Writeup of Chat with Clemens Klokmose 23-11-15

Thanks to Clemens for another great chat - here's a writeup and some thoughts for those who couldn't make it.

Firstly, top news - Clemens' Webstrates paper has won BEST PAPER AWARD at UIST 2015! CONGRATS TO THER MAXX to Clemens for this brilliant result - http://uist.acm.org/uist2015/awards - paper available here: http://www.klokmose.net/clemens/wp-content/uploads/2015/08/webstrates.pdf

We started off with Clemens describing current work on webstrates - he is introducing an element named a "Facet" - a DOM node whose purpose is to house a script. We should try to have a demo or see some examples of this at next chat. This started us pretty quickly back to our conversations re the RES POTENTIA vs RES EXTENSA - Clemens asked me to summarise "those aspects here which are separated which are commonly entangled" which I'll try to do:

So, here's a quick recap of our existing planned architecture in this area -

POTENTIA I:
The "catalogue of instantiable artefacts". In Infusion's terms, the registry of "component defaults". The collection of the "kinds of things that might exist", together with their ability to entail each other's existence (for example, as subcomponents). This, we currently have.

POTENTIA II:
The "registry of aligned user expressions". That is - a record of those things which have been asked to be brought into existence by user expressions, and their desired locations. This we do not yet have explicitly in Infusion. Components are currently brought into existence directly through function calls - and no separate record is made of their names and arguments.

EXTENSA:
The runtime of live component (object) references.

Future framework workflow:
Authors interact with the system by issuing updates to POTENTIA I and POTENTIA II. The framework then has the mission to bring the EXTENSA into consistency with these expressions. Updates to the EXTENSA take place in "fits" (transactions) at the start and end of which the extensa is self-consistent as well as consistent with both potentia. Updates to the potentia will be reverted cleanly if they cause an error during the corresponding extensa update.

Authors may also interact with the extensa directly - which updates we expect to be captured in model material in the extensa. As we established in earlier chats, potentia and extensa coincide in the area of models (all potentia are actually models), and any such updates are automatically hoisted up into POTENTIA II. Note that, since any updates perceived by any extensa whilst it is "extended", due to be captured back up into its potentia when it is time for it to be sent to bed (that is, pushed back up the authorial chain), can only be as the result of what could be called "observation" - that is, updates received to its models via "materialization strategies" with respect to things outside the system - which could more simply be termed "observation" (or measurement). This, amazingly, means that our somewhat wonky-seeming terminology is not actually entirely at odds with Whitehead's original conception as presented by Kauffman in his NPR article publicising the res potentia.

So, Clemens asked me to try to state more clearly "what it is in this area that is mixed together in traditional programming languages". Which, initially, can be described as "the expressive statements which cause things to come into existence, and the resulting things themselves".

Here's a programming language statement - written in current Infusion-style (which also suffers from this problem) so that we can correspond it better to the future:

```
function makeThing () {
    var thing = fluid.thing({arg: "value"});
    ...
}
```

So, the "problem" here is that both the INTENTION and the RESULT are private - the fact that fluid.thing has been requested is knowable only to the runtime at that moment, and the result is stored in a inaccessible area - a function scope.

In the corresponding "Nexus-type" expression, we might say:

```
fluid.construct(["thing"], {
    type: "fluid.thing",
    arg: "value"
});
```

This, rather than a direct instruction to construct something HERE AND NOW, could be taken simply as a registration into POTENTIA II (although it might actually be honoured on the spot too - as it is in the current framework).

Now, this expression has a well-defined global address ("thing" at the global tree root), and it is more imaginable that the framework's machinery can take a record of what the user's expression was that is expected to be there (the 2nd argument to fluid.construct captures this). Authorially then, this record could be acted on directly by this user or other users, rather than having to interact with its end-product, the runtime component, instead.

Comparison with other kinds of system:

i) Imperative, or even, compiled systems
Such as those written in C/C++, Java, JavaScript etc. The problems are as above as shown in "standard Infusion" - intention is "pushed" into the system by a unidirectional process and can't be recovered. In a compiled language the situation is even worse since the user's expression exists at a completely different logical level (a text file quite separate from the executing runtime) and can't even be recovered in theory.

ii) Live programming systems
Such as those written in Self, Smalltalk etc. -
User intention CAN be recovered, but it can't be separated from the running system itself. This means that the system can't be worked on "by any other tools than itself". It's hard to write tools for the system since it is impossible split off a description of part of the system into a standalone document. Webstrates is a special case which we need to treat separately - its runtime is in theory amenable to tools since it is a DOM/extensa - the problem lies in its lack of POTENTIA I, and attempts to improve this will degrade the purity of its extensa, which was never architected to act as a potentia.

iii) Functional programming systems
At one extreme, type-rich systems such as Haskell, ML, etc. and at the other, type-poor systems such as Lisp/Schema etc.
These mix together the worlds of user expressions and runtime using an often complex workflow - work is done in "stages", perhaps as part of a partial compilation process which does part of its work as information becomes available from the user and environment. However, the upshot is still the same - since the fundamental structuring device is still the same, the FUNCTION.

This leads to the central observation which positions the FUNCTION as the FIRST EVIL OF COMPUTER SCIENCE. This is because a function, necessarily by its nature, acts by a process of ERASURE. In every language, in every case, the workflow is the same - a set of ARGUMENTS are evaluated, and then submitted to some EXECUTION MACHINERY (the function's implementation) - and then the arguments are ERASED from the system and replaced by the RESULTS, the RETURN VALUE OF THE FUNCTION.

This makes it clear that any system based on the FUNCTION as the central structuring device can never be the embodiment of a successful user programming idiom, since the user's original intention, erased by the function execution process, can never be recovered in order to be acted upon by other collaborators.

I hope this moves the discussion forward a bit although there's still clearly a huge amount to be said/understood.

Another topic we dealt with on Monday was a great question from Clemens about the extent to which "merging", the central structuring device of Infusion, was intended to be/capable of being directly exposed as a user programming idiom. I said that it was intended to be directly exposed - since it is a primitive that can easily be explained through a variety of idioms - I gave a interface idiom/metaphor imagining a set of stacks of coloured shapes, which might occlude each other in irregular ways, leading to different parts of them being exposed in the full structure. We're familiar with "occlusion" as a visual idiom, and the structure of the JSON configuration is sufficiently plain that we expect the direct action of occlusion (shapes/objects "nearer" to the user than those "behind" them) to explain the resulting structural shape. We could even imagine colour, say, acting as a kind of "provenance" in the resulting merged structure - for example, a piece of "yellow" exposed configuration would betray its influence from an original user expression which was also "yellow" and not occluded by something in front of it.

Anyway, more soon - reflections/feedback/pushback on the above welcome - our next call is scheduled for Monday 7th December, 3pm CET -

Looking forward!

Antranig