Harmonious Authorship from Different Representations (Work in Progress)

Antranig Basman[1], Colin Clark[1], and Clayton Lewis[2]

[1] Fluid Project, OCAD University, Toronto, Canada
amb26@ponder.org.uk / cclark@ocad.ca
[2] University of Colorado, Boulder
clayton.lewis@colorado.edu

Abstract. We describe the Infusion system, which is a library, language system or *integration domain* implemented in JavaScript, as well as a mentality and model for thinking about the expression of end-user applications. We promise that this system will bring together the worlds of different kinds of users engaged on different tasks at different times, and allow them shared authorial access to the same artefacts which are presented to each in a notation appropriate to them.

Keywords: POP-II-A. individual differences; POP-II.C. cognitive dimensions, data flow, visual languages; POP-I.A. programming economy; POP-I.C. web;

1 Introduction

Differing notations bring different affordances — and are suited for different audiences and different tasks(Blackwell & Green, 2003). For example, some notations, with low *viscosity* might be appropriate during initial development of a new system, whilst others, with few *hidden dependencies* might be more appropriate during maintenance. Some, with powerful *abstractions*, might be suited for experts, whilst others, with good *visibility* might be better suited to novices or end-users. Traditionally, the choice of notation for a particular task implies more than a skin-deep commitment to a particular style of representation and way of working. For example, the choice of a conventional programming language such as Java or Haskell, based on the core representation of a stream of textual characters forming its source code, strongly limits the kinds of alternative notation which can be provided for other tasks and audiences. Correspondingly, the choice of a visual programming idiom such as Scratch(Maloney, Resnick, & al., 2010), Blockly(Google Developers, 2015), or Max/MSP(Cycling 74, 2007), cuts off the potential for engaging with audiences familiar with the power of traditional text editors and IDEs.

Our work for some years at the Fluid Project has been to construct a system, Infusion, which offers not just a single "middle way" between such extremes of notation, but also schemes for navigating between different notations in which "the same artefact" might be expressed. This will naturally involve some compromise between the needs of different audiences — in our examples above, the gap between the notational worlds of the visual and non-visual is "not simply a matter of notation". The differences between the structure and referential style of, say, a Java program and a Scratch program are too profound to allow one to be usefully transformed and represented in the style of another. Every system constructed so far strongly betrays its heritage of having been initially conceived as a "visual" or "textual" system from the start, and no amount of tooling or authoring support can ever bridge such a divide.

In development since 2007, Infusion still counts as a "work in progress" — although several aspects of its nature have become clear over the years, and our group uses it every day for their real-world programming tasks, there is a lot of ground still to cover in making its idiom consistent in all areas, and tackling the tough architectural challenges related to managing "programs" which can be authored live by a distributed group of authors using dissimilar representations. Work on visual and remote tooling for the system is only just beginning. Infusion is a system inspired by the web and designed for the web — it is library of standard JavaScript which can be included in any modern browser, and harmoniously coexists with applications written in standard markup and widgets. It is also suitable for standalone JavaScript runtimes such as `node.js`.

1.1 Quick guide to the structure of an Infusion application

An Infusion application consists of blocks of *configuration* expressed in the JSON serialization format, together with a collection of *globally namespaced*, ideally *pure function* expressed in the *base language*, which in our language is JavaScript. The configuration is organised as a set of globally named elements which are known as **grades**, which fulfil a few of the traditional roles of *types* in other systems, but fail to qualify in other areas. The configuration part of the system, since it consists of pure *state* which comes with a natural coordinate system, is ripe for transforming, expressing, and authoring in a variety of forms.

1.2 The role of the "base language"

The part of the design expressed as code in the base language can only be authored by software experts using traditional software tools — we need to limit our exposure in the long run to a very finite collection of well-known and well-curated instances. The need to involve a software expert in any particular design must become rare and rare. In parallel, as time passes, we expect the features used within the base language gradually become more and more impoverished — for example, we at the earliest stage abandoned the features of JavaScript used for "object" formation using prototypal inheritance — following that, the features used for forming closures for expressing higher-order functions — and are in the process of abandoning those used to achieve side-effects through assignment and then finally those used to express looping and general control structures other than conditionals[1]. The resulting "impoverished base language" will then itself become far more amenable to novices, authoring using non-traditional tools and will in essence lose much claim to identity as being JavaScript at all — the same core subset used for producing a limited range of expressions designating pure functions with no control structures beyond conditionals and structural recursion might as well be taken from any language. We expect that any dialect used for expression of actual *computations* will be ejected from the system and be treated with the same caution that is currently applied in pure functional languages for code with side-effects or performing I/O in general. We'll talk more about the *avoidance of computation* in section 2.1.

1.3 "One man's excess intention is another man's secondary notation"

A crucial requirement in order to meet our goal of harmonious authorship from different notations, is the construction of notations as free as possible from the expression of ***excess intention***. Excess intention results when the notation we have available unavoidably captures more than what we intend to express in our design. Traditional programming languages, especially procedural ones, are famously rich in excess intention — some of which are being recognised and combatted by newer notations, others of which are not. Here are two examples we have characterised:

Sequential Intention — Imperative programming languages unnecessarily force the creator to commit to an exact sequence of executed instructions, which is usually far in excess of the real requirements underlying the goals he is interested in. This is a criticism that is broadly acknowledged, and some responses to it are becoming widespread — for example, as expressed in the model of *dataflow programming*, or in *monadic* styles of packaging control flow.

Artefact Boundary Intention — Object-oriented languages force the designer into a single, exhaustive decomposition of their domain of interest into a non-overlapping collection of *objects* with well-defined names, properties, relations and contracts. However, another view of exactly the same domain by a creator with different aims, skills or preoccupations might very well decompose it into an entirely different set of entities — which, with luck, at least bear a strict hierarchical relationship with those from the first view, but perhaps may not. One answer to this kind of of issue is named within the Design Patterns community as the *Facade Pattern*(Gamma, Helm, Johnson, & Vlissides, 1994) — however, such Facades not only require new application code, but must respect existing boundaries — one cannot construct a Facade which encloses "half an object".

There are other excesses we could talk about, but each raises the same issue. In transforming from one notation to another, one must somehow capture all that is "excess" from one viewpoint with respect to another, and store it somewhere as an annotation to the structure — in exactly the same way one would be required to capture a *secondary notation* that had been attached in a notation, to preserve it during a stage of processing that was blind to it. This hugely complicates the design of tools manipulating such notations, because such notation is not in fact truly secondary — even though the author was uninterested in it, this fact cannot be deduced from the notation, and the artefact cannot function without some expression of it. The lines of code in a function body must be supplied in *some* order — some of which are invalid as expressions, some of which are valid but don't capture the author's intention, and only a few of which are actually permissible equivalent expressions.

1.4 The historical and methodological process of constructing Infusion

The bipartite model consisting of "configuration over a base language" is also crucial to the methodology by which Infusion was developed and will continue to be developed. At its inception in 2007, Infusion applications consisted of standard JavaScript code, loosely organised around records containing "options", of which the JSON configuration simply represented *default values*. This historical structure gave rise to the name by which the directives holding configuration were and still are issued to the system, `fluid.defaults`. As year succeeded to year, the power and capability of the configuration system steadily increased, and the number, diversity and

[1] "... cursed their feet, cursed the ground, and vowed that none should walk on it again" — Douglas Adams, *The Restaurant at the End of the Universe*, Chapter 10

complexity of the functions that required application code steadily decreased. At any time, the space of possible application functionality consisted of an "explained portion" which could be expressed as configuration, and the unknown "unexplained portion" which still required conventional code. Development of the framework proceeded by a scientific process whereby a generalisation was proposed that might move part of the "unexplained portion" to configuration — and was validated if we found that future designs that had not been anticipated could productively make use of the feature or configuration style. Features which were found to be used only in one or few designs, the ones that prompted the invention of the feature, were pruned from the framework.

2 Theoretical underpinning and links to existing paradigms

2.1 The role of programming languages and computational power

It is arguable whether the system we will present here is best described as a "programming language", a "framework" or some other thing — it shares in some clear characteristics of both. The best designation that we have found so far is that of an *integration domain* (Kell, 2009) — an arena for the naming and scheduling of effects, computations and their units of organisation — rather than an area in which computation is expressed directly. This issue, we feel, has long misdirected the field — since every notation which has been put into the role of "programming language" has been put under immediate pressure to demonstrate that it can express any computation ("is Turing-complete") in order to qualify for this role. On the contrary, an "integration domain", as noted by Kell, can easily be endowed with lesser computational power, and we argue, should be so. It is crucial, for example, for the transparency and responsiveness of authoring tools, that relationships between parts of a program can be determined by the exercise of limited computation power. The Self family of languages emphasise the importance of such responsiveness for the feeling of authors that "the thing on the screen is the thing itself"(Ungar & Smith, 2013) — and a system or language whose structure implies the potential for unbounded computations (for example, those of a complex type system such as ML or Haskell) directly fights this aim. Such type systems, if provided, should be an optional "bolt-on" addition to the system just for the use of a particular audience. Recent work on "gradual typing"(Siek & Taha, 2006) has tended in this direction, but so far there is little work on systems promising multiple independent, completely optional type systems for the same artefacts.

2.2 The first-class role of state, and transparent access to it

We promote the use of *transparent, publically addressible state*. The Infusion system should maintain all state, not just that of its users, but any that it maintains for its own book-keeping, in public view, with each piece of state available through an utterable public address. This is at odds with both *object-oriented* and *functional* programming, which insist that the state which the application manages on behalf of users must be hidden from view — either through *data hiding* in the former paradigm, or *prohibition of side-effects* in the latter.

Firstly, publically addressable state is the touchstone of the prevalent REST(Fielding, 2000) style of conversation or API for web applications, and this analogy has guided our development since the start. REST stands for **representational state transfer**, and each of these words carries crucial and essential meaning. Firstly that *state is represented* in the conversation, rather than opaque tokens representing mere *behaviour* or *methods* as is common in procedural or object-oriented API contracts, and secondly that it is *transferred* — that is, that the representation is an *exhaustive* summary of the state that can be used to moved it from place to place.

Secondly, such *exhaustive representation* is crucial for the *perceived liveness* of any proposed system. In terms of our discussion of the Self language(Ungar & al., 2014), for example, one fault possessed by Self as well as systems influenced by it, such as JavaScript, is the divergence between what could be called *constructional type* and *live type*. This is exemplified in JavaScript, for example, by the non-identity between the object property named `constructor`, holding the name of the creator function used to instantiate an object, and the one named `__proto__` which identifies its prototype chain. Therefore, the in-memory representation of an object incorporates many hidden pieces of bookkeeping information (*hidden dependencies*) which guarantee that the object can never be made straightforwardly equivalent with a particular serialisation of itself. Self provides the *Transporter* which exports objects to an essentially opaque format only readable by another Transporter, whilst JavaScript provides natively only for serialisation to JSON — which can't represent either variety of type information.

Thirdly, the manifest nature of public state is crucial for many of the most successful embodiments of end-user programming. For example, in the spreadsheet paradigm, the programming surface consists purely of values in a grid. Each grid element has a well-known and mostly stable *public address* which can be used to access its value. Unfortunately from here on, the spreadsheet idiom starts to fall down, since any programming directives which are issued must skulk in a "hidden world" behind each cell, unaddressable either as a whole or in part. There have been efforts to address this deficiency within the spreadsheet paradigm in particular(Burnett & al., 2001) — in general, we conclude that public addressibility of all design elements is completely crucial in order for a design to be able to exhibit good *visibility* and a lack of *hidden dependencies* when required — one can always choose to hide what might be visible in contexts where it is undesirable, but one cannot chose to make visible that which does not even have an utterable address.

2.3 A system inspired by the Web, and built for the Web — IoCSS selectors

The Web represents the most highly successful "evolved strategy" for dealing with the problem of distributed and shared authorship. Whilst it appears to fall short of what are claimed as its antecedent blueprints, for example, in Ted Nelson's elaborate hypertext system Project Xanadu(Nelson, 1982), as well as being regularly claimed as a deficient abstraction by object-oriented and functional programmers alike, we feel that there is a great deal to study, admire, and learn from the solutions and strategies that it embodies.

We referred in the previous section to the importance of the REST idiom, first used to describe the stateless and resource-oriented nature of web protocols, to the design of our system. Another successful idiom which is essential for the day-to-day running of the web is CSS, the scheme familiar to designers and developers alike for describing the styling information applied to web pages. This scheme fills a crucial role in brokering between distributed authors of "the same document" who live in different communities, with differing workflows and tools. The space of DOM elements in a web page is a "shared authorial space" which must be malleable in the face of demands of varying strength from different ends of the process (design and logic). The space of CSS selectors can be "negotiated" in that the requirement to identify a particular piece of the document could be met "opportunistically" by choosing a selector which matches it contingently and unstably, or arranging/negotiating to alter the document structure to allow a selector to match it more stably and semantically.

By analogy, we have named this system of selectors ***IoCSS***, named after the framework's role as an "Inversion of Control" system. This implies that what has been previously thought of as "an application" has been endowed with a regular but free-form *cellular* structure. In the case of an Infusion application, the *cellular unit* is the *component*, rather than as it is with an HTML document the *DOM node*. The affordances of an Infusion component are unusual set against those of typical units of software designs, given that they may be freely embedded recursively, and that further subcomponents may be injected into existing parents without their "knowledge" or disturbing the design — in object terms, Infusion components offer the possibility for *containment without dependency*, which is not possible in an object-oriented system.

Once we have the cellular structure in place, we now need some *landmarks*. In the DOM, these are provided by *CSS class names*, *tag names* and other well-known DOM attributes. In an Infusion document, these are provided by the *context names* which can be derived from the *grade names* attached to a component (which we first met in section 1.1) and the *member name* used to embed it in its parent.

IoCSS selectors are most often used with the `distributeOptions` construction which forms the framework's "polymorphism at a distance" scheme. Options material can be routed from any one source in the tree to any number of targets elsewhere in it. This scheme can be seen as similar both to *advice* offered in an *Aspect-Oriented Programming*, as well as encodings of a `diff`s between values of state held within a revision control system.

3 Examples

3.1 A small example involving relay

The *model relay system* is the configuration used to set up permanent, possibly transforming, relationships between different bodes of state. This kind of capability is also currently comprised under today's descriptions of *reactive systems*, particularly seen in the so-called *functional reactive programming* (FRP). In Figure 1, we'll set up a small system consisting of two pieces of state linked by a transforming relay, held in two different components, and then show how we can interact with it from the base language (JavaScript).

```
fluid.defaults("examples.simpleRelay", {
    gradeNames: "fluid.component",
    components: {
        celsiusHolder: {
            type: "fluid.modelComponent",
            options: {
                model: {
                    celsius: 22
                }
            }
        },
        fahrenheitHolder: {
            type: "fluid.modelComponent",
            options: {
                modelRelay: {
                    source: "{celsiusHolder}.model.celsius", // IoC reference to celsius model field in the other component
                    target: "{that}.model.fahrenheit",       // could be shortened to just "fahrenheit"
                    singleTransform: {
                        type: "fluid.transforms.linearScale",
                        factor: 9/5,
                        offset: 32
                    }
                }
            }
        }
    }
});
```

Fig. 1. Short example showing a transforming relay

To start with, it's worth noting that so far our design consists of no base language code whatever. A single function call, `fluid.defaults`, is necessary to register the configuration with the system, but in other styles of interaction, for example the *Nexus* described in section 4.1 even this can be dispensed with. Naturally we will

need to execute some further base language code to create instances of this system and experiment with them, but one can imagine that this also could be dispensed with in other visual/non-visual authoring environments which might feature, for example, a graphical "playground" in which instances can be set up and torn down by direct manipulation.

In Figure 2 follows a sample conversation that we will have with the system. To start with, we will "decorate" the base system with some *model listeners* which will react to changes in the model values and report on them. We can do this i) without further application code, and ii) without needing to modify the above definitions. After that, we will use the language-level API to trigger modifications to the values and hence the reports.

```
// Designate a "decorated variety" of our simpleRelay type which will log messages on model changes
fluid.defaults("examples.reportingRelay", {
    gradeNames: "examples.simpleRelay",
    distributeOptions: [{ // options distributions route options to the subcomponents in the tree compactly
        record: {
            funcName: "fluid.log",
            args: ["Celsius value has changed to", "{change}.value"]
        },
        target: "{that celsiusHolder}.options.modelListeners.celsius"
    }, {
        record: {
            funcName: "fluid.log",
            args: ["Fahrenheit value has changed to", "{change}.value"]
        },
        target: "{that fahrenheitHolder}.options.modelListeners.fahrenheit"
    }]
});
fluid.setLogging(true); // send any logging output to the console
// Construct an instance of our decorated tree type
var tree = examples.reportingRelay();
  // This will immediately report:
  // Celsius value has changed to 22
  // Fahrenheit value has changed to 71.6
tree.celsiusHolder.applier.change("celsius", 20);
  // Celsius value has changed to 20
  // Fahrenheit value has changed to 68
tree.fahrenheitHolder.applier.change("fahrenheit", 451);
  // Fahrenheit value has changed to 451
  // Celsius value has changed to 232.7777777777778
```

Fig. 2. Example of operating a transforming relay - output is shown in comments

This shows that the relay has set up a *lens* between the state held in the two components. The relay operates from the point of construction onwards — and ensures that the model constraint is satisfied by the initial system as well as with respect to modifications at either end of the relay. This relationship will persist until one or other of the related components is destroyed — which they might be, for example, with a call to `tree.celsiusHolder.destroy()`. As well as tearing down all relationships, this will also remove the instance from its parent, as required by the *cellular model* described in section 2.3.

3.2 The Preferences Editing Framework

As an example of a real-life view application built using Infusion, Figure 3 shows an instance of our *UIOptions tool*, itself an instance of our *Preferences Editing Framework*. This application can be dropped into any web page to allow the user to customise the presentation of the page — for example, by selecting a custom font size, line spacing, contrast colour scheme or other accessibility adaptations.

Rather than being constructed from raw JavaScript code, this UI and its substructure are composed using Infusion's model-based approach. The raw materials for such a UI consist of a *Primary Schema*, which is a standard JSON schema (see `http://json-schema.org`) describing the data types of the preferences being edited, and an *Auxiliary Schema* which is essentially a dehydrated form of Infusion component tree. This implies that integrators can mix and match different panels into their UI, and customise all aspects of the existing ones, by editing JSON documents rather than writing code. Our plan is eventually to build a direct-manipulation style self-editing view that will allow end-users and integrators to achieve this directly, in the style of the Morphic user interface tool for the Self language described in (Ungar & Smith, 2013).

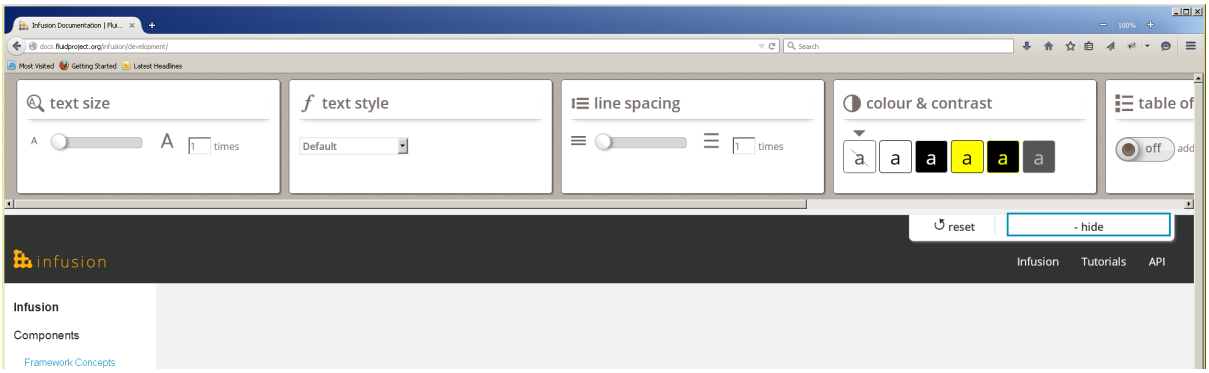

Fig. 3. Screenshot of a Preferences Editor instance built using Infusion, running on our documentation site

4 Current Work and Future Developments

4.1 The Nexus - Support for Live and Distributed Authoring

For the EU project "Prosperity4All" (P4A - see `http://www.prosperity4all.eu/`), part of the overall Global Public Inclusive Infrastructure project (GPII - see `gpii.net`), we will be developing a portable and self-contained embodiment of the framework's facility as an "integration domain" named the *Nexus*. A Nexus consists of a standard instance of the Infusion system, supplied with HTTP and WebSockets endpoints which allow all of the core lifecycle actions to be operated remotely as well as via local function calls. For example, fresh defaults can be issued into the system by an HTTP PUT to `defaults/<component.name>`, or a fresh instance constructed at a particular path by an HTTP POST to `construct` . Also, one can enter into a persistent conversation with any model in the tree by opening a WebSockets connection to start a bidirectional flow in which outgoing model changes are reported in a live stream, with the system responding to incoming change notifications on the other leg of the connection.

Since new "types", instances, and their relationships can be introduced into the system by pure web conversations with payloads of pure state, we have an ideal environment for sources and sinks of state to discover each other, and set up (possibly transforming) relationships between their respective bodies of state. To the extent that this can be done without application code, this can be done freely by all kinds of citizens, although it requires for its setup that the sources and sinks themselves have been provided with relevant adaptors of their behaviour to the *integral state model* as well as the system to have all the relevant transformers already built into it.

We plan to use this system to construct novel aggregations of raw devices to form *accessibility technologies* (ATs), which will be able to adapt any means of input that a user finds appropriate onto any addressable target "application" or other sink of state such as an output device. The decomposition of updates from a text buffer into constituent Nexus messages will also be useful in other environments. In working with the Flocking(Clark, 2015) system for audio synthesis on the web, we plan to produce a system which will close up the gap between the nature of *performance* and *score* — by treating both as harmoniously cooperating elements on a common footing in a sea of state.

References

Blackwell, A. F., & Green, T. R. (2003). Notational systems - the cognitive dimensions of notations framework. In J. M. Carroll (Ed.), *HCI Models, Theories and Frameworks: Towards a Multidisciplinary Science.* Morgan Kaufmann.

Burnett, M., & al. (2001). Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming*, *11*(2), 155-206.

Clark, C. B. D. (2015). Flocking: Creative audio synthesis for the web. Retrieved from `http://flockingjs.org`

Cycling 74. (2007). Max/MSP: History and Background. Retrieved from `http://web.archive.org/web/20090609205550/http://www.cycling74.com/twiki/bin/view/FAQs/MaxMSPHistory`

Fielding, R. (2000). *Architectural styles and the design of network-based software architectures* (PhD thesis). University of California, Irvine.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley.

Google Developers. (2015). Blockly: Language design philosophy. Retrieved from `https://developers.google.com/blockly/about/language`

Kell, S. (2009). The mythical matched modules: overcoming the tyranny of inflexible software construction. In *OOPSLA '09 proceedings of the 24th ACM SIGPLAN* (p. 881-888). ACM.

Maloney, J., Resnick, M., & al. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, *10*(4).

Nelson, T. H. (1982). *Literary machines.* Mindful Press.

Siek, J. G., & Taha, W. (2006). Gradual typing for functional languages. In *Scheme and functional programming* (p. 81-92). ACM.

Ungar, D., & al. (2014). The Self Handbook. Retrieved from `http://handbook.selflanguage.org/4.5/`

Ungar, D., & Smith, R. B. (2013). The thing on the screen is supposed to be the thing itself. Retrieved from `http://davidungar.net/Live2013/Live_2013.html`