You can program what you want but you cannot compute what you want

Alireza S. Abyaneh and Christoph M. Kirsch

Department of Computer Sciences
University of Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract. Computers are the most fascinating machines ever invented. Virtually everyone uses them in one form or another every day. However, most people only have a vague understanding of how computers work, let alone how to program them. Yet computing has become a commodity almost like energy, food, or water. The question is if the general public, for modern society to work properly, needs to understand computing better than what people generally know about, say, producing electricity or clean water. We argue that the intractability and even undecidability of so many important problems in computer science are the reason that computing is indeed different. It is the limits of computability, not just the capabilities of computers, that is the source of unbounded potential in the automation of everything. The challenge is to teach people not just programming but also how programming is the neverending process of overcoming those limits. We have developed a system called selfie that implements a self-referential compiler, emulator, and hypervisor that can compile, execute, and virtualize itself. We use selfie to teach undergraduate and graduate students computer science from first principles. In particular, we show them how self-referentiality in selfie is capability and limitation of computing at the same time. Here, we discuss ongoing early work on integrating verification technology into selfie as yet another way of exploring what computing is.

1 Introduction

How many people know how to read and write and understand at least some elementary arithmetic? What about elementary set theory, in particular, Cantor's diagonal argument then? And, if some people know and understand that, do they also know how diagonalization beautifully explains the limits of computability [13] as the source of unbounded potential in computing? Teaching and understanding Cantor is actually not that hard and should be part of any school curriculum. Connecting Cantor to computing, however, that is, teaching and understanding Gödel, Turing, and others, is a lot more difficult. But there ought to be a way to reach out to larger audiences. The trend towards teaching how to code already in school is an important step forward. However, we believe that understanding the limits of computability is at least as important as learning how to code, if not more important, especially for broader audiences.

We have developed a software system called selfie[1] in a tiny subset of C called C* that implements a self-compiling compiler, a self-executing emulator of a tiny MIPS subset called MIPSter targeted by the compiler, and a self-hosting hypervisor that virtualizes a MIPSter machine [8]. Selfie is written in a single, self-contained file of around 7k lines of C* code. Selfie compiles itself, executes the code it generates including the emulator itself, and can even virtualize the execution of that code.

Through compilation, emulation, and virtualization, selfie provides three different perspectives on how to create the semantics of formalisms such as C* and MIPSter code using these very formalisms. Selfie, just like Cantor's diagonal argument, employs self-referentiality. Learning about self-referentiality is difficult for many students but nevertheless seen by us as key to understanding basic principles of computer science including the limits of computability. Selfie is a sandbox for teaching undergraduate and graduate students computer science from first principles. However, we also use selfie in classes targeting broader audiences by identifying and exemplifying in selfie basic principles of computer science everyone should know about [8].

In order to provide another perspective on computing and in particular its limitations and how to overcome them, we have recently begun a new project integrating verification technology into selfie. We have already started implementing a state-of-the-art SAT solver in selfie and are working on an SMT solver and a symbolic execution engine for MIPSter. We report on the effort here which is still in an early stage but has already lead to some interesting insights.

We first introduce the programming language C* and the MIPSter instruction set by example and point out that we removed undefined behavior in all signed integer operations of C* through wrap-around semantics. However, as part of an ongoing effort, we are also exploring alternatives such as using unsigned integers only. Establishing a well-defined semantics of C* and MIPSter before trying to verify anything is in fact the first positive outcome of the effort.

We then provide an overview of selfie and argue that the simplicity and realism of system and language may lead to something beyond a purely educational effort. In particular, we show performance data comparing execution time and code size of programs written in C and ported to C* when compiled with state-of-the-art C compilers. The programs are mostly microbenchmarks written for this purpose but also one macrobenchmark which is a state-of-the-art SAT solver. It turns out that modern C compilers produce code for both versions with essentially the same performance characteristics. In short, C* is simple but still fast, motivating us to see it as a promising target for verification from within the system. We conclude with an outlook on how the project may evolve.

2 Programming

C* is a tiny subset of the programming language C [7]. C* features global variable declarations with optional initialization as well as procedures with parameters

[1] http://selfie.cs.uni-salzburg.at

and local variables. C* has five statements (assignment, while loop, if-then-else, procedure call, and return) as well as five built-in functions that are sufficient to bootstrap selfie (`exit`, `malloc`, `open`, `read`, `write`). In particular, there is no `free` and no `close`. C* features standard arithmetic (`+`, `-`, `*`, `/`, `%`) and comparison (`==`, `!=`, `<`, `<=`, `>`, `>=`) operators as well as integer, character, and string literals. C* includes the unary `*` operator for dereferencing pointers hence the name but excludes data types other than `int` and `int*` [12], bitwise and Boolean operators, and many other features.

Listing 1.1. Simplified `atoi` procedure in selfie

```
int atoi(int* s) {
  int i;
  int n;
  int c;

  // the conversion of the ASCII string in s to its
  // numerical value n begins with the leftmost digit in s
  i = 0;

  // and the numerical value 0 for n
  n = 0;

  // load character (one byte) at index i in s from memory
  // requires bit shifting since memory access is in words
  c = loadCharacter(s, i);

  // loop until s is terminated
  while (c != 0) {
    // the numerical value of ASCII-encoded decimal digits
    // is offset by the ASCII code of '0' (which is 48)
    c = c - '0';

    // assert: 0 <= c <= 9 and 0 <= n * 10 + c <= INT_MAX

    // use base 10 to compute numerical value
    n = n * 10 + c;

    // go to the next digit
    i = i + 1;

    c = loadCharacter(s, i);
  }

  return n;
}
```

The C* grammar is LL(1) with six keywords (`int`, `while`, `if`, `else`, `return`, `void`) and 22 symbols (`=`, `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `,`, `(`, `)`, `{`, `}`, `;`, `integer`, `character`, `string`, `identifier`). Whitespace is ignored including one-line comments (`//`).

For an example of C* code, consider Listing 1.1 which shows a simplified C* implementation of the standard `atoi` procedure for converting a (decimal) number represented as an ASCII string into its numerical integer value. An ASCII string in C is a null-terminated sequence of bytes, one byte per character, contigously stored in memory. Here, the only non-standard part of the implementation are the calls to `loadCharacter` in Lines 15 and 31 to retrieve the individual characters of the string `s`. Because of the lack of a byte-size data type such as `char` we can only access memory at the granularity of signed integers (or pointers to signed integers). Signed integers contain four bytes, that is, up to four characters which are then retrieved individually through bit shifting, see Listing 1.2 for details, in particular Lines 13 through 15.

Listing 1.2. `loadCharacter` procedure in selfie

```
 1  int loadCharacter(int* s, int i) {
 2    // assert: i >= 0
 3    int a;
 4
 5    // a is the index of the word where the
 6    // to-be-loaded i-th character in s is
 7    a = i / SIZEOFINT;
 8
 9    // shift to-be-loaded character to the left
10    // resetting all bits to the left of it then
11    // shift to-be-loaded character all the way
12    // to the right and return
13    return rightShift(leftShift(*(s + a),
14      ((SIZEOFINT - 1) - (i % SIZEOFINT)) * 8),
15        (SIZEOFINT - 1) * 8);
16  }
```

The C* semantics in selfie is, to the best of our knowledge, standard C semantics except for undefined behavior through arithmetic overflow. In particular, C* programs compiled and executed by selfie implement standard C semantics with 32-bit wrap-around semantics for all arithmetic operators on signed integers and pointers. This is true even if the bootstrapping compiler does not implement wrap-around semantics (except for multiplication). The system nevertheless prints a console warning for any overflow (and division by zero) that occurs during runtime. However, because of efficiency concerns, the actual result of a multiplication operation depends on the semantics implemented by the bootstrapping compiler.

The MIPSter instruction set generated by selfie is a tiny subset of MIPS32 [5]. It consists of 16 instructions (`nop`, `addu`, `subu`, `multu`, `divu`, `mfhi`, `mflo`, slt, jr, syscall, addiu, lw, sw, beq, jal, j). MIPSter allows straightforward compilation of C* programs into MIPSter code. Bitwise and sub-word data transfer instructions are not needed.

Listing 1.3. MIPS assembly for the atoi procedure in selfie

```
0x168(~18): lw $t0,-12($fp) // while (c != 0)
0x16C(~18): addiu $t1,$zero,0
0x170(~18): beq $t0,$t1,4[0x184]
0x178(~18): addiu $t0,$zero,1
0x17C(~18): beq $zero,$zero,2[0x188]
0x184(~18): addiu $t0,$zero,0
0x188(~18): beq $zero,$t0,31[0x208]
0x190(~21): lw $t0,-12($fp) // c = c - '0'
0x194(~21): addiu $t1,$zero,48
0x198(~21): subu $t0,$t0,$t1
0x19C(~21): sw $t0,-12($fp)
0x1A0(~26): lw $t0,-8($fp) // n = n * 10 + c
0x1A4(~26): addiu $t1,$zero,10
0x1A8(~26): multu $t0,$t1
0x1AC(~26): mflo $t0
0x1B8(~26): lw $t1,-12($fp)
0x1BC(~26): addu $t0,$t0,$t1
0x1C0(~26): sw $t0,-8($fp)
0x1C4(~29): lw $t0,-4($fp) // i = i + 1
0x1C8(~29): addiu $t1,$zero,1
0x1CC(~29): addu $t0,$t0,$t1
0x1D0(~29): sw $t0,-4($fp)
0x1D4(~31): lw $t0,8($fp) // push s onto call stack
0x1D8(~31): addiu $sp,$sp,-4
0x1DC(~31): sw $t0,0($sp)
0x1E0(~31): lw $t0,-4($fp) // push i onto call stack
0x1E4(~31): addiu $sp,$sp,-4
0x1E8(~31): sw $t0,0($sp)
0x1EC(~31): jal 0xE1D[0x3874] // call loadCharacter(s, i)
0x1F4(~31): addiu $t0,$v0,0 // c = loadCharacter(s, i)
0x1F8(~31): addiu $v0,$zero,0
0x1FC(~31): sw $t0,-12($fp)
0x200(~34): beq $zero,$zero,-39[0x168] // go back to while
```

Listing 1.3 shows the MIPSter code generated by selfie for the while loop in the atoi code in Listing 1.1. The first line reads as follows. The instruction lw $t0,-12($fp) is stored in memory at address 0x168 and has been generated for source code at approximately Line 18. In fact, the instruction loads the current value of the local variable c occurring in the loop condition c != 0 into the temporary CPU register $t0. The value is stored in memory on the call stack, 12 bytes below the address to which the frame pointer $fp refers. The next instruction loads the value `0` into another temporary register `$t1` to prepare for the comparison with `$t0` in the following branch instruction. The code that follows is inefficient but straightforward to generate keeping the compiler simple. It loads `1` or `0` into `$t0` depending on whether the loop condition evaluates to true or false, respectively. Only then the branch instruction at `0x188` either enters the loop body or terminates the loop by branching to the first instruction past the code implementing the loop body at `0x208`.

Listing 1.4. C* code preventing integer overflows in the `atoi` procedure in selfie

```
...
while (c != 0) {
  c = c - '0';

  if (c < 0)
    // c was not a decimal digit
    return -1;
  else if (c > 9)
    // c was not a decimal digit
    return -1;

  // use base 10 but avoid integer overflow
  if (n < INT_MAX / 10)
    n = n * 10 + c;
  else if (n == INT_MAX / 10) {
    if (c <= INT_MAX % 10)
      n = n * 10 + c;
    else if (c == (INT_MAX % 10) + 1)
      // s must be terminated next, check below
      n = INT_MIN;
    else
      // s contains a decimal number larger than INT_MAX
      return -1;
  } else
    // s contains a decimal number larger than INT_MAX
    return -1;

  i = i + 1;
  c = loadCharacter(s, i);

  if (n == INT_MIN)
    if (c != 0)
      // n is INT_MIN but s is not terminated yet
      return -1;
}
...
```

The rest of the code is hopefully self-explanatory leaving us more space to show in Listing 1.4 the actual implementation of `atoi` in selfie that prevents the occurrence of any integer overflows during scanning of integer literals. The code is considerably more complex than the simplified code in Listing 1.1 but nevertheless an important part of the educational experience with selfie. In fact, the whole implementation of selfie is designed to avoid integer overflows and there are indeed none in our experiments which include self-compilation, self-execution, and self-hosting of selfie. A proof of absence is of course another story which is ongoing work as discussed next.

3 Computing

Selfie is a self-contained system of a C* compiler, a MIPSter emulator, and a MIPSter hypervisor implemented in a single 7k-line file of C* code.[2] Selfie can compile, execute, and virtualize itself. In particular, the C* compiler targets the MIPSter emulator which can execute any MIPSter code including its own implementation any number of times until time and space run out. The MIPSter hypervisor virtualizes the machine emulated by the emulator and can therefore host any MIPSter code execution, again including its own implementation any number of times. The difference between emulator and hypervisor is that the emulator interprets MIPSter code while the hypervisor asks the machine on which it runs to interpret MIPSter code on its behalf in temporal and spatial isolation through context switching and virtual memory. Thus the hypervisor requires at least one emulator instance to run on.

Self-compilation, self-execution, and self-hosting with selfie enable three distinguished features of the system:

1. Selfie not only compiles itself, it can even execute the compiled code in the same invocation of the system, to compile itself again and enable checking if the code it then generates is the same as the code it executes (fixed-point of self-compilation). The backend of the compiler is even implemented next to the frontend of the emulator. In particular, encoding and decoding of machine instructions is literally done next to each other in the source code. Also, system call wrappers are generated by the compiler next to the actual system call implementations in the emulator.
2. Selfie can execute any MIPSter code including itself [11]. Interestingly, executing an emulator such as mipster on itself is arguably the simplest form of an operating system kernel (top emulator) running on a given processor (bottom emulator), just very inefficiently as interpreter of code rather than through context switching and virtual memory. However, the top emulator does provide a machine instance perfectly isolated from the machine instance on which it runs as provided by the bottom emulator. In fact, enhancing the top emulator to emulate multiple isolated machine instances is easy using a dedicated interpreter per instance. With selfie this can be done by students

[2] https://github.com/cksystemsteaching/selfie/releases/tag/Festschrift17

in a one-week homework assignment. It is only the speed of code execution that gets exponentially slower in the number of emulators running on top of each other, but for that there is virtualization.

3. Selfie can virtualize a MIPSter machine for hosting the execution of any MIPSter code including itself. In contrast to code interpretation, however, machine virtualization maintains the speed of code execution modulo the overhead of context switching and virtual memory. Nevertheless, emulator and hypervisor in selfie share most of their code and are supposed to provide functionally indistinguishable machine instances, just through very different means. Selfie can even alternate between emulation and virtualization of the same machine instance at runtime.

Selfie has originally been developed exclusively for educational purposes. By now, we use the system in introductory architecture, compiler, and operating systems classes. There is also an advanced operating systems class based on selfie as well as an introductory computer science class for students not majoring in computer science. In that class selfie helps exemplifying basic principles of computer science. A textbook in early draft form is available online.[3]

3.1 Software Verification in Selfie

We recently noticed that selfie has also potential in making more advanced topics such as software verification accessible to broader and younger audiences. We have therefore started a project integrating verification technology into selfie. Surprisingly, this effort appears to have potential in research as well. Selfie is simple yet realistic. In fact, programming language and software system are so simple that selfie may be amenable to formal reasoning from within the system itself. Moreover, C* is still sufficiently realistic to serve as input to state-of-the-art compiler optimizations. In short, C* is simple but may still be fast.

While there is considerable amount of literature on software verification and related topics there is little hands-on information on how to do what we are interested in. Ideally, we would like to have selfie perform self-verification of non-trivial correctness properties [9] and self-optimization of its own implementation. This may or may not be feasible but at least verifier and optimizer "only" need to work on selfie code and not "any" code. We are even free to trade-off complexity between subject and object. In other words, we can make the system smarter or the code, that is, its proof obligations simpler.

Because of selfie's dual role in education and research, we are interested in the absolute simplest yet sufficiently efficient design. There are essentially two key challenges that we are facing in this project. The first challenge is to figure out what the simplest way of doing things actually is. For example, it is difficult to choose the right data structures sufficient for our purpose. The second challenge is to figure out which optimizations dominate others in their effect on performance and scalability and are actually needed for our purpose.

[3] http://leanpub.com/selfie

So far, we have considered SAT and SMT solvers [2], (bounded) model checkers [1, 2, 14], symbolic execution engines [3, 4], and even inductive theorem provers [6]. The very first step we have already taken is to implement a naïve SAT solver in selfie which we call babysat. We also implemented a parser for SAT instances encoded in the DIMACS CNF file format. The babysat algorithm simply enumerates all possible variable assignments of a given SAT instance and takes 58 lines of C* code. In contrast, even the most naïve implementations we found online feature some form of optimization such as a watchlist, for example. However, at least for SAT, instead of introducing optimizations one by one, we decided to take microsat,[4] an existing state-of-the-art SAT solver implemented in C, port it to C*, and integrate it into selfie.

We are now working on implementations of a naïve SMT solver with babysat and microsat in the backend as well as a naïve symbolic execution engine for MIPSter code. The idea is to stay away from any optimizations unless we have evidence that they are really needed to make the system scale up to selfie. Also, naïve implementations often turn out to have significant pedagocial value helping students and us understand the problem better.

3.2 C* Performance

To our initial surprise, we noticed that the C* port of microsat is not slower than the original C version in our experiments. The port of microsat to C* essentially requires three potentially performance-relevant modifications:

1. structs and arrays are eliminated by removing their declarations and replacing their access operators with adequate pointer arithmetic and casting,
2. control-flow statements such as `break`, `continue`, and `goto` are eliminated by introducing new variables that enable modifying the truth value of loop conditions without changing the rest of the program state, and
3. logical operators such as `&&`, `||`, and `!` are eliminated by replacing them with adequately nested `if` statements.

All other modifications are minor. In particular, macros are easily expanded, `for` loops are turned into `while` loops, and, more surprisingly, all bitwise operators in the microsat implementation can be replaced by simple integer arithmetic.

We report on a macrobenchmark with microsat and on eight microbenchmarks each focusing on a particular aspect of the three modifications. Our experiments ran on a 512GB NUMA machine with four 16-core 2.3 GHz AMD Opteron 6376 processors (16KB L1 data cache, 64KB L1 instruction cache, 16MB L2 cache, 16MB L3 cache) and Linux kernel version 3.13.0. We used gcc 6.4 and 7.2 as well as clang 4.0.1 to generate 32-bit binaries with `-O0` and `-O3` optimization levels. For simplicity, selfie only supports 32-bit binaries.

Tables 1, 2, and 3 show the data obtained with x86 binaries generated by gcc 6.4, gcc 7.2, and clang 4.0.1, respectively. The microsat performance data

[4] https://github.com/marijnheule/microsat

Table 1. Performance and binary size of C* over C using gcc 6.4

Benchmark	-O0				-O3			
	C [sec.]	C* [sec.]	Perf. [%]	Size [%]	C [sec.]	C* [sec.]	Perf. [%]	Size [%]
microsat	4036.26	6912.24	58.3	64.1	2671.34	2678.82	99.6	88.9
struct	97.41	327.48	29.7	91.8	0.23	0.25	92	92.8
array	160.63	308.83	52	91.6	7.09	39.05	18.2	101.1
array pointer	-	270.85	59.3	91.3	-	7.13	99.4	93.6
struct with `int*`	-	326.9	29.8	91.5	-	0.23	100	93
array of `int*`	-	306.61	52.4	91.3	-	7.12	99.6	93.6
break	65.12	65.51	99.4	99.6	26.71	17.17	155.6	100.5
logical and	90.63	90.6	100	100	41.49	41.49	100	100
logical or	118.85	100.15	118.7	100	68.02	68.06	99.9	100

Table 2. Performance and binary size of C* over C using gcc 7.2

Benchmark	-O0				-O3			
	C [sec.]	C* [sec.]	Perf. [%]	Size [%]	C [sec.]	C* [sec.]	Perf. [%]	Size [%]
microsat	4037.47	6907.40	58.4	64	2866.97	2817.88	101.9	77.8
struct	110.77	327.92	33.8	91.6	0.31	0.31	100	92.9
array	160.03	312.1	51.3	91.4	4.69	37.67	12.5	98.3
array pointer	-	264.67	60.5	91.1	-	4.7	99.8	93.3
struct with `int*`	-	325.67	34	91.6	-	0.31	100	92.9
array of `int*`	-	307.86	52	91.3	-	4.69	100	93.3
break	64.81	65.51	98.9	99.6	26.71	26.71	100	99.9
logical and	90.06	90.14	99.9	100	24.13	24.1	100.1	100
logical or	120.55	101.89	118.3	100	44.92	44.91	100	100

Table 3. Performance and binary size of C* over C using clang 4.0.1

Benchmark	-O0				-O3			
	C [sec.]	C* [sec.]	Perf. [%]	Size [%]	C [sec.]	C* [sec.]	Perf. [%]	Size [%]
microsat	4011.74	7787.89	51.5	73.3	2630.39	2514.90	104.2	90.1
struct	89.26	418.07	21.4	98.2	33.8	22.7	148.9	98.2
array	106.02	416.14	25.5	98.3	34.42	32.66	105.4	98.3
array pointer	-	329.07	32.2	98.2	-	6	573.7	98.2
struct with `int*`	-	418.85	21.3	98.2	-	33.86	99.8	98.2
array of `int*`	-	420.94	25.2	98.2	-	6.01	572.7	98.2
break	72.05	72.41	99.5	99.9	27.63	27.63	100	99.9
logical and	99.4	98.93	100.5	100	48.27	48.27	100	100
logical or	131.77	137.98	95.5	100	65.97	65.97	100	100

shows the total execution time of running microsat on the industrial benchmark of the SAT 2004 competition[5] with a timeout of 120 seconds. Instances that took less than 5 seconds to solve were excluded. We repeated the experiment to obtain a ±5% margin of error with a probability of 90%. For the microbenchmark performance data the margin of error is ±2% with a probability of 99%.

In nearly all cases, the size of x86 binaries generated from C* versions of the code is either the same or less than the size of the binaries generated from the corresponding C versions. Without any compiler optimizations (option `-O0`), code generated from C* versions is generally slower by up to around 79%. There is one notable exception which is the microbenchmark replacing the logical operator `||` with adequate `if` statements. Code generated from the C* version with both versions of gcc runs around 18% faster in this case.

With compiler optimizations (option `-O3`), the picture is quite different. Performance is generally the same for both the C and C* versions in nearly all benchmarks, in particular the macrobenchmark with microsat. However, there are some notable exceptions as well. For example, struct access mimicked in C* and compiled with clang is faster than the original in C. However, with array access and gcc it is the opposite.

Structs and Arrays. In order to measure the impact of porting C structs and arrays to C* pointer arithmetics and casting, we designed five microbenchmarks that share the code in Listing 1.5 for exercising struct and array accesses in two nested `while` loops. The `input1` and `input2` parameters were set to 200,000,000 and 100, respectively.

Listing 1.5. Code for microbenchmarking struct and array access performance

```
int* ptr; // used in inlined code
...;
while (i < input1) {
  j = input2;
  while (j > 0) {
    // inline code here for microbenchmarking
    // struct and array access performance
    ...;
    j = j - 1;
  }
  i = i + 1;
}
```

Listing 1.6 shows the C code for measuring struct and array access performance. For example, for measuring struct access performance we inlined the body of the `struct-access` procedure into the body of the inner `while` loop in Listing 1.5.

[5] http://www.satcompetition.org

Listing 1.6. struct and array microbenchmarks in C

```c
struct strc_t {
  int* f1;
  int f2;
  int f3;
};
struct strc_t* strc;

void struct_access() {
  strc->f1 = ptr + j;
  strc->f2 = j;
  strc->f3 = strc->f2 + j;
}

void array_access() {
  strc->f1[j] = 1;
  strc->f2 = j;
  strc->f3 = strc->f2 + j;
}
```

Listing 1.7 shows the C* version of the C code in Listing 1.6 using getters and setters for struct and array access through pointer arithmetics and casting. This method is used in our C* port of microsat as well as in the implementation of selfie. The array access performance with C* is poor using both versions of gcc with `-O3` since they fail to move the loop-invariant base address of the array outside the loops.

Listing 1.7. struct and array microbenchmarks in C*

```c
int* getF1(int* strc) { return (int*) *strc; }
int  getF2(int* strc) { return *(strc + 1);  }
int  getF3(int* strc) { return *(strc + 2);  }
void setF1(int* strc, int* f1) { *strc = (int) f1; }
void setF2(int* strc, int  f2) { *(strc + 1) = f2; }
void setF3(int* strc, int  f3) { *(strc + 2) = f3; }

int* strc;

void struct_access() {
 setF1(strc, ptr + j);
 setF2(strc, j);
 setF3(strc, getF2(strc) + j);
}

void array_access() {
 *(getF1(strc) + j) = 1;
 setF2(strc, j);
 setF3(strc, getF2(strc) + j);
}

int* array;

void array_pointer_access() {
 *(array + j) = 1; // with array set to getF1(strc)
 setF2(strc, j);
 setF3(strc, getF2(strc) + j);
}
```

We therefore designed another microbenchmark in C* for measuring array access performance when using a variable called `array` that caches the pointer to the beginning of the accessed array. The data shows that this modification restores the array access performance of the optimized C* code compiled with both versions of gcc. With clang performance even multiplies by a factor of five. However, the code generated for the array access microbenchmark in C as well as in C* is inefficient.

Listing 1.8. struct and array microbenchmark with content typed as `int*`

```
int* getF1(int** strc) {return *strc; }
int  getF2(int** strc) {return (int) *(strc + 1);}
int  getF3(int** strc) {return (int) *(strc + 2);}
void setF1(int** strc, int* f1) {*strc = f1;}
void setF2(int** strc, int  f2) {*(strc + 1) = (int*) f2;}
void setF3(int** strc, int  f3) {*(strc + 2) = (int*) f3;}

int** strc;

void struct_with_intstar_access() {
 setF1(strc, ptr + j);
 setF2(strc, j);
 setF3(strc, getF2(strc) + j);
}

void array_of_intstar_access() {
 *(getF1(strc) + j) = 1;
 setF2(strc, j);
 setF3(strc, getF2(strc) + j);
}
```

The fact that array access performance with C* is so poor with both versions of gcc made us look even closer and design two more microbenchmarks as shown in Listing 1.8. The code is not proper C* which we acknowledge by underlining the obtained data. However, the code still reveals that gcc is able to optimize struct and array access through pointer arithmetics and casting if the struct is declared as `int**` rather than just `int*` and casting in the getters and setters is adapted accordingly.

Control Flow. Listing 1.9 shows on the left C code using a `break` statement and on the right its equivalent C* version. The C* code avoids the `break` statement by using a variable `tmp` for saving and restoring the loop variable `j` which we modify temporarily in order to mimic the behavior of the original C code. We used this technique in the C* port of microsat to eliminate `break`, `continue`, and `goto` statements.

Listing 1.9. C code and its equivalent C* version replacing `break`

```
                                    int tmp;
while (i < input1) {                while (i < input1) {
  j = input2;                         j = input2;
  while (j > 0) {                     while (j > 0) {
    if (j < input3) {                   if (j < input3) {
      ...;                                ...;
                                          tmp = j;
      break;                              j = 0;
    }                                   } else
    j = j - 1;                            j = j - 1;
  }                                   }
                                      j = tmp;
  i = i + 1;                          i = i + 1;
}                                   }
```

The data shows that performance is generally unaffected except when using gcc 6.4 with `-O3`. In this case, surprisingly, performance increases significantly with the C* version.

Boolean Operators. Listing 1.10 shows the microbenchmark for eliminating the logical operator `&&` using adequately nested `if` statements. The microbenchmark for `||` works similarly. We used this technique in the C* port of microsat.

Listing 1.10. C code and its equivalent C* version replacing `&&`

```
while (i < input1) {                while (i < input1) {
  j = input2;                         j = input2;
  while (j > 0) {                     while (j > 0) {
    if (j < input3 && j % 12) {         if (j < input3)
                                          if (j % 12) {
      ...;                                  ...;
    }                                     }
    j = j - 1;                          j = j - 1;
  }                                   }
  i = i + 1;                          i = i + 1;
}                                   }
```

The data shows that performance is generally maintained. In fact, the generated code is almost the same for the C and C* versions using any of the compilers with `-O3`. As mentioned before, performance even improves for `||` when using both versions of gcc with `-O0`.

4 Conclusions

Software verification is difficult. Many important problems in that field are intractable or even undecidable. However, that challenge and the fact that verification provides another way of constructing the semantics of formalisms is our motivation to try integrating verification technology into selfie. We see verification integrated with compilation, emulation, and virtualization as key to advancing both the rigorous and efficient design of software systems as well as the computer science education of broader audiences. What is the meaning of code and how is it constructed by a machine? What happens during execution? How does that become a utility? Why can I not compute everything and how is this a good thing? The simplicity and realism of selfie has already helped us give increasingly better answers to some of these questions. We conclude that with the verification technology already available there is a good chance that we are able to continue that development with selfie even in a largely intractable and undecidable problem domain.

Acknowledgements. Selfie is the result of more than ten years of studying and teaching systems engineering. Many colleagues and students have been involved in its development through inspiring conversations as well as numerous coding sessions. In particular, we thank Martin Aigner, Sebastian Arming, Christian Barthel, Armin Biere, Heidi Graf, Andreas Haas, Marijn Heule, Thomas Hütter, Michael Lippautz, Cornelia Mayer, James Noble, Simone Oblasser, Peter Palfrader, Sarah Sallinger, Raja Sengupta, and Ana Sokolova as well as all the students who have taken our compiler and operating systems classes over the years. The design of the C* compiler is inspired by the Oberon compiler of Professor Niklaus Wirth from ETH Zurich [15]. The design of the microkernel in selfie is inspired by microkernels of Professor Jochen Liedtke from University of Karlsruhe [10].

Work on selfie has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund (FWF) Grant S11404-N23 and S11411-N23), a Google PhD Fellowship, and a Google Research Grant.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1579, pp. 193–207. Springer (1999)
2. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)

3. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI). pp. 209–224. USENIX Association (2008)
4. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Proc. Symposium on Network and Distributed Systems Security (NDSS). pp. 151–166 (2008)
5. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann (2011)
6. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer (2000)
7. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice Hall (2000)
8. Kirsch, C.: Selfie and the basics. In: Proc. ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). ACM (2017)
9. Kumar, R.: Self-compilation and self-verification. Phd thesis, University of Cambridge (2015)
10. Liedtke, J.: Toward real microkernels. Commun. ACM 39(9), 70–77 (Sep 1996)
11. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proc. ACM Annual Conference. pp. 717–740 (1972)
12. Richards, M., Whitby-Strevens, C.: BCPL: The Language and its Compiler. Cambridge University Press (2009)
13. Sipser, M.: Introduction to the Theory of Computation. International Thomson Publishing (1996)
14. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. Proceedings of the IEEE 103(11), 2021–2035 (2015)
15. Wirth, N.: Compiler Construction. Addison Wesley (1996)