DUE: Monday 17 April 2006

1. Minimal Fixed Points

By unfolding ("unwinding") the following recursive functions, infer what the minimal fixed point function is. Then, by direct substitution into the right-hand-side functional, prove that the inferred function is indeed the fixed point. That is, verify directly that the fixed point identity holds, given in (5.19), p. 142, Watt. Assume that all functions are defined only on the non-negative integers.

(a) $f\ n\ =$ **if** *zero n* **then false else not** $(f(succ\ n))$

(b) $f\ n\ =$ **if** *zero n* **then** 1 **else** $2 \cdot f(pred\ n)$

2. Interpreting the Semantic Description of Δ

Watt text, p. 119, Exercise 4.5. For each question, show the semantic function or functions upon which you base your answer, and explain by analyzing those semantic functions how your answer was deduced. Answers can also be based upon the definitions of semantic domains. Some answers require that you explain certain auxiliary functions as well.

Ground Rule: you need to defend your answer by explaining how it follows by the formal semantics—just quoting the discussion in Appendix B is not adequate.

Note: there is no item ''(k)''.

3. Lazy Evaluation of Arguments

Change the semantics of IMP_d given in Example 3.8, pp. 74-76, so that an argument to a subroutine (procedure or function) is evaluated *in the environment that is valid where the parameter is first (dynamically) accessed* inside the body of the procedure. In other words, give the semantics of lazy evaluation (for one-argument subroutines).

This is *not the same as dynamic binding*. In dynamic binding (discussed in class) the arguments are evaluated *at the time of call*, and free names in the subroutine body are resolved in the environment at the time of call.

For this problem, assume that free names are *statically bound*. Argument expressions are not evaluated at call time, but only when the formal parameter is referenced in the subroutine body.

Note that you have only to worry about *r*-values of arguments, since there is no way for information to be returned through the argument list.

You may assume either of the following semantics for argument evaluation, **but you must describe which of these your semantics describes**:

(a) Call by Name: the argument is *re-evaluated* each time the formal is referenced in the subroutine body

(a) Call by Need, *r*-value: the argument is *evaluated once* when the formal is first referenced, and the same *r*-value is used for every subsequent reference to the formal.

As usual provide the following:

(1) changes to syntax

(2) contextual constraints (informally)

(3) changes to semantic domains

(4) changes to signatures of semantic functions

(5) all required semantic rules, along with any auxiliary functions needed. Focus on the rules that *change*.

HINT: The meaning of an argument is now a function in the domain $\mathsf{Argument} = \mathsf{Store} \rightarrow \mathsf{Value}$. This function will be evaluated when the corresponding formal is referenced. Think *thunk*, but remember this is semantics, and not language implementation. You are in search of the *meaning* of delayed arguments.

4. Functions with Side Effects

Refer to the extension of IMP with function and procedure declarations (Examples 3.8 and 3.9 of the Watt text). Assume that parameters are passed using a definitional mechanism (e.g., via *bind-parameter* semantics from Example 3.9).

Suppose IMP's function declaration is changed to mimic Pascal, as follows:

`func` *I*(*FP*) ~ *C*

(Note that in this language, as in Pascal, the body of a function definition is a `Command`, not an `Expression`.) When the function named *I* is called, the command *C* is executed. Inside *C* there must be one or more assignment commands of the form *I* **`:=`** *E* where *I* is the *function identifier* (this amounts to a contextual constraint, since it is not enforceable by the grammar). The *latest* value assigned in this way—prior to the function's return—is taken to be the result returned by the function call.

(i) Explain whether the domain of functions is still adequately modeled by the domain:

$$\text{Function} = \text{Argument} \rightarrow \text{Store} \rightarrow \text{Value}$$

(ii) Modify the equations defining the semantic function *elaborate* to handle such function declarations. For simplicity, **assume that recursive function calls are not supported**.

HINT: Although parameters are passed using definitional methods, the assignments *I* **`:=`** *E* to the function identifier imply that an *allocate* has been performed—somewhere!

(iii) What major implications does this single language change have for the rest of the semantics? Do not give a full semantics of the rest of the language, but describe what semantic function signatures have to change. Briefly discuss major effects on the writing of the semantic functions that will be entailed.