Complementary science of interactive programming systems

Is it worth looking at the history of programming, not just for the sake of history, but in order to discover lost ideas that could be utilized by present-day computer scientists to advance the state of the art of programming? We answer the question in the affirmative, propose a methodology for doing so and present an early experiment that recovers a number of interesting programming ideas from, against all odds, Commodore 64 BASIC.

Programming systems and languages

The history of programming may seem too short to comprise multiple incommensurable scientific paradigms, but there have certainly been interesting shifts over time. I focus on the shift from *programming systems* to *programming languages* discussed by Gabriel (2012) and hinted by Priestley (2011). Work on programming systems considers interactive environments like Smalltalk or Interlisp. It focuses on what is happening in the system; a study of programming system considers source code, state of the running system as well as user interactions. Now dominant work on programming languages focuses on meaning of a program and considers solely the code. This shift enabled much contemporary programming language research and led to the development of useful program analysis tools, including e.g., type systems. But what ideas have been victim to Kuhn loss associated with the paradigm shift? What programming ideas are lost if we only think in terms of programming languages?

Past work on programming systems explored, for example, ways of modifying programs while running (Bobrow et al., 1988) and interesting ways of constructing programs by interacting with the programming environment (Smith, 1977). Such research cannot be easily expressed in terms of the *programming language* research paradigm and many of the systems that implemented it are no longer available in an executable format. So, how can we recover such ideas and make them available to present-day computer scientists?

Complementary science

To be effective, specialist science needs to accept certain assumptions about what it studies and computer science is no difference. However, this means that there are interesting questions outside of its core focus. The methodology of *complementary science* proposed by Chang (2012) aims to study such questions. It proposes looking at the history of science, and use the historical perspective to recover forgotten knowledge, use it for critical assessment of contemporary scientific research and attempt to further develop such forgotten ideas. As Chang suggests, such extension of past research is worthwhile in cases where scientists abandoned (some aspects of) a scientific research paradigm too early.

We argue that the move from programming systems to programming languages is one such case where computer scientists abandoned a research paradigm too early. In the case of programming, the methodology may be even more valuable. First, programming research does not aim for truth, but aims for adequacy (Cross, 2006). As such, it may abandon ideas not because of any experimental failure, but for more social reasons. Second, ideas on programming also do not come solely from the world of (computer) science as many innovative systems are built by practitioners and “hackers”. Complementary science provides a method that can reintegrate such ideas into the realm of academic research.

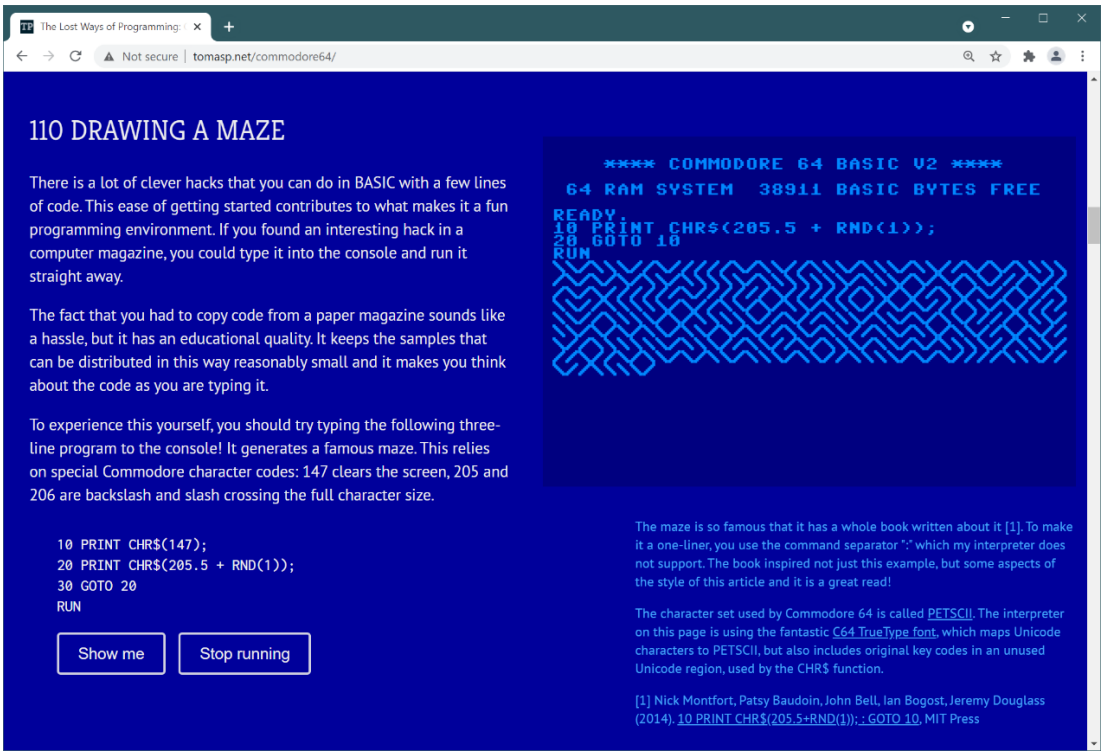

Figure 1. Reconstruction of a programming environment for Commodore64 BASIC, running a well-known one-liner for generating a maze (Montfort et al, 2012). Available at http://tomasp.net/commodore64

Case study: Reconstructing Commodore64 BASIC

In our initial work on studying programming systems using the method of complementary science, we explore the interactive programming environment of Commodore64 BASIC (Figure 1). This is a perfect example of a programming system outside of the mainstream computer science – the BASIC language "mutilates the mind beyond recovery" (Dijkstra, 1984) and the Commodore64 programming environment seem like a toy for hobbyists. Yet, the programming environment has a number of interesting characteristics. For example:

- *Interaction uniformity*. The system uses a single interaction for both editing programs and running programs. If a line starts with a number, it is inserted into the program, otherwise it is executed directly.
- *Simple & flexible workspace.* The use of line numbers makes it easy to write and test parts of a program independently, as well as to debug programs (without a dedicated debugger) by setting variable values and jumping to a desired line.

Studying programming systems

Interactive programming systems, both past and present, present an interesting challenge for their rigorous study. In particular, many of their interesting features are only apparent when following a realistic interaction with the system. This is difficult to reconstruct for systems that are no longer available or rely on specialized hardware. Edwards et al. (2019) review a number of possible methods that have been used for presentations of interactive programming systems in recent years, including video recordings and interactive web-based essays. We believe those approaches can be fruitfully combined with the complementary science methodology.

In particular, the Commodore64 BASIC reconstruction (Figure 1), developed by the first author, is an interactive web-based essay that combines a tutorial walkthrough that illustrates the development of a simple game with an interactive console where the game can be run. The console only supports a limited subset of the system, but is representative enough to illustrate the interesting aspects of the programming environment. The reader can type and run code on their own, but also use the “Show me” button to replay the interaction.

Conclusions

We believe that the history of programming contains numerous interesting paths not taken that are worth recovering and that could provide inspiration for contemporary research. Doing so is, however, not simply a matter of reading old academic papers. In order to gain a fundamental understanding, we need a historical perspective that understands systems through the perspective of the paradigm from which they originate. The case of *programming systems* vs. *programming languages* illustrates this point well. If we look merely at the language used by a programming system, we can easily miss the most important characteristic of the system, that is how the programmer interacts with it. To realize this, we need a historically-informed reading of the system. Finally, reconstructing ideas about interactive programming systems and making them accessible to contemporary computer scientist (a goal we adopt from *complementary science*) also requires the use of suitable media that allow interaction – such as interactive web-based essays.

References

Gabriel, R.P. (2012). *The structure of a programming language revolution*. In Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software (pp. 195-214).

Priestley, M. (2011). *A science of operations: machines, logic and the invention of programming*. Springer Science & Business Media.

Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G. and Moon, D.A. (1988). *Common lisp object system specification*. ACM Sigplan Notices, 23(SI), pp.1-142.

Smith, D.C. (1977). *A computer program to model and stimulate creative thought*. Basel: Birkhauser.

Chang, H. (2012). *Is water H_2O? Evidence, realism and pluralism* (Vol. 293). Springer Science & Business Media.

Cross, N. (2006). *Designerly Ways of Knowing*. ISBN 978-1-84628-300-0. Springer.

Montfort, N., Baudoin, P., Bell, J., Douglass, J. and Bogost, I. (2014). 10 PRINT CHR$(205.5 + RND(1));: GOTO 10. MIT Press

Dijkstra, E. (1984). *The threats to computing science* (EWD898). Delivered at ACM 1984 South Central Regional Conference, November 16–18, Austin, Texas.

Edwards, J., Kell, S., Petricek, T. and Church, L. (2019). *Evaluating programming systems design*. In proceedings of PPIG 2019.