Issue 043: Types

Adrian Kosmaczewski

April 4th, 2022

Welcome to the forty-third issue of *De Programmatica Ipsum*, dedicated to the subject of *Types*. In this edition:

- Graham crafts a Motorola 6809 CPU emulator[1] to show that types are useful, but not needed.
- Adrian reviews history and literature[2] to show that types are useful, but should not be abused.
- In the Library section[3], Graham reviews "Recoding Gender" by Janet Abbate.[4]

This magazine denounces and abhors Russia's invasion of Ukraine. We urge Russia to withdraw its troops, and NATO to stop its expansion immediately. There must be an immediate reduction of nuclear weaponry worldwide, and we call on all sides to de-escalate and seek peaceful solutions. We express our deepest solidarity to all those campaigning for an end to the war, often under very difficult conditions, in Russia and Ukraine.

Enjoy this issue! Please subscribe to our free newsletter[5] to stay updated about new releases, share the articles on social media, or contribute[6] if you would like to support our work.

[1] https://deprogrammaticaipsum.com/it-is-not-the-types-that-will-help-you/
[2] https://deprogrammaticaipsum.com/apples-and-oranges/
[3] https://deprogrammaticaipsum.com/category/library/
[4] https://deprogrammaticaipsum.com/janet-abbate/
[5] https://deprogrammaticaipsum.com/newsletter/
[6] https://deprogrammaticaipsum.com/contribute/

Cover photo by Uday Awal[7] on Unsplash[8].

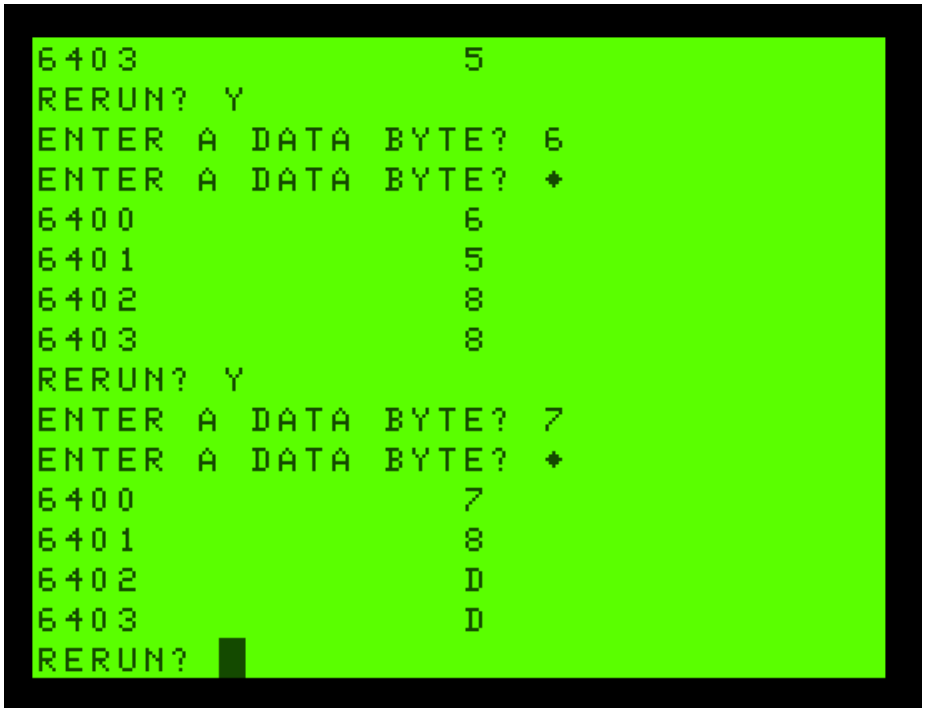
[7]https://unsplash.com/@udayawal?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[8]https://unsplash.com/s/photos/c%2B%2B-code?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

It Is Not The Types That Will Help You

Graham Lee

April 4th, 2022



We go back and forth, if you will pardon the pun, on whether programs have types or not. In the very early days of digital computers you had digits (as likely to be decimal or octal as binary), and that was your lot. But then still in the very early days of digital computers you had LISP, which has two types: atoms and lists. In the sort of early days of digital computers you got BASIC which has a whopping three types: numbers, strings, and arrays.

Early proponents of types as useful tools came from the C programming community. In the August 1983 Byte magazine, Johnson and Kernighan[1] advocate for C's type system as a tool for creating "higher-level models", programs that act on representations of ideas rather than on bytes and words.

Of course, this was 36 issues after the Smalltalk[2] edition of Byte, which had told the world about a *very* expressive modelling system for higher-level abstractions that had exactly four types: Objects, Classes (which worked like Objects), Small Integers, and Symbols. Later, the Self language would demonstrate that this was one too many and that Classes are not actually needed.

See, types themselves are not actually higher-level models, they are constraints on the programs you are allowed to write. There is an uncountably large (though not infinite, assuming you have a practical computer) number of computer programs you could write. Of these,

[1] https://archive.org/details/byte-magazine-1983-08/page/n49/mode/2up
[2] https://deprogrammaticaipsum.com/issue-25-smalltalk/

nearly all of them are wrong, in that if you want a program that does something, these programs do not do that. But, good news, some of them are right! You just have to find the right ones.

Imposing types in particular places like the parameters to functions or the values of temporary variables constrains the programs you create to ones that satisfy the limitations of the type. No longer can your program put any old value into the rax register then call Employee#give_raise: now it must ensure that that value is a pointer in memory to an instance of a Money class with a defined currency and positive amount. The compiler says so!

Now there are a whole lot of incorrect programs you can no longer write. There are a whole lot of *correct* programs you can no longer write too. We all hope that there is still at least one correct program available, and that it is easier to search this space to find it.

But you can use types as much as you like and still not make life any easier for yourself. Let us start with a program that does not use types. It is a calculator for the nth Fibonacci number: put an index in byte 25,600 of your computer's memory, and it will write the appropriate entry from the Fibonacci sequence into byte 25,603.

This program is correct. I know that because I designed it to be so, and also I tested it both on paper and on a real computer. It is written in machine code for the Motorola 6809 CPU[3], but to aid readability I present it here in Motorola assembly language.

```
; push the direct page onto the stack
PSHS DP
; set a new direct page
LDA #64
TFR A, DP
; load sequence number from memory
LDA 0
; if A = 0 then the answer is 0
CMPA #0
BNE #6
STA 03
; pull the old direct page from the stack
PULS DP
; return
RTS
; jumping is easier when you know instructions have even addresses
NOP
; if A = 1 then the answer is 1
CMPA #1
BNE #6
STA 03
PULS DP
RTS
NOP
; initialise scratch variables: byte 1 = 0
LDB #0
STB 01
; byte 2 = 1
```

[3] https://en.wikipedia.org/wiki/Motorola_6809

```
LDB #1
STB 02
; loop starts here
; B = byte 2 + byte 1
; note: byte 2 already in B
ADDB 01
; byte 3 = B
STB 03
; copy byte 2 to byte 1
LDB 02
STB 01
; copy byte 3 to byte 2
LDB 03
STB 02
; decrement A
SUBA #1
; if A != 1 then jump back to loop
CMPA #0
BNE #4
; return: result is in byte 3 (and also 2)
PULS DP
RTS
NOP
BRA #-24
```

When I keep making assertions like "it is correct" and "it works", I of course mean within a limited realm. The calculation of the Fibonacci sequence values occurs in the B register of the 6809 which is 8 bits wide, so only the first 13 values of the sequence can be calculated correctly before overflow occurs. I could have used the D register for 16 bits of Fibonacci goodness, but then I would have had to store the counter outside the A register (maybe in memory pointed to by X) and would have needed four scratch bytes, not two. Software is all about compromises.

Anyway, let us imagine that our hero has learned that types make it easy to "reason about" code, and decides to make a typed version of the Fibonacci program. So they start to design appropriate types:

```
struct MC6809 {
    union {
        struct {
            uint8_t a;
            uint8_t b;
        } ab;
        uint16_t d;
    };
    union {
        uint8_t cc;
        struct {
            uint8_t e:1;
            uint8_t f:1;
            uint8_t h:1;
```

```
            uint8_t i:1;
            uint8_t n:1;
            uint8_t z:1;
            uint8_t v:1;
            uint8_t c:1;
        } flags;
    } cc;
    uint16_t x;
    uint16_t y;
    uint16_t u;
    uint16_t s;
    uint16_t pc;
};
```

Well, we know that the program works on a 6809 CPU, why not take advantage of that knowledge? We continue:

```
typedef uint8_t memory[65536];

struct computer {
    memory RAM;
    struct MC6809 CPU;
};
```

And write some functions that use these types correctly:

```
struct computer tick(struct computer startState) {
    uint8_t instruction = startState.RAM[startState.CPU.pc];
    return dispatch(instruction, startState);
}
```

This even looks like it is going to return a new value rather than mutating state, which we all know is an important part of writing a correct program!

Eventually the program is complete, and we can see the entry point:

```
int main(int argc, char **argv) {
    struct computer emptyMachine = {0};
    uint8_t fibonacci[] = {
        0x34, 0x08, 0x86, 0x64, 0x1f, 0x8b, // ...
    };
    struct computer loaded = loadProgram(emptyMachine, 0x6404, fibonacci);
    loaded.CPU.pc = 0x6404;
    loaded.RAM[0x6400] = atoi(argv[1]);
    struct computer finalState = runUntilHalt(loaded);
    printf("%d Fibonacci number is %d\n",
           finalState.RAM[0x6400],
           finalState.RAM[0x6403]);
}
```

This program correctly calculates the Fibonacci number; at least as correctly as the original one did. But the type annotations do not give that away. What the type annotations tell us is that we are emulating a Motorola 6809...hopefully that array of bytes we load into it is the correct program!

In fact given many of the fancy type systems on the market, the Fibonacci program has a pretty boring type. Here is one choice in the TypeScript language:

```
function fibonacci(n: bigint): bigint;
```

We know that we are going to get an integer out. Of course we knew that on the 6809 because there is no floating point unit!

A programmer with an expensive type system does not write better programs, any more than a novice guitarist becomes a stadium-grade shredder by buying a fancy guitar.

Cover photo by the author.

Apples And Oranges

Adrian Kosmaczewski

April 4th, 2022

Stanford Professor Jerry Cain spent the first 17 lessons of his 2007 Programming Paradigms lecture[1] (CS107[2]) explaining how to build a generic set of data manipulation functions using plain C, carefully showing how all those "bit patterns" are represented in memory. The resulting code, featuring a relatively large amount of casts to and from `void *` pointers, can sort and search arrays of integers, strings, floating point numbers, and pretty much anything that can be referenced with a pointer. Which is the same as to say, a lot.

The description of these capabilities is enlightening, and highly recommended to any and every professional software developer; please watch this series carefully. If anything, because the final conclusion of this first section should bring to mind a major revelation.

Types are a nice, and certainly a useful, thing to have, but they are not, by any means, a *conditio sine qua non* to build good software, whatever your definition of "good software" might be. In this article we are going to review the various ways in which type systems modify the experience of software developers while writing code; some for the best, some for the worst.

Definitions

Let us state the obvious: the "bit patterns" that Professor Cain describes in his lessons do not carry any type information whatsoever. Deep down in your computer, everything you read in this web browser session is nothing but a seemingly incoherent, yet semantically coherent, series of sequences of ones and zeros.

[1] https://www.youtube.com/playlist?list=PL9D558D49CA734A02
[2] https://see.stanford.edu/Course/CS107

Types in programming languages have followed a long evolution since the late 1950s, but have always had one primary and unique goal: to reduce programming errors through the manipulation of metadata indicating the range of bit patterns that should be considered valid during the run time of a program. As explained by Henk Barendregt[3]:

> Although the analogy is not perfect, the type assigned to a term may be compared to the dimension of a physical entity. These dimensions prevent us from wrong operations like adding 3 volts to 2 ampères.

(Barendregt, Henk. 1991. "Lambda Calculi with Types." In Handbook of Logic in Computer Science.)

As any tool, types can misused, and their introduction and functionality must be carefully analyzed, as explained by Don Syme himself[4], the creator of the (quite strongly typed) F# programming language:

> As an aside, something strange happens when one tries to have rational conversations about the above downsides with people strongly advocating expansion of type-level programming capabilities – it's almost like they don't believe the downsides are real (for example they might argue the downsides are "the choice of the programmer" or "something we should solve" – no, they are intrinsic). (...) This happens all the time once extensive type-level programming facilities are available in a language – they immediately, routinely get mis-applied in ways that makes code harder to understand, excludes beginners and fails to convince those from the outside.

Interesting. You mean that types are not the silver bullet that anyway does not exist[5]? It is refreshing and calming to learn that Mr. Syme has such a strong opinion on the subject, and such a protective instinct around his creation; that explains why programming in F# is considered by many (including this author) as a very pleasant experience.

Leslie Lamport[6], Turing Award winner and creator of LaTeX, together with computer scientist Lawrence Paulson[7], have exactly the same opinion:

> Types should be used if and only if they help more than they hinder.

Types can hinder the developer experience. What else?

> As programmers know, an unduly restrictive type system can make it hard to write perfectly reasonable expressions. Whitehead and Russell realized that a type discipline has to be flexible. The proof of a theorem like x □ {x } must not depend on the type of x. They invented (in 1910!) the concept we now call polymorphism, which they called typical ambiguity.

(Lamport, Leslie, and Lawrence C. Paulson. 1999. "Should Your Specification Language Be Typed." ACM Transactions on Programming Languages and Systems 21 (3): 502–26. https://doi.org/10.1145/319301.319317.)

Unfortunately, hearing zealots arrogantly[8] pushing for their preferred type system, one can see that pragmatism, approachability, usability, and readability are too often *not* primary

[3] https://en.wikipedia.org/wiki/Henk_Barendregt
[4] https://github.com/fsharp/fslang-suggestions/issues/243#issuecomment-916079347
[5] http://worrydream.com/refs/Brooks-NoSilverBullet.pdf
[6] https://en.wikipedia.org/wiki/Leslie_Lamport
[7] https://en.wikipedia.org/wiki/Lawrence_Paulson
[8] http://www.paulgraham.com/noop.html

design goals for some languages.

Many more problems are solved by strategically placing assertions[9] in code, than by using the latest fad in type systems; if anything, because assertions work at runtime instead of just at compile time.

Let us begin by the most visible, annoying effect of a type system: longer build times[10] during compilation; C++, Swift, and Rust[11] developers will surely agree with me in this point. On the other hand, languages like C#, TypeScript, and Go, show that it is indeed possible to provide a pragmatic yet strong approach to type systems, with extremely fast compilers generating optimized code in a very short amount of time.

(Interestingly, C# and TypeScript have the same person[12] behind, one with a long trail of successful developer experience delivery.)

The key buzzword to keep in mind while evaluating type systems is then, by all means, "developer experience."

Option Strict

Here is an interesting observation: in order to make programming more accessible to everyone, "scripting" and "hobbyist" languages not only allow developers to store any kind of value in a variable; in fact they do not even require them to declare said variables.

Just pick a symbol name, assign an integer to it, and voilà. If you ever need to reuse the same name for a string, be our guest! And thanks to implicit type conversions, if your string contains a number, maybe we will silently convert it to its numeric value, but we will not tell you, like, ever.

To put it bluntly, the Cain "bit patterns" of a string representing the number seven change from `00110111` to `00000111`. Not the same.

Well, yes, you will notice at runtime when you app crashes, and then Gary Bernhardt will invoke Watman to the rescue[13]. Hopefully your coding guidelines recommend Hungarian Notation[14] to help you name your variables.

The thing is, scripting languages tend to be popular in industry and academia, too. This is because they allow for very quick write-execute-debug cycles, usually involving a REPL of some kind; the famous "developer experience" shines in this universe. These languages tend to be excellent choices for prototyping, MVPs, small scripts, "glue" code, ERPs, nuclear plant controllers, and more.

Programs written with these languages become victims of their own success, and then humans become victims thereof.

Hence, following requests from said industry and academia, dynamically typed programming languages started to feature optional strict typing capabilities since the 1990s.

Regular readers of this magazine know that this author started his career with this mutant contraption called VBScript. This is how we made sure that our code had a decent level of

[9] https://deprogrammaticaipsum.com/assertions/
[10] https://hirrolot.github.io/posts/why-static-languages-suffer-from-complexity
[11] https://fasterthanli.me/articles/why-is-my-rust-build-so-slow
[12] https://en.wikipedia.org/wiki/Anders_Hejlsberg
[13] https://www.destroyallsoftware.com/talks/wat
[14] https://en.wikipedia.org/wiki/Hungarian_notation

readability.

```
Option Explicit
```

The creators of VB.NET, a language targeting an object oriented, garbage collected runtime, includes a similar statement in their language.

```
Option Strict On
```

Until a few years ago, Psalm[15] was your best bet to prevent type errors in your PHP code. The latest versions of the language have the ability to insert type declarations[16] wherever they make sense, as well as a specific directive to enforce stricter type checks.

```
<?php
declare(strict_types=1);
```

For those not using TypeScript yet, a proposal[17] has been recently made to add type hints to pure JavaScript code, instead of using JSDoc type comments[18]. In the meantime, the language has had a "strict mode" statement since ECMAScript 5, introduced in 2009.

```
"use strict"
```

And so does Perl, which still powers a lot of things on the Internet to this day.

```
use strict;
use warnings;
```

In a similar vein, Python[19] 3.10, released in October 2021, introduced type hints[20]. And the new Pyjion[21] JIT compiler uses type information to perform quite an array of optimizations[22].

If you need help with C's weak typing, the Cello[23] library enables type objects[24] for C11.

Do these "strict modes" help in the framework of dynamically or weakly typed languages? The experience shows that yes, as soon as your script grows beyond the fiery limit of 100 lines, and as soon as the staff count in a startup grows beyond its founder, having code that makes intentions explicit rather than implicit is a huge bonus.

Type Inference

This author has already mentioned, in a previous article[25], the fashion trends that govern the choice of type systems in languages since the 1970s. Dynamically-typed languages returned in vogue every 10 years or so, until type inference became the new big thing at the end of the 2000s.

[15] https://psalm.dev/
[16] https://www.php.net/manual/en/language.types.declarations.php
[17] https://github.com/giltayar/proposal-types-as-comments
[18] https://jsdoc.app/tags-type.html
[19] https://deprogrammaticaipsum.com/issue-35-python/
[20] https://docs.python.org/3/library/typing.html
[21] https://www.trypyjion.com/
[22] https://pyjion.readthedocs.io/en/latest/optimizations.html
[23] https://libcello.org/
[24] https://libcello.org/learn/a-fat-pointer-library
[25] https://deprogrammaticaipsum.com/the-truce-of-type-inference/

> Type inference is a must-have feature these days. Recent, popular, "modern" languages have it: Scala[26], F#[27], Go[28], Rust[29], Swift[30], TypeScript[31], Dart[32]. Older languages have been adapted to have it too: C++[33], Java[34], C#[35]. It is such an ubiquitous feature nowadays that Kotlin[36] does not even mentions it explicitly in its documentation.

Type inference systems bring the developer experience of scripting languages into the realm of compiled languages; of course, if your compile cycle is long (again, C++ and Rust, I am looking at you) these benefits might be a bit lost, but there is a substantial bonus anyway.

Where type inference shines, however, is when a language support generics; arguably, C++ templates have become more common and usable since C++11 included the `auto` keyword. No more trying to make the compiler happy!

```
external static auto auto(auto &ref) [&auto] {
    return reinterpret_cast<auto> (auto) auto;
}
```

And should you require something more akin to scripting language variables in your C++ code, there is always the Any[37] type to help you. Or, you know, just dynamic_cast[38] your way out of trouble.

The boundaries between programming languages get blurry as typing features cross from language to language following the latest trends.

Generics And Adverbs

Some language designers (like Don Syme above) agree to add new features to their type systems but only after a thoughtful process. As this article hits the database, the developers of Go have just released version 1.18, including a "much-anticipated"[39] feature, Generics[40]. On the other hand, there are lots of other type-related features that Go might never get[41], and that is actually a good thing.

But let us talk about generics; they are another major staple of "modern"[42] programming languages. At their most basic level, the idea behind Generics is quite easy to understand: I want this variable to hold an array of oranges, and that one to hold an array of apples, and not having any lemons[43] appearing anywhere. Very simple and very handy indeed. Visual

[26] https://docs.scala-lang.org/tour/type-inference.html
[27] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/type-inference
[28] https://tour.golang.org/basics/14
[29] https://doc.rust-lang.org/rust-by-example/types/inference.html
[30] https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID322
[31] https://www.typescriptlang.org/docs/handbook/type-inference.html
[32] https://dart.dev/guides/language/sound-dart#type-inference
[33] https://isocpp.org/wiki/faq/cpp11-language#auto
[34] https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html
[35] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/expressions#1163-type-inference
[36] https://kotlinlang.org/docs/basic-types.html
[37] https://github.com/kocienda/Any
[38] https://en.cppreference.com/w/cpp/language/dynamic_cast
[39] https://www.infoworld.com/article/3645228/go-118-arrives-with-much-anticipated-generics.html
[40] https://go.dev/doc/tutorial/generics
[41] https://www.infoworld.com/article/3650015/5-useful-features-you-wont-be-seeing-in-go.html
[42] https://deprogrammaticaipsum.com/the-great-rewriting-in-rust/
[43] https://deprogrammaticaipsum.com/the-various-meanings-of-quality/

Basic.NET even uses the `Of` keyword to literally express[44] that we want an array "of" oranges here, no apples allowed here thankyouverymuch.

This is in stark contrast to Cocoa's `NSArray`[45], for example, where you can happily mix and match any fruits you like.

```
NSArray *fruits = @[apple, @"orange", 42];
```

(This author misses Objective-C[46] and its non-obnoxious nature[47]. *Le sigh.*)

Very quickly languages extended the idea of generic containers, from arrays to hashtables, then to sets, structures, objects, and then why not functions? If a function works with stuff that is "countable", like apples and oranges, then we can pass a collection thereof and see how many items they have. But if we have an array of water, well, water not being countable, we will not get a proper answer; and the compiler will let us know about our foolishness.

Which begs the question, how does one define "countable"?

A common trick of popular programming languages is to be somewhat similar to spoken ones. Without needing to go as far as AppleScript goes[48], programming languages help us pretend that functions and methods are *verbs*; that data structures and objects are *nouns*; and that type information are *adjectives*. Following this reasoning, we could also find *adverbs* useful, that is, words that modify verbs, adjectives, and whole sentences.

It turns out that `interfaces`, which would be represented in C++ like `abstract classes` only holding pure virtual functions, work great as adverbs in the sense explained above, and that is the reason why many programming environments use adverbs as names of interfaces, protocols, or traits: .NET IDisposable[49], Ruby Observable[50], Java Runnable[51], UIKit MPPlayableContentDataSource[52], Swift Identifiable[53], POCO Nullable[54], PHP Stringable[55], etc.

And hence `ICountable` could be defined[56] as an interface with a single method returning the number of elements of a collection.

And then Swift developers followed the lead of their Objective-C godfathers, called their interfaces `protocols`, and became crazy about protocol-oriented programming[57]. They started mixing types, enums, interfaces, generics, in more or less savant proportions, and programs became theorems to be proven at compile time, so that everybody could start building abstractions on top of other abstractions instead of, you know, making programs that are actually useful, maintainable, readable, and understandable by others.

[44] https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/data-types/generic-types
[45] https://developer.apple.com/documentation/foundation/nsarray
[46] https://deprogrammaticaipsum.com/brad-cox/
[47] https://akos.ma/blog/the-developer-guide-to-migrate-across-galaxies/#4-objective-c-aka-coolia
[48] https://deprogrammaticaipsum.com/the-english-likeness-monster/
[49] https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=net-6.0
[50] https://docs.ruby-lang.org/en/3.1/Observable.html
[51] https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/Runnable.html
[52] https://developer.apple.com/documentation/mediaplayer/mpplayablecontentdatasource
[53] https://developer.apple.com/documentation/swift/identifiable
[54] https://docs.pocoproject.org/current/Poco.Nullable.html
[55] https://www.php.net/manual/en/class.stringable.php
[56] https://stackoverflow.com/a/59769261
[57] https://developer.apple.com/videos/play/wwdc2015/408/

Furthermore, we can start talking about the generics system in C++ and Rust, and realize that they are "monomorphic", contrary to Swift's "polymorphic" system, and write lots of blog posts and record mebibytes of podcasts about them. There is people who know a lot more about this so I will just refer the reader to them[58].

Here we are hoping Swift developers will read what Don Syme or Leslie Lamport think about types, and remember that programming was originally meant as a way to make computers solve problems for users, and *not* for writing mathematical theorems to be solved at compile time. In the meantime, we have ComparableComparator[59].

```
struct ComparableComparator<Compared> where Compared : Comparable
```

To be fair, this is not entirely the fault of "modern" languages. It would be unjust and wrong to forget the contribution of Andrei Alexandrescu[60]'s template metaprogramming in this lineage, as explained in the hallmark 2001 book "Modern C++ Design"[61]. Alexandrescu used C++ templates and its much dreaded multiple inheritance capabilities, to define "policy based design", an idea that has since been incorporated into the Boost libraries[62] and which can yield powerful, efficient, if arcane, constructions.

```
using HelloWorldEnglish = HelloWorld<OutputPolicyWriteToCout, LanguagePolicyEnglish>;
HelloWorldEnglish hello_world;
hello_world.Run();  // Prints "Hello, World!"
```

(Source: Wikipedia[63])

To reach the realm of generic code, C developers have remained faithful for more than 50 years to their casting to and from `void *` pointers, as shown by Professor Cain.

Data Exchange

Data representation languages can also benefit from types, and thus suffer a similar fate, to that of programming languages.

At one point we had XML Schemas[64], which were a standard, complete, useful way to validate XML documents before sending, receiving, or otherwise processing them. Senior .NET developers reading this article, who wrote web services around 2003, will surely remember the WSDL[65] language representation of the .NET types being exchanged, autogenerated from the C# code of the service beneath; a *de facto* direct ancestor of Swagger[66]. And just like with Swagger, generating a client out of a WSDL declaration was as easy as selecting a menu on Visual Studio .NET.

This was the current state of affairs in .NET 1.0, exactly 20 years ago. There has not been a lot of progress in this area since the beginning of the century, to be honest.

Arguably and understandably enough, XML was not really readable by humans–those of you who tried to debug an XSLT stylesheet know what I am talking about. Thus Douglas

[58] https://gankra.github.io/blah/swift-abi/
[59] https://developer.apple.com/documentation/foundation/comparablecomparator
[60] https://en.wikipedia.org/wiki/Andrei_Alexandrescu
[61] https://en.wikipedia.org/wiki/Modern_C%2B%2B_Design
[62] https://www.boost.org/community/generic_programming.html#policy
[63] https://en.wikipedia.org/wiki/Modern_C%2B%2B_Design#Simple_example
[64] https://en.wikipedia.org/wiki/XML_schema
[65] https://www.w3.org/TR/wsdl.html
[66] https://en.wikipedia.org/wiki/Swagger_(software)

Crockford[67] begat JSON, and now we need stuff like JSON Schema[68] to validate our data structures, and it is 2002 all over again, but this time with curly brackets instead of angle brackets. Big deal.

For data exchange purposes in large systems, Protobuf[69], MessagePack[70], Apache Thrift[71] and Apache Avro[72] are arguably better choices than JSON. They provide strong type definitions for the structures to be exchanged, generating tight binary representations of the payloads at run time. And all of this with strong support across a variety of programming languages.

It is the fervent opinion of this humble author, that teams finding themselves needing JSON Schema for their projects would be better served with one of the options enumerated in the paragraph above. But, of course, nothing beats JSON for its ease of use, so this author is not holding its breath.

Cloud and DevOps engineers do not have it any easier: let us hope they do lint their YAML[73], because those tabulations are really tricky to copy & paste from Stack Overflow. Languages used for "Infrastructure as Code" initiatives have lots of types, defining all of the things one can spend money for on a hyperscaler; just peek into any Terraform or Kubernetes manifest to convince yourself.

Dependent Types

Many indicators show Dependent Types[74] as the next big thing in type systems. At its core, a dependent type is a type whose definition depends on values, on actual data, defining a logic system with specific boundaries to be checked at compile time.

> Types matter. That's what they're for—to classify data with respect to criteria which matter: how they should be stored in memory, whether they can be safely passed as inputs to a given operation, even who is allowed to see them. Dependent types are types expressed in terms of data, explicitly relating their inhabitants to that data. As such, they enable you to express more of what matters about data.

(Altenkirch, Thorsten, Conor McBride, and James McKinna. 2005. "Why Dependent Types Matter.")

The simplest example of a dependent type could be a primitive Rust array of a fixed size[75].

```
let mut array: [i32; 3] = [0; 3];
```

Another common example are algebraic data types (ADTs) like those found in TypeScript.

```
const x : 4 | 5 | 6 = 5;
```

In F# and many other functional languages, ADTs are powerful constructions also referred to as "Discriminated Unions", which, when used with criteria and taste, can greatly increase

[67] https://deprogrammaticaipsum.com/douglas-crockford/
[68] https://json-schema.org/
[69] https://en.wikipedia.org/wiki/Protocol_Buffers
[70] https://en.wikipedia.org/wiki/MessagePack
[71] https://en.wikipedia.org/wiki/Apache_Thrift
[72] https://en.wikipedia.org/wiki/Apache_Avro
[73] https://www.redhat.com/sysadmin/check-yaml-yamllint
[74] https://en.wikipedia.org/wiki/Dependent_type
[75] https://doc.rust-lang.org/std/primitive.array.html

the readability of the code.

```
type MeasurementUnit = Cm | Inch | Mile
```

The theory of dependent types is beyond the scope of this article. Their underpinnings are related to a mathematical concept called Intuitionistic Type Theory[76], described by the Swedish mathematician Per Martin-Löf[77]. A disciple of Martin-Löf, Johan Georg Granström, wrote a complete book about this theory, conveniently called "Treatise on Intuitionistic Type Theory"[78], published by Springer. Another book about dependent types is "The Little Typer"[79] by Daniel P. Friedman and David Thrane Christiansen, published by MIT Press.

For a deeper understanding of type systems, Benjamin C. Pierce[80] has written the most definitive bibliography[81] on the subject, also published by MIT Press.

Conclusion

Screaming from the top of your lungs that a monad is simply a monoid of the category of endofunctors[82] does not help anyone build better software. On the other hand, a good developer experience does help good developers write good code.

As I write these lines, a colleague of mine sighed in the company chat about an error spat by his compiler. The decryption of this error is left to the reader as an exercise; suffice to say that Scala's type system was involved.

```
type mismatch;
 found   : com.fasterxml.jackson.databind.node.ObjectNode
 required: ?{def map(x$1: ? >: <error> => play.twirl.api.HtmlFormat.Appendable): ?}
   (which expands to)  ?{def map(x$1: ? >: <error> => play.twirl.api.Html): ?}
Note that implicit conversions are not applicable because they are ambiguous:
 both method twirlJavaCollectionToScala in object TwirlHelperImports of type [T](x: Iterable[T]): Iterabl
 and method iterable AsScalaIterable in trait ToScalaImplicits of type [A](i: Iterable[A]): Iterable[A]
 are possible conversion functions from com.fasterxml.jackson.databind.node.ObjectNode to ?{def map(x$1:
```

Imagine being greeted by such a stack trace on a Monday morning.

Language and programming tool makers can (and arguably must) help developers in building quality software with a reasonably good developer experience. This is very different than becoming strong typing zealots (or indulging in any other similar petty flame war). Only the former actually matters, while the latter is a mere distraction.

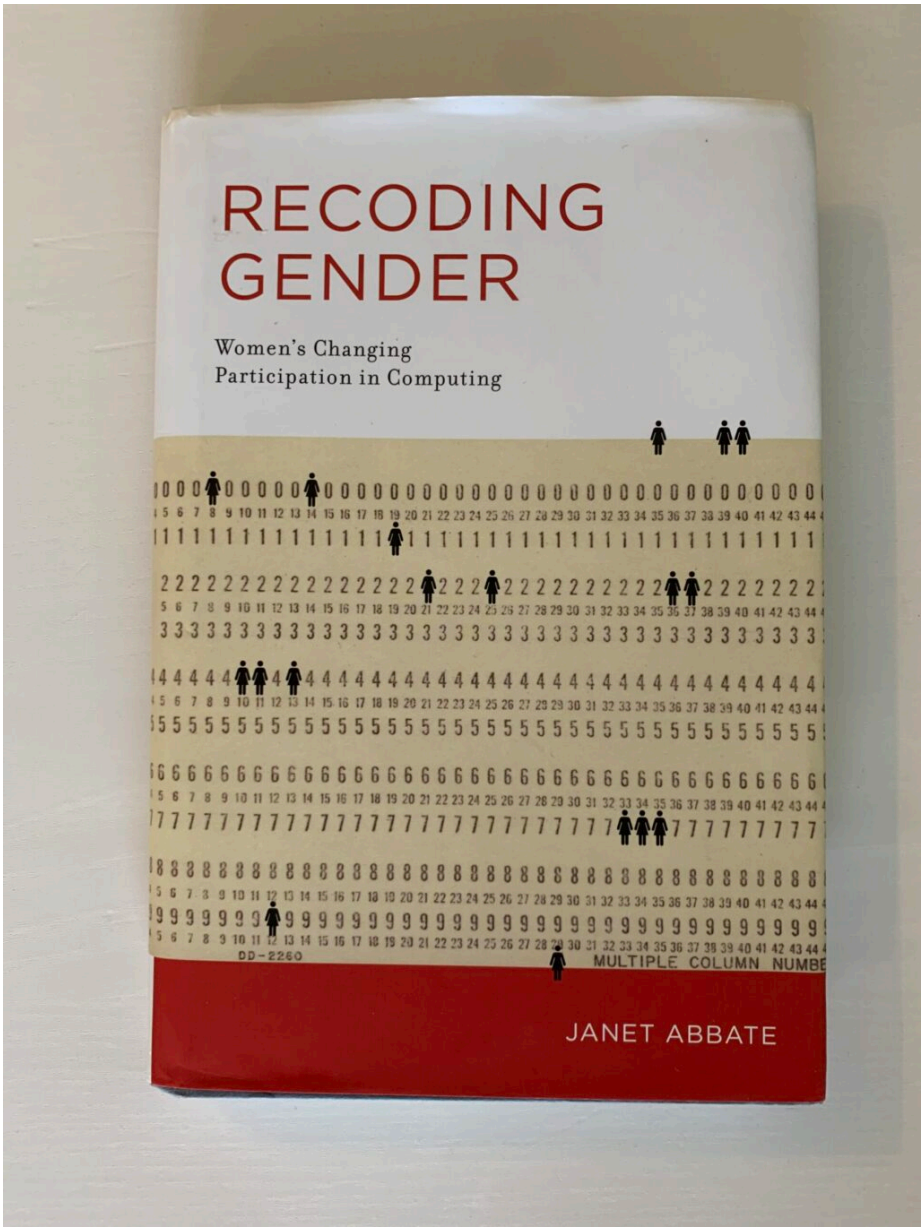
Cover photo by Tom Grünbauer[83] on Unsplash[84].

[76] https://en.wikipedia.org/wiki/Intuitionistic_type_theory
[77] https://en.wikipedia.org/wiki/Per_Martin-L%C3%B6f
[78] https://link.springer.com/book/10.1007/978-94-007-1736-7
[79] https://mitpress.mit.edu/books/little-typer
[80] https://en.wikipedia.org/wiki/Benjamin_C._Pierce
[81] https://mitpress.mit.edu/contributors/benjamin-c-pierce
[82] https://stackoverflow.com/a/3870310
[83] https://unsplash.com/@tomgruenbauer?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[84] https://unsplash.com/s/photos/apples-oranges?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Janet Abbate

Graham Lee

April 4th, 2022



We are told that history repeats itself[1], first as tragedy, then as farce. This is certainly true of the history of computing, at least as far as its telling is concerned.

The tragedy of the history of computing was focusing on the parts where men were the heroes. In the 1940s, computers were built by such giants as Alan Turing (even this is not

[1] https://deprogrammaticaipsum.com/history-repeating/

true: the giants of Colossus were Bill Tutte, Max Newman, and Tommy Flowers), Konrad Zuse, J. Presper Eckert, John Mauchly, and Jon von Neumann. Mysteriously, nobody seemed to do any programming for them until decades later, when more great men came along: Backus, Dijkstra, Knuth, Wirth, Naur, and of course the even greater men like Fred Brooks who managed them to their successes.

The farce came when addressing the gender discrimination of computing was attempted not by making any repairs to the education system, to the workplace, to the culture of gendered roles of work, but by peppering the names of the great *women* of computing into this history. Take the same story, add women, and stir. Male speakers would rattle off lists of names of female contributors to their male audiences to show how far we had come: I know who Ada Lovelace was! And Grace Hopper! Kateryna Yuschenko! Katherine Johnson! Karen Spärck Jones! I know about as many names of women computer scientists as men whose first name is Steve, I am an ally!

Those who read the work from which that "history repeats itself" quote is paraphrased, Karl Marx's "XVIII Brumaire of Louis Napoleon", will know that it also contains the following indictment of the "great man" view of history. Our Turings, Dijkstras, even our Hoppers and Lovelaces are not actors independent of history who saw some prophetic vision of how the world can be. They are normal members of the societies they lived in, who swept along in and contributed to the currents of history flowing around them.

> Men make their own history, but they do not make it as they please; they do not make it under self-selected circumstances, but under circumstances existing already, given and transmitted from the past. The tradition of all dead generations weighs like a nightmare on the brains of the living.

Thus the true history of computing is not that there were no computers until Turing invented computers, and there were no worthwhile computers until Jobs invented the polyvinyl carbonate box with a keyboard on top. It is the history of the people who were making circuits with the thermionic valves already in use as radio amplifiers. It is the history of the people who plugged cables into breadboards to change the function of the computers. It is the history of the people who mail-ordered a Science of Cambridge MK14 home and played with it—or maybe never got around to playing with it.

That someone created JavaScript is immaterial. That millions of people decided to get jobs that other people decided to call "front-end software engineers" and make things that billions of people use to order pizza, rig elections, and send cat pictures to their nieces: now *that* is history in the making!

Janet Abbate uncovers that story in *Recoding Gender: Women's Changing Participation in Computing*.** This is the history of the people who were involved in computing, and how their preconceptions of what different genders were capable of advanced or retarded their contribution to this developing field. Starting with the second world war, when all programmers were women because the interesting stuff was the maths and engineering, moving through the twentieth century when the computer boys take over[2] as they notice that there is interesting maths and engineering in that there programming stuff that they would like to take the credit for themselves.

The peak year for women studying computer science was 1984 (37% of CS grads were women). At around the same time, women were pioneering in the field of remote-working contract programmers, and in some cases (notably Steve Shirley) being found to violate

[2] https://thecomputerboys.com

workplace equal opportunities laws by creating jobs that suited the gendered expectations of women as homemakers and full-time parent.

Since women pioneered in the field of programming (Betty Snyder invented the first program that wrote another program, giving Grace Hopper the idea for the compiler), managers have had the idea that the work of programming the computer must be a simple rote exercise that can be reduced to mechanistic work, even automated away by another computer. Taking the history of computing as a whole, Github Copilot is the latest embarrassing failure[3] in nearly a century of "automatic programming" development: an embarrassment that tries to hide the huge amounts of manual effort (from workers of all genders) that has gone into creating the so-called automatic program. Every time the managers of the industry pointed at the computer as an electronic brain that was programming itself, the curtain would be swept aside to reveal the programmers who wrote the program that wrote the program.

This has been a needfully brief tour of the ideas in a very short, very important book. *Recoding Gender* is part of the MIT Press History of Computing[4] series, along with Programmed Inequality: How Britain Discarded Women Technologists and Lost Its Edge in Computing[5] by Mar Hicks. To understand why we are here today, read these books and learn how we got here.

Cover photo by the author.

[3] https://deprogrammaticaipsum.com/innovationscript/
[4] https://mitpress.mit.edu/books/series/history-computing
[5] https://deprogrammaticaipsum.com/mar-hicks/