DUE: Monday 1 May 2006

1. Aliasing

Broadly defined, *aliasing* occurs whenever there are 2 or more reference paths to the same location or object.

(a) Distinguish between *pointer aliasing* and *name aliasing.* Illustrate your explanations with examples from some programming language. Could there be other kinds of aliasing? Illustrate with examples.

(b) The following procedure divides the parameters a, b by their greatest common divisor, leaving them as a "simplest ratio" without common factors. It is used to reduce a ratio $a : b$ to lowest terms. All arguments are assumed positive integers. Assume that at call time variable references are passed (CBR).

```
procedure simpleratio(var a, b : integer) ; { a, b > 0 and integer }
    var c, d : integer;
    begin
        c:= a; d:= b;
        while c <> d do
            if c > d then c:= c − d
                else d:= d − c ;
        b:= b div c;
        a:= a div c;
    end { simpleratio };
```

Describe what happens with calls like $simpleratio(x, x)$? How can this bug be fixed?

(c) The *Pascal User Manual* states: "the actual parameter [corresponding to a **var** formal parameter] must be a variable . . . Furthermore, if *x1*, · · · , *xn* are the actual variables that correspond to the formal **var** parameters *v1*, · · · , *vn*, then *x1*, · · · , *xn* should be *distinct* variables."

The phrase following the "Furthermore" has become known as the *actualization taboo* (since the *Manual* says "should" but not "must", it seems to be taken as a moral but not a legal injunction).

If the taboo is violated, what kind of aliasing can occur, in the terminology of part (a)?

(d) Another source of aliasing can occur when there are variables free in a procedure body. Describe what can happen by an example from a specific language. Can the parameter-passing mechanism adopted by the language affect this kind of aliasing, by perhaps preventing it or by preventing certain kinds of aliasing?

2. Boolean Operators

We want to add the Boolean operators '**&&**' (''and'') and '‖' (''or'') to the language IMP (Example 3.6, p. 66 of text). To this end, we extend the syntax of expressions by adding the following productions:

```
Expression ::=   · · ·
    | Expression && Expression
    | Expression ‖ Expression
```

(a) Which of the semantic domains **Value**, **Storable**, and **Bindable** need to change as a result of this extension to the language?

(b) Complete the following semantic equation, under the assumptions listed below:

$evaluate[\![E_1 \&\& E_2]\!]\, env\ sto =$

(i) Assuming that short-circuit evaluation is **not** being used.

(ii) Assuming that short-circuit evaluation **is** being used.

3. Multiple Arguments

Extend the IMP language of Watt, Example 3.9 to allow multiple parameters. See the discussion at (3.104, 3.105). *HINT:* Remarkably little of the syntax and semantics has to change, but you will have to deal with argument *lists* in Argument *.

4. Continuation Semantics of `break`

Using as a basis the model language IMP$_g$ developed in the text and in class, describe a **denotational continuation semantics** for the language IMP$_g$ extended to allow **`break`** statements within **`while`** loops. Assume the following simiplifications:

- that the only looping structure in IMP$_g$ is the **`while`** syntactic structure.
- that expressions have *no side-effects or jumps* (so that **only command continuations** are needed and expression continuations do not have to be discussed).

Extend the language syntax to have a new Command called **`break`**. If a **`break`** occurs in the body of a **`while`**, execution is to continue with the continuation of the entire **`while`** construct. If a **`break`** occurs with no surrounding loop, the resulting program is invalid syntactically. The **`break`** is always made to the continuation of the *closest surrounding* loop construct.

HINT: If **`while`** *E* **`do`** *C* executes in environment *e*, what environment must *C* execute in, if breaks *out of C* are to be allowed?

(a) Provide grammar rule(s) for the new syntax. Only show *changes*.

(b) Describe any new contextual constraints (static semantics). Only describe *new* constraints. If there are none, say so.

(c) Give the new semantic rule for

$$execute : \text{Command} \rightarrow \text{Environ} \rightarrow \text{Cont} \rightarrow \text{Cont}$$

$$execute\ [\![\mathbf{while}\ E\ \mathbf{do}\ C]\!]\ env\ cont$$

(d) Give the semantic rule for

$$execute\ [\![\mathbf{break}]\!]$$