CSC 530 Lecture Notes Week 10

Algebraic Semantics

I. A grand vision.

A. Algebraic semantics intended to have a wide scope.

B. In is grandest form, it’s a unifying theoretical framework for:

Grand vision, cont'd

1. Programming languages, including formal semantics and compilers.

2. Software engineering, including object-oriented development, formal specification, formal testing, and verification.

3. Theoretical foundations of computability.

4. High-level computer architecture.

Grand vision, cont’d

C. Joseph Goguen is of sufficient intellect to pull it off.

II. How papers present the grand vision.

A. Paper 34 presents a broad, early overview.

B. Paper 35 discusses algebraic theory applied to object-oriented formal specification and verification.

Papers, cont'd

C. Paper 36 describes practical algebraic programming in OBJ.

D. Paper 37 focus on how an algebraic approach solves well-known problems in programming langs.

Papers, cont'd

E. Paper 38 outlines how an algebraic model of computation can be used to design hardware.

F. Paper 39 is the OBJ3 language reference manual.

Papers, cont'd

G. Paper 40 describes the Maude language, a popular successor to OBJ.

H. Paper 41 is an overview of the OBJ family of languages; lots of links.

III. A mind-altering experience.

A. An algebraic PL has the following unique properties:

1. It's fully declarative.

2. An algebraic program and its specification are identical.

3. An algebraic PL and its formal semantics are identical.

4. Operational execution, semantic evaluation, and formal verification use the same mechanism.

IV. A stack ADT as an initial example.

```
obj STACK is sort Stack .

    protecting NAT .

    op push : Stack Nat -> Stack .
    op pop : Stack -> Stack .
    op top : Stack -> Nat .
    op emptyStack : -> Stack .
    op emptyElem : -> Nat .

    var S : Stack .
    var E : Nat .

    eq pop(emptyStack) = emptyStack .
    eq pop(push(S, E)) = S .

    eq top(emptyStack) = emptyElem .
    eq top(push(S, E)) = E .

endo
```

V. Executing algebraically-defined ADTs.

A. Execution performed using *term rewriting*, a.k.a, *reduction*.

B. A program is a *term*, which is simply an application functions to arguments.

C. To perform reduction, equations used as *pattern matching rules*.

D. Consider the following program:

Executing, cont'd

```
in Stack

obj MAIN is

    *** "Protecting" imports.
    protecting STACK .
    protecting INT .

    *** "Op" declares functions.
    op main : -> Stack .

    *** A parameterlesss op is a
    *** single-assignment variable.
    op s1 : -> Stack .
    op s2 : -> Stack .
    op i : -> Int .
```

Executing, cont’d

```
  *** Equations declare what
  *** the program does.
  eq s1 = pop(push(push(push(
            emptyStack, 1), 2), 3)) .
  eq i = top(push(push(push(
           emptyStack, 1), 2), 3)) + 1 .
  eq s2 = push(push(s1, i), 5) .

  eq main = pop(s2) .

endo
```

Executing, cont'd

```
*** The following executes program main.
reduce main .

*** The result of execution is the stack
*** push(push(push(emptyStack,1),2),4).
```

VI. Additional examples.

A. An ML-like list ADT.

B. A set-like ADT.

C. A binary search tree.

D. A parameterized list.

E. A bubble sorter.

F. A simple PL, called "Fun", similar to SIL and Tennent 13.2.

VII. Algebraic program proofs.

A. Section 4 of paper 35 describes how term rewriting is used for proof.

B. Hence, proofs about programs use the same term rewriting technique as used for program execution.

C. Here's an example.

Proofs, cont'd

```
obj NAT is sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat [prec 1] .
  op _+_ : Nat Nat -> Nat [assoc comm prec 3] .
  vars M N : Nat .
  eq M + 0 = M .
  eq M + s N = s(M + N).
  op _*_ : Nat Nat -> Nat [prec 2] .
  eq M * 0 = 0 .
  eq M * s N = M * N + M .
endo
obj VARS is protecting NAT .
  ops m n : -> Nat .
endo

***> first show two lemmas, 0*n=0 and sm*n=m*n+n
***> base for first lemma
reduce 0 * 0 == 0 .
***> induction step for first lemma
obj HYP1 is using VARS .
  eq 0 * n = 0 .
endo
reduce 0 * s n == 0 .
*** thus we can assert
```

```
obj LEMMA1 is protecting NAT .
  vars N : Nat .
  eq 0 * N = 0 .
endo

***> base for second lemma
reduce in VARS + LEMMA1 : s n * 0 == n * 0 + 0 .
***> induction step for second lemma
obj HYP2 is using VARS .
  eq s m * n = m * n + n .
endo
reduce s m * s n == (m * s n)+ s n .
*** so we can assert
obj LEMMA2 is protecting NAT .
  vars M N : Nat .
  eq s M * N = M * N + N .
endo

obj SUM is protecting NAT .
  op sum : Nat -> Nat .
  var N : Nat .
  eq sum(0) = 0 .
  eq sum(s N) = s N + sum(N) .
endo
***> show sum(n)+sum(n)=n*sn
***> base case
reduce in SUM + LEMMA1 : sum(0) + sum(0) == 0 *
```

```
***> induction step
obj HYP is using SUM + VARS .
  eq sum(n) + sum(n) = n * s n .
endo
reduce in HYP + LEMMA2 : sum(s n) + sum(s n) ==
```

Proofs, cont'd

D. Here's an OBJ *structural induction* proof, as in paper 35.

Proofs, cont'd

```
in list

th FN is sort S .
  op f : S -> S .
endth

obj MAP[F :: FN] is protecting LIST[F] .
  op map : List -> List .
  var X : S .
  var L : List .
  eq map(nil) = nil .
  eq map(X L) = f(X) map(L) .
endo

in map

obj FNS is protecting INT .
  ops (sq_)(dbl_)(_*3) : Int -> Int .
  var N : Int .
  eq sq N = N * N .
  eq dbl N = N + N .
  eq N *3 = N * 3 .
endo

reduce in MAP[(sq_).FNS] : map(0 nil 1 -2 3) .  **
reduce in MAP[(dbl_).FNS] : map(0 1 -2 3) .     **
reduce in MAP[(_*3).FNS] : map(0 1 -2 nil 3) .  **
```