Theory Design

data theory

program theory

Theory Design

data theory

push s x

program theory

Theory Design

data theory

$s := push\ s\ x$

program theory

Theory Design

data theory

$s := push\ s\ x$

program theory

$push\ x$

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

Program-Stack Theory

syntax

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

axioms

$top' = x \Leftarrow push\ x$

$ok \Leftarrow push\ x.\ pop$

Program-Stack Theory

syntax

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

axioms

$top' = x \Leftarrow push\ x$

$ok \Leftarrow push\ x.\ pop$

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

Program-Stack Theory

syntax

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

axioms

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

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

$$\Leftarrow \quad \begin{array}{l} ok \\ push\ x.\ pop \end{array}$$

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

Program-Stack Theory

syntax

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

axioms

$top' = x \Leftarrow push\ x$

$ok \Leftarrow push\ x.\ pop$

ok

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

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

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

Program-Stack Theory

syntax

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

axioms

$top' = x \Leftarrow push\ x$

$ok \Leftarrow push\ x.\ pop$

Program-Stack Theory

syntax

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

axioms

$top' = x \Leftarrow push\ x$

$ok \Leftarrow push\ x.\ pop$

$top' = x$

$\Leftarrow \quad push\ x$

Program-Stack Theory

syntax

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

axioms

$top' = x \Leftarrow push\ x$

$ok \Leftarrow push\ x.\ pop$

$top' = x$

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

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

new *s*: [**X*] implementer's variable

Program-Stack Implementation

new s: [*X] implementer's variable

$push \;=\; \langle x\colon X\cdot \;\; s := s;;[x]\rangle$

Program-Stack Implementation

new s: [*X] implementer's variable

$push \;=\; \langle x\colon X\cdot \;\; s := s;;[x]\rangle$

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

Program-Stack Implementation

new s: [*X] implementer's variable

$push \;=\; \langle x\colon X\cdot \;\; s := s;;[x]\rangle$

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

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

Program-Stack Implementation

new s: [*X] implementer's variable

$push \;=\; \langle x\colon X\cdot \; s := s;;[x]\rangle$

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

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

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
=	$\top$	

Program-Stack Implementation

new s: [*X] implementer's variable

$push \;=\; \langle x\colon X\cdot \; s := s;;[x] \rangle$

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

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

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.

Program-Stack Implementation

new s: [*X] implementer's variable

$push \;=\; \langle x\colon X\cdot \; s := s;;[x]\rangle$

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

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

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

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

$ok \Leftarrow push\ x.\ pop$

$isempty' \Leftarrow mkempty$

Fancy Program-Stack Theory

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

$ok \Leftarrow push\ x.\ pop$

$isempty' \Leftarrow mkempty$

Fancy Program-Stack Theory

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

$ok \Leftarrow push\ x.\ pop$

$isempty' \Leftarrow mkempty$

Fancy Program-Stack Theory

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

$ok \Leftarrow push\ x.\ pop$

$isempty' \Leftarrow mkempty$

Weak Program-Stack Theory

$top' = x \Leftarrow push\ x$

$top' = top \Leftarrow balance$

$balance \Leftarrow ok$

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

Weak Program-Stack Theory

$top' = x \Leftarrow push\ x$

$top' = top \Leftarrow balance$

$balance \Leftarrow ok$

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

$count' = 0 \Leftarrow start$

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

$count' = count + 1 \Leftarrow pop$

Program-Queue Theory

$$isemptyq' \Leftarrow mkemptyq$$

$$isemptyq \Rightarrow front'{=}x \wedge \neg isemptyq' \Leftarrow join\ x$$

$$\neg isemptyq \Rightarrow front'{=}front \wedge \neg isemptyq' \Leftarrow join\ x$$

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

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

Program-Queue Theory

$isemptyq' \Leftarrow mkemptyq$ 

$isemptyq \Rightarrow front'{=}x \wedge \neg isemptyq' \Leftarrow join\ x$

$\neg isemptyq \Rightarrow front'{=}front \wedge \neg isemptyq' \Leftarrow join\ x$

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

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

Program-Queue Theory

$$isemptyq' \Leftarrow mkemptyq$$

$$isemptyq \Rightarrow front'{=}x \wedge \neg isemptyq' \Leftarrow join\ x$$

$$\neg isemptyq \Rightarrow front'{=}front \wedge \neg isemptyq' \Leftarrow join\ x$$

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

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

Program-Queue Theory

$isemptyq' \Leftarrow mkemptyq$

$isemptyq \Rightarrow front'{=}x \wedge \neg isemptyq' \Leftarrow join\ x$

$\neg isemptyq \Rightarrow front'{=}front \wedge \neg isemptyq' \Leftarrow join\ x$ ←

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

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

Program-Queue Theory

$isemptyq' \Leftarrow mkemptyq$

$isemptyq \Rightarrow front'{=}x \wedge \neg isemptyq' \Leftarrow join\ x$

$\neg isemptyq \Rightarrow front'{=}front \wedge \neg isemptyq' \Leftarrow join\ x$

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

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

Program-Queue Theory

$isemptyq' \Leftarrow mkemptyq$

$isemptyq \Rightarrow front'{=}x \wedge \neg isemptyq' \Leftarrow join\ x$

$\neg isemptyq \Rightarrow front'{=}front \wedge \neg isemptyq' \Leftarrow join\ x$

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

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

Program-Tree Theory

Program-Tree Theory

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

Program-Tree Theory

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

node:= 3

Program-Tree Theory

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

node:= 3

Variable *aim* tells what direction you are facing.

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* *aim*:= *left* *aim*:= *right*

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* *aim*:= *left* *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.

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* *aim*:= *left* *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

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

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

$work \Leftarrow ok$

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

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

$work \Leftarrow work.\ work$

Program-Tree Theory

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

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

$work \Leftarrow ok$

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

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

$work \Leftarrow work.\ work$

Program-Tree Theory

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

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

$work \Leftarrow ok$

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

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

$work \Leftarrow work.\ work$