CSC 530 Lecture Notes Week 3, Part 2

Discussion of Assignments 1 and 2
More on Type Theory
Introduction to Typing in ML

I. "Naive" alist from Assignment 1

A. Bindings of the form

```
( name value ).
```

B. Two categories:

1. Var binding pair

```
( var-name
    data-value )
```

2. Function binding triple

```
( function-name
    formal-parms
    function-body )
```

Naive alist, Cont'd

C. Distinguish bindings by lengths.

D. Created and modified three ways:

1. Var bindings by `setq`
2. Func bindings `defun`
3. Func call bindings by

   `(f` a_1 `...` a_n`)`

Naive alist, Cont'd

E. Search for bindings in LIFO discipline

F. What is *naive* -- does not accurately represent scoping rules of Common Lisp.

II. Alist as environment and store

A. Consider two formal semantic structures

1. *Environment* holds static attributes

2. *Store* holds runtime values

   a. *stack* store holds function activations

   b. *state* store holds global vars

Environemnt and store, Cont'd

B. In pure untyped Lisp, env and store can be combined

C. Requires some additional structure on the alist.

III. Less naive alist layout

A. A number of ways to lay out.

B. Here's one:

```
(  ( state-store )
   ( environment )
   ( stack-store ) )
```

where `stack-store` has sub-alists, one per active function.

Less naive, Cont'd

C. Same bindings as before, organized into separate areas:

1. state-store holds global bindings by `setq`

2. environment holds bindings by `defun`

3. stack-store holds bindings by `apply` and `bind`

Less naive, Cont'd

D. Simple LIFO management replaced with:

1. Stack-store is still LIFO

2. State-store and environment managed any old way

3. Two-phase var binding search:

   a. First, topmost act rec

   b. Then state-store

4. Func binding search only in env

IV. Comparing naive vs non-naive

A. What's wrong with naive layout?

B. Defines *dynamic scoping* instead of *static scoping*.

C. Consider:

```
(setq x 1)
(setq y 2)
(defun f(y) (g))
(defun g() (+ x y))
```

Comparing, Cont'd

1. With static scoping, `(f 10)` and `(g)` return same result -- *3*.

2. With dynamic scoping, `(f 10)` returns *11*, `(g)` returns *3*.

D. Cause is traceable to naive alist handling.

Comparing, Cont'd

1. Simply start at end and search backward

2. Finds stack bindings first

3. May find in act rec

4. This explains above behavior.

Comparing, Cont'd

E. Rule for non-naive alist looks *only* in topmost act rec then state-store

F. Was dynamic binding ever used?

1. Sure -- before *Common* Lisp.
2. Stems from ease of interpretation

Comparing, Cont'd

G. Some things to consider:

1. Makes sense from implementation perspective.

2. Dynamic scoping and strong typing do not get along well.

3. With "Pascalization" of PLs, dynamic scoping is a relic.

V. Another view of Assmnt 1 and 2

A. Continuing objective is to investigate fundamental semantics PLs

B. What do programming constructs *really mean*?

C. These semantics are *operational*.

D. Why interpreters and why Lisp?

VI. Hints on Assignment 2

A. Put type bindings in env, along with `xdefuns`

1. form `( name type )`

2. Independent of value bindings in stores

3. E.g., for

```
(xdefvar x int 10)
```

`(x int)` in env, `(x 10)` in state-store

Hints, Cont'd

B. `xcheck` mirrors `xeval`

   1. Define `check-X` analogous to `eval-X`

   2. E.g., `check-xsetq` analog of `eval-xsetq`.

C. `equiv` function used internally

D. No really hairy test cases

Now on to Further Discussion of Type Theory

VII. Relevant readings -- Section 2 of papers

VIII. **Recap of kinds of typedness**

A. From Notes 3

1. Strong versus weak
2. Static versus dynamic
3. Mono- versus polymorphic
4. Encapsulated versus flat
5. Subtyped versus non-subtyped
6. Generic versus non-generic

Kinds of typedness, cont'd

B. Items 1-3 in Assignment 2.

C. Items 4-6 in Assignments 3 and 4.

IX. **Type encapsulation**

A. Used construct *absract datatypes*.

B. ADT defined as:

1. Hidden *representation*

2. *Operations*, with hidden imple’ns

Encapsualation, cont'd

C. Requires PL support, in some form

1. *Information hiding*.

2. *Packaging construct*.

Encapsualation, cont'd

D. Info hiding features differ syntactically, but same semantically.

E. Packaging differs both syntactically semantically.

Encapsualation, cont'd

1. Syntactic distinctions of no concern.

2. Important semantic distinction is whether ADT denotes a type.

3. When it does, construct is *first-class*.

4. When it does not, it is *second-class*.

X. Info hiding in PLs

A. Simula -- hidden/non-hidden, single class body

B. Modula-2 -- import/export, two-part module body

C. Ada -- private/non-private, two- or three-part package body

Info hiding, cont'd

D. C++ -- public/private/protected, crude one- or two-part class body

E. Java -- public/private/protected, one-part class body

F. Cardelli and Wegner existential types -- quantified variables, one-part representation (as a body)

XI. Second-class encapsulation

A. Decl of ADT independent of type declaration

B. Consider Modula-2 example:

```
definition module Stack;
    type Stack;
    procedure Push(var s: Stack;
        elem: integer);
    procedure Pop(var s: Stack):
        integer;
    procedure Peek(s: Stack):
        integer;
end Stack.
```

Second-class encapsulation, cont'd

```
implementation module Stack;
    const Size = 100;
    type Stack = array[1..Size]
        of integer;
    var curtop: integer;
    (* ... implementations
        of Push, Pop, and Peek *)
end Stack.

(* program *) module TestIntStack;
    import Stack;
    var s: Stack;
        i: integer;
begin
    Stack.Push(s, 1);
    i := Stack.Pop(s);
end TestIntStack.
```

Second-class encapsulation, cont'd

C. Noteworthy features

1. Module "Stack" distinct from type "Stack"

2. Info hiding via two-part module

D. Note calling form of ADT ops:

```
Stack.Push(s, 1);
i := Stack.Pop(s);
```

XII. First-class encapsulation

A. Decl of ADT declares a type.

B. Consider C++ example

First-class encapsulation, cont'd

```
class Stack {
  public:
    void Push(int elem);
    int Pop();
    int Peek();
  protected:
    const int Size = 100;
    int curtop;
    int body[Size];
};

/* ... implementations of ops */

main() {
    Stack s;
    int i;

    s.Push(1);
    i = s.Pop();
}
```

First-class encapsulation, cont'd

C. Noteworthy features

1. Class "Stack" is type "Stack"

2. Info hiding via explicit keywords.

3. Calling form of ADT ops:

```
s.Push(1);
i := s.Pop();
```

XIII. State-free ADTs

A. Consider variant of C++ stack:

```
class Stack {
  public:
    Stack();
    void push(int elem);
    void pop();
    int top();

  equations:
    pop(Stack()) = null;
    pop(s.push(e)) = s;

    top(Stack()) = null;
    top(s.push(e)) = e;
};
```

State-free ADTs, cont'd

```
/* NO implementations of
   Push, Pop, and Peek */

main() {
    Stack s;
    int i;

    s = s.Push(1);
    i = s.Pop();
}
```

State-free ADTs, cont'd

B. Noteworthy features ...

1. Still first-class ADT.

2. Info hiding is *unecessary*!

3. No op implementations necessary!

4. Side-effect-free calling:

```
s = s.Push(1);
i = s.Pop();
```

State-free ADTs, cont'd

C. Does this mean anything?

1. Most definitely yes.

2. It's C++ syntax for the OBJ algebraic language

XIV. Object-orientedness, part 1

A. Relation between ADTs and OO?

B. Partial ans: OO needs ADTs.

C. Need *first-class* ADTs for OO?

D. Most say yes.

E. However, there's Booch's "OOP in Ada".

XV. **Subtyping**

A. Allows *parent* type from which *child* types *inherit*.

B. Does subtyping require ADT?

1. Theoretic answer is no.

2. In practice, yes in most PLs.

3. Notable exception is OBJ.

Subtyping, cont'd

C. Wide variety of issues:

1. Multiple or single inheritance.
2. Representation inheritance.
3. Representation and/or operation overriding.
4. Op behavior inheritance.
5. Strong, weak, static, or dynamic typing.

XVI. Subtyping in common PLS

A. **Simula --** single rep+ops inheritance, full override, strong static typing

B. **Smalltalk --** single rep+ops inheritance, full override, weakish dynamic typing

C. **Modula-2**, **Ada** (pre 9X) -- no subtyping

Subtyping in PLs, cont'd

D. **C++ --** multiple rep+ops inheritance, full override, weakish mostly static typing

E. **Emerald, Owl --** single ops-only inheritance, no override, strong static typing, behavior inheritance

F. **Java --** single reps inheritance (classes), multiple ops inheritance (interfaces), full override, strong mostly static typing

XVII. **Generics**

A. A form of parameterized type

B. Almost all languages provide generics with ADTs.

1. Unencapsulated parameterized types reasonable in theory.

2. Euclid is notable.

Generics, cont'd

C. A Modula-2esque example

```
(* A generic stack module. *)
definition module Stack(
        Size: integer,
        ElemType: type);

    type Stack;

    procedure Push(var S: Stack;
        Elem: ElemType);

    procedure Pop(var S: Stack):
        ElemType:

    procedure Peek(S: Stack):
        ElemType;

end Stack.
```

Generics, cont'd

```
implementation module Stack;

    type Stack = array[1..Size]
        of ElemType;

    var curtop: integer;

    (* ... implementations of
        Push, Pop, and Peek *)

end Stack;
```

Generics, cont'd

```
(* program *) module TestIntStack;

    import Stack(100, integer);

    var S: Stack;
        i: integer;

begin

    Stack.Push(S, 1);

    i := Stack.Pop(S);

end TestIntStack;
```

Generics, cont'd

```
(* program *) module TestThreeStacks;
    import Stack(100, integer)
        as IntStack100;
    import Stack(200, integer)
        as IntStack200;
    import Stack(200, real)
        as RealStack200;

    var SI100: IntStack100.Stack;
    var SI200: IntStack200.Stack;
    var SR200: RealStack200.Stack;

begin
    IntStack100.Push(SI100, 1);
    IntStack200.Push(SI200, 2);
    RealStack200.Push(SR200, 2.5);
    (* etc. ... *)
end TestThreeStacks;
```

XVIII. **OO, complete picture**

A. So, what exactly constitutes OO?

B. Danforth and Tomlinson's take:

Is it possible that OOP is simply ADTs? Not an unreasonable conjecture ... . However, there is more -- inheritance.

C. Summary, OOP = first-class ADTs + inheritance.

OO complete picture, cont'd

D. Almost universally accepted that generics *NOT* needed for OO.

E. An inherent conflict of OOP:

1. a primary goal of data abstraction is to hide info

2. a primary goal of inheritance is to share info

F. Does this mean that OOP is a crock?