

ICS/CSE 141: Programming Languages

Embedded & Domain-specific Languages

School of Information and Computer Science
University of California, Irvine

Chris Stork
`cstork@ics.uci.edu`

Overview

1. Remarks
2. Embedded & Domain-specific Languages
3. A Database Query Language

Remarks

Denotational Semantics of Conditionals

Conditionals can lead to one of two states

- $\mathcal{C}[\text{if } e \text{ then } c1 \text{ else } c2](s)$
= if $\mathcal{E}[e](s)$ then $\mathcal{C}[c1](s)$ else $\mathcal{C}[c2](s)$

Example: Given $s_0(x) = 1, s_0(y) = 2$ initial state

- $\mathcal{C}[\text{if } x > y \text{ then } x := y \text{ else } y := x](s_0)$
= if $\mathcal{E}[x > y](s_0)$ then $\mathcal{C}[x := y](s_0)$ else $\mathcal{C}[y := x](s_0) = \dots$
= if false then $\mathcal{C}[x := y](s_0)$ else $\mathcal{C}[y := x](s_0)$
= $\mathcal{C}[y := x](s_0) = \text{modify}(s_0, y, \mathcal{E}[x](s_0))$
= $\text{modify}(s_0, y, 1) = s_1$

with $s_1(x) = 1, s_1(y) = 1$ final state

Denotational Semantics of While-Loops

While loops can lead to **any** number of states!

- $\mathcal{C}[\text{while } e \text{ do } c](s)$
 $= \text{if not } \mathcal{E}[e](s) \text{ then } s \text{ else } \mathcal{C}[\text{while } e \text{ do } c](\mathcal{C}[c](s))$
- The **recursion** is a little tricky (mathematically), but we'll ignore that...

Example: Given $s_0(x) = 0, s_0(y) = 2$ initial state

- $\mathcal{C}[\text{while } y > x \text{ do } x := x + 1](s_0)$
 $= \text{if not } \mathcal{E}[y > x](s_0) \text{ then } s \text{ else}$
 $\mathcal{C}[\text{while } y > x \text{ do } x := x + 1](\mathcal{C}[x := x + 1](s_0)) = \dots$
 $= \text{if false then } s \text{ else}$
 $\mathcal{C}[\text{while } y > x \text{ do } x := x + 1](\text{modify}(s_0, x, 1)) = \dots$
 $= \mathcal{C}[\text{while } y > x \text{ do } x := x + 1](s_1) = \dots$
 $= \mathcal{C}[\text{while } y > x \text{ do } x := x + 1](s_2) = \dots = s_2$

with $s_2(x) = 2, s_2(y) = 2$ final state

Uses of Denotational Semantics

- Unambiguous Language Specification, e.g. in R5RS!
- Translation to Functional Language
- Study of type systems and other, more theoretical, issues

Side-Effect Example

```
static int g;  
g = 0;  
...  
int f(int a) {return a + (g++);}  
...  
x = f(1);  $\Rightarrow$  x = 1  
y = f(1);  $\Rightarrow$  y = 2
```

What's bad about side-effects?

- hidden dependencies decrease readability
- reduced modularity
- hinder optimizations (they “sequentialize”)
- often un(der)specified: $a[i] = i++;$ or
 $x = i++ * i++$

Referential Transparency Again

Two equal expressions can substitute each other without affecting the meaning of a functional program.

- provides for easy equational reasoning about a program
- does not rely on a particular notion of equality
 - Reference equality, shallow equality and deep equality cannot be distinguished by functional means
- is a major contrast to imperative programming

Equals can be replaced by equals

Referential Transparency Again II

$$(3 * f(a, b) + b) * (3 * f(a, b) - c)$$

is equivalent with

```
let t = 3*f(a,b)
in (t + b) * (t - c)
```

Would that work if $f(a, b)$ had the side-effect of incrementing c ?

Referential Transparency Again III

Referential Transparency can also be destroyed by aliasing...

Embedding a Database Query Language in Scheme

Note that what follows corresponds to an online demonstration of what's in the file `query.scm` version 1.13!

Studying that file might be more instructive!

Embedded & Domain-specific Languages

You most likely come in contact with new and different programming languages in form of

- Embedded languages, e.g. Lua (game scripting), Guile in Texmacs and Gnome, Elisp in Emacs, and every config file!
- Domain-specific languages, e.g. SQL, Infocom
- Glue/Scripting languages, e.g. Tcl, Python, Perl, Lua

As you can see that these definitions overlap!

We are going to embed a database query languages in Scheme!

Association Lists

For us a simple list of pairs/lists suffices as a database. :-)

Alist Examples

```
((a . 1) (b . 2) ... (z .26))  
((1 . 2) ("key" . "value") (a . b))  
((1 2 3) ("key" "value 1" val2 8) (a x y z))
```

The simplest implementation of a dictionary, hash, bag,...

Let's think of db entries as **relations** between objects! ... Simpsons

assoc & lookup

A list of pairs is also called an **alist**.

Queries

- Builtin (`assoc head alist`) returns the first association with `car head`

assoc & lookup

A list of pairs is also called an **alist**.

Queries

- Builtin (`assoc head alist`) returns the first association with `car head`
- To get all associations have to build our own `all-assocs...`

assoc & lookup

A list of pairs is also called an **alist**.

Queries

- Builtin (`assoc head alist`) returns the first association with `car head`
- To get all associations have to build our own `all-assocs...`

...learn about **named let** & loops

Matching Queries

Only looking up relations is not very exciting. How about:

```
<set up Simpsons db>
(match '(dog ?x) ==> santas-little-helper
(match '(sibling marge ?x) ==> ((?x . selma) (?x . patty))
(match '(sibling ?x ?y) ==> (((?x . marge) (?y . selma))
                             ((?x . marge) (?y . patty)))
(match '(sibling homer ?x) ==> ())
```

Pick the last, most general, reply syntax, i.e.

```
(match '(dog ?x)) ==> (((?x . santas-little-helper)))
(match '(sibling marge ?x)) ==> (((?x . selma) (?x . patty)))
(match '(male homer)) ==> (())
```

Encoding Failures and Success

`match` is a “fallible” function, i.e. it might fail. We encode

- **failure** as the **empty list** `()` and
- **success** as a **list with one element**, e.g. `(a)` or `(())` and
- **multiple successes** as **multiple elements of a list**, e.g. `(a b)`.

match

```
(match query db-entry)
;; match-entry: q -> e -> [f]
```

```
(match-db q) ; perform query over *db* returning matching frames
;; match: q -> [f]
```

```
(reap fallible-function list)
;; reap: (a -> [s]) -> [a] -> [s]
```

`mmatch`

Why can't we have the same logical variable appear multiple times?

```
(mmatch '(likes ?narcissist ?narcissist))  
=> (((?narcissist . homer)))
```

Use the frame...

Logical connectives

We want to use AND, OR, and NOT!

See `query` test, etc.

Assignment

Read Chapter 15 in Mitchell.

Thank You!