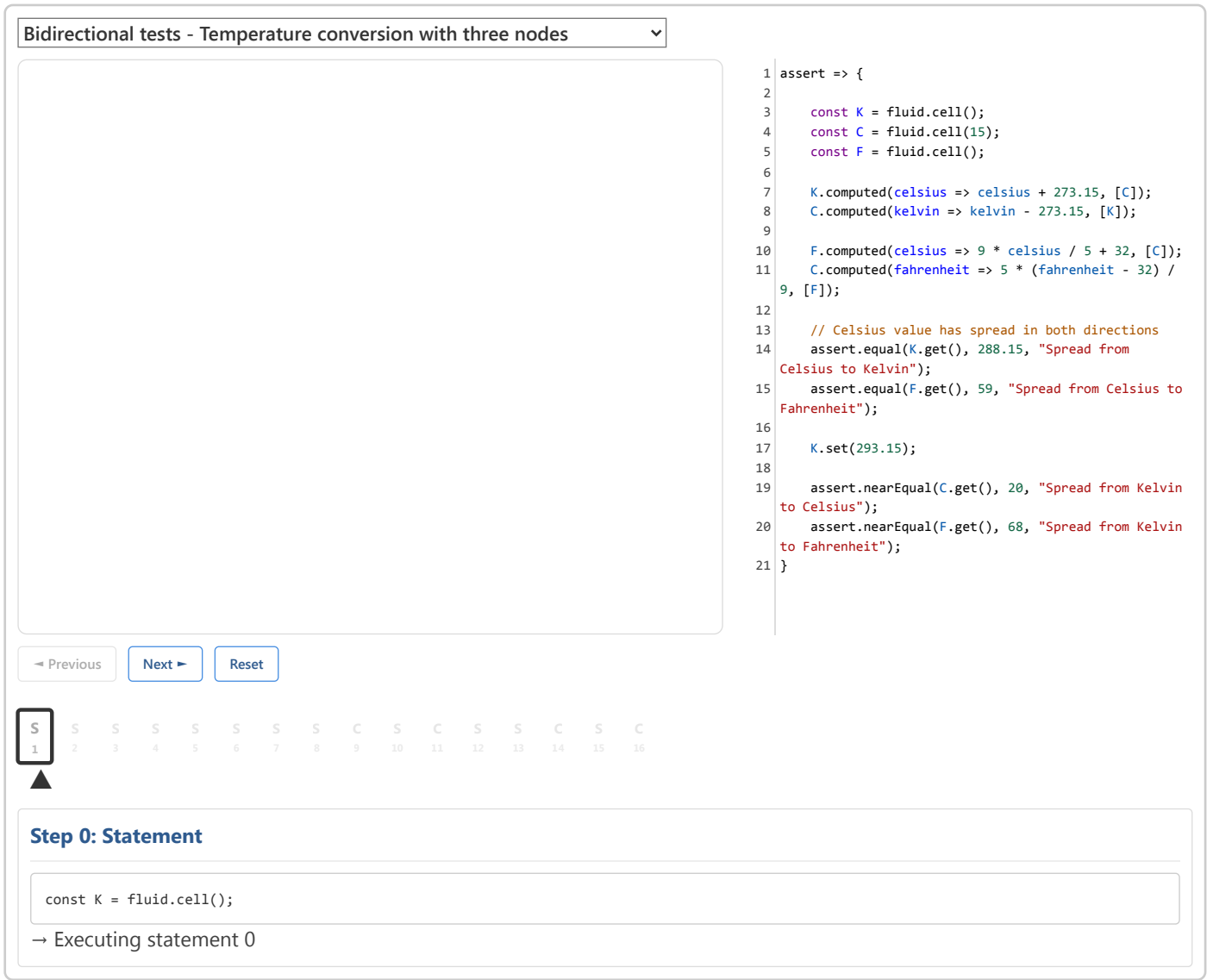
fluid.cell: A reactive implementation supporting malleable substrates – February 20, 2026

Submission to the March Substrates 2026 Workshop, co-located with <Programming 2026>.

This is a static version of the posting, with the live version is available at https://ponder.org.uk/post/2026-02-20-reactivity-for-malleability.

Abstract

This post explains why reactivity is a key requirement for openly authored, malleable substrates, and motivates `fluid.cell`, a reactive library designed to support the needs of these substrates. It preserves the good properties of widely available commodity reactive systems whilst expanding their domain. I explain the core reactive competencies of glitch freedom and early cutoff, and how `fluid.cell` delivers these competencies whilst supporting reactive graphs with bidirectional arcs, asynchronous propagation of reactive updates and allows the cause of these updates to be tracked to their source, and why these expanded capabilites are vital to support successful substrates. The post includes a interactive visual demo of the reactive library on some simple test fixtures.



For explanation of colour coding and some notes, see the demo key below.

Why reactivity?

Why do we need reactivity in a substrate? In my vision of an openly authored, malleable substrate, its interface and capabilities need to adjust dynamically to the gestures of the authors. As well as reflecting its change of function as it is built and maintained, the substrate also needs to reflect the needs of different audiences, sometimes simultaneously.

Recently I have been working with a “commodity” reactive system, preact-signals, and part of my journey with that system was described in December 2025’s Understanding Reactivity posting. I learned a lot from these commodity systems about what valuable regularities and guarantees a reactive system needs to support, and also about the limitations of these commodity systems, some of which are fundamental, and some incidental. This learning fed into the development of `fluid.cell`, a reactive library designed to support the needs of a malleable substrate; it preserves the good properties of commodity reactive systems whilst expanding their domain.

Much of the discussion here is anticipated by my 2025 Substrates Vision Statement which calls for many of the properties of the reactive system which has now been built. Documentation for `fluid.cell` is available on this site, and a live demonstration can be seen above.

Reactivity for disclosure

Bridging the gap between use and change — following Lennart Lövstrand’s notion in his 1990 Pressing the issues with buttons that “it should be as easy to change the environment as it is to use it” — could sound straightforward, but is actually a multi-faceted, socio-technical issue. Central to its difficulty is that *not all audiences will want this gap bridged*, and few will want it bridged all the time. This implies that these malleable capabilities need to lie latent within an ordinary-looking artefact, cheaply and fluidly ready to spring into existence when required. This is one of the key loci at which a substrate needs to deploy reactivity. I’ve referred to this site of adaptation under the heading of disclosable computing — those users who want to simply *use* an artefact shouldn’t be bothered by the fact that it is malleable. Otherwise highly successful substrates such as Boxer and the Lively Kernel are hampered by their lack of graceful support for disclosability. A Boxer interface always exposes all of its authoring capabilities, and a Lively interface, whilst it can be “locked down” still exposes this potential in a visually and technologically distracting way.

Hyperclay is an interesting recent HTML-based system with a clean approach to disclosure — it features a “Save-Strip-Restore” cycle that injects and removes interface elements related to malleability. However it is not a reactive or generically adaptable system.

Reactivity for liveness

Apart from the disclosure process, it’s highly desirable that reactivity is available throughout the lifecycle of a substrate. Closely aligned with the disciplines of substrate-oriented and malleable programming is the discipline of live programming. To maintain the impression that The thing on the screen is supposed to be the actual thing (David Ungar, 2013), the system should adapt smoothly and immediately to whatever gestures the author has made to modify it. This contributes to the impression that the substrate represents a material under the author’s direction rather than having a hidden realm “offstage” where important things happen. Josh Horowitz in his 2024 Technical Dimensions of Feedback has surveyed many of these dimensions, including “reactivity” in the everyday sense as experienced by users rather than a specific technical phenomenon.

Fine-grained and coarse-grained reactivity, push vs pull

Reactivity is a poorly-characterised phenomenon, coming with many kinds of granularities. A central source of confusion has been that the extremely popular React framework is not itself reactive in the strongest sense[1]. Subsequently, the distinction between “fine-grained” and “coarse-grained” reactivity emerged, in order to account for the failure of some frameworks to support graphs built of autonomously reactive values. A further axis of confusion is between “push-based” and “pull-based” reactive systems. The former are sometimes called “stream-based” systems and since they also inherently suffer from glitches, which I describe later, they lie outside our interest.

Substrate nature must coexist with imperative and functional backgrounds

It is possible to take the purity of a substrate’s design too far. A successful substrate needs to coexist not only against the background of what imperatively or functionally constructed software there already is, but also with the computational literacies that users already have. Pushing too rigorously for complete homogeneity, for a substrate constructed of a single material supporting an operation from a single paradigm, will produce a system with ergonomic problems. For example, Boxer and Subtext are systems with an admirable and total homogeneity, yet their primitives are so rigorous as to not permit ordinary algebraic expressions such as *(A * B) + (C * D)* to be idiomatically expressed. This could be considered an example of what diSessa (1985) in A Principled Design for an Integrated Computational Environment describes as the “formalist bug” of providing a too sparse set of primitives from which to build all functions.

A successful substrate will cover and re-express some of the same ground as existing computational systems, but will not intrude on semantics that the public has already mastered at a grade school level.

`fluid.cell` offers reactive competence for substrates

Previous incarnations of Infusion, my substrate supporting open authorship, imagined that we could apply a relatively coarse-grained reactivity[2], but work over the last decade bringing fine-grained reactivity to numerous JavaScript frameworks such as preact-signals, solid signals, Vue signals, etc. has shown that these systems can be highly performant, principled and capable.

The community's wisdom has established two main properties of reactive competence that the best of these commodity reactive systems can enjoy. These have gone by various names over the years, but I'll adopt names for them which have become pretty well-attested from high-profile treatments - *glitch freedom* and *early cutoff*.

However, big gaps exist between even the best-of-breed of such "commodity" reactive systems and the requirements of a malleable, openly authorable substrate.

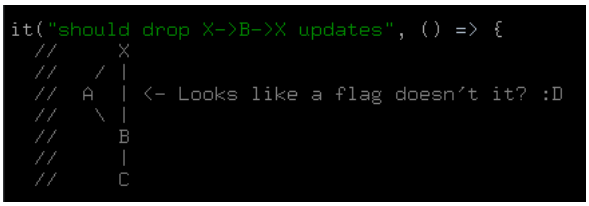
I've bridged these gaps with a fine-grained reactive library, `fluid.cell` which is specifically tailored to support open, malleable systems. Whilst it will be used to replace preact-signals as the underpinning of Infusion, `fluid.cell` is a self-contained small library with clearly explainable guarantees and capabilities which could support the development of other malleable systems too.

Firstly I'll describe the two main properties constituting reactive competence which I have distilled from these commodity systems, and then I will explain how `fluid.cell` greatly extends the domain over which it provides these guarantees compared to commodity systems, to meet the needs of substrates. Some of the journey towards this implementation is described in my work in progress posting understanding reactivity.

Two pillars of reactive competence

Glitch freedom

Glitch freedom is a consistency property which has gone by various names as it was independently rediscovered by many workers over the last decades. The property is about dataflows which diverge and then reconverge — it is essential that only one update is received by a node at the top of a "diamond" of dataflow convergence, reflecting data which has been drawn from a consistent data horizon, rather than a mixture of stale and fresh data. The diagram below shows the simplest possible graph which might glitch, with a core of 3 nodes, with a short path directly from X down to B, and a longer path from X to B via A. It's essential that when X updates, we don't happen to activate the arc X->B first, but make sure that we provide a consistent horizon of data to B by updating A from X first.



The simplest possible graph that might glitch, made of 3 nodes

An early and thorough treatment was in Kenny Tilton's Cells system for CLOS, which in 2001, 5 years before the coining of the terms "glitch" and "reactive programming", had already solved the problems of reactivity embedded in imperative programming languages. Kenny named this property "data integrity" which I quite like and think it is a shame it didn't stick. Jonathan Edwards named a closely related property as Coherence in 2009.

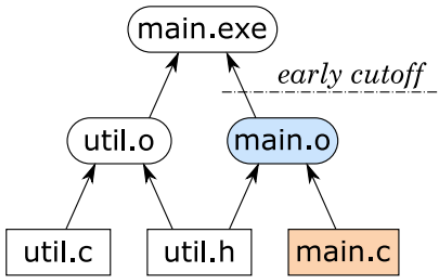
The naming of the property which has taken hold came in Cooper & Krishnamurthi's 2006 FrTime paper which I cover in my archaeology of glitches posting.

Early cutoff

The other pillar of reactive competence is an efficiency guarantee, rather than a consistency property. It's essential that the reactive system does not do a certain kind of "obviously unnecessary" work — if a reactive update leaves the value of a node unchanged, that node's "update" should not be used to schedule further updates of nodes which might depend on it.

This property is less hard-edged than glitch freedom since in practice there is a huge sliding scale of kinds of "unnecessary work" which a sufficiently clever reactive system could avoid. You can consult treatments like Robert Lord's How to Recalculate a Spreadsheet and Jane Street's Seven Implementations of Incremental for some of the fruity possibilities, but basic early cutoff support represents a straightforward everyday level of competence that every reactive system should aspire to.

The diagram below shows a canonical situation, taken from Mokhov, Mitchell and Peyton Jones' 2018 Build Systems à la Carte which is not ostensibly about reactive systems but build systems, but in practice the domains (as the paper argues, gearing through the primordial reactive system, Excel) are much the same. A user has changed a source file, `main.c`, in a way which leads to no change in the produced object file `main.o`, perhaps by editing a comment. A competent reactive build system should spot this, and, via early cutoff, not then go on to rebuild the executable `main.exe` once it sees that `main.o` is unchanged.



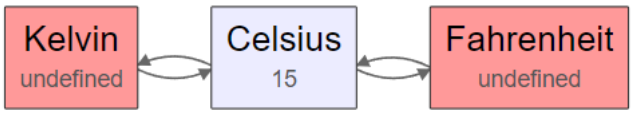
Early cutoff in a simple graph of C language artefacts

This might seem to be a quite basic variety of competence but in fact several popular reactive systems, such as knockout and Adapton, were created without it.

Supporting open substrates

Whilst maintaining the two pillars of reactive competence, glitch freedom and early cutoff, here are the new use cases and capabilities which `fluid.cell` supports in order to support an open substrate:

Support for graphs with bidirectional arcs, malleably authorable



A reactive graph with three nodes, two bidirectional arcs

This diagram shows a reactive graph with three nodes connected by two bidirectional arcs, of the kind that is impossible to support in a commodity reactive system. In this context it represents a toy domain of a temperature conversion app holding temperatures in three scales, of the kind described in Harmonious Authorship from Different Representations (Basman et al, 2015). Whilst niche, this is already a simple kind of domain that should be natural to represent rather than require peculiar contortions or asymmetries (such as, for example, use of Vue-signals' notion of a writeable computed).

There are plenty of examples in the user domain of why it might be idiomatic to support such structures — for example the seminal Cooper & Krishnamurthi 2006 FrTime paper mentions an example of RGB and HSV views in a color selection window, but there are more fundamental reasons in an openly authorable system. Such an open system must always be considered *authorially incomplete*, simply a projection on behalf of one or more authors out of a larger design, and hence cannot subscribe to the core religion which underlies much modern systems design of a *single source of truth*. Inhabiting a pluralistic informational space implies there can be no privileged node from which dataflow only proceeds outwards.

Handling bidirectional arcs requires a fundamental change in representation of the reactive graph with respect to all commodity implementations, which opportunistically fold together a "node plus edge" structure into a single data structure on the assumption that each computed edge uniquely corresponds to a node, and that each node has only one incoming edge. Changes are also needed in many parts of the algorithm which traverse and invalidate the graph.

For example, in a traditional reactive library, e.g. preact-signals, a `Computed` signal "is-a" `Signal` and directly encodes the `_sources` nodes that it depends on:

```
declare class Signal<T = any> {
        _value: unknown;
...

declare class Computed<T = any> extends Signal<T> {
        _fn: () => T;
        _sources?: Node;
...
```

and it is used as so:

```
    const a = signal(1);
    const b = computed(() => a.value + 1);
```

In `fluid.cell` there are separate node and edge structures, respecting the fact that a computed node may be computable along multiple edges:

```
@typedef {Object} Cell
@property {any} _value - The current value stored in the cell.
@property {Edge[]|null} _inEdges - Array of incoming edges which could update this node

@typedef {Object} Edge
@property {Function} fn - The function to be called to compute the value
@property {Cell[]|null} sources - Sources in reference order
```

and it is used as so:

```
    const A = fluid.cell(1);
    const B = fluid.cell().computed(a => a + 1, [A]);
```

Note that because relations are built up separately from the workflow of building up cells, it is possible to decouple the process of loading or defining parts of the substrate from building up relationships amongst those parts. In this simplest case, for example, one could have written

```
    const A = fluid.cell(1);
    const B = fluid.cell();
    B.computed(a => a + 1, [A]);
```

which is essential when dealing with cells whose relationships are cyclic (see the bidirectional samples at the head of this page) or not knowable by the same author who defined the cells.

Agnostic to asynchrony

The issue of whether a value is available right away, or requires an asynchronous I/O operation to fetch it, is one that is guaranteed to be absolutely unfathomable to the end user, yet to a typical stack-hugging developer is one of a frightful viral nature. Bob Nystrom admirably sums up the situation in his What Color is Your Function posting, explaining the viral nature of asynchronous operations in code expressed in functional or imperative forms.

Modern language features such as `async/await` provide close facsimiles to the user that asynchronous code works like synchronous code, but the analogy breaks down pretty quickly, requires foresight, and, as Bob explains, a switch from a synchronous to asynchronous source of data must cascade all the way up a chain of consuming functions.

Reactive systems, which by nature operate on values as they become ready, promise to free the user from having the issue of asynchrony in mind, which is usually irrelevant to their domain. However, very few reactive systems have so far delivered on this promise by putting asynchronous operations on a level footing with synchronous ones. Ryan Carniato's in-progress rewrite of solid signals is the notable exception, and `fluid.cell` adopts with thanks his test cases verifying reactive competence in the face of asynchronous propagation, which our implementation passes.

Accounting for cause of updates

In the background section we explained that whilst moving beyond the "execution paradigm" which both imperative and functional idioms tie us to, we must coexist with them gracefully as well as respecting the computational literacies that users already have.

However, a successful substrate will also need to offer equivalent guarantees, and especially, equivalent capabilities for explaining the state of the system, as we got from other paradigms. Reactivity has the potential to replace imperative and functional systems with far more malleable and democratic alternatives, but without providing the equivalent reassurances that users got from the coordinates of program counters and stack frames to explain the system's trajectory, we will again have serious ergonomic

problems. `fluid.cell` provides a dedicated primitive `findCause` which is available whenever a reactive update is in progress, which will trace back through the graph to the original cause of the update. This can then be available to the substrate's own self-tooling which visually exports this information to the user — in the demo below, this path of nodes is highlighted in red during a reactive update.

Short-circuit behaviour on errors and incompletion

`fluid.cell`'s mechanism for supporting asynchronous computations, based around a specially marked payload named an unavailable value generalises to some other substrate requirements.

The short-circuit behaviour of unavailable values in the reactive graph is as follows:

A reactive computation any of whose dependencies is unavailable is short-circuited, and produces a further unavailable value, possibly accumulating an extra annotation holding the address of the computation
A reactive effect any of whose dependencies is unavailable does not activate.

The lifecycle of a malleable substrate is quite different to that of a conventional reactive system. A commodity reactive system expects the dependence structure of its graph handed to it as a *fait accompli* — its structure is already *implied* by the dependence structure of the body of code which has started to execute, and dependencies merely need to be *discovered*. However, a malleable substrate needs to supervise its own construction, which implies that considerable time is spent traversing reactive material on startup whose dependencies have not yet been loaded or constructed. The short-circuit behaviour of unavailable values provides a natural semantic to allow the substrate to make progress on constructing parts of the graph which are available without attempting to activate computations with missing values, a typical problem with commodity reactive libraries.

Analogously to accounting for the cause of updates, a substrate also needs to account for the causes of errors, and route the user's attention through its surface to the cause that a value is unavailable, either through outstanding I/O, design incompletion or a structural/syntacic error. Unavailable values also handle this case within the substrate, accumulating all the addresses of substrate cells on the paths between visible error sites and their causes. `fluid.cell` is agnostic about the substrate structure which it supports and so does little to interpret these aspects of the payloads to unavailable values which are intended to be processed in a substrate-specific way.

Dynamic dependency discovery

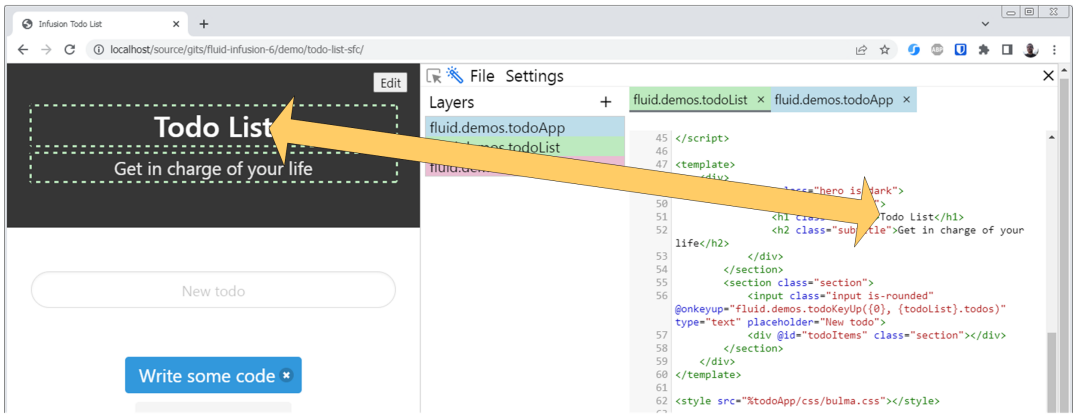
This is a standard feature of all commodity reactive frameworks, but in the context of substrate work this appears as somewhat of a surplus feature. In an imperative/functional environement which privileges the world of executing code, it is an idiomatic and powerful feature to subscribe a reactive computation to dependencies as they are discovered through the course of executing a conventional function encoding a reactive computation. However, in an openly authored substrate constituting an integration domain, the principle of *interface hiding*[3] implies that executing code should properly have no power of reference to other than its immediate arguments. However, `fluid.cell` retains this feature to support interoperability with imperative codebases.

Motivation and roadmap

In this section I respond to some comments from reviewers by motivating the need for bidirectionality in real applications and situating the `fluid.cell` implementation within the roadmap of implementing a malleable substrate.

Consequences of bidirectionality

Support for bidirectional dataflow is required naturally in non-authoritarian app designs. The image below shows a snapshot of the UI of the self-editable Todo list application which you can run live from the Infusion demos. The application naturally exposes two representations of the same resource — the application interface itself, on the left, which has been put into self-editing mode using the "magic wand" icon, and the source code representation on the right. The orange arrow shows the path of bidirectional dataflow, which respects the fact that the application should be equally editable from either representation, and has us further bear in mind that additional linked representations may appear and disappear dynamically.



Bidirectional dataflow entailed by a non-authoritarian design

When updates “go against the flow” in an application built on authoritarian unidirectional dataflows, the developer is snarled up in peculiar stateful makeshifts to deal with the book-keeping, or else non-standard primitives exposed by the reactive system such as Vue’s writeable computed. These are awkward enough for professional developers let alone something which we expect to be exploitable by end-user developers. Here’s the code contrivance required in the older version of Infusion’s own demo for this feature, which was built on the unidirectional preact-signals reactive library and CodeMirror 5:

One one side, here’s the “readEffect” in the CodeMirror component, which can’t be a proper citizen of the reactive graph (a computed) since it is paired up with an update acting in the opposite direction. Note the stovepipe guarding against cyclic dataflows using an ad-hoc flag `inReadUpdate`:

```
readEffect: {
    $effect: {
        func: (text, instance) => {
            instance.inReadUpdate = true;
            fluid.codemirror.updateText(instance, text);
            instance.inReadUpdate = false;
            fluid.invokeLater(() => instance.refresh());
        },
        args: ["{self}.text", "{self}.instance"]
    }
}
```

The writes in the other direction consume these flags like so:

```
instance.writeEffect = fluid.effect(validText => {
    if (instance.firstValid && holder.writeText && !instance.inReadUpdate) {
        holder.writeText(validText);
    }
    instance.firstValid = true;
}, [validText]);
```

Developers take this kind of paradigm-breaking hack for granted as part of the costs of building real-world apps and seldom try to take account of them by feeding back the need for them to library developers.

Roadmap to malleability

`fluid.cell` aims to be agnostic as to how it is used to implement the primitives of a malleable substrate layered, but I have a particular roadmap for how this is to be achieved in Infusion. The design layer above the reactive cells aggregates these cells into ***layers*** which are the unit of design reuse, as well as the unit of structural malleability. The contents of one or more layers can be composited by dictionary merging into ***components*** which capture reusable units of reactive dataflow.

The demos at the top of this page, whilst they represent an important class of use cases for interfacing with bodies of standard imperative code, don’t represent the dominant pattern envisioned in a substrate. They have been implemented here to establish parity with features in standard reactive libraries, as well as being a teaching aid in order to demonstrate how such features work in traditional contexts.

Whilst it would be possible, and supported, to manipulate the reactive graph from inside the body of the imperative code snippets implementing `computed` and `effect` arcs, this is not considered idiomatic under the model of an integration domain. In an integration domain, code is not expected to refer to arbitrary parts of a design using traditional programming language scopes, but instead as far as possible is expected to only refer to its immediate arguments.

Other primitives than the following ones could be supported by the overlying substrate, but here I briefly describe the ones used by Infusion, which seem to constitute a small, self-sufficient set. Fuller documentation for these will appear in future postings.

Infusion's strategy ***lifts*** the contents of reactive datacells up into the structure of the substrate through two main mechanisms, structural lensing and indirection:

Structural lensing through `$if` and `$for`

A reactive cell holding a boolean value can be lensed into the existence or non-existence of a component by referring to it in the `$if` member of a component's layer, e.g. in a textual, JSON form:

```
assignee: {
    $component: {
        $layers: "fluid.templateViewComponent",
        $if: "{somethingInScope}.enabled",
        ...
    }
}
```

The component will exist at times when `{somethingInScope}.enabled` resolves to `true` and will not exist when it resolves to `false`.

A reactive cell holding an array value can be lensed into the existence of a corresponding array of components by referring to it in the `$for` member of a component's layer, e.g.

```
todoItems: {
$component: {
    $layers: "fluid.tests.todoItem",
        $for: {
        source: "{todoList}.todos",
            value: "todo",
            key: "itemIndex"
    },
    ...
}
},
```

These constructs are named after similar conditional and looping primitives seen in imperative programming, but rather than being control flow constructs, they instead set up reactive correspondences which are navigable through the kinds of bidirectional relationships described above.

Polymorphism via indirection: `$layers$`

To achieve dynamic polymorphism in the substrate, the names of layers held in layer definitions can be looked up through reference into cell contents. Through this correspondence, reactive updates to the data cells become naturally lifted into structural updates of the dataflow graph itself. In a traditional programming paradigm this would be considered an exercise of metaprogramming.

For example, in the malleable IDE usable in the demo above, the flavour of editor constructed to edit a particular component is geared through this reference to `{self}.layerRec.editorModeLayer` in the `$layers` member of the component — this holds a string which, under the standard reactive evaluation semantics, will be looked up to a layer name and then composited onto the component instantiated at this path:

```
editorHolders: {
    $component: {
        $layers: ["fluid.editor", "{self}.layerRec.editorModeLayer"]
```

Full explanations of Infusion layer and component primitives are out of scope for this discussion and will appear in future postings — this discussion here is just intended to prime intuition about how coarse and fine-scaled malleability of the substrate as a whole is intended to arise from ordinary reactive updates to dataflow cells.

Feature comparison

Here is table showing a feature comparison of `fluid.cell` against a number of historical reactive implementations — "Discovery" indicates dynamic dependency discovery and "Bx ⟺" indicates support for bidirectional reactive edges.

Year	System	Discovery	Glitch-Freedom	Early Cutoff	Async	Bx ⟺	Causes
2001	cells	✓	✓	✓			
2006	FrTime	✓	✓	?		✓~	
2010	knockout	✓	✓				
2012	S.js	✓		✓			
2014	mobX	✓	✓	✓			
2020	Observable	✓	✓	✓	✓		
2023	Solid	✓	✓	✓	✓		
2026	fluid.cell	✓	✓	✓	✓	✓	✓

Demo key

You can see it working above. Understanding how control flow works in detail when embedding reactive primitives within an imperative backdrop can be a puzzle, so the legendary “flag” test case from preact-signals, together with some others, is here animated in a timeline alternating between imperative statement steps (S) in blue, and reactive computation steps (C) in purple.

In addition, the cause of any current reactive computation, that is, the path to the node whose update originally scheduled it, is highlighted in red.

You can step forwards and backwards through the test fixture to see nodes and arcs being built up and torn down, and also the reactive cell values and states being altered. Each node can be in one of three reactive states —

Clean — The cell is fully up to date with all of its dependencies
Dirty — The cell is provably dirty, since one of its immediate dependencies has recently updated
Check — The cell may be dirty, since a distant upstream dependency has recently updated

These states and their colours are taken from Milo Mighdoll’s admirable Super Charging Fine-Grained Reactivity illustration of his own Reactively system, from which a good part of the `fluid.cell` implementation has been adapted.

Note that without support for asychronous propagation of reactive values, this demo could not have been implemented, since behind the scenes the original synchronous test fixture is rewritten to an asynchronous one which then suspends waiting for user interaction.

---

1. Section 4.4.4 of Software and How it Lives On, Basman et al, 2016, describes a bulky, transaction-oriented style of reactivity named “Queen of Sheba adaptation”. ↩
2. Ryan’s 2021 article is excellent but betrays the complexity of this terrain in the analysis of Svelte’s reactivity — he observes that a touchstone of why Svelte is not reactive is that one can “read a derived value on the next line of code it isn’t updated. It is definitely not synchronous.”, yet the latest incarnation of his signals system, solid-signals, shows just the same behaviour. I actually agree with him that Svelte is not reactive, but not with this exact reasoning of why not, and yet at the same time I also support the behaviour of his latest test cases. ↩
3. Described in Stephen Kell’s The Mythical Matched Modules (2009). ↩