A Good Function

Overview

This page summarizes the qualities of a "good" function when building an Infusion application. It was originally posted by Antranig Basman to the fluid-work mailing list in January 2015.

A "good function" is:

1. A pure function, one without side-effects whose return value depends only on its arguments
2. Available at a stable public name in a global namespace
3. Performs work only linearly proportional to the size of its arguments (considered as structures)

The first criterion has long been of recognised importance to Computer Science. The second criterion has been increasingly recognised by us in Fluid, especially in our more mature work after, say, mid-2009. The third criterion is only more recently becoming recognised in its value an implications.

The prime goal of all our work is transparency; primarily, transparency of the systems that we create from the point of view of our integrators, users, and the tools that they use (including any UI of the resulting system itself). Without transparency, we can't deliver on accessibility, which requires all the behaviour of the system to be at the user's disposal.

Criterion 3 is specifically a reaction against the very popular notion of a "computationally complete" or "Turing-complete" language or system. By definition, such a system can "compute any computable function" and therefore also by definition one that is as least transparent as possible. A system which can "compute any computable function" could indeed do absolutely anything, including consuming unbounded resources of time and space, as well as possibly never terminating. These are "classically undesirable" features of any user system - but together with these undesirable properties come other, analogous problems of the kind we are more interested in in our communities: a system based on this kind of code is hard to analyse, hard to transform and adapt for other purposes, and hard to author by non-specialists. I think it's only more recently becoming clear that addressing such problems from the angle of strict bounds on complexity (and thus on coding styles) is a productive way of finding our way to meeting these more precarious and indefinable values through relatively tractable and hard-edged, measurable properties.

This issue came up since I was recently talking over with Colin a "transforming function" I had written for our OAuth-secured flow manager's test fixtures, named "gpii.test.cloudBased.oauth2.buildDisruptedFixture".

There are a few things about this function which are rather clunky and undesirable - although it does at least embody a model that "Infusion IoC is a JSON dialect that has JavaScript as its metalanguage", implying that we at least have a metalanguage, and it is a commonly-understood and widely used one—as opposed to other environments, which either have no metalanguage or have an ad hoc one written in an unintelligible dialect such as "aspects". Eliminating those kinds of systems leaves only LISP in the same camp as us, which astonishingly succeeded in being a language which almost precisely coincided with its own metalanguage.

Why do we need a metalanguage of this or any kind? It is because this is the language that our authoring tools will one day be written in. But for the time being, if we left the issue by saying that "our metalanguage is JavaScript", it could be argued that we've made our problem substantially worse - JavaScript, being Turing-complete is itself very hard to author effectively and transparently. So I was left pondering, "What is that, if anything, that is actually 'good' about functions such as 'gpii.test.cloudBased.oauth2.buildDisruptedFixture'?, and could initially only come up with points 1 and 2 above, which didn't seem terribly promising.

However, putting condition 3 into the mix does seem to leave us somewhat further along. For a start, there are vastly fewer functions that satisfy condition 3, and there are also vastly fewer language features and coding styles which are permissible in them. They are able to use all the basic operators of JavaScript (member access, arithmetic/logic operations, etc.), as well as one of the control flow elements (if/then/else). However, they must obey strong restrictions in order to be "obviously provable" from a static analysis that they meet condition 3. These restrictions include:

1. Any looping primitives must be "structural" - that is, take the form of a fluid.each or fluid.transform call applied to their direct arguments in order to be obviously "non-work-magnifying" - that is, not doing work which grows faster than the size of their arguments - this is referred to as the use of structural recursion (as opposed to "generative recursion")
2. Any functions that they call must also obey the same "good function" conditions 1-3 listed above

It is sort of promising that the space of these "good functions" appear to be algebraically closed in this way. That is, a "good function" which obeys these stylistic restrictions and only calls other "good functions" is itself provably "good." This suggests that we are identifying something potentially stable and meaningful.

Model transformations

The classic embodiment of "good functions" are the transform functions attached to our model transformation framework documented at http://wiki.gpii.net/index.php/Architecture_-_Available_transformation_functions - these functions are all classically "good" but unfortunately our system is still far too primitive to allow functions like "gpii.test.cloudBased.oauth2.buildDisruptedFixture" to be encoded in it. Even if we could encode it, the resulting structure would still be quite verbose and hard to read/author with our current encoding. Eventually our transforms should be powerful and usable enough to be the "expression of choice" for essentially all "good functions" such as this one. However, it is helpful to identify the way in which this function and others might be "on the way to becoming good" ... as well as starting to shrink the language space in which our "metalanguage" is to be expressed.

Notes
a) Note that the "cost criterion" in 3 could be relaxed slightly (or sometimes) to permit functions of $O(n \log n)$ in input size, which would gain us access to elementary sorting and searching algorithms. But the original form of the criterion is the one which could most straightforwardly be checked by authoring tools, allowing the location of "good" and "non-good" functions of various levels of quality to be immediately identified.

b) Note that criterion 1 especially in JavaScript should be clarified to read "and one that does not modify the structures passed as its arguments"

c) None of this discussion directly touches the issue of how we "do" handle mutable state when we meet it, which is a separate though linked discussion - the link being our model transformation system -