

Greg Chaitin, Computer Programmer

Rocky Bernstein

As a friend of Greg Chaitin, and an avid follower and reader of his writings, one thing that strikes me is that there is very little mentioned about the 25 years or so that he spent at IBM, most of it at IBM Research. Perhaps the situation is analogous to that of Archimedes[Wiki, 2020]. His mathematical works are well preserved, but we don't really know the extent of his considerable practical engineering and invention.

Greg started out as a IBM Service Engineer in Argentina and I came to know him when he worked at IBM Research.

We worked on a compiler[Auslander and Hopkins, 1982a][Auslander and Hopkins, 1982b][Wiki, 2020] for IBM's first RISC, then called 801[Wiki, 2020a]. It later became known as the POWER architecture[Wiki, 2020b].

The work he did on that project was first rate and I learned a lot from him about writing code. Some of this is described below. In addition, he used insightful algorithms. You can find a little bit about his work on the register allocator in a couple of papers[Chaitin et al., 1981][Chaitin, 1982]. These papers have become the standard references whenever something describes register allocation by graph coloring, which is called Chaitin-style graph coloring.

The papers convey the elegance of the idea but, having studied the working code as it appeared inside the compiler, I don't think they do justice to the underlying engineering or volume of output Greg produced when I had the pleasure of working with him on that project.

It is surprising there is not more about this period in biographical or autobiographical writings, and I would like to address that here.

A little background.

I started working at IBM Research right after getting a masters' degree. Although I had worked as a programmer and written code in school and on part-time jobs, this was my first full-time job at any large company. The project I worked on was an optimizing compiler. The code was hundreds of thousands of lines long, written by maybe a dozen people over several years. This kind of scale is still true for many compilers and interpreters. The "optimizing" part of "optimizing compiler" means that clever techniques are applied to make the code run fast.

Needless to say in any such endeavor of this size there is bound to be a lot of chaos. Here is an example that I encountered in watching a bug get fixed.

Before Greg's involvement, after the input source program was parsed, information was mostly managed in expression trees. The structure for an expression node could have a number of fields.

The compiler also consisted of a number of "phases," as found in many compilers. In particular, the phases were:

1. A parse phase
2. A code-improvement phase
3. A register-allocation phase
4. A final assembly to machine-code phase

One day, a bug turned up because a field that was needed in the final assembly phase got mangled. That field was called "ppb" for "phil's private bits." Phil worked on the parse phase.

Peter, who worked on the code-improvement phase, occasionally needed to store a bit of information for his purposes. He figured that since his phase came after Phil's phase, he could do a little code optimization of his own and reuse phil's private bits since, as he put it, Phil wasn't looking. That way he wouldn't have to add another temporary field which would make the expression-tree structure larger and more unwieldy. So "phil's private bits" became "peter's private bits" when Phil wasn't looking.

Unfortunately Hank, who worked on the final assembly phase, needed the information stored in phil's private bits. So, phil's private bits weren't as private as Phil or Peter thought. This is the kind of chaos that was common for a code base of this size.

Greg's phase, affectionately known as Phase III, was the register-allocation phase. When I studied Greg's code in order to take over the phase, I was amazed at how clean, efficient and elegant it was. Not at all like phil's/peter's private bits. At one point, I thought I could come

up with an improvement, only to find that, even though it hadn't been mentioned in the papers describing the code, it was already implemented.

The first thing that struck me about Greg's code was that it followed the kind of formalism found in physics equations. Physics often uses short names in a particular domain. For example E , m and c in $E = mc^2$; this code was like that. There was a short and consistent notation for program variables that represented all registers, another for a single register, an idiom for looping over a register, and so on. This was very different from the rest of the code base.

In contrast to the looseness described in the Phil/Peter bug, there was strict consistency, clarity, succinctness, and focus on purpose which was applicable generally. This was the first time I had encountered code that exuded a sense of art rather than the typical robotic engineering that I was familiar with and pervaded the rest of the code.

As I said, the *algorithm* behind the register allocation can be found in a couple of papers. However what you won't find printed anywhere is a description of the excellent engineering that made this work. So I'd like to describe this next.

To support working on the register-coloring algorithm, Greg invented a register-transfer language and its display format. It has a couple of features not found in many register-transfer languages, such as a way to indicate that an instruction has more than one output. For example the IBM 360/370 multiplication instructions would set a *pair* of registers when the operation completed. So in the register-transfer language, both of these registers were on the left-hand side of an assignment statement. If an instruction, like 'add', set a condition bit such as 'carry', that bit was indicated as well. When a register went "dead" [Wiki, 2020c]—the last time a value from that register was used—it had a tick mark after it.

All of this was needed in the register-allocation process, and the concise and precise language Greg invented for this purpose made it very easy to understand what was going on and how registers got allocated or "colored."

Nowadays, it is not uncommon in compilers to transform the tree representation that comes out of a parser (Phase I) into a register-based representation. But circa 1982 it was not a widespread practice. Greg's work predates the invention of Single Static Assignment, or SSA, which followed shortly afterward at IBM, presumably under Greg's influence. Both promoted the proliferation of register-based intermediate languages.

Greg's work was contemporaneous with the first edition of the Dragon Book [Aho, 1977] which popularized this idea.¹ Compare this with the earlier books by David Gries [Gries, 1977] or

¹The first edition described a register allocation scheme that was suited for stack-architecture CPUs. Later editions of this book switched to describing the register allocation scheme that Greg developed.

McKeeman et al.[McKeeman et al., 1970]. In the 1970s, there were several compilers that worked off of a tree representation. Some interpreters, such as those for Lisp, Perl5, or Korn shell, still work off of expression trees.

When Greg joined the project, the final assembly phase, Phase IV, used the expression trees built in Phase I, so there was quite a bit of engineering and coding that Greg had to do to convert the expression trees into his register-transfer language. And since the assembly phase still used those trees, when I started there was work to convert the results *back* into expression trees after coloring so that Phase IV could do its work. That idea that you could wall yourself off from the surrounding chaos blew my mind, but by doing so Greg had created a little corner of the world with order in the middle of a big mess.

Eventually, Greg convinced Hank, the person who worked on the phase after Greg's phase, to use his more elegant register-transfer language to drive final assembly. Hank told me that in doing so, his code got a lot better, simpler, shorter, and easier to maintain.

I have used the technique of coming up with an elegant or more appropriate representation and language in which to think and describe things in, even if it means writing additional and messy engineering code to do transformations both into and out of your language. Each time this comes up there is great benefit. I am grateful to Greg for adding this technique to my programming arsenal. In algorithms you sometimes find this idea in conjunction with, say, the Discrete Fast Fourier Transform. However it is not something usually mentioned in conjunction with programming.

This kind of elegance and simplification was not an isolated incident, but part of the fiber of Greg's programming. Here is another example.

After this work was completed, Greg embarked on a totally different aspect of the compiler, the binder, sometimes called a linker. This is a piece of code that is needed after the source code is compiled into machine code and just before running or loading the code. It adjusts the code so that it can work in an arbitrary section of computer virtual memory.

To do this, the compiler needed to be modified so that it used only those instructions that were position-independent. Nowadays this is called PIC (Position-Independent Code)[Wiki, 2020]. In other words, certain instructions or instruction formats had to be avoided. These typically include the "absolute address" forms of the "load from memory" and "store from memory" instructions.

The first versions of Greg's binder started out by more or less following what some other clever IBM Researchers, notably Chris Stephenson, did to perform this process on conventional IBM System/370 hardware. In the beginning Greg probably knew no more about this process than

any casual programmer would. He finished a preliminary version of the code which worked and reflected his understanding of Chris's code for the 370. It was probably more or less a translation of that code adapted to the new hardware instructions and architecture.

However after this was completed and working, he had an epiphany—the process of binding together a set of machine code fragments could be thought of as a data-dependency driven process which uses a topological sort to pull everything together followed by a garbage-collection process that removes those parts of the code fragments that are bundled in “object decks” but aren't needed for the particular program that is being bound together.

At this point he discarded his thousands of lines of code and rewrote everything from scratch in a couple of days! I had never heard of anyone doing such a thing.

The end result was, of course a very concise and clean piece of code. Getting this code adapted into IBM's product engineering went pretty quickly. Most of the things that came out of IBM Research needed significant rewriting before they would be considered by the Product team, but this was not the case here due to its usefulness, simplicity and elegance. It became known as IBM's TOC binder[IBM, 2020].

One other incident in conjunction with the TOC binder attests to its unity of purpose and simplicity.

One day I happened to be in Greg's office. Usually, everyone worked on a bullpen that had 6–8 terminals in it. (Unix was developed in such an environment, too.) Before the days of IRC and Slack, a bullpen like this had the same effect. If you had a problem, you'd just blurt it out. Someone might hear you and quickly tell you what was going on.

However we also had private offices. Greg's office, like his code, was free from clutter. It consisted of a single terminal on a desk otherwise empty except for a phone. A single framed picture hung on the wall. Nothing more.

I had dropped by there and Dick, who was working on a database for our operating system, came in to describe a problem he had encountered. Just as Sherlock Holmes would do, Greg faced Dick the entire time, listening intently and quietly. After that, there was a pause of about 30 seconds as he reflected. Then after asking a few questions, he offered a theory of what might be wrong. With a couple of clicks on the terminal he brought up a file, just one small part of the rather substantial code. In a couple of clicks more, he found the problem spot. The whole process took less than 5 minutes.

It was amazing to me that he figured out the bug completely without having run the code even once. In order to verify that his theory and fix were correct, the code had to be run, but

that came later. This kind of bug fixing is something I have rarely encountered. The fact that something like this can be done at all I attribute to the simpleness, cleanness and overall good organization of the code. He could reason about its behavior without having to run it.

My last example of clean concise code is published.

Greg wrote computer simulations for five well-known physical models[Chaitin, 1985]:

- a satellite going around the Earth according to Newton,
- propagation of an electromagnetic wave according to Maxwell,
- the same satellite going around the Earth according to Einstein,
- an electron moving in a one-dimensional potential according to Schrödinger, and
- sums over all histories according to Feynman.

Each of these simulations fits on a single page of APL2 code. This includes setting up the data and drawing plots of the output. In fact this code was so short and elegant that for a while several of the 30-line simulations replaced the picture that had been hanging in his office.

References

- [Aho, 1977] Aho, Alfred V. and Ullman, J. (1977). *Principles of Compiler Design*. Addison-Wesley, New York.
- [Auslander and Hopkins, 1982a] Auslander, M. and Hopkins, M. (1982a). An overview of the PL.8 compiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, page 22–31, New York, NY, USA. Association for Computing Machinery.
- [Auslander and Hopkins, 1982b] Auslander, M. and Hopkins, M. (1982b). An overview of the PL.8 compiler.
http://rsim.cs.uiuc.edu/arch/qual_papers/compilers/auslander82.pdf. On-line pdf.
- [Chaitin, 1982] Chaitin, G. J. (1982). Register allocation and spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, page 98–105, New York, NY, USA. Association for Computing Machinery.

- [Chaitin, 1985] Chaitin, G. J. (1985). A computer gallery of mathematical physics—a course outline.
<http://www.softwarepreservation.org/projects/apl/Papers/ComputerGallery>.
- [Chaitin et al., 1981] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. (1981). Register allocation via coloring. *Computer Languages*, 6(1):47 – 57.
- [Gries, 1977] Gries, D. (1977). *Compiler construction for digital computers*. Wiley, New York.
- [IBM, 2020] IBM (2020). Overview TOC AIX.
<https://www.ibm.com/developerworks/rational/library/overview-toc-aix/index.html>. Accessed 2020-11-13.
- [Mckeeman et al., 1970] Mckeeman, W. M., Horning, J. J., and Wortman, D. B. (1970). *A Compiler Generator*. Prentice-Hall.
- [Wiki, 2020] Wiki (2020). Archimedes. <https://en.wikipedia.org/wiki/Archimedes>. Accessed 2020-11-13.
- [Wiki, 2020a] Wiki (2020a). IBM 801. https://en.wikipedia.org/wiki/IBM_801. Accessed 2020-11-13.
- [Wiki, 2020b] Wiki (2020b). IBM POWER microprocessors.
https://en.wikipedia.org/wiki/IBM_POWER_microprocessors. Accessed 2020-11-13.
- [Wiki, 2020c] Wiki (2020c). Live variable analysis.
https://en.wikipedia.org/wiki/Live_variable_analysis. Accessed 2020-11-13.
- [Wiki, 2020] Wiki (2020). PL/8. <https://en.wikipedia.org/wiki/PL/8>. Accessed 2020-11-13.
- [Wiki, 2020] Wiki (2020). Position-independent code.
https://en.wikipedia.org/wiki/Position-independent_code. Accessed 2020-11-13.