

## Data-Stack Theory

syntax: *stack empty push pop top*

axioms:

*empty: stack*

*push: stack  $\rightarrow$  X  $\rightarrow$  stack*

*pop: stack  $\rightarrow$  stack*

*top: stack  $\rightarrow$  X*

*empty, push stack X: stack*

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

or  $P \text{ empty} \wedge \forall s: \text{stack}. \forall x: X. Ps \Rightarrow P(\text{push } s \ x)$

$= \forall s: \text{stack}. Ps$

*push s x  $\neq$  empty*

*push s x = push t y  $= s=t \wedge x=y$*

*pop (push s x) = s*

*top (push s x) = x*

## Data-Stack Implementation

$$stack = [*int]$$

$$empty = [nil]$$

$$push = \lambda s: stack. \lambda x: int. s+[x]$$

$$pop = \lambda s: stack. \mathbf{if} s=empty \mathbf{then} empty \mathbf{else} s [0;..\#s-1]$$

$$top = \lambda s: stack. \mathbf{if} s=empty \mathbf{then} 0 \mathbf{else} s (\#s-1)$$

Proof (last axiom):

$$\begin{aligned}
 & top (push s x) = x \\
 = & top ((\lambda s: stack. \lambda x: int. s+[x]) s x) = x \\
 = & top (s+[x]) = x \\
 = & (\lambda s: stack. \mathbf{if} s=empty \mathbf{then} 0 \mathbf{else} s (\#s-1)) (s+[x]) = x \\
 = & (\mathbf{if} s+[x]=[nil] \mathbf{then} 0 \mathbf{else} (s+[x]) (\#(s+[x])-1)) = x \\
 = & (s+[x]) (\#s) = x \\
 = & x = x
 \end{aligned}$$

usage:

**var**  $a, b$ : *stack*

$a := \text{empty}$ .  $b := \text{push } a \ 2$

consistent? yes, we implemented it.

complete? no, the boolean expressions

$\text{pop } \text{empty} = \text{empty}$

$\text{top } \text{empty} = 0$

are unclassified. Proof: implement twice.

user ensures that <b>only</b>	theory	implementer ensures that
stack properties are	as	<b>all</b> stack properties
relied upon	firewall	are provided

## Simple Data-Stack Theory

$pop: stack \rightarrow stack$  is too strong; it implies  $pop\ empty: stack$

$top: stack \rightarrow X$  is too strong; it implies  $top\ empty: X$

induction is unnecessary

$empty$  is unnecessary

$stack \neq null$

$push\ s\ x: stack$

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

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

## Data-Queue Theory

*emptyq: queue*

*join: queue → X → queue*    or    *join q x: queue*

*join q x ≠ emptyq*

*join q x = join r y = q=r ∧ x=y*

*leave: queue → queue*    or    *leave q: queue*    or

*q ≠ emptyq ⇒ leave q: queue*

*front: queue → X*    or    *front q: X*    or

*q ≠ emptyq ⇒ front q: X*

*emptyq, join B X: B ⇒ queue: B*

*leave (join emptyq x) = emptyq*

*q ≠ emptyq ⇒ leave (join q x) = join (leave q) x*

*front (join emptyq x) = x*

*q ≠ emptyq ⇒ 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  $\equiv$  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  $\neq$  null*

*graft t x u: tree*

*left (graft t x u) = t*

*root (graft t x u) = x*

*right (graft t x u) = u*

## Data-Tree Implementation

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

$emptree = [nil]$

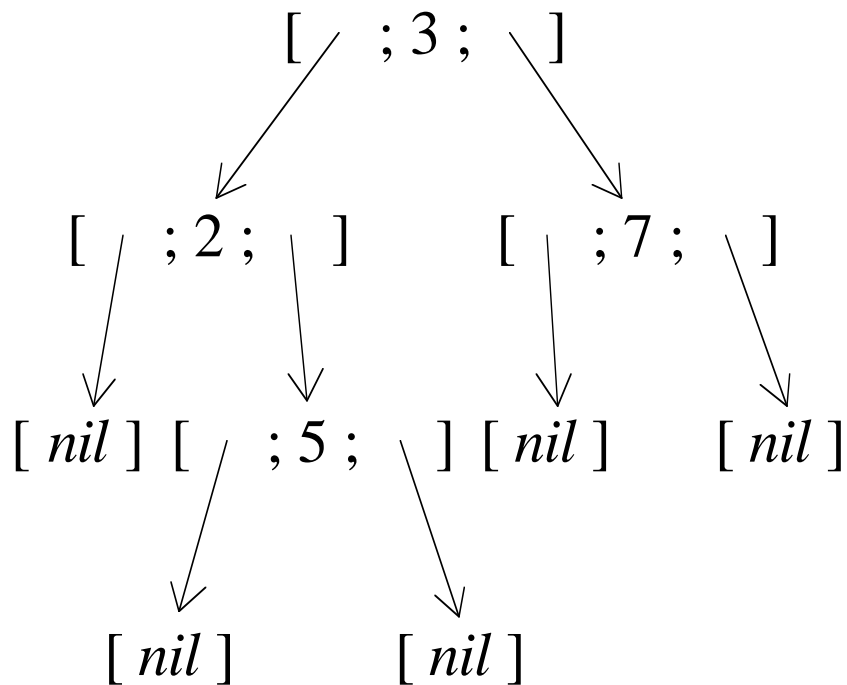
$graft = \lambda t: tree. \lambda x: int. \lambda u: tree. [t; x; u]$

$left = \lambda t: tree. t\ 0$

$right = \lambda t: tree. t\ 2$

$root = \lambda t: tree. t\ 1$

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



*tree = emptytree, graft tree int tree*

*emptytree = 0*

*graft = λt: tree. λx: int. λu: tree.*

*"left" → t | "root" → x | "right" → u*

*left = λt: tree. t "left"*

*right = λt: tree. t "right"*

*root = λt: tree. t "root"*

*"left" → ("left" → 0  
 | "root" → 2  
 | "right" → ("left" → 0  
 | "root" → 5  
 | "right" → 0 ) )*

*| "root" → 3*

*| "right" → ("left" → 0  
 | "root" → 7  
 | "right" → 0 )*

## Program-Stack Theory

syntax:  $push$  (a procedure with parameter of type  $X$ )

$pop$  (a program)

$top$  (of type  $X$ )

axioms:

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

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

$ok$

$$\Leftarrow push\ x.\ pop$$

$$= push\ x.\ ok.\ pop$$

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

$top'=x$

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

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

## Program-Stack Implementation

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

$push = \lambda x: X. s := s + [x]$

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

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

Proof (first axiom):

$$\begin{aligned}
 & ( top' = x \iff push\ x ) && \text{replace } push \text{ and } top \\
 = & ( s'(\#s' - 1) = x \iff s := s + [x] ) && \text{List Theory} \\
 = & \top
 \end{aligned}$$

consistent? yes, implemented.

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

## Fancy Program-Stack Theory

$$top' = x \wedge \neg isempty' \Leftarrow push\ x$$

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

$$isempty' \Leftarrow mkempty$$

## Weak Program-Stack Theory

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

$$top' = top \Leftarrow balance$$

$$balance \Leftarrow ok$$

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

This allows

$$count' = 0 \Leftarrow start$$

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

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

## Program-Queue Theory

$$i\text{empty}q' \Leftarrow mk\text{empty}q$$

$$i\text{empty}q \Rightarrow \text{front}'=x \wedge \neg i\text{empty}q' \Leftarrow \text{join } x$$

$$\neg i\text{empty}q \Rightarrow \text{front}'=\text{front} \wedge \neg i\text{empty}q' \Leftarrow \text{join } x$$

$$i\text{empty}q \Rightarrow (\text{join } x. \text{leave} = mk\text{empty}q)$$

$$\neg i\text{empty}q \Rightarrow (\text{join } x. \text{leave} = \text{leave}. \text{join } x)$$

## Program-Tree Theory

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

Variable *aim* tells what direction you are facing.

Program *go* moves you to the next node in the direction you are facing, and turns you facing back the way you came.

$$(aim=up) = (aim' \neq up) \Leftarrow go$$

$$node'=node \wedge aim'=aim \Leftarrow go. work. go$$

$$work \Leftarrow ok$$

$$work \Leftarrow node:=x$$

$$work \Leftarrow a=aim \neq b \wedge (aim:=b. go. work. go. aim:=a)$$

$$work \Leftarrow work. work$$

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.

## Data Transformation

user's variables  $u$

implementer's variables  $v$

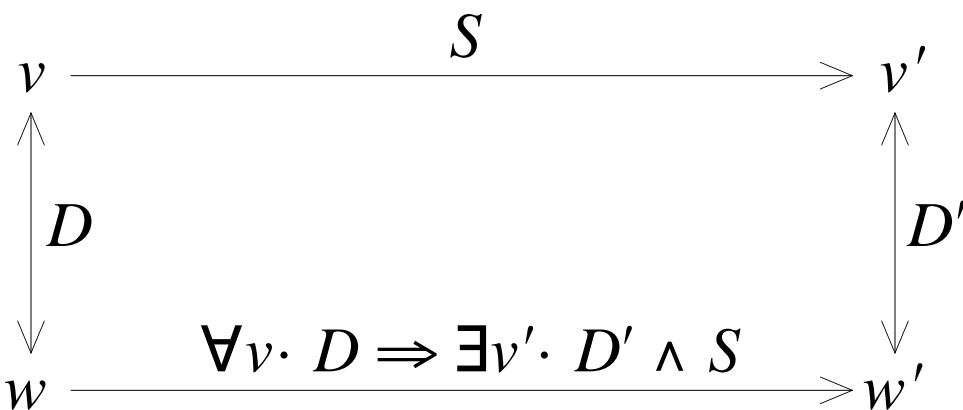
new implementer's variables  $w$

data transformer  $D$  relates  $v$  and  $w$  such that

$$\forall w. \exists v. D$$

specification  $S$  is transformed to

$$\forall v. D \Rightarrow \exists v'. D' \wedge S$$



Example:

user's variable  $u: \text{bool}$

implementer's variable  $v: \text{nat}$

operations

$$\text{zero} = v := 0$$
$$\text{increase} = v := v + 1$$
$$\text{inquire} = u := \text{even } v$$

new implementer's variable  $w: \text{bool}$

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

*zero* becomes

$$\begin{aligned}
& \forall v. w = \text{even } v \Rightarrow \exists v'. w' = \text{even } v' \wedge (v := 0) \\
= & \forall v. w = \text{even } v \Rightarrow \exists v'. w' = \text{even } v' \wedge u' = u \wedge v' = 0 && \text{1-pt} \\
= & \forall v. w = \text{even } v \Rightarrow w' = \text{even } 0 \wedge u' = u && \text{change variable} \\
= & \forall r: \text{even nat}. w = r \Rightarrow w' = \top \wedge u' = u && \text{1-pt} \\
= & w' = \top \wedge u' = u \\
= & w := \top
\end{aligned}$$

*increase* becomes

$$\begin{aligned}
& \forall v. w = \text{even } v \Rightarrow \exists v'. w' = \text{even } v' \wedge (v := v+1) \\
= & \forall v. w = \text{even } v \Rightarrow \exists v'. w' = \text{even } v' \wedge u' = u \wedge v' = v+1 && \text{1-pt} \\
= & \forall v. w = \text{even } v \Rightarrow w' = \text{even } (v+1) \wedge u' = u && \text{change var} \\
= & \forall r: \text{even nat}. w = r \Rightarrow w' = \neg r \wedge u' = u && \text{1-pt} \\
= & w' = \neg w \wedge u' = u \\
= & w := \neg w
\end{aligned}$$

*inquire* becomes

$$\begin{aligned}
& \forall v. w = \text{even } v \Rightarrow \exists v'. w' = \text{even } v' \wedge (u := \text{even } v) \\
= & \forall v. w = \text{even } v \Rightarrow \exists v'. w' = \text{even } v' \wedge u' = \text{even } v \wedge v' = v \\
= & \forall v. w = \text{even } v \Rightarrow w' = \text{even } v \wedge u' = \text{even } v && \text{change var} \\
= & \forall r: \text{even nat}. w = r \Rightarrow w' = r \wedge u' = r && \text{1-pt} \\
= & w' = w \wedge u' = w \\
= & u := w
\end{aligned}$$

Example:

user's variable  $u: \mathit{bool}$

implementer's variable  $v: \mathit{bool}$

operations

$$\mathit{set} = v := \top$$
$$\mathit{flip} = v := \neg v$$
$$\mathit{ask} = u := v$$

new implementer's variable  $w: \mathit{nat}$

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

*set* becomes

$$\begin{aligned}
 & \forall v. v = \text{even } w \Rightarrow \exists v'. v' = \text{even } w' \wedge (v := \top) \\
 = & \text{even } w' \wedge u' = u \\
 \Leftarrow & w := 0
 \end{aligned}$$

*flip* becomes

$$\begin{aligned}
 & \forall v. v = \text{even } w \Rightarrow \exists v'. v' = \text{even } w' \wedge (v := \neg v) \\
 = & \text{even } w' \neq \text{even } w \wedge u' = u \\
 \Leftarrow & w := w + 1
 \end{aligned}$$

*ask* becomes

$$\begin{aligned}
 & \forall v. v = \text{even } w \Rightarrow \exists v'. v' = \text{even } w' \wedge (u := v) \\
 = & \text{even } w' = \text{even } w = u' \\
 \Leftarrow & u := \text{even } w
 \end{aligned}$$

## Limited Queue

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

$mkemptyq = p := 0$

$isemptyq = p = 0$

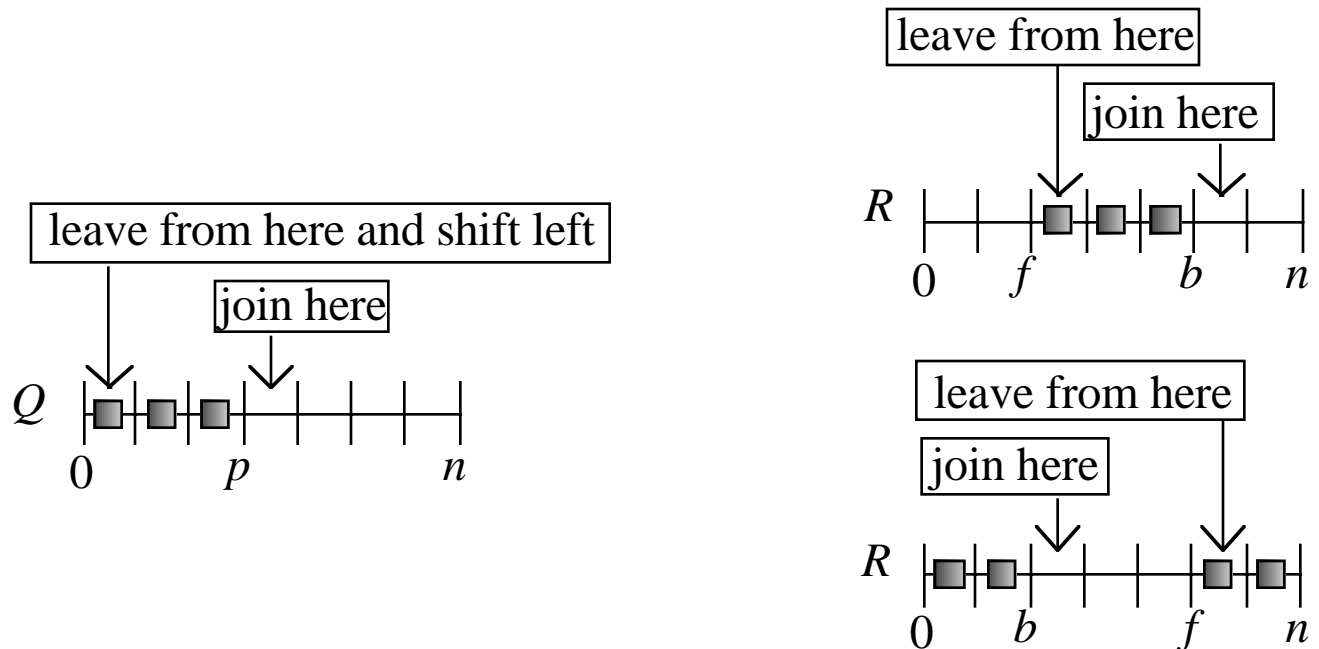
$isfullq = p = n$

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

$leave = \mathbf{for}\ i := 1; .. p\ \mathbf{do}\ Q(i-1) := Qi. p := p - 1$

$front = Q0$

New implementer's variables:  $R: [n^*X]$  and  $f, b: 0, ..n$



Data transformer  $D$  :

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

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

$$\begin{aligned}
& \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge mkemptyq \\
= & \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge p'=0 \wedge Q'=Q \\
= & f=b' \\
\Leftarrow & f:=0. b:=0
\end{aligned}$$

$$\begin{aligned}
& \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge (u:=isemptyq) \\
= & \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge u'=(p=0) \wedge p'=p \wedge Q'=Q \\
= & \quad f < b \wedge f < b' \wedge b-f = b'-f \\
& \quad \wedge R[f;..b] = R'[f';..b'] \wedge \neg u' \\
\vee & \quad f < b \wedge f > b' \wedge b-f = n+b'-f \\
& \quad \wedge R[f;..b] = R'[(f';..n); (0;..b')] \wedge \neg u' \\
\vee & \quad f > b \wedge f < b' \wedge n+b-f = b'-f \\
& \quad \wedge R[(f;..n); (0;..b)] = R'[f';..b'] \wedge \neg u' \\
\vee & \quad f > b \wedge f > b' \wedge b-f = b'-f \\
& \quad \wedge R[(f;..n); (0;..b)] = R'[(f';..n); (0;..b')] \wedge \neg u'
\end{aligned}$$

$\neg u'$  in every case.  $f=b$  is missing. unimplementable.

New transformer  $D$  :

$$\begin{aligned}
& m \wedge 0 \leq p = b-f < n \wedge Q[0;..p] = R[f;..b] \\
\vee & \neg m \wedge 0 < p = n-f+b \leq n \wedge Q[0;..p] = R[(f;..n); (0;..b)]
\end{aligned}$$

$$\begin{aligned}
& \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge mkemptyq \\
= & \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge p'=0 \wedge Q'=Q \\
= & m' \wedge f=b' \\
\Leftarrow & m:=\top. f:=0. b:=0
\end{aligned}$$

$$\begin{aligned}
& \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge (u:=isemptyq) \\
= & \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge u'=(p=0) \wedge p'=p \wedge Q'=Q \\
= & \quad m \wedge f < b \wedge m' \wedge f < b' \wedge b-f = b'-f \\
& \quad \wedge R[f;..b] = R'[f';..b'] \wedge \neg u' \\
\vee & \quad m \wedge f < b \wedge \neg m' \wedge f > b' \wedge b-f = n+b'-f \\
& \quad \wedge R[f;..b] = R'[(f';..n); (0;..b')] \wedge \neg u' \\
\vee & \quad \neg m \wedge f > b \wedge m' \wedge f < b' \wedge n+b-f = b'-f \\
& \quad \wedge R[(f;..n); (0;..b)] = R'[f';..b'] \wedge \neg u' \\
\vee & \quad \neg m \wedge f > b \wedge \neg m' \wedge f > b' \wedge b-f = b'-f \\
& \quad \wedge R[(f;..n); (0;..b)] = R'[(f';..n); (0;..b')] \wedge \neg u' \\
\vee & \quad m \wedge f=b \wedge m' \wedge f=b' \wedge u' \\
\vee & \quad \neg m \wedge f=b \wedge \neg m' \wedge f=b' \\
& \quad \wedge R[(f;..n); (0;..b)] = R'[(f';..n); (0;..b')] \wedge \neg u' \\
\Leftarrow & u' = (m \wedge f=b) \wedge f'=f \wedge b'=b \wedge R'=R \\
= & u:= m \wedge f=b
\end{aligned}$$

$$\begin{aligned}
& \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge (u := \text{isfull}q) \\
= & \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge u' = (p = n) \wedge p' = p \wedge Q' = Q \\
\Leftarrow & u := \neg m \wedge f = b
\end{aligned}$$

$$\begin{aligned}
& \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge \text{join } x \\
= & \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge Q' = Q[0;..p]^+ x^+ Q[p+1;..n] \\
& \quad \wedge p' = p+1 \\
\Leftarrow & Rb := x. \text{ if } b+1 = n \text{ then } (b := 0. m := \perp) \text{ else } b := b+1
\end{aligned}$$

$$\begin{aligned}
& \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge \text{leave} \\
= & \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge Q' = Q[(1;..p); (p;..n)] \wedge p' = p-1 \\
\Leftarrow & \text{ if } f+1 = n \text{ then } (f := 0. m := \top) \text{ else } f := f+1
\end{aligned}$$

$$\begin{aligned}
& \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge (u := \text{front}) \\
= & \forall Q, p. D \Rightarrow \exists Q', p'. D' \wedge u' = Q0 \wedge p' = p \wedge Q' = Q \\
\Leftarrow & u := Rf
\end{aligned}$$

## **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
- reprogramming each operation

They should

- + state the transformer
- + transform the operations