Lesson 15

• More on Denotational semantics
• Not 1-1 bindings: Aliases and overloading
• Shallow and deep binding
Semantic Domains in Denotational Semantics

• Semantic domains must allow to model partiality and potentially infinite computations
• They must provide solutions to two kinds of recursive definitions
  – Definition of interpretation functions on iterative constructs, like “while Exp do Com”
    \[ C\{\text{while } e \text{ do } c\} s = \]
    \[ \text{if } E\{e\} s = \text{false then } s \text{ else } C\{\text{while } e \text{ do } c\} (C\{c\}s) \]
  – Recursive definitions of domains

For example, if we add parameterless procedures as expressible values in LOOP, we get the recursive domain equations shown below

\[
\begin{align*}
P &= S \rightarrow S & \text{procedures} \\
E &= N + P & \text{expressible values} \\
S &= \text{Var} \rightarrow E & \text{states}
\end{align*}
\]
Kleene fixed-point theorem

• Complete Partial Orders satisfy all the listed requirements
  – A complete partial order (CPO) is a partial order with a least
    element and such that every increasing chain has a supremum
• An element $x \in D$ is a fixed point of a function $F: D \rightarrow D$ if
  $F(x) = x$
• Theorem: Every continuous function $F$ over a complete
  partial order (CPO) has a least fixed-point, which is the
  supremum of chain

$$F(\bot) \leq F(F((\bot))) \leq \ldots \leq F^n(\bot) \leq \ldots$$

• This can be exploited to solve recursive domain equations
  and recursive definitions of interpretation functions
• In the following: Domain = CPO
Defining CPO’s

• Several operators can help defining domains
  – Given a set $X$, its *lifting* $X \perp$ is $X \cup \{\perp\}$, where $\perp < y$ for all $y$ in $X$ (simply add $\perp$ to $X$ as least element)
    • we obtain primitive domains ($\text{Bool}_\perp$, $\text{Nat}_\perp$, $\text{Ide}_\perp$, $\text{Loc}_\perp$, ...)
  – Domains are closed under the following operations
    1. $D_1 \times D_2$  product (pairs)
    2. $D_1 + D_2$  sum (disjoint union)
    3. $D_1 \Rightarrow D_2$  continuous functions
    4. $D^n = D \times D \times ... \times D$  lists of length $n$
    5. $D^* = D^0 + D^1 + ... + D^k + ...$  finite lists
Nota;on

• Product: \( A \times B = \{(a,b) \mid a \in A, b \in B\} \)

  \( Projections: \quad \pi_1((a,b)) = a \quad \pi_2((a,b)) = b \)

• Sum (Union): \( A + B = \{(a,0) \mid a \in A\} \cup \{(b,1) \mid a \in B\} \)

  \( Injections: \quad \text{in}_1(a) = (a,0) \quad \text{in}_2(b) = (b,1) \)

  \( Case \ analysis: \quad ((x,y): A+B \text{ as } A) = (y=0 \rightarrow x, \text{ error}) \)

\( Note: \) we do not describe error handling and propagation

• Function space: \( A \rightarrow B = \{f \mid f: A \rightarrow B \text{ is a continuous function}\} \)

  \( Application: \) if \( f: A \rightarrow B \) and \( a \in A, \) then \( f(a) \in B \)

  Often brackets are omitted: \( f(a) \) becomes \( f \, a \)

• For defining functions we use as metalanguage the \( \lambda\)-notation
λ-notation

λ-terms: $\quad t ::= x \mid \lambda x.t \mid t \, t \mid t \mapsto t,t \mid (t)$

- $x$ variable
- $\lambda x.t$ abstraction, defines an anonymous function
- $t \, t'$ application of function $t$ to argument $t'$
- $t \mapsto t_0, \, t_1$ $= t_0$ if $t = true$ conditional $= t_1$ if $t = false$

- $[\lambda$-abstraction] $\quad x : A, \, t(x) : B$
  $\quad \text{function definition}$ $\quad \lambda x.t : A \rightarrow B$

- $[\beta$-conversion] $\quad (\lambda x.t) \, t' = t \, [t'/x]$
Introducing semantic domains

• Syntactic and semantic domains depend on the language
• Often, main syntactic domains: **Exp**, **Com**, **Decl** for Expressions, Commands, Declarations
• Representation of memory as domain \( S: \text{Var} \to \text{N} \) too simplistic. Needs to model:
  – same variable declared in multiple scopes
  – aliases
  – different effects of assignment (copy of value/reference)
  – pointers/references, ...
• Typical solution: indirect binding of variable names to values.
  – **Ide** (identifiers): syntactic domain for program variables, formal parameters, ...
  – **Loc** (locations): semantic domain for storage variables, references, pointers, ...
  – **Env = Ide \to Dval** (environments)
  – **Store = Loc \to Sval** (stores)

\[
\begin{align*}
x & \begin{array}{c}
\text{5}
\end{array} \quad r(x) = l \quad r: \text{Env} \quad x: \text{Ide} \\
\text{5} & \quad s(l) = 5 \quad s: \text{Store} \quad l: \text{Loc}
\end{align*}
\]

Example: “variable x contains 5”
Environments, Stores and Values

• Several domains for values:
  – **Sval** (storable values) values that can be stored in memory
    • Includes at least **N, Bool, ...**
  – **Dval** (denotable values) values that can be bound to variables
    • Includes **Loc**
  – **Eval** (expressible values) results of evaluation of expressions
    • Typically includes **Sval**

• Domains of values can differ greatly among languages

• Eg: in Pascal **Sval** = Num + Bool and **Dval** = Loc + Arrays + Proc + Label + ...

• Stores are equipped with primitive operations
  – **content** : **Loc** → **Store** → **Sval**
    content(l)(s) = s(l)  content l s = s l
    content = λl.λs.s l
  – **update** : (**Loc** × **Sval**) → **Store** → **Store**
    update = λx. λs . λl. (l = π_0(x) ↦ π_1(x), s l)
    update (l', v) s l = (l = l' ↦ v, s l)
Semantic interpretation functions

• The semantic interpretation functions are

\[ D: \text{Decl} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow (\text{Env} \times \text{Store}) \]
  – A declaration can modify both the env. and the store

\[ C: \text{Cmd} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow \text{Store} \]
  – Assumes that commands cannot change the env.

\[ E: \text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow \text{Eval} \]
  – Assumes absence of side effects

OR

\[ E: \text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow (\text{Eval} \times \text{Store}) \]
  – Allows for side effects during expr. evaluation
Example: Declaration of variables and assignment

**Syntax**

Decl ::= var Ide = Exp | Exp ::= ... Com ::= Exp := Exp | ...

**Semantics: declaration**

$D\{\text{var } x = e\} r s = (r[l/x], s[n/l])$

where $l = \text{newloc}(s)$

and $n = E\{e\} r s$

Allocates a new location bound to $x$ and containing $n$

**Semantics: assignment**

$C \{e1 := e2\} r s = \text{update}(x \text{ as Loc, } v \text{ as Sval}) s2$

where $(x, s1) = E\{e1\} r s$

and $(v, s2) = E\{e2\} r s1$

Evaluates first $e1$ then $e2$: store changes are propagated.

**Semantic interpretation functions**

$D: \text{Decl} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow (\text{Env} \times \text{Store})$

$C: \text{Cmd} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow \text{Store}$

$E: \text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow \text{Eval} \text{ no side eff}$

$E: \text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow (\text{Eval} \times \text{Store})$

$\text{Env} = \text{Ide} \rightarrow \text{Dval}$

$\text{Store} = \text{Loc} \rightarrow \text{Sval}$

$\text{Dval} = \ldots + \text{Loc} + \ldots$

$\text{Eval} = \ldots + \text{Loc} + \text{Sval} + \ldots$
Example: Parameterless procedures and blocks with static and dynamic scoping

### Syntax

<table>
<thead>
<tr>
<th>Decl ::= ...</th>
<th>proc Ide {Com}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Com ::= ...</td>
<td>call Ide</td>
</tr>
<tr>
<td></td>
<td>{Decl; Com}</td>
</tr>
</tbody>
</table>

### Semantic domains

- **Declarations**
  - **Decl**
  - $D: \text{Decl} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow (\text{Env} \times \text{Store})$

- **Dynamic scoping**
  - **Proc** = $\text{Env} \rightarrow \text{Store} \rightarrow \text{Store}$

- **Static scoping**
  - **Proc** = $\text{Store} \rightarrow \text{Store}$

### Dval = ... + Proc

#### Semantics: blocks

<table>
<thead>
<tr>
<th>$C{d; c} r s = C{c} r' s'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>where $(r', s') = D{d} r s$</td>
</tr>
</tbody>
</table>

#### Semantics: parameterless, dynamic scoping

- **$D\{\text{proc } p\{c\}\} r s = (r[k / p], s)$**
  - where $k = \lambda r'. \lambda s'. C\{c\} r' s'$
- **$C\{\text{call } p\} r s = (r(p) \text{ as } \text{Proc}) s$**

#### Semantics: no parameter static scoping

- **$D\{\text{proc } p\{c\}\} r s = (r', s)$**
  - no recursion!
  - where $r' = r[\lambda s'. C\{c\} r s' / p]$
- **$C\{\text{call } p\} r s = (r(p) \text{ as } \text{Proc}) s$**

#### Recursion

- **$D\{\text{proc } p\{c\}\} r s = (r[\alpha_0 / p], s)$**
  - where $\alpha_0$ minimal fixed point of
  - $\alpha = \lambda s'. C\{c\} r[\alpha / p] s'$
ALIASES AND OVERLOADING
DEEP AND SHALLOW BINDING
Not 1-to-1 bindings: **Aliases**

**Aliases**: two or more names denote the same object
Arise in several situations:

- **Pointer-based data structures**

  Java:
  Node n = new Node("hello", null);
  Node n1 = n;

- **common** blocks (**Fortran**), variant records/unions (**Pascal, C**)

- Passing (by name or by reference) variables accessed non-locally

  ```
  double sum, sum_of_squares;
  ...
  void accumulate(double& x)
  {
    sum += x;
    sum_of_squares += x * x;
  }
  ...
  accumulate(sum);
  ```
Problems with aliases

• Make programs more confusing
• May disallow some compiler’s optimizations

int a, b, *p, *q;
...
a = *p; /* read from the variable referred to by p*/
*q = 3; /* assign to the variable referred to by q */
b = *p; /* read from the variable referred to by p */

• Cause for a long time of inefficiency of C versus FORTRAN compilers
Not 1-to-1 bindings: Overloading

• A name that can refer to more than one object is said to be **overloaded**
  – Example: + (addition) is used for integer and floating-point addition in most programming languages
• Overloading is typically resolved at **compile time**
• Semantic rules of a programming language require that the context of an overloaded name should contain sufficient information to deduce the intended binding
• Semantic analyzer of compiler uses type checking to resolve bindings
• Ada, C++, Java, … function overloading enables programmer to define alternative implementations depending on argument types (**signature**)
• Ada, C++, and Fortran 90 allow built-in operators to be overloaded with user-defined functions
  – enhances expressiveness
  – may mislead programmers that are unfamiliar with the code
First, Second, and Third-Class Subroutines

- **First-class object**: an object entity that can be passed as a parameter, returned from a subroutine, and assigned to a variable
  - Primitive types such as integers in most programming languages
    - The object is in **Sval**, **Eval** and **Dval**
- **Second-class object**: an object that can be passed as a parameter, but not returned from a subroutine or assigned to a variable
  - Fixed-size arrays in C/C++
    - The object is in **Dval**
- **Third-class object**: an object that cannot be passed as a parameter, cannot be returned from a subroutine, and cannot be assigned to a variable
  - Labels of goto-statements and subroutines in Ada 83

- Functions in Lisp, ML, and Haskell are unrestricted first-class objects
- With certain restrictions, subroutines are first-class objects in Modula-2 and 3, Ada 95, (C and C++ use function pointers)
Scoping issues for first/second class subroutines

• Critical aspects of scoping when
  – Subroutines are passed as parameters
  – Subroutines are returned as result of a function

• Resolving names declared *locally* or *globally* is obvious
  – **Global** objects are allocated statically (or on the stack, in a fixed position)
    • Their addresses are known at compile time
  – **Local** objects are allocated in the activation record of the subroutine
    • Their addresses are computed as \textit{base of activation record} + \textit{statically known offset}
“Referencing” (“Non-local”) Environments

• If a subroutine is passed as an argument to another subroutine, when are the static/dynamic scoping rules applied?
  1) When the reference to the subroutine is first created (i.e. when it is passed as an argument)
  2) Or when the argument subroutine is finally called

• That is, what is the referencing environment of a subroutine passed as an argument?
  – Eventually the subroutine passed as an argument is called and may access non-local variables which by definition are in the referencing environment of usable bindings

• The choice is fundamental in languages with dynamic scope: **deep binding (1) vs shallow binding (2)**

• The choice is limited in languages with static scope
Effect of Deep Binding in Dynamically-Scoped Languages

Program execution:

```
main(p)
  bound:integer
  bound := 35
  show(p,older)
    bound:integer
    bound := 20
    older(p)
      return p.age > bound
    if return value is true
      write(p)
show(p,older)
```

```
function older(p:person):boolean
  return p.age > bound
procedure show(p:person,c:function)
  bound:integer
  bound := 20
  if c(p)
    write(p)
procedure main(p)
  bound:integer
  bound := 35
  show(p,older)
```

Program prints persons older than 35
Effect of Shallow Binding in Dynamically-Scopded Languages

The following program demonstrates the difference between deep and shallow binding:

```plaintext
function older(p:person):boolean
    return p.age > bound
procedure show(p:person,c:function)
    bound:=20
    if c(p)
        write(p)
procedure main(p)
    bound:=35
    show(p,older)
    older(p)
    return p.age > bound
if return value is true
    write(p)
```

Program execution:

```
20
Program prints persons older than 20
```