De-escalating software

or ‘the importance of not losing at Tetris’

Stephen Kell
S.R.Kell@kent.ac.uk

University of Kent
Introduction
“When we undertake to write a compiler, we begin not by saying ‘What table mechanism shall we use?’ but ‘What table mechanism shall we build?’ . . . [My vision is that the builder] will be able to say ‘I will use a String Associates A4 symbol table, in size 500x8,’ and therewith consider it done. As a bonus he may later experiment with alternatives to this choice, without incurring extreme costs.”
help i've forgotten what computers are for

HELLO we're symbolics, it's 1982, we just made the greatest personal mind-amplifier that the world will ever know, here's what you can use it for

Symbolics, Inc. designs, manufactures, sells, and supports advanced state-of-the-art, high-performance, single-user computer systems that feature a highly interactive man/machine interface. These systems were designed in response to the growing demand for increasing the productivity of highly skilled professional staff in various high technology disciplines. Present applications include the design of very large scale integrated (VLSI) circuits, symbolic mathematical analysis, genetic engineering, seismic studies for oil and mineral exploration, training simulation, software production, and artificial intelligence research and development. The system design objective, achieved to an extent never before offered commercially, has been to greatly enhance programmer and user productivity.

twitter.com/...
Some well-known programming wisdom:

“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re as clever as [possible] when you write it, how will you ever debug it?”

—Brian Kernighan
from *The Elements of Programming Style* (with P.J. Plauger)
Some well-known programming wisdom:

“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re your compiler is as clever as [possible] when you write it, how will you ever debug it?”
Compilers are very advanced machines

A tiny example due to Chris Lattner...

```c
void contains_null_check (int *P) {
    int dead = *P;
    if (P == 0)
        return;
    *P = 4;
}
```

After optimization, it becomes (effectively)

```c
void contains_null_check (int *P) {
    *P = 4;
}
```

Why? ‘It’s permitted by undefined behaviour in C.’
Why really?

C compilers have become extreme ‘performance squeezers’.

They don’t have to be. It’s counterproductive!

- greater effort per unit product
- harder to debug $\rightarrow$ workarounds, not fixes

And so it escalates:

- increasing ‘expertise’ required of programmer
- more \{complex, fragmented\} tooling
- generate more work to ‘rewrite the C’
- more code $\rightarrow$ more demand for optimisation (!)

We become invested deeper and deeper in this cycle.
Software performance is no longer about infrastructure!

It’s a systemic problem of how software is developed.


Infrastructure gains are soaked up by a ‘software sponge’.

It is not simply ‘saving time to spend on features’.

Escalation ensures features *remain costly to implement*.

The malaise is with the industrial roots of software culture.
Functional languages are no better
Lest you think I was just ranting about the madness of C . . .

We’ve proved ‘well-typed programs don’t go wrong’!

Let’s get rid of those run-time tags . . .
“A few patients survived longer with transplants of various organs. On the other hand, the total social cost exacted by medicine ceased to be measurable in conventional terms. Society can have no quantitative standards by which to add up the illusion, social control, prolonged suffering, loneliness, genetic deterioration, and frustration produced by medical treatment.”

—from ‘Tools for Conviviality’ (1973)
The blame game

Escalators can often be identified by *blaming the human*.

‘Fix your code!’

‘Remember: we work for the machines!’
Another escalator: ‘let’s make a new $X$’

Maybe you don’t like C. So create a new language!

How will people interface with older code? Hmm…

```c
struct Point
{
    int x;
    int y;
};
```
Local<Value> GetPointX(Local<String> prop,
    const AccessorInfo &info) {
    Local<Object> self = info.Holder();
    Local<External> wrap = Local<External>::Cast(self->
    GetInternalField(0));
    void* ptr = wrap->Value();
    int value = static_cast<Point*>(ptr)->x_;
    return Integer::New(value);
}

void SetPointX(Local<String> prop, Local<Value> value,
    const AccessorInfo& info) {
    Local<Object> self = info.Holder();
    Local<External> wrap = Local<External>::Cast(self->
    GetInternalField(0));
    void* ptr = wrap->Value();
    static_cast<Point*>(ptr)->x_ = value->Int32Value();
}
What just happened

We revere the *internal* and denigrate the *external*.

- special word: *legacy*

This disregard is not shared by empirical science

- ‘external validity’

Nor is it shared by all engineers

- design as a discipline

Massively counterproductive.

- ++integration_cost, ++reimplementation
- --tool_power, --maintainability
Forking off a new *whatever* is just ‘what we do’.

It is perceived as a free operation.

Integration is someone else’s problem...

... and affects only people who have themselves to blame.

They should have used the shiny new thing from the start!

It’s the future!
Better ways: possible, but still not done

“Integration is linking your .o files together, freely intercalling functions... you don’t have a foreign loader, you don’t coerce types across function-call boundaries, you don’t make one language dominant, and you don’t make the woes of your implementation technology impact the entire system.

“[All these] can be addressed in a Lisp implementation. This is just not the way Lisp implementations have been done...”

—Richard P. Gabriel
“Lisp: good news, bad news, how to win big”
AI Expert, 1994
The transition from Local to Handle won't happen for a while. It's more of a cleanup step after everything else is done, and there's no urgency since there shouldn't be any performance impact.

The callback signature changes alone break almost every single line of v8-using code I've written (tens of thousands of them), and I am still...
This Is My Jam will become a read-only time capsule on **September 26, 2015**. This means you won’t be able to post anymore, but you’ll be able to browse a new archive version of the site.
But keeping the jams flowing doesn’t just involve our own code; we interoperate with YouTube, SoundCloud, Twitter, Facebook, The Hype Machine, The Echo Nest, Amazon, and more. Over the last year, changes to those services have meant instead of working on Jam features, 100% of our time’s been spent updating years-old code libraries and hacking around deprecations just to keep the lights on. The trend is accelerating with more breaking/shutting off each month, soon exceeding our capacity to fix it.
Illich would say…

Among linked software, there is a radical monopoly

It is a monopoly of the recent.

‘If you can’t keep up with change, that’s your problem.’

This affects anyone on a budget (including researchers).

It’s not ‘your’ problem; it’s one of technologies and tools.

…and the culture which created them

…and the culture which they create.
Energy as inequity

Sometimes, a little project will become ‘hot’.

Investment of effort in a codebase is good, surely?

Maybe not, if it lessens others’ ability to benefit

The more power expended on a codebase

… the more power is needed to use or contribute

Think: Linux kernel, Android, LLVM, …
What can we *do* about all this?

- opt out of society?
- take shelter from the worst?
- join in, and enjoy job security?
Yet more advanced technology...?

The founder of Mint.com, Aaron Patzer, has been researching alternative urban transportation under a company called Swift over the past six months, but he has determined that the personal maglev system he had been envisioning is economically not viable for a company to produce. Patzer described all of his findings and development in a blog post (hat tip to Tech Crunch), including the high economics of such a transportation network.
Illich: “I have chosen ‘convivial’ as a technical term to designate a modern society of responsibly limited tools.”

“Commuter transportation leads to negative returns when it admits, anywhere in the system, speeds much above those reached on a bicycle.”
‘Responsible self-limitation’

We are quite used to this idea.

One example: information hiding

Another example: pure functional programming

These are evidently not the only limitations we need.

They may not even be among the best ones to choose.

To transcend the present, we need new ways to limit ourselves.
Self-limitation 1: against performance-squeezing

It is hard to definitively forbid ‘performance squeezing’.

One idea: for language impls, insist on debuggability.

Ask: what are the *externalities*?

Ask: what story do the metrics not tell?

Performance comes at what cost? (aside: or what COST?)
Self-limitation 2: if it stacks, it must federate

Pre-Internet, sending e-mail across networks was *possible*

...if the right *gateways* were available + running

Deploying *new applications* was beyond the means of most
Self-limitation 2: if it stacks, federate it

IP: an interface that *federated* the network abstraction
- obviated the escalating need for ALGs

What else can we federate?
Self-limitation 2: if it stacks, federate it

My own liballocs project federates memory abstractions

- Unix memory is no longer raw bytes; ‘typed allocations’
- A step towards federating high-level language impls
Federability is also what separates O-O from ADTs…

‘Interoperability’ has been named the essence of object-orientation

- Aldrich, Onward! 2013
Self-limitation 3: degradeable hiding

“The formats of control blocks used in queues in operating systems and similar programs must be hidden within a ‘control block module’. It is conventional to make such formats the interfaces between various modules. Because design evolution forces frequent changes on control block formats, such a decision often proves extremely costly.”

D.L. Parnas

*On the criteria to be used in decomposing systems into modules*

CACM, December 1972
“One of the reasons why many old MIDI instruments continue to be musically viable is... due [to] a means for externalizing the complete state of a musical device: all its patches, voice parameters, and settings. MIDI’s designers only anticipated [these messages’] use as a means for loading and saving patches to and from external storage. In practice, however, this [also] enabled an unexpected ecosystem of third-party, software-based patch editors and alternative control hardware to emerge.”

Colin Clark and Antranig Basman

*Tracing a Paradigm for Externalization*, 2017
Information hiding is a heuristic based on anticipation.

‘I predict these details might change. Hide them.’

‘I predict these details won’t change. Expose them.’

What if our predictions are wrong?

We get this wrong all the time. Interface churn!

Provide a separate door to expose the details anyway!

Not doing so is a recipe for disposability.
Tools for de-escalation

“We must guard against falling into the damaging rejection of all machines as if they were works of the devil.”

It is not a contradiction that software can help de-escalate itself.

Such software should engender much less future programming
Tools for reconciliation

Constantly spawning: abstractly similar, concretely different

How can we reconcile them? Currently: at great cost.

edit or patch

| A | A' | B |

| A | glue code | B |

glue coding

abstraction layer

| A'' | B |

| original target of A |

| B |
McIlroy wanted interchangeable ‘at reasonable effort’

Problems:
- non-1-to-1 mappings
- context-sensitive
- data, not just code

Need tools which (semi-)automate the reconciliation of interface differences.
Tools for integration

Hardware (and other domains)
- chip *invents its view* on outside
- keeps components simple
- … and composable

Software:
- no equivalent
Tools for description

$ man 5 proc

... 

/proc/\[pid\]/maps
A file containing the currently mapped memory regions and their access permissions.

The format of the file is:

... 

If ‘format’ were machine-readable, I wouldn’t have to write:

```c
int nfields = sscanf ( linebuf,
  "%lx-%lx %c%c%c%c %8x %2x:%2x %d %4095[\x01-\x09\x0b-\xff]\n",
  &entry_buf->first, &entry_buf->second, &entry_buf->r, &entry_buf->w, &
  entry_buf->x,
  &entry_buf->p, &entry_buf->offset, &entry_buf->devmaj, &entry_buf->
  devmin,
  &entry_buf->inode, entry_buf->rest);
```

p.47
Culture for de-escalation

“Cultural change” is a problem, not a solution

We need a culture that values empowering individuals

… not providing warm bodies to feed the beast.

There’s a lot of wall to tear down. How?
An unsuccessful tactic: pleading

“With Project Oberon we have demonstrated that flexible and powerful systems can be built with substantially fewer resources in less time than usual. The plague of software explosion is not a ‘law of nature’. It is avoidable, and it is the software engineer’s task to curtail it.”


No doubt deliberate effort *can* build simple software, but

- A new, parallel ecosystem won’t shift culture.
- It contributes to the escalation!
Probably also unsuccessful: embarrassing

My modest proposal: your website should not exceed in file size the major works of Russian literature.
Culture for de-escalation

Those of us who are teachers wield enormous power.

The norm is to teach the 1970s industrial view of software.

... without even acknowledging this as a culture!

Wanted: not just ‘shaping the future’...

... ‘shaping the shaping of the future’!

Programming languages, programming culture.
Teaching for conviviality
We mostly teach \{internal, industrial\} viewpoints

“a project”, “a client”, build “a system”

Performance and reliability seen as internal...

... not systemic effects

‘I optimised it and it runs faster!’

‘I proved it correct!’
Teaching for de-escalation

Most of us build software to teach, and to research

Teachable software is convivial software!

(Research software, not always so much...)

Challenge: make the teachable viable

... not parallel (Oberon), but in the mainstream ecosystem.

Efforts like musl hint at the possibility.

But remember my core message: hopelessness.
The moral of Tetris:

“Development is only sustainable if it makes efforts to conserve complexity”

It is a game we will continue to lose.

Thank you for your indulgence.
Picture credits

Ford assembly line – Literary Review (public domain)
Von Neumann – Alan Richards (via the IAS and CHM)
traffic – public domain (US EPA)
McIlroy speaking – Brian Randell
McIlroy relaxing – Brian Randell
Ken and Den – Peter Hamer (CC BY-SA)
Illich – unknown (fair use)
escalators at Lloyd’s – phogel (CC BY-SA)
Brian Kernighan – Ben Lowe (CC BY 2.0)
bicycle – own work
e-mail gateways – own work
hourglass – Xander89 (CC BY-SA 2.0)
rat’s nest – Lohray (CC BY-SA 3.0) [adapted from]
Niklaus Wirth – Tyomitch
Tetris – based on screenshot by Aldair Tetrovisk (public domain)