Computation Rules and Fixed Points

Conventional ways of passing arguments are at odds with the fixed point approach we have adopted as our means of describing recursion. Since our semantic equations are a means of describing functions, we will restrict our attention to functional descriptions here, but the problems extend to more general circumstances (e.g., passing arguments by "reference"). The terminology used in this note is that an identifier appearing as an argument in a function definition is referred to as a *parameter*, and an expression appearing in an application of a function is referred to as an *argument*.

In the context of defining pure functions, there are two alternative means of "transmitting" arguments in a function application. Most common is passing an argument *by-value* -- this means that at call time each argument expression is evaluated, and that value is used in the function body whenever there is a reference to the corresponding parameter; this is also referred to as *eager* (or inside-out) evaluation. The second alternative is passing an argument *by-name* -- this means that at call time no argument expressions are evaluated, and instead these expressions are evaluated only when in the execution of the function body, a reference to the corresponding parameter is encountered; this is also referred to as *lazy* (or outside-in) evaluation Each of these approaches has been adopted in various languages -- eager evaluation in e.g., C, ML, and Java, and lazy evaluation in e.g. Algol 60, Miranda, and Haskell. There are actually several variants to these approaches, but in our pure functional context they are not significant to the defined function (we ignore efficiency concerns).

We illustrate the effect of these issues with an example. Consider the definition of a function (over all integers Z)

$\mathcal{P}$: f(x,y) = **if** x>10 **then** x-1 **else** f(x+2, f(x,y+1)).

With this definition, different functions are obtained depending whether arguments are passed by value or by name.

Consider f(9,1)

• by-value/eager/inside-out

$f_e(9,1) = f_e(11, f_e(9,2))$

$= f_e(11, f_e(11, f_e(9,3)))$

$= f_e(11, f_e(11, f_e(11, f_e(9,4))))$

= … = undefined

• by-name/lazy/outside-in

$f_\ell(9,1)$ = **if** 9>10 **then** … **else** $f_\ell(9+2, f_\ell(9,1+1))$

$= f_\ell(9+2, f_\ell(9, 1+1))$

= **if** 9+2>10 **then** (9+2)-1 **else** …

= 10

Hence $f_e \neq f_\ell$. Using our partial order notation, we can express the definition of f_e as

$f_e(x,y)$ = **if** x>10 **then** x-1 **else** Ω.

Assertion: Eager argument transmission is not a “fixed point rule” – that is, the function defined recursively using by-value need not be the least (or any) fixed point.
Proof:
The definition $\mathcal{P}$ given above can be seen to serve as a counter-example. Using the descriptions

$f_e(x,y)$ = **if** x>10 **then** x-1 **else** Ω, and

$[\mathcal{P}(f_e)]\ (x,y)$ = **if** x>10 **then** x-1 **else** $f_e(x+2,f_e(x,y+1))$

we can verify that function f_e is not a fixed point of program $\mathcal{P}$, much less the least fixed point. This is an immediate consequence of the observations:

- $f_e(9,1) = \Omega$, and
- $[\mathcal{P}(f_e)]\ (9,1) = f_e(11, f_e(9,3)) = 10$.

This is a disturbing outcome since it means that when we examine the definition

$\mathcal{P}$: f(x,y) = **if** x>10 **then** x-1 **else** f(x+2, f(x,y+1))

we cannot think of the instances of 'f' appearing in the function body on the right as denoting the same function as the instance being defined on the left!! Using one name for two different things is an invitation to confusion and to counter-intuitive results.

Theorem: Lazy argument transmission *is* a “fixed point rule” – that is, the function defined recursively using by-name is the least fixed point.

We will not prove this result – the development can be found in the Manna book on our reserve list. However, the basic reason for this result is that the conditional (if-then-else) used in function definitions is “lazy” – that is, either the then-part is evaluated and the else-part is not, or vice-versa. We already observed this effect above where $f_e(9,1)$ is undefined while $[P(f_e)]\ (9,1)$ is 10.

So the least fixed point approach conforms to lazy evaluation, but not to eager evaluation. Thus in numerous cases of practical interest where eager evaluation is used, the least fixed point idea does not immediately capture the computation. This does not conflict with the use of the fixed point concept as the definitional basis for recursion in our semantic functions. For eager evaluation, the semantic function definitions simply incorporate a description of the required computational behavior. We had an example of this in our text describing eager evaluation of procedure arguments in Pelican, where the application of the evaluate function to the argument preceded the application of the execute function to the procedure body. Finally, note that if a function is total (defined for all arguments), then eager and lazy evaluation produce the same definition so in these cases the fixed point analysis applies directly.