substratus unicus

Stephen Kell
King's College London

My own research has been pursuing two related aims: bringing Smalltalk-style dynamism to a conventional Unix environment, and overcoming the tendency of language to fragment the programming ecosystem. The appeal of dynamism is immediacy, succinctness, flexibility, and its ability to address and explore an open world. The world is dynamic! It respects some but few invariants, is prone to both data change and schema change (or 'invariant change'!). It is large, but it is explorable given the right tools. A dynamic medium is such a tool—one for first thinking through, and then systematising, an interaction with the world.

Since the appropriate degrees of systematisation and formalisation vary from task to task, it is important that the same medium allows varying that degree. This is also one of many arguments for lessening language barriers: to allow gradual migration to more rigid but more statically tractable languages as the systematisation progresses, including to (in some cases) the prospect of verifiable assurance.

My unusual focus on extending a conventional Unix environment is best explained by defining 'substrate' unusually. I will be intending mostly to talk about a hypothetical 'low-level substrate'.

What is a substrate?

Substratum: that which has been strewn, scattered, spread out or under. While accepting that meaning need not follow etymology, I will adopt this definition. Substrates must be widely deployed, or intended to be so. They must be 'a thing' rather than a scattering of small things: the strewn parts must somehow interact or have the potential to do so, at least via what grows above them.

In the world of computing, we have one 'most successful' global substrate, namely the Internet. One can quibble about its successes and failures. Undeniably though, it has bootstrapped itself far enough to supplant an obviously worse state of affairs, and made possible various new things (good and bad!).

Software is in a pre-Internet state. One piece of software is helplessly isolated from most others, and isolation can be avoided only by accident of buy-in to the same platforms and ecosystems ('each a model Earth') or by Herculean ad-hoc integration efforts, whether glue coding or data migration.

These arguments are likely familiar among workshop attendees. My tentative answer is somehow to minimise the 'buy-in' hurdle faced by new platforms, by means of the IP-esque, 'hourglass' concept of universality: how can we underpin a maximally wide variety of existing systems and technologies, imbuing what sits above them with programmable substrate-like qualities (to the extent that that makes sense)?

Pasts to learn from

If Smalltalk-style dynamism is the goal, why isn't Smalltalk the solution? Smalltalk-style systems have many of the right attributes, but lack 'interaction with the world'. They *are* their own world. This limits them on both ends: they support authoring but primarily 'from scratch', and they support publishing but primarily 'in toto' (as images). The self-contained design has always been a limitation (Rob Pike's analogy of a 'model Earth') but fits conspicuously poorly with today's highly interconnected world.

(I am aware: some partial solutions exist! Meanwhile, an ultra-connected design brings its own hazards. I revisit this latter point under 'staying lean', below.)

The Internet as a concept was political, and I suspect a substrate must also be political to an extent. In short, the 'personal computing' ethos of empowering individuals and small groups to *work with* a soft material, with 'independent efficiency' (to borrow Illich)—not what Big Tech providers are going for, but hopefully not controversial in this audience. Of course the Internet's founding politics now look naïve: its 'flat' ideals, although suc-

cessful in displacing telcos as network gatekeepers, have largely been defeated by new 'native' corporate giants who now fulfil a similar role. As with social media, a substrate with values must defend them, perhaps by seeing its future controllers as a potential adversary.

POSIX is arguably another substrate: it is certainly spread and scattered about. However, it has not done for software what the Internet did for communication. It never had the relevant design intentions: to unify that which had been kept separate and to lower the cost of application development and deployment. POSIX is a base-level abstraction; in the Internet context, its analogue is 'a network', remaining in need of federating, in contrast to a 'network of networks'. The Internet's abstraction is at the meta-level. (My research vision is of an evolved meta-level, broadly compatible with and complementary to POSIX, that fill this gap.)

There are any number of interesting, ingenious live programming systems (that term to be taken as widely as possible) that unify editing with programming. I am definitely not expert in this literature and I look forward to learning at the workshop. None I have seen is sufficiently focused on interconnection *from the lower levels upwards* to meet my goals.

Research problems

Communicating with aliens The heading borrows Licklider's pithy framing. How can we design and build a computational medium that is outward-looking rather than enclosing and hermetic? Resonances exist with Lanier's 'phenotropic computing' [**?**] and with Shaw's thematic of 'connectors versus components' in the software architecture literature [**?**]. Capturing patterns of communication, in a re-usable or adaptable fashion, is a black-sheep problem in computer science but is key to any interconnected software substrate.

To pick a thought experiment, raher than piling ever more utility code into a 'standard library' what would a programmable environment look like that forbade itself from defining any standard library? Rather, it would provide only the primitives for being taught how to interact with 'found' code on the host system. This is a valid alternative means of bootstrapping functionality, and one that immediately shifts the focus from 'abstractions for code' to 'abstractions for interaction'. One might start out by something resembling a domain-specific language for 'calling conventions' (suitably generalised) and a way to tabulate known instantiations of a convention (e.g. the Linux system call table) One simple interaction might be 'if I put this number into this register, then issue this instruction, a `gettimeofday` system call occurs'. These primitives might not need to be Turing-complete.

Another way to think of this, returning to the networking analogy and supposing we equate 'code' with 'services', would be that our concern is *gatewaying* existing services rather than defining new ones. Gatewaying is quantified, not pointwise: a gatewaying solution applies to a large space of code in one go. In other words it is 'at the meta-level', at least architecturally.

'Wide-spectrum media' How can one medium let us start with a notebook of personal experiments and end with a 'blueprint' of a robust, reproducible and well-understood application or unit of functionality? In programming language circles, this is traditionally seen as a language design problem, with approaches such as gradual typing or incremental software verification. For the reasons given above I am inclined to see it instead as something closer to an system design issue: any used notation and its semantics are less important that the workflows that enable the gradual progression.

Keeping it lean Software tends to grow all wrong. Understanding why and how software grows over-large is an essential socio-technical research challenge. It has received little attention to date. It relates to substrates, in that the tendency towards 'aircraft carriers for the mind' appears partly an unintended growth phenomenon, with a dose of path dependence; it was evident even as Wirth wrote in an era and context of personal computing. Today, with the industry predicated on a whacked economics of extreme aggregation, tech giants may indeed set out to design an aircraft carrier (scalable no doubt, but at what COST?). Meanwhile the commodity software (and hardware) of 40 years ago looks refreshingly simple and hackable.

Much as I critiqued the disconnected model Earths, an ultra-connected design puts leanness further from reach: the more interconnected our software, the greater the temptation of importing large and ill-fitting units from outside as a substitute for local care and thought (the 'left-pad' effect). This practice has been conveniently compensated in a few ways: by the particular capacity of software to take mountains of complexity and shovel them conveniently and ignorably 'under the carpet' (this is called 'abstraction' and has been made a virtue), by the ongoing availability of faster hardware, and by the normalisation of buying it and throwing the old stuff away.

Temptation is especially high because the compensation effect is more of an externality: under today's business models, very often the actor receiving the benefit does not pay its cost. When web developers take the shovelware approach, it is end users who must upgrade their device.

Software resource accounting is a mainstay of operating system design but its reach does not extend to this

scenario. It is worth questioning why.

Disagreements

Compatibilism I learned this word at the LIVE 2024 workshop. How much does pre-existing code matter? I have an extreme compatibilist preference, while recognising that in practice, both clean-slate and evolutionary approaches are illuminating. I would, however, rebut the common presumption that compatibility is an albatross, or implies clinging to all the wrong ideas of the past. Compatibility is a burden when it demands creating a new facsimile of old behaviour. It is precisely that which I propose to avoid, by embracing the existing code that already does the job.

Distribution How important is distribution? I suspect some may disagree on this. I myself have largely ignored distribution this far in my work, but feel that calling anything I create a 'substrate', in the 21st century, would have to tackle it.

Can there be only one? I tend to argue for pluralism in most things. But just as how IP is the unique waist in the hourglass, I tend to imagine that for a distributed low-level substrate, there must be only one. Of course this could be a 'winner' arising from a period of competition. I'd welcome more pluralist views, however.

The web platform I excluded the world-wide web from my earlier surveys, but this is obviously unfair. I generally have an intuitive dislike of web technologies, without fully understanding why. They appear over-complex, having too many moving parts and too reliant on servers that cost money to run (thanks to problems with today's Internet, admittedly) and/or limited, appliance-ised hosting environments. I have a lot to learn here, though. For now I would still tend to pitch solutions that layer underneath the web rather than within it, but I await conversion.

Methods

At a high level, building a substrate and gaining experience with it is the only method that seems credible. Case studies showing how are one obvious approach. Task-focused user studies are possible, although given a reasonably mature system, a more ethnographic approach may become possible and would be more convincing. A compatibilist focus creates other nearer-term evaluation stories, such as retrofit studies ('it works with this much existing stuff') or performance delta studies ('I didn't slow it down too much').

What I'd like to see arising

Intuitively, across communities that neighbour or intersect with this one—local-first and live programming come to mind—I tend to see a recurring deficit in willingness to approach the lower-level. Necessarily, 'operating-system-like' services provide for communication among distinct applications and user data. These services sit underneath applications in the stack, but providing some cross-cutting services e.g. of user identity and possibly even resource accounting (noting my 'externality' point earlier). The problem is that communication outwith the application boundary is too often left as a problem for users to solve, maybe by some primitive import/export mechanism and/or by assuming a world where the system itself has conquered all (see 'compatibilism' above).

I would like to see In short, a theme I am drawn to is that the important problems in 'substrates' appear to require a solution of which at least part is rather less like a live programming environment or collaborative editor, and more like a distributed operating system. That is not a clearly delineated contrast. It may amount to nothing and I would like others to critique it.

A realisable prototype might be a a wide-area distributed OS realised as a 'partially shared system image' overlaid on a conventional Unix userland, and implemented by interposition at the syscall level. It would be small-scale, suitable for a small group of collaborating users, and can be thought of as 'timesharing after timesharing'. A single user device could potentially participate in many such partial sharings at a time, and the barrier to entry should be strictly less than installing a new operating system, and should not require running a public web server. Onto this one could then graft a a communication-oriented and/or more 'live' evolution of Unix-esque programming abstractions. One public-facing device participating in the overlay would be sufficient to make any of the overlay's state publicly addressable. A major part of this 'evolution' is mutually gatewaying as many 'spaces' as possible, across network, filesystem and memory; I have omitted elaboration of this here, but it is not dissimilar to Alan Kay's 'every object should have a URL'—although to me DNS seems a more interesting network namespace than URLs.

References

to be provided later, apologies! I will put a referenced (and perhaps lightly tweaked) version at https://humprog.org/~stephen/research/reports/kell25substratus.pdf in time for the workshop.

********** PAPER 9 **********
AUTHORS: Stephen Kell
TITLE: substratus unicus

++++++++++ REVIEW 1 (Gilad Bracha) +++++++++
Following up on our conversation at WGLD:

I don't really fully understand how this universal federation stuff is supposed to work. And I don't see how using it will feel like a substrate; I don't think substrate is just a new term for platform, which is what I get from the paper.

What I would like to know more about is how this concretely relates to: "bringing Smalltalk-style dynamism to a conventional Unix environment". Where and when can I see that working?

More broadly, I am as far away from a "compatibilist" as you will find. For me, the point of these systems is to be a joy to work with. They need to be be clean and uniform.

I do agree that distribution is very important (in the sense of distributed computing; also in the sense of deployment).

++++++++++ REVIEW 3 (Jonathan Edwards) +++++++++
This vision statement approaches substrates from an OS point of view. That is in contrast to the more common PL point of view and my own DB point of view. As such I very much welcome Stephen's voice in our discussion.

There are so many nuggets of insight in this piece that I feel the need to quote them here as an aid to memory.

"Software is in a pre-Internet state."
Wonderful analogy! It predicts that, like the Internet, there will be a one-time transition to a hegemony, and that this will be seen as inevitable in retrospect.

"There are any number of interesting, ingenious live programming systems ... None I have seen is sufficiently focused on interconnection *from the lower levels upwards* to meet my goals."
This might be our essential point of contention, so I need to understand it more clearly.

"Communicating with aliens ... Capturing patterns of communication, in a re-usable or adaptable fashion, is a black-sheep problem in computer science but is key to any interconnected software substrate.

Wide-spectrum media: How can one medium let us start with a notebook of personal experiments and end with a ‘blueprint’ of a robust, reproducible and well understood application or unit of functionality?

Keeping it lean: Software tends to grow all wrong."

"Compatibilism ... Compatibility is a burden when it demands creating a new facsimile of old behaviour. It is precisely that which I propose to avoid, by embracing the existing code that already does the job."
This is how the Ring corrupts even the wise! Embracing the old code, however lightly, imprints an impedance mismatch. Only a solid and carefully crafted abstraction barrier can avoid contagion, in my opinion.

"Can there be only one? ... just as how IP is the unique waist in the hourglass, I tend to imagine that for a distributed low-level substrate, there must be only one."

"the important problems in ‘substrates’ appear to require a solution ... rather less like a live programming environment or collaborative editor, and more like a distributed operating system."
This strikes me as an important observation, but I must confess to having avoided engaging with these issues.

Offered as a provocation, here is Avery Pennarun of Tailscale https://tailscale.com/blog/new-internet
"Everyone’s attitude is still stuck in the 1990s, when operating systems mattered. That’s how Microsoft stole the fire from IBM and ruled the world, because writing portable software was so hard that if you wanted to… interconnect… one program to another, if you wanted things to be compatible at all, you had to run them on the same computer, which meant you had to standardize the operating system, and that operating system was DOS, and then Windows.

The web undid that monopoly. Now javascript matters more than all the operating systems put together, and there’s a new element that controls whether two programs can talk to each other: HTTPS. If you can HTTPS from one thing to another, you can interconnect. If you can’t, forget it."

++++++++++ REVIEW 4 (Antranig Basman) +++++++++
I am enjoying writing this response since there are so many points of contact between our research programmes - in fact, several of your papers (Mythical Matched Modules, 2009; In Search of Types, 2014; Some Were Meant for C, 2017) are foundational for mine. I also share your feeling that “a substrate”, once produced, should in some sense be canonical or universal, but on the other hand can’t help feeling that now I have at length produced one that it might meet few of your needs. Whilst I have few hopes of overcoming your intuitive distaste of web technologies (which I feel many workshop participants will share), I would like better to understand “the space between us” and see what ropes can be slung across it.

Firstly - points of agreement. Like you I am a strong compatibilist and believe the first goal of a substrate is to help its users deal with the space of technology as it lies. Given the web *is* “how it lies” I then find a lack of taste for it and a desire to target the layer beneath it a bit puzzling - since for the vast majority of technology users there is no layer beneath this that they can recognise. I’m not doctrinaire about this and imagine potential for a future space where as you say, a “period of competition” has won a niche for a widely distributed lower level, but feel that we have to accept that for the web, this period of competition has long been over and like a planet, it has “swept its orbit clean”. And thankfully so - how much worse would the world be if any of the far less pluralist stacks, such as Silverlight, Adobe Flash, Google Gears or the like had got a foothold. But there are other stacks that could be attractive targets in the near term, interestingly emerging from game platforms such as Godot or the Lua-based LÖVE.

Strong resonance with the idea of a “personal computing ethos of empowering individuals and small groups to work with a soft material, with independent efficiency”, and also agree that a substrate must be political to an extent. Naturally a substrate must appeal to a mode of production which is at variance with how most software is produced today, by top-down organisational hierarchies at the service of closed-form commercial objectives. We’ve discussed in the past how these hierarchies have created a seeming moral absolute out of quite situated technological moralities such as Parnas’ information hiding, Liskov’s substitution etc.

Further strong resonance with the point that “that communication outwith the application boundary is too often left as a problem for users to solve, maybe by some primitive import/export mechanism and/or by assuming a world where the system itself has conquered all”. A successful substrate should have its boundary to the outside world as permeable as possible, which implies that its file formats should be as simple and transparent as possible. It’s interesting to hear you cast our goal as “Smalltalk-style dynamism” since correctly in the same breath you cast this as the Achilles heel of all Smalltalk-like systems -- they suffer from an irremediable “image problem” with their abstractions cast in impermeable terms only intelligible within the same system image. And to me this *is* the Smalltalk style, the very thing that we seek to get beyond. We need to do far better than Smalltalk-style dynamism.

The idea that an important limitation of web technologies is that they are “too reliant on servers that cost money to run” seems funny, since all stacks which _cannot_ be cast in terms of commodity data formats and protocols are inevitably going to cost more money to run! And I have to say that my own primary mode of deploying applications is reliant on servers which cost no money to run, that is GitHub’s “Pages” servers. A couple of years ago I also used to be a bit ideological about server running costs but realise now that the ecosystem is barely constrained by this quantity. It’s far more constrained by the costs of interoperation, juggling notionally compatible but actually divergent file formats, duct-taping things together in a way which doesn’t incur endless ongoing dependencies on maintainers, etc -- issues right at the heart of your research programme on integration domains.

So to a final central contradiction - this paragraph is terribly interesting:

a solution of which at least part is rather less like a live programming environment or collaborative editor, and more like a distributed operating system. That is not a clearly delineated contrast. It may amount to nothing and I would like others to critique it.

Having now produced a substrate of which a considerable part *is* like a live programming environment or like a collaborative editor, I’d love to explore this contrast. I find it hard to escape Jonathan’s essential characterisation that a substrate supports “a WYSIWYG document, DB, & PL in one”. I guess there is a fine distinction here to tease out - Jonathan’s vision statement *summarises his definition* with this slogan, but as I see it this is not necessarily the definition of the substrate itself, but rather an essential capability that it allows for. So this distinction may be the key to breaking out of this contrast.

I’m personally less interested in how the capabilities of the substrate get “distributed” and it seems there are a lot of capable researchers interested in that problem. For example in this group we have folks working on/with systems like AutoMerge, Croquet etc. and I expect this work in itself to be successful enough, but leave untouched the issue of what it actually is that gets distributed, and how its meaning is assembled once it arrives at some personal but shared computing site.

So to ask the initial question again - how might a substrate written by me and you differ rather than being unique? I have the main point of comparison of your Cake language (Component Adaptation and Assembly Using Interface Relations, 2010). Reading this again reassures me that there are more similarities than differences - this is a “language of correspondences” which is “not Turing powerful”. The central missing capabilities in Cake, that I’ve been struggling, I think now successfully, to accommodate in my system Infusion are that

i) It is not _reactive_ - it does not explain or produce a mechanism for how expressions of correspondences which vary over time are to be honoured - as far as I can see it is a build-time system which assumes these have been specified once in their entirety. In this it seems interestingly similar to another integration/open authorial system Jonathan pointed me to, USD - https://openusd.org/release/intro.html - the Universal Scene Description system used by the animation community to explain how 3d scenes are assembled out of assets. It would be interesting for us to talk about whether I am missing some obvious route for an apparently static integration language to be “hoisted” into a reactive one without much work, but personally I’ve found reactivity to be the central difficulty I’ve needed to grapple with.

ii) It does not explain how the expressions of multiple authors with different integration specifications are meant to be overlaid and resolved. This *is* something which USD tackles comprehensively, but from my point of view with greater complexity than is needed.

As far as i can see, both of these are essential capabilities - we are aiming, as you say, for a pluralist system where “A single user device could potentially participate in many such partial sharings at a time”. You say in the next sentence that “onto this one could then graft a
a communication-oriented and/or more ‘live’ evolution of Unix-esque programming abstractions” but I am unconvinced that these values could be grafted onto a system where they have not been designed in from the start. To me the liveness of the resulting system has been the main challenge, but I look forward to us critiquing whether some simpler decomposition of these requirements is available.

++++++++++ REVIEW 5 (Steven Githens) +++++++++
Really great paper. As someone who is mostly working at the 'higher level' of substrates I particularly enjoyed the systems levels aspects to suplement the usual 'lower level' issues we discussion which are usually about syncronizing data, version control etc.

The "Communicating with aliens" second paragraph proposing the forbiddence of a standard library, is particularly thought provoking to me, largely because of my work in Boxer where historically the number of primitives have been kept quite low, but as a programmer my knee jerk reaction is always to add more "things" to some type of "standard library". I hope we're able to talk about this at the session, and what it looks like to perform various modern tasks, such as playing an audio file, in this type of environment.

"If Smalltalk-style dynamism is the goal, why isn’t Smalltalk the solution?", I really love this, as well as the disagreement sections on compatibilism and "Can there be only one?" and the further growth of these thinkings from your work on integration domains. I expect it will be a very good counterweight to many of us in the session that have accidentally slipped in to building "aircraft carriers for the mind"!

++++++++++ REVIEW 6 (Yann Trividic) +++++++++
This paper presents a concise and efficient take by the author on the concept of substrates.

Overall, I found the contribution very clear. I will try to be short and just bring up a few comments.

The term compatibilism is quite well-framed, I would be interested to have the opportunity to discuss it further and to generally give more space to the question of interoperability in the workshop. What kind of compatibility should we strive for?

Many of the papers are addressing the question of substrates at the scale of the stack, whether it is by subverting it or by reinventing it. Nevertheless, addressing a problem at this scale necessarily implies that we have to consider big tech economics. Thus, we have to take into account that fostering a nice, interoperable software ecosystem is not an objective shared by all. In a capitalist world, most incentives can be brought down to making profit. From that observation, how do we actually make robust propositions for substrates that comply with these constraints (or acknowledge them, at least)?

I agree, defining substrates must come through a political proposition, and I would be really delighted to see this workshop bring up such a proposal.

(Same as at least one other reviewer, I prefer signing my reviews, as it is not clear why they should be anonymous in this context!)

++++++++++ REVIEW 7 (Camille Gobert) +++++++++
Sorry for the late comment—I really wanted to leave one earlier but I got caught up by various events!

This statement explores the idea of a "low-level" substrate, at the level of the operating system rather than at the level of a program running on top of it (like most interactive programming environments). It discusses why it matters as well as a number of trade-offs, such as not forming an isolated world/staying lean by rejecting compatibilism.

This statement, just like much of your previous work, really speaks to me—thank you for pursuing this ideal!

I am curious to hear more about what a "distributed" OS-level substrate would be—I'm not sure to grasp the idea. Since you mention local-first software and IP/DNS as inspirations, are you thinking of a world where every meaningful bit of memory has a kind of UUID, not unlike what CRDT libraries such as Automerge do and what Edwards suggests in their statement? If so, how would that fit in a world where most existing objects that live in a computer's memory do not have one: would that go against the goal of something like liballocs? What would be the benefit of it: helping other (possibly remote) programs read and write those pieces of information? Could we achieve something meaningful without changing existing programs in depth?

To further follow what Petricek says in their statement: what would be the barriers that would make this difficult? Why is it not already available, or why did it disappear if it existed before? I feel like there were crazy OS-level ideas a few decades ago (I can't help but think of Plan9 stuff here): do we know why they did not succeed and whether these reasons would still apply today?

I never heard of "compatibilism" before, but I really like the term, thanks! Is there any public reference/recording of whatever taught you this term at LIVE 2024? I feel like I'm getting more and more interested in being a "compatibilist" myself, though I feel like there might be different "sorts": for example, creating a brand new program that helps interact with established file formats feels less problematic to me than advocating for a new ("better") file format.

I have a bunch of other thoughts to share, but I'll probably be easier to chat about that during or after the workshop, if you're interested!