Data-Stack Theory

syntax: \textit{stack} \emptyset \textit{push} \textit{pop} \textit{top}

axioms:

\textit{empty}: \textit{stack}
\textit{push}: \textit{stack} \rightarrow X \rightarrow \textit{stack}
\textit{pop}: \textit{stack} \rightarrow \textit{stack}
\textit{top}: \textit{stack} \rightarrow X

\emptyset, \textit{push stack} X: \textit{stack}
\emptyset, \textit{push} B X: B \Rightarrow \textit{stack}: B

or \hspace{1cm} P \emptyset \land \forall s: \textit{stack} \cdot \forall x: X \cdot Ps \Rightarrow P(\textit{push} s x)
\hspace{1cm} = \forall s: \textit{stack} \cdot Ps

\textit{push} s x \neq \emptyset

\textit{push} s x = \textit{push} t y \hspace{1cm} = \hspace{1cm} s=t \land x=y

\textit{pop} (\textit{push} s x) = s
\textit{top} (\textit{push} s x) = x
Data-Stack Implementation

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

Proof (last axiom):

\[ top (push s x) = x \]
\[ = top ((\lambda s: stack \cdot \lambda x: int \cdot s[x]) s x) = x \]
\[ = top (s[x]) = x \]
\[ = (\lambda s: stack \cdot \text{if } s=\text{empty} \text{ then } 0 \text{ else } s (#s-1)) (s[x]) = x \]
\[ = (\text{if } s+[x]=[nil] \text{ then } 0 \text{ else } (s+[x]) (#(s+[x])-1)) = x \]
\[ = (s+[x]) (#s) = x \]
\[ = x = x \]
usage:

```plaintext
var a, b: stack

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

consistent? yes, we implemented it.

complete? no, the boolean expressions

- `pop empty = empty`
- `top empty = 0`

are unclassified. Proof: implement twice.

user ensures that only stack properties are relied upon as firewall

implementer ensures that theory as all stack properties 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

$stack \neq \text{null}$

*push s x*: $stack$

$pop (push s x) = s$

$top (push s x) = x$
Data-Queue Theory

emptyq: queue

join: queue \rightarrow X \rightarrow queue \text{ or } join q x: queue

join q x \neq emptyq

join q x = join r y \Rightarrow q = r \land x = y

leave: queue \rightarrow queue \text{ or } leave q: queue \text{ or }

\quad q \neq emptyq \Rightarrow leave q: queue

front: queue \rightarrow X \text{ or } front q: X \text{ or }

\quad q \neq emptyq \Rightarrow front q: X

emptyq, join B X: B \Rightarrow queue: B

leave (join emptyq x) = emptyq

q \neq emptyq \Rightarrow leave (join q x) = join (leave q) x

front (join emptyq x) = x

q \neq emptyq \Rightarrow front (join q x) = front q
Strong Data-Tree Theory

$\textit{emptree}: \textit{tree}$

$\textit{graft}: \textit{tree} \rightarrow X \rightarrow \textit{tree} \rightarrow \textit{tree}$

$\textit{emptree}, \textit{graft} B X B: B \Rightarrow \textit{tree}: B$

$\textit{graft} t x u \neq \textit{emptree}$

$\textit{graft} t x u = \textit{graft} v y w = t=v \land x=y \land u=w$

$\text{left} (\textit{graft} t x u) = t$

$\text{root} (\textit{graft} t x u) = x$

$\text{right} (\textit{graft} t x u) = u$

Weak Data-Tree Theory

$\text{tree} \neq \textit{null}$

$\textit{graft} t x u: \textit{tree}$

$\text{left} (\textit{graft} t x u) = t$

$\text{root} (\textit{graft} t x u) = x$

$\text{right} (\textit{graft} t x u) = u$
**Data-Tree Implementation**

\[
tree = \text{emptree}, \text{graft} \; tree \; \text{int} \; tree
\]

\[
\text{emptree} = [\text{nil}]
\]

\[
\text{graft} = \lambda t: \text{tree} \cdot \lambda x: \text{int} \cdot \lambda u: \text{tree} \cdot [t; x; u]
\]

\[
\text{left} = \lambda t: \text{tree} \cdot t \; 0
\]

\[
\text{right} = \lambda t: \text{tree} \cdot t \; 2
\]

\[
\text{root} = \lambda t: \text{tree} \cdot t \; 1
\]

\[
[[[\text{nil}]; 2; [[\text{nil}]; 5; [\text{nil}]]]; 3; [[\text{nil}]; 7; [\text{nil}]]]
\]

```
     [     ; 3 ;     ]
    /      \      /      \      /      \      /      \    
   [     ; 2 ;     ]    [     ; 7 ;     ]
  /        \    /        \    /        \    /        \  
 [ nil ] [ ; 5 ; ] [ nil ] [ nil ]
 /  \  /     \  /     \  /     \  /     \  
[ nil ] [ nil ] [ nil ]
```

7.6
\[\text{tree} = \text{emptree}, \text{graft tree int tree} \]

\[\text{emptree} = 0\]

\[\text{graft} = \lambda t: \text{tree} \cdot \lambda x: \text{int} \cdot \lambda u: \text{tree}.
\]

\[\text{"left"} \rightarrow t \mid \text{"root"} \rightarrow x \mid \text{"right"} \rightarrow u\]

\[\text{left} = \lambda t: \text{tree} \cdot t \text{ "left"} \]

\[\text{right} = \lambda t: \text{tree} \cdot t \text{ "right"} \]

\[\text{root} = \lambda t: \text{tree} \cdot t \text{ "root"} \]

\[\text{"left"} \rightarrow (\text{"left"} \rightarrow 0 \mid \text{"root"} \rightarrow 2 \mid \text{"right"} \rightarrow (\text{"left"} \rightarrow 0 \mid \text{"root"} \rightarrow 5 \mid \text{"right"} \rightarrow 0) ) \mid \text{"root"} \rightarrow 3 \mid \text{"right"} \rightarrow (\text{"left"} \rightarrow 0 \mid \text{"root"} \rightarrow 7 \mid \text{"right"} \rightarrow 0) \]
Program-Stack Theory

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

pop (a program)  

top (of type $X$)  

axioms:  

top' = x  \iff push x  
ok  \iff push x. pop  

7.8

ok  

\iff push x. pop  

\iff push x. ok. pop  

\iff push x. push y. pop. pop  

top' = x  

\iff push x. ok  

\iff push x. push y. push z. pop. pop
Program-Stack Implementation

\[ \textbf{var} \ s: \ [*X] \quad \text{implementer's variable} \]

\[ \textit{push} = \lambda x: X \cdot s := s + [x] \]

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

\[ \textit{top} = s (#s-1) \]

Proof (first axiom):

\[ ( \textit{top}'=x \iff \textit{push} \ x ) \quad \text{replace \textit{push} and \textit{top}} \]

\[ = ( s'(\#s'-1)=x \iff s := s + [x] ) \quad \text{List Theory} \]

\[ = \top \]

consistent? yes, implemented.

complete? no, we can prove very little if we start with \textit{pop}
Fancy Program-Stack Theory

\[ top' = x \land \neg isempty' \iff push \ x \]
\[ ok \iff push \ x \cdot pop \]
\[ isempty' \iff mkempty \]

Weak Program-Stack Theory

\[ top' = x \iff push \ x \]
\[ top' = top \iff balance \]
\[ balance \iff ok \]
\[ balance \iff push \ x \cdot balance \cdot pop \]

This allows

\[ count' = 0 \iff start \]
\[ count' = count+1 \iff push \ x \]
\[ count' = count+1 \iff pop \]
Program-Queue Theory

\[
\begin{align*}
\text{isemptyq'} & \iff \text{mkemptyq} \\
\text{isemptyq} & \Rightarrow \text{front'} = x \land \neg\text{isemptyq'} \iff \text{join } x \\
\neg\text{isemptyq} & \Rightarrow \text{front'} = \text{front} \land \neg\text{isemptyq'} \iff \text{join } x \\
\text{isemptyq} & \Rightarrow (\text{join } x. \ \text{leave} \ = \ \text{mkemptyq}) \\
\neg\text{isemptyq} & \Rightarrow (\text{join } x. \ \text{leave} \ = \ \text{leave}. \ \text{join } x)
\end{align*}
\]
**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.

\[
(\text{aim} = \text{up}) = (\text{aim}' \neq \text{up}) \iff \text{go}
\]

\[
\text{node}' = \text{node} \land \text{aim}' = \text{aim} \iff \text{go. work. go}
\]

\[
\text{work} \iff \text{ok}
\]

\[
\text{work} \iff \text{node} := x
\]

\[
\text{work} \iff a = \text{aim} \neq b \land (\text{aim} := b. \text{go. work. go. aim} := a)
\]

\[
\text{work} \iff \text{work. work}
\]

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

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

![Diagram](attachment:image.png)
Example:

user's variable $u: bool$

implementer's variable $v: nat$

operations

$\text{zero} \quad = \quad v:= 0$

$\text{increase} \quad = \quad v:= v+1$

$\text{inquire} \quad = \quad u:= \text{even } v$

new implementer's variable $w: bool$

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

\[ \forall v \cdot w = \text{even } v \Rightarrow \exists v' \cdot w' = \text{even } v' \land (v := 0) \]

\[ = \forall v \cdot w = \text{even } v \Rightarrow \exists v' \cdot w' = \text{even } v' \land u' = u \land v' = 0 \quad 1\text{-pt} \]

\[ = \forall v \cdot w = \text{even } v \Rightarrow w' = \text{even } 0 \land u' = u \quad \text{change variable} \]

\[ = \forall r : \text{even nat} \cdot w = r \Rightarrow w' = T \land u' = u \quad 1\text{-pt} \]

\[ = w' = T \land u' = u \]

\[ = w := T \]

increase becomes

\[ \forall v \cdot w = \text{even } v \Rightarrow \exists v' \cdot w' = \text{even } v' \land (v := v+1) \]

\[ = \forall v \cdot w = \text{even } v \Rightarrow \exists v' \cdot w' = \text{even } v' \land u' = u \land v' = v+1 \quad 1\text{-pt} \]

\[ = \forall v \cdot w = \text{even } v \Rightarrow w' = \text{even } (v+1) \land u' = u \quad \text{change var} \]

\[ = \forall r : \text{even nat} \cdot w = r \Rightarrow w' = \neg r \land u' = u \quad 1\text{-pt} \]

\[ = w' = \neg w \land u' = u \]

\[ = w := \neg w \]

inquire becomes

\[ \forall v \cdot w = \text{even } v \Rightarrow \exists v' \cdot w' = \text{even } v' \land (u := \text{even } v) \]

\[ = \forall v \cdot w = \text{even } v \Rightarrow \exists v' \cdot w' = \text{even } v' \land u' = \text{even } v \land v' = v \]

\[ = \forall v \cdot w = \text{even } v \Rightarrow w' = \text{even } v \land u' = \text{even } v \quad \text{change var} \]

\[ = \forall r : \text{even nat} \cdot w = r \Rightarrow w' = r \land u' = r \quad 1\text{-pt} \]

\[ = w' = w \land u' = w \]

\[ = u := w \]
Example:

user's variable \( u : bool \)

implementer's variable \( v : bool \)

operations

\[
\begin{align*}
set & = v := \top \\
flip & = v := \neg v \\
ask & = u := v
\end{align*}
\]

new implementer's variable \( w : nat \)

data transformer \( v = even \; w \)
set becomes

\[ \forall v \cdot v = \text{even } w \implies \exists v' \cdot v' = \text{even } w' \land (v := T) \]
\[ \iff \text{even } w' \land u' = u \]

flip becomes

\[ \forall v \cdot v = \text{even } w \implies \exists v' \cdot v' = \text{even } w' \land (v := \neg v) \]
\[ \iff \text{even } w' \oplus \text{even } w \land u' = u \]

ask becomes

\[ \forall v \cdot v = \text{even } w \implies \exists v' \cdot v' = \text{even } w' \land (u := v) \]
\[ \iff \text{even } w' = \text{even } w = u' \]
\[ \iff u := \text{even } w \]
Limited Queue

Old implementer's variables: $Q: [n*X]$ and $p: \text{nat}$

$\text{mkemptyq} = p := 0$

$\text{isemptyq} = p = 0$

$\text{isfullq} = p = n$

$\text{join x} = Qp := x. \ p := p+1$

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

$\text{front} = Q0$

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

![Diagram]

Data transformer $D$:

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

\[ \lor \quad 0 < p = n-f+b \leq n \land Q[0;..p] = R[(f;..n); (0;..b)] \]
∀ Q, p · D ⇒ ∃ Q', p' · D' ∧ mkemptyq

= ∀ Q, p · D ⇒ ∃ Q', p' · D' ∧ p' = 0 ∧ Q' = Q

= f = b'

⇐ f := 0. b := 0

∀ Q, p · D ⇒ ∃ Q', p' · D' ∧ (u := isemptyq)

= ∀ Q, p · D ⇒ ∃ Q', p' · D' ∧ u' = (p = 0) ∧ p' = p ∧ Q' = Q

= f < b ∧ f' < b' ∧ b = f + b' − f' ∧ R[f;..b] = R'[f';..b'] ∧ ¬ u'

∨ f < b ∧ f' > b' ∧ b = n + b' − f' ∧ R[f;..b] = R'[f';..n]; (0;..b')] ∧ ¬ u'

∨ f > b ∧ f' < b' ∧ n + b = b' − f' ∧ R[(f;..n); (0;..b)] = R'[f';..b'] ∧ ¬ u'

∨ f > b ∧ f' > b' ∧ b = b' − f' ∧ R[(f;..n); (0;..b)] = R'[f';..n]; (0;..b')] ∧ ¬ u'

¬ u' in every case. f = b is missing. unimplementable.

New transformer D:

m ∧ 0 ≤ p = b − f < n ∧ Q[0;..p] = R[f;..b]

∨ ¬ m ∧ 0 < p = n − f + b ≤ n ∧ Q[0;..p] = R[(f;..n); (0;..b)]
∀Q, p · D ⇒ ∃Q', p' · D' ∧ mkemptyq

= ∀Q, p · D ⇒ ∃Q', p' · D' ∧ p'=0 ∧ Q'=Q

= m' ∧ f'=b'

⇐ m:= T . f:= 0 . b:= 0

∀Q, p · D ⇒ ∃Q', p' · D' ∧ (u:= isemptyq)

= ∀Q, p · D ⇒ ∃Q', p' · D' ∧ u'=(p=0) ∧ p'=p ∧ Q'=Q

= m ∧ f<b ∧ m' ∧ f'<b' ∧ b−f = b'−f

∧ R[f;..b] = R'[f';..b'] ∧ ¬u'

∨ m ∧ f<b ∧ ¬m' ∧ f'>b' ∧ b−f = n+b'−f'

∧ R[f;..b] = R'[(f';..n); (0;..b')] ∧ ¬u'

∨ ¬m ∧ f>b ∧ m' ∧ f'<b' ∧ n+b−f = b'−f'

∧ R[(f;..n); (0;..b)] = R'[(f';..b'); (0;..b')] ∧ ¬u'

∨ ¬m ∧ f>b ∧ ¬m' ∧ f'>b' ∧ b−f = b'−f'

∧ R[(f;..n); (0;..b)] = R'[(f';..b'); (0;..b')] ∧ ¬u'

∨ m ∧ f=b ∧ m' ∧ f'=b' ∧ u'

∨ ¬m ∧ f=b ∧ ¬m' ∧ f'=b'

∧ R[(f;..n); (0;..b)]=R'[(f';..n); (0;..b')] ∧ ¬u'

⇐ u' = (m ∧ f=b) ∧ f'=f ∧ b'=b ∧ R'=R

= u:= m ∧ f=b
\[ \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \land (u := \text{isfullq}) \]

\[ = \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \land u'(p = n) \land p' = p \land Q' = Q \]

\[ \Leftarrow u := \neg m \land f = b \]

\[ \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \land \text{join} \ x \]

\[ = \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \land Q' = Q[0..p] + x + Q[p+1..n] \]

\[ \land p' = p + 1 \]

\[ \Leftarrow Rb := x. \ \text{if} \ b + 1 = n \ \text{then} \ (b := 0. \ m := \bot) \ \text{else} \ b := b + 1 \]

\[ \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \land \text{leave} \]

\[ = \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \land Q' = Q[(1..p); (p;..n)] \land p' = p - 1 \]

\[ \Leftarrow \text{if} \ f + 1 = n \ \text{then} \ (f := 0. \ m := \top) \ \text{else} \ f := f + 1 \]

\[ \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \land (u := \text{front}) \]

\[ = \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \land u' = Q0 \land p' = p \land Q' = Q \]

\[ \Leftarrow u := Rf \]
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