Example: the language TINY

- Two kinds of constructs: expressions (**E**) and commands (**C**)
- Both constructs can contain identifiers (**I**)—strings of letters and digits beginning with a letter

Syntax of TINY:

E ::= 0 | 1 | true | false | I | not E | E_1 = E_2 | E_1 + E_2

C ::= I := E | output E | if E then C_1 else C_2 |
while E do C | C_1; C_2

Standard semantics

- Denotational semantics is not able to handle constructs of real languages like
  - jumps, or
  - aliasing (multiple names of one variable).
- Therefore, more sophisticated denotations are needed—*standard semantics* is used.
- Standard semantics is based on
  - transforming states indirectly via *continuations*,
  - splitting the binding of identifiers to values into two parts: id→variable, variable→value.

Direct semantics vs. continuation semantics

- Denotational semantics is an example of *direct semantics*.
- In direct semantics:
  - each construct *directly* denotes its input/output transformation;
  - the transformation of the complete program is a combination of its components' transformations;
  - the result of a construct is <u>always</u> passed to the rest of the program $\Rightarrow$ the rest of the program has to cope with abnormal values $\Rightarrow$ it is stuffed with test for these values.

Direct semantics vs. continuation semantics

- In continuation semantics:
  - denotation of a construct depends on the rest of the program—or *continuation*—following it;
  - each construct decides for itself where to pass its result:
    - usually to the *normal continuation* (corresponds to the textually following code);
    - to the continuation corresponding to an error stop;
    - to the continuation corresponding to the code following a label, target of a jump.

Continuations

- A continuation is a function from an intermediate result expected by the rest of the program (e.g. state, or value–state pair) to the *final answer* (e.g. state + error, or output + error).
- *Command continuations* correspond to the rest of the program following a command and form a domain

  Cont = State → [State + {error}]
- *Expression continuations* correspond to the code following expressions and form a domain

  Econt = Value → State → [State + {error}]

Continuation denotations of constructs

- In a continuation semantics the denotations of constructs are functions of continuations and states.
- The continuation semantic functions are of type:

  $\boldsymbol{E}$: **Exp** $\rightarrow$ **Econt** $\rightarrow$ **State** $\rightarrow$ **[State + {error}]**

  $\boldsymbol{C}$: **Com** $\rightarrow$ **Cont** $\rightarrow$ **State** $\rightarrow$ **[State + {error}]**

  and they are defined so that:

$$\boldsymbol{E}\,[\mathrm{E}]\;\mathrm{k}\;\mathrm{s} = \begin{cases} \mathrm{k}\;\mathrm{v}\;\mathrm{s'} & \text{E has value v, transforms s to s'} \\ \textbf{error} & \text{otherwise} \end{cases}$$

$$\boldsymbol{C}\,[\mathrm{C}]\;\mathrm{c}\;\mathrm{s} = \begin{cases} \mathrm{c}\;\mathrm{s'} & \text{if C transforms s to s'} \\ \textbf{error} & \text{otherwise} \end{cases}$$

Sample semantic clauses of TINY

- Domains

  State = Memory $\times$ Input $\times$ Output

  Memory = Ide $\rightarrow$ [Value + {unbound}]

  Input = Value*

  Output = Value*

  Value = Num + Bool

  Cont = State $\rightarrow$ [State + {error}]

  Econt = Value $\rightarrow$ Cont

- Functions

  E: Exp $\rightarrow$ Econt $\rightarrow$ Cont

  C: Com $\rightarrow$ Cont $\rightarrow$ Cont

Sample semantic clauses of TINY

- Expressions:

  $\boldsymbol{E}$ **[0]** k s = k 0 s, or by canceling s: $\boldsymbol{E}$ **[0]** k = k 0

  $\boldsymbol{E}$ **[read]** k (m,i,o) = **null** i $\rightarrow$ **error**, **k (hd i)** (m,**tl** i,o)

  $\boldsymbol{E}$ [I] k (m,i,o) = (m I = **unbound**) $\rightarrow$ **error**, k (m I) (m,i,o)

  $\boldsymbol{E}$ **[not** E] k s = $\boldsymbol{E}$ **[E]** (λ v s' . **isBool v** $\rightarrow$ k (not v) s',
  error) s

- Commands:

  $\boldsymbol{C}$ [I := E] c = $\boldsymbol{E}$ [E] (λ v (m,i,o) . c (m[v/I],i,o))

  $\boldsymbol{C}$ **[output** E] c = $\boldsymbol{E}$ [E] (λ v (m,i,o) . c (m,i,v.o))

  $\boldsymbol{C}$ [C_1; C_2] c s = $\boldsymbol{C}$ [C_1] **(**$\boldsymbol{C}$ [C_2] c) s

Final answer of the program

- A state as the final answer of running a program is unnatural. In practice, it is just output.
- Once outputted information should not be retrieved by the rest of the program $\Rightarrow$ the output must not be passed to it as a member of state.
- Once outputted information must not be lost, if an error occurs (probe ***C*** **[output 0]** c with c **=** λ**s.error**).
- An output of a program need not be finite. Consider the nonterminating program

  x:=0; while true do (output x; x:=x+1)

  Its output is 0.1.2.3…

Final answer of the program

- New domain equations are:
  State = Memory $\times$ Input
  Memory = Ide $\rightarrow$ [Value + {unbound}]
  Input = Value*
  Value = Num + Bool
  Cont = State $\rightarrow$ Ans
  Econt = Value $\rightarrow$ Cont
  Ans = {error, stop} + [Value $\times$ Ans]
- The semantic clause for output has to be changed:
  C **[output** E**]** c = ***E*** **[**E**]** (λ v s . (v, c s))

Sharing

- Sometimes distinct identifiers can denote the same variable $\Rightarrow$ assigning to one of them will change the value of the others.
- Sharing may occur:
  - directly—as the result of explicit command or declaration, e.g. **let** I_1 **==** I_2.
  - indirectly—e.g. (in PASCAL) by declaring a procedure of the form

    procedure P(**var** x:**real**, **var** y:**real**)…

    and executing a call **P**(z,z).

    Both x and y share the variable denoted by z.

Locations

- Sharing is enabled by two-level association between identifiers and values:
  1. an identifier is bound to a variable,
  2. the variable is bound to a value.
- In formal semantics, the term *location* is used rather than *variable*.
- Locations are modeled by the domain **Loc**.
- The only structure on **Loc** is **=**: $\mathbf{Loc} \times \mathbf{Loc} \to \mathbf{Bool}$ which tests locations for equality.

Stores

- *Stores* model the binding of locations to values.
- The domain **Store** is defined as

  $$\mathbf{Store} = \mathbf{Loc} \rightarrow [\mathbf{Sv} + \{\mathbf{unused}\}]$$

  where **Sv** is a domain of *storable values*.
- The function **new**: **Store** $\rightarrow$ **[Loc + {error}]** returns an unused location, or error, if an unused location is not available.
- The notation $v_1,\ldots,v_n/i_1,\ldots,i_n$ denotes the "little" store:

  $\lambda i \,.\, i = i_1 \rightarrow v_1, \ldots, i = i_n \rightarrow v_n,$ **unused**

Environments

- *Environments* model binding of identifiers.
- The domain **Env** of environments is defined as

  $$\mathbf{Env} = \mathbf{Ide} \rightarrow [\mathbf{Dv} + \{\mathbf{unbound}\}]$$

  typical members will be r, r', r_1, r_2 etc.
- **Dv** is the domain of *denotable values*. Sometimes, it can identify with **Loc**, but for most languages it contains also constants, procedures etc.
- For $d_1,\ldots,d_n \in \mathbf{Dv}$, $I_1,\ldots,I_n \in \mathbf{Ide}$, $r_1, r_2 \in \mathbf{Env}$ there are defined the following notations:
  - $d_1,\ldots,d_n/I_1,\ldots,I_n = (\lambda I\ .\ I{=}I_1 \rightarrow d_1,\ldots,\ I{=}I_n \rightarrow d_n,\ \mathbf{unbound})$
  - $r_1[r_2] = (\lambda I\ .\ r_2\ I = \mathbf{unbound} \rightarrow r_1\ I,\ r_2\ I)$

Standard domains of values

- Unlike in TINY, in most languages we distinguish several value domains. The most important are:
  - *storable* values **Sv**—can be stored in locations; typical members will be v, v', v_1, v_2 etc.;
  - *denotable* values **Dv**—can be denoted by an identifier in the environment; typical members will be d, d', d_1, d_2 etc.;
  - *expressible* values **Ev**—results of expressions; typical members will be e, e', e_1, e_2 etc.
- Other domains can also be needed, e.g. *outputable* values, *R-values* (domain **Rv**) etc.

Declarations and scope

- *Declaration* binds an identifier to a certain location.
- *Scope* of a declaration are the parts of a code where the declaration holds. (It is also possible to speak about scope of an identifier.)
- Example: declaration **var I = E**. It's effect is:
  1. a new location, say i, is obtained;
  2. E's value is stored in i;
  3. i is bound to I in the environment.
- In standard semantics declarations change the environment and possibly the store. On the other side, command do only change the store.

Standard domains of continuations

<u>Domain of command continuations Cc</u>

- Definition: $Cc = Store \rightarrow Ans$
- Ans is a language-dependent domain of final answers
- Typical members will be c, c', c_1, c_2 etc.

<u>Domain of expression continuations Ec</u>

- Definition: $Ec = Ev \rightarrow Store \rightarrow Ans$ (or more neatly $Ec = Ev \rightarrow Cc$)
- Typical members will be k, k', k_1, k_2 etc.

<u>Domain of declaration continuations Dc</u>

- Def.: $Dc = Env \rightarrow Store \rightarrow Ans$ (or $Dc = Env \rightarrow Cc$)
- Typical members will be u, u', u_1, u_2 etc.

Standard semantic functions

- The following semantic functions are used:
  ***E*: Exp→Env→Ec→Store→Ans** for expressions,
  ***C*: Com→Env→Cc→Store→Ans** for commands, and
  ***D*: Dec→Env→Dc→Store→Ans** for declarations.
- The intuitive meanings are (omitting errors etc.):

E [E] r k s = k e s’	e is E’s value in environment r and store s, s’ is the store after E’s evaluation.
C [C] r c s = c s’	s’ is the store after C’s execution in environment r and store s.
D [D] r u s = u r’ s’	r’ consists of bindings specified in D, s’ results from D’s evaluation (with respect to r and s).

L and R values

- Consider l_1 and l_2 denoting locations i_1 and i_2 and the command $l_1 := l_2$. There are two possibilities:
  - location i_2 is stored in location i_1, or
  - the *contents* of location i_2 is stored in location i_1.
- The second case is the common one $\Rightarrow$ in standard semantics we assume that expressions on the right of assignments have their values *dereferenced*—i.e. have their values looked up in the store if they are locations.

L and R values

- The following terminology is used:
  - expression's *L-value* is a value needed on the left of an assignment—a location; it is obtained by ***E*** without any dereferencing.
  - expression's *R-value* is a value needed on the right of an assignment; it is (normally) obtained by dereferencing the value obtained by ***E***.
- It is traditional to define new semantic functions
  ***L*: Exp→Env→Ec→Cc** and
  ***R*: Exp→Env→Ec→Cc**
  for obtaining L-values and R-values, respectively.

Procedures

- **proc** I(I_1);C—declaration of procedure named I with formal parameter I_1 and body C.
- I(E)—call of the procedure I; C is executed in an environment identical to the one in which the procedure was declared, except that I_1 denotes the E's value.
- The above type of evaluation is called *static binding*. There are also other types of bindings, e.g. *dynamic binding* using the *call time environment*.
- E in I(E) is called the *actual parameter*.

Procedures

- Domain of procedure values is $\mathsf{Proc} = \mathsf{Cc} \to \mathsf{Ev} \to \mathsf{Store} \to \mathsf{Ans}$ (or $\mathsf{Proc} = \mathsf{Cc} \to \mathsf{Ec}$); typical members will be p, p', p_1, p_2 etc.
- Intuitively, if $p \in \mathsf{Proc}$ then:

  $p\ c\ e\ s = c\ s'$ where s' is the store resulting from execution of p's body.
- The procedure declaration binds a procedure value to an identifier as follows:

  $\boldsymbol{D}$ **[proc** $I(I_1);C$**]** $r\ u = u\ (p/I)$, where $p = \lambda ce\ .\ \boldsymbol{C}\ [C]\ r[e/I_1]\ c$
- The semantics of procedure call is:

  $\boldsymbol{C}\ [I(E)]\ r\ c\ s =$

  $\boldsymbol{E}\ [I]\ r\ (\lambda e_1 s_1.\mathrm{isProc}\ e_1 \to \boldsymbol{E}\ [E]\ r\ (\lambda e_2 s_2.e_1\ c\ e_2\ s_2)\ s_1,\ \textbf{error})\ s$

Functions

- Notice that function calls, unlike procedure calls, are expressions and also function bodies are expressions.
- Domain: $\text{Fun} = \text{Ec} \rightarrow \text{Ec}$
  Typical members will be f, f', f_1, f_2 etc.
- The semantics of function declaration is:

  $\boldsymbol{D}$ **[fun** $I(I_1);E$**]** $r\ u = u\ (f/I)$, where $f = \lambda ke\ .\ \boldsymbol{E}$ **[**E**]** $r[e/I_1]\ k$

- The semantics of function call is:

  $\boldsymbol{E}$ **[**$I(E)$**]** $r\ k\ s =$
  $\boldsymbol{E}$ **[**I**]** $r\ (\lambda e_1 s_1.\text{isFun}\ e_1 \rightarrow \boldsymbol{E}$ **[**E**]** $r\ (\lambda e_2 s_2.e_1\ k\ e_2\ s_2)\ s_1$, **error**$)\ s$

- <u>Notice</u> that Proc and Fun are denotable values, i.e. their members have to be in Dv.