Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

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{**while** *e* **do** *c*}s =

  if *E*{*e*}s = false then s else *C*{**while** *e* **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

P = S → S	procedures
E = N + P	expressible values
S = Var → E	states

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*

$$F(\perp) \leq F(F((\perp)) \leq \ldots \leq F^n(\perp) \leq ..$$

- 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* $\mathbf{X}_\perp$ is $X \cup \{\perp\}$, where $\perp < y$ for all y in X (simply add $\perp$ to X as least element)
    - we obtain primitive domains ($\mathbf{Bool}_\perp$, $\mathbf{Nat}_\perp$, $\mathbf{Ide}_\perp$, $\mathbf{Loc}_\perp$, **…**)
  - Domains are closed under the following operations

1. $\mathbf{D_1 \times D_2}$ product (pairs)
2. $\mathbf{D_1 + D_2}$ sum (disjoint union)
3. $\mathbf{D_1 \rightarrow D_2}$ **continuous** functions
4. $\mathbf{D^n = D \times D \times \ldots D}$ lists of length *n*
5. $\mathbf{D^* = D^0 + D^1 + \ldots + D^k + \ldots}$ finite lists

Notation

- Product: A x B = {(a,b) | a ∈ A, b ∈ B}
  Projections: π_1((a,b)) = a π_2((a,b)) = b
- Sum (Union): A + B = {(a,0) | a ∈ A} U {(b,1) | a ∈ B}
  Injections: $\mathbf{in}_1$(a) = (a,0) $\mathbf{in}_2$(b) = (b,1)
  Case analysis: ((x,y): A+B **as A**) = (y=0 → x, error)

Note: we do not describe error handling and propagation

- Function space: A → B = {f | f: A → B is a continuous function}
  Application: if f: A → B and a ∈ A, then f(a) ∈ B
  Often brackets are omitted: **f(a)** becomes **f a**
- For defining functions we use as metalanguage the ***λ-notation***

λ-notation

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

- 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 $t = false$
- [λ-abstraction]
  function definition
  $$\frac{x : A,\ t(x) : B}{\lambda x.t : A \rightarrow B}$$
- [β-conversion]
  function application $(\lambda x.t)\ t' = t\ [t'/x]$

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: Var → 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** → **Dval** (environments)
  - **Store** = **Loc** → **Sval** (stores)

Example: "variable x contains 5"

x [5]

r(x) = l   r: **Env**   x: **Ide**

s(l) = 5   s: **Store**   l : **Loc**

Environments, Stores and Values

Env = Ide → Dval (environments)
Store = Loc → Sval (stores)

- 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 x Sval) → Store → Store*
    $update = \lambda x.\ \lambda s\ .\ \lambda l.\ (l = \pi_0(x) \mapsto \pi_1(x),\ s\ l)$
    $update\ (l', v)\ s\ l = (l = l' \mapsto v,\ s\ l)$

Semantic interpretation functions

- The semantic interpretation functions are

D: **Decl** → Env → Store → (Env x Store)

– A declaration can modify both the env. and the store

C: **Cmd** → Env → Store → Store

– Assumes that commands cannot change the env.

E: **Exp** → Env → Store → Eval

– Assumes absence of side effects

OR

E: **Exp** → Env → Store → (Eval x 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{**var** *x* = *e*} r s = (r[l/x], s[n/l])

where l = *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 = *update(x, v)* s

where x = *E*{e1} r s ***as*** Loc

and v = *E*{e2} r s ***as*** Sval

No side-effects, no coercion

Semantics: assignment with side effects

C {e1 := e2} r s = *update(x* ***as*** Loc*, v* ***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: Decl → Env → Store → (Env x Store)

C: Cmd → Env → Store → Store

E: Exp → Env → Store → Eval *no side eff*

E: Exp → Env → Store → (Eval x Store)

Env = Ide → Dval

Store = Loc → Sval

Dval = … + Loc + …

Eval = … + Loc + Sval + …

Example: Parameterless procedures and blocks with static and dynamic scoping

Syntax

Decl ::= … | **proc** Ide {Com}

Com ::= … | **call** Ide
| {Decl; Com}

Semantics: blocks

C{ {d; c} } r s = *C*{c} r' s'
where (r',s') = *D{d}* r s

Semantics: parameterless, dynamic scoping

D{**proc** *p*{c}} r s = (r[k /p], s)
where k = ***λr'.*** *λs'.C*{*c*} **r'** s'

C{**call** *p*} r s = (r(p) as **Proc**) **r** s

Semantic domains

Declarations **Decl**

D: Decl → Env → Store → (Env x Store)

Dynamic scoping

Proc = Env → Store → Store

Static scoping

Proc = Store → Store

Dval = … + Proc

Semantics: no parameter static scoping

D{**proc** *p*{c}} r s = (r', s) **no recursion!**
where r' = r[*λs'.C*{*c*} r s' / p]

C{**call** *p*} r s = (r(p) as **Proc**) s

D{**proc** *p*{c}} r s = (r[α_0/p], s) **recursion**
where α_0 *minimal fixed point of*
α = *λs'.C*{*c*} r[α/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 (**Pacal**, **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 *base of activation record + 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

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

```
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 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)
```

Deep binding

Program prints persons older than 35

Effect of Shallow Binding in Dynamically-Scoped Languages

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

```
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 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)
```

Shallow binding

Program prints persons older than 20