CSC 530 Lecture Notes Week 3

A Brief Review of Lambda Calculus

Introduction to Programming Language Type Systems

I. Readings -- papers 6 - 9

II. What is lambda calculus?

A. Used in readings.

B. Also used in our Lisp work.

C. Here’s a comparison with Lisp.

Lambda-Calculus, cont'd

Lambda Calc	Lisp
λ x.x	`(lambda (x) x)`

f = λ x.x	`(defun f (x) x)`
	-- *or* --
	`(setq f (lambda (x) x))`

f 1	`(f 1)`
	-- *or* --
	`(apply f (list 1))`

g = λ x.f x	`(defun g (x) (f x))`

Lambda-Calculus, cont'd

D. Notationally, some details differ.

E. We henceforth use Lisp notation.

1. As Hudak and Cardelli/Wegner use it.

2. Also for assignment 2.

Lambda-Calculus, cont'd

F. What is a lambda expr?

1. An anonymous function value.

2. E.g.,
   `(lambda (x) (+ x 1))`.

3. Apply to actual parameters

```
(apply
  (lambda (x) (+ x 1))
  '(10))
```

which delivers 11.

Lambda-Calculus, cont’d

4. These are the same:

```
(defun f (x) (+ x 1))
```

versus

```
(setq g
  (lambda (x) (+ x 1)))
```

5. Common Lisp disallows `(g 10)`

   a. We must `(apply g ’(10))`

   b. Just a technicality

Lambda-Calculus, cont’d

G. Where we use lambda exprs.

1. In solution to assignment 1.

2. Treat unevaluated function body as data.

Lambda-Calculus, cont'd

H. What else for lambda?

1. The grandparent of purely functional notations.

2. Denotational semantics defines *everything* as a function

Lambda-Calculus, cont’d

3. For example,

```
(setq memory
  (lambda (x)
    (cadr (assoc x
      ’((x 10) (y 20) (z 30))))))
```

4. Treats memory as function applied to name of memory location.

5. Look up of `y` is:

```
(apply memory ’(y))
```

Lambda-Calculus, cont'd

6. To add a new binding:

```
(defun add-binding
              (memory binding)

  (nconc (cadadr (cdadar
    (nthcdr 5 memory)))
      (list binding)))
```

Lambda-Calculus, cont'd

7. Wrap your head around these defs.

8. `add-binding` is exactly Tennent's "memory perturbation" function.

III. What does it mean to be typed?

A. *Def'n:* every data value has a *type*.

B. A type is

1. a *basic* (or primitive or atomic) type

2. a *composite* (or constructed or non-atomic) type

C. Primitive types are finite or infinite sets

Mean to be typed?, cont'd

D. Composite types use composition rules, e.g.,

1. a record type is composed of values of two or more types

2. an array type is composed of zero or more values of the same type

Mean to be typed?, cont’d

E. Type constrains how value may be interpreted.

F. In Cardelli an Wegner’s colorful metaphor

1. a typed value is *clothed*

2. an untyped value is *naked*

IV. Kinds of typedness

A. Strong versus weak typing

B. Static versus dynamic typing

C. Monomorphic versus polymorphic

D. Encapsulated versus flat

E. Subtyped versus non-subtyped

F. Generic versus non-generic

V. Spectrum of typeless to typeful

A. Lisp is weak, dynamic, monomorphic, flat, non-generic.

B. C is somewhat weak, static, monomorphic, flat, non-generic.

C. C++ is weakish, mostly static, subtype polymorphic, encapsulated, generic.

Spectrum, cont'd

D. Ada is strong, static, monomorphic, encapsulated, generic.

E. ML is strong, static, parametrically polymorphic, encapsulated, generic.

F. Java is strong, static (dynamically que- riable), subtype polymorphic, encapsu- lated, non-generic.

VI. The evolution of typing

A. LISP

B. FORTRAN

C. ALGOL 60

D. SIMULA 67

E. ALGOL 68

Evolution, cont'd

F. Pascal

G. Smalltalk

H. Modula-2

I. Ada

Evolution, cont’d

J. Modula-3 and Oberon

K. ML

L. C++

M. Java

N. C#

VII. Kinds of polymorphism

A. Genuine or "universal"

1. E.g.,

```
forall type T,
  function Eq(x:T, y:T) =
    x = y;
```

2. Function works for any args of same type with equality

Kinds of polymorphism, cont'd

B. Universal quantification is the most general form

1. Called *parametric*

2. Involves *type variables*

3. Type vars hold the position of a variable number of types

Kinds of polymorphism, cont'd

C. Another form through inheritance

1. E.g.,

```
class A = ... ;
class B subclass of A = ... ;
class C subclass of a = ... ;
function Eq(x:A, y:A) =
    x = y;
```

2. `Eq` is polymorphic over types A, B, and C.

3. Due to inheritance rules

Kinds of polymorphism, cont'd

D. Less general than parametric polymorphism,

1. Called *subtype* or *inclusion*

2. Standard rule -- function defined on parent type is polymorphic on all subtypes

Kinds of polymorphism, cont'd

E. *Apparent* or ad-hoc polym'ism

1. Two forms are overloading and coercion.

2. What distinguishes genuine from apparent?

    a. With overloading, separate function body for every set of arg types

    b. With coercion, types of actual parms are forced.

VIII. Type expression sublanguages

A. PLs provide linguistic features

B. Built-in atomic types

C. Mechanisms to build arrays, records, etc.

D. Type sublanguages vary widely.

E. We'll factor out syntactic details, focus on fundamental semantics.

IX. Types as sets

A. Definition above is two-fold

1. Base set of primitives

2. Set of composition rules

B. More basic formal def is entirely in terms of sets

C. We’ll discuss later in quarter

X. Lisp-based typed lambda calculus

A. Assmnt 2 entails type checking

B. We add typing primitives and rules to standard Lisp

C. Here's an overview:

Lisp-based types, cont'd

```
(deftype name type)

  where type is
   * sym, int, real, string,
     or bool
   * composite form
   * name of a def'd type
   * type var of the ?X


(array type [bounds])

    where bounds is integer,
    (int int) pair, or type var


(record fields)

    where fields is (name type)
    pairs or single type var
```

Lisp-based types, cont'd

```
(union fields)
    where fields is (name type)
    pairs or single type var

(function args [outs] [suchthat])
    where args and outs are names
    or (name type) pairs;  args,
    outs may be a single type var;
    suchthat is of form
        (suchthat predicate)
```

Lisp-based types, cont'd

D. `xdefun` extended as follows:

```
(defun name args [outs]
    [suchthat] body)
```

Lisp-based types, cont'd

E. Literal values for each types:

```
Type       Literal Denotation
==============================
sym        quoted atom

int        atom, integerp true

real       atom, numberp true,
           integerp false

string     atom, stringp true

bool       t or nil
```

Type literals, cont'd

array	list, elems meet array type specs
record	list, elems meet record type specs
union	value of one of field types
function	name of defun'd func or lambda expr