Benchmarks for C CorrectnessVerifiers

Andrew W. Appel
August 2022

Part 1: Lennart Beringer's slides about VST

Part 2: my slides about benchmark examples

Foundational verification of C programs using VST

Lennart Beringer

Verified Software Toolchains Workshop

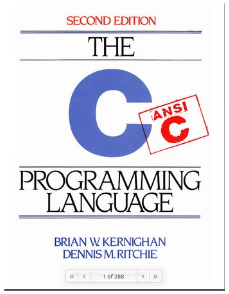
July 2022

PRINCETON UNIVERSITY

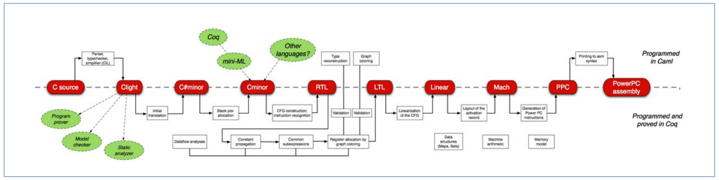
Isaac Newton Institute for Mathematical Sciences

Verified Software Toolchain

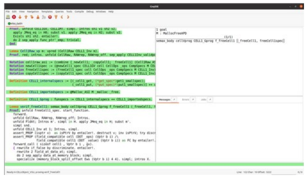
Expressive separation logic $$\frac{\vdash P\ \{c\}\ Q}{\vdash P * R\ \{c\}\ Q * R}$$ for



implemented and proven sound with respect to



in the proof assistant enabling semi-automated

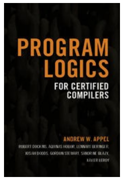


construction of **functional correctness** proofs which connect to

model-level reasoning.

Verified Software Toolchain (VST)

Goal: expressive & practical logic, provably sound w.r.t. 



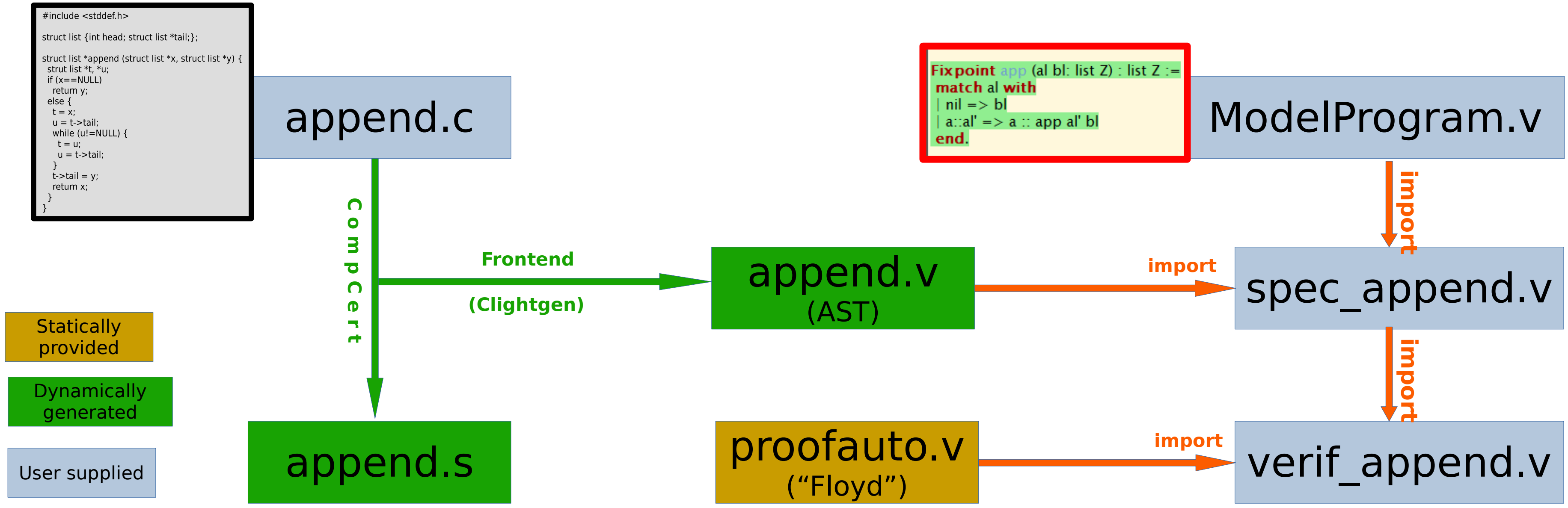
CUP 2014

- Expressive assertions: separation logic, shallowly embedded
- Artifact of interest: CompCert Clight
- Reasonable automation using symbolic execution + SL tactics
- Full support for concurrency, function pointers (higher-order),
- Machine-checked soundness w.r.t. Clight operational semantics

Cao, Beringer, Gruetter, Dodds, Appel:
VST-Floyd: A separation logic tool to verify correctness of C programs.
Journal of Automated Reasoning 2018

Characteristics and tool flow

- verification happens on CompCert's program representation (Clight AST) in Coq
- forward symbolic execution + "manual" solving of remaining side conditions
- can construct loop invariants "on the fly" (no source code annotations)
- may refer to (Coq) values resulting from already processed code segment



Properties

- **(memory) safety**
- **absence of undefined behavior, uncontrolled overflow,...**
- **partial-correctness (incl. adherence to ”external-interaction protocols”)**
- **concurrency: no races (“well-synchronized”)**
- **information confinement (from SL)**

… for CompCert Clight ‘programs’, w.r.t. (constructions derived from) operational semantics in Coq.

Verified Software Toolchain

Outline

Applications

- Memory management:

  malloc/free, garbage collection

- Cryptographic primitives
- Networking and communication

Recent extensions

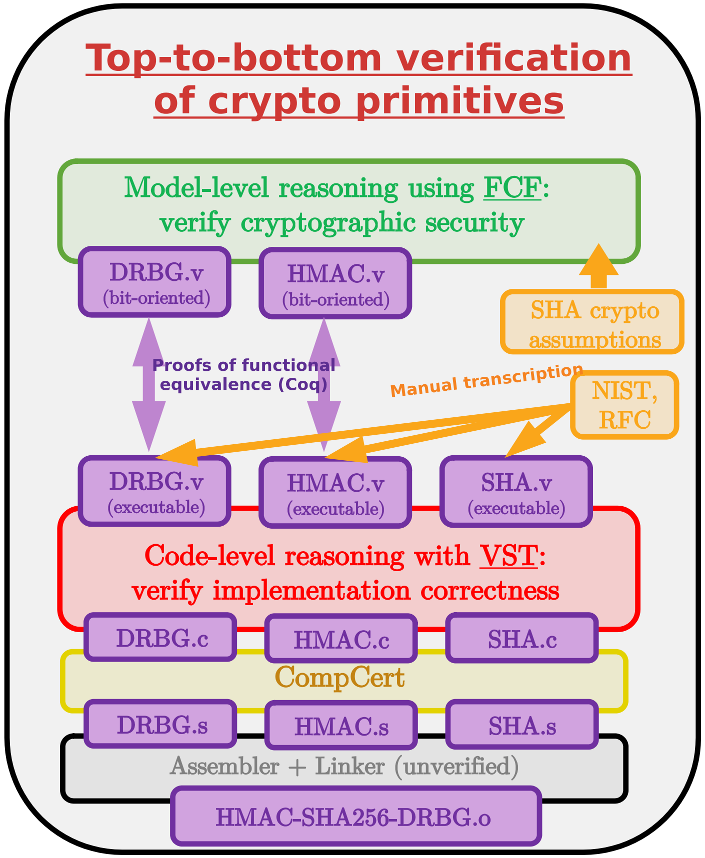
- Modular programming => modular specification and verification
- External interactions and concurrency

Verified crypto primitives

Appel: Verification of a Cryptographic Primitive: SHA-256. TOPLAS 2015

Beringer et al: Verified Correctness and Security of OpenSSL HMAC. USENIX Security 2015

Ye et al.: Verified Correctness and Security of mbedTLS HMAC-DRBG. CCS 2017



External uses

Schwabe et al: A Coq Proof of the Correctness of X25519 in TweetNaCl. CSF'21

Judging from VST pull requests: at least one blockchain

Judging from a job advert: AWS

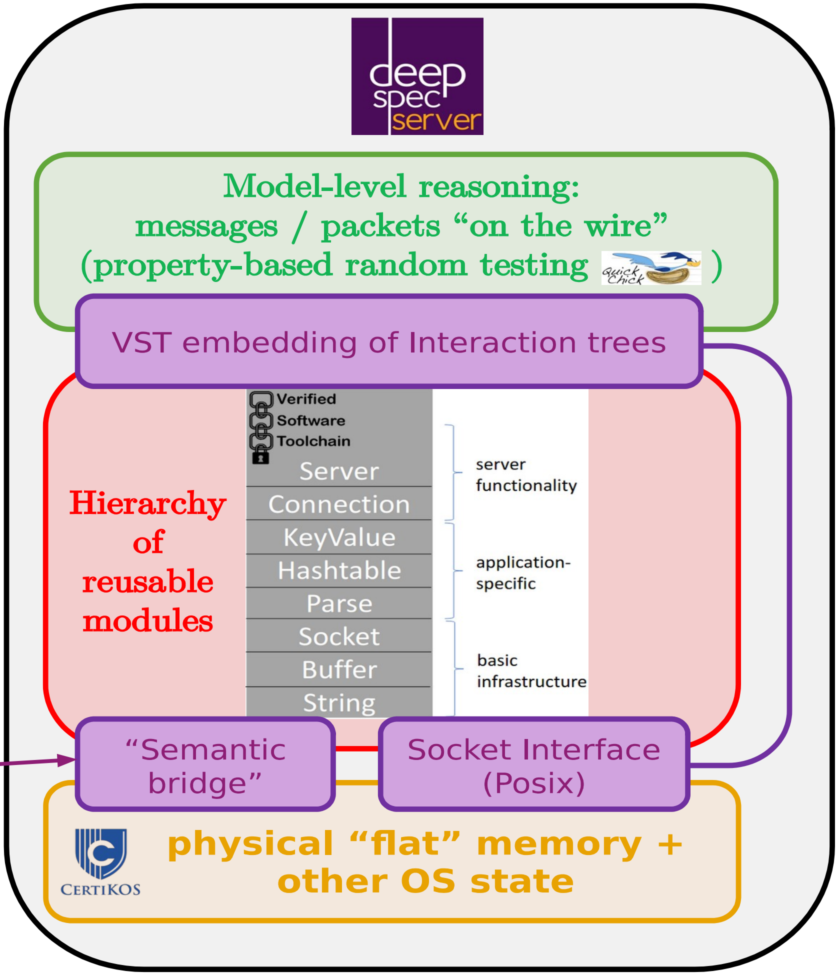
DeepSpec Web server

“echo server”: cf. talk by B. Jacobs

Koh et al.: From C To Interaction Trees. CPP’19

Zhang et al: Verifying an HTTP Key-Value Server with Interaction Trees and VST. ITP’21

Mansky et al: Connecting Higher-Order Separation Logic to a First-Order Outside World. ESOP’20



Memory management

Verified Sequential Malloc/Free

Andrew W. Appel
Princeton University
USA
appel@princeton.edu

David A. Naumann
Stevens Institute of Technology
USA
naumann@cs.stevens.edu

Abstract

We verify the functional correctness of an array-of-bins (segregated free-lists) single-thread malloc/free system with respect to a correctness specification written in separation logic. The memory allocator is written in standard C code compatible with the standard API; the specification is in the Verifiable C program logic, and the proof is done in the Verified Software Toolchain within the Coq proof assistant. Our "resource-aware" specification [illegible] will successfully return [illegible] specification that allows [illegible] it wants to. We also pro[illegible] (refin[illegible]): the resource-aware specification implies a resource-oblivious spec.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Functionality**; **Software verification**.

Keywords: memory management, separation logic, formal verification

ACM Reference Format:
Andrew W. Appel and David A. Naumann. 2020. Verified Sequential Malloc/Free. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management (ISMM '20), June 16, 2020, London, UK*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3381898.3397211

we now verify the correctness of malloc/free. This also serves as a demonstration and assessment of the verification tool.

C's malloc/free library casts undifferentiated bytes to and from the data structures that it uses internally; the client of the library casts to and from its own structs and arrays. In the process, implicit alignment restrictions must be respected. For a formal verification, it is not enough that the program actually respect these restrictions: we want the program logic (or verification tool) to be sound w.r.t. those restrictions—it should [illegible] violate them.

In f[illegible] object-size restrictions, and [illegible] C are quite subtle [35]. We want the program logic (and verification tool) to be sound (proved sound with a machine-checked proof) with respect to the operational semantics of C (including alignment constraints, etc.).

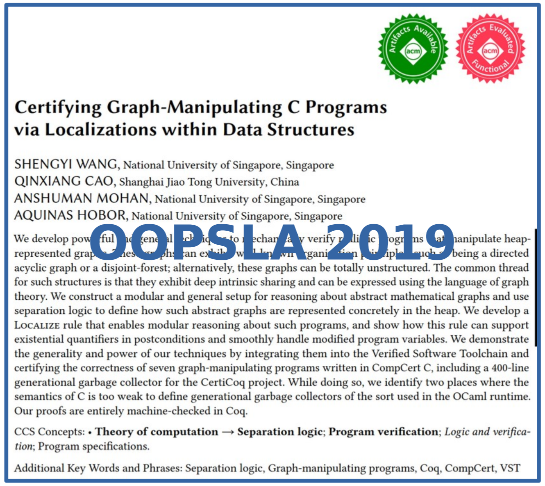
Allocation and freeing should be (amortized) constant-time. The usual method is Weinstock's array-of-bins data structure for quickly finding free blocks of the right size [36]. Large blocks must be treated separately; a modern memory allocator can manage large blocks directly using the mmap system call.

To do formal verification, we should use a suitable program logic. C programs that use pointer data structures are most naturally specified and verified in separation logic (SL) [29].

There must be a formal specification—otherwise we cannot prove correctness, only weaker properties such as mem-

Reused in / linked to other VST projects

Future work:
high-performance, concurrent,...



Certifying Graph-Manipulating C Programs via Localizations within Data Structures

SHENGYI WANG, National University of Singapore, Singapore
QINXIANG CAO, Shanghai Jiao Tong University, China
ANSHUMAN MOHAN, National University of Singapore, Singapore
AQUINAS HOBOR, National University of Singapore, Singapore

We develop pow[illegible] verify r[illegible] manipulate heap-represented grap[illegible] being a directed acyclic graph or a disjoint-forest; alternatively, these graphs can be totally unstructured. The common thread for such structures is that they exhibit deep intrinsic sharing and can be expressed using the language of graph theory. We construct a modular and general setup for reasoning about abstract mathematical graphs and use separation logic to define how such abstract graphs are represented concretely in the heap. We develop a Localize rule that enables modular reasoning about such programs, and show how this rule can support existential quantifiers in postconditions and smoothly handle modified program variables. We demonstrate the generality and power of our techniques by integrating them into the Verified Software Toolchain and certifying the correctness of seven graph-manipulating programs written in CompCert C, including a 400-line generational garbage collector for the CertiCoq project. While doing so, we identify two places where the semantics of C is too weak to define generational garbage collectors of the sort used in the OCaml runtime. Our proofs are entirely machine-checked in Coq.

CCS Concepts: • **Theory of computation → Separation logic**; **Program verification**; *Logic and verification*; Program specifications.

Additional Key Words and Phrases: Separation logic, Graph-manipulating programs, Coq, CompCert, VST

Nontrivial reasoning about graphs in SL

Theory of ramification

Current use: garbage collection of

Communication / networking

A Verified Messaging System

WILLIAM MANSKY, Princeton University, USA
ANDREW W. APPEL, Princeton University, USA
ALEKSEY NOGIN, HRL Laboratories, LLC, USA

We present a concurrent-read exclusive-write buffer system with strong correctness and security properties. Our motivating application for this system is the distribution of sensor values in a multicomponent vehicle-control system, where some components are unverified and possibly malicious, and other components are vehicle-control-critical and must be verified. Valid participants are guaranteed correct communication (i.e., the writer is always able to write to an unused buffer, and readers always read the most recently published value), while invalid readers or writers cannot compromise the correctness or liveness of valid participants. There is only one writer, all operations are wait-free, and there is no extra process or thread mediating communication. We prove the correctness of the system with valid participants by formally verifying a C implementation of the system in Coq, using the Verified Software Toolchain extended with an atomic exchange operation. The result is the first C-level mechanized verification of a nonblocking communication protocol.

CCS Concepts: • **Software and its engineering → Software verification**; *Semantics*; • **Computer systems organization** → *Embedded software*;

Additional Key Words and Phrases: shared-memory communication, shared-memory concurrency, concurrent separation logic

ACM Reference format:
William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A Verified Messaging System. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 87 (October 2017), 28 pages.
https://doi.org/10.1145/3133911

OOPSLA 2017

Early example of shared-memory communication

Code provided by or developed jointly with industrial partners

Verified Erasure Correction in Coq with MathComp and VST

Joshua M. Cohen, Qinshi Wang, and Andrew W. Appel

Princeton University, Princeton NJ 08544, USA
in CAV'22: 34th International Conference on Computer Aided Verification, August 2022

Abstract. Most methods of data transmission and storage are prone to errors, leading to data loss. Forward erasure correction (FEC) is a method to allow data to be recovered in the presence of errors by encoding the data with redundant parity information determined by an error-correcting code. There are dozens of classes of such codes, many based on sophisticated mathematics, making them difficult to verify using automated tools. In this paper, we present a formal, machine-checked proof of a C implementation of FEC based on Reed-Solomon coding. The C code has been actively used in network defenses for over 25 years, but the algorithm it implements was partially unpublished, and it uses certain optimizations whose correctness was unknown even to the code's authors. We use Coq's Mathematical Components library to prove the algorithm's correctness and the Verified Software Toolchain to prove that the C program correctly implements this algorithm, connecting both using a modular, well-encapsulated structure that could easily be used to verify a high-speed, hardware version of this FEC. This is the first end-to-end, formal proof of a real-world FEC implementation; we verified all previously unknown optimizations and found a latent bug in the code.

Keywords: Reed-Solomon coding · functional correctness verification · interactive theorem proving.

CAV 2022

Code in production use for 20 years

Reengineered mathematical justification of algorithmic design choices

Concurrency (led by W. Mansky, UIUC)

Verified Software Toolchain
CSL with lock invariants (Gotsman/Hobor)

Iris
IPM
Expressive Invariants
Ghost state

TaDA
Logically atomic specifications

Verified Software Toolchain 2022

Atomic triple

l: local
p: public

$\langle s.\ \mathsf{is\_stack}\ p \mid \mathsf{stack}\ s\rangle\ \mathsf{push}(v)\ \langle \mathsf{is\_stack}\ p \mid \mathsf{stack}\ (v :: s)\rangle$

$$\langle s.\ P_l \mid P_p(s)\rangle\ c\ \langle Q_l \mid Q_p(s)\rangle$$

True before c

Invariant until linearization point (maybe for different s)

True after c

True immediately after lin. point

Applicable to

- concurrency primitives (eg locks)
- data structure operations (push/pop,..)

Sharma et al, ASL'22

compare proofs of atomic specs for BST ops based on hand-over-hand locking with

- atomic lock specs
- Gotsman/Hobor lock specs

Verified Software Toolchain

Modular verification of modular and object-oriented C code

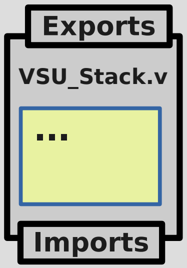
1. Function specification subsumption and intersection specs

- enables module reuse with different specs for different clients

$$\phi_1 \wedge \phi_2 <: \phi_i \qquad \frac{\psi <: \phi_1 \quad \psi <: \phi_2}{\psi <: \phi_1 \wedge \phi_2}$$

2. Verified Software Units

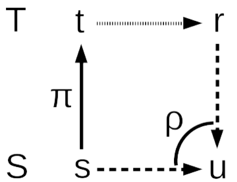
- VST-verified compilation units for Clight
- Composable at API-level
  - compatible with syntactic composition of Clight ASTs
  - sound w.r.t. VST's whole-program guarantee (C-level linking)
  - use abstraction principles of Coq to foundationally enforce representation independence



3. Dataless object encodings

- semantic subtyping
- code sharing (opaque)
- behavior inheritance and overriding

Positive Subtyping (Hofmann & Pierce '95)

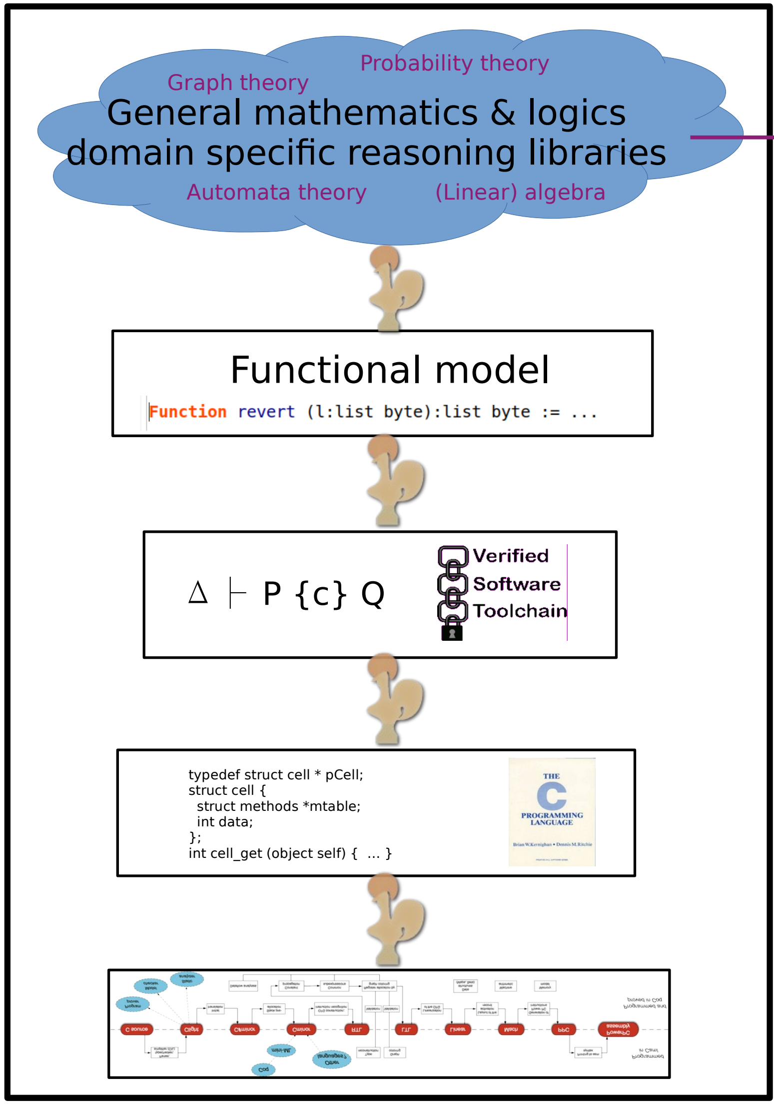


Object predicate entailments in SL
(with behaviors)

$$S_\beta s \vdash T_\beta(\pi\, s) * \forall r.\, T_\beta r \mathrel{-\!\!*} S_\beta(\rho\, r\, s)$$

Verified Software Toolchain

Coq as foundational IDE



Tools and libraries in Coq

- Mathcomp
- Foundational Cryptography Framework
- Flocq, Gappa, VCFloat
- CertiGraph
- Reglang
- Bignum, Interval, coqprime, coquelicot, ..
- Relation algebra, approximate data structures, Narcissus

Many in CoqPlatform!

Next steps: synthesis and automation

- Watanabe et al. ICFP’21
- Connect to CN, Verifast, … ?
- SAT/SMT tactics, hammer, list-solve, …

Axiomatic base / assumptions

- Operational semantics
- Definitions of domain-specific concepts in model
- Coq infrastructure (kernel, extraction, runtime system,…)

Benchmarks for C <u>correctness</u> verifiers

Andrew W. Appel

Princeton University

August 2022

Desiderata

Multilevel: The C program must not only be proved to implement a functional model, the functional model must be proved to actually do the desired thing.

Unified: The C-program proof and high-level proof should be done in the same logical framework so that they can be composed into a single end-to-end theorem.

Composable: if not in the same logical framework, *some* principled way to link the high-level proof with the low-level verification.

Low-expressive: The proof system for C-program proofs should be expressive enough to verify "dusty deck" programs that do all-too-clever things with data representations.

High-expressive: The proof system for high-level proofs should be expressive enough to verify high-level specifications with entirely nontrivial application-specific mathematics.

Modular: modular verification of modular programs with data abstraction.

Open-source: The C program and its proofs (low-level and high-level) should be open-source.

Documented: (if possible) in a paper so people can understand what it's about.

32/64-bit: Programs/proofs that are not portable must be marked as 32-bit or 64-bit.

Peter Sewell’s feature list

- **integer arithmetic: usual arithmetic conversions, arithmetic UBs,** ABI variants
- **floats: floating types**
- **characters and strings: string literals (incl. their potential aliasing),** C11 character-set features
- **structured data: basic structs, basic enums, basic unions,** ~~struct-as-value memory accesses and arguments/returns, union-as-value memory accesses and arguments/returns,~~ **compound type initialisers,** ~~bitfields,~~ flexible array members, variable length arrays
- **control flow: C evaluation order, loops (for, while, do, break, continue), switch** (**structured** vs general), ~~C goto~~, **function pointers,** ~~setjump,~~ signal
- **function calls: mutable function parameters,** variadic arguments, **function parameters of array type with "static" or ***
- **lifetime: block lifetimes**, thread-local storage
- **memory object model: basic model, passing addresses of locals in function calls, storing addresses of locals (e.g. in globals)**, pointer/integer casts (incl. arith on unused/unused bits), **pointer arithmetic** using offset-of, subobject provenance, **effective types**, ~~uninitialised reads,~~ restrict, register
- unsequenced races
- **concurrency (C/C++11 or Linux-kernel fragments), volatile**
- C11 generic selection
- standard library
- ...and various ownership idioms.

A benchmark for C program verification

Marko van Eekelen Daniil Frumin
Léon Gondelman Robbert Krebbers
Sjaak Smetsers Freek Verbeek Benoît

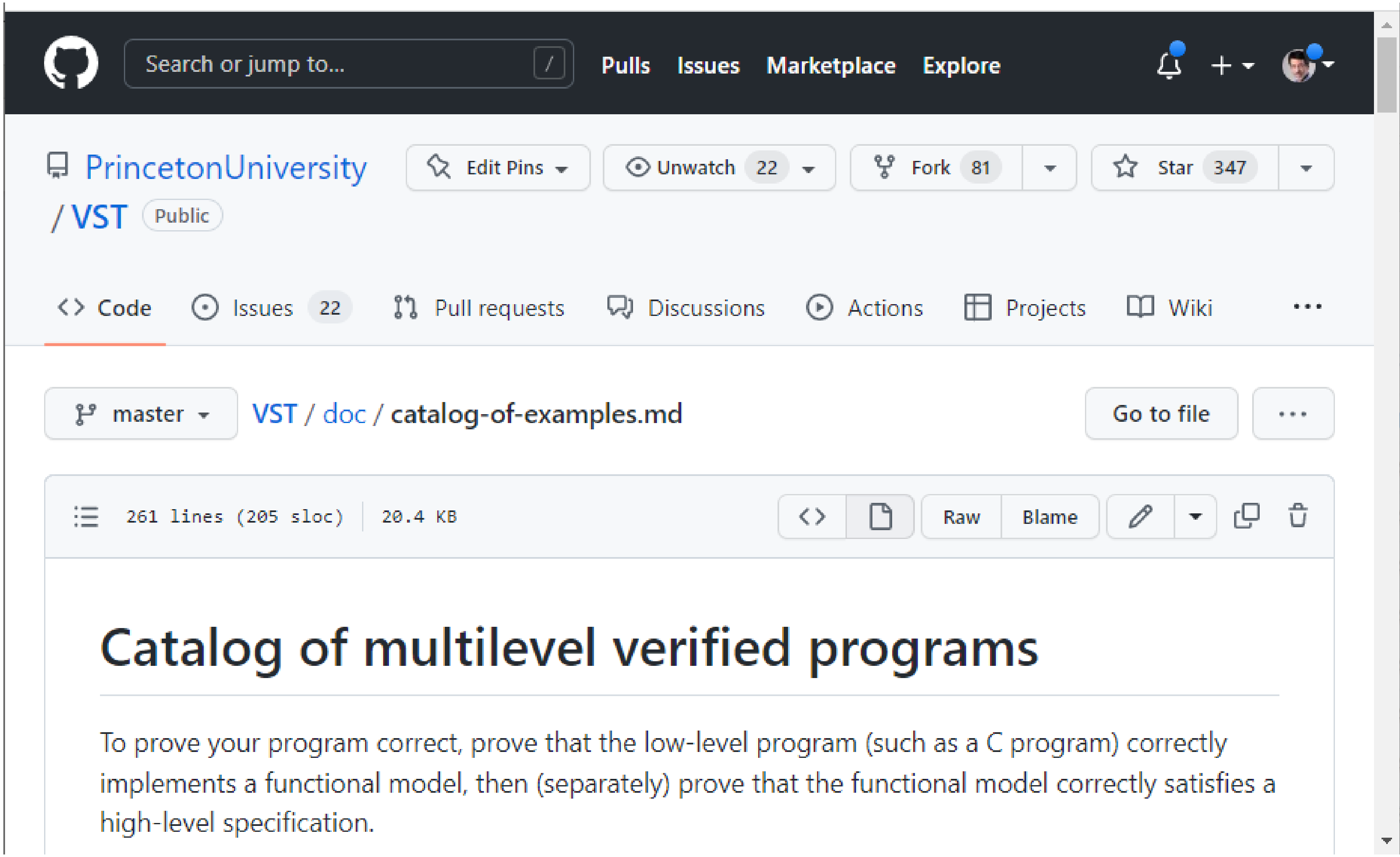
April 1, 2019

Abstract

We present twenty-five C programs, as a benchmark for C p
methods. This benchmark can be used for system demons
cation effort between systems, and as a friendly competitio
a scoring formula that allows a verification system to score

Points scored on the benchmark thus far

	VeriFast	VST
Total	4	50
`fac1.c`	4	4
`fac2.c`		4
`fac3.c`		4
`fac4.c`		4
`fac6.c`		4
`cat1.c`		4
`cat2.c`		4
`malloc1.c`		4
`qsort1.c`		4
`qsort3.c`		4
`qsort4.c`		3
`sqrt1.c`		4
`sqrt1.c`		3



Catalog of multilevel verified programs

To prove your program correct, prove that the low-level program (such as a C program) correctly implements a functional model, then (separately) prove that the functional model correctly satisfies a high-level specification.

Desiderata

Multilevel: The C program must not only be proved to implement a functional model, the functional model must be proved to actually do the desired thing.

Unified: The C-program proof and high-level proof should be done in the same logical framework so that they can be composed into a single end-to-end theorem.

Composable: if not in the same logical framework, *some* principled way to link the high-level proof with the low-level verification.

Low-expressive: The proof system for C-program proofs should be expressive enough to verify "dusty deck" programs that do all-too-clever things with data representations.

High-expressive: The proof system for high-level proofs should be expressive enough to verify high-level specifications with entirely nontrivial application-specific mathematics.

Modular: modular verification of modular programs with data abstraction.

Open-source: The C program and its proofs (low-level and high-level) should be open-source.

Documented: (if possible) in a paper so people can understand what it's about.

32/64-bit: Programs/proofs that are not portable must be marked as 32-bit or 64-bit.

VST's catalog

SHA-256: Secure Hash Algorithm

Theorem: an early release of OpenSSL correctly implements the FIPS-180 specification.

HMAC: Hash-based Message Authentication Code, a keyed cryptographic hash function, from OpenSSL.

Theorem. OpenSSL 0.9.1c correctly implements the FIPS standards for HMAC and SHA, and implements a cryptographically secure PRF (pseudorandom function) subject to the usual assumptions about SHA.

HMAC-DRBG: Widely used cryptographic random number generator standardized by NIST.

Theorem. mbedTLS HMAC-DRBG correctly implements the NIST 800-90A standard, and produces pseudorandom output: an adversary with $< 2^{78}$ cycles has a $0.5+2^{-52}$ chance of predicting the next bit.

Forward Erasure Correction: Reconstruct missing network packets (or RAID disks).

Theorem: The dusty-deck 1997 C program, if no more than h of (k+h) packets are lost, will reconstruct them.

Quicksort (three different versions from cbench benchmark suite)

Theorem: The C programs sort correctly.

Newton's method square root (from cbench benchmark suite)

Theorem: Accurately computes square roots within 5 ulp.

VST's catalog (cont'd)

Ordinary Differential Equation by Leapfrog integration
Theorem: a correct solution within the accuracy bound, including discretization and round-off error.

Binary Search Trees
Theorem: correctly implements a lookup table.

Concurrent messaging system, using lock-free atomics
Theorem: receiver reads latest version of the message.

Generational garbage collector
Theorem: correctly preserves an isomorphic graph while deleting garbage.

Malloc-free system with size classes
Theorem: satisfies the expected separation-logic spec.

Abstract and concrete data types
The "pile" program demonstrating modular ADT verification using Verified Software Units.

Many small program examples: from VST's demonstration (and continuous integration) suite.