Generative Programming for Fast and Secure System Software

PI: Tiark Rompf
Email: tiark@purdue.edu, Web: tiarkrompf.github.io

Project Overview

How should we build systems that are performant, secure, and correct? Today, systems-level software such as network stacks, databases, and control code in embedded devices is almost always developed in C. This is suboptimal, for at least two reasons. First, development is far less agile and productive than in higher level languages; and second, low-level code in unsafe languages invites security vulnerabilities.

High-level languages, on the other hand, rule out entire classes of vulnerabilities through built-in memory safety guarantees, and many of them provide elaborate mechanisms for programming with contracts, which enforce user-defined specifications at runtime. But while implementations of high-level languages have come a long way, programmers cannot trust them to deliver the same reliable performance as handwritten C code. As a consequence, PL advances are mostly lost on programmers operating under tight performance constraints. Sound static verification of general-purpose C code is possible, but has an extraordinary cost in terms of user annotations, and is thus rarely done in practice.

User code:

```
def min(a: Rep[Int], b: Rep[Int]) = if (a < b) a else b

val xs = ... // type Rep[Array[Int]]
val res = xs.fold(Int.MaxValue)(min)
```

Generated code:

```
int *x16 = .. // input data
int x17  = .. // input length

int x32 = 0x7FFFFFFF;
/*@ loop invariant (0 <= x24 <= x17);
    loop invariant \forall int x23;
      (0 <= x23 < x24)  ==>  (x16[x23] >= x32);
    loop assigns x24,x32;
    loop variant (x17 - x24); */
for (int x24 = 0; x24 < x17; x24++) {
  int x29 = x16[x24];
  int x31 = x32;
  int x33 = 0;
  int x34 = x29 < x31;
  if (x34) {
    x33 = x29;
  } else {
    x33 = x31;
  }
  x32 = x33;
}
int x35 = x32; // result
```

Figure 1: High-level user code in Scala (top) generates low-level C code (bottom), extended with automatically inferred annotations that enable static verification.

State of the Art: Generative Programming for Performance Motivated by the apparent trade-off between productivity and performance, the PI and other researchers have argued for a rethinking of the role of high-level languages in performance critical code, with the goal of allowing developers to leverage high-level programming abstractions without the hefty price in performance. The shift in perspective that enables this vision of "abstraction without regret" is a properly executed form of generative programming: instead of running the whole system in a high-level managed language runtime, the abstraction power of high-level languages can be focused on generating and composing pieces of low-level code—effectively acting as a glorified macro system.

Generative programming has proven successful for numerical kernels such as FFTs [7], for DSLs in big-data processing [2], but also in domains like database query engines [6]. In these traditional strongholds of low-level programming, generative programming has largely lived up to its promise: programmers reap the benefits of programming in a high-level language without the low-level language drawbacks.

Proposed Research: Generative Programming for Performance, Security, and Correctness Previous work has only looked at productivity benefits, not at security and correctness. Since low-level code is still being generated, the safety guarantees usually associated with high-level languages (e.g. memory safety) do not carry over. And what about functional correctness? We have to trust the high-level code and the generator, either or both of which might contain bugs.

The main thrust of this proposal is to demonstrate that "abstraction without regret" holds for security and verification in the same way it does for performance. We propose to extend generative programming methods to emit low-level C code enriched with specifications so that the code can be formally verified using existing C-level verification tools, as shown in Figure 1.

Fast and Secure Network Protocols

As particular area of application, we will show that it is possible to implement a stack of network protocols in a very high-level programming style, with state-of-the-art performance and verified memory safety. Our first focus will be on the HTTP protocol, but eventually, we plan to extend our approach towards cryptographic protocols auch as TLS/SSL, and also towards multiple layers of protocols, alle the way up from TCP/IP.

Parsing as Attack Vector The area of network protocols and input parsing is extremely relevant in practice. Working with untrusted input requires care, and many vulerabilities arise at this boundary, which are often overlooked. For example, after the celebrated static verification of PolarSSL 1.1.8 [9], a remote code execution vulnerability was found due to a bug in an ASN.1 parser in the X.509 module, which was not part of the verification effort [1, 8]. Exploitable bugs in HTTP parsers have been reported for all major web servers, e.g. Nginx [4] and Apache [3].

module / instance	lang.	LoP	LoC	#	s.
1. **Selection Sort**	Scala	41	115		
sorted & in-place permut.					
ints by $\leq$	C	88	26	70	7
ints by $\geq$	C	88	26	70	7
int pairs by first proj.	C	116	43	89	11
int pairs by lex.	C	130	52	97	20
int vectors by length	C	128	49	109	15
2. **Linear Algebra**	Scala	32	97		
misc. safe & spec.					
matrix $+,\times,\cdot$	C	104	51	101	22
member	C	23	22	18	2
index where	C	26	17	20	3

Figure 2: Verification effort: lines of proof/specification (LoP) & lines of code (LoC) in Scala code generator and generated C code, which Frama-C verifies using given number of goals (#) in given seconds (s.).

From Fast to Verified We plan to implement a stack of network protocols from TCP/IP up to and including HTTP and HTTPS in a very high-level programming style, with state-of-the-art performance and verified memory safety. In our preliminary work, we used parser *generator* combinators [5] to build an HTTP parser that is as fast as Nginx. We show performance results in Figure 3, comparing to the HTTP parser of Nginx, which is written in C as well. On a benchmark for parsing HTTP responses and validating the payload length, our approach is competitive, processing one million items per second.

parser	requests per second
(baseline) nginx	$(0.94 \pm 0.01) \cdot 10^6$
(our) staged	$(1.00 \pm 0.01) \cdot 10^6$

Figure 3: HTTP perf. (higher is better)

The key idea behind this proposal is to extend this generative approach to emit *specifications* along with C code, so that the generated code can be verified by existing C-level tools. We plan to use Frama-C and the ANSI C Specification Language (ACSL). We show some preliminary results for verified sorting and linear algebra kernels in Figure 2.

References

[1] ARMmbed. PolarSSL security advisory 2014-04, 2015. https://tls.mbed.org/tech-updates/security-advisories/polarssl-security-advisory-2014-04.

[2] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. PACT, 2011.

[3] CVE. 2002-0392: Apache security advisory. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0392.

[4] CVE. 2013-2028: nginx security advisory. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028.

[5] M. Jonnalagedda, T. Coppey, S. Stucki, T. Rompf, and M. Odersky. Staged parser combinators for efficient data processing. In *OOPSLA*, 2014.

[6] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[7] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in scala: towards the systematic construction of generators for performance libraries. In *GPCE*, 2013.

[8] J. Regehr. Comments on a formal verification of PolarSSL, 2015. http://blog.regehr.org/archives/1261.

[9] TrustInSoft. PolarSSL 1.1.8 verification kit, 2015. http://trust-in-soft.com/polarSSL_demo.pdf.

Data Policy

We intend to publish results of this project at academic conferences, and to release all developed software artifacts as open source under a permissive license.

Budget

Funding is requested for one gradutate student for one year, including tuition remission, equipment, and travel.

- Personnel - Grad Staff (.50 FTE): $27,797.04
- Employee Benefits - Grad Staff (.50 FTE): $2,612.92
- Grad Fee Remissions: $10,260.00
- Travel (conferences, domestic and abroad): $4,000.00
- Equipment (laptop/workstation and cloud computing resources): $3,000.00
- **Total: $37,420.22**