Denotational Semantics

Establish the meaning of a program by specifying a meaning for each *phrase* of the Abstract Syntax Tree (AST).

- declarations
- commands
- expressions

The mean of a phrase is defined by the meaning of each sub-phrase.

- The meaning is its *denotation.*
- Established by providing *semantic functions* that map phrases to their denotations.

Example: Binary Numbers

Numerals are syntactic entity; numbers are *semantic*.

```
Numeral -> 0
        -> 1
        -> Numeral 0
        -> Numeral 1
```

Each (binary) numeral denotes a single member of the domain:

```
Natural = { 0, 1, 2, ... }
```

to formalize this, we define the semantic function:

```
valuation: Numeral -> Natural
```

with the definition:

```
valuation [[0]] = 0
valuation [[1]] = 1
valuation [[N 0]] = 2*valuation N
valuation [[N 1]] = 2*valuation N + 1
```

The valuation function's 4 equations correspond to the phrases of the (abstract) syntax for our binary language.

Look at evaluation sequence for "110":

```
valuation[[110]]
   = 2*valuation[[11]]
   = 2*(2*valuation[[1]]+1)
   = 2*(2*1+1)
   = 6
```


Calculator Example

Abstract Syntax

```
Command -> Expression =

Expression -> Numeral
           -> Expression + Expression
           -> Expression - Expression
           -> Expression * Expresssion
```

Support Functions:

sum: Integer x Integer -> Integer
difference: Integer x Integer -> Integer
product: Integer x Integer -> Integer

Semantic Functions:

exec: Command -> Integer
eval: Expression -> Integer
valuation: Numeral -> Natural

Calculator Example (cont)

Function Definitions:

```
exec[[ E = ]] = eval E

eval[[ N ]] = valuation N
eval[[ E1+E2 ]] = sum(eval E1,eval E2 )
eval[[ E1-E2 ]] = difference(eval E1,eval E2 )
eval[[ E1*E2 ]] = product(eval E1,eval E2 )
```

Example

```
execute[[ 40-3*9= ]]
   = eval[[ 40-3*9 ]]
   = product(eval[[ 40-3]], eval[[ 9 ]])
   = product(difference(eval[[40]], eval[[ 3 ]]),
              valuation[[9]])
   = product(difference(40,3),9)
   = 333
```


Basic Ideas of DS

- Each phrase of the language is specified by a value in some domain. The value is the *denotation* of the phrase (or, the phrase *denotes* the value)
- For each phrase **class** (group, collection, etc.), we provide a domain D of its denotations, and introduce a *semantic function* **f** to map each phrase to its denotation. Notationally, this is written:

  ```
  f: P -> D
  ```

- Each semantic function is defined by a number of *semantic equations*, one for each distinct phrase in the phrase class.

Domains

Domains are “sets” of values. The following structures are often used.

- Primitive Domains
  integers, characters, truth-values, enumerations, etc.

- Cartesian Product Domains
  Cross product of multiple domains.

```
let pay = (rate x hours, dollars) in
  ...
let (amount,denom) = pay in
```

- Disjoint Union Domains
  Elements are chosen from *either (any)* component domain, and appropriately **tagged**.

```
shape = rect( RxR ) + circle(R) + point
```

A 'tagged' union...

- Function Domains
  Each element of the function domain D -> D’ is a function that maps elements of D to elements of D’.

  Assume domain: *Integer -> Truth-Values*. Example elements are:

```
odd, even, positive, prime, etc.
```

- Sequence Domains
  Generally used for mapping I/O streams.

  Each element of the sequence domain D* is a *finite* sequence of zero or more elements of D.

  if $(x \in D)$ and $(s \in D^*)$ then $x \bullet s \in D^*$

Why Domains

With iterative and recursive programs, there is always the possiblity that a program will not terminate (normally).

- Divide by zero

For this reason, we cannot use simple sets to represent values (or denotations). We introduce the symbol "$\perp$" to represent computations that fail to terminate normally.

```
reciprocal 0 -> ⊥
```

A computation that results in "$\perp$" contains less information than one that results in a value from the domain.

We define the relation "$\leq$" over the domain to establish when elements contain less information. This relation establishes a *partial order* over elements of the domain.

reflexive	$x \leq x$
symmetric	$x \leq y$ and $y \leq x \Rightarrow x = y$
transitive	$x \leq y$ and $y \leq z \Rightarrow x \leq z$

Domain Diagrams

- Primitive
  $\{ \perp, f, t \}$, $\{ \perp, u, d \}$, $\{ \perp, 0, 1, 2, 3, \ldots \}$

  $x \leq x'$ iff $x = x'$ or $x = \perp$

- Cartesian Products
  Truth-Value x Direction =
  $\{ (\perp,\perp), (f,\perp), (\perp,u), (\perp,d), (t,\perp), (f,u), (f,d), (t,u), (t,d) \}$

  $(x,y) \leq (x',y')$ iff $(x \leq x')$ and $(y \leq y')$

- Disjoint-Unions
  Truth-Value + Direction =
  { *left* f, *left* t, *right* u, *right* d }

left $x \leq$ *left* x'	iff $x \leq x'$
right $x \leq$ *right* x'	iff $x \leq x'$
$\perp \leq z$	for all z

The relation "≤" allows comparison of information content between two elements. "$x \leq y$" means that y can be obtained by adding information to 'x' (but not *changing* that already in x).

Strict functions: $f\ \perp \text{->} \perp$

Consider:

$f1 = \{ up \rightarrow false, down \rightarrow true, \perp \rightarrow \perp \}$ $\quad \lambda x.(x == down)$
$f2 = \{ up \rightarrow false, down \rightarrow false, \perp \rightarrow false \}$ $\quad \lambda x.(false)$
$f3 = \{ up \rightarrow false, down \rightarrow \perp, \perp \rightarrow true \}$
$f4 = \{ up \rightarrow true, down \rightarrow true, \perp \rightarrow false \}$

F3: ($\perp \leq$ *down*), yet (f3 down) $\leq$ (f3 $\perp$)

F4: can establish if computation terminates.

monotonic: $(\forall x,x')\ (x \leq x') \Rightarrow (fx \leq fx')$

The more information we add to an argument, the more information will be obtained by applying the function to the argument.

- function domains
  All elements of function domains are monotonic.
  (More information in argument, the more to be obtained by applying the fuction to the argument).

Given domain D = A -> B,

$\forall\ (f, f' \in D)\ f \leq f'$ iff $(\forall\ x \in A)\ fx \leq f'\,x$

Consider domain *Direction -> Truth-Value*

$\{ \quad \{u\rightarrow f, d\rightarrow f, \perp\rightarrow f\}, \{u\rightarrow t, d\rightarrow t, \perp\rightarrow t\},$
$\{u\rightarrow f, d\rightarrow f\}, \{u\rightarrow f, d\rightarrow t\}, \{u\rightarrow t, d\rightarrow f\}, \{u\rightarrow t, d\rightarrow t\},$
$\{u\rightarrow f\}, \{d\rightarrow f\}, \{d\rightarrow t\}, \{u\rightarrow t\},$
$\{\}$
$\}$

(unspecified entries go to "$\perp$")

A Model for Storage

A location in storage is either

- unused
- *undefined*
- or holds a value (storable values)

Given domains *Location*, *Storables* and a *Store*, use functions:

- empty-store: Store
- allocate: Store -> Store x Location
- deallocate: Store x Location -> Store
- update: Store x Location x Storable -> Store
- fetch: Store x Location -> Storable

allocate S = (S', L) where L is *unused* in S but *undefined* in S'
deallocate(allocate store) = store
fetch(update(store, location, storable), location) = storable

Essentially, Store is the domain:

Store = Location $\rightarrow$ (*stored* Storable + *undefined* + *unused*)

Define auxilary functions:

empty-store = λloc.*unused*

allocate store =
let loc = get-unused-loc(store) **in** (store[loc$\rightarrow$ *undefined*], loc)

deallocate (store, location) = store[location$\rightarrow$ *unused*]

update(store, loc, storable) = store[loc $\rightarrow$ *stored* storable]

fetch(store, location) =
let stored-value(*stored* storable) = storable
stored-value(*undefined*) = fail
stored-value(*unused*) = fail
in
stored-value(store (location))

Example: Command Language

```
Command -> Expression =
        -> Expression = Register
        -> Command Command
Expression -> Numeral
           -> Expression + Expression
           -> Expression - Expression
           -> Expression * Expression
           -> Register
Register -> X
         -> Y
```

Storable = Integer
Location = { loc1, loc2 }

eval:	Expression -> (Store -> Integer)
exec:	Command -> (Store -> Store x Integer)
loc:	Register -> Location

eval[[N]] sto = valuation N
eval[[E_1+E_2]] sto =
  sum(eval E_1 sto, eval E_2 sto)
eval[[Register]] sto = fetch(sto, loc R)

Similarly for other operators...

exec[[E =]] sto =
  let int = eval E sto **in** (sto, int)

exec[[E = R]] sto =
  let int = eval E sto **in**
  let sto’ = update(sto, loc R, int) **in**
  (sto’, int)

exec[[C_1 C_2]] sto =
  let (sto’, int) = exec C_1 sto **in**
  exec C_2 sto’

loc[[X]] = loc_1
loc[[Y]] = loc_2

exec [[**4 * 3 = X X + 5 =**]] empty-store

= **let** (sto$_1$, int) = exec[[4*3 = X]] empty-store **in**
  exec [[X + 5 =]] sto$_1$

= **let** (sto$_1$, int$_1$) =
  let int$_2$
    = mult(eval 4 empty-store, eval 3 empty-store) **in**
  let sto$_2$ = update(empty-store, loc X, int$_2$) **in**
  (sto$_2$, int$_2$)
in
exec [[X + 5 =]] sto$_1$

= **let** (sto$_1$, int$_1$) =
  let int$_2$ = mult(4, 3) **in**
  let sto$_2$ = update(empty-store, loc X, int$_2$) **in**
  (sto$_2$, int$_2$)
in
exec [[X + 5 =]] sto$_1$

= **let** (sto$_1$, int$_1$) =
  let sto$_2$ = update(λloc.*unused*, loc$_1$, 12) **in** (sto$_2$, 12)
in exec [[X + 5 =]] sto$_1$

= **let** (sto$_1$, int$_1$)
  = **let** sto$_2$ = λloc.*unused* [loc$_1$ → 12] **in** (sto$_2$, 12)
in exec [[X + 5 =]] sto$_1$

= exec [[X + 5 =]] λloc.*unused* [loc$_1$ → 12]

= **let** int = eval[[X + 5 =]] (λloc.*unused* [loc$_1$ → 12])
in (λloc.*unused* [loc$_1$ → 12], int)

= **let** int = sum(eval X (λloc.*unused* [loc$_1$ → 12]),
      eval 5 (λloc.*unused* [loc$_1$ → 12]))
in (λloc.*unused* [loc$_1$ → 12], int)

= **let** int = sum(eval X (λloc.*unused* [loc$_1$ → 12]), 5)
in (λloc.*unused* [loc$_1$ → 12], int)

= **let** int = sum(fetch((λloc.*unused* [loc$_1$→ 12]), loc[[X]]), 5)
in (λloc.*unused* [loc$_1$ → 12], int)

= **let** int = sum(fetch((λloc.*unused* [loc$_1$→ 12]), loc$_1$), 5)
in (λloc.*unused* [loc$_1$ → 12], int)

= **let** int = sum(12, 5)
in (λloc.*unused* [loc$_1$ → 12], int)

A Model for Environments

This is similar to our model for storage, except that we're *binding* Identifiers (generally) to bindable values. Assume the domains *Identifier*, *Bindable* and *Environ*. and functions:

empty:	Environ
bind:	Identifier x Bindable -> Environ
overlay:	Environ x Environ -> Environ
find:	Environ x Identifier -> Bindable

Bind creates the singleton environment.

Overlay combines environments (but overides duplicate mappings).

env1 = overlay(bind(i, 1), bind(j, 2))
env2 = overlay(bind(j, 3), bind(k, 4))

overlay(env2, env1) = { i ->1, j->3, k->4 }.

Given the defintion for *Environ* as:

$$\text{Environ} = \text{Identifier} \to (\textit{bound}\ \text{Bindable} + \textit{unbound})$$

we can define auxilary functions:

empty-environ = λloc.*unbound*

bind(I, bindable) =
  λI'.if I' = I then *bound* bindable else *unbound*

Recall that we're defining a new environment

overlay(env', env) =
  λI.if env'(I) $\neq$ *unbound* then env'(I) else env(I)

find(env, I) =
  let bound-value(*bound* bindable) = bindable
    bound-value(*unbound*) = fail
  in
  bound-value(env(I))

Example: Declaration Language

```
Expression -> Numeral
           -> Expression + Expression
           -> ...
           -> Identifier
           -> let Declaration in Expression

Declaration -> val Identifier = Expression
```

Define Domain

Bindable = Integer

And Denotations:

evaluate: Expression -> (Environ -> Integer)
elaborate: Declaration -> (Environ -> Environ)


```
evaluate[[ N ]] env = valuation N
evaluate[[ E1+E2 ]] env =
    sum(evaluate E1 env, evaluate E2 env)
```

Similarly for other operators...

```
evaluate[[ I ]] env = find( env, I )
evaluate[[ let D in E ]] env =
    let env' = elaborate D env in
    evaluate E (overlay(env', env))


elaborate[[ val I = E ]] env =
    bind( I, evaluate E env )
```

Assume env_1 = { m -> 10 }

eval [[**let** val n = m + 5 **in** m + n]] env_1
= **let** env2 = elaborate[[val n = m + 5]] env_1 **in**
eval [[m+n]] overlay(env_2,env_1)
= **let** env2 = bind(n, eval[[m+ 5]] env_1) **in**
eval [[m+n]] overlay(env_2,env_1)
= **let** env2 = bind(n, sum(eval[[m]]env_1, eval[[5]] env_1)) **in**
eval [[m+n]] overlay(env_2,env_1)
= **let** env2 = bind(n, sum(find(env_1, m), 5)) **in**
eval [[m+n]] overlay(env_2,env_1)
= **let** env2 = bind(n, sum(10, 5)) **in**
eval [[m+n]] overlay(env_2,env_1)
= **let** env2 = bind(n, 15) **in**
eval [[m+n]] overlay(env_2,env_1)
= eval [[m+n]] { m->10, n->15 }
= sum(eval[[m]]{m->10, n->15}, eval[[n]]{m->10, n->15})
= sum(find({m->10, n->15}, m), find({m->10, n->15}, n))
= sum(10, 15)
= 25

A Simple Imperative Language

```
Command -> skip
        -> Identifier := Expr
        -> let Declaration in Command
        -> Command ; Command
        -> if Expr then Command else Command
        -> while Expr do Command
Expr -> Numeral
     -> false | true
     -> Identifier
     -> Expr + Expr
     -> Expr < Expr
Declaration  -> const Identifer ~ Expr
             -> var Identifier : Type
Type -> bool
     -> int
```

Identifiers must be used in scope
Expressions are typed

Define Domains

Value = *truth-value* Truth-Value + *integer* Integer
Storable = Value
Bindable = *value* Value + *variable* Location

And Denotations:

exec: Command -> (Environ -> Store -> Store)
eval: Expression -> (Environ -> Store -> Value)
elab: Declaration -> (Environ-> Store -> Environ x Store)

Commands

```
exec[[ skip ]] env sto = sto

exec[[ I := E ]] env sto =
    let val = eval E env sto in
    let variable loc = find( env, I ) in
    update( sto, loc, val )

exec[[ let D in C ]] env sto =
    let (env', sto') = elaborate D env sto in
    execute C (overlay (env', env )) sto'
```

need sto', since it might allocate storage

```
exec[[ C1 ; C2 ]] env sto =
    exec C2 env (exec C1 env sto)

exec[[ if E then C1 else C2 ]] env sto =
    if eval E env sto = truth-value true
        then exec C1 env sto
        else exec C2 env sto
```

```
exec[[ while E do C ]] =
   let executeWhile env sto =
      if eval E env sto = truth-value true
         then executeWhile env
                 (execute C env sto)
         else sto
   in
   executeWhile
```

Note that *executeWhile* is recursively defined.

Expressions

```
eval[[ N ]] env sto = integer(valuation N)
eval[[ false ]] env sto = truth-value false
eval[[ I ]] env sto =
   let coerce( sto, value val ) = val
         coerce( sto, variable loc ) =
            fetch( sto, loc )
   in coerce( sto, I )

eval[[ E1 + E2 ]] env sto =
   let integer int1 = eval E1 env sto in
   let integer int2 = eval E2 env sto in
   integer( sum( int1, int2 ))

eval[[ E1 < E2 ]] env sto =
   let integer int1 = eval E1 env sto in
   let integer int2 = eval E2 env sto in
   truth-value( less( int1, int2 ))
```

Elaborations

Recall that the domain for elaborations is:

elab: Declaration -> (Environ-> Store -> Environ x Store)

```
elab[[ const I ~ E ]] env sto =
   let val = eval E env sto in
   (bind( I, value val), sto)
```

In this case, only the environment has been modified

```
elab[[ var I : T ]] env sto =
   let (sto', loc) = allocate sto in
   (bind( I, variable loc), sto')
```

In this case, both the environment and the storage have been changed.

Abstractions

An *abstraction* is a value that embodies a computation.
One *calls* the abstraction to perform the computation, but only the final result is visible (not the method employed).

Extend the expression abstract syntax to add functions of 1 parameter:

```
Expr -> ...
     -> Identifier ( Actual-Parameter )

Decl -> ...
     -> fun Identifer ( Formal-Parameter )
          = Expr

Formal-Parmeter -> Identifier : Type

Actual-Parameter -> Expression
```

Define Domains

Argument = Integer
Function = Argument -> Integer

Only integers as parameters, only integers for results...

Bindable = *integer* Integer + *function* Function

Both integers and function abstractions are bindable.

eval: Expression -> (Environ -> Integer)
elab: Declaration -> (Environ-> Environ)

Elaborating a formal param binds the identifier to the argument:

bindParam: Formal-Parameter -> (Argument-> Environ)

Actual arguments are evaluated like expressions

giveArg: Actual-Parameter -> (Environ -> Argument)

The semantic functions for these are:


```
bindParam[[ Ident : Type ]] arg
   = bind( Ident, arg )

giveArg[[ E ]] env = eval E env
```

The equation for the function call is:

```
eval[[ Ident( AP )]] env
   =  let function f = find( env, Ident ) in
      let arg = giveArg AP env in
      f arg
```

That is, the function call is no more than applying the argument to the function as defined in the current environment. The function definition is:

```
elab[[ fun I ( FP ) = E ]] env
   = let f arg =
         let parEnv = bindParam FP arg in
         eval E (overlay( parEnv, env ))
      in bind( I, function f )
```

This is a *static* binding (environment is bound at declaration).

Procedures

In general, functions and procedures can modify the external storage. Essentially, store is an additional argument:

Procedure = Argument -> Store -> Store

```
elab[[ proc id( FP ) ~ C ]] env sto =
   let proc arg sto' =
      let env' = bindParam FP arg in
         execute C (overlay( env', env )) sto
   in
   (bind( id, procedure proc ), sto )
```

```
execute[[ id( AP ) ]] env sto =
   let procedure proc = find( env, id ) in
   let arg = giveArg AP env sto in
   proc arg sto
```


Parameters

We modelled only simple (value) type parameters. Most languages support more complex passing schema.

```
formalParam  -> const id : type
             -> var id : type

actualParam  -> expr
             -> var id
```

Recall that this is *abstract* syntax. Annotating the actual parameter with "var" expresses the semantics of the call directly.

Now, *giveArg: actualParam -> (Environ -> Store -> Argument)*

```
giveArg[[ E ]] env sto = value(eval E env sto)
giveArg[[ var id ]] env sto =
   let variable loc = find( env, id ) in
   variable loc
```

The latter definition gives us the *location* of the parameter (rather than simply evaluating it).

Copy-In, Copy-Out Parameters

copyIn: FormalParm -> (argument -> store -> environ x store)
copyOut:
FormalParam -> (Environ -> argument -> store -> store)

```
copyIn[[ value id:t ]] (value val) sto =
   let (sto', local) = allocate sto in
   (bind(id, variable local),
      update( sto', local, val ))

copyIn[[ result id:t ]] (variable loc) sto =
   let (sto', local) = allocate sto in
   (bind(id, variable local), sto' )

copyOut[[ value id:t]] env(value val) sto = sto

copyOut[[result id:t]] env(variable loc) sto =
   let variable local = find( env, id ) in
   update( sto, loc, fetch( sto, local ))
```


Finally, the procedure elaboration:

```
elab[[ proc id( FP ) ~ C ]] env sto =
   let proc arg sto' =
      let (parenv, sto'')
            = copyIn FP arg sto' in
      let sto'''
         = exec C (overlay(parenv,env)) sto''
      in copyOut FP parenv arg sto'''
   in
   (bind( id, procedure proc ), sto )
```

note that ‘C’ will not be executed until procedure invocation.

Composite Types

Primary issue is whether fields of the composite variable can be selectively updated? If *not*, we have a simpler system.

Assume language capabilities:

```
const   z ~ (true, 0);
var     p,q : (bool, int);
...
p := z;
q := (false, snd p + 1);
```

First try ... composite pairs

```
expr -> ...
     -> ( expr, expr )
     -> fst expr
     -> snd expr
type -> bool
     -> int
     -> ( type, type )
```


Add a domain to model the paired value And update the domain of values:

Pair-Value = Value x Value
Value = *truthValue* Truth-Value + *integer* Integer +
pairValue Pair-Value
Storable = Value

We need to update the *eval* function to add the new construct:

```
eval[[ (E1, E2)]] env sto
   =  let val1 = eval E1 env sto in
      let val2 = eval E2 env sto in
      pair-value (val1, val2)
```

And add the extraction methods:

```
eval[[ fst E ]] env sto =
   let pairValue(v1,v2)=eval E env sto in v1
eval[[ snd E ]] env sto =
   let pairValue(v1,v2)=eval E env sto in v2
```

Second Try ... Selective Updates

First, augment the abstract grammar:

```
command -> ...
        -> V-name := expr

expr -> ...
     -> V-name
     -> ( expr, expr )

V-name  -> Identifier
        -> fst V-name
        -> snd V-name

type -> bool
     -> int
     -> ( type, type )
```

Vname can be a simple identifier, or reference to either of the fields of the pair.

Domains:

Pair-Value = Value x Value
Value = *truthValue* Truth-Value + *integer* Integer +
pairValue Pair-Value

Pair-Variable= Variable x Variable

However, pairs are no longer storable, so:

Storable = *truthValue* Truth-Value + *integer* Integer

Variable = *primitive* Location + *pairVariable* PairVariable

The language allows for fetch/store of paired variables, so we need to incorporate that. We add the following access functions:

fetchVar: Store x Variable -> Value
updateVar: Store x Variable x Value -> Store

With the definitions:

fetchVar(sto, *primitive* loc) = fetch(sto, loc)
fetchVar(sto, *pairVariable*(var_1, var_2)) =
pairValue(fetchVar(sto, var_1), fetchVar(sto, var_2))

updateVar(sto, *primitive* loc, storable)
= update(sto, loc, storable)
updateVar(sto, *pairVariable*(var_1, var_2), *pairValue*(val_1, va1_2))
= **let** sto' = updateVar(sto, var_1, val_1)
in updateVar(sto', var_2, va1_2)

Now, consider:

execute[[V := E]] env sto
= **let** val = eval E env sto **in**
let *variable* var = <u>identify</u> V env **in**
updateVar(sto var, val)

Simple. except we need to define *identify.* And we need to allocate storage for variables.

ValueOrVariable = *value* Value + *variable* Variable

identify: V-name -> (Environ -> ValueOrVariable)

```
identify[[ I ]] env = find( env, I )

identify[[ fst V ]] env =
   let
      first( value( pairValue( v1, v2 ))
            = value v1
      first( variable( pairVariable( w1, w2))
            = variable w1
   in first( identify V env )
```

First is a local function that matches one of the two forms of a *ValueOrVariable*, and returns that form (properly tagged).

The form for `identify[[ snd V ]]` is similarly defined.

Finally, we do the storage allocation

Let the domain be:

Allocator = Store -> Store x Variable

And the semantic function:

allocateVar: Type -> Allocator

This fits into the existing elaboration functions:

```
elab[[ var id : T ]] env sto
   = let (sto', var ) = allocateVar T sto in
      (bind( I, var ), sto')

allocateVar[[ bool ]] sto
   = let (sto', loc) = allocate sto in
      (sto', primitive loc)

allocateVar[[ (T1, T2) ]] sto =
   let (sto', var1) = allocateVar T1 sto in
   let (sto'', var2) = allocateVar T2 sto in
      (sto'', pairVariable (var1, var2))
```


Failure

In direct semantics, failure clauses must be incorporated into the semantic equations:

```
sum: Integer x Integer -> Integer

sum( int1, int2 )
   = if abs( int1 + int2 ) <= maxint
        then int1 + int2
        else fail
sum( fail, int )  = fail
sum( int, fail )  = fail
sum( fail, fail ) = fail
```

Consider the equations:

```
exec[[ I := E ]] env sto =
    let val = eval E env sto in
    let variable loc = find( env I ) in
    update( sto, loc, val )
```

If “E” evaluates to *fail,* then *update* will yield *fail,* and the entire clause will yield *fail.*

exec[[C_1; C_2]] env sto =
  exec C_2 env (exec C_1 env sto)

If “c1” *fails,* then this is applied to the execution of “c2”, which will also fail.