

## Formal Semantics

Three approaches:

- **Operational semantics** reasons about executions of programs on abstract machines
- **Axiomatic Semantics** reasons about properties of programs: relate program to specification
- **Denotational semantics** reasons about meanings (denotations) of programs

Denotational semantics:

- **Compositional**
- **Map to meaning**
  - **Concrete Semantics** translates to a simpler language. Use meaning in simpler language.

1

## Two key questions

What mathematical objects represent meanings of programs?

- **Semantic domains**

How do we map a program to such an object?

- **Syntax-directed definitions** (compositional) (meaning of construct is function of meanings of parts)

2

## Denotational Semantics

**Semantic Domain** (Domain) A (structured) set of meanings.

**Syntactic Domain** Parts of abstract syntax that you would expect to have the same semantic domain. (Type)

**Valuation Function** Compositional map from syntactic domain to semantic domain.

**Example: Binary numerals**

**Syntax:**

$B ::= BD \mid D$

$D ::= 0 \mid 1$

**Syntactic Domains:**

$B \in \text{Binary-numeral}$

$D \in \text{Binary-digit}$

**Semantic Domains:**

$\{zero, one\}, \quad \mathbb{N} = \{zero, one, two, \dots\}$

**Valuation Functions:**

$\mathcal{D}[[0]] = zero \quad \mathcal{D}[[1]] = one$

$\mathcal{B}[[D]] = \mathcal{D}[[D]]$

$\mathcal{B}[[BD]] = ((\mathcal{B}[[B]] \text{ times two}) \text{ plus } \mathcal{D}[[D]])$

3

## Another Example: regular expressions

**Syntax:**

$R ::= A \mid \emptyset \mid R R \mid R|R \mid R^*$

$A ::= a \mid b \mid \dots$

**Syntactic Domains:**

$A \in \text{alphabet}$

$R \in \text{regular expressions}$

**Semantic Domains: Languages (= sets of strings)**

**Valuation Functions:**

$\mathcal{A}[[a]] = \{''a''\} \quad \mathcal{A}[[b]] = \{''b''\} \quad \dots$

$\mathcal{R}[[A]] = \mathcal{A}[[A]]$

$\mathcal{R}[[\emptyset]] = \{\}$

$\mathcal{R}[[R_1 R_2]] = \{concat(s_1, s_2) \mid s_1 \in \mathcal{R}[[R_1]], s_2 \in \mathcal{R}[[R_2]]\}$

$\mathcal{R}[[R_1 \mid R_2]] = \mathcal{R}[[R_1]] \cup \mathcal{R}[[R_2]]$

$\mathcal{R}[[R^*]] = \{''''\} \cup \mathcal{R}[[R R^*]]$

**Last is not compositional: uses whole of  $R^*$ .**

**Instead:**

**Let**  $L_0 = \{''''\}$  (the set of only the empty string)

$L_{n+1} = \{concat(s_1, s_2) \mid s_1 \in L_n, s_2 \in \mathcal{R}[[R]]\}$

**then**  $\mathcal{R}[[R^*]] = \bigcup_{\{0 \leq i\}} L_i$

**Infinite objects may need infinite description, or use fixed-points**

4

## Statement-based Languages

Semantic domains: Find mathematical objects to correspond to

- program
- statement
- expression
- number, boolean, ...

Use  $\lambda$ -calculus as convenient notation for valuation functions.

Can code environment, store, ... in  $\lambda$ -calculus: then get **Concrete semantics** as translation into  $\lambda$ -calculus.

- Very close to compiler (“syntax-directed translation”)
- Corresponds to **definitional interpreter**

In addition to other benefits, may **generate implementation from semantics!**

- **But not fast ones—yet**

5

## Semantic Domains

Every value space represented by a semantic **domain**

- Elementary domains: Int, Bool
- Pairs of domains
- Disjoint sums of domains
- Unions of domains
- Functions from domains to domains

Domain is set with special properties:

- **complete partial order**
- **has least element**

Motivation for least elements:

```
int f (int x) {
  if (x < 2)
    return x + 1;
  else
    while (1) {};
}
```

What is meaning of  $f(5)$  ?

6

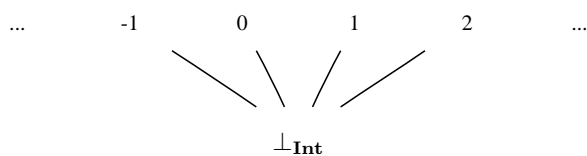
## Domain properties

Every domain has  $\perp$  or “undefined” element:

can represent non-terminating computation

Partial order “approximation”

$\perp_D \sqsubseteq x$  for every  $x$  in domain  $D$



Approximation means

- “is less defined than”
- “terminates less often than”

7

## Theoretical foundation — Partial orders

A partial order is a relation that is

- reflexive**  $x \sqsubseteq x$
- transitive** if  $x \sqsubseteq y, y \sqsubseteq z$  then  $x \sqsubseteq z$
- antisymmetric** if  $x \sqsubseteq y, y \sqsubseteq x$ , then  $x = y$

and there may be **incomparable**  $x$  and  $y$ : neither  $x \sqsubseteq y$  nor  $y \sqsubseteq x$ .

Many, many examples:

- **set inclusion on sets (subset relation)**
- “approximates” on partial functions (“refinement order”)
  - $f \sqsubseteq g$  if  $f$  and  $g$  agree where both are defined, and  $g$  is defined everywhere  $f$  is (set inclusion on domains)
- “is an instance of” relation on types

Notations abound:  $\leq, \preceq, \sqsubseteq, \sqsupseteq, \infty, \dots$

8

## Foundation: Complete partial order

A cpo is a partial order  $\sqsubseteq$  with:

- Ascending chain  $a_1 \sqsubseteq a_2 \sqsubseteq a_3 \dots$  has **least upper bound**  $\sqcup\{a_i\}$   
 “least” upper bound makes fewest assumptions

$Nat$  with usual order **is not** a cpo.

$Nat \cup \infty$  with usual order **is** a cpo.

A **domain** (pointed cpo) is a cpo with:

- Bottom element  $\perp \sqsubseteq x$  for all  $x$

$Int \cup \infty$  **is not** a domain.

$Nat \cup \infty$  **is** a domain.

9

## Function domains

Given cpo  $S$ , what is domain of functions  $S \rightarrow S$ ?

Make all functions total

**map to  $\perp$  where not otherwise defined**

Define  $\sqsubseteq'$  on functions from  $S$  to  $S$  by

$$f \sqsubseteq' g \text{ iff } f(x) \sqsubseteq g(x) \forall x \in S$$

Well-behaved function is:

- **monotonic:**  $\forall x, y, x \sqsubseteq y$  implies  $f(x) \sqsubseteq f(y)$
- **continuous:**  $f(\sqcup\{a_i\}) = \sqcup\{f(a_i)\}$
- Note: continuous implies monotonic.

(A function  $f$  is **strict** iff  $f(\perp) = \perp$ )

Let  $A \rightarrow B$  be the set of **total, continuous** functions from  $A$  to  $B$ .

**(Thm: This set is also a cpo)**

11

## Recursive domains

Cpos provide a model for functions

Scheme: Functions as values:

*Value* must include  $Value \rightarrow Value$

What is  $Value \rightarrow Value$ ?

Can't be functions from *Value* to *Value*

**(counting argument shows too many!)**

Must cut down the number of functions somehow

choose those that are **not “over-defined”**

Technical analysis, but intuitive result!

10

## Fixed points

**A continuous function has a fixed point.**

- $\perp \sqsubseteq f(\perp)$  because  $\perp \sqsubseteq x$  for all  $x$
- By previous, monotonicity,  $f(\perp) \sqsubseteq f(f(\perp))$   
(Apply  $f$  to both sides)
- $\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq f(f(f(\perp))) \sqsubseteq \dots$   
is an ascending chain.
- **Notation:**  $f^{(0)}(\perp) \equiv \perp$ ,  $f^{(1)}(\perp) \equiv f(\perp)$ ,  
 $f^{(2)}(\perp) \equiv f(f(\perp))$ , ...
- Then  $F = \sqcup\{f^{(i)}(\perp)\}$  is the **least fixed point of  $f$**   
(exploit continuity):  
 $f(F) = f(\sqcup\{f^{(i)}(\perp)\}) = \sqcup\{f^{(i+1)}(\perp)\}$   
 $= \sqcup(\perp \cup \{f^{(i+1)}(\perp)\}) = \sqcup\{f^{(i)}(\perp)\} = F$

**Implication:** define functions in  $A \rightarrow B$  inductively.

**Tarski's fixed point theorem:**

Let  $f : D \rightarrow D$  be a continuous function on a domain  $D$ . Then  $f$  has a least fixed point  $fix(f)$ , which can be calculated by

$$fix(f) = \bigsqcup_{i \geq 0} f^{(i)}(\perp)$$

12

Let  $u$  be the function mapping everything to  $\perp$

Consider

$$\begin{aligned}
 g &= \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times f(n-1) \\
 u &= \{\perp \mapsto \perp, 0 \mapsto \perp, 1 \mapsto \perp, 2 \mapsto \perp, 3 \mapsto \perp, \dots\} \\
 g(u) &= \{\perp \mapsto \perp, 0 \mapsto 1, 1 \mapsto \perp, 2 \mapsto \perp, 3 \mapsto \perp, \dots\} \\
 g(g(u)) &= \{\perp \mapsto \perp, 0 \mapsto 1, 1 \mapsto 1, 2 \mapsto \perp, 3 \mapsto \perp, \dots\} \\
 g(g(g(u))) &= \{\perp \mapsto \perp, 0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto \perp, \dots\} \\
 g(g(g(g(u)))) &= \{\perp \mapsto \perp, 0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6, \dots\}
 \end{aligned}$$

Alternate notation as partial function:

$$\begin{aligned}
 u &= \{\} \text{ (nowhere defined, maps all to } \perp) \\
 g(u) &= \{0 \mapsto 1\} \\
 g(g(u)) &= \{0 \mapsto 1, 1 \mapsto 1\} \\
 g(g(g(u))) &= \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2\} \\
 g(g(g(g(u)))) &= \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6\}
 \end{aligned}$$

Note that

$$u \sqsubseteq g(u) \sqsubseteq g(g(u)) \sqsubseteq \dots \text{ indefinitely}$$

We can define the factorial function by

$$\text{fact} = \sqcup \{g^{(n)}(u)\}$$

13

Many useful domains:

- Basic domains
- Sums and products
- Function domains

Define domain for  $\mu$ Scheme using recursive domain equations:

$$D = \text{Int} + \text{Bool} + \text{Sym} + D \times D + (D \rightarrow D)$$

Solve for  $D$  using fixed-point techniques

$D$  as ML datatype:

```
datatype D = I of int | B of bool
           | S of string | P of D*D | F of D->D
```

14

## Semantics of imperative programs

Essential Features:

- Sequential execution
- Implicit data structure **the store**
  - Existence independent of any program
  - Not mentioned in language syntax
  - Program phrases may access and update it

Relationship between store and programs:

- Critical for evaluation of phrases: meaning depends on stored values
- Communication between phrases: Phrases set values in store for use by other phrases; sequencing mechanism establishes communication order
- Only one copy exists during execution.

Semantics of statements as functions from store to store

15

16

## Part II

### Mapping programs to domains

## Straight-line programs

### Types:

$$\begin{aligned}
 n : V &= Int \\
 i, j : Ide \\
 \sigma : S &= Ide \rightarrow V \\
 Exp &= S \rightarrow V \\
 Stm &= S \rightarrow S
 \end{aligned}$$

### Functions:

$$\begin{aligned}
 plus &: (V \times V) \rightarrow V \\
 update &: (S \times Ide \times V) \rightarrow S \\
 I &: ID \rightarrow Ide \\
 \mathcal{N} &: NUM \rightarrow V \\
 \mathcal{E} &: exp \rightarrow Exp \\
 S &: stm \rightarrow Stm
 \end{aligned}$$

$$\begin{aligned}
 \sigma_0(i) &= 0 \\
 update(\sigma, i, n) &= \lambda j. \text{if } i = j \text{ then } n \text{ else } \sigma(j)
 \end{aligned}$$

17

### Grammar rules and semantic equations:

$$\begin{aligned}
 stm \Rightarrow ID := exp & \quad S[[stm]]\sigma = update(\sigma, I[[ID]], \mathcal{E}[[exp]]\sigma) \\
 stm \Rightarrow stm_1 ; stm_2 & \quad S[[stm]]\sigma = S[[stm_2]](S[[stm_1]]\sigma) \\
 exp \Rightarrow NUM & \quad \mathcal{E}[[exp]]\sigma = \mathcal{N}[[NUM]] \\
 exp \Rightarrow ID & \quad \mathcal{E}[[exp]]\sigma = \sigma(I[[ID]]) \\
 exp \Rightarrow exp_1 + exp_2 & \quad \mathcal{E}[[exp]]\sigma = plus(\mathcal{E}[[exp_1]]\sigma, \mathcal{E}[[exp_2]]\sigma)
 \end{aligned}$$

### Language features:

- Executes statements in order, no control flow.
- Identifiers not declared before use.
- No order of evaluation in expressions.
- Store is a function threaded through the definition. Store is never duplicated.

18

## Straight-line Programs Example

$$\begin{aligned}
 & S[[x := 1; x := x+x]]\sigma_0 \\
 \equiv & S[[x := x+x]](S[[x := 1]]\sigma_0) \\
 \equiv & S[[x := x+x]](update(\sigma_0, I[[x]], \mathcal{E}[[1]]\sigma_0)) \\
 \equiv & S[[x := x+x]](update(\sigma_0, l_x, one)) \\
 \equiv & \\
 & S[[x := x+x]]\sigma_1 \text{ where } \begin{cases} \sigma_1(l_x) = one \\ \sigma_1(l_y) = \sigma_0(l_y) \text{ for } y \neq x \end{cases} \\
 \equiv & update(\sigma_1, I[[x]], \mathcal{E}[[x+x]]\sigma_1) \\
 \equiv & update(\sigma_1, l_x, plus(\mathcal{E}[[x]]\sigma_1, \mathcal{E}[[x]]\sigma_1)) \\
 \equiv & update(\sigma_1, l_x, plus(\sigma_1(I[[x]]), \sigma_1(I[[x]]))) \\
 \equiv & update(\sigma_1, l_x, plus(\sigma_1(l_x), \sigma_1(l_x))) \\
 \equiv & update(\sigma_1, l_x, plus(one, one)) \\
 \equiv & update(\sigma_1, l_x, two) \\
 \equiv & \sigma_2 \text{ where } \begin{cases} \sigma_2(l_x) = two \\ \sigma_2(l_y) = \sigma_1(l_y) = \sigma_0(l_y) \text{ for } y \neq x \end{cases}
 \end{aligned}$$

1940's style machine: look in memory for value.

Note: semantics had not specified domain for *Ide*.  
 Using  $I[[i]] = l_i$ . We are spelling out elements of semantic domain *Int* to distinguish them from elements of syntactic domain *Num*.

19

## Semantics in ML

```

(* Syntax: forget it, someone else can write parser *)

(* Syntactic domains *)
type NUM = int
type ID = string
datatype exp = V1 of NUM
              | Id of ID
              | op + of (exp * exp)
datatype stm = Assign of (ID * exp)
              | Seq of (stm * stm)

(* Semantic domains *)
type V = int
type Ide = string
type S = Ide -> V
type Exp = S -> V
type Stm = S -> S

(* Store functions *)
fun s0(_) = 0
fun update (s : S , i, n) =
    fn j : Ide => if i = j then n else s(j)

(* expression functions *)
val plus : (V * V) -> V = fn (n, m) => Int.+ (n, m)

(* Valuation Functions *)
val Iv : ID -> Ide = fn (x) => x
    
```

20

```

val Nv : NUM -> V = fn (x) => x

val rec Ev : exp -> Exp =
  fn (Vl n)      => (fn s => Nv n)
  | (Id x)       => (fn s => s (Iv x))
  | (e1 + e2) =>
    (fn s => plus ((Ev e1) s, (Ev e2) s))

val rec Sv : stm -> Stm =
  fn (Assign (i, e)) =>
    (fn s => update (s, Iv i, Ev e s))
  | (Seq (stm1, stm2)) =>
    (fn s => Sv stm2 (Sv stm1 s))
;
(* Example code:
  X := 1 ;
  Y := 2 ;
  X := X + Y
and find value of X *)

Sv (Seq (Assign ("X", Vl 1),
        Seq (Assign ("Y", Vl 2),
            Assign ("X", ((Id "X") + (Id "Y"))))))
s0 "X"
(*
val it = 3 : V
*)

```

Types:

- $V = Int$
- $Ide = Identifiers$
- $l : L = Locations$
- $\rho : Env = Ide \rightarrow L$
- $\sigma : S = L \rightarrow V$
- $Exp = Env \rightarrow S \rightarrow V$
- $Stm = Env \rightarrow L \rightarrow S \rightarrow S$

New idea: mutation (assignment)

New idea: block structure

Program returns a value in variable Answer

But watch out: ML is call-by-value, semantics by substitution (call-by-name).

Functions:

- plus :  $(V \times V) \rightarrow V$
- upds :  $(S \times L \times V) \rightarrow S$
- upde :  $(Env \times Ide \times L) \rightarrow Env$
- $l_0 : L$
- next :  $L \rightarrow L$
- $I : ID \rightarrow Ide$
- $\mathcal{N} : NUM \rightarrow V$
- $\mathcal{E} : exp \rightarrow Exp$
- $S : stm \rightarrow Stm$
- $\mathcal{P} : prog \rightarrow V$

- next( $l_i$ ) =  $l_{i+1}$
- $\sigma_0(l) = 0$
- upds( $\sigma, l, n$ ) =  $\lambda l'. \text{if } l = l' \text{ then } n \text{ else } \sigma(l')$
- $\rho_0(i) = l_0$
- upde( $\rho, i, l$ ) =  $\lambda j. \text{if } i = j \text{ then } l \text{ else } \rho(j)$

Grammar rules and semantic equations:

- prog  $\Rightarrow$  stm  $\mathcal{P} \llbracket prog \rrbracket = S \llbracket stm_1 \rrbracket \rho_1 (next\ l_0) \sigma_0 l_0$
- stm  $\Rightarrow$  ID := exp  $S \llbracket stm \rrbracket \rho l \sigma = upds(\sigma, \rho(I \llbracket ID \rrbracket), \mathcal{E} \llbracket exp_1 \rrbracket) \rho \sigma$
- stm  $\Rightarrow$  { VAR ID; stm }  $S \llbracket stm \rrbracket \rho l \sigma = S \llbracket stm_1 \rrbracket (upde(\rho, I \llbracket ID \rrbracket, l)) (next\ l) \sigma$
- stm  $\Rightarrow$  stm; stm  $S \llbracket stm \rrbracket \rho l \sigma = S \llbracket stm_2 \rrbracket \rho l (S \llbracket stm_1 \rrbracket \rho l \sigma)$
- exp  $\Rightarrow$  NUM  $\mathcal{E} \llbracket exp \rrbracket \rho \sigma = \mathcal{N} \llbracket NUM \rrbracket$
- exp  $\Rightarrow$  ID  $\mathcal{E} \llbracket exp \rrbracket \rho \sigma = \sigma(\rho(I \llbracket ID \rrbracket))$
- exp  $\Rightarrow$  exp + exp  $\mathcal{E} \llbracket exp \rrbracket \rho \sigma = plus(\mathcal{E} \llbracket exp_1 \rrbracket \rho \sigma, \mathcal{E} \llbracket exp_2 \rrbracket \rho \sigma)$

Program returns the value of the special variable Answer which is in the store at location  $l_0$

$\rho_1 : Env = upde(\rho_0, I \llbracket Answer \rrbracket, l_0)$

One location for the answer, one location for each level of block nesting

## Left and Right Values

### Types:

$$\begin{aligned}
 V &= \text{Int} \\
 \text{Ide} &= \text{Identifiers} \\
 l : L &= \text{Locations} \\
 \rho : \text{Env} &= \text{Ide} \rightarrow L \\
 \sigma : S &= L \rightarrow V \\
 \text{RExp} &= \text{Env} \rightarrow S \rightarrow V \\
 \text{LExp} &= \text{Env} \rightarrow S \rightarrow L \\
 \text{Stm} &= \text{Env} \rightarrow L \rightarrow S \rightarrow S
 \end{aligned}$$

### New idea: if

### New idea: expressions for locations

((if p then x else y) := 0)

(Icon has this feature, Common Lisp has subset of feature)

### Functions:

$$\begin{aligned}
 \text{plus} &: (V \times V) \rightarrow V \\
 \text{upds} &: (S \times L \times V) \rightarrow S \\
 \text{upde} &: (\text{Env} \times \text{Ide} \times L) \rightarrow \text{Env} \\
 l_0 &: L \\
 \text{next} &: L \rightarrow L \\
 \sigma_0 &: S \\
 \rho_0 &: \text{Env} \\
 \rho_1 &: \text{Env} = \text{upde}(\rho_0, l(\text{Answer}), l_0) \\
 \text{ifnz}_r &: (V \times V \times V) \rightarrow V \\
 \text{ifnz}_l &: (V \times L \times L) \rightarrow L \\
 I &: \text{ID} \rightarrow \text{Ide} \\
 \mathcal{N} &: \text{NUM} \rightarrow V \\
 \mathcal{R} &: \text{exp} \rightarrow \text{RExp} \\
 \mathcal{L} &: \text{exp} \rightarrow \text{LExp} \cup \{\text{wrong}\} \\
 S &: \text{stm} \rightarrow \text{Stm} \\
 \mathcal{P} &: \text{prog} \rightarrow V
 \end{aligned}$$

25

26

### Grammar rules and semantic equations:

$$\begin{aligned}
 \text{prog} \Rightarrow \text{stm} & \quad \mathcal{P}[\llbracket \text{prog}_0 \rrbracket] = S[\llbracket \text{stm}_1 \rrbracket] \rho_1 (\text{next } l_0) \sigma_0 l_0 \\
 \\
 \text{stm} \Rightarrow \text{exp} := \text{exp} & \quad S[\llbracket \text{stm}_0 \rrbracket] \rho l \sigma = \text{upds}(\sigma, \mathcal{L}[\llbracket \text{exp}_1 \rrbracket] \rho \sigma, \mathcal{R}[\llbracket \text{exp}_2 \rrbracket] \rho \sigma) \\
 \text{stm} \Rightarrow \{ \text{VAR ID} ; \text{stm} \} & \quad S[\llbracket \text{stm}_0 \rrbracket] \rho l \sigma = S[\llbracket \text{stm}_1 \rrbracket] (\text{upde}(\rho, l[\llbracket \text{ID} \rrbracket], \text{next } l)) (\text{next } l) \sigma \\
 \text{stm} \Rightarrow \text{stm} ; \text{stm} & \quad S[\llbracket \text{stm}_0 \rrbracket] \rho l \sigma = S[\llbracket \text{stm}_2 \rrbracket] \rho l (S[\llbracket \text{stm}_1 \rrbracket] \rho l \sigma) \\
 \\
 \text{exp} \Rightarrow \text{NUM} & \quad \mathcal{R}[\llbracket \text{exp}_0 \rrbracket] \rho \sigma = \mathcal{N}[\llbracket \text{NUM} \rrbracket] \\
 \text{exp} \Rightarrow \text{ID} & \quad \mathcal{R}[\llbracket \text{exp}_0 \rrbracket] \rho \sigma = \sigma(\rho(l[\llbracket \text{ID} \rrbracket])) \\
 \text{exp} \Rightarrow \text{exp} + \text{exp} & \quad \mathcal{R}[\llbracket \text{exp}_0 \rrbracket] \rho \sigma = \text{plus}(\mathcal{R}[\llbracket \text{exp}_1 \rrbracket] \rho \sigma, \mathcal{R}[\llbracket \text{exp}_2 \rrbracket] \rho \sigma) \\
 \text{exp} \Rightarrow \text{IF exp}_1 \text{ THEN exp}_2 \text{ ELSE exp}_3 & \quad \mathcal{R}[\llbracket \text{exp}_0 \rrbracket] \rho \sigma = \text{ifnz}_r(\mathcal{R}[\llbracket \text{exp}_1 \rrbracket] \rho \sigma, \mathcal{R}[\llbracket \text{exp}_2 \rrbracket] \rho \sigma, \mathcal{R}[\llbracket \text{exp}_3 \rrbracket] \rho \sigma) \\
 \\
 \text{exp} \Rightarrow \text{NUM} & \quad \mathcal{L}[\llbracket \text{exp}_0 \rrbracket] \rho \sigma = \text{wrong} \\
 \text{exp} \Rightarrow \text{ID} & \quad \mathcal{L}[\llbracket \text{exp}_0 \rrbracket] \rho \sigma = \rho(l[\llbracket \text{ID} \rrbracket]) \\
 \text{exp} \Rightarrow \text{exp} + \text{exp} & \quad \mathcal{L}[\llbracket \text{exp}_0 \rrbracket] \rho \sigma = \text{wrong} \\
 \text{exp} \Rightarrow \text{IF exp}_1 \text{ THEN exp}_2 \text{ ELSE exp}_3 & \quad \mathcal{L}[\llbracket \text{exp}_0 \rrbracket] \rho \sigma = \text{ifnz}_l(\mathcal{R}[\llbracket \text{exp}_1 \rrbracket] \rho \sigma, \mathcal{L}[\llbracket \text{exp}_2 \rrbracket] \rho \sigma, \mathcal{L}[\llbracket \text{exp}_3 \rrbracket] \rho \sigma)
 \end{aligned}$$

27

## Control Flow: while loops

### Types:

$$\begin{aligned}
 V &= \text{Int} \\
 \text{Ide} & \\
 \sigma : S &= \text{Ide} \rightarrow V \\
 \text{Exp} &= S \rightarrow V \\
 \text{Stm} &= S \rightarrow S
 \end{aligned}$$

### Functions:

$$\begin{aligned}
 \text{plus} &: (V \times V) \rightarrow V \\
 \text{update} &: (S \times \text{Ide} \times V) \rightarrow S \\
 \text{ifnz} &: (V \times S \times S) \rightarrow S \\
 \text{fix} &: (S \rightarrow S) \rightarrow S \\
 I &: \text{ID} \rightarrow \text{Ide} \\
 \mathcal{N} &: \text{NUM} \rightarrow V \\
 \mathcal{E} &: \text{exp} \rightarrow \text{Exp} \\
 S &: \text{stm} \rightarrow \text{Stm}
 \end{aligned}$$

28

## Control Flow: while loops, try 1

...

$stm \Rightarrow \text{IF } exp \text{ THEN } stm \text{ ELSE } stm$   
 $S[[stm_0]]\sigma = \text{ifnz}(\mathcal{E}[[exp_1]]\sigma, S[[stm_1]]\sigma, S[[stm_2]]\sigma)$

$stm \Rightarrow \text{skip}$   $S[[stm_0]]\sigma = \sigma$

$stm \Rightarrow \text{WHILE } exp \text{ DO } stm$   
 $S[[\text{IF } exp \text{ THEN } stm; \text{WHILE } exp \text{ DO } stm \text{ ELSE } skip]]$

**Problem: denotational semantics must be compositional**

**the trick above works for operational semantics**

Like Kleene\* in semantics of regular expressions, we need to use a fixed point to define semantics of while.

## Grammar rules and semantic equations:

$stm \Rightarrow \text{ID} := exp$   $S[[stm_0]]\sigma = \text{update}(\sigma, I[[ID]], \mathcal{E}[[exp_1]]\sigma)$

$stm \Rightarrow stm ; stm$   $S[[stm_0]]\sigma = S[[stm_2]](S[[stm_1]]\sigma)$

$stm \Rightarrow \text{IF } exp \text{ THEN } stm \text{ ELSE } stm$   
 $S[[stm_0]]\sigma = \text{ifnz}(\mathcal{E}[[exp_1]]\sigma, S[[stm_1]]\sigma, S[[stm_2]]\sigma)$

$stm \Rightarrow \text{WHILE } exp \text{ DO } stm$   
 $S[[stm_0]] = \text{fix}(\lambda f. \lambda \sigma. \text{ifnz}(\mathcal{E}[[exp_1]]\sigma, f(S[[stm_1]]\sigma), \sigma))$

$stm \Rightarrow \text{skip}$   $S[[stm_0]]\sigma = \sigma$

$exp \Rightarrow \text{NUM}$   $\mathcal{E}[[exp_0]]\sigma = \mathcal{N}[[NUM]]$

$exp \Rightarrow \text{ID}$   $\mathcal{E}[[exp_0]]\sigma = \sigma(I[[ID]])$

$exp \Rightarrow exp + exp$   $\mathcal{E}[[exp_0]]\sigma = \text{plus}(\mathcal{E}[[exp_1]]\sigma, \mathcal{E}[[exp_2]]\sigma)$

**New: while, skip**

Have dropped nested block structure for simplicity.

Have dropped factoring of environment and store for simplicity.

29

30

## Statement Continuations

Types:

$V = \text{Int}$   
 $Ide = \text{Identifiers}$   
 $\sigma : S = Ide \rightarrow V$   
 $A = \text{Answers}$   
 $\theta : C = S \rightarrow A$   
 $Exp = S \rightarrow V$   
 $Stm = C \rightarrow C$   
 $Prog = A$

New ideas:

- **answers**
- **statement continuations** (what to do next)  
start to be able to define goto, raise, ...
- **simple application here: allowed to stop computation**  
 $Answers = \{\text{finished}(v), \text{stopped} \mid v \in V\}$

31

## Statement Continuations

In direct style had  $Stm = S \rightarrow S$   
Statement may change contents of store.

With statement continuations have  
 $Stm = C \rightarrow C = (S \rightarrow A) \rightarrow S \rightarrow A$

What is this?

The meaning of a Program is an Answer. A statement takes “what more do we need to do to the store to get the answer” and returns the rest of the work needed to get from the store to the answer after this statement has been executed.

Had:

$stm \Rightarrow stm ; stm$   
 $S[[stm_0]]\sigma = S[[stm_2]](S[[stm_1]]\sigma)$   
 $stm_2$  with new store

Now:

$stm \Rightarrow stm ; stm$   
 $S[[stm_0]]\theta = S[[stm_1]](S[[stm_2]]\theta)$   
 $stm_1$  with fn taking modified store

32

## Functions and Values:

$\text{plus} : (V \times V) \rightarrow V$   
 $\text{upds} : (S \times \text{Ide} \times V) \rightarrow S$   
 $\text{ifnz} : (V \times A \times A) \rightarrow S$   
 $\text{done} : C$   
 $\sigma_0 : S$   
 $I : \text{ID} \rightarrow \text{Ide}$   
 $\mathcal{N} : \text{NUM} \rightarrow V$   
 $\mathcal{E} : \text{exp} \rightarrow \text{Exp}$   
 $S : \text{stm} \rightarrow \text{Stm}$   
 $\mathcal{P} : \text{prog} \rightarrow \text{Prog}$

$\text{done}(\sigma) = \text{finished}(\sigma(I[\text{Answer}]))$

33

## Statement Continuations Example

$\mathcal{P}[\text{Answer} := 2; \text{Answer} := \text{Answer}+1]$   
 $\equiv S[\text{Answer} := 2; \text{Answer} := \text{Answer}+1] \text{ done } \sigma_0$   
 $\equiv (S[\text{Answer} := 2])(S[\text{Answer} := \text{Answer}+1] \text{ done}) \sigma_0$   
 $\equiv (\lambda\sigma. (S[\text{Answer} := \text{Answer}+1] \text{ done})$   
 $\quad (\text{upds}(\sigma, I[\text{Answer}], \mathcal{E}[2]\sigma)) \sigma_0$   
 $\equiv (S[\text{Answer} := \text{Answer}+1] \text{ done})$   
 $\quad (\text{upds}(\sigma_0, I[\text{Answer}], \mathcal{E}[2]\sigma_0))$   
 $\equiv (S[\text{Answer} := \text{Answer}+1] \text{ done})(\text{upds}(\sigma_0, \iota_{\text{Answer}}, \text{two}))$   
 $\equiv (S[\text{Answer} := \text{Answer}+1] \text{ done}) \sigma_1$   
 $\equiv (\lambda\sigma. \text{done}(\text{upds}(\sigma, I[\text{Answer}], \mathcal{E}[\text{Answer}+1]\sigma))) \sigma_1$   
 $\equiv \text{done}(\text{upds}(\sigma_1, I[\text{Answer}], \mathcal{E}[\text{Answer}+1]\sigma_1))$   
 $\dots$   
 $\equiv \text{done}(\text{upds}(\sigma_1, \iota_{\text{Answer}}, \text{three}))$   
 $\equiv \text{done } \sigma_2$   
 $\equiv \text{finished}(\sigma_2(I[\text{Answer}]))$   
 $\equiv \text{finished}(\sigma_2(\iota_{\text{Answer}}))$   
 $\equiv \text{finished}(\text{three})$

where

$\sigma_1 \equiv \text{upds}(\sigma_0, \iota_{\text{Answer}}, \text{two})$   
 $\sigma_2 \equiv \text{upds}(\sigma_1, \iota_{\text{Answer}}, \text{three})$

35

## Grammar rules and semantic equations:

$\text{prog} \Rightarrow \text{stm}$   
 $\mathcal{P}[\text{prog}_0] = S[\text{stm}_1] \text{ done } \sigma_0$   
 $\text{stm} \Rightarrow \text{ID} := \text{exp}$   
 $S[\text{stm}_0]\theta = \lambda\sigma. \theta(\text{upds}(\sigma, I[\text{ID}], \mathcal{E}[\text{exp}_1]\sigma))$   
 $\text{stm} \Rightarrow \text{stm} ; \text{stm}$   
 $S[\text{stm}_0]\theta = S[\text{stm}_1](S[\text{stm}_2]\theta)$   
 $\text{stm} \Rightarrow \text{IF exp THEN stm ELSE stm}$   
 $S[\text{stm}_0]\theta = \lambda\sigma. \text{ifnz}(\mathcal{E}[\text{exp}_1]\sigma, S[\text{stm}_1]\theta\sigma, S[\text{stm}_2]\theta\sigma)$   
 $\text{stm} \Rightarrow \text{STOP}$   
 $S[\text{stm}_0]\theta = \lambda\sigma. \text{stopped}$   
 $\text{exp} \Rightarrow \text{NUM} \quad \mathcal{E}[\text{exp}_0]\sigma = \mathcal{N}[\text{NUM}]$   
 $\text{exp} \Rightarrow \text{ID} \quad \mathcal{E}[\text{exp}_0]\sigma = \sigma(I[\text{ID}])$   
 $\text{exp} \Rightarrow \text{exp} + \text{exp} \quad \mathcal{E}[\text{exp}_0]\sigma = \text{plus}(\mathcal{E}[\text{exp}_1]\sigma, \mathcal{E}[\text{exp}_2]\sigma)$

$\text{done}(\sigma) = \text{finished}(\sigma(I[\text{Answer}]))$

## Have dropped while for simplicity.

34

## Expression Continuations

Types:

$V = \text{Int}$   
 $\text{Ide} = \text{Identifiers}$   
 $\sigma : S = \text{Ide} \rightarrow V$   
 $A = \text{Answers}$   
 $\theta : C = S \rightarrow A$   
 $\kappa : K = V \rightarrow C$   
 $\text{Exp} = K \rightarrow C$   
 $\text{Stm} = C \rightarrow C$   
 $\text{Prog} = A$

New idea: expression appears in a **context**, which **expects a value with which to continue computing**.

Expression continuations enforce evaluation order of expressions.

36

## Functions:

$\text{plus} : (V \times V) \rightarrow V$   
 $\text{upd} : (S \times \text{Ide} \times V) \rightarrow S$   
 $\text{ifnz} : (V \times A \times A) \rightarrow S$   
 $\text{done} : V \rightarrow A$   
 $\text{error} : A$   
 $\sigma_0 : S$   
 $I : \text{ID} \rightarrow \text{Ide}$   
 $\mathcal{N} : \text{NUM} \rightarrow V$   
 $\mathcal{E} : \text{exp} \rightarrow \text{Exp}$   
 $\mathcal{S} : \text{stm} \rightarrow \text{Stm}$   
 $\mathcal{P} : \text{prog} \rightarrow \text{Prog}$

## Grammar rules and semantic equations:

$\text{prog} \Rightarrow \text{stm} \quad \mathcal{P} \llbracket \text{prog}_0 \rrbracket = \mathcal{S} \llbracket \text{stm}_1 \rrbracket (\lambda \sigma. \text{error}) \sigma_0$   
 $\text{stm} \Rightarrow \text{ID} := \text{exp} \quad \mathcal{S} \llbracket \text{stm}_0 \rrbracket \theta = \mathcal{E} \llbracket \text{exp}_1 \rrbracket (\lambda v. \lambda \sigma. \theta(\text{upd}(\sigma, I \llbracket \text{ID} \rrbracket, v)))$   
 $\text{stm} \Rightarrow \text{RETURN exp} \quad \mathcal{S} \llbracket \text{stm}_0 \rrbracket \theta = \mathcal{E} \llbracket \text{exp}_1 \rrbracket (\lambda v. \lambda \sigma. \text{done}(v))$   
 $\text{stm} \Rightarrow \text{stm} ; \text{stm} \quad \mathcal{S} \llbracket \text{stm}_0 \rrbracket \theta = \mathcal{S} \llbracket \text{stm}_1 \rrbracket (\mathcal{S} \llbracket \text{stm}_2 \rrbracket \theta)$   
 $\text{exp} \Rightarrow (\text{stm} ; \text{exp}) \quad \mathcal{E} \llbracket \text{exp}_0 \rrbracket \kappa = \mathcal{S} \llbracket \text{stm}_1 \rrbracket (\mathcal{E} \llbracket \text{exp}_1 \rrbracket \kappa)$   
 $\text{exp} \Rightarrow \text{NUM} \quad \mathcal{E} \llbracket \text{exp}_0 \rrbracket \kappa = \kappa(\mathcal{N} \llbracket \text{NUM} \rrbracket)$   
 $\text{exp} \Rightarrow \text{ID} \quad \mathcal{E} \llbracket \text{exp}_0 \rrbracket \kappa = \lambda \sigma. \kappa(\sigma(I \llbracket \text{ID} \rrbracket)) \sigma$   
 $\text{exp} \Rightarrow \text{exp} + \text{exp} \quad \mathcal{E} \llbracket \text{exp}_0 \rrbracket \kappa = \mathcal{E} \llbracket \text{exp}_1 \rrbracket (\lambda v_1. \mathcal{E} \llbracket \text{exp}_2 \rrbracket (\lambda v_2. \kappa(\text{plus}(v_1, v_2))))$

Program meaning will be **error** unless **RETURN** statement is used.

Note order of evaluation in **+** is specified (in call-by-value or call-by-name). A good idea if you are going to allow statements in expressions.

37

38

## Expression Continuations Example

$\mathcal{E} \llbracket 1+2 \rrbracket \kappa$   
 $\equiv \mathcal{E} \llbracket 1 \rrbracket (\lambda v_1. \mathcal{E} \llbracket 2 \rrbracket (\lambda v_2. \kappa(\text{plus } v_1 \ v_2)))$   
 $\equiv (\lambda v_1. \mathcal{E} \llbracket 2 \rrbracket (\lambda v_2. \kappa(\text{plus } v_1 \ v_2))) \mathcal{N} \llbracket 1 \rrbracket$   
 $\equiv (\lambda v_1. \mathcal{E} \llbracket 2 \rrbracket (\lambda v_2. \kappa(\text{plus } v_1 \ v_2))) \text{one}$   
 $\equiv \mathcal{E} \llbracket 2 \rrbracket (\lambda v_2. \kappa(\text{plus one } v_2))$   
 $\equiv (\lambda v_2. \kappa(\text{plus one } v_2)) \mathcal{N} \llbracket 2 \rrbracket$   
 $\equiv \kappa(\text{plus one two})$   
 $\equiv \kappa \text{three}$

$\mathcal{S} \llbracket \text{RETURN } 1+2 \rrbracket \theta$   
 $\equiv \mathcal{E} \llbracket 1+2 \rrbracket (\lambda v. \lambda \sigma. \text{done}(v))$   
 $\equiv (\lambda v. \lambda \sigma. \text{done}(v)) \text{three}$   
 $\equiv (\lambda \sigma. \text{done}(\text{three}))$

$\mathcal{P} \llbracket \text{RETURN } 1+2 \rrbracket (\lambda \sigma. \text{error}) \sigma_0$   
 $\equiv \mathcal{S} \llbracket \text{RETURN } 1+2 \rrbracket (\lambda \sigma. \text{error}) \sigma_0$   
 $\equiv (\lambda \sigma. \text{done}(\text{three})) \sigma_0$   
 $\equiv \text{done}(\text{three})$

## raise and handle

$V = \text{Int}$   
 $\text{Ide} = \text{Identifiers}$   
 $S = \text{Ide} \rightarrow V$   
 $A = \text{Answers}$   
 $C = S \rightarrow A$   
 $K = V \rightarrow C$   
 $\text{Exp} = K \rightarrow K \rightarrow C$   
 $\text{Stm} = C \rightarrow K \rightarrow C$   
 $\text{Prog} = A$

New idea: each statement & expression takes **two** continuations:

- **normal termination** of evaluating an expression.
- **terminate** evaluating an expression **by raising** an exception

39

40

## Functions:

plus :	$(V \times V) \rightarrow V$
upd :	$(S \times Ide \times V) \rightarrow S$
ifnz :	$(V \times A \times A) \rightarrow S$
done :	$V \rightarrow A$
error :	$A$
$\sigma_0$ :	$S$
$I$ :	$ID \rightarrow Ide$
$\mathcal{N}$ :	$NUM \rightarrow V$
$\mathcal{E}$ :	$exp \rightarrow Exp$
$S$ :	$stm \rightarrow Stm$
$\mathcal{P}$ :	$prog \rightarrow Prog$

## Grammar rules and semantic equations:

prog $\Rightarrow$ stm	$\mathcal{P} \llbracket prog_0 \rrbracket = S \llbracket stm_1 \rrbracket (\lambda \sigma. error) (\lambda v. \lambda \sigma. error) \sigma_0$
stm $\Rightarrow$ ID := exp	$S \llbracket stm_0 \rrbracket \theta \kappa_h = \mathcal{E} \llbracket exp_1 \rrbracket (\lambda v. \lambda \sigma. \theta(\text{upd}(\sigma, I \llbracket ID \rrbracket, v))) \kappa_h$
stm $\Rightarrow$ RETURN exp	$S \llbracket stm_0 \rrbracket \theta \kappa_h = \mathcal{E} \llbracket exp_1 \rrbracket (\lambda v. \lambda \sigma. done(v)) \kappa_h$
stm $\Rightarrow$ stm ; stm	$S \llbracket stm_0 \rrbracket \theta \kappa_h = S \llbracket stm_1 \rrbracket (S \llbracket stm_2 \rrbracket \theta \kappa_h) \kappa_h$
exp $\Rightarrow$ (stm; exp)	$\mathcal{E} \llbracket exp_0 \rrbracket \kappa \kappa_h = S \llbracket stm_1 \rrbracket (\mathcal{E} \llbracket exp_1 \rrbracket \kappa \kappa_h) \kappa_h$
exp $\Rightarrow$ NUM	$\mathcal{E} \llbracket exp_0 \rrbracket \kappa \kappa_h = \kappa(\mathcal{N} \llbracket NUM \rrbracket)$
exp $\Rightarrow$ ID	$\mathcal{E} \llbracket exp_0 \rrbracket \kappa \kappa_h = \lambda \sigma. \kappa(\sigma(I \llbracket ID \rrbracket)) \sigma$
exp $\Rightarrow$ exp + exp	$\mathcal{E} \llbracket exp_0 \rrbracket \kappa \kappa_h = \mathcal{E} \llbracket exp_1 \rrbracket (\lambda v_1. \mathcal{E} \llbracket exp_2 \rrbracket (\lambda v_2. \kappa(\text{plus}(v_1, v_2)))) \kappa_h \kappa_h$
exp $\Rightarrow$ exp handle ID => exp	$\mathcal{E} \llbracket exp_0 \rrbracket \kappa \kappa_h = \mathcal{E} \llbracket exp_1 \rrbracket \kappa (\lambda v. \lambda \sigma. \mathcal{E} \llbracket exp_2 \rrbracket \kappa \kappa_h (\text{upd}(\sigma, I \llbracket ID \rrbracket, v)))$
exp $\Rightarrow$ raise exp	$\mathcal{E} \llbracket exp_0 \rrbracket \kappa \kappa_h = \mathcal{E} \llbracket exp_1 \rrbracket \kappa_h \kappa_h$

41

42

## Gotos and Labels

### Types:

$V$	=	$Int$
$Ide$	=	$Identifiers$
$\sigma : S$	=	$Ide \rightarrow V$
$A$	=	$Answers$
$\theta : C$	=	$S \rightarrow A$
$\tau : Cenv$	=	$Ide \rightarrow C$
$Exp$	=	$S \rightarrow V$
$Stm$	=	$Cenv \rightarrow C \rightarrow (Cenv \times C)$
$Prog$	=	$A$

**New idea: associate a label with a continuation**

Needs **fixed point** to find all labels that could be targets of GOTOS.

Uses a continuation environment to map labels to continuations.

### Functions and Values:

plus :	$(V \times V) \rightarrow V$
upds :	$(S \times Ide \times V) \rightarrow S$
updc :	$(Cenv \times Ide \times C) \rightarrow Cenv$
ifnz :	$(V \times A \times A) \rightarrow S$
snd :	$(Cenv \times C) \rightarrow C$
done :	$C$
$\sigma_0$ :	$S$
$I$ :	$ID \rightarrow Ide$
$\mathcal{N}$ :	$NUM \rightarrow V$
$\mathcal{E}$ :	$exp \rightarrow Exp$
$S$ :	$stm \rightarrow Stm$
$\mathcal{P}$ :	$prog \rightarrow Prog$

43

44

Grammar rules and semantic equations:

$\text{prog} \Rightarrow \text{stm}$   
 $\mathcal{P} \llbracket \text{prog}_0 \rrbracket = \text{snd}(\text{fix}(\lambda(\tau, \theta). \mathcal{S} \llbracket \text{stm}_1 \rrbracket \tau \text{ done})) \sigma_0$

$\text{stm} \Rightarrow \text{ID} := \text{exp}$   
 $\mathcal{S} \llbracket \text{stm}_0 \rrbracket \tau \theta = (\tau, \lambda \sigma. \theta(\text{upds}(\sigma, I \llbracket \text{ID} \rrbracket, \mathcal{E} \llbracket \text{exp}_1 \rrbracket \sigma)))$

$\text{stm} \Rightarrow \text{ID}:$   
 $\mathcal{S} \llbracket \text{stm}_0 \rrbracket \tau \theta = (\text{updc}(\tau, I \llbracket \text{ID} \rrbracket, \theta), \theta)$

$\text{stm} \Rightarrow \text{stm} ; \text{stm}$   
 $\mathcal{S} \llbracket \text{stm}_0 \rrbracket \tau \theta = \text{uncurry}(\mathcal{S} \llbracket \text{stm}_1 \rrbracket)(\mathcal{S} \llbracket \text{stm}_2 \rrbracket \tau \theta)$

$\text{stm} \Rightarrow \text{IF exp GOTO ID}$   
 $\mathcal{S} \llbracket \text{stm}_0 \rrbracket \tau \theta = (\tau, \lambda \sigma. \text{ifnz}(\mathcal{E} \llbracket \text{exp}_1 \rrbracket \sigma, \tau(I \llbracket \text{ID} \rrbracket), \theta \sigma))$

$\text{exp} \Rightarrow \text{NUM } \mathcal{E} \llbracket \text{exp}_0 \rrbracket \sigma = \mathcal{N} \llbracket \text{NUM} \rrbracket$

$\text{exp} \Rightarrow \text{ID } \mathcal{E} \llbracket \text{exp}_0 \rrbracket \sigma = \sigma(I \llbracket \text{ID} \rrbracket)$

$\text{exp} \Rightarrow \text{exp} + \text{exp}$   
 $\mathcal{E} \llbracket \text{exp}_0 \rrbracket \sigma = \text{plus}(\mathcal{E} \llbracket \text{exp}_1 \rrbracket \sigma, \mathcal{E} \llbracket \text{exp}_2 \rrbracket \sigma)$

For simplicity do not involve locations.

45

Types:

$V = \text{INT of } Int + \text{FN of } (K \rightarrow K)$   
 $Ide = \text{Identifiers}$   
 $Env = Ide \rightarrow V$   
 $A = \text{Answers}$   
 $K = V \rightarrow A$   
 $Exp = Env \rightarrow K \rightarrow A$   
 $Prog = A$

New idea: functions as values

like  $\mu$ Scheme uses left-to-right evaluation, enforced by expression continuations.

Environment, but no store since no assignment.

Note  $V$  is a sum type with injections INT, and FN.

46

Functions:

$\text{plus} : (Int \times Int) \rightarrow Int$   
 $\text{print} : V \rightarrow (K \rightarrow A)$   
 $\text{upd} : (Env \times Ide \times V) \rightarrow Env$   
 $\text{done} : K$   
 $\rho_0 : Env$   
 $I : ID \rightarrow Ide$   
 $\mathcal{N} : NUM \rightarrow V$   
 $\mathcal{E} : \text{exp} \rightarrow Exp$   
 $\mathcal{P} : \text{prog} \rightarrow Prog$

Grammar rules and semantic equations:

$\text{prog} \Rightarrow \text{exp } \mathcal{P} \llbracket \text{prog}_0 \rrbracket = \mathcal{E} \llbracket \text{exp}_1 \rrbracket \rho_0 \text{ done}$

$\text{exp} \Rightarrow \text{NUM } \mathcal{E} \llbracket \text{exp}_0 \rrbracket \rho \kappa = \kappa(\mathcal{N} \llbracket \text{NUM} \rrbracket)$

$\text{exp} \Rightarrow \text{ID } \mathcal{E} \llbracket \text{exp}_0 \rrbracket \rho \kappa = \kappa(\rho(I \llbracket \text{ID} \rrbracket))$

$\text{exp} \Rightarrow (\text{exp exp})$   
 $\mathcal{E} \llbracket \text{exp}_0 \rrbracket \rho \kappa = \mathcal{E} \llbracket \text{exp}_1 \rrbracket \rho(\lambda \text{FN}(f). \mathcal{E} \llbracket \text{exp}_2 \rrbracket \rho(f \kappa))$

$\text{exp} \Rightarrow (\text{lambda (ID) exp})$   
 $\mathcal{E} \llbracket \text{exp}_0 \rrbracket \rho \kappa = \kappa(\text{FN}(\lambda \kappa_2. \lambda v. \mathcal{E} \llbracket \text{exp}_1 \rrbracket(\text{upd}(\rho, I \llbracket \text{ID} \rrbracket, v)) \kappa_2))$

$\text{exp} \Rightarrow +$   
 $\mathcal{E} \llbracket \text{exp}_0 \rrbracket \rho \kappa = \kappa(\text{FN}(\lambda \kappa_1. \lambda \text{INT}(i_1). \kappa_1(\text{FN}(\lambda \kappa_2. \lambda \text{INT}(i_2). \kappa_2(\text{INT}(\text{plus}(i_1, i_2))))))))$

$\text{exp} \Rightarrow \text{PRINT } \mathcal{E} \llbracket \text{exp}_0 \rrbracket \rho \kappa = \kappa(\text{FN}(\lambda \kappa_1. \lambda v_1. \text{print } v_1 \kappa_1))$

47

48

## Direct style:

```
fun fact n =
  let fun f (i, p) =
        if i = 0 then p else f (i-1, i*p)
      in f(n, 1)
    end
```

## Continuation-passing style:

```
fun f (i, p) k =
  eq i 0 (fn () => k p)
    (fn () =>
      minus i 1 (fn i' =>
        times i p (fn p =>
          f (i', p) (fn x => k x))))))
fun fact n k =
  f (n, 1) (fn x => k x)

-      : int * int -> int
minus : int -> int -> (int -> Answer)
      -> Answer

=      : int * int -> bool
eq     : int -> int -> (unit -> Answer)
      -> (unit -> Answer) -> Answer
```

49

## Need denotation of types e.g.

$$\llbracket \text{bool} \rrbracket = \{\perp_B, \text{truth}, \text{falsity}\}$$

Only well-typed terms have meanings so we give meanings to type derivations rather than to abstract syntax trees: judgement

$$\Gamma \vdash M : \tau$$

is given a meaning

$$\llbracket \Gamma \vdash M \rrbracket$$

The meaning of a well-typed term  $M$  given type environment  $\Gamma$  is a function from meanings of variables (of the right type) in the type environment to a meaning for the term (which has to be in the set of meanings described by  $\llbracket \tau \rrbracket$ ).

Extend with store, continuations as needed

See Pitts, Section 6 on denotational semantics of PCF (simply-typed Scheme subset)

50

## Theoretical uses of denotational semantics

**Prove operational semantics correct** Given a denotational semantics for a language, and for data used in the abstract machine: environment, store, etc., show that axioms of the operational semantics are true, show that the rules preserve truth.

**Proofs that are difficult or impossible in operational semantics:**

$$\text{fix}(g \circ f) = g(\text{fix}(f \circ g))$$

or Bekić's Theorem: The least fixed point of a tuple of functions can be defined in terms of the least fixed points of the individual functions.

## To remember

Denotational semantics:

- compositional: can have one (or more) semantic clauses per grammar rule.
- maps program phrases to meanings

Domains:

- approximation ordering
- new domains from old: example domain of continuous functions.

Mapping programs to domains:

- think of environment and store as functions
- loops become fixed-points
- continuations model flow of control

Continuation semantics of program vs. direct semantics of translation of program into continuation-passing style.

51

52

## Types and functions

```

type V      = int
type Ide    = string
type S      = Ide -> V
datatype A  = ERROR
            | ANSWER of V (* value returned *)
type C      = S -> A
type K      = V -> C
type Exp    = K -> C
type Stm    = C -> C

```

```

exception NotFound
val plus    : V * V -> V = op +
val upd     : S * Ide * V -> S
            = fn (sigma, n, v) => fn n' =>
              if n = n' then v else sigma n'
val done    : V -> A = fn x => ANSWER x
val sigma0  : S      = fn n => raise NotFound

```

53

## Abstract syntax

```

datatype prog = PROG of stm
and          stm = op := of Ide * exp
              | RETURN of exp
              | SEQs  of stm * stm
and          exp = SEQe  of stm * exp
              | NUM   of V
              | ID    of Ide
              | op +  of exp * exp

```

54

## ML code — Expression continuations

## Semantic Equations

```

fun P (PROG stm1) = S stm1 (fn s => ERROR) sigma0
and S (id := exp1) th =
  | S (RETURN exp1) th =
    E exp1 (fn v => fn s => th (upd(s, id, v)))
  | S (SEQs (stm1, stm2)) th = S stm1 (S stm2 th)
and E (SEQe (stm1, exp1)) k = S stm1 (E exp1 k)
  | E (NUM v) k = k v
  | E (ID n) k = (fn s => k (s n) s)
  | E (exp1 + exp2) k =
    E exp1 (fn v1 =>
      E exp2 (fn v2 => k (plus(v1, v2))))

```

55