Substrate vision statement

Google docs version
Jonathan Edwards

What is a substrate?

I define a Substrate as:

- A complete and self-sufficient programming system,
- with a persistent code & data store,
- providing a direct-manipulation UI on that state.
- Supports live programming.
- Programming & using are on a spectrum, not distinct.
- Conceptually unified — not a “stack”.
- Summarized as a slogan: a WYSIWYG document, DB, & PL in one.

The canonical examples of a substrate are Smalltalk and LISP systems. HyperCard and Flash were much-beloved beginner-friendly substrates. Spreadsheets are by far the most successful substrate, and an inspiring existence proof that alternative programming experiences are possible. Webstrates construct a substrate in the web browser, though with a different definition:

> “We define shareable dynamic media as collections of information substrates (or substrates for short). Substrates are software artifacts that embody content, computation and interaction, effectively blurring the distinction between documents and applications.”

What are the benefits?

- Building applications as documents/images provides a more consistent user experience and a simpler developer experience.
- There is a gentle progression from user to developer.
- Beginners can quickly build non-trivial applications, and easily become competent.The parade of no-code/low-code tools attests that this need is still unmet.
- Less code is required because inessential impedance mismatches are dissolved.
- The programming experience is improved by having a small set of tools working throughout the substrate, rather than specialized tools for each specialized technology in a stack.
- Live programming and ubiquitous observability makes it easier to understand, debug, and modify code.
- The confidence of working in a human-scale world that is coherent and knowable.

What are our major research problems?

- Can a substrate be pluralistic? Related to work on integration domains, component models, and malleability.
- Can the stack actually be unified or does it reflect essential specializations?
- Substrates have been built upon dynamically typed PLs. Could a statically typed PL serve instead?
- Substrates have been built upon PLs (Smalltalk/LISP) and UIs (the browser). How about building upon a DB instead?
- Edit calculi (see my personal research statement).
- UI for navigating the substrate: inspector windows, outline, zoomable canvas, etc.
- Provenance and observability.
- Programming by Example.
- Modes of collaboration: multiplayer, git, and beyond.
- Substrate as an ecosystem: libraries, packages, and DLC.
- Lifting Unix or the browser into a substrate.
- Interoperating with the mainstream: calling/serving HTTP APIs, reading/writing standard file formats.
- What is the role of LLMs? Will they obsolete the need for substrates?
- What is a “killer app” that justifies substrates?
- How do we evaluate our results? See UIST Author Guide.
- Where do we regularly meet, publish, and present?

What do we seem to disagree about?

- What kind of user are we targeting?
- Should a substrate include a full programming experience?
- Should a substrate be self-contained or can it expose underlying standard tech?

Are we even a field?

To be a field of research there should be a productive exchange of ideas. We should be citing each other as related work, and extending or critiquing each other’s ideas. History indicates that progress accelerates when there is a healthy mix of competition and collaboration. To foment that interaction it helps to have a regular meeting place.

What I’d like to see result from this workshop.

- Identifying research problems and disagreements.
- Defining one or more canonical examples like TodoMVC to compare substrates.
- Publishing a report of the meeting, like the 1968 NATO SE conference.
- Creating a Wikipedia page for Software_Substrate citing our report.
- Planning to meet again, perhaps a one-off like Dagstuhl, or an annually recurring event.
- Inflaming rivalries and alliances!

Personal Research Statement

[These ideas have been the subject of many discussions with Tomas Petricek.]

Who is the user? My target users are the beginners and non-technical people that embraced HyperCard, and the "power-users" of spreadsheet fame, but who need more power and generality. I want to give them the power of Smalltalk without losing the user-friendliness and "conviviality" of HyperCard and spreadsheets.

Who isn't the user? I am not targeting users like myself, nor current professional programmers. I do not want to build a "tool for thought" for intellectuals. I would rather build a "tool for getting stuff done" by ordinary people.

I make some opinionated design choices:

- Focus on data first. Users care more about data than code.
- The data model unifies key features of documents, relational DBs, and PLs.
- Static typing to benefit the PX and UX, schema change for flexibility.
- Built-in user-friendly distributed version control.

The benefit of a unified data model is to avoid the infamous *impedance mismatches* creating much complexity when shuttling information between the DB, PL, and UI. My experiments have converged on a model like that of statically typed FP languages: a tree of records, sums, (homogeneously typed) lists, and atomic values except that:

1. Data is mutable.
2. Every edge of the tree (record fields, sum components, list elements) has an internally generated globally unique permanent ID.
3. There are cross-links in the tree, defined as a path of IDs from the root, subject to certain static and dynamic constraints.
4. Type/schema change is a first-class operation.

To establish a solid theoretical foundation I am exploring an *Edit Calculus*. Generally speaking an edit calculus formalizes the interaction between a user and a stateful system, where edits are operations mutating the state. This particular edit calculus originated by asking how a substrate could be statically typed? Changing a data type must simultaneously adapt any instances of that type, called schema migration in DBs. The problem is that you can't tell how to migrate the data just by comparing types before and after. For example, was a field moved or was it deleted and a new field inserted? It is necessary to capture the user's *intention* as they interactively edit the type. I formalize this with a set of edit operations that are surfaced in the UI to capture the intention of a type change and accordingly migrate instances.

But beyond schema migration the edit calculus turns out to also enable new modes of collaboration. I generalize the theories of Operational Transformation (OT) and Convergent Replicated DataTypes (CRDTs) to support collaboration like that in distributed version control systems (DVCS). An edit calculus over a data model not only defines what the edits do, but also makes rules for how an edit *migrates* forwards or backwards through other edits so as to preserve the user's original intention. From these rules we can generate analogs of DVCS

capabilities: diffing, reverting, merging, and cherry-picking. Unlike traditional DVCS systems like git these capabilities:

1. Integrate changes to code, data, and types/schema.
2. Operate to preserve intentions rather than concrete differences.
3. Span multiple modes of collaboration, from traditional transactions to multi-player collab to loosely coupled version control.
4. Function in an open world of documents, not a bounded repository.
5. Present a coherent conceptual model abstracting from the implementation.
6. Provide a feature-complete GUI.

Collaboration is a relatively new feature for substrates. The classic systems focused on (and in large part invented) the personal computing experience. I am betting that the edit calculus can provide collaboration capabilities that not only match but exceed those of mainstream programming tools. Winning that bet would reframe the narrative: instead of substrates being beginner versions of "real programming" they are a new technology with unique benefits for a different audience. For example end-user merging of document variants. Could a substrate like Notion/Airtable with end-user programming and next-gen version control be a killer app?

My research prototype is called Baseline. Progress has been reported in: Version Control for Structure Editing; Managed Copy & Paste; Operational Version Control; DB usability: as if.

There are still major unsolved research problems:

- I haven't worked out transactional and multiplayer modes.
- I haven't found a clean algebra of edits on the data model comparable to relational algebra. Maybe the model is not quite right yet.
- The migration rules currently struggle with duplication and irreversible edits. These situations clash with my intuition that the migration rules should satisfy certain symmetry properties to be correct. Something has to give.
- I need an executable specification language for the migration rules.
- I need to prove or property-based-test some notion of correctness for the edit calculus.
- Data comes first but ultimately there needs to be an embedded programming language that can at least do queries. My vision is to extend the edit calculus into a full-fledged PL with novel capabilities, including my previous experiments on Subtext. The fallback is to use a conventional PL design.
- Much of the work so far has been iterating on the UX of version control on structured data. There is much work left to do.

********** PAPER 3 **********
AUTHORS: Jonathan Edwards
TITLE: Substrate vision statement

++++++++++ REVIEW 1 (Gilad Bracha) +++++++++
I like the definition of substrate. It is more precise and prerscriptive than others - which has pros and cons. I do feel a bit ambivalent about the "not a stack' requirement. Even in a conceptually integrated system, there will be layers: compilers, basic tooling like debuggers and code browsers, UI frameworks, etc. They just need to be open and malleable and made of the same "stuff" (same PL, bytecode etc).

Overall, very good coverage of topics and issues. Below are some of my concerns/reservations.

I confess that I personally have never been a fan of databases. Programming environments should subsume them, not become them. To paraphrase Dan Ingalls: A database is a response to PL failures (wrt persistence and data management) - there shouldn't be one.

Collaboration/local first is a major topic. I admire your insistence on developing a theory up front - but I feel that theories come later, after some working systems.

AI is mentioned in passing, but to me it is the largest issue of all. The justification for improvements in programming will be in how they help manage the use of AI; if they reduce cognitive load for people, they may well reduce it for AI as well. And they must help manage AI behavior (say by testing and verification).

The tension between "standard tech" and a clean pure approach is evident in several submissions. It's a trade-off. It is a test of design how you pull it off, but also constrained by your available resources. We typically have very little resources in this area, and so are pushed toward pragmatics. The results are less than ideal. For example, I find that using HTML as the markup language is very much a double-edged sword. I chose that route, and am far from certain if it was the right choice. Yet nothing else is adequate at the moment; markdown is not nearly powerful enough. Perhaps defining a clean markup language is a possible subgoal?

A similar tradeoff occurs at the PL level. Many opt for JavaScript. Here, I cannot compromise.

Lastly, don't you find that using EasyChair is an admission of failure? If we cannot produce something much better than EasyChair for our own use, what's the point? Surely some dogfooding should also be a subgoal as well.

++++++++++ REVIEW 3 (Tomas Petricek) +++++++++
The statements combines general observations on software substrates with an outline of a more specific personal research agenda. Separating these two is a good move as they are often somewhat interleaved in other vision statements (and so it is harder to separate more general points from specific research interest).

The definition of a "Substrate" in the vision is interesting, but seems to be quite specific. I think it would be interesting to see how we can extract some more general description (or perhaps, something that characterizes the notion in some way that does not involve listing specific features?) Finding a definition is certainly harder than I thought!

The vision statement is also representative of a couple of "modernist" submissions that propose a new system - a substrate that will "take over the full stack" and replace it with something sensible. I'm personally also on this side of the research spectrum, but I'm very curious to see how this can be combined with the more "post-modernist" approaches based on existing software ecosystems. It seems that for the "substrates movement" to succeed, it is crucial that we find a way to reconcile the two research directions - learn from each other and advance. (Perhaps,

the "modernist" visions can be seen as being a bit like "formal PL research" that ignores practical concerns, but can explore ideas in a more pure form, whereas the "post-modern" approaches see how to actually implement systems that work in practice?)

Regarding the specific research vision - there are a couple of submissions that focus on some kind of programmable document substrate. Perhaps one outcome of the meeting could be some sort of analysis of design considerations for such systems? This is certainly something I've thought about too! (But there is also Ampleforth, etc.)

++++++++++ REVIEW 4 (Camille Gobert) +++++++++
This statement provides an overview of substrates as a research theme: it proposes a definition with key characteristics, discusses benefits and challenges of thinking in terms of substrates and questions if and how we should make a research field out of it. It then presents the direction that the author has started exploring: their focus on non-programmers who need programming, their technical choices/preferences (such as bringing easy-to-use version control everywhere and UUIDs for every data structures) and their attempt to formalise software evolution in the form of an _edit calculus_.

The list of items that define a substrate is very clear, and it would be very helpful to conclude the workshop with such a precise definition. That being said, I personally agree with Basman's statement: to me, several of these properties are desirable rather than mandatory. In particular, I fear that "a persistent code & data store" might mean that many substrates constitute information silos, which might only be viewed and manipulated to the substrate's UI, whereas I think that a substrate might be a "view" of information it does not necessarily contain (but perhaps that invalidates the point about not being a stack?)

I also really appreciate all the meta-level questions raised in this statement. They might constitute a useful guideline to steer the actual workshop! The diversity of the statements submitted to the workshop makes me question whether we should be a single "research field", in particular because we might be interested in very different aspects of research on substrates, with different methods and related communities (for example, how to implement them vs. how to evaluate our interaction with them). Maybe we could start by defining the concept and leaving each community (PL, HCI, etc.) appropriate the concept?

Furthermore, this statement, along with others (such as Dubroy's), seem to point out that we need to define who/what substrates are for. I second this. Could we even go further and make the notion more plural? Although the workshop talks about "software substrates", some statements focus on the role of substrates for AI whereas others focus on low-level OS concepts and data structures. Moreover, previous work from Beaudouin-Lafon and Mackay introduce other types of substrates, such as information and interaction substrates (as mentioned by, e.g., Basman, Klokmose, Trividic and myself). Is there something common to all of these views on substrates? Are there systematic differences that would justify creating multiple families of substrates, with different purposes (e.g., interaction vs. computation), which may eventually be studied by different communities?

++++++++++ REVIEW 5 (Patrick Dubroy) +++++++++
This is a much more restricted vision of what a substrate is than the definition I have in my head!
What do you think the benefits are of this relatively tight, prescriptive vision of substrates?

Interesting that you also look to UIST for inspiration. In my own vision statement, I very nearly described my idea of what the field of Substrate Studies might look like as, "What UIST used to be". It may be interesting to discuss some of the critiques of the UIST research culture (Landay's "I'm done with CHI/UIST" and buxton/greenberg come to mind) and ask ourselves questions like: How do we avoid these problems? And, what makes us different?

I love idea of identifying canonical examples! I suspect the process of defining these would also help elucidate our definition of "substrate".

I also like the idea of producing a report of the meeting…and optimistically think I'd also enjoy meeting again. :-)

To your personal vision statement:
- I appreciate how you begin by answering who it's for. I'd like to see more research do this. It also connects with my own suggestion that defining the purpose of a substrate should be an important value in this nascent field. I'd incorporate your angle and say that we should always answer the what AND who.
- Your discussion of the edit calculus is a good example/argument for the holistic perspective of substrates as a field, and why the isolated fields of databases, PL, and HCI aren't perhaps sufficient .