Dead Programming

Michael Coblenz
Carnegie Mellon University
Pittsburgh, Pennsylvania
mcoblenz@cs.cmu.edu

ABSTRACT

Live programming has recently become a popular topic of research. This paper introduces dead programming, a promising new direction in programming language design.

KEYWORDS

Programming languages, Live programming, Dead programming

ACM Reference Format:
Michael Coblenz. 2018. Dead Programming. In *Proceedings of ACH SIGBOVIK (SIGBOVIK '18).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Live programming [1] espouses a responsive model in which the programming environment provides continuous feedback to the programmer regarding program behavior. Unfortunately, this approach to programming suffers from several flaws, which we address in our new design. First, in spite of a programmer's best effort, programs in live programming languages can include bugs; fixing these bugs in a live program is a hazardous occupation [2]. Second, as with traditional programming languages, programs written in live programs can have poor performance; consider the fact that live programs run on limited hardware and are written with traditional features such as recursion and loops. Third, live programs can be very expensive to construct, since they require expensive programmers to write them.

Dead programming addresses all of the above problems with a novel programming model. In fact, dead programming consists of a whole suite of tools, which enable superior programmer productivity. In this paper, we describe the first programming language, DEAD, designed to support dead programming, along with a compiler and optimizer. Although DEAD programs can be executed on traditional hardware, we also show how novel hardware can optimize execution beyond that which currently-available production equipment can attain.

2 PROGRAMMING LANGUAGE

In this section, we describe the programming language, DEAD, which we have successfully used to write dead software.

2.1 Syntax

P ::= .
  | P NOP

V ::= ()

SIGBOVIK '18, April 1, 2018, Pittsburgh, PA USA
2016. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.2 Static Semantics

DEAD only supports one value, `()`, to which every program evaluates. Every program has type `unit`.

$$\frac{}{\vdash \cdot : \texttt{unit}} \qquad \frac{\vdash P : \texttt{unit}}{\vdash P\ \texttt{NOP} : \texttt{unit}}$$

2.3 Dynamic semantics

Every DEAD program evaluates to the value (). The proof is left to the reader as a simple exercise in induction on the dynamic semantics.

$$\frac{}{\vdash \cdot \Downarrow ()} \qquad \frac{\vdash P \Downarrow v}{\vdash P\ \texttt{NOP} \Downarrow ()}$$

The astute reader may notice that programs written in DEAD have no side effects, since the semantics require no operations on a store. DEAD is a pure functional language, as are most new, trendy languages. As is widely understood, mutable state is a primary cause of difficulty reasoning about program behavior. DEAD addresses this problem in a similar fashion to Haskell, another widely-used pure functional language. We expect that by facilitating ease of understanding and being substantially simpler than Haskell, DEAD will quickly become a very popular language, as it is a particularly convenient way of expressing programs that do nothing.

Some readers may be troubled by the lack of expressivity of a program that has no side effects. We remind the reader that, in the end, the output of every computer is a matter of converting electricity to heat, an operation that can be conducted quite effectively by a DEAD program.

2.4 Runtime environment

Conveniently, most modern CPUs already implement DEAD directly because they support a NOP instruction. Thus, the implementation of a runtime environment is quite straightforward and requires no overhead, unlike most existing approaches to programming. The use of a compiler is quite unnecessary. However, many programmers will avail themselves of an optimizer.

2.5 Compiler optimization

We show in this section how the traditional liveness analysis can be applied to DEAD programs. Note that no DEAD instruction reads from any register or memory address, which means that none of the temps written to are live. Thus, a correctly-implemented optimizer can remove every instruction from a DEAD program, leaving an optimal (zero-length) program.

As there are no facilities to invoke external library functions, dead code removal can safely remove all existing library code.

The result is that, although DEAD is an expressive language, supporting programs of arbitrary size, every DEAD program optimizes optimally down to a zero-length program. Thus, the programmer can choose exactly how much electricity to convert to heat when running the program for a given execution environment, thus exactly replicating the behavior of a program written in a traditional programming language.

2.6 Hardware support

Current hardware environments offer efficient conversion of electricity to heat. However, in many cases this side effect is undesirable. We propose, then, a new programming environment, which is better-suited to running DEAD programs. We have successfully executed DEAD programs on a Russet potato. In addition to being an aesthetically pleasing addition to any office environment, the POTATO machine supports local farmers. It does have the disadvantage of eventually sprouting non-dead appendages; this is an opportunity for future work.

3 THE FUTURE: UNDEAD PROGRAMMING

Undead programming represents the next frontier in programming environments. Although one might argue that the POTATO execution environment is already undead, future work should more thoroughly explore the space of undead languages and environments.

REFERENCES

[1] Sean McDirmid. 2007. Living It Up with a Live Programming Language. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 623–638. https://doi.org/10.1145/1297027.1297073

[2] Aplana Software. 2015. When developers try to fix a bug in production. https://www.youtube.com/watch?v=75wa8Lx4yc4. (2015).