NOTES ON DENOTATIONAL SEMANTICS.

1. Introduction.

Here show, by way of example, the use of denotational semantics for the specification of semantics of programming languages.

In general, a denotational semantic description consists of three parts: a set of syntactic domains, a set of semantic domains, and a set of semantic functions. The semantic functions map syntactic domains into semantic domains. Typically the syntactic domains are AST's; a semantic function might map an AST into a value (in a semantic domain), which "denotes" the meaning of the construct.

Example:

Suppose we were to give the denotational semantic description of a two-operand machine language, for a 16-bit machine with 8 registers. The syntactic domain might be the set of AST's of the form < Opcode Operand Operand >. The semantic domains might as follows:

Register:	{0,1,2,3,4,5,6,7}
Value:	(16-bit binary number)
Address:	(16-bit binary number)
Memory:	Address → Value
RegValues:	Register → Value

A semantic function might be EE (for EEvaluate):

$$EE: AST \rightarrow (RegValues \times Memory) \rightarrow (RegValues \times Memory)$$

EE takes an AST, then the current configuration of the machine, i.e. a (registers,memory) pair, and produces a new (registers,memory) pair that shows the effect of the operation. For example, consider a "register-to-memory" add operation:

$$EE[+ \; r \; m] = \lambda(R,M).\ \text{let Result} = R\ r + M\ m\ \text{in}\ (R,(\lambda a.a\ \text{eq}\ m \rightarrow \text{Result} \mid M\ a))$$

The meaning of an operation "+ r m", where r is a register and m is a memory address, is a function that:

1.- Takes a RegValues function R, and a Memory function M;

2.- Applies R to r (obtaining the contents of register r), applies M to m (obtaining the contents of address m), adds the two values, and calls the result "Result";

3.- Returns a pair consisting of:

  a) The same RegValues fucntion R, and

  b) A new memory function that, when applied to m, returns Result, and behaves like M otherwise.

The "meaning" of the operation is evident in this description: register-to-memory operation "+" adds the contents of the given register and the given memory location. It also updates the given memory location with the result of the addition, and leaves the registers unchanged.

THE DENOTATIONAL SEMANTICS OF TINY.

2. The "$\circ$" operator.

The "$\circ$" operator is defined as follows:

$$\circ = \lambda f.\ \lambda g.\ \lambda x.\ f\ x\ \text{eq error} \rightarrow \text{error} \mid g(f\ x)$$

i.e., "$\circ$" takes two functions, f and g, and yields a new function on x that returns g(f x) if (f x) does NOT evaluate to error, and error if (f x) does evaluate to error. "$\circ$" simply serves as a convenient way to put error checking in our denotational definition everywhere without laboriously specifying it ourselves everywhere. In the denotational semantics description of Tiny we will be using "$\circ$" as an INFIX operator, writing $f \circ g$ instead of $\circ\ f\ g$.

Thus

$$f \circ g = \lambda x.\ f\ x\ \text{eq error} \rightarrow \text{error} \mid g(f\ x)$$

This is purely a syntactic convention of ours that turns out to make denotational semantics descriptions read easier. It allows a "left-to-right flow" reading of such descriptions, exemplified below. Consider an example where

$$f = \lambda y.\ 2/y$$
$$g = \lambda z.\ z+3$$

Then $f \circ g$ =

$$\lambda x.\ (\lambda y.2/y)\ x\ \ \text{eq error} \rightarrow \text{error} \mid (\lambda z.z+3)\ ((\lambda y.2/y)x) =$$
$$\lambda x.\ \ \ \ 2/x\ \ \ \ \text{eq error} \rightarrow \text{error} \mid 2/x+3$$

Consider applying $f \circ g$ to an actual argument, say, 4:

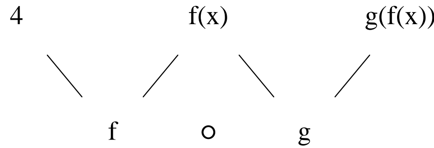
$$(f \circ g)\ 4 =$$
$$2/4\ \text{eq error} \rightarrow \text{error} \mid 2/4+3 =$$
$$2/4+3 = 3\ 1/2.$$

Now consider

$$(f \circ g)\ 0 =$$
$$2/0\ \text{eq error} \rightarrow \text{error} \mid 2/0+3 =$$
$$\text{error, since } 2/0 \text{ eq error (division by zero).}$$

It is to catch things like division by zero or undeclared identifiers that we use "$\circ$".

Now here's the left-to-right bent for "$\circ$". Re-consider $(f \circ g)$ 4. The computation can be depicted by the diagram



4 gets "sent into" f, and the result pops out on the other side of f, i.e. (f x). This result (f x) gets sent into g (unless it equals error, in which case the computation in g is skipped), and the final answer is g(f x). So, you can read "∘" expressions very naturally in a left-to-right manner. In general, read the expression

$$f_1 \circ f_2 \circ \ldots \circ f_k$$

as denoting that function of x that first sends x through f_1, the result of which is sent through f_2, the result of which is sent through f_3, ..., the result of which is sent through f_k, with error checking at each step so that if the evaluation of any f_i applied to its argument is "error", then the evaluation of functions f_{i+1} ... f_k are SKIPPED and the final answer is "error".

"∘" therefore, takes two functions and produces a third.

3. The "= >" operator.

"x = > f" denotes "x eq error → error | f x".

The difference between "∘" and "= >" is that "∘" takes two FUNCTIONS, while "= >" takes a value and a function.

Note: If we have

$$x => f \circ g \circ h \ldots$$

we can replace this with

$$(x => f) => g \circ h \ldots$$

Also, "∘" is left associative.

4. TINY's rules.

4.1. Syntactic domains.

AST= E+C+P, where
E= 0 | 1 | 2 ... | true | false | read | Id | <not E> | <≤ E E> | <+ E E>
C = <:= I E> | <Print E> | <if E C C> | <while E C> | <; C_1 C_2>
P = <program C>

4.2. Semantic Domains.

State: Mem × Input × Output
Mem: Id → Val

Output: Val*
Input: Val*
Val: Num + Bool

4.3. Functionalities of semantic functions.

EE: $E \rightarrow State \rightarrow (Val \times State)$
CC: $C \rightarrow State \rightarrow State$
PP: $P \rightarrow Input \rightarrow Output$

4.4. Definitions of semantic functions.

Auxiliary functions:

Return: $Val \rightarrow State \rightarrow (Val \times State)$
$\lambda v. \quad \lambda s. \quad (v,s)$

Check: $Domain \rightarrow (Val \times State) \rightarrow (Val \times State)$
$\lambda D. \quad \lambda(v,s). \quad v \in D \rightarrow (v,s) \mid error$

Dummy: $State \rightarrow State$
$\lambda s. \quad s$

Cond: $(State \rightarrow State) \rightarrow (State \rightarrow State) \rightarrow (ValxState) \rightarrow State$
$\lambda F_1. \quad \lambda F_2. \quad \lambda(v,s). \quad s => (v \rightarrow F_1 \mid F_2)$

Replace: $Mem \rightarrow Id \rightarrow Val \rightarrow Mem$
$\lambda m. \quad \lambda i. \quad \lambda v. \quad (\lambda i'.\ i' \text{ eq } i \rightarrow v \mid m\ i')$

"Return" takes a value and a state and returns them as a tuple, which is the output of EE. "Check" checks that the Val parameter is in the Domain. If so, it merely returns the Val and State. "Dummy" doesn't do very much: it returns the given state. "Cond" takes two State-to-State functions, and a (Value,State) pair. It returns the result of applying one of the functions to the State, depending on the Value. Finally, "Replace" takes an old Mem, Id, and Val, and produces a new Mem in which the Id is associated with the Val.

Now for EE, CC, and PP:

EE[0] = Return 0; EE[1] = Return 1; EE[2] = Return 2; ... etc. ...

EE[true] = Return true; EE[false] = Return false

EE[read] = λ(m,i,o). Null i $\rightarrow$ error | (Head i, (m, Tail i, o))

> "read" pulls the first input symbol from the input, and replaces the input with the rest of the input. Example: EE[read](m, (5,3,1), o) for any memory m and output o has value (5, (m, (3,1), o)). The latter is a Val $\times$ State pair. The new State reflects the fact that a value has been pulled off the input. Null checks for the nil tuple. Head gets the first element of the tuple, and Tail gets the tuple containing all but the first element.

$EE[I] = \lambda(m,i,o).\ m\ I\ eq\ \bot \rightarrow error \mid (m\ I,\ (m,i,o))$

Look up an identifier; return its value along with the same state that came in.

$EE[<not\ E>] = EE[E] \circ (Check\ Bool) \circ (\lambda(v,s).((not\ v),s))$

Evaluate the expression, make sure it is boolean, and negate it.

$$
\begin{array}{lll}
EE[<\leq E_1\ E_2>] = & EE[E_1] \circ & \\
& (Check\ Num) \circ & \\
& (\lambda(v_1,s_1).s_1 & => EE[E_2] \\
& & => (Check\ Num) \\
& & => (\lambda(v_2,s_2).(v_1 \leq v_2, s_2)) \\
&) &
\end{array}
$$

Evaluate the first argument, then the second. Return their comparison, and the new state.

$$
\begin{array}{lll}
EE[<+\ E_1\ E_2>] = & EE[E_1] \circ & \\
& (Check\ Num) \circ & \\
& (\lambda(v_1,s_1).s_1 & => EE[E_2] \\
& & => (Check\ Num) \\
& & => (\lambda(v_2,s_2).(v_1 + v_2, s_2)) \\
&) &
\end{array}
$$

Evaluate the arguments and return their sum, with the new state.

$CC[<:=\ I\ E>] = \quad EE[E] \circ (\lambda(v,(m,i,o)).\ (Replace\ m\ I\ v,\ i,\ o))$

$CC[<Print\ E>] = \quad EE[E] \circ (\lambda(v,(m,i,o)).\ (m,i,o\ aug\ v))$

$CC[<if\ E\ C_1\ C_2>] = EE[E] \circ (Check\ Bool) \circ (Cond\ CC[C_1]\ CC[C_2])$

$CC[<while\ E\ C>] = EE[E] \circ (Check\ Bool) \circ (Cond\ (CC[<;\ C\ <while\ E\ C>>])\ Dummy)$

$CC[<;\ C_1\ C_2>] = CC[C_1] \circ CC[C_2]$

$PP[<program\ C>] = (\lambda i.\ CC[C]((\lambda i.\bot),\ i,\ nil)) \circ (\lambda(m,i,o).o)$

The meaning of a program <program C> is a function that takes an input i, computes the value of CC[C] applied to an initial configuration with an everywhere-undefined memory, the input i, and a null output, takes the resulting value from CC[C](...) and sends it into a function that discards everything but the output.

END OF DENOTATIONAL DESCRIPTION OF TINY

5. A Tiny example.

Let us work out a TINY example (which won't be so tiny):

What is the meaning of

PP[<program <Print <+ read <+ 1 2>>>>] 5 ?

PP[<program ...>] is a function from input to output; the input in our particular case is nil aug 5; we expect the output to be nil aug (5+1+2) = nil aug 8.

PP[<program <Print <+ read <+ 1 2>>>>] 5
= ((λi. CC[<Print ...>]($\lambda i.\bot$, i, nil)) ∘ (λ(m,i,o).o))5
= 5 => (λi. CC[<Print ...>]($\lambda i.\bot$, i, nil)) ∘ (λ(m,i,o).o)
= (5 => (λi. CC[<Print ...>]($\lambda i.\bot$, i, nil))) => (λ(m,i,o).o)
= CC[<Print ...>]($\lambda i.\bot$, 5, nil) => (λ(m,i,o).o)
= ($\lambda i.\bot$, 5, nil) => CC[<Print ...>] => (λ(m,i,o).o)
= ($\lambda i.\bot$, 5, nil) => EE[<+ read ...>] ∘ (λv,(m,i,o).(m,i,o aug v)) => (λ(m,i,o).o)
= ($\lambda i.\bot$, 5, nil) => EE[<+ read ...>] => (λv,(m,i,o).(m,i,o aug v)) => (λ(m,i,o).o)

Now to send the argument ($\lambda i.\bot$,5,nil) into EE[<+ read...>] , we must first compute the meaning of EE[<+ read...>]. This is a perfect time to test out our claim that we can compute the meaning of a program fragment independently. So let's compute EE[<+ read...>], and then substitute that value back into the above line.

EE[<+ read <+ 1 2>>] = EE[read]
∘ (Check Num)
∘ (λ(v_1,s_1). s_1 => EE[<+ 1 2>]
=> (Check Num)
=> (λ(v_2,s_2).(v_1+v_2,s_2))
)

Well, how about computing EE[<+ 1 2>] first, and then returning to our subproblem.

We now have pushed two problems on our stack, and are evaluating

EE[<+ 1 2>] = EE[1]
∘ (Check Num)
∘ (λ(v_1,s_1). s_1 => EE[2]
=> (Check Num)
=> (λ(v_2,s_2). (v_1+v_2,s_2))
)

What are EE[1] and EE[2]? Return 1 and Return 2, respectively, or λs.(1,s) and λs.(2,s), respectively.

Plug them in:

EE[<+ 1 2>] = λs. (1,s)
∘ (Check Num)
∘ (λ(v_1,s_1). s_1 => λs. (2,s)
=> (Check Num)
=> (λ(v_2,s_2). (v_1+v_2,s_2))
)

Simplify the expression $s_1 => \ldots$:

$$
\begin{aligned}
& s_1 => \lambda s.(2,s) => (\text{Check Num}) => (\lambda(v_2,s_2).(v_1+v_2,s_2)) \\
= \; & (2,s_1) => (\text{Check Num}) => (\lambda(v_2,s_2).(v_1+v_2,s_2)) \\
= \; & (2,s_1) => (\lambda(v_2,s_2).(v_1+v_2,s_2)) \\
= \; & (v_1+2,s_1)
\end{aligned}
$$

So:

$EE[<+ 1 2>] = (\lambda s.(1,s)) \circ (\text{Check Num}) \circ (\lambda(v_1,s_1).(v_1+2,s_1))$

Now compute $(\lambda s.(1,s)) \circ (\text{Check Num})$:

$$
\begin{aligned}
& = \lambda x.(\lambda s.(1,s))x \text{ eq error} \rightarrow \text{error} \mid (\text{Check Num}) ((\lambda s.(1,s))x) \\
& = \lambda x.(1,x) \text{ eq error} \rightarrow \text{error} \mid (\text{Check Num}) (1,x) \\
& = \lambda x.(\text{Check Num}) (1,x) \\
& = \lambda x.(1,x) \\
& = \lambda s.(1,s)
\end{aligned}
$$

So:

$$
\begin{aligned}
EE[<+ 1 2>] & = (\lambda s.(1,s)) \circ (\lambda(v_1,s_1).(v_1+2,s_1)) \\
& = \lambda x. (\lambda s.(1,s)) x \text{ eq error} \rightarrow \text{error} \mid (\lambda(v_1,s_1). (v_1+2,s_1)) ((\lambda s.(1,s))x) \\
& = \lambda x. (1,x) \text{ eq error} \rightarrow \text{error} \mid (\lambda(v_1,s_1). (v_1+2,s_1)) (1,x) \\
& = \lambda x. (\lambda(v_1,s_1).(v_1+2,s_1)) (1,x) \\
& = \lambda x. (1+2,x) \\
& = \lambda x. (3,x) \\
& = \lambda s. (3,s)
\end{aligned}
$$

This is as we expect: <+ 1 2> produces the value 3 and the same input state as it receives.

Now back to the earlier subproblem:

$$
\begin{aligned}
EE[<+ \text{ read } <+ 1 2>>] = \; & EE[\text{read}] \\
& \circ (\text{Check Num}) \\
& \circ (\lambda(v_1,s_1).s_1 \; => (\lambda s.(3,s)) \\
& \qquad\qquad\qquad => (\text{Check Num}) \\
& \qquad\qquad\qquad => (\lambda(v_2,s_2). (v_1+v_2,s_2)) \\
& \;)
\end{aligned}
$$

Simplify the $s_1 => \ldots$ expression:

$$
\begin{aligned}
& s_1 => (\lambda s.(3,s)) => (\text{Check Num}) => (\lambda(v_2,s_2).(v_1+v_2,s_2)) \\
= \; & (3,s_1) => (\text{Check Num}) => (\lambda(v_2,s_2).(v_1+v_2,s_2)) \\
= \; & (3,s_1) => (\lambda(v_2,s_2). (v_1+v_2,s_2)) \\
= \; & (v_1+3,s_1)
\end{aligned}
$$

So

$$EE[<+ \text{ read } <+ 1 2>>] = EE[\text{read}] \circ (\text{Check Num}) \circ (\lambda(v_1,s_1).(v_1+3,s_1)) .$$

Now let us substitute this value back into our computation for PP[<program ...>]. If you recall, our last line in this computation was

$$(\lambda i.\bot,\ 5,\ nil) => EE[<+\ read\ ...>] => (\lambda(v,(m,i,o)).(m,i,o\ aug\ v)) => (\lambda(m,i,o).o)\ .$$

This now becomes

$$
\begin{array}{lll}
 & (\lambda i.\bot,5,nil) & => EE[read] \\
 & & \circ\ (Check\ Num) \\
 & & \circ\ (\lambda(v_1,s_1).(v_1+3,s_1)) \\
 & & => (\lambda(v,(m,i,o)).(m,i,o\ aug\ v)) \\
 & & => (\lambda(m,i,o).o) \\
 & & \\
= & (\lambda i.\bot,5,nil) & => EE[read] \\
 & & => (Check\ Num) \\
 & & => (\lambda(v_1,s_1).(v_1+3,s_1)) \\
 & & => (\lambda(v,(m,i,o)).(m,i,o\ aug\ v)) \\
 & & => (\lambda(m,i,o).o) \\
= & Null\ 5 \rightarrow error| & \\
 & (Head\ 5,(\lambda i.\bot,Tail\ 5,nil)) & => (Check\ Num) \\
 & & => (\lambda(v_1,s_1).(v_1+3,s_1)) \\
 & & => (\lambda(v,(m,i,o)).(m,i,o\ aug\ v)) \\
 & & => (\lambda(m,i,o).o) \\
 & & \\
= & (5,(\lambda i.\bot,nil,nil)) & => (Check\ Num) \\
 & & => (\lambda(v_1,s_1).(v_1+3,s_1)) \\
 & & => (\lambda(v,(m,i,o)).(m,i,o\ aug\ v)) \\
 & & => (\lambda(m,i,o).o) \\
 & & \\
= & (5,(\lambda i.\bot,nil,nil)) & => (\lambda(v_1,s_1).(v_1+3,s_1)) \\
 & & => (\lambda(v,(m,i,o)).(m,i,o\ aug\ v)) \\
 & & => (\lambda(m,i,o).o) \\
 & & \\
= & (8,(\lambda i.\bot,nil,nil)) & => (\lambda(v,(m,i,o)).(m,i,o\ aug\ v)) \\
 & & => (\lambda(m,i,o).o) \\
 & & \\
= & (\lambda i.\bot,nil,nil\ aug\ 8) & => (\lambda(m,i,o).o) \\
 & & \\
= & nil\ aug\ 8\ (Whew!) &
\end{array}
$$

An exercise. Denotational semantics have a property that no other method seen so far has had: The meaning of any construct can be obtained, usually as a function of the state in which that meaning is to be evaluated. Note that we refer to the meaning of ANY construct, i.e. any portion of a program. To illustrate further this property, and as an exercise, you may want to derive the value of

PP[<program <Print <+ read <+ 1 2>>>>] (notice we have NOT applied it to 5),

and then apply the result to 5 to see if you get the answer 8.