# Abstraction and Subsumption in Modular Verification of C Programs

Lennart Beringer and Andrew W. Appel

Princeton University, Princeton NJ 08544, USA

**Abstract.** Representation predicates enable *data abstraction* in separation logic, but when the same concrete implementation may need to be abstracted in different ways, one needs a notion of *subsumption*. We demonstrate *function-specification subtyping*, analogous to subtyping, with a *subsumption* rule: if $\phi$ is a funspec_sub of $\psi$, that is $\phi <: \psi$, then $x : \phi$ implies $x : \psi$, meaning that any function satisfying specification $\phi$ can be used wherever a function satisfying $\psi$ is demanded. We extend previous notions of Hoare-logic sub-specification, which already included parameter adaption, to include *framing* (necessary for separation logic) and impredicative bifunctors (necessary for higher-order functions, i.e. function pointers). We show intersection specifications, with the expected relation to subtyping. We show how this enables compositional modular verification of the functional correctness of C programs, in Coq, with foundational machine-checked proofs of soundness.

**Keywords:** Foundational Program Verification; Separation Logics; Specification Subsumption

## 1 Introduction

Even in the 21st century, the world still runs on C: operating systems, runtime systems, network stacks, cryptographic libraries, controllers for embedded systems, and large swaths of critical infrastructure code are either directly hand-coded in C or employ C as intermediate target of compilation or code synthesis. Analysis methods and verification tools that apply to C thus remain a vital area of research. The Verified Software Toolchain (VST) [4] is a semi-automated proof system for functional-correctness verification of C programs that integrates two long-standing lines of research: (i) program logics with machine-checked proofs of soundness; (ii) practical verification tools for industry-strength programming languages. VST consists of three main components:

**Verifiable C** [3] is a higher-order impredicative concurrent separation logic covering almost all the control-flow and data-structuring features of C (we currently omit goto and by-copy whole-struct assignment);

**VST-Floyd** [7] is a library of lemmas, definitions, and automation tactics that assist the user in applying the program logic to a program, using forward symbolic execution, with separation logic assertions as symbolic states;

To appear in FM2019: 23rd International Symposium on Formal Methods, October 2019. This copy has the full appendix; the conference proceedings version has an abbreviated appendix.

**The semantic model** justifies the proof rules, exploiting the theories of step-indexing, impredicative quantification, separation algebras, and concurrent ghost state. The semantic model is the basis of a machine-checked proof [4], in Coq, that the Verifiable C program logic is sound w.r.t. the operational semantics of CompCert Clight. Thus the user's Coq proof *in* Verifiable C composes with our soundness proof *of* Verifiable C and with Leroy's CompCert compiler correctness proof [15] to yield an end-to-end proof of the functional correctness of the assembly-language program.

VST's key feature—distinguishing it from tools such as VCC [8], Frama-C [11], or VeriFast [9]—is that it is *entirely* implemented in the Coq proof assistant. A user imports C code into the Coq development environment and applies VST-Floyd's automation—computational decision procedures from Coq's standard library, plus custom-built tactics for forward symbolic execution and entailment checking—to construct formal derivations in the Verifiable C program logic. The full power of Coq and its libraries are available to manipulate application-specific mathematics. The semantic validity of the proof rules—machine-checked by Coq's kernel—connects these derivations to Clight, i.e. CompCert's representation of parsed and determinized C code.

Recent applications of VST include the verification of cryptographic primitives from OpenSSL [2, 6] and mbedTLS [24], an asynchronous communication mechanism [17], and an internet-facing server component [13]. Ongoing efforts elsewhere include a generational garbage collector and a malloc-free library.

*Motivated by these applications,* we now add support for data abstraction, a key enabler of scalability. As shown in previous work [21], separation logic can easily express data abstraction, using abstract predicates: just as the client program of an abstract data type (ADT) can be written without knowing the representation, verification of the client can proceed without knowing the representation. In type theory, this is the principle of *existential types* [18].

But in real-life modular programming, the same function may want more than one specification. For example, a function may expose a concrete specification to "friend" functions that know the representation of internal data and a more abstract specification for clients that do not. In this case, one should not have to verify the function-body twice, once for each specification; instead, one should verify the function-body with respect to the concrete specification, then prove the concrete implies the abstract. Again, type theory provides an appropriate notion: *subtyping* [22]. In other cases, it may be desirable to specify different use cases of a function—applying, for example, to different input configurations, or to different control flow paths—using different specifications, perhaps using different abstract predicates. Yet again, type theory provides a useful analogue: *intersection types*, a form of ad-hoc polymorphism.

These observations motivate the use of type-theoretic principles as guidelines for developing specification mechanisms and automation features for abstraction. We now take a step in this direction, focusing primarily on the notion of subtyping. The observation that Hoare's original rule of consequence is insufficiently powerful in languages with (recursive) procedures motivated research into *parameter adaptation*, by (among others) Kleymann, Nipkow, and Naumann [12, 20, 19]. Indeed, Kleymann observed that ([12], page 9)

– *in proving that the postcondition has been weakened, one may also assume the precondition of the conclusion holds...*
– *one may adjust the auxiliary variables in the premise. Their value may depend on the value of auxiliary variables in the conclusion and the value of all program variables in the initial state.*

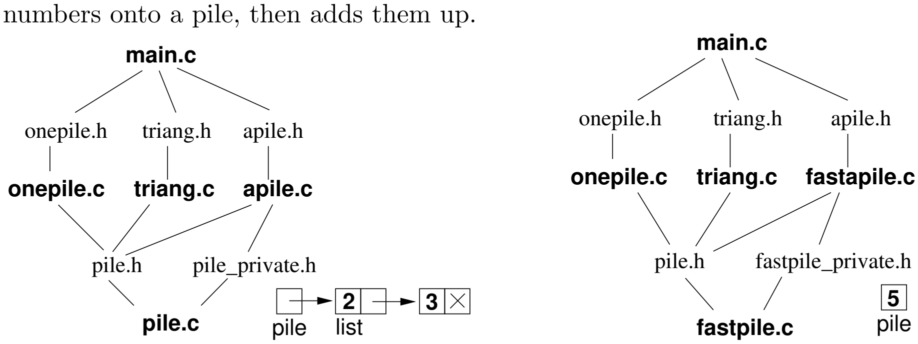
But these developments were carried out for small languages and predate the emergence of separation logic. The present article hence revisits these ideas in the context of VST, by developing a powerful notion of function-specification subtyping for higher-order impredicative separation logic. Our treatment improves on previous work in several regards:

– We support function-specifications of function pointers, as part of our support for almost the entire C language. Kleymann only considers a single (anonymous, parameterless, but possibly recursive) procedure, while Nipkow supports mutual recursion between named procedures.
– Our notion of subtyping avoids direct quantification over states, thus permitting a higher-order impredicative separation logic in the style of VST and Iris [10], where "assertion" must be an abstract type with a step-indexed model rather than simply $\mathsf{state}\to\mathsf{Prop}$. This is necessary to fully support function pointers and higher-order resource invariants (for concurrent programming). In contrast, Kleymann's and Nipkow's assertions are predicates over states, and the side conditions of their adaptation rules explicitly quantify over states. Naumann's formulation using predicate transformers captures the same relationship in a slighty more abstract manner.
– VST associates function specifications to globally named functions in its proof context $\Delta$ and includes a separation logic assertion func_at that attaches specifications to function-pointer values. Our treatment integrates subsumption coherently into proof contexts, func_at, and the soundness judgment. We support subsumption at function call sites but also incorporate subsumption in a notion of (proof) context subtyping that is reminiscent of record subtyping [22]. This will allow bundling function specifications into specifications of objects or modules that can be abstractly presented to client programs and are compatible with behavioral subtyping [16, 14, 23].
– We introduce intersection specifications and show that their interaction with subsumption precisely matches that of intersection types.

Our presentation is example-driven: we illustrate several use cases of subsumption on concrete code fragments in Verifiable C. Technical adaptations of the model that support these verifications have been machine-checked for soundness, but in the paper we only sketch them. The full Coq proofs of our example are in the VST repo, github.com/PrincetonUniversity/VST in directory progs/pile.

## 2 Function specifications in Verifiable C

Our main example is an abstract data type (ADT) for *piles*, simple collections of integers. Figure 1 (on the next page) shows a modular C program that throws numbers onto a pile, then adds them up.




The diagram at left shows that pile.c is imported by onepile.c (which manages a single pile), apile.c (which manages a single pile in a different way), and triang.c (which computes the $n$th triangular number). The latter three modules are imported by main.c. Onepile.c and triang.c import the abstract interface pile.h; apile.c imports also the low-level concrete interface pile_private.h that exposes the representation—a typical use case for this organization might be when apile.c implements representation-dependent debugging or performance monitoring.

When—as shown on the right—pile.c is replaced by a faster implementation fastpile.c (code in Figure 3) using a different data structure, apile.c must be replaced with fastapile.c, but the other modules need not be altered, *and neither should their specification or verification.*

Figure 2 presents the specification of the pile module, in the Verifiable C separation logic. Each C-language function identifier (such as _Pile_add) is bound to a funspec, a function specification in separation logic.

Before specifying the functions (with preconditions and postconditions), we must first specify the data structures they receive as arguments and return as results. Linked lists are specified as usual in separation logic: listrep is a recursive definition over the abstract ("mathematical") list value $\sigma$, specifying how it is laid out in a memory footprint rooted at address $p$. Then pilerep describes a memory location containing a pointer to a listrep.

A funspec takes the form, WITH $\vec{x} : \vec{\tau}$ PRE ... POST .... For example, take Pile_add_spec from Figure 2: the $\vec{x}$ are bound Coq variables visible in both the precondition and postcondition, in this case, $p$:val, $n$:Z, $\sigma$:list Z, $gv$:globals, where $p$ is the address of a pile data structure, $n$ is the number to be added to the pile, $\sigma$ is the sequence currently represented by the pile, and $gv$ is a way to access all named global variables. The PREcondition is parameterized by the C-language formal parameter names _p and _n. An assertion in Verifiable C takes the form, PROP(*propositions*)LOCAL(*variable bindings*)SEP(*spatial conjuncts*).

```
/* pile.h */
typedef struct pile *Pile;
Pile Pile_new(void);
void Pile_add(Pile p, int n);
int Pile_count(Pile p);
void Pile_free(Pile p);

/* onepile.h */
void Onepile_init(void);
void Onepile_add(int n);
int Onepile_count(void);

/* apile.h */
void Apile_add(int n);
int Apile_count(void);

/* triang.h */
int Triang_nth(int n);

/* triang.c */
#include "pile.h"
int Triang_nth(int n) {
  int i,c;
  Pile p = Pile_new();
  for (i=0; i<n; i++)
    Pile_add(p,i+1);
  c = Pile_count(p);
  Pile_free(p);
  return c;
}

/* onepile.c */
#include "pile.h"
Pile the_pile;
void Onepile_init(void)
 {the_pile = Pile_new();}
void Onepile_add(int n)
 {Pile_add(the_pile, n);}
int Onepile_count(void)
 {return Pile_count(the_pile);}
```

```
/* pile_private.h */
struct list {int n; struct list *next;};
struct pile {struct list *head;};

/* pile.c */
#include <stddef.h>
#include "stdlib.h"
#include "pile.h"
#include "pile_private.h"
Pile Pile_new(void) {
  Pile p = (Pile)surely_malloc(sizeof *p);
  p→head=NULL;
  return p;
}
void Pile_add(Pile p, int n) {
  struct list *head = (struct list *)
      surely_malloc(sizeof *head);
  head→n=n;
  head→next=p→head;
  p→head=head;
}
int Pile_count(Pile p) {
  struct list *q;
  int c=0;
  for(q=p→head; q; q=q→next)
    c += q→n;
  return c;
}
void Pile_free(Pile p) { . . . }

/* apile.c */
#include "pile.h"
#include "pile_private.h"
#include "apile.h"
struct pile a_pile = {NULL};
void Apile_add(int n)
  {Pile_add(&a_pile, n);}
int Apile_count(void)
  {return Pile_count(&a_pile);}
```

**Fig. 1.** The pile.h abstract data type has operations *new, add, count, free.* The triang.c client adds the integers 1–$n$ to the pile, then counts the pile. The pile.c implementation represents a pile as header node (struct pile) pointing to a linked list of integers. At bottom, there are two modules that each implement a single "implicit" pile in a module-local global variable: onepile.c maintains a pointer to a pile, while apile.c maintains a struct pile for which it needs knowledge of the representation through pile_private.h.

```
(* spec_pile.v *)
(* representation of linked lists in separation logic *)
Fixpoint listrep (σ: list Z) (x: val) : mpred :=
 match σ with
 | h::hs ⇒ EX y:val, !! (0 ≤ h ≤ Int.max_signed) &&
      data_at Ews tlist (Vint (Int.repr h), y) x
      ∗ malloc_token Ews tlist x ∗ listrep hs y
 | nil ⇒ !! (x = nullval) && emp
 end.

(* representation predicate for piles *)
Definition pilerep (σ: list Z) (p: val) : mpred :=
 EX x:val, data_at Ews tpile x p ∗ listrep σ x.

Definition pile_freeable (p: val) :=
  malloc_token Ews tpile p.

Definition Pile_new_spec :=
 DECLARE _Pile_new
 WITH gv: globals
 PRE [ ] PROP() LOCAL(gvars gv) SEP(mem_mgr gv)
 POST[ tptr tpile ]
   EX p: val,
     PROP() LOCAL(temp ret_temp p)
     SEP(pilerep nil p; pile_freeable p; mem_mgr gv).

Definition Pile_add_spec :=
 DECLARE _Pile_add
 WITH p: val, n: Z, σ: list Z, gv: globals
 PRE [ _p OF tptr tpile, _n OF tint ]
    PROP(0 ≤ n ≤ Int.max_signed)
    LOCAL(temp _p p; temp _n (Vint (Int.repr n));
            gvars gv)
    SEP(pilerep σ p; mem_mgr gv)
 POST[ tvoid ]
    PROP() LOCAL()
    SEP(pilerep (n::σ) p; mem_mgr gv).

Definition sumlist : list Z → Z := List.fold_right Z.add 0.

Definition Pile_count_spec :=
 DECLARE _Pile_count
 WITH p: val, σ: list Z
 PRE [ _p OF tptr tpile ]
    PROP(0 ≤ sumlist σ ≤ Int.max_signed) LOCAL(temp _p p)
    SEP(pilerep σ p)
 POST[ tint ]
    PROP() LOCAL(temp ret_temp (Vint (Int.repr (sumlist σ))))
    SEP(pilerep σ p).
```

**Notation key**

mpred  predicate on memory

EX  existential quantifier
!!  injects Prop into mpred
&&  nonseparating conjunction
data_at $\pi$ $\tau$ $v$ $p$  is  $p \mapsto v$, separation-logic mapsto at type $\tau$, permission $\pi$

malloc_token $\pi$ $\tau$ $x$  represents "capability to deallocate $x$"

Ews  the "extern write share" gives write permission

_Pile_new is a C identifier

WITH  quantifies variables over PRE/POST of funspec

The C function's return type, tptr tpile, is "pointer to **struct** pile"

PROP(...) are pure propositions on the WITH-variables

LOCAL(... temp _p $p$ ...) associates C local var _p with Coq value $p$

gvars $gv$  establishes $gv$ as mapping from C global vars to their addresses

SEP($R_1$; $R_2$)  are separating conjuncts $R_1 * R_2$

mem_mgr $gv$ represents *different* states of the malloc/free system in PRE and POST of any function that allocates or frees

**Fig. 2.** Specification of the pile module (Pile_free_spec not shown).

In this case the PROP asserts that $n$ is between 0 and max-int; LOCAL asserts[1] that address $p$ is the current value of C variable _p, integer $n$ is the value of C variable _n, and $gv$ is the global-variable access map. The precondition's SEP clause has two conjuncts: the first one says that there's a *pile* data structure at address $p$ representing sequence $\sigma$; the second one represents the memory-manager library. The spatial conjunct (mem_mgr $gv$) represents the private data structure of the memory-manager library, that is, the global variables in which the malloc-free system keeps its free lists.

The SEP clause of the POSTcondition says that the *pile* at address $p$ now represents the list $n{::}\sigma$, and that the memory manager is still there.

Verifying that pile.c's functions satisfy the specifications in Fig. 2 using VST-Floyd is done by proving Lemmas like this one (in file verif_pile.v):

**Lemma** body_Pile_new: semax_body Vprog Gprog f_Pile_new Pile_new_spec.
**Proof**. ... *(∗7 lines of Coq proof script∗)*.... **Qed**.

This says, in the context Vprog of global-variable types, in the context Gprog of function-specs (for functions that Pile_new might call), the function-body f_Pile_new satisfies the function-specification Pile_new_spec.

**Linking**

A modular proof of a modular program is organized as follows: CompCert parses each module M.c into the AST file M.v. Then we write the specification file spec_M.v containing funspecs as in Figure 2. We write verif_M.v which imports spec files of all the modules from which M.c calls functions, and contains semax_body proofs of correctness (such as body_Pile_new at the end of §2), for each of the functions in M.c.

What's special about the main() function is that its separation-logic precondition has all the initial values of the global variables, merged from the global variables of each module. In spec_main we merge the ASTs (global variables and function definitions) of all the M.v by a simple, computational, syntactic function. This is illustrated in the Coq files in VST/progs/pile.

VST's main soundness statement is that, when running main() in CompCert's operational semantics, in the initial memory induced from all global-variable initializers, the program is safe and correct—with a notion of partial correctness in interacting with the world via effectful external function calls [13] and returning the "right" value from main.

## 3 Subsumption of function specifications

We now turn to the replacement of pile.c by a more performant implementation, fastpile.c, and its specification—see Figure 3. As fastpile.c employs a different

[1] A LOCAL clause temp _p $p$ asserts that the current value of C local variable _p is the Coq value $p$. If $n$ is a mathematical integer, then Int.repr $n$ is its projection into 32-bit machine integers, and Vint projects machine integers into the type of scalar C-language values.

```
/* fastpile_private.h */
struct pile { int sum; };

/* fastpile.c */
#include ...
#include "pile.h"
#include "fastpile_private.h"
Pile Pile_new(void)
  {Pile p = (Pile)surely_malloc(sizeof *p); p→sum=0; return p; }
void Pile_add(Pile p, int n)
  {int s = p→sum; if (0≤ n && n≤ INT_MAX-s) p→sum = s+n; }
int Pile_count(Pile p) {return p→sum;}
void Pile_free(Pile p) {free(p);}

(* spec_fastpile.v *)
Definition pilerep (σ: list Z) (p: val) : mpred :=
 EX s:Z, !! (0 ≤ s ≤ Int.max_signed ∧ Forall (Z.le 0) σ ∧
             (0 ≤ sumlist σ ≤ Int.max_signed → s=sumlist σ))
   && data_at Ews tpile (Vint (Int.repr s)) p.

Definition pile_freeable := (* looks identical to the one in fig.2 *)
Definition Pile_new_spec := (* looks identical to the one in fig.2 *)
Definition Pile_add_spec := (* looks identical to the one in fig.2 *)
Definition Pile_count_spec := (* looks identical to the one in fig.2 *)
```

**Fig. 3.** fastpile.c, a more efficient implementation of the pile ADT. Since the only query function is count, there's no need to represent the entire list, just the sum will suffice. In the verification of a client program, the pilerep separation-logic predicate has the same signature: list Z $\rightarrow$ val $\rightarrow$ mpred, even though the representation is a single number rather than a linked list.

data representation than pile.c, its specification employs a different representation predicate pilerep. As pilerep's type remains unchanged, the function specifications look virtually identical[2]; however, the VST-Floyd proof scripts (in file verif_fastpile.v) necessarily differ. Clients importing only the pile.h interface, like onepile.c or triang.c, cannot tell the difference (except that things run faster and take less memory), and are specified and verified only once (files spec_onepile.v / verif_onepile.v and spec_triang.v / verif_triang.v).

But we may also equip fastpile.c with a more low-level specification (see Figure 4) in which the function specifications refer to a different representation predicate, countrep. In reasoning about clients of this low-level interface, we do not need a notion of of "sequence"—in contrast to pilerep in Fig. 3. The new specification is less abstract than the one in Fig. 3, and closer to the implemen-

[2] Existentially abstracting over the internal representation predicates would further emphasize the uniformity between fastpile.c and pile.c—a detailed treatment of this is beyond the scope of the present article.

```
(* spec_fastpile_concrete.v *)
Definition countrep (s: Z) (p: val) : mpred := EX s':Z,
  !! (0 ≤ s ∧ 0 ≤ s' ≤ Int.max_signed ∧ (s ≤ Int.max_signed → s'=s)) &&
  data_at Ews tpile (Vint (Int.repr s')) p.

Definition count_freeable (p: val) := malloc_token Ews tpile p.

Definition Pile_new_spec := ...

Definition Pile_add_spec :=
 DECLARE _Pile_add
 WITH p: val, n: Z, s: Z, gv: globals
 PRE [ _p OF tptr tpile, _n OF tint ]
    PROP(0 ≤ n ≤ Int.max_signed)
    LOCAL(temp _p p; temp _n (Vint (Int.repr n)); gvars gv)
    SEP(countrep s p; mem_mgr gv)
 POST[ tvoid ]
    PROP() LOCAL() SEP(countrep (n + s) p; mem_mgr gv).

Definition Pile_count_spec := ...
```

**Fig. 4.** The fastpile.c implementation could be used in applications that simply need to keep a running total. That is, a *concrete* specification can use a predicate countrep: Z → val → mpred that makes no assumption about a sequence (list Z). In countrep, the variable $s'$ and the inequalities are needed to account for the possibility of integer overflow.

tation. The subsumption rule (to be introduced shortly) allows us to exploit this relationship: we only need to explicitly verify the code against the low-level specification and can establish satisfaction of the high-level specification by recourse to subsumption. This separation of concerns extends from VST specifications to model-level reasoning: for example, in our verification of cryptographic primitives we found it convenient to verify that the C program implements a *low-level functional model* and then separately prove that the low-level functional model implements a high-level specification (e.g. cryptographic security).[3] In our running example, fastpile.c's low-level functional model is *integer* (the Coq Z type), and its high level specification is list Z.

[3] For example: in our proof of HMAC-DRBG [24], before VST had function-spec subsumption, we had two different proofs of the function f_mbedtls_hmac_drbg_seed, one with respect to a more concrete specification drbg_seed_inst256_spec and one with respect to a more abstract specification drbg_seed_inst256_spec_abs. The latter proof was 202 lines of Coq, at line 37 of VST/hmacdrbg/-drbg_protocol_proofs.v in commit 3e61d2991e3d70f5935ae69c88d7172cf639b9bc of https://github.com/PrincetonUniversity/VST. Now, instead of reproving the function-body a second time, we have a funspec_sub proof that is only 60 lines of Coq (at line 42 of the same file in commit c2fc3d830e15f4c70bc45376632c2323743858ef).

To formally state the desired subsumption lemma, observe that notation like DECLARE _Pile_add WITH ... PRE ... POST ... is merely VST's syntactic sugar for a pair that ties the identifier _Pile_add to the funspec WITH...PRE...POST. For _Pile_add we have two such specifications,

```
spec_fastpile.Pile_add_spec: ident*funspec                (* in Figure 3 *)
spec_fastpile_concrete.Pile_add_spec: ident*funspec       (* in Figure 4 *)
```

and our notion of *funspec subtyping* will satisfy the following lemma

```
Lemma sub_Pile_add: funspec_sub (snd spec_fastpile_concrete.Pile_add_spec)
                                (snd spec_fastpile.Pile_add_spec).
```

and similarly for Pile_new and Pile_count. Specifically, we permit related specifications to have different WITH-lists, in line with Kleymann's adaptation-complete rule of consequence

$$\frac{\vdash \{P'\}c\{Q'\}}{\vdash \{P\}c\{Q\}} \forall Z.\, \forall \sigma.\, P Z \sigma \to \forall \tau.\, \exists Z'.(P' Z' \sigma \wedge (Q' Z' \tau \to Q Z \tau))$$

where assertions are binary predicates over auxiliary and ordinary states, and $Z, Z'$ are the WITH values.[4]

Our subsumption applies to function specifications, not arbitrary statements $c$. In the rule for function calls, it ensures that a concretely specified function can be invoked where callers expect an abstractly specified one, just like the subsumption rule of type theory: $\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau}$. It is also reflexive and transitive.

*Support for framing* An important principle of separation logic is the frame rule:

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{P * R\}} (\text{modifiedvars}(c) \cap \text{freevars}(R) = \emptyset)$$

We have found it useful to explicitly incorporate framing in funspec_sub, because abstract specifications may have useless data. Consider a function that performs some action (e.g., increment a variable) using some auxiliary data (e.g., an array of 10 integers):

```
int incr1(int i, unsigned int *auxdata)   { auxdata[i%10] += 1; return i+1; }
```

The function specification makes clear that the private contents of the auxdata is, from the client's point of view, unconstrained; the implementation is free to store anything in this array:

[4] We give Kleymann's rule for total correctness here. VST is a logic for partial correctness, but its preconditions also guarantee safety; Kleymann's partial-correctness adaptation rule cannot guarantee safety.

```
Definition incr1_spec := DECLARE _incr1
WITH i: Z, a: val, π: share, private: list val
 PRE [ _i OF tint, _auxdata OF tptr tuint ]
    PROP (0 ≤ i < Int.max_signed; writable_share π)
    LOCAL(temp _i (Vint (Int.repr i)); temp _auxdata a)
    SEP(data_at sh (tarray tuint 10) private a)
 POST [ tint ]
   EX private': list val, PROP() LOCAL(temp ret_temp (Vint (Int.repr (i+1))))
                          SEP(data_at π (tarray tuint 10) private' a).
```

You might think the auxdata is useless, but (i) real-life interfaces often have useless or vestigial fields; and (ii) this might be where the implementation keeps profiling statistics, memoization, or other algorithmically useful information.

Here is a different implementation that should serve any client just as well:

```
int incr2(int i, unsigned int *auxdata)    { return i+1; }
```

Its natural specification has an empty SEP clause:

```
Definition incr2_spec := DECLARE _incr2
 WITH i: Z
 PRE [ _i OF tint, _auxdata OF tptr tuint ]
    PROP (0 ≤ i < Int.max_signed)   LOCAL(temp _i (Vint (Int.repr i)))   SEP()
 POST [ tint ]
    PROP()   LOCAL(temp ret_temp (Vint (Int.repr (i + 1))))   SEP().
```

The *formal* statement that incr2 serves any client just as well as incr1 is another case of subsumption:

**Lemma** sub_incr12: funspec_sub (snd incr2_spec) (snd incr1_spec).

In the proof, we use (data_at $\pi$ (tarray tuint 10) *private* a) as the *frame*.

If the auxdata is a global variable instead of a function parameter, all the same principles apply:

```
int global_auxdata[10];
int incr3(int i)    { global_auxdata[i%10] += 1; return i+1; }
int incr4(int i)    { return i+1; }
```

We define a funspec for incr3 whose SEP clause mentions the auxdata, we define a funspec for incr4 whose SEP clause is empty, and we can prove,

**Lemma** sub_incr34: funspec_sub (snd incr4_spec) (snd incr3_spec).

For another example of framing, consider again Figure 2, the specification of pilerep, pile_freeable, Pile_new_spec, etc. One might think to combine pile_freeable (the memory-deallocation capability) with pile_rep (capability to modify the contents) yielding a single combined predicate pilerep'. That way, proofs of client programs would not have to manage two separate conjuncts.

That would work for clients such as triang.c and onepile.c, but not for apile.c which has an initialized global variable (a_pile) that satisfies pilerep but *not* pile_freeable (since it was not obtained from the malloc-free system). Furthermore, the specifications of pile_add and pile_count do not mention pile_freeable in their pre- or postconditions, since they have no need for this capability.

By using funspec_sub (with its framing feature), we can have it both ways. One can easily make a more abstract spec in which the funspecs of pile_new, pile_add, pile_count, pile_free all take pilerep' in their pre- and postconditions; onepile and triang will still be verifiable using these specs. But in proving funspec_sub, therefore, specifications for pile_add and pile_count now *do* implicitly take pile_freeable in their pre- and postconditions, even though they have no use for it; this is the essence of the frame rule.

## 4 Definitions of funspec subtyping

Except in certain higher-order cases, we use this notion of function specification:

NDmk_funspec (f: funsig) (cc: calling_convention)
    ($A$: Type) (Pre Post: $A \to \mathsf{environ} \to \mathsf{mpred}$): funspec.

To construct a *nondependent* (ND) function spec, one gives the function's C-language type signature (funsig), the calling convention (usually cc=cc_default), the precondition, and the postcondition. $A$ gives the type of variable (or tuple of variables) "shared" between the precondition and postcondition. Pre and Post are each applied to the shared value of type $A$, then to a local-variable environment (of type environ) containing the formal parameters or result-value (respectively), finally yielding an mpred, a spatial predicate on memory.

For example, to specify an increment function with formal parameter _p pointing to an integer in memory, we let $A = \mathsf{int}$, so that

$$\mathsf{Pre} = \lambda i : A.\, \lambda \rho.\, \rho(\_\mathsf{p}) \mapsto i \quad \text{and} \quad \mathsf{Post} = \lambda i : A.\, \lambda \rho.\, \rho(\_\mathsf{p}) \mapsto (i+1).$$

This form suffices for most C programming. But sometimes in the presence of higher-order functions, one wants impredicativity: $A$ may be a tuple of types that includes the type mpred. If this is done naively, it cannot typecheck in CiC (there will be universe inconsistencies); see the Appendix.

*General funspec.* Higher-order function specs are (mostly) beyond the scope of this paper. When precondition and postcondition must predicate over predicates, we must ensure that each is a *bifunctor*, that is, we must keep track of *covariant* and *contravariant* occurrences, and so on. This approach was outlined by America and Rutten [1] and has been implemented both in Iris [10] and VST.[5]

VST's most general form of function spec is,

```
Inductive funspec :=
  mk_funspec: forall (f: funsig) (cc: calling_convention) (A: TypeTree)
    (P Q: forall ts, dependent_type_functor_rec ts (AssertTT A) mpred)
    (P_ne: super_non_expansive P) (Q_ne: super_non_expansive Q), funspec.
```

[5] Bifunctor function-specs in VST were the work of Qinxiang Cao, Robert Dockins, and Aquinas Hobor.

Here, super_non_expansive is a proof that the precondition (or postcondition) is a nonexpansive (in the step-indexing sense) bifunctor; see the Appendix. The *nondependent* (ND) form of mk_funspec shown above is simply a derived form of dependent mk_funspec.

*Too-special funspec subtyping.* Let's consider the obvious notion of funspec subtyping: $\phi_1$ is a subtype of $\phi_2$ if the precondition of $\phi_2$ entails the precondition of $\phi_1$, and the postcondition of $\phi_1$ entails the postcondition of $\phi_2$.

**Definition** far_too_special_NDfunspec_sub ($f_1$ $f_2$ : funspec) :=
 **let** $\Delta$ := funsig_tycontext (funsig_**of**_funspec $f_1$) **in**
 **match** $f_1$, $f_2$ **with**
  NDmk_funspec $fsig_1$ $cc_1$ $A_1$ $P_1$ $Q_1$ , NDmk_funspec $fsig_2$ $cc_2$ $A_2$ $P_2$ $Q_2$ $\Rightarrow$
   $fsig_1 = fsig_2 \wedge cc_1 = cc_2 \wedge A_1{=}A_2 \wedge$ ($\forall x : A_1$, $\Delta$, $P_2$ nil $x \vdash P_1$ nil $x$) $\wedge$
   ($\forall x : A_1$, (ret0_tycon $\Delta$), $Q_1$ nil $x \vdash Q_2$ nil $x$)
 **end**.

We write $\Delta$, $P_2$ nil $x \vdash P_1$ nil $x$, where $P_1$ and $P_2$ are the preconditions of $f_1$ and $f_2$, nil expresses that these are nondependent funspecs (no bifunctor structure), and $x$ is the value shared between precondition and postcondition. The type-context $\Delta$ provides the additional guarantee that the formal parameters are well typed, and ret0_tycon $\Delta$ guarantees that the return-value is well typed.

This notion of funspec-sub is sound (w.r.t. subsumption), but barely useful: (1) it requires that the witness types of the two funspecs be the same ($A_1{=}A_2$), (2) it doesn't support framing, and (3) it requires $Q_1 \vdash Q_2$ even when $P_2$ is not satisfied. *Each* of these omissions prevents the practical use of funspec-sub in real verifications, but only (1) and (3) were addressed in previous work [12, 20].

*Useful, ordinary funspec subtyping.* If NDmk_funspec were a constructor, we could define,

**Definition** NDfunspec_sub ($f_1$ $f_2$ : funspec) :=
 **let** $\Delta$ := funsig_tycontext (funsig_**of**_funspec $f_1$) **in**
 **match** $f_1$, $f_2$ **with**
  NDmk_funspec $fsig_1$ $cc_1$ $A_1$ $P_1$ $Q_1$, NDmk_funspec $fsig_2$ $cc_2$ $A_2$ $P_2$ $Q_2$ $\Rightarrow$
   $fsig_1 = fsig_2 \wedge cc_1 = cc_2 \wedge$
   $\forall x_2 : A_2$,
    $\Delta$, $P_2$ nil $x_2$ $\vdash$
     EX $x_1{:}A_1$, EX $F$:mpred, ((($\lambda\rho.F$) $*$ $P_1$ nil $x_1$) &&
       !! ((ret0_tycon $\Delta$), ($\lambda\rho.F$) $*$ $Q_1$ nil $x_1 \vdash Q_2$ nil $x_2$))
 **end**.

Here, each of the three deficiencies is remedied: the witness value $x_1 : A_1$ is existentially derived from $x_2 : A_2$, the frame $F$ is existentially quantified, and the entailment $Q_1 \vdash Q_2$ is conditioned on the precondition $P_2$ being satisfied.

This version of funspec-sub is, we believe, fully general for NDmk_funspec, that is, for function specifications whose witness types $A$ do not contain (covariant or contravariant) occurrences of mpred. We present the general, dependent funspec-sub in the Appendix, with its constructor mk_funspec, and show the construction of NDmk_funspec as a derived form. And actually, since NDmk_funspec is not really a constructor (it is a function that applies the constructor mk_funspec), we must define NDfunspec_sub as a pattern-match on mk_funspec; see the Appendix.

## 5 The subsumption rules

The purpose of funspec_sub is to support subsumption rules.

Our Hoare-logic judgment takes the form $\Delta \vdash \{P\}c\{Q\}$ where the context $\Delta$ describes the types of local and global variables and the funspecs of global functions. We say $\Delta <: \Delta'$ if $\Delta$ is at least as strong as $\Delta'$; in Verifiable C this is written tycontext_sub $\Delta$ $\Delta'$. Again, this relation is reflexive and transitive.

**Definition** *(glob_specs):* If $i$ is a global identifier, write (glob_specs $\Delta$)!i to be the option(funspec) that is either None or Some $\phi$.

**Lemma** *funspec_sub_tycontext_sub:* **Suppose** $\Delta$ agrees with $\Delta'$ on types attributed to global variables, types attributed to local variables, current function return type (if any), and differs only in *specifications* attributed to global functions, in particular: For every global identifier $i$, if (glob_specs $\Delta$)!i=Some $\phi$ then (glob_specs $\Delta'$)!i=Some $\phi'$ and funspec_sub $\phi$ $\phi'$. **Then** $\Delta <: \Delta'$.

*Proof.* Trivial from the definition of $\Delta <: \Delta'$.

**Theorem** *(semax_Delta_subsumption):*

$$\frac{\Delta <: \Delta' \qquad \Delta' \vdash \{P\}c\{Q\}}{\Delta \vdash \{P\}c\{Q\}}$$

*Proof.* Nontrivial. Because this is a logic of higher-order recursive function pointers, our Coq proof[6] in the modal step-indexed model uses the Löb rule to handle recursion, and unfolds our rather complicated semantic definition of the Hoare triple [4].

But this is not the only subsumption rule we desire. Because C has function-pointers, the general Hoare-logic function-call rule is for $\Delta \vdash \{P\}e_f(e_1,\ldots,e_n)\{Q\}$ where $e_f$ is an expression that evaluates to a function-pointer. Therefore, we cannot simply look up $e_f$ as a global identifier in $\Delta$. Instead, the precondition $P$ must associate the value of $e_f$ with a funspec. Without subsumption, the rules are:

$$\frac{\begin{array}{c}(\mathsf{glob\_specs}\ \Delta)!f = \mathsf{Some}\ \phi\\ \Delta \vdash f \Downarrow v\\ \Delta \vdash \{\mathsf{func\_ptr}\, v\, \phi\ \wedge\ P\}c\{Q\}\end{array}}{\Delta \vdash \{P\}c\{Q\}} \qquad \frac{\begin{array}{c}\Delta \vdash e_f \Downarrow v\\ \Delta \vdash e_1 \Downarrow v_1 \ldots \Delta \vdash e_n \Downarrow v_n\\ P * F \vdash \mathsf{func\_ptr}\, v\, \phi\\ \phi(w) = \{P\}\{Q\}\end{array}}{\Delta \vdash \{P * F\}e_f(e_1, e_2, \ldots, e_n)\{Q * F\}}$$

[6] See file veric/semax_lemmas.v in the VST repo.

The rule semax_fun_id at left says, if the global context $\Delta$ associates identifier $f$ with funspec $\phi$, and if $f$ evaluates to the address $v$, then for the purposes of proving $\{P\}c\{Q\}$ we can assume the stronger precondition in which address $v$ has the funspec $\phi$.

The semax_call rule says, if $e_f$ evaluates to address $v$, and the precondition factors into conjuncts $P*F$ that imply address $v$ has the funspec $\phi$, then choose a witness $w$ (for the WITH clause), instantiate the witness of $\phi$ with $w$, and match the precondition and postcondition of $\phi(w)$ with $P$ and $Q$; then the function-call is proved. (Functions can return results, but we don't show that here.)

To turn semax_call into a rule that supports subsumption, we simply replace the hypothesis $\phi(w) = \{P\}\{Q\}$ with $\phi <: \phi' \ \wedge \ \phi'(w) = \{P\}\{Q\}$.

To reconcile semax_Delta_subsumption and semax_fun_id, we build $<:$ into the definition of the predicate func_ptr $v$ $\phi$, i.e. permit $\phi$ to be more abstract than the specification associated with address $v$ in VST's underlying semantic model ("rmap").

## 6 Intersection specifications

In some of our verification examples, we found it useful to separate different use cases of a function into separate function specifications. One can easily do this using a pattern that discriminates on a boolean value from the WITH list jointly in the pre- and postcondition:

WITH $b : bool, \vec{x} : \vec{\tau}$ PRE `if` $b$ `then` $P_1$ `else` $P_2$ POST `if` $b$ `then` $Q_1$ `else` $Q_2$.

To attach different WITH-lists to different cases, we may use Coq's sum type to define a type such as Variant $T$ := case1: int | case2: string. and use it in a specification

WITH $\vec{x} : \vec{\tau}, t : T, \vec{y} : \vec{\sigma}$
PRE $[\ldots]$ **match** $t$ **with** case1 i $\Rightarrow P_1(\vec{x}, i, \vec{y})$ | case2 s $\Rightarrow P_2(\vec{x}, s, \vec{y})$ **end**
POST $[\ldots]$ **match** $t$ **with** case1 i $\Rightarrow Q_1(\vec{x}, i, \vec{y})$ | case2 s $\Rightarrow Q_2(\vec{x}, s, \vec{y})$ **end**.

which amounts to the *intersection* of
WITH $\vec{x} : \vec{\tau}, i : int, \vec{y} : \vec{\sigma}$ PRE $[\ldots]$ $P_1(\vec{x}, i, \vec{y})$ POST $[\ldots]$ $Q_1(\vec{x}, i, \vec{y})$ and
WITH $\vec{x} : \vec{\tau}, s : string, \vec{y} : \vec{\sigma}$ PRE $[\ldots]$ $P_2(\vec{x}, i, \vec{y})$ POST $[\ldots]$ $Q_2(\vec{x}, i, \vec{y})$.

Generalizing to arbitrary index sets, we may—for a given function signature and calling convention—combine specifications into specification *families*. (We show the nondependent (ND) case; the Coq proofs cover the general case.)

**Definition** funspec_Pi_ND sig cc (I:Type) (A : I → Type)
(Pre Post: forall i, A i → environ → mpred): funspec := ...

In previous work [5] we showed how relational (2-execution) specifications can be encoded as unary VDM-style specifications. Intersection specifications internalize VDM's "sets of specifications" feature.

The interaction between this construction and subtyping follows precisely that of intersection types in type theory: the lemmas

**Lemma** funspec_Pi_ND_sub: forall fsig cc I A Pre Post i,
funspec_sub (funspec_Pi_ND fsig cc I A Pre Post)
(NDmk_funspec fsig cc (A i) (Pre i) (Post i)).

**Lemma** funspec_Pi_ND_sub3: forall fsig cc I A Pre Post g (i:I)
(HI: forall i, funspec_sub g (NDmk_funspec fsig cc (A i) (Pre i) (Post i))),
funspec_sub g (funspec_Pi_ND fsig cc I A Pre Post).

are counterparts of the typing rules $\wedge_{j \in I} \tau_j <: \tau_i$ (for all $i \in I$) and $\dfrac{\forall i,\ \sigma <: \tau_i}{\sigma <: \wedge_{i \in I} \tau_i}$, the specializations of which to the binary case appear on page 206 of TAPL [22]. We expect these rules to be helpful for formalizing Leavens and Naumann's treatment of specification inheritance in object-oriented programs [14].

## 7 Conclusion

Even without funspec subtyping, separation logic easily expresses data abstraction [21]. But real-world code is modular (as in our running example) and reconfigurable (as in the substitution of fastpile.c for pile.c). Therefore a notion of specification re-abstraction is needed. We have demonstrated how to extend Kleymann's notion from commands to functions, and from first-order Hoare logic to higher-order separation logic with framing. We have a full soundness proof for the extended program logic, in Coq. Our funspec_sub integrates nicely with our existing proof automation tools and our existing methods of verifying individual modules. As a bonus, one's intuition that function-specs are like the "types" of functions is borne out by our theorems relating funspec_sub to intersection types.

*Future work:* When a client module respects data abstraction, such as onepile.c and triang.c in our example, its Coq proof script does not vary if the implementation of the abstraction changes (such as changing pile.c to fastpile.c). But in our current proofs of the running example, the proof scripts need to be rerun on the changed definition of pilerep. As footnote 2 suggests, this could be avoided by the use of existential quantification, in Coq, to describe data abstraction at the C module level.

## Appendix: Fully general funspec_sub

[The FM'19 conference-proceedings version of this paper is identical except that this appendix is abbreviated.]

NDfunspec_sub as introduced in Section 4 specializes the "real" subtype relation $\phi <: \psi$ in two regards: first, it only applies if $\phi$ and $\psi$ are of the NDfunspec form, i.e. the types of their WITH-lists ("witnesses") are trivial bifunctors as they do not contain co- or contravariant occurrences of mpred. Second, it fails to exploit step-indexing and is hence unnecessarily strong. Our full definition is as follows (Definition funspec_sub_si in veric/seplog.v):

```
Definition funspec_sub_si (f1 f2 : funspec):mpred :=
let Δ := funsig_tycontext (funsig_of_funspec f1) in
match f1, f2 with
 mk_funspec fsig1 cc1 A1 P1 Q1 _ _, mk_funspec fsig2 cc2 A2 P2 Q2 _ _ ⇒
   !!(fsig1 = fsig2 ∧ cc1 =cc2) &&
     ! (ALL ts2:list Type, ALL x2: F A2, ALL ρ:environ,
        (local (tc_environ Δ) ρ && P2 ts2 x2 ρ)
        ⤃ EX ts1:list Type, EX x1: F A1, EX F:mpred, (F ∗ P1 ts1 x1 ρ) &&
               ALL ρ':environ,
                  !( (local (tc_environ (ret0_tycon Δ)) ρ' && F ∗ Q1 ts1 x1 ρ')
                    ⤃ Q2 ts2 x2 ρ'))
end.
```

We first note that funspec_sub_si is not a (Coq) Proposition but an mpred – indeed, step-indexing has nothing interesting to say about pure propositions! That is, $P \vdash Q$ means, "for all resource-maps $s$, $P\,s$ implies $Q\,s$," but this can be too strong: $P \Rrightarrow Q$ means, "for all resource-maps $s$ whose step-index is $\leq$ the current 'age', $P\,s$ implies $Q\,s$." Recursive equations of mpreds, of the kind that come up in object-oriented patterns, can tolerate $\Rrightarrow$ where they cannot tolerate $\vdash$ [4, Chapter 17].

Second, both funspecs are constructors (mk_funspec $fsig$ $cc$ $A$ $P$ $Q$ _ _) as discussed in Section 4, but the two final arguments (the proofs that $P$ and $Q$ are super-non-expansive) are irrelevant for the remainder of the definition and hence anonymous. We also abbreviate operator dependent_type_functor_rec with $\mathcal{F}$.

Third, the definition makes use of the following operators (details on the penultimate two operators can be found in [4], Chapter 16):

| | |
|---|---|
| !! | inject a Coq proposition into VST's type mpred |
| && | (logical) conjunction of mpreds |
| ALL | universal quantification lifted to mpred |
| EX | existential quantification lifted to mpred |
| ! | "unfash" |
| $\Rrightarrow$ | "fashionable implication" |

In particular, the satisfaction of $P_2$ implies, only with the "precision" (in the step-indexed sense) at which $P_2$ is satisfied, that $Q_1$ implies $Q_2$

It is straightforward to prove that funspec_sub_si is reflexive, transitive, and specializes to NDfunspec_sub. To obtain soundness of context subtyping (semax_Delta_subsumption), we Kripke-extend the previous definition of VST's main semantic judgment semax. We also refined the definition of the predicate func_ptr: a stronger version of rule semax_fun_id permits the exposed specification $f$ to be a (step-indexed) abstraction of the specification $g$ stored in VST's resource-instrumented model:

```
Definition func_ptr_si f (v: val): mpred := EX b: block,
  !! (v = Vptr b Ptrofs.zero) && (EX g:_, funspec_sub_si g f && func_at g (b, 0)).
```

As func_at refers to the memory, this notion is again an mpred. Again, users who don't have complex object-oriented recursion patterns can avoid the step-indexing by using this non-step-indexed variant,

**Definition** func_ptr f (v: val): mpred := EX b: block,
!! (v = Vptr b Ptrofs.zero) && (EX g:_, !!(funspec_sub g f) && func_at g (b, 0)).

as the following lemma shows:

**Lemma** func_ptr_fun_ptr_si f v: func_ptr f v ⊢ func_ptr_si f v.

As one might expect, both notions are compatible with further subsumption:

**Lemma** func_ptr_si_mono fs gs v:
funspec_sub_si f g && func_ptr_si f v ⊢ func_ptr_si g v.

**Lemma** func_ptr_mono fs gs v: funspec_sub f gs → (func_ptr f v ⊢ func_ptr g v).

With these modifications and auxiliary lemmas in place, we have formally reestablished the soundness proof of VST's proof rules, justifying all rules given in this paper.

## References

1. Pierre America and Jan Rutten. Solving reflexive domain equations in a category of complete metric spaces. *Journal of Computer and System Sciences*, 39(3):343–375, 1989.
2. Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. on Programming Languages and Systems*, 37(2):7:1–7:31, April 2015.
3. Andrew W. Appel, Lennart Beringer, Qinxhiang Cao, and Josiah Dodds. Verifiable C: applying the Verified Software Toolchain to C programs. `https://vst.cs.princeton.edu/download/VC.pdf`, 2019.
4. Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge, 2014.
5. Lennart Beringer. Relational decomposition. In *Interactive Theorem Proving (LNCS 6898)*, pages 39–54, Berlin, 2011. Springer.
6. Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium*, pages 207–221. USENIX Assocation, August 2015.
7. Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reasoning*, 61(1-4):367–422, 2018.
8. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

9. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011.
10. Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.
11. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.
12. Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Asp. Comput.*, 11(5):541–566, 1999.
13. Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C Pierce, and Steve Zdancewic. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 234–248. ACM, 2019.
14. Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Trans. on Programming Languages and Systems*, 37(4):13:1–13:88, 2015.
15. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
16. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
17. William Mansky, Andrew W. Appel, and Aleksey Nogin. A verified messaging system. In *Proceedings of the 2017 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '17. ACM, 2017.
18. John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, July 1988.
19. David A. Naumann. Deriving sharp rules of adaptation for Hoare logics. Technical Report 9906, Department of Computer Science, Stevens Institute of Technology, 1999.
20. Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In Julian C. Bradfield, editor, *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Proceedings*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2002.
21. Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 247–258, 2005.
22. Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, Mass., 2002.
23. Cees Pierik and Frank S. de Boer. A proof outline logic for object-oriented programming. *Theor. Comput. Sci.*, 343(3):413–442, 2005.
24. Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17).* ACM, 2017.