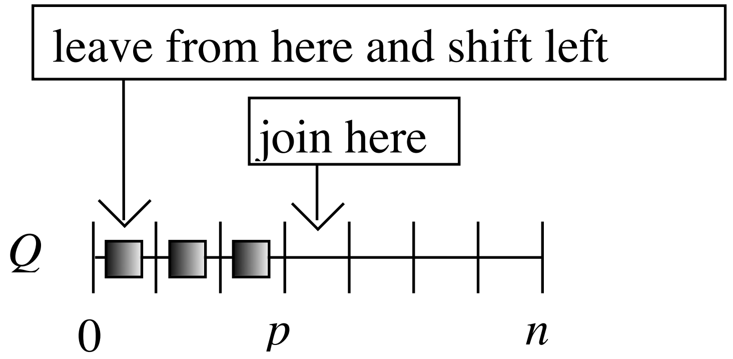
$isemptyq = c:= p=0$

$isfullq = c:= p=n$

$join = Qp:= x.\ \ p:= p+1$

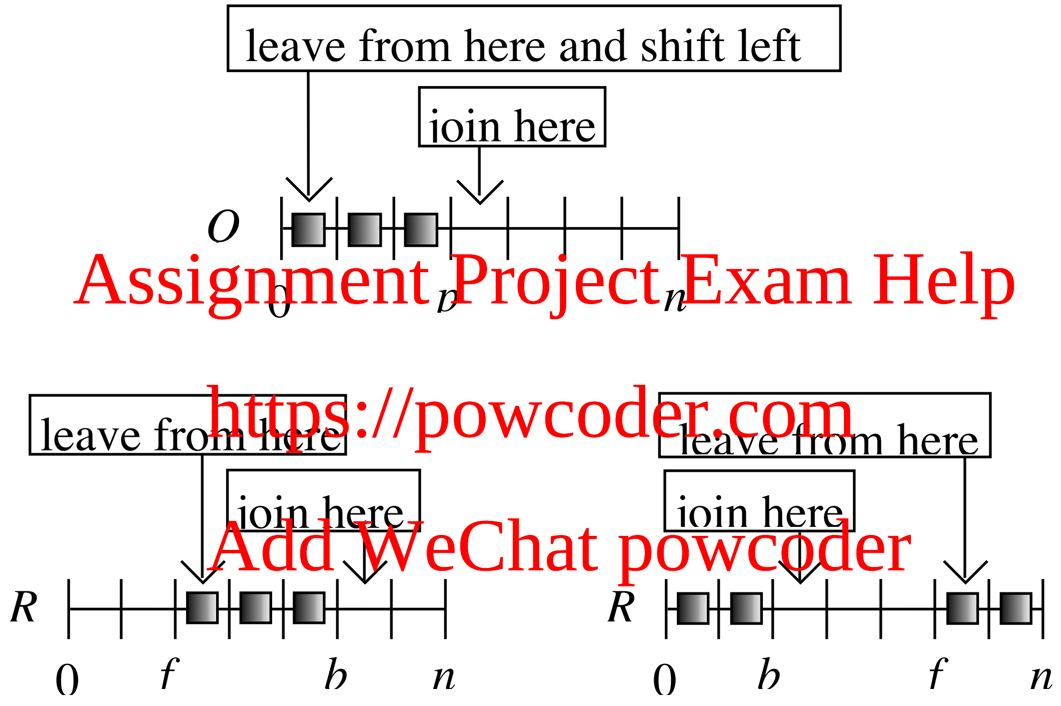
$leave =$ **for** $i:= 1;..p$ **do** $Q(i-1):= Qi$ **od**. $p:= p-1$

$front = x:= Q0$



Limited Queue

new implementer's variables: R: $[n{*}X]$ and f, b: $0,..n$



data transformer D :

$$0 \le p = b{-}f < n \;\wedge\; Q[0;..p] = R[f;..b]$$

$$\vee \quad 0 < p = n{-}f{+}b \le n \;\wedge\; Q[0;..p] = R[(f;..n);\ (0;..b)]$$

Limited Queue

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge mkemptyq$$

$$= \quad \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge (p:= 0)$$

$$= \quad \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge p'=0 \wedge Q'=Q \wedge c'=c \wedge x'=x$$

$$= \quad f'=b' \wedge c'=c \wedge x'=x$$

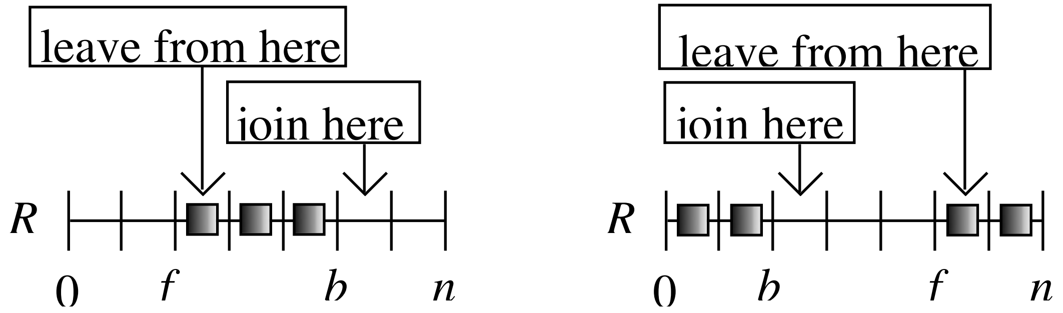
$$\Leftarrow \quad f:= 0.\ \ b:= 0$$

Limited Queue

$$\begin{array}{lll}
 & & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land isemptyq \\
= & & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land (c{:=}\ p{=}0) \\
= & & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land c'{=}(p{=}0) \land p'{=}p \land Q'{=}Q \land x'{=}x \\
= & & \quad f{<}b \;\land\; f'{<}b' \;\land\; b{-}f = b'{-}f' \\
 & \land & \quad R[f;..b] = R'[f';..b'] \;\land\; x'{=}x \;\land\; \neg c' \\
 & \lor & \quad f{<}b \;\land\; f'{>}b' \;\land\; b{-}f = n{+}b'{-}f' \\
 & \land & \quad R[f;..b] = R'[(f';..n);\ (0;..b')] \;\land\; x'{=}x \;\land\; \neg c' \\
 & \lor & \quad f{>}b \;\land\; f'{<}b' \;\land\; n{+}b{-}f = b'{-}f' \\
 & \land & \quad R[(f;..n);\ (0;..b)] = R'[f';..b'] \;\land\; x'{=}x \;\land\; \neg c' \\
 & \lor & \quad f{>}b \;\land\; f'{>}b' \;\land\; b{-}f = b'{-}f' \\
 & \land & \quad R[(f;..n);\ (0;..b)]{=}R'[(f';..n);\ (0;..b')] \;\land\; x'{=}x \;\land\; \neg c'
\end{array}$$

$f{=}b$ is missing! unimplementable!

Limited Queue



data transformer D :

$$m \;\wedge\; 0 \le p = b{-}f < n \;\wedge\; Q[0;..p] = R[f;..b]$$

$$\vee\;\; \neg m \;\wedge\; 0 < p = n{-}f{+}b \le n \;\wedge\; Q[0;..p] = R[(f;..n);\,(0;..b)]$$

Limited Queue

$$\forall Q, p\cdot\ D \Rightarrow \exists Q', p'\cdot\ D' \wedge mkemptyq$$

$$= \quad m' \ \wedge\ f'=b' \ \wedge\ c'=c \ \wedge\ x'=x$$

$$\Leftarrow \quad m:=\top.\ f:=0.\ b:=0$$

Limited Queue

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge isemptyq$$

$$\begin{aligned}
= \quad & \quad m \wedge f<b \wedge m' \wedge f'<b' \wedge b-f = b'-f \\
& \wedge \quad R[f;..b] = R'[f';..b'] \wedge x'=x \wedge \neg c' \\
\vee \quad & \quad m \wedge f<b \wedge \neg m' \wedge f'>b' \wedge b-f = n+b'-f' \\
& \wedge \quad R[f;..b] = R'[(f';..n); (0;..b')] \wedge x'=x \wedge \neg c' \\
\vee \quad & \quad \neg m \wedge f>b \wedge m' \wedge f'<b' \wedge n+b-f = b'-f' \\
& \wedge \quad R[(f;..n); (0;..b)] = R'[f';..b'] \wedge x'=x \wedge \neg c' \\
\vee \quad & \quad \neg m \wedge f>b \wedge \neg m' \wedge f'>b' \wedge b-f = b'-f' \\
& \wedge \quad R[(f;..n); (0;..b)] = R'[(f';..n); (0;..b')] \wedge x'=x \wedge \neg c' \\
\vee \quad & m \wedge f=b \wedge m' \wedge f'=b' \wedge x'=x \wedge c' \\
\vee \quad & \quad \neg m \wedge f=b \wedge \neg m' \wedge f'=b' \\
& \wedge \quad R[(f;..n); (0;..b)]=R'[(f';..n); (0;..b')] \wedge x'=x \wedge \neg c' \\
\Leftarrow \quad & c' = (m \wedge f=b) \wedge f'=f \wedge b'=b \wedge R'=R \wedge x'=x \\
= \quad & c:= m \wedge f=b
\end{aligned}$$

Limited Queue

$\forall Q, p\cdot\ D \Rightarrow \exists Q', p'\cdot\ D' \wedge \mathit{isfullq}$

$\Leftarrow \quad c{:=}\ \neg m \wedge f{=}b$

$\forall Q, p\cdot\ D \Rightarrow \exists Q', p'\cdot\ D' \wedge \mathit{join}$

$\Leftarrow \quad R\ b{:=}\ x.\ \ \textbf{if}\ b{+}1{=}n\ \textbf{then}\ b{:=}\ 0.\ \ m{:=}\ \bot\ \textbf{else}\ b{:=}\ b{+}1\ \textbf{fi}$

$\forall Q, p\cdot\ D \Rightarrow \exists Q', p'\cdot\ D' \wedge \mathit{leave}$

$\Leftarrow \quad \textbf{if}\ f{+}1{=}n\ \textbf{then}\ f{:=}\ 0.\ \ m{:=}\ \top\ \textbf{else}\ f{:=}\ f{+}1\ \textbf{fi}$

$\forall Q, p\cdot\ D \Rightarrow \exists Q', p'\cdot\ D' \wedge \mathit{front}$

$\Leftarrow \quad x{:=}\ R\,f$

Data Transformation

No need to replace the same number of variables

can replace fewer or more

No need to replace entire space of implementer's variables

do part only

Can do parts separately

data transformers can be conjoined

People really do data transformations by

defining the new data space and reprogramming each operation ✗

They should

state the transformer and transform the operations ✓