Bicycles for the Mind Have to Be See-Through

Kartik Agaram
ak@akkartik.com

ABSTRACT

This paper describes ongoing research on building software to be comprehensible to its users so that they can tailor it to their needs in the field. Our test-bed is a computing stack called Mu that deemphasizes a clean interface in favor of a few global implementation properties: small implementation size, few distinct notations, parsimonious dependencies, a simple dependency graph that avoids cycles, and early warning on breaking changes. Assuming a 32-bit x86 processor and (for now) a basic third-party Unix-like kernel, Mu builds up from raw machine code to a memory-safe but less expressive language than C.

Our approach to keeping software comprehensible is to reduce information hiding and abstraction, and instead encourage curiosity about internals. Our hypothesis is that abstractions help insiders who understand a project but hinder newcomers who understand only that project's domain. Where recent efforts to create "bicycles for the mind" have tended to focus on reducing learning time and effort, we explore organizing the curriculum to be incrementally useful, providing an hour of actionable value for an hour (or three) of study. The hope is that rewarding curiosity will stimulate curiosity in a virtuous cycle, so that more people are motivated to study and reflect on the difference between good vs bad design and good vs bad architecture, even as the study takes place over a lifetime of specialization in other domains. Spreading expertise in design is essential to the creation of a better society of more empowered citizens. Software tools have a role to play in this process, both by exemplifying good design and by providing visceral illustrations of the consequences of design choices.

CCS CONCEPTS

• **Human-centered computing** → **Open source software**; • **Software and its engineering** → **Layered systems**.

KEYWORDS

software literacy

ACM Reference Format:
Kartik Agaram. 2020. Bicycles for the Mind Have to Be See-Through. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3397537.3397547

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3397547

1 INTRODUCTION

> *In order for code to be living structure, even the tools used to make the code need to be living structure.*
>
> Christopher Alexander

In "Tools for conviviality" [12], Ivan Illich describes two schools of tool-making, the means by which humans influence their environment. *Manipulative* tools focus on immediate productivity. They use what's easily available and convenient to use, and build on it. Little attention is given to the process by which tools are created, leading to deep dependency chains of tools that need other tools to be built. Over time, special interests capture the apparatus of tool-making and management. Flaws in specific tools are papered over with more tools. What used to be considered the means toward some other end gradually turn into ends in themselves, their production and nurturing requiring increasing quantities of manpower, from people not motivated to work on this endeavor and therefore needing increasing amounts of management.

In response, Illich proposes an alternative school he calls *convivial* tools. According to him, the primary goal when making a tool shouldn't be to just make some tactical activity more convenient. Instead, it should be to preserve the degrees of freedom of individuals as we explore alternative combinations of human and machine. Individuals should decide for themselves the uses to put their own manpower to. We should celebrate tools that preserve individual agency, and shun tools that reduce human agency. Concretely, if a tool doesn't do quite what you need, don't try to paper over its deficiencies with a second tool. The maintenance burden of both will lead to compounding claims on your time, and on the time of others, thereby reducing the degrees of freedom of human society as a whole. Instead, take the first tool *out*, and think about the problem anew.

Taking tools out of existing workflows is difficult. Our social arrangements incline us to take artifacts for granted once they've been introduced (and experience some level of adoption). People accultured in our society tend to expect a level of specialization, with problems partitioned according to fairly static boundaries. Most people's careers are circumscribed by these boundaries. If we grow used to something that we don't know how to build or manage, the prospect of losing it is painful.

If we stipulate that Illich's goal is desirable, the problem then becomes how to encourage greater dissemination of knowledge about tools. How do we build tools that can be maintained by end users in the field, without support from authors and experts? The Mu project explores one approach to this problem in the domain of software: to support modification in the field, keep everything simple enough to be comprehensible by anyone, without exception. Mu is a computing stack designed from the ground up:

- to have a strict complexity budget, to "fit in a single brain";
- to not grow complex over time; and

- to reward curiosity, and encourage people to understand its internals.

These goals stem from three major influences. In chasing large-scale comprehension without access to original authors, we try to follow the dictums of Peter Naur [19] and add detail to them. In emphasizing degrees of freedom for end users, we follow Christopher Alexander [1], particularly as recounted by Richard Gabriel [6]. In deemphasizing black boxes, we follow the observations of Gregor Kiczales [16] and the dialectic between his criticisms and the lessons of Parnas [20] and others.

This paper reports on progress towards these goals. Our hypothesis is that using fewer abstractions—carefully designed to leak in just the right ways—can make the maintenance task more approachable to end users. Since dependencies introduce their own abstractions, we minimize external dependencies. In particular, since most mainstream software prioritizes use as a black box rather than comprehension of internals, we try to minimize dependencies on mainstream software.

We don't have a definitive conclusion yet on the efficacy of this approach. If we fail to falsify our hypothesis, we'd like for Mu to demonstrate an alternative way for people to collaborate over software: by exchanging complete working stacks (all software running on a computer, with the possible exception of firmware) designed to be manually merged for individual contexts. We hope taking control of the developer experience in this manner will lead to more comprehensible software infrastructure.

Strategies

Mu aims to accomplish these goals using the following strategies:

(1) It almost unfailingly implements high-level constructs out of lower-level ones. Dependencies flow 'down', and we avoid cycles in the dependency graph as far as possible.
(2) It uses as little mainstream code (interface-driven, built with whatever's handy, indirectly depending on C) as possible.
(3) It uses as few notations (languages, syntaxes, intermediate representations) as possible.
(4) It prioritizes safety over syntactic convenience.
(5) It focuses on encoding intention. The ideal: if making a change raises no errors, then no regressions should have occurred. If an error is raised, it should be obvious to an end-user that the expected behavior is superior to the erroneous behavior.
(6) It encourages users to fork it, both technically and socially.

Here's how these strategies support Mu's goals:

- Minimizing dependencies reduces the number of moving parts and therefore the total cognitive load of the stack. Keeping dependencies *decomplected* [10] also aids comprehension. When learning a strange new codebase by oneself, metacircular implementations are hard to understand; they can seem like circular reasoning.
- Mainstream software isn't as disciplined about avoiding or topologically sorting dependencies. Using it violates the previous rationale.
- Mainstream software is usually designed to hide implementation details behind an interface, which goes against our plan to expose implementation details but keep them simple.
- Any mainstream software we introduce is a source of noise when we try to falsify our hypothesis. We'd like to be confident in a negative result rather than wonder if, say, the C compiler we used hindered full-stack comprehension.
- All notations may need at some point to be learned by an end-user, so it seems worth restricting their numbers. While notation is a tool of thought [13], *many different notations* can be a hindrance to thought.
- Deemphasizing syntax reduces implementation size at the lower levels of the stack.
- Deemphasizing syntax causes source code to be closer to generated code, and the programmer's mental model to be closer to the machine. The programmer is habituated to communicate precise intent (and discover it in the process) rather than expect the machine to make (inevitably fallible) assumptions.
- Reified intention (using types and tests) provides early warning of breakage when the system is modified by outsiders; outsiders can't be expected to have the context of the project's development history or capabilities for thorough manual testing.
- Forks provide an escape hatch for sub-communities of end-users to bud off as their needs diverge, deleting features they don't use to compensate for new features they *do* need. Tests can help divergent sub-communities continue to share features with a bounded amount of effort.

Prior approaches

These strategies borrow from much past work. For example, Forth systems [3, 21] emphasize parsimonious dependencies (Strategies 1 and 2 above) but give up on safety (Strategy 4) in the process. Smalltalk systems [7] emphasize safety (like many other high-level languages) while exposing a large fraction of their internals. However, there usually remains a kernel that requires exiting Smalltalk to modify. Lisp Machines [8] built up all the way from custom hardware (Strategy 1) while remaining safe. Lisp, Forth and Smalltalk all emphasize uniform notation (Strategy 3), though they also have strong and divergent opinions on what that notation should be. While they all expose their internals to modification in various structured ways, it seems easy for small modifications to their internals to cause regressions both subtle and catastrophic. Modification requires expertise of all the scenarios their environments are designed to handle, expertise that can only be obtained out of band from the tools themselves.

The STEPS project [15] explored ways to construct a complete stack with low implementation effort. Mu tries to follow in its spirit, while being more parsimonious with dependencies and notations. The work of Basman and Tchernavskij [2] describes a mature society based on forking (Strategy 6) that we aim towards.

2 AN AUSTERE STACK

As mentioned above, Mu is parsimonious with its dependencies. At run time, Mu packages programs either as ELF binaries running on Linux or as bootable disk images bundled with a third-party Unix-like kernel. Mu programs don't rely on libc or any other mainstream libraries. Eventually we will build the OS kernel in Mu as well.

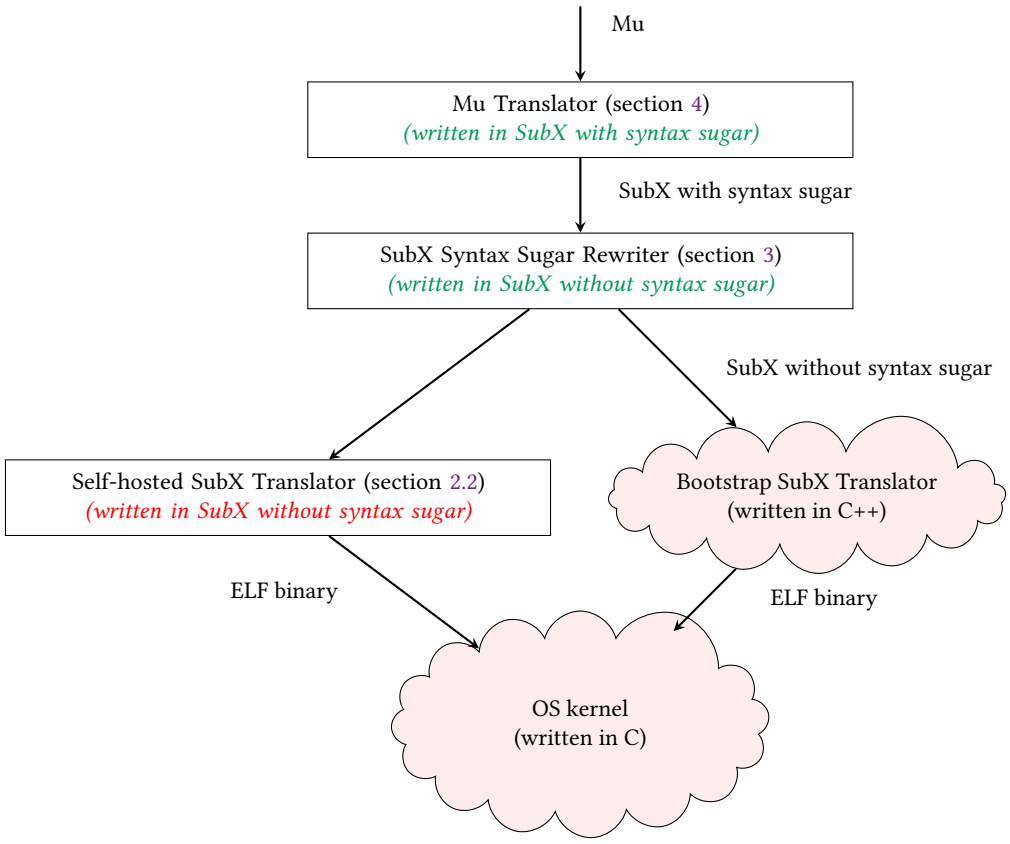


Figure 1: Building Mu programs. Edges represent languages, while nodes represent tools and indicate the language they're implemented in. Most nodes are built in the language they emit (green). Only the self-hosted translator is metacircular (red). The self-hosted and bootstrap translators emit identical ELF binaries given identical source programs. The clouds highlight areas that still depend on mainstream software.

At build time, Mu is designed to be parsimonious with its dependencies but also easy for a newcomer to build. It can be built using two approaches. First, it can be bootstrapped on any Unix-like platform with a C (and C++) compiler. Second, once a Mu machine is bootstrapped, Mu can build itself without any further reliance on C (barring the OS kernel). These two approaches have complementary strengths and weaknesses; C is relatively familiar but C compilers have large dependency trees, while Mu as we'll see later looks strange but only relies on a few generic OS syscalls and is easy to audit. Both approaches emit identical binaries, permitting *diverse double compilation* [24] to counter the Thompson-Karger "trusting trust" attack [14, 22].

Figure 1 shows the toolchain graphically and provides an overview of the rest of this paper. The Mu computing stack provides two notations: an unsafe notation for machine code (called SubX) and a type-safe and memory-safe statement-oriented language (eponymously called Mu) that mostly translates 1:1 to machine code. In each notation we try to do as much as possible with localized rewrite rules (syntax sugar).

2.1 Bedrock: the 32-bit x86 instruction set

Currently Mu supports only the Intel x86 instruction set. Portability is explicitly a non-goal. Portability guarantees require extra effort to maintain, and can be particularly challenging for newcomers who may not be inclined to ensure their changes work on platforms they don't regularly run. Rather than attempt to fight a losing battle, we retreat from it entirely.

Mu supports only a regular subset of the entire x86 instruction set, mostly restricting itself to 32-bit instructions that have been widely available since 1990. Since we also only rely on a bare minimum of features from the OS kernel, Mu programs should be easy to get up and running on any computer people have access to (though they do currently require access to a Unix for bootstrapping). Currently we only support integer operations. We will eventually support a similarly regular subset of floating-point operations.

As Figure 1 shows, at the lowest levels our goal is to come up with a reasonably ergonomic syntax for x86 machine code *that can be implemented in itself*. Given this hard constraint, we don't try to abstract over the underlying machine. Programmers working at this level are exposed to the constraints and complexities of the

instruction set. Given that, it's worth taking some time to take stock of it.

The x86 instruction set is variable-length; instructions may be anywhere from 1 to 14 bytes long and contain various subsets of 13 different *arguments* in addition to a variable-length *opcode* that designates the operation to be performed. Arguments range in size from 2 to 32 bits in length, and multiple arguments are often packed to share a single byte.

It has 8 32-bit registers and 8 overlapping 8-bit registers. It addresses RAM by byte even though most operations read and write 4 bytes at a time.

It supports the usual complement of instructions: arithmetic or logical operations, jumps, pushes and pops, function call and return instructions. Most instructions operate on no more than 2 *operands*.

It supports a fairly baroque set of addressing modes. Each instruction can access at most one memory location. The task of the addressing mode is to determine operands based on arguments in the bitfields of an instruction.

2.2 SubX: a habitable notation for programming in machine code

At this point it's worth taking a step back to think about why we don't all program in machine code, at least in place of unsafe languages. Ignoring aesthetics, there are some essential difficulties:

- Non-textual data is not a good fit for human senses.
- Packing bits into bytes is error-prone.
- A variable instruction set requires path-dependent encoding. Small errors in one instruction can cause later opcodes to silently be interpreted as arguments and vice versa. Errors may manifest unboundedly far from their root causes, making diagnosis intractable.
- Computing offsets for jumps is error-prone.
- Various other error conditions can yield terse or cryptic results.

Given our tight complexity budget for a self-hosted syntax, these difficulties are our top priorities. To begin, we follow a long tradition (e.g. Edmund Grimley Evans [9]) and write programs in a hexadecimal representation of ASCII bytes, converting them to binary before running them. The rest of the stack can now work with programs as textual representations of numbers.

Metadata: While machine code consists of undifferentiated numbers, mixing in some redundant information can help improve the quality and timeliness of errors. Conventional Assembly languages do so by creating *mnemonics* for opcodes and translating mnemonics and operands into machine code. However, Assembly mnemonics can often expand to multiple opcodes depending on the arguments, and translation can get complex. Translating opcodes can also convolute the logic for good error messages. For example, Assemblers may need to inform the programmer that there's just no instruction in x86 to multiply `ecx` by `edx`.

For easy translation and simple but robust error messages, we work with numbers directly, but append *metadata* to them after a slash. Here are some example instructions in SubX, illustrating the simple and the complex:

```
c3/return
```

```
68/push  4/imm32
0f 82/jump-if-lesser  4/disp32
8b/copy  1/mod/*+disp8  1/rm32/ecx  8/disp8  0/r32/eax
```

Instructions always occur one to a line. Only the numbers at the start of each word represent computation encoded in the final binary. The rest of each word is metadata. In the first two instructions the metadata is relatively easy to read. The third example demonstrates an instruction with multiple opcodes. The fourth example looks up memory at the address in `ecx` and saves the result to `eax`.

The SubX translator ignores unrecognized metadata. In the above examples '`/return`', '`/push`' and '`/ecx`' are just comments for human readers. However, metadata can also affect translation by *labeling* arguments to be processed in different ways. As the SubX translator processes opcodes, it checks for expected argument labels. For example, it expects the opcode '68' ('push') to provide a single argument and for that argument to be labeled with '`/imm32`'. Any discrepancies are immediately flagged as translation errors. In more complex instructions, metadata also permit the SubX translator to perform the tedious task of packing bitfields into bytes and ordering them correctly.

Labels: Like Assembly languages, SubX provides facilities for binding addresses to memorable names using *labels* that get automatically replaced with either their absolute address or a displacement relative to the current instruction. Labels naming functions are distinguished. Jumps across functions are illegal, as are calls to labels within functions.

String literals: One major advance over Assembly languages is support for string literals. In SubX you can provide a string literal anywhere an address is expected. The translator creates room for the string literal elsewhere and replaces the reference to it with its new address.

Supporting string literals adds some complexity to the SubX parser and label translation, but they're deemed essential because they enable a key feature: automated tests that can give helpful error messages when they fail. Colocating failure messages next to their use is essential to self-documenting tests.

Tests: As mentioned above, tests are an essential component of programming in machine code using SubX, and they're provided right from the start. The mechanism for tests is merely a special function called '`run-tests`' that is automatically code-generated for each program. When called, it calls all functions in the program that start with the prefix '`test-`'. Given this mechanism, everything else can be built into the vocabulary of functions.

Tests are particularly essential when programming in machine code because there is no task so simple that I[1] get it right the first time.

Summary: That completes our quick tour of the core of SubX: hexadecimal numbers, metadata, labels, string literals and an automatically generated test harness. Figure 2 shows these features interacting in a larger code sample.

[1]This paper uses the first person plural to acknowledge collaboration with others on the Mu project, but the first person singular when referring to its author.

```
factorial:  # n : int -> int/eax
    # . prologue
    55/push-ebp
    89/copy                         3/mod/direct    5/rm32/ebp    4/r32/esp   .             .             # copy esp to ebp
    53/push-ebx
    # if (n <= 1) return 1
    b8/copy-to-eax  1/imm32
    81          7/subop/compare     1/mod/*+disp8   5/rm32/ebp   .           8/disp8       1/imm32       # compare *(ebp+8)
    7e/jump-if-<=  $factorial:end/disp8
    # ebx = n-1
    8b/copy                         1/mod/*+disp8   5/rm32/ebp   3/r32/ebx   8/disp8       .             # copy *(ebp+8) to ebx
    4b/decrement-ebx
    # eax = factorial(n-1)
    # . . push args
    53/push-ebx
    # . . call
    e8/call  factorial/disp32
    # . . discard args
    81          0/subop/add         3/mod/direct    4/rm32/esp  .           .             4/imm32       # add to esp
    # return n * factorial(n-1)
    f7          4/subop/multiply    1/mod/*+disp8   5/rm32/ebp               8/disp8       .             # multiply *(ebp+8) into eax
    # TODO: check for overflow
$factorial:end:
    # . epilogue
    5b/pop-to-ebx
    89/copy                         3/mod/direct    4/rm32/esp   5/r32/ebp   .             .             # copy ebp to esp
    5d/pop-to-ebp
    c3/return

test-factorial:
    # eax = factorial(5)
    # . . push args
    68/push  5/imm32
    # . . call
    e8/call  factorial/disp32
    # . . discard args
    81          0/subop/add         3/mod/direct    4/rm32/esp  .           .             4/imm32       # add to esp
    # check-ints-equal(eax, 120, msg)
    # . . push args
    68/push  "F - test-factorial"/imm32
    68/push  0x78/imm32/expected-120
    50/push-eax
    # . . call
    e8/call  check-ints-equal/disp32
    # . . discard args
    81          0/subop/add         3/mod/direct    4/rm32/esp  .           .             0xc/imm32     # add to esp
    # end
    c3/return
```

Figure 2: An example function (in orange, for computing the factorial of an integer) and a test for it (in green), written in the SubX notation without any syntax sugar. A discipline of tabular organization, label naming and comments at multiple levels of detail help the human manage complexity.

All of SubX's features are implemented twice: once in the bootstrap C++ translator and a second time in the self-hosted translator written in SubX. All these mechanisms were straightforward enough that the two translators are able to stay in sync over a year, build programs reproducibly, and emit identical binaries given identical sources.

The C++ translator requires less than 3k LoC (including comments, whitespace and tests). The self-hosted translator is built as a pipeline of phases, each reading from '`stdin`' and writing to '`stdout`'. Each phase is small and includes thorough automated tests to aid comprehension. From the bottom up:

- *hex*: converts hexadecimal bytes into a binary file. 1400 lines of SubX, 150 excluding comments and tests, 5KB binary.
- *survey*: translates labels into addresses, and computes the ELF header. 5k lines of SubX, 900 excluding comments and tests, 10KB binary.
- *pack*: combines bitfields into bytes. 6k lines of SubX, 840 excluding comments and tests, 7.5KB binary.
- *dquotes*: translates string literals into labels. 2k lines of SubX, 380 excluding comments and tests, 6.5KB binary.
- *tests*: code-generates the '`run-tests`' function. 300 lines of SubX, 130 excluding comments and tests, 6KB binary.

Line counts above don't include the common vocabulary of functions, but its usage is reflected in binary sizes. One caveat with the self-hosted translator: it currently provides no error-checking. We currently rely on the C++ translator for good error messages. Development so far has focused on build- and run-time tools; the development environment will also eventually be ported over.

All told, we've written some 40k lines of machine code using the SubX notation. The crucial hypothesis in designing it was that the *implementation properties* of parsimonious dependencies and minimal metacircularity trump superficial aesthetics of the syntax.

```
4 run:  inst: ff 6/subop/push 1/mod/*+disp8 5/rm32/ebp 8/disp8
4 run:  inst: e8/call scan-next-byte/disp32
4 run:  inst: 81 0/subop/add 3/mod/direct 4/rm32/esp 0xc/imm32
4 run:  inst: 3d/compare-eax-and 0xffffffff/imm32/Eof
4 run:  inst: 74/jump-if-equal $convert-next-octet:end/disp8
4 run:  inst: e8/call from-hex-char/disp32
4 run:  inst: 89/copy 3/mod/direct 1/rm32/ecx 0/r32/eax
4 run:  inst: ff 6/subop/push 1/mod/*+disp8 5/rm32/ebp 0x10/disp8
4 run:  inst: ff 6/subop/push 1/mod/*+disp8 5/rm32/ebp 0xc/disp8
4 run:  inst: ff 6/subop/push 1/mod/*+disp8 5/rm32/ebp 8/disp8
4 run:  inst: e8/call scan-next-byte/disp32
4 run:  inst: ff 6/subop/push 1/mod/*+disp8 5/rm32/ebp 8/disp8
4 run:  inst: e8/call scan-next-byte/disp32
4 run:  inst: 81 0/subop/add 3/mod/direct 4/rm32/esp 0xc/imm32
4 run:  inst: 3d/compare-eax-and 0xffffffff/imm32/Eof
4 run:  inst: 74/jump-if-equal $convert-next-octet:end/disp8
4 run:  inst: e8/call from-hex-char/disp32
5 run:  regs: 0: _____61  1: bdffffe8  2: _______0  3: _______0  4: bdffffd0  5: bdffffd4  6: _______0  7: _______0
5 run:  0x090064cd opcode: e8
5 run:  == label $from-hex-char:check0
5 run:  inst: 3d/compare-eax-with 0x30/imm32/0
5 run:  inst: 7c/jump-if-lesser $from-hex-char:abort/disp8
5 run:  == label $from-hex-char:check1
5 run:  inst: 3d/compare-eax-with 0x66/imm32/f
5 run:  inst: 7f/jump-if-greater $from-hex-char:abort/disp8
5 run:  == label $from-hex-char:check2
5 run:  inst: 3d/compare-eax-with 0x39/imm32/9
5 run:  inst: 7f/jump-if-greater $from-hex-char:check3/disp8
5 run:  == label $from-hex-char:check3
5 run:  inst: 3d/compare-eax-with 0x61/imm32/a
5 run:  inst: 7c/jump-if-lesser $from-hex-char:abort/disp8
5 run:  == label $from-hex-char:letter
5 run:  inst: 2d/subtract-from-eax 0x57/imm32/a-10
5 run:  inst: c3/return
5 run:  popping value 0x090064d2
5 run:  incrementing ESP to 0xbdffffd0
5 run:  jumping to 0x090064d2
4 run:  inst: 89/copy 3/mod/direct 1/rm32/ecx 0/r32/eax
4 run:  inst: ff 6/subop/push 1/mod/*+disp8 5/rm32/ebp 0x10/disp8
4 run:  inst: ff 6/subop/push 1/mod/*+disp8 5/rm32/ebp 0xc/disp8
4 run:  inst: ff 6/subop/push 1/mod/*+disp8 5/rm32/ebp 8/disp8
4 run:  inst: e8/call scan-next-byte/disp32
```

Figure 3: Screenshot of the trace browser for time-travel debugging. Top: some instructions executed, along with a highlighted 'cursor line'. Bottom: the screen after drilling down into the cursor line, revealing details about the '`call`' instruction as well as the instructions executed in the callee (lighter background color). The leftmost column shows the *depth* of each line.

Experience shows that while it may look strange at first, especially when read passively, it's easy to comprehend with a tight interactive loop of making changes and rerunning tests.

2.3 Debugging SubX programs

While it is relatively painless to gain fluency in SubX, debugging programs in machine code remains challenging. The computational substrate is fundamentally unsafe; the only error messages available are the ones that were manually added. Mu's minimal ELF binaries don't include any debugging information. Instead of relying on interactive debuggers, SubX relies on some unconventional tools.

Emulation. Before we built the SubX translator, we first implemented an *emulator* in C (7k LoC) for the subset of x86 machine code supported by SubX. Emulating ELF file loading and instruction execution provides the opportunity for greater error checking.

Traces and time-travel debugging: Emulated runs can be configured to emit a *trace* of instructions as they are executed, including after each instruction the state of registers and symbolic labels as they are reached. Traces cheaply provide the benefits of time-travel debugging, allowing us to step forwards or back through a program's execution at will.

Traces can get quite voluminous. To help slice and dice them we annotate each line of the trace with a 'depth', and use a *trace browser*, a zoomable UI in text mode that folds away instructions at lower depths. This setup allows us to hide the details inside function calls and individual instructions. The trace browser allows us to gain an overview of a run and selectively drill down as needed. Figure 3 demonstrates the trace browser and a drill-down operation.

Watch points: While traces emit register state after each instruction, it's prohibitive to emit the contents of memory. To answer questions about when a location was last modified and by whom, we use *watch points*. As it executes instructions, the SubX emulator monitors for labels that start with a '`$watch-`' prefix. When it encounters one, it saves the computed effective address and starts emitting its value after every time step. This instrumentation allows us to determine, for any point in the trace, the call stack of the instruction that last modified any watched address.

Summary: One attribute these techniques share is a high "power-to-weight ratio". They take very little code to implement, and once implemented they are robust and work reliably. The key skill when using them is one familiar to anyone used to debugging by inserting '`print`' statements: to not be afraid to modify the program as we debug it. A little extra time spent tuning the trace can cause most bugs to become obvious. Tools to scale up debug-by-print seem like a promising area of further research.

3 SYNTAX SUGAR FOR SUBX

As Figure 2 shows, we give up a lot of syntactic ergonomics when using SubX. Now that we have a baseline that satisfies the implementation properties we care about, it's worth trying to reclaim some syntactic niceties at low complexity cost. The passes in this section are no longer implemented in C++, only in SubX. They do their own error handling.

3.1 Addressing modes

As we mentioned above, x86 instructions may have up to 13 logical arguments that help determine up to two operands. The first operand is specified by providing at most one of 7 arguments, specifying either a register or a literal encoded in the instruction itself (`/r32`, `/imm8` `/imm16`, `/imm32`, `/disp8`, `/disp16` and `/disp32`). The bulk of complexity lies in the second ('`reg/mem`') operand, which can be specified by various *combinations* of 7 arguments:

- `/mod`
- `/rm32`
- `/base`
- `/index`
- `/scale`
- `/disp8`
- `/disp32`

The first bit of syntax sugar follows conventional Assembly languages and provides a concise syntax for specifying the '`reg/mem`' operand. Figure 5 shows the grammar for this syntax sugar, with '[]' surrounding optional tokens. Some example expressions:

- `%eax`
- `*edx`
- `*(esi+4)`
- `*(eax + ecx<<2 + 8)`
- `*Total-widgets`

Whitespace is permitted within parentheses, but not immediately after the '`%`' or '`*`' sigils. Metadata is not permitted within parentheses, so as to keep expressions visually short.

```
factorial:  # n : int -> int/eax
    # . prologue
    55/push-ebp
    89/<- %ebp 4/r32/esp
    53/push-ebx
    # if (n <= 1) return 1
    b8/copy-to-eax 1/imm32
    81 7/subop/compare *(ebp+8) 1/imm32
    7e/jump-if-<= $factorial:end/disp8
    # ebx = n-1
    8b/-> *(ebp+8) 3/r32/ebx
    4b/decrement-ebx
    # eax = factorial(n-1)
    # . . push args
    53/push-ebx
    # . . call
    e8/call factorial/disp32
    # . . discard args
    81 0/subop/add %esp 4/imm32
    # return n * factorial(n-1)
    f7 4/subop/multiply-into-eax *(ebp+8)
    # TODO: check for overflow
$factorial:end:
    # . epilogue
    5b/pop-to-ebx
    89/<- %esp 5/r32/ebp
    5d/pop-to-ebp
    c3/return

test-factorial:
    # factorial(5)
    # . . push args
    68/push 5/imm32
    # . . call
    e8/call factorial/disp32
    # . . discard args
    81 0/subop/add %esp 4/imm32
    # check-ints-equal(eax, 120, msg)
    # . . push args
    68/push "F - test-factorial"/imm32
    68/push 0x78/imm32/expected-120
    50/push-eax
    # . . call
    e8/call check-ints-equal/disp32
    # . . discard args
    81 0/subop/add %esp 0xc/imm32
    # end
    c3/return
```

Figure 4: The function and test of Figure 2 rewritten to use syntax sugar for addressing modes.

Such expressions are translated using the following 5 rewrite rules, where words in bold are variables, and '`N()`' converts a string register name to its 3-bit code:

(1) '`%`**`reg`**' ⟹ '`3/mod N(`**`reg`**`)/rm32`'
(2) '`*`**`reg`**' ⟹ '`0/mod N(`**`reg`**`)/rm32`'
(3) '`*`**`Label`**' ⟹ '`0/mod 5/rm32 `**`Label`**`/rm32`'
(4) '`*(`**`reg`** `+ `**`disp`**`)`' ⟹ '`2/mod N(`**`reg`**`)/rm32 `**`disp`**`/disp32`'
(5) '`*(`**`base`** `+ `**`index`** `<< `**`scale`** `+ `**`disp`**`)`'
⟹ '`2/mod 4/rm32 N(`**`base`**`)/base N(`**`index`**`)/index `**`scale`**`/scale `**`disp`**`/disp32`'

Armed with this syntax sugar, we can now rewrite the code of Figure 2 into Figure 4. Tabular organization is no longer required. Implementing this syntax sugar requires 4.6k lines of SubX, 900 excluding comments and tests, and the resulting binary is 9KB large.

3.2 Function calls

The new syntax for addressing modes now enables a nice syntax for function calls. In raw x86 (assuming a relatively standard calling convention), function calls require pushing arguments on the stack, performing a '`e8/call`', and then popping the arguments off the stack. (Results are typically returned in registers.)

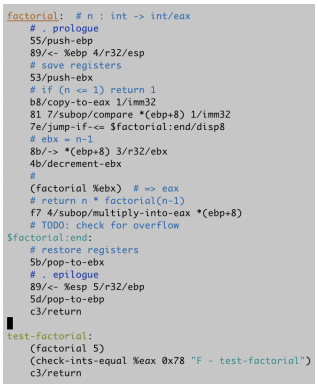
x86 has an instruction to push a '`reg/mem`' operand, which allows us to rewrite a syntax like this:

```
(find-next %eax "/" 3)
```

...into this:

```
reg/mem  ::= direct | indirect | offset | indexed | constant
direct   ::= '%' register
indirect ::= '*' register
offset   ::= '*(' register '+' disp ')'
indexed  ::= '*(' register '+' index ['<<' scale] ['+' disp] ')'
constant ::= '*' label
register ::= 'eax' | 'ecx' | 'edx' | 'ebx' | 'esp' | 'ebp' | 'esi' | 'edi'
label    ::= non-register identifier
disp     ::= 32-bit integer
scale    ::= 2-bit integer
```

Figure 5: Grammar for addressing-mode syntax sugar.



```
factorial:  # n : int -> int/eax
    # . prologue
    55/push-ebp
    89/<- %ebp 4/r32/esp
    # save registers
    53/push-ebx
    # if (n <= 1) return 1
    b8/copy-to-eax 1/imm32
    81 7/subop/compare *(ebp+8) 1/imm32
    7e/jump-if-<= $factorial:end/disp8
    # ebx = n-1
    8b/-> *(ebp+8) 3/r32/ebx
    4b/decrement-ebx
    #
    (factorial %ebx)  # => eax
    # return n * factorial(n-1)
    f7 4/subop/multiply-into-eax *(ebp+8)
    # TODO: check for overflow
$factorial:end:
    # restore registers
    5b/pop-to-ebx
    # . epilogue
    89/<- %esp 5/r32/ebp
    5d/pop-to-ebp
    c3/return

test-factorial:
    (factorial 5)
    (check-ints-equal %eax 0x78 "F - test-factorial")
    c3/return
```

Figure 6: The function and test of Figure 4 rewritten to use syntax sugar for function calls.

```
# push args in reverse order
68/push             3/imm32
68/push             "/"/imm32
ff 6/subop/push     %eax
# call
e8/call             find-next/disp32
# pop args
81 0/subop/add      %esp                0xc/imm32
```

Observe the clean composability of string literals, addressing mode expressions and numbers. The only minor details here are selecting the right push opcode for literals vs non-literals, and computing the number of bytes to pop off the stack.

Armed with this syntax sugar, we can now rewrite the code of Figure 4 into Figure 6. Low-level comments in grey now turn rare. Implementing this syntax sugar requires 1.8k lines of SubX, 450 excluding comments and tests, and the resulting binary is 7KB large.

3.3 Structured control flow

The final bit of syntax sugar that seems to provide a high power-to-weight ratio is the elimination of unstructured jumps in favor of structured conditionals and loops. Assembly languages either don't provide syntax for structured control flow, or carve out exceptions to their statement-oriented nature to support recursive syntax. SubX does neither, but instead supports a simpler syntax that sticks to the statement-oriented nature of the underlying machine. The conventional syntax for a conditional:

```
if (%eax == 0) {
  ...
}
```

...is expressed in SubX as:

```
{
  81 7/subop/compare      %eax          0/imm32
  75/jump-if-not-equal    break/disp8
  ...
}
```

Each 'break' label is translated into the location of the containing '}'. Similarly, the conventional syntax for a loop:

```
while (%eax == 0) {
  ...
}
```

...is expressed in SubX as:

```
{
  81 7/subop/compare      %eax          0/imm32
  75/jump-if-not-equal    break/disp8
  ...
  eb/jump                 loop/disp8
}
```

The 'loop' label is translated into the location of the containing '{':

```
_loop1:
  81 7/subop/compare      %eax          0/imm32
  75/jump-if-not-equal    _break1/disp8
  ...
  eb/jump                 _loop1/disp8
_break1:
```

The SubX translator ensures that curly brackets are balanced, and replaces each matching pair with a unique pair of label names.

Armed with this syntax sugar, we can now rewrite the code of Figure 6 into Figure 7. Labels now become rare. Implementing this syntax sugar requires 360 lines (using syntax sugar for function calls), 120 excluding comments and tests, and the resulting binary is 6KB large.

Summary: This concludes our tour of SubX, spanning the core notation as well as syntax sugar. To translate SubX programs into ELF binaries, we pass them through a shell pipeline composed of all the phases outlined so far:

```
factorial:  # n : int -> int/eax
    # . prologue
    55/push-ebp
    89/<- %ebp 4/r32/esp
    # save registers
    53/push-ebx
    # if (n <= 1) return 1
    81 7/subop/compare *(ebp+8) 1/imm32
    {
      7f/jump-if-greater break/disp8
      b8/copy-to-eax 1/imm32
    }
    # if (n > 1) return n * factorial(n-1)
    {
      7e/jump-if-lesser-or-equal break/disp8
      8b/-> *(ebp+8) 3/r32/ebx
      4b/decrement-ebx
      (factorial %ebx)  # => eax
      f7 4/subop/multiply-into-eax *(ebp+8)
    }
    # restore registers
    5b/pop-to-ebx
    # . epilogue
    89/<- %esp 5/r32/ebp
    5d/pop-to-ebp
    c3/return

test-factorial:
    (factorial 5)
    (check-ints-equal %eax 0x78 "F - test-factorial")
    c3/return
```

Figure 7: The function and test of Figure 6 rewritten to use syntax sugar for structured control flow.

```
cat $* |braces |calls |sigils \
  |tests |dquotes |pack |survey |hex > a.elf
```

Except for the bootstrap 3kLoC C++ translator, the entire stack and any programs running on it require zero external dependencies for building. Only a minimal Unix-like OS kernel is required for running.

4 MU: A LOW-LEVEL YET TYPE-SAFE AND MEMORY-SAFE LANGUAGE

SubX helps us minimize dependencies on mainstream software. In this section we describe how we build on SubX to obtain a memory-safe language: Mu. Since we now have a fairly habitable [5] notation, we can expand our ambitions a bit. However, we're still fairly parsimonious in the features we introduce since the implementation language is unsafe.

Mu's trajectory follows that of early C to some extent; both languages were implemented without a high-level language, and so implementation simplicity was important. However Mu learns from the trajectory of C, where compilers became metacircular at the first opportunity, and the availability of a high-level language for implementation has caused ever-smarter compilers, at the cost of explosive growth in compiler complexity. We don't want to make the same trade-off.

To avoid ever needing a complex compiler, Mu is designed to *never* need any compiler optimizations. It achieves this aim by staying as close to the machine as possible (without compromising memory safety). In particular, we aim as far as possible for each statement of safe Mu to expand to a single instruction of unsafe SubX.

One consequence of aspiring to a 1:1 mapping with machine code: we don't abstract away registers. Since x86 machine code constrains instructions to no more than one memory operand, it makes sense to make programmers manage registers explicitly. We'll still *verify* the register allocation, as described below.

4.1 Syntax

Figure 8 shows a small example program in the safe language Mu. Just like in C, programs are sequences of function, type and global variable declarations. Variable definitions put the type after the name, separated by a ':'. They may also specify a register that the variable occupies, separated by a '/'. Variables not in a register will be placed in memory.

```
type point {
  x: int
  y: int
}

fn higher a: point, b: point -> result/eax: int {
  var ay: int
  ay <- get a, y
  var by/eax: int <- get b, y
  compare ay, by
  {
    break-if <=
    result <- copy ay
    return
  }
  result <- copy by
}
```

Figure 8: An example program in Mu's safe language.

Figure 8 illustrates a few constraints, mostly imposed by the x86 processor. Functions must always get their arguments on the stack, and must always return results in registers. The registers they return their results in are part of their signature, and every call must conform to such constraints. Some primitives (such as 'get' here) can return results in arbitrary registers. No primitive can operate on more than one variable in memory. Remaining operands must be in registers. We have special instructions for accessing members of records ('get') and arrays ('index'). Even though they may translate to the same opcodes under the hood, giving them separate names helps document intention and enforce type-correctness. Conversely, the 'compare' instruction translates to different opcodes depending on whether the first or the second instruction lies in memory. (We continue to use SubX's scheme for structured control flow, abstracting the opcodes as 'break' and 'loop' instruction families rather than labels.)

4.2 Type system

Mu has a strong but unambitious type system. The only goal is to avoid the memory corruption issues that plague the level below. Type-checking happens in a pass between parsing and code-generation, and for the most part consists of matching types between the left and right hand side of instructions. When the instruction is a function call, the type-checker takes the function signature into account. When the instruction is a 'get', the type-checker takes the record definition into account.

Individual instructions impose their own constraints on the types they require. For example, while the underlying SubX opcodes allow any two values to be added together, Mu enforces that the 'add' instruction can only operate on and yield 'int' values.

The 'index' instruction always performs bounds-checking at run time; this constitutes an exception where a Mu statement requires more than one SubX instruction to implement.

Types can be compound, and we express them as s-expressions, for example '(array int)'. An 'index' instruction on a variable of type '(array point)' yields an output of type 'point'.

Mu will have sum types or tagged unions; some superficial syntactic details remain to be finalized, such as the choice of keywords for defining and reading them. We're also trying to integrate Ceylon-style [17] anonymous union types like 'int|err'.

```
fn higher a: (addr point), b: (addr point) -> result/eax: int {
  ...
}
```

Figure 9: Figure 8 modified to operate on aliases of its arguments. Calls remain unchanged.

4.3 Variable declarations

SubX creates space for local variables using 'push' instructions, and reclaims them using either 'pop' instructions or by updating the stack registers. In the safe language we disallow the 'push' and 'pop' instructions, as well as direct access to the 'esp' and 'ebp' registers that manage the stack. Instead, the 'var' keyword updates the stack pointer based on the type of the variable being declared, and all local variables are reclaimed at the next enclosing '}'.

```
# variable on stack; no initializer allowed
var x: int

# variable in register; must be initialized
# with a valid instruction
var x/eax: int <- copy 0
```

All variable declarations automatically initialize their underlying memory at run time (arrays and strings are prefixed with a length) and automatically reclaim it when exiting their block; this constitutes a second exception where a Mu statement requires more than one SubX instruction to implement.

4.4 Addresses

A key aspect of memory-safety is managing addresses. Addresses can be used for three purposes: to reduce copying, to alias variables, and to manage long-lived variables on the heap. In Mu we try to separate these intentions as far as possible using two address types: 'addr' and 'handle'. Functions arguments with type 'addr' indicate call by reference. Figure 9 shows how Figure 8 would be modified to accept its arguments by reference.

Values of type 'addr' are intended to be short-lived. They cannot be saved inside user-defined types. To preserve safety, Mu tracks functions that could reclaim memory and raises an error if an 'addr' variable's lifetime intersects with *any* memory reclamation.

The second address type is the 'handle'. It is the result of a heap allocation, and it's the only kind of address that can be saved inside a user-defined type. Handles are fat pointers; they include an *allocation id* [18] that is also present in their payload. Handles can't access their payload directly; they must first be converted to an 'addr' using the 'lookup' instruction. The 'lookup' instruction constitutes the third and final exception where a Mu statement requires more than one SubX instruction to implement. On every 'lookup' instruction, Mu compares at runtime the allocation id of the handle with the allocation id of the underlying payload. A mismatch in allocation id indicates an attempt to access memory that has been reclaimed, and causes the entire program to abort. Aborting immediately on an attempt to corrupt memory prevents many security vulnerabilities and simplifies debugging when compared to C.

```
# before
{
  var x/eax: int <- copy 0
  break-if =
  ...
}

# after
{
  50/push-eax                # save
  b8/copy-to-eax 0/imm32     # initialize
  {
    74/jump-if-equal break/disp8
    ...
  }
  58/pop-to-eax              # restore
}
```

Figure 10: A more complex translation with variables and control flow.

4.5 Blocks

SubX simulates blocks using '{' and '}' labels. Mu makes such blocks real syntactic entities that are safe to use. Any variables declared within a block are always reclaimed when exiting it. For example, code like this:

```
{
  var x: int
  ...
}
```

is translated to SubX code like this:

```
{
  68/push  0/imm32                  # allocate
  ...
  81 0/subop/add  %esp  4/imm32     # reclaim
}
```

Register variables also require similar stack management, this time to save any shadowed variables and restore them on block exit.

While the above rewriting reclaims variables when getting to the bottom of a block, we also need to handle early exits using the 'break' and 'loop' instructions. Figure 10 shows an example of such a translation.

Mu is designed to ensure that any live registers entering a block were written to as the same variable. It also ensures that live registers entering any loop blocks (loop-carried dependencies) were written to as the same variable when exiting the loop block. These two checks suffice to validate the manual register allocation performed by the programmer.

Between initializing memory, validating registers and handling early exits, the interplay between blocks and variable declarations is the most complex sub-system in the Mu translator.

Summary: Between this section and the last, I've now finished describing all of the Mu stack at a fairly low level of detail. I hope I have demonstrated that one can get approximately to the level of C (lower expressiveness but higher safety) with a modest outlay of code and external dependencies, and that it is possible to explain how it all works in a matter of a dozen pages.

5 REIFYING INTENTION

While we now have a stack that fits in a single person's head, it's not yet a fair comparison with mainstream software because all

software starts out small and simple. We wouldn't want to pit the quality of Mu against that of early C and Unix implementations. Might Mu too grow bloated and complex over say a decade, particularly in the unlikely event it achieves significant adoption? Complexity often stems from the arrival of newcomers and the gradual forgetting of the causes for a design [19]. This section describes some pervasive mechanisms in the implementation of Mu to combat complexity creep over time, and to educate people about rationales at all levels of the stack.

5.1 Error messages

One source of complexity in mainstream software is the pursuit of syntax. New notations are often marketed in a 'dead' form [23] like a webpage or book, where one must judge them by how they look, without interacting with them in any way. The need for such marketing in turn drives the design of surface syntax. I think much of this pursuit is misguided:

(1) Design effort spent on surface syntax comes at the expense of other considerations.
(2) Syntax often requires relatively non-leaky abstractions, which increase implementation complexity.
(3) Syntax creates greater impedance mismatch between the interface exposed to the programmer and the work done for the programmer. Bridging this gap becomes harder when tools must communicate with the programmer.

Mu explores a contrary approach. It's designed from the ground up to minimize impedance mismatch at all levels. As a result, the error messages are easy to write. We can assume more knowledge from people, because we front-load education to optimize for the long term rather than the initial experience. Several design choices in Mu illustrate this dynamic:

- SubX requires opcodes rather than mnemonics. While mnemonics look like English words and are therefore 'friendly' at first glance, natural language is often misleading when applied to programming. The instruction 'add' has many sharp edges in Assembly language that the English word 'add' fails to encompass. Using opcodes requires some initial orientation but, I hypothesize, will provide fewer unpleasant surprises over a lifetime of use.
- Mu and SubX are statement-oriented and provide minimal syntax, so that translation steps are tractable to present to users. The entire toolchain is designed to be transparent, to mention locations of temporary files, and to encourage people to look inside them after a compiler error. Developers who benefit from these features today will hopefully contribute to preserving them in future.
- SubX's ability to emulate machine code programs and emit traces allows error messages to be terse and refer people to the trace of an execution. Traces and assumptions about a more active user reduce the amount of infrastructure needed to present stack frame information on an error.
- Since people are constantly encouraged to browse large traces using the packaged zoomable UI, the details of how Mu programs are run are more likely to be in the programmer's mental model, and therefore can be referred to in error messages.

Arguably the work done so far represents the easy part of the task. Any high-level expression-oriented language built atop Mu will need a lot more work to preserve the sense of transparency while performing more complex transformations on user code. Initial experience, however, is promising. Where modern compilers grow more complex and so need to manage more state for error messages which in turn causes greater complexity, Mu shows that transparent implementations can cause people to understand details and so require simpler error messages.

5.2 Testable interfaces

Conventional wisdom today is that automated tests are important. Conventional wisdom also advises us to test business logic, not I/O. What about programs that perform lots of I/O? Most programs in my experience tend to have complex I/O flows, whether they're webservers going through multiple levels of proxying, web applications that need to render on multiple browsers, or desktop software like text editors. In all these cases the ability to test I/O must be slowly recreated over time using heavyweight frameworks like Selenium. In spite of all our efforts, tests don't detect all possible failures, as evidenced by the pervasive use of manual certification, release candidates and canarying when deploying programs.

I argue that the conventional wisdom to test business logic, not I/O is a consequence of OS interfaces that predate the modern emphasis on tests. If our OS interfaces were designed to be testable, testing I/O would be as lightweight as any other kind of test.

While Mu doesn't yet implement an OS kernel, it wraps conventional OS syscalls to research the benefits of more testable interfaces. Our key pillars are dependency injection and fakes. For example, printing to screen shouldn't be written as:

```
print("hello")
```

It should be written:

```
print(REAL_SCREEN, "hello")
```

Explicitly passing in an identifier for real hardware now allows the interface to support passing in *fake* hardware in automated tests.

Unix's insistence on everything being a file is a hindrance here. We don't even have a syscall to print to screen! SubX's lowest-level helpers for writing to '`stdout`' accept either a file descriptor or a stream object. As a result, Mu's tests pervasively write to a stream and then check its contents to validate results.

A previous prototype went much further than this. Figure 11 summarizes the modalities that we have researched in the past, and the best interfaces we were able to design.

Much work remains here. When Mu eventually gets an OS kernel, we'll need testable interfaces not just for high-level abstractions like a hard disk but also for internal details like disk buffers. They will have to permit encoding a wide variety of intentions, such as what exactly we would like to happen when, say, a '`sync()`' syscall is executed. Under what circumstances will a computer lose data? Unix makes this question hard to reason about.

Not all interfaces to the OS are syscalls. We'd like to be able to simulate a context switch between two specific instructions, or run the component under test in a sub-process that we can then interrupt at will to check if it's blocked waiting on input from a specific channel (e.g. the keyboard). Again, we have some high-level

syscall	new dependency injected	fake for the dependency
`print()`	screen	2D array of characters
`getchar()`	keyboard	byte stream
`exit()`	opaque descriptor	continuation
socket calls	resources	map from URLs to contents
`read()`	file system	map from URLs to contents

Figure 11: Some planned syscalls for the Mu computer.

answers here in the context of a virtual machine, but need more work to arrive at the right interface on a real system.

5.3 White-box tests

A second bit of conventional wisdom around testing is to test only what can be robustly tested. As a consequence, mainstream software doesn't write automated tests for performance. Mu's traces help here. So far we've only discussed traces of machine code instructions run in the emulator (Figure 3), but all Mu programs emit a trace, a versatile append-only in-memory log of structured *facts* deduced about the domain during the course of a program. Automated tests can now inspect the trace and check not just the final result but also that it was deduced in the correct manner. This approach gives us two benefits:

(1) We can now write stable and robust tests denominated in the units of our choice. For example, a '`sort`' function may emit an event to the trace on every swap, thereby permitting a performance test to check that doubling the size of the input doesn't cause the number of swaps to quadruple. Traces are useful not just for performance, but for say testing that a triangle is visible in a graphics pipeline, or that a failover occurred correctly when the master crashed in a distributed system.
(2) We can now write fine-grained unit tests without making assumptions about architecture. For example, Mu's parser emits tokens to the trace. Tests run not just the parser but all of the translator and check just the events in the trace *labeled* under the *namespace* of the parser. Arranging the test in this manner gives us the flexibility to radically reorganize the translator, moving to a parallel or lazy parsing implementation. Conventional tests would need to be rewritten in such situations. With traces they don't. (Mu is helped here by having lightweight fake hardware that allows integration tests to run almost as fast as unit tests.)

We aren't just treating the component under test as a black box but inspecting its internals. In combination with testable OS syscalls, white-box tests can check that a function doesn't allocate memory, or that an email is sent even if the disk is full, or that a function doesn't block on I/O, or that a context switch at a specific point to a specific process doesn't cause a data race.

5.4 Cherishing forks

Most software today is on a trajectory of ever-increasing complexity. Features are rarely deleted. Over time complexity invariably reaches a threshold where new features take longer and longer to add. At this point the project becomes much more selective about adding new features. The result of this trajectory is that early users have a huge impact on the feature set of a program, often long after they stop using it! Compatibility, while useful in moderation, has over time the effect of ossifying all the degrees of freedom.

Much of this trajectory stems from people's reluctance to modify software, a tendency Mu works hard to counteract. Given a stack that is easy to understand, it's worth revisiting the value of compatibility. If people can modify a project for their needs, the next step is to start deleting features they don't use. In the process they can reclaim vitality that has ossified under a "high feature load".

This hypothesis remains untested so far. Mu has no forks yet. But the intent is to be extremely encouraging of forks, and to help them take on an independent existence. Normally features the original authors don't like tend to languish. If forks become easy to create, then it's easy to spin off forks, and to direct demand for specific features to specific forks. Managing changes between long-lived divergent forks is an open problem, but I suspect its difficulty is over-rated.

5.5 Summary

I'm more certain of this section than any other in this paper. Even if Mu's specific architectural choices turn out to be wrong, the mechanisms of forks, traces, abstraction-friendly OS interfaces and abstraction-hostile error messages seem compelling for Illich's agenda of promoting conviviality and managing the complexity of our software supply chains.

6 CONCLUSION

> *Before I built a wall I'd ask to know*
> *What I was walling in or walling out…*
>
> Robert Frost

Among programmers there's a long-standing lament about the excessive complexity in software projects. Ivan Illich points out that this tendency towards complexity isn't restricted to software, but rather stems from sociopolitical causes: the need to structure society with certain specialization and compatibility constraints. Given these constraints, Conway's Law [4] takes over. The complexity of our software infrastructure largely reflects the complexity in our social arrangements.

The original sin is to try to decompose a system into an "inside" and an "outside". While the instinct is laudable, it invariably leads to specialization. Repeat it over and over, and the number of such boundaries compounds, constraining us all.

What is to be done? Mu attempts a high-risk maneuver I think of as "Conway's Law in reverse". We stubbornly refuse to provide the

usual dichotomies of inside vs outside, and by this means hope the benefits of this approach (if they exist) cause society to consider it, with a very slow cascade of consequences resulting in a competing social arrangement.

This paper has demonstrated Mu deconstructing the dichotomy between "inside" and "outside" on 6 different levels:

(1) By selecting notations at the lowest level for *implementation properties* (minimal dependencies, simple dependency graphs) rather than clean interfaces.
(2) By using traces to show details rather than modules to hide details.
(3) By making tests white-box rather than black-box.
(4) By encouraging the distribution of whole computers rather than packages.
(5) By encouraging users to fork open source software rather than submit to a centralized governance process.
(6) By optimizing for learnability rather than familiarity.

In the process, we've taken on some risks. Mu programs are aesthetically unappealing. Debugging may be harder with this approach. Mu makes new kinds of errors possible: traces that lie about what a program did, opcodes that are easy to mistype so they don't match their metadata, and so on. But if we truly believe that the software development process should prioritize comprehension over ease of authorship, it's worth exploring what we gain by taking on these drawbacks.

While Mu's build and runtime environment is fairly independent of mainstream software, it still relies on a host machine for the *development* environment. Tools like the emulator and trace browser require C++, as does getting error messages during development. Over time these tools will be reimplemented in Mu.

Higher-level languages and tools may require greater complexity to implement. If we can't keep them comprehensible, we'll keep them out. High-level features provide flexibility and are particularly useful for prototyping. We will let mainstream software support prototyping use cases, and focus Mu's niche on converting prototypes into sustainable personal infrastructure.

The presence of run-time checks may well result in a stack that is slower than the mainstream, in spite of the performance benefits of streamlining the stack vertically (fewer layers) and horizontally (more forks, less complexity per fork). Even if it's slower, it's worth questioning what speed buys us. Unconstrained growth in performance is a political and economic goal for those selling computation by volume. If we each had one computer to truly call our own, truly running for our benefit, would it really run at full throttle day in and day out? Our experience with Mu suggests that it might be better to treat performance—and much else that we tend to obsess over—as something to satisfice rather than optimize.

Creating an entire new stack may seem like tilting at windmills, but the mainstream Software-Industrial Complex suffers from obvious defects even in the eyes of those who don't share our philosophy. Projects tend to accumulate inertia and slow down over time. Security is a constant concern, and review is lacking in spite of source code being increasingly open and available. Vulnerabilities when found often have a high blast radius. These problems seem foundational, and they are reasons for anyone to explore starting anew [11].

ACKNOWLEDGMENTS

While this paper has just one author, I've used the first-person plural when describing the Mu project to acknowledge code, suggestions and other contributions from many people. In particular: Max Bernstein, Caleb C, Ella C, Remo Dentato, Mek Karpeles, Dave Long, Stephen Malina, Jeremiah Orians, Nick P, Venkatesh Rao, Charles Saternos, Kragen Sitaker. I'd also like to thank the anonymous reviewers and everyone who gave feedback on various versions of the argument over the years. Special thanks to Brandon Hudgeons and Konrad Hinsen for going over the draft with a fine-tooth comb. Finally, this work would not have been possible without the support and forbearance of my wife, Rebecca, over many years of yak-shaving.

REFERENCES

[1] Christopher Alexander. 1979. *The Timeless Way of Building.* Oxford University Press, New York.
[2] Antranig Basman and Philip Tchernavskij. 2018. What Lies in the Path of the Revolution. In *Proceedings of the 29th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2018).*
[3] L. Brodie. 2004. *Thinking Forth.* Fig Leaf Press, Forth Interest Group.
[4] Melvin E. Conway. 1968. How Do Committees Invent? *Datamation* (April 1968). http://www.melconway.com/research/committees.html
[5] Richard P. Gabriel. 1998. Habitability and Piecemeal Growth. In *Patterns of Software: Tales from the Software Community.* Oxford University Press.
[6] Richard P. Gabriel. 1998. The Quality Without A Name. In *Patterns of Software: Tales from the Software Community.* Oxford University Press.
[7] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., USA.
[8] Richard D. Greenblatt, Thomas F. Knight, John T. Holloway, and David A. Moon. 1980. A LISP Machine. *SIGIR Forum* 15, 2 (March 1980), 137–138. https://doi.org/10.1145/1013881.802703
[9] Edmund Grimley-Evans. 2001. Bootstrapping a simple compiler from nothing. http://web.archive.org/web/20020625203905/http://www.rano.org/bcompiler.html
[10] Rich Hickey. [n.d.]. Simple made Easy. ([n. d.]). https://www.infoq.com/presentations/Simple-Made-Easy Strange Loop, 2011.
[11] A.O. Hirschman and American Council of Learned Societies. 1970. *Exit, Voice, and Loyalty: Responses to Decline in Firms, Organizations, and States.* Harvard University Press.
[12] Ivan Illich. 1973. *Tools for Conviviality.*
[13] Kenneth E. Iverson. 1980. Notation as a Tool of Thought. *Commun. ACM* 23, 8 (Aug. 1980), 444–465. https://doi.org/10.1145/358896.358899
[14] P.A. Karger and R.R. Schell. 1974. *Multics security evaluation: Vulnerability analysis.* Technical Report ESD-TR-73-193. Electronic Systems Division, Hanscom Air Force Base. https://doi.org/10.1109/CSAC.2002.1176286
[15] Alan Kay, Dan Ingalls, Yoshiki Ohshima, Ian Piumarta, and Andreas Raab. 2006. *Steps toward the reinvention of programming.* Technical Report. Viewpoints Research Institute.
[16] Gregor Kiczales. 1994. Why are black boxes so hard to reuse? Invited Talk, OOPSLA.
[17] Fabian Muehlboeck and Ross Tate. 2018. Empowering Union and Intersection Types with Integrated Subtyping. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018). https://doi.org/10.1145/3276482
[18] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. *SIGPLAN Not.* 45, 8 (June 2010), 31–40. https://doi.org/10.1145/1806651.1806657
[19] Peter Naur. 1985. Programming as theory building. *Microprocessing and microprogramming* 15, 5 (1985), 253–261. https://doi.org/10.1016/0165-6074(85)90032-8
[20] D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623
[21] Kragen Sitaker. 2008. StoneKnifeForth. https://github.com/kragen/stoneknifeforth
[22] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (Aug. 1984), 761–763. https://doi.org/10.1145/358198.358210
[23] Bret Victor. 2012. Stop drawing dead fish. In *ACM SIGGRAPH.* http://worrydream.com/#!/StopDrawingDeadFish
[24] David A. Wheeler. 2009. *Fully Countering Trusting Trust through Diverse Double-Compiling.* Ph.D. dissertation. George Mason University, Fairfax, VA, USA.