VSTlib: Library Components for Verified C Programs

Coq Workshop, July 2023, Białystok, Poland

Andrew W. Appel

Princeton
University

a program logic for C

and

a tool in Coq for proving programs

foundationally sound
higher-order impredicative
ghostly concurrent
separation logic
for C programs
in Coq
with tactical proof automation

“Rich” Functional Correctness Proofs

CFML

Programs verified

SHA-256 (cryptographic hash, 2014)
HMAC (cryptographic authentication, 2015)
HMAC-DRBG (crypto. random numbers, 2017)
Concurrent messaging system (2017)
Generational garbage collector (2019)
Malloc/free system (2020)
Quicksort, etc. (2020)
Newton's method in floating-point (2020)
FEC (Reed-Solomon error correction, 2021)
Differential equation integrator (2022)
Jacobi-iteration linear solver (2023)

Outline

- ✓ What is VST
- ➢ VST function specs and proofs
- Modular verification of modular programs
- The library: VSTlib

C program / fun. specification

```
int myfunc(int x) {
  int *p;
  p = (int*)malloc(sizeof(*p));
  if (p) {
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1;
}
```

Definition myfunc_spec :=
 DECLARE _myfunc
 WITH x: Z, gv: globals
 PRE [tint]
  PROP (Int.min_signed ≤ x < Int.max_signed)
  PARAMS (Vint (Int.repr x))
  GLOBALS (gv)
  SEP (mem_mgr gv)
 POST [tint]
  PROP()
  RETURN(Vint (Int.repr (x+1)))
  SEP(mem_mgr gv).

C program / fun. specification

```
int myfunc(int x) {
  int *p;
  p = (int*)malloc(sizeof(*p));
  if (p) {
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1;
}
```

```
Definition myfunc_spec :=
 DECLARE _myfunc
 WITH   x: Z,   gv: globals
 PRE [ tint ]
    PROP (Int.min_signed ≤ x < Int.max_signed)
    PARAMS (Vint (Int.repr x))
    GLOBALS (gv)
    SEP (mem_mgr gv)
 POST [ tint ]
    PROP()
    RETURN(Vint (Int.repr (x+1)))
    SEP(mem_mgr gv).
```

C program / fun. specification

```
int myfunc(int x) {
  int *p;
  p = (int*)malloc(sizeof(*p));
  if (p) {
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1;
}
```

```
Definition myfunc_spec :=
 DECLARE _myfunc
 WITH   x: Z,   gv: globals
 PRE [ tint ]
    PROP (Int.min_signed ≤ x < Int.max_signed)
    PARAMS (Vint (Int.repr x))
    GLOBALS (gv)
    SEP (mem_mgr gv)
 POST [ tint ]
    PROP()
    RETURN(Vint (Int.repr (x+1)))
    SEP(mem_mgr gv).
```

C program / fun. specification

```
int myfunc(int x) {
  int *p;
  p = (int*)malloc(sizeof(*p));
  if (p) {
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1;
}
```

```
Definition myfunc_spec :=
 DECLARE _myfunc
 WITH  x: Z,  gv: globals
 PRE [ tint ]
   PROP (Int.min_signed ≤ x < Int.max_signed)
   PARAMS (Vint (Int.repr x))
   GLOBALS (gv)
   SEP (mem_mgr gv)
 POST [ tint ]
   PROP()
   RETURN(Vint (Int.repr (x+1)))
   SEP(mem_mgr gv).
```

C program / fun. specification

f_myfunc

```
int myfunc(int x) {
  int *p;
  p = (int*)malloc(sizeof(*p));
  if (p) {
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1;
}
```

Definition myfunc_spec :=
DECLARE _myfunc
WITH x: Z, gv: globals
PRE [tint]
PROP (Int.min_signed ≤ x < Int.max_signed)
PARAMS (Vint (Int.repr x))
GLOBALS (gv)
SEP (mem_mgr gv)
POST [tint]
PROP()
RETURN(Vint (Int.repr (x+1)))
SEP(mem_mgr gv).

Definition Γ := [. . . ; myfunc_spec; . . . ; malloc_spec; free_spec; . . .].

Lemma body_myfunc: semax_body Γ f_myfunc myfunc_spec.

Proving a function body correct

Lemma body_myfunc: semax_body Γ f_myfunc myfunc_spec.
Proof.

Qed.

Assertions

```
int myfunc(int x) {
  int *p;
  p = (int*)malloc(sizeof(*p));
  if (p) {
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1;
}
```

PROP (. . .)
LOCAL (gvars gv; temp _x (Vint (Int.repr x)))
SEP (mem_mgr gv)

PROP (. . .)
LOCAL (temp _p p;
gvars gv;
temp _x (Vint (Int.repr x)))
SEP (mem_mgr gv;
malloc_token Ews tint p;
data_at Ews tint Vundef p))

PROP (. . .)
LOCAL (temp _p p;
gvars gv;
temp _x (Vint (Int.repr x)))
SEP (mem_mgr gv;
malloc_token Ews tint p;
data_at Ews tint (Vint (Int.repr x)) p))

Assertions

```
int myfunc(int x) {
  int *p;
  p = (int*)malloc(sizeof(*p));
  if (p) {
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1;
}
```

PROP (. . .)
LOCAL (gvars gv; temp _x (Vint (Int.repr x)))
SEP (mem_mgr gv)

PROP (. . .)
LOCAL (gvars gv; temp _x (Vint (Int.repr (x+1))))
SEP (mem_mgr gv)

Proving a function body correct

```
Lemma body_myfunc: semax_body Vprog Gprog f_myfunc myfunc_spec.
Proof.
 start_function.
 forward_call (malloc_spec_sub(tint))  gv.
 Intros p.
 if_tac.  (* is p equal to NULL? *)
 - (* p == NULL *)
   forward_if; subst; try contradiction.
   forward.
 - (* p <> NULL *)
   forward_if; subst; try contradiction.
   Intros.
   forward.
   forward.
   forward.
   forward_call (free_spec_sub (tint)) (p,gv).
   if_tac; try contradiction; cancel.
   forward.
Qed.
```

```
PROP ( . . . )
LOCAL (gvars gv; temp _x (Vint (Int.repr x)))
SEP (mem_mgr gv)
```

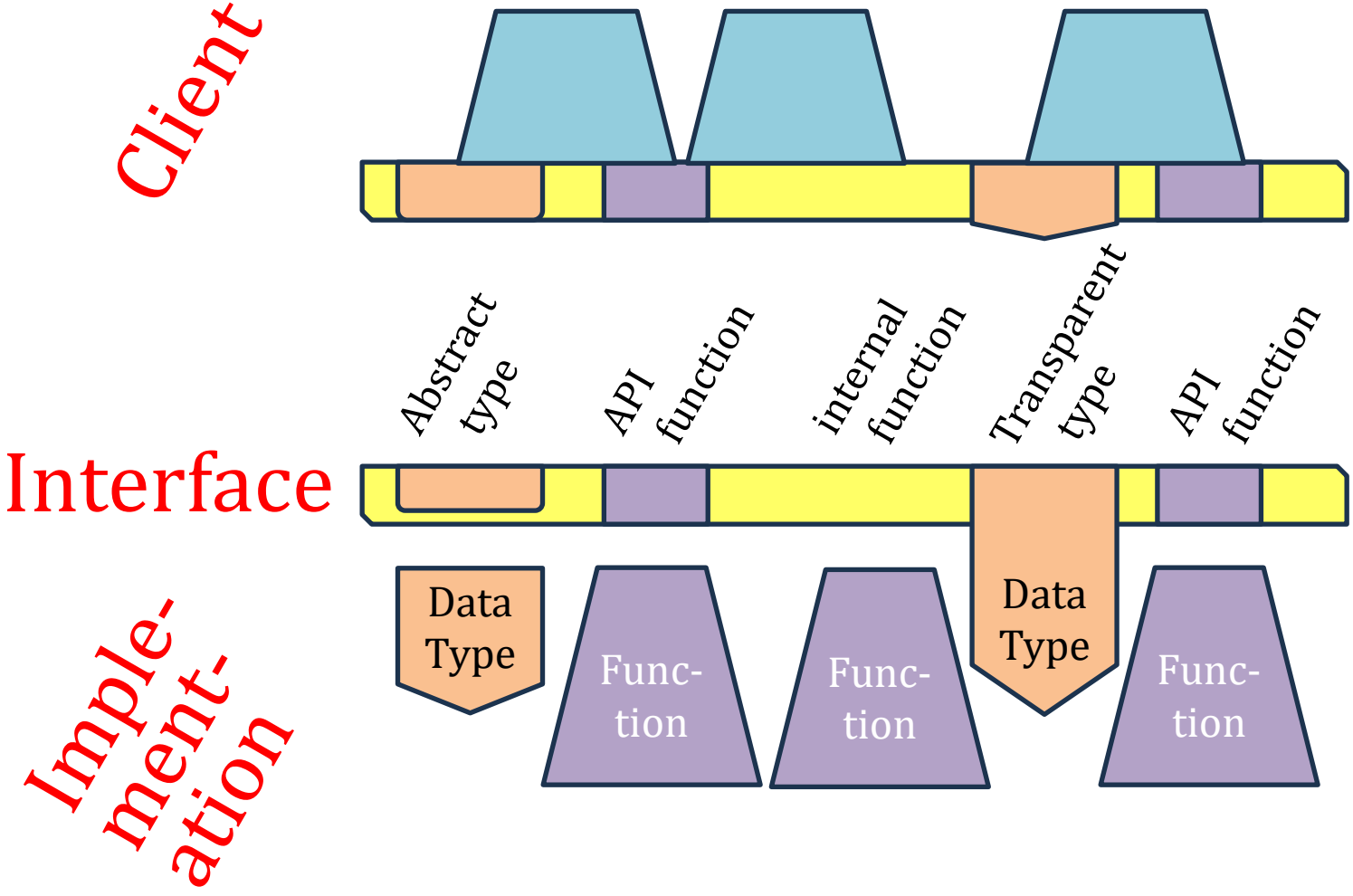
```
PROP ( . . . )
LOCAL (temp _p p;
          gvars gv;
          temp _x (Vint (Int.repr x)))
 SEP (mem_mgr gv;
       malloc_token Ews tint p;
       data_at   Ews   tint  Vundef   p))
```

```
PROP ( . . . )
LOCAL (temp _p p;
          gvars gv;
          temp _x (Vint (Int.repr x)))
SEP (mem_mgr gv;
      malloc_token Ews tint p;
      data_at   Ews   tint  (Vint (Int.repr x))   p))
```

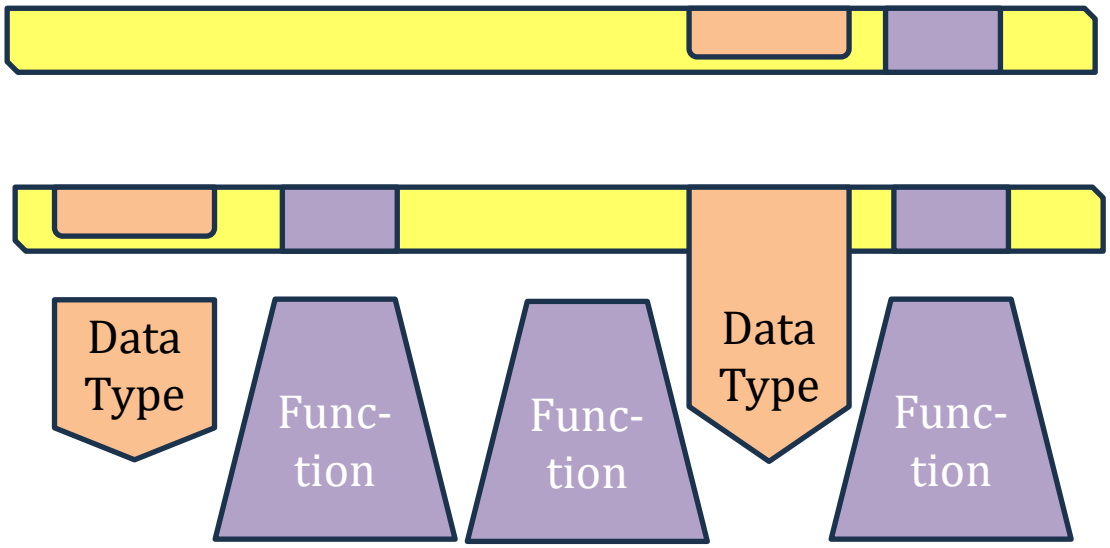
Outline

- ✓ What is VST
- ✓ VST function specs and proofs
- ➢ Modular verification of modular programs
- The library: VSTlib

Modular Verification of Modular Programs



Interface subtyping



VSU

The modules-and-interfaces calculus of VST is called

Verified Software Units [Beringer '21]

The calculus supports:

- Abstract or transparent data types (via opaque or transparent separation-logic predicates)
- Flexible connection of .c files to VSU descriptions
- Function-spec subtyping w/ subsumption
- Hiding internal components (functions, types)
- Multiple implementations for same interface

C program / fun. specification

f_myfunc

```
int myfunc(int x) {
  int *p;
  p = (int*)malloc(sizeof(*p));
  if (p) {
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1;
}
```

Definition myfunc_spec :=
DECLARE _myfunc
WITH x: Z, gv: globals
PRE [tint]
PROP (Int.min_signed ≤ x < Int.max_signed)
PARAMS (Vint (Int.repr x))
GLOBALS (gv)
SEP (mem_mgr gv)
POST [tint]
PROP()
RETURN(Vint (Int.repr (x+1)))
SEP(mem_mgr gv).

Definition Γ := [. . . ; myfunc_spec; . . . ; malloc_spec; free_spec; . . .].

Lemma body_myfunc: semax_body Γ f_myfunc myfunc_spec.

C program / fun. specification

f_myfunc

```
int myfunc(int x) {
  int *p;
  p = (int*)malloc(sizeof(*p));
  if (p) {
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1;
}
```

Definition myfunc_spec :=
DECLARE _myfunc
WITH x: Z, gv: globals
PRE [tint]
PROP (Int.min_signed ≤ x < Int.max_signed)
PARAMS (Vint (Int.repr x))
GLOBALS (gv)
SEP (mem_mgr gv)
POST [tint]
PROP()
RETURN(Vint (Int.repr (x+1)))
SEP(mem_mgr gv).

"Abstract Specification Interface"

Definition Γ := [myfunc_spec] ++ mallocASI ++ fooASI ++ barASI.

Lemma body_myfunc: semax_body Γ f_myfunc myfunc_spec.

C program / fun. specification

f_myfunc

```
int myfunc(int x) {
  int *p;
  p = (int*)malloc(sizeof(*p));
  if (p) {
    *p = x;
    x = (*p)+1;
    free(p);
    return x;
  }
  else return x+1;
}
```

Definition myfunc_spec :=
DECLARE _myfunc
WITH x: Z, gv: globals
PRE [tint]
PROP (Int.min_signed ≤ x < Int.max_signed)
PARAMS (Vint (Int.repr x))
GLOBALS (gv)
SEP (mem_mgr gv)
POST [tint]
PROP()
RETURN(Vint (Int.repr (x+1)))
SEP(mem_mgr gv).

"Abstract Predicate Declaration"

Outline

- ✓ What is VST
- ✓ VST function specs and proofs
- ✓ Modular verification of modular programs
- ➢ The library: VSTlib

`#include <stdlib.h>`

We take for granted . . .

Currently available

https://github.com/PrincetonUniversity/VST/tree/master/lib#readme

Name	Header	spec	VSU		Done?	
math	math.h	spec_ math	verif_math	Axiom-atized	mostly	
malloc	stdlib.h	spec_ malloc	verif_malloc	Axiom-atized	Done	standard system allocator
memmgr	memmgr.h			**Proved**	(coming soon)	verified malloc/free system
atomics	stdatomic.h, SC_atomics.h	spec_ SC_atomics	verif_ SC_atomics	Axiom-atized	Done	atomic load, store, CAS, etc.
locks	threads.h, VSTthreads.h	spec_locks	verif_locks	**Proved**	Done	busy-wait locks
threads		spec_ threads	verif_ threads	**Proved**	Done	

math.h

Double-precision functions: acos, acosh, asin, asinh, atan atan2, atanh, cbrt, copysign, cos, cosh, exp, exp2, expm1, fabs, pow, sqrt, sin, sinh, tan, tanh, fma, frexp, ldexp, nan, nextafter, trunc

Single-precision functions: acosf,acoshf,asinf,asinhf,atanf,atan2f,atanhf,cbrtf,copysignf, cosf,coshf,expf,exp2f,expm1f,fabsf,powf,sqrtf,sinf,sinhf,tanf,tanhf,fmaf,frexpf,ldexpf,nanf, nextafterf, truncf

Not yet axiomatized: *all long-double (128-bit float) functions; and both the double-precision and single-precision versions of* ceil, erf, erfc, fdim, floor, fmax, fmin, fmod, hypot, ilogb, j0, j1, jn, lgamma, llrint, llround, log, log10, log1p, log2, logb, lrint, modf, nearbyint, nexttoward, remainder, remquo, rint, round, scalbln, scalbn, tgamma, y0, y1, yn

Accuracy: The GNU library documentation explains how accurate each function is (from rigorous experimental measurement). VSTlib's funspecs axiomatize that accuracy as documented.

malloc/free

Axiomatization of the system malloc/free library.

```
Definition malloc_spec  {cs: compspecs} (t: type) :=
 DECLARE _malloc
  WITH gv: globals
  PRE [ size_t ]
     PROP (0 ≤ sizeof t ≤ Ptrofs.max_unsigned;
           complete_legal_cosu_type t = true;
           natural_aligned natural_alignment t = true)
     PARAMS (Vptrofs (Ptrofs.repr (sizeof t))) GLOBALS (gv)
     SEP (mem_mgr gv)
   POST [ tptr tvoid ]
    EX p: val,
     PROP ()
     RETURN (p)
     SEP (mem_mgr gv;
          if eq_dec p nullval
           then emp
           else (malloc_token Ews t p * data_at_ Ews t p)).
```

malloc/free

Axiomatization of the system malloc/free library.

```
Definition mall
 DECLARE _ma
  WITH gv: glo
  PRE [ size_t ]
    PROP (0 ≤
          compl
          natura
    PARAMS (V
    SEP (mem_
  POST [ tptr t
   EX p: val,
    PROP ()
    RETURN (p
    SEP (mem_
      if eq_de
      then em
      else (m
```

Coming soon: installation of the Appel/Naumann malloc/free (ISMM'20), which satisfies the same spec, but **verified, not axiomatized**.

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 can guarantee when malloc will successfully return a block, unlike the standard Posix specification that allows malloc to return NULL whenever it wants to. We also prove subsumption (refinement): 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 refuse to verify programs that violate them.

In fact, the alignment restrictions, object-size restrictions, and integer-overflow properties of 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 can-

atomic memory operations

C11 standard provides atomic load/store/CAS operations, in versions "sequentially consistent," "release-acquire," and "relaxed". We provide only the SC versions.

```
atom_int *make_atomic(int v);
extern int atom_load(atom_int *tgt);
extern void atom_store(atom_int *tgt, int v);
extern int atom_CAS(atom_int *tgt, int *c, int v);
extern int atom_exchange(atom_int *tgt, int v);
extern void free_atomic(atom_int *tgt);

extern atom_ptr *make_atomic_ptr(void * v);
extern void* atomic_load_ptr(atom_ptr *tgt);
extern void atomic_store_ptr(atom_ptr *tgt, void *v);
extern int atomic_CAS_ptr(atom_ptr *tgt, void **c, void *v);
extern void* atomic_exchange_ptr(atom_ptr *tgt, void *v);
extern void free_atomic_ptr(atom_ptr *tgt);
```

Axiomatized by William Mansky

shared-memory threads

A thin, portable layer over the C11 or posix_threads systems.

```
void spawn(int (*f)(void*), void* args);

void exit_thread(int r);
```

Axiomatized by William Mansky

semaphores ("daring" locks)

A thin, portable layer over the C11 or posix_threads systems.

```
lock_t makelock(void);

void freelock(lock_t lock);

void acquire(lock_t lock);

void release(lock_t lock);
```

Implemented and **verified** by William Mansky. Currently only busy-wait locks.

atomic memory operations

C11 standard provides atomic load/store/CAS operations, in versions "sequentially consistent," "release-acquire," and "relaxed". We prov

```
atom_int *make_ato
extern int atom_lo
extern void atom_s
extern int atom_CA
extern int atom_ex
extern void free_a

extern atom_ptr *m
extern void* atomi
extern void atomic
extern int atomic_                                          *v);
extern void* atomic_exchange_ptr(atom_ptr *tgt, void *v);
extern void free_atomic_ptr(atom_ptr *tgt);
```

Bringing Iris into the Verified Software Toolchain

William Mansky[1]

University of Illinois Chicago, USA

July 2022

Abstract. The Verified Software Toolchain (VST) is a system for proving correctness of C programs using separation logic. By connecting to the verified compiler CompCert, it produces the strongest possible guarantees of correctness for real C code that we can compile and run. VST included concurrency from its inception, in the form of reasoning about lock invariants, but concurrent separation logic (CSL) has advanced by leaps and bounds since then. In this paper, we describe efforts to integrate advancements from Iris, a state-of-the-art mechanized CSL, into VST. Some features of Iris (ghost state and invariants) are re-implemented in VST from the ground up; others (Iris Proof Mode) are imported from the Iris development; still others (proof rules for atomic operations) are axiomatized, with the hope that they will be made foundational in future versions. The result is a system that can prove correctness of sophisticated concurrent programs implemented in C, with fine-grained locking and non-blocking atomic operations, that yields varying soundness guarantees depending on the features used.

https://arxiv.org/pdf/2207.06574.pdf

How to get VSTlib

opam repository add coq-released

opam install coq-vst-lib

C include directory:

```
$(OPAM_SWITCH_PREFIX)/lib/coq/user-contrib/VSTlib/include
```

C sources directory:

```
$(OPAM_SWITCH_PREFIX)/lib/coq/user-contrib/VSTlib/src
```

Coq specs and theorems:

```
From VSTlib Require Import spec_malloc.    (* etc *)
```

Some current clients of VSTlib

Client	description	Client of:
LAProof (Kellison et al.)	sparse linear algebra, proved correct and **floating-point accurate**	math
Jacobi (Tekriwal et al.)	Jacobi-method linear solver, with **proved correctness and convergence**	math, malloc
pardotprod (Appel)	Parallel dot product	malloc, atomics, locks

Conclusion

A programming language is only as good as its libraries

A proof system is only as good as its libraries

A program verification system is only as good as
its libraries of verified components

VSTlib is just getting started . . . help it grow!