Abstract Data Types and Software Validation

Guttag, Horowitz, and Musser

Algebraic Specifications of Abstract Data Types

- Specification
  - syntax
  - semantics
- Representation
- Programs (Implementation)

type Stack[elementtype: Type]

syntax

NEWSTACK → Stack,
PUSH(Stack, elementtype) → Stack,
POP(Stack) → Stack,
TOP(Stack) → elementtype U {UNDEFINED},
ISNEW(Stack) → Boolean,
REPLACE(Stack, elementtype) → Stack

syntax

NEWSTACK → Stack,
PUSH(Stack, elementtype) → Stack,
POP(Stack) → Stack,
TOP(Stack) → elementtype U {UNDEFINED},
ISNEW(Stack) → Boolean,
REPLACE(Stack, elementtype) → Stack

semantics

declare stk:Stack, elm:elementtype;
POP(NEWSTACK) = NEWSTACK,
POP(PUSH(Stk,elm)) = stk,
TOP(NEWSTACK) = UNDEFINED,
TOP(PUSH(stk,elm)) = elm,
ISNEW(NEWSTACK) = TRUE,
ISNEW(PUSH(stk,elm)) = FALSE,
REPLACE(stk,elm) = PUSH(POP(stk),elm)

Every operation is one of

- Constructor
- Modifier
- Selector

Constructor

- Can construct any instance of the type being defined using only constructors
- No proper subset of the constructors can do this
- For a given type the constructors are not necessarily unique

Consider

type Sequence[elementtype: Type]

syntax

NEWSEQ → Sequence,
APPENDR (Sequence, elementtype) → Sequence,
APPENDL (Sequence, elementtype) → Sequence,
FIRST(Sequence) → elementtype U {UNDEFINED},
LAST(Sequence) → elementtype U {UNDEFINED},
ISEMPTY(Sequence) → Boolean

Canonical Form

- The expression defining an instance of a type is said to be in *canonical form* if it only uses the constructor operators for that type
- Every instance of a type has a *unique* canonical form

type Array[domaintype:Type, rangetype:Type]

syntax

NEWARRAY → Array,
ASSIGN(Array,domaintype,rangetype) → Array,
ACCESS(Array,domaintype) → rangetype,

semantics

declare arr:Array, dval,dval1:domaintype, rval:rangetype;
ACCESS(NEWARRAY,dval) = UNDEFINED,
ACCESS(ASSIGN(arr,dval,rval),dval1) =
IF dval = dval1
THEN rval
ELSE ACCESS(arr,dval1).

type Stack[elementtype: Type]

syntax

NEWSTACK → Stack,
PUSH(Stack, elementtype) → Stack,
POP(Stack) → Stack,
TOP(Stack) → elementtype U {UNDEFINED},
ISNEW(Stack) → Boolean,
REPLACE(Stack, elementtype) → Stack

representation STAK(Array[Integer,elementtype],Integer)
→ Stack[elementtype],

syntax NEWSTACK → Stack,
PUSH(Stack, elementtype) → Stack,
POP(Stack) → Stack,
TOP(Stack) → elementtype U {UNDEFINED},
ISNEW(Stack) → Boolean,
REPLACE(Stack, elementtype) → Stack,

programs declare arr:Array, t:Integer, elm:elementtype;
NEWSTACK = STAK(NEWARRAY,0),
PUSH(STAK(arr,t),elm) = STAK(ASSIGN(arr,t+1,elm),t+1),
POP(STAK(arr,t)) =
IF t=0 THEN STAK(arr,0) ELSE STAK(arr,t-1),
TOP(STAK(arr,t)) = ACCESS(arr,t),
ISNEW(STAK(arr,t)) = (t=0)
REPLACE(STAK(arr,t),elm) =
IF t=0 THEN STAK(ASSIGN(arr,1,elm),1)
ELSE STAK(ASSIGN(arr,t,elm),t).

Demonstrating Correctness of the Implementation

- Formally this is an application of many-sorted algebras
- Correctness is demonstrated by showing the existence of a homomorphic mapping from the implementation algebra to the abstract algebra

Basic Proof Technique

The basic proof technique for verifying an implementation of a data type is to show that each of the axiomatic specifications for the data type (i.e., the rewrite rules) is satisfied when the programs are substituted for the operations that they implement in the axiom.

Assume: stk = STAK(arr,t)

Show:
TOP(PUSH(stk,elm)) = elm

type Soda_Machine

syntax

Init → Soda_Machine,
Pay(Soda_Machine) → Soda_Machine,
Select(Soda_Machine,Flavor) → Soda_Machine,
Dispense(Soda_Machine,Cup) → Soda_Machine,
Number_Of_Coins(Soda_Machine) → Integer,
Current_Selection(Soda_Machine) → Flavor,
Full(Soda_Machine,Cup) → Boolean,
Cup_Contents(Soda_Machine,Cup) → Flavor

syntax

Init → Soda_Machine,
Pay(Soda_Machine) → Soda_Machine,
Select(Soda_Machine,Flavor) → Soda_Machine,
Dispense(Soda_Machine,Cup) → Soda_Machine,
Number_Of_Coins(Soda_Machine) → Integer,
Current_Selection(Soda_Machine) → Flavor,
Full(Soda_Machine,Cup) → Boolean,
Cup_Contents(Soda_Machine,Cup) → Flavor

semantics

declare sm: Soda_Machine, c,cc:Cup, f:Flavor;
Number_Of_Coins(Init) = 0,
Number_Of_Coins(Pay(sm)) = Number_Of_Coins(sm) +1,
Number_Of_Coins(Select(sm,f)) = Number_Of_Coins(sm),
Number_Of_Coins(Dispense(sm,c)) =
if Number_Of_Coins(sm) >= 2 & ~Full(sm,c)
then Number_Of_Coins(sm) -2
else Number_Of_Coins(sm)
fi

syntax

Init → Soda_Machine,
Pay(Soda_Machine) → Soda_Machine,
Select(Soda_Machine,Flavor) → Soda_Machine,
Dispense(Soda_Machine,Cup) → Soda_Machine,
Number_Of_Coins(Soda_Machine) → Integer,
Current_Selection(Soda_Machine) → Flavor,
Full(Soda_Machine,Cup) → Boolean,
Cup_Contents(Soda_Machine,Cup) → Flavor

semantics

declare sm: Soda_Machine, c,cc:Cup, f:Flavor;
Full(Init,c) =False,
Full(Pay(sm),c) = Full(sm,c),
Full(Select(sm,f),c) = Full(sm,c),
Full(Dispense(sm,c),cc) =
if c=cc & Number_Of_Coins(sm) >= 2 & ~Full(sm,c)
then True
else Full(sm,cc)
fi

HW#7

You are to prove Axiom #6 of the algebraic specification for the symbol table given in the Guttag, et. al. paper. You may assume that symtab=SYMT(stk).

Symboltable Data Type

type Symboltable

syntax

INIT → Symboltable,
ENTERBLOCK(Symboltable) → Symboltable,
ADDID(Symboltable, Identifier, Attributelist) → Symboltable,
LEAVEBLOCK(Symboltable) → Symboltable,
ISINBLOCK(Symboltable, Identifier) → Boolean,
RETRIEVE(Symboltable, Identifier) →
Attributelist U {UNDEFINED}.

semantics
declare symtab: Symboltable, id, id1: Identifier, attrlist: Attributelist;
1) LEAVEBLOCK(INIT) == INIT,
2) LEAVEBLOCK(ENTERBLOCK(symtab)) == symtab,
3) LEAVEBLOCK(ADDID(symtab, id, attrlist))
== LEAVEBLOCK(symtab),
4) ISINBLOCK(INIT, id) == FALSE,
5) ISINBLOCK(ENTERBLOCK(symtab), id) == FALSE,
6) ISINBLOCK(ADDID(symtab, id, attrlist), idl)
== IF id = idl
THEN TRUE
ELSE ISINBLOCK(symtab, idl),
7) RETRIEVE(INIT, id) == UNDEFINED,
8) RETRIEVE(ENTERBLOCK(symtab), id)
== RETRIEVE(symtab, id),
9) RETRIEVE(ADDID(symtab, id, attrlist), idl)
== IF id = idl
THEN attrlist
ELSE RETRIEVE(symtab, idl).

representation
SYMT(Stack[Mapping[Identifier, Attributelist]]) → Symboltable.
programs
declare stk: Stack, id: Identifier, attrlist: Attributelist;
INIT = SYMT(PUSH(NEWSTACK, NEWMAP)),
ENTERBLOCK(SYMT(stk)) = SYMT(PUSH(stk, NEWMAP)),
ADDID(SYMT(stk), id, attdist)
= SYMT(REPLACE(stk, DEFMAP(TOP(stk), id, attrlist))),
LEAVEBLOCK(SYMT(stk))
= IF ISNEW(POP(stk))
THEN SYMT(REPLACE(stk, NEWMAP))
ELSE SYMT(POP(stk)),
ISINBLOCK(SYMT(stk), id) = ISDEFINED(TOP(stk), id),
RETRIEVE(SYMT(stk), id)
= IF ISNEW(stk)
THEN UNDEFINED
ELSE IF ISDEFINED(TOP(stk), id)
THEN EVMAP(TOP(stk), id)
ELSE RETRIEVE(SYMT(POP(stk)), id).

Mapping Data Type

type Mapping[domaintype: Type, rangetype: Type]

syntax

NEWMAP → Mapping,

DEFMAP(Mapping, domaintype, rangetype) → Mapping,

EVMAP(Mapping, domaintype) →

rangetype U {UNDEFINED},

ISDEFINED(Mapping, domaintype) → Boolean.

semantics
declare map: Mapping, dval, dval1 : domaintype,
rval: rangetype;
EVMAP(NEWMAP, dval) == UNDEFINED,
EVMAP(DEFMAP(map, dval, rval), dvall)
== IF dval = dvall THEN rval
ELSE EVMAP(map, dvall),
ISDEFINED(NEWMAP, dval1) == FALSE,
ISDEFINED(DEFMAP(map, dval, rval), dval1)
== IF dval = dval1 THEN TRUE
ELSE ISDEFINED(map, dval1).

Affirm Specifications

Typename
Local declarations
Interfaces
Ruleset

Ruleset

- Automatically applied rules
  - Axiom
  - Rulelemma
- Controlled rules
  - Define
  - Schema

Secure Terminal in Affirm

Uses types buffer and host_level, which must be defined

```
type buffer;
declare dummy: buffer;

interface sanitized_buffer: buffer;

end {buffer} ;
```



```
type host_level;
declare dummy, l: host_level;

interfaces
      high, low: host_level;
interface induction(l): Boolean;

axioms
      high = low == FALSE,
      low = high == FALSE;

schema induction(l) == cases(Prop(high), Prop(low));

end {host_level} ;
```



```
type secure_terminal;
needs types host_level, buffer;

declare dummy, st: secure_terminal;
declare l: host_level;
declare b: buffer;

interfaces
      connected(st), reviewed(st), accepted(st), sanitized(st),
      connect_ok(st, l): Boolean;
interfaces
      terminal_level(st), active_host(st): host_level;
interfaces
      terminal_buffer(st), send_data(st): buffer;
interfaces
      init, connect_to(st, l), disconnect(st), sanitize(st), review_data(st),
     change_level(st), receive_data(st, b): secure_terminal;
```


Define

- Defines not automatically applied

```
define connect_ok(st,l) == ~connected(st)
          and (sanitized(st) or terminal_level(st)=l);
```


Axioms for Connected Variable

```
axioms
      connected(init) == FALSE,
      connected(disconnect(st)) == FALSE,
      connected(sanitize(st)) == connected(st),
      connected(connect_to(st,l)) == if connect_ok(st,l)
                                     then TRUE
                                     else connected(st),
      connected(change_level(st)) == connected(st),
      connected(receive_data(st, b)) == connected(st);
```

Axioms for Reviewed Variable

```
axioms
    reviewed(init) == FALSE,
    reviewed(disconnect(st)) == reviewed(st),
    reviewed(sanitize(st)) ==
                    (connected(st) and reviewed(st)),
    reviewed(connect_to(st,l)) == if connect_ok(st,l)
                                  then false
                                  else  reviewed(st),
    reviewed(change_level(st)) == reviewed(st),
    reviewed(receive_data(st, b)) == reviewed(st);
```


Axioms for Accepted Variable

```
axioms
    accepted(init) == FALSE,
    accepted(disconnect(st)) == accepted(st),
    accepted(sanitize(st)) ==
                    (connected(st) and accepted(st)),
    accepted(connect_to(st,l)) == if connect_ok(st,l)
                                  then false
                                  else accepted(st),
    accepted(change_level(st)) == accepted(st),
    accepted(receive_data(st, b)) == accepted(st);
```


Axioms for Sanitized Variable

```
axioms
    sanitized(init) == TRUE,
    sanitized(disconnect(st)) == sanitized(st),
    sanitized(sanitize(st)) ==
                    (connected(st) imp sanitized(st)),
    sanitized(connect_to(st,l)) == if connect_ok(st,l)
                                   then false
                                   else sanitized(st),
    sanitized(change_level(st)) == sanitized(st),
    sanitized(receive_data(st, b)) == sanitized(st);
```


Axioms Regrouped for Init

```
axioms
    connected(init) == FALSE,
    reviewed(init) == FALSE,
    accepted(init) == FALSE,
    sanitized(init) == TRUE,
    terminal_buffer(init) == sanitized_buffer;
```


Axioms Regrouped for Disconnect

```
axioms
    connected(disconnect(st)) == FALSE,
    reviewed(disconnect(st)) == reviewed(st),
    accepted(disconnect(st)) == accepted(st),
    sanitized(disconnect(st)) == sanitized(st),
    terminal_level(disconnect(st)) == terminal_level(st),
    active_host(disconnect(st)) == active_host(st),
    terminal_buffer(disconnect(st)) ==
                                    terminal_buffer(st);
```


Schema for Induction Proof

```
schema secterInduction(st)
    == cases(Prop(init),
        all st, l (IH(st) imp Prop(connect_to(st,l))),
        all st (IH(st) imp Prop(disconnect(st))),
        all st (IH(st) imp Prop(sanitize(st))),
        all st (IH(st) imp Prop(review_data(st))),
        all st (IH(st) imp Prop(change_level(st))),
        all st, b (IH(st) imp Prop(receive_data(st,b))));
```