

## Abstraction and Subsumption in Modular Verification of C Programs

Lennart Beringer · Andrew W. Appel

September 16, 2020

**Abstract** The type-theoretic notions of existential abstraction, subtyping, subsumption, and intersection have useful analogues in separation-logic proofs of imperative programs. We have implemented these as an enhancement of the Verified Software Toolchain (VST).

VST is an impredicative concurrent separation logic for the C language, implemented in the Coq proof assistant, and proved sound in Coq. For machine-checked functional-correctness verification of software at scale, VST embeds its expressive program logic in dependently typed higher-order logic (CiC). Specifications and proofs in the program logic can leverage the expressiveness of CiC—so users can overcome the abstraction gaps that stand in the way of top-to-bottom verification: gaps between source code verification, compilation, and domain-specific reasoning, and between different analysis techniques or formalisms.

Until now, VST has supported the specification of a program as a flat collection of function specifications (in higher-order separation logic)—one proves that each function correctly implements its specification, assuming the specifications of the functions it calls. But what if a function has more than one specification?

In this work, we exploit type-theoretic concepts to structure specification interfaces for C code. This brings modularity principles of modern software engineering to concrete program verification. Previous work used representation predicates to enable *data abstraction* in separation logic. We go further, introducing *function-specification subsumption* and *intersection specifications* to organize the multiple specifications that a function is typically associated with. As in type theory, if  $\phi$  is a `funspec_sub` of  $\psi$ , that is  $\phi <: \psi$ , then  $x : \phi$

---

This work was funded by the National Science Foundation under the awards 1005849 (*Verified High Performance Data Structure Implementations*, Beringer) and 1521602 *Expedition in Computing: The Science of Deep Specification*, Appel).

Department of Computer Science, 35 Olden St, Princeton University, Princeton, NJ 08540  
E-mail: eberinge@cs.princeton.edu, appel@princeton.edu

implies  $x : \psi$ , meaning that any function satisfying specification  $\phi$  can be used wherever a function satisfying  $\psi$  is demanded. Subsumption incorporates separation-logic framing and parameter adaptation, as well as step-indexing and specifications constructed via mixed-variance functors (needed for C’s function pointers).

**Keywords** Foundational Program Verification · Separation Logics · Specification Subsumption · Intersection specifications

## 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 written in C, or employ C as intermediate target of compilation or code synthesis. Analysis methods and verification tools for C remain a vital area of research.

Given the complexity of such code bases and C’s flexibility in organizing code into multiple layers and libraries, verification tools must scale both algorithmically and logically. *Algorithmic* scalability means reduced specification overhead and increased automation (less manual interactivity) in verifying individual function bodies. Verification tools based on SAT/SMT do this well. Tools based on interactive proof assistants have achieved scalability using tactic-based and reflectional automation [10], although the performance still lags behind SMT-based tools.

*Logical* scalability means *modular verification of modular programs*: the ability to compose verified code units either horizontally or vertically in accordance with the principles of modern software engineering, and to link them to verified compilers or domain-specific reasoning formalisms. Compared to SMT-based tools, proof-assistant-based tools can use more expressive (higher-order, or dependently typed) logics—which can be accessed in the program logic’s assertion language, allowing more accurate modeling of the C program’s functional correctness. And those expressive logics allow us to reason *about* the program logic—that is, prove it sound from first principles—within the proof assistant, for end-to-end machine-checkable assurance arguments.

This article focuses on aspects of logical scalability that allow modular reasoning about modular programs: abstraction, subsumption, and intersection of function specifications for C. We demonstrate that these concepts can be formulated in a way that mirrors the type-theoretic formulations of (existential) abstraction, subtyping, and intersection types, in the concrete setting of the Verified Software Toolchain (VST) [5].

*Background.* VST 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 industrial strength programming languages. VST consists of three main components:

**Verifiable C** [4] 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** [10] is a library of lemmas, definitions, and automation tactics that help the user apply the program logic to a program, using forward symbolic execution, with separation logic assertions as symbolic states;

**The semantic model** [5] 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 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 [24] 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 [12], Framac C [19], or VeriFast [15]—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.

Applications of VST include the verification of cryptographic primitives from OpenSSL [3,9] and mbedTLS [37], an asynchronous communication mechanism [26], an internet-facing server component [21], a generational garbage collector [35], and a malloc-free library [6]. Of these, the cryptographic applications [9,37] and the reactive server [21] best illustrate the ability of proof-assistant-embedded program logics to bridge the abstraction gap between code-level verification and model-level (semantic) reasoning: in the cryptographic examples, Coq’s functional programming language, Gallina, is employed to implement executable (and hence testable) specification programs that are referred to in the pre- and post-conditions of the VST specifications. On the other hand, these functions are also the object of model-level reasoning, to formally establish cryptographic security; the functional programs hence decouple semantic reasoning from C-level code verification. In the case of the server, the VST specifications refer to *interaction trees* [36], an executable embedding of effectful functional programs in Coq that is equipped with a theory of (weak) bisimulation. Again, model-level reasoning is decoupled from but connected to the verification of the C code.

*The need for subsumption.* Appel and Nauman’s VST verification of a malloc-free system [6] is an application of the new funspec-subsumption method that we describe here. That verification uses abstract predicates with existentially

hidden definitions (which verifiably prevent user programs from breaking the abstraction barrier of the library); and it uses subsumption: the implementations of `malloc` and `free` are first verified w.r.t. a library-wide abstraction predicate (and corresponding function specifications) that exposes the existence of internal freelists for objects of different size. This specification is immediately useful to clients that are resource-aware at the granularity of (approximate) object sizes. As a benefit, such clients need not check that a pointer returned by `malloc` is nonnull: allocation (provably) never fails. A second, more abstract API specification employs a less informative predicate that does not keep track of the available free space; clients using this interface must check the return value of `malloc` against `NULL`, to learn whether `malloc` could satisfy the request for  $N$  bytes. The two library specifications are related by the notion of `funspec.sub` developed in the present paper.

The use of abstract predicates to facilitate data abstraction in separation logics is well established [30], and captures the software engineering principle of representation hiding: 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 invariant. In type theory, this is the principle of *existential types* [27].

But as the `malloc-free` case study illustrates, the same function may want more than one specification: different clients may need different abstractions; indeed, some clients even require full representation exposure (see our running example below). Of course, 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 most concrete specification and then prove that it implies the more abstract one(s). Again, type theory provides an appropriate notion: *subtyping* [31]. In other cases, it may be convenient 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 to let type-theoretic principles guide the development of specification mechanisms and automation features for abstraction. This article takes a principled 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 [20, 29, 28]. Indeed, Kleymann observed that ([20], 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 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 specifications of function pointers, a key construct of the C programming language that requires the program logic to support higher-order reasoning. When a function pointer at location  $v$  is communicated as an argument of a call to some function  $g$ ,  $g$ 's precondition needs to associate  $v$  with a specification—so that the invocation within  $g$  of the function at  $v$  can be verified. A similar issue occurs in spawning threads. To support these language features, VST's semantic model contains—and specification subsumption must hence support—a step-indexed “predicates-in-the-heap” construction [5] that is substantially more complex than the simple state spaces employed in Kleymann's and Nipkow's formalizations. Indeed, Kleymann only considers a single (anonymous, parameterless, but possibly recursive) procedure, while Nipkow supports mutual recursion between named procedures. In short, previous developments were carried out for research languages whereas our work supports virtually all of C (i.e., as noted above, except for `goto` and by-copy struct assignment).
- Our notion of subtyping avoids direct quantification over states. Indeed, “assertion” in VST (and in the similarly expressive Iris framework [17]) is not simply  $\text{state} \rightarrow \text{Prop}$ , but is an abstract type that hides the complexities of the step-indexed model; no `state` type is visible in the program logic. In contrast, Kleymann's and Nipkow's assertions are predicates over states, and the side conditions of their adaptation rules explicitly quantify universally over states. Naumann's formulation using predicate transformers captures the same relationship as Kleymann and Nipkow, slightly more abstractly.
- VST's proof context  $\Delta$  maps globally named functions to their specifications; VST's separation-logic assertion `func.at` 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 [31]. This allows bundling function specifications into specifications of objects or modules that can be abstractly presented to client programs and are reminiscent of behavioral subtyping [25, 22, 32].
- We introduce intersection specifications and show that their interaction with subsumption precisely matches that of intersection types.
- A technical advance over the conference version of this paper [8] is that function specifications avoid any mentioning of formal parameter names. This permits function implementations to be modified more freely ( $\alpha$ -conversion) and adds flexibility in the definition of intersection specifications.
- VST's program logic, including the new subsumption features, is proved sound with a machine-checked proof in Coq.

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 are only sketched. The full Coq proofs are in the VST repo, [github.com/PrincetonUniversity/VST](https://github.com/PrincetonUniversity/VST), commit 948f536: our running example is in directory `progs/pile`.

**Relationship to conference version:** a preliminary version of this article appeared at FM’19, under the same title [8]. The main technical improvement, since that publication, is the elimination of formal parameter names from the surface syntax and the semantic interpretation of function specifications. A key aspect of this improvement is a restructuring of the way parameter passing is modeled in VST’s function call rule. Besides simplifying this proof rule and its associated automation tactics, this restructuring also enables the proof of a technical lemma that mirrors the effect of subsumption (which is formulated from the perspective of the caller) at the callee-side. Another ingredient is a new rule of adaptation for VST’s statement-level judgment that again resembles Kleymann’s rule but additionally supports framing and the multiple ways in which control flow may exit from a code block in C.

Surface-level effects of these improvements are visible in the modified function specifications throughout this article; high-level aspects are described in Sections 3 and 6. Technical details are contained in the Coq formalization. Specifically, (i) the soundness proof required intricate surgery, particularly in the rules for function calls and the interpretation of the auxiliary judgement form for function bodies; (ii) the automation scripts that drive concrete verifications needed syntactically minor, but technically nontrivial modifications, in the tactics for function calls, in the notations and definitions for introducing function specifications and attaching them to function definitions, and in the tactic that initiates the verification of a function body, to correctly treat the logical counterpart of frame stack creation and of binding arguments to local variable names.

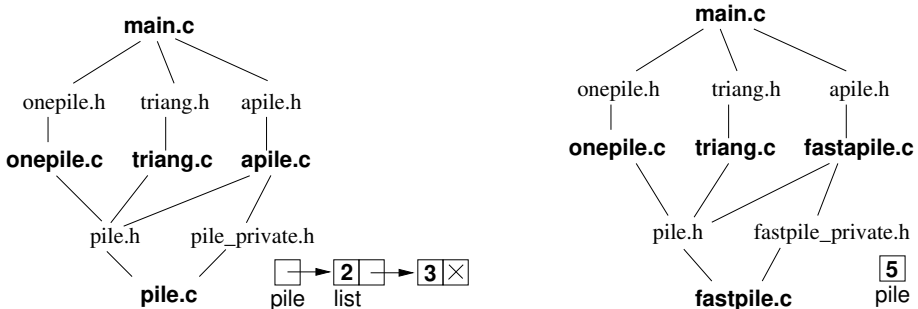


Fig. 1 Module dependency diagrams of two configurations of the `pile` program.

## 2 Motivating example: managing piles

Our main example is an abstract data type (ADT) for *piles*, simple collections of integers.

```

/* 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. 2** 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`.

Figure 2 shows a modular C program that throws numbers onto a pile, then adds them up. Figure 1 (*left*) shows that `pile.c` is called upon 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 data representation—a typical use case for this organization might be when `apile.c` implements representation-dependent debugging or performance monitoring. Thus, *representation-revealing and representation-hiding specifications must both be supported*.

Figure 1 (*right*) shows that when `pile.c` is replaced by a faster implementation `fastpile.c` (code in Figure 4) 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*. Of course, the C language definition does not require the implementation `fastpile.c` to employ the same formal parameter names as `pile.c`, and neither one necessarily uses the same identifiers as the function prototype in `pile.h`. Thus, we use formal parameter name `pp` in Figure 4. *Language-level modularity aspects of APIs, including the opacity of parameter names, should be respected by specifications and their subsumption*.

Figure 3 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 3: the  $\vec{x}$  are bound Coq variables visible in both the precondition and postcondition, in this case,  $p:\text{val}$ ,  $n:\mathbb{Z}$ ,  $\sigma:\text{list } \mathbb{Z}$ ,  $gv:\text{globals}$ , 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 `PRE`condition first lists the C-language types of all formal parameters and then contains an assertion of the form

`PROP(propositions) PARAMS(pvals)`  
`GLOBALS(global bindings) SEP(spatial conjuncts).`

In this case the `PROP` asserts that  $n$  is between 0 and `max-int`; `PARAMS` lists the values received via the formal parameters; `GLOBALS` associates the *global bindings* (typically, exactly  $gv$ , or empty) to VST’s semantic space of global identifiers; `SEP` contains a list of spatial (memory affecting) predicates. This 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`)



```

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

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

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

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

Definition Pile_add_spec :=
  DECLARE _Pile_add
  WITH p: val, n: Z,  $\sigma$ : list Z, gv: globals
  PRE [ tptr tpile, tint ]
  PROP(0  $\leq$  n  $\leq$  Int.max_signed)
  PARAMS (p; Vint (Int.repr n))
  GLOBALS (gv)
  SEP(pilerep  $\sigma$  p; mem_mgr gv)
  POST[ tvoid ]
  PROP() LOCAL()
  SEP(pilerep (n:: $\sigma$ ) p; mem_mgr gv).

Definition sumlist : list Z  $\rightarrow$  Z := List.fold_right Z.add 0.

Definition Pile_count_spec :=
  DECLARE _Pile_count
  WITH p: val,  $\sigma$ : list Z
  PRE [ tptr tpile ]
  PROP(0  $\leq$  sumlist  $\sigma$   $\leq$  Int.max_signed)
  PARAMS (p) GLOBALS () SEP(pilerep  $\sigma$  p)
  POST[ tint ]
  PROP()
  RETURN(Vint (Int.repr (sumlist  $\sigma$ )))
  SEP(pilerep  $\sigma$  p).

```

**Fig. 3** Specification of the pile module (Pile.free.spec not shown).

### 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

PARAMS lists the arguments  
that will be associated  
with the formal parameters  
during (logical) stack  
frame construction

GLOBALS (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

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 *parameter values* are all of (Coq) type `val`, i.e. are all CompCert values. Here,  $p$  represents the pointer to the pile, whereas the second argument, `Vint(Int.repr n)` projects the mathematical integer  $n$  into the space of 32-bit machine integers and then injects it into `val` using the constructor `Vint`.

The Coq type of each spatial conjunct is `mpred`, VST’s opaque abstraction of step-indexed memory predicates. To first approximation, `mpred` can be thought of as  $(\text{mem} \times R) \rightarrow \mathbf{Prop}$  where `mem` is the space of CompCert memories,  $R$  is the space of VST’s resource maps (an instrumentation that assigns, among other things, specifications to locations that hold function-pointers; see [5] for details on VST’s predicates-in-the-heap model) and `Prop` is Coq’s type of propositions. However, an abstraction barrier hides this expansion from the user, providing instead logical operators to combine and manipulate memory predicates; indeed, Coq’s list notation (semicolon) is interpreted as separating conjunction `*` in SEP clauses.

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. In addition, the `POSTcondition` lists the return type. When the return type is nonvoid (see e.g. `Pile.new`), the `POSTcondition`’s `RETURN` clause gives the value.

In VST, the proposition that a function body  $f$  satisfies its specification  $\phi$  is written `semax_body`, or in mathematical notation,

$$\Gamma \vdash_{\text{semax\_body}} f : \phi$$

where  $\Gamma$  is the list of all funspecs of functions that the body of  $f$  might call.

To prove a  $\vdash_{\text{semax\_body}}$  claim, one does a Hoare-logic proof on the function-body  $f$ , with respect to the precondition and postcondition of  $\phi$ .

Verifying that `pile.c`’s functions satisfy the specifications in Fig. 3 using VST-Floyd is done by proving Lemmas like this one (in file `verif_pile.v`):

**Lemma** `body_Pile_add`: `semax_body Vprog Gprog f_Pile_add Pile_add_spec`.

**Proof.** ... (\*16 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.add` might call), the function-body `f_Pile.add` satisfies the function-specification `Pile.add_spec`.

Returning to the discussion of `fastpile.c`, note that while the optimized representation does not actually maintain a list, the functions still satisfy the specifications in `spec_pile.v`, which pretend maintenance of integer sequences. This is crucial for enabling the code substitution as described above. However, we will (in Section 4) additionally equip `fastpile.c` with a second specification, for a representation predicate that is not phrased in terms of sequences.

*Linking.* We organize a modular proof of a modular program as follows: For each module  $M$  (such as  $M = \text{pile}$ ), CompCert parses `M.c` into the AST file `M.v`. Then we write the specification file `spec.M.v` containing funspecs as in

Figure 3. 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 for each of the functions in `M.c`.

So, for example, `pile.c` is parsed into the file `pile.v` that just contains its abstract syntax tree; the user writes `spec_pile.v` containing specifications (fun-specs) for the functions in `pile.c`, and the user writes `verif_pile.v` containing correctness proofs for those functions.

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` files 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 that interacts with the world via effectful external function calls [21] and returns the “right” value from `main`.

### 3 Parameter-nameless function specifications

The previous section introduced the surface notation for specifications as typically seen by VST users. Desugaring this notation indicates that such definitions yield *nondependent function specifications*, which suffice for most cases, except for certain higher-order situations (see below):

`NDmk_funspec` (*tsig*: typesig) (*cc*: calling\_convention)(*A*: Type)  
 (Pre:  $A \rightarrow \text{argsEnviron} \rightarrow \text{mpred}$ )(Post:  $A \rightarrow \text{environ} \rightarrow \text{mpred}$ ): funspec.

To construct a nondependent (ND) function spec, one thus gives the function’s C-language type signature (`typesig`), the calling convention (usually `cc_default`; this is also what the notation mechanism silently expands to), 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* and yield an `mpred`, i.e. a spatial predicate on memories. Postconditions additionally take an environment, comprising bindings for global variables, addressable local variables, and (non-addressable) temporaries:

**Inductive** `environ` :=

`mkEnviron`: forall (*ge*: `genviron`) (*ve*: `venviron`) (*te*: `tenviron`), `environ`.

Preconditions take a global-variable environment and a list of (`CompCert`) values:

**Definition** `argsEnviron`:Type := `genviron` \* (list val).

This definition of `NDmk_funspec` differs from the conference version of this article [8], and from previous expositions of VST [10, 5]; the previous definition

NDmk\_funspec (*funsig*: funsig) (*cc*: calling\_convention)(*A*: Type)  
 (Pre Post:  $A \rightarrow \text{environ} \rightarrow \text{mpred}$ ): funspec.

included the formal parameter names in the signature and used a local-variable environment for binding these to actual values in the precondition. The elimination of parameter names has several benefits:

- it simplifies the proof rule for function calls by allowing us to define a simpler parameter-passing mechanism;
- it captures the fact that formal parameter names are invisible (irrelevant) to callers of a function;
- concretely, the previous notion of specification was not compatible with  $\alpha$ -renaming of parameters; trying to explicitly incorporate  $\alpha$ -renaming into our earlier definitions of funspec-sub [8] and of specification intersection turned out to be overly complex;
- the simplified parameter passing mechanism enables two additional lemmas related to parameter adaptation – see rules `semax_body_funspec_sub` and `semax_adapt_frame` in Section 6.

To illustrate, consider an increment function that has a formal parameter `_p` pointing to an integer in memory. We let  $A = \text{int}$ . The previous system would express the contract as

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

Now, we can express the precondition without referring to `_p`:

$$\text{Pre} = \lambda i : A. \lambda(g, [p]). p \mapsto i.$$

Here,  $g$  represents the global-variable environment (ignored in the body of this simple precondition) and  $[p]$  is a singleton list of arguments. Thus, a client can directly instantiate  $p$  with some value  $v$  rather than having to construct the singleton environment  $\rho = \_p \mapsto v$  (which requires knowing `_p`).

*General function specifications.* Nondependent function specifications suffice for most C programming. But sometimes in the presence of higher-order functions (and hence: function pointers!), 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).

While the details of higher-order function specifications are beyond the scope of this paper, we briefly sketch some key aspects. First, when precondition and postcondition are higher-order, in that their auxiliary variables are *predicates* (such as `mpred`), we must ensure that each is a *bifunctor*. That is, we must keep track of *covariant* and *contravariant* occurrences of `mpred`. That means that the type of a WITH-list,  $A$ , must be given in (semi)deeply embedded form called a `TypeTree`. That is: each “ordinary” Coq type  $\tau$  may be shallowly embedded as `ConstType( $\tau$ )` constructor;  $\tau$  may include arrow types, product types, etc., as long as there are no mentions of `mpred`. Around this we wrap a deeply embedded description of types, including the special constructor

Mpred, and then ProdType for product types, ArrowType, SigType, PiType, *et cetera*.

The purpose of this (semi)deep embedding is to keep track of the covariant and contravariant occurrences of `mpred`; we can reflect `TypeTree` into `Type`, but we can also inspect the `TypeTree` to calculate the required pattern of covariance and contravariance proofs required as part of a `funspec` definition—concrete pre- and postconditions need to come equipped with proofs that they are *nonexpansive*. This approach was outlined by America and Rutten [2] and has been implemented both in *Iris* [17] and *VST*.<sup>1</sup>

For most functions, whose `WITH-list` does not mention `mpred`, all of this complexity is hidden from the user: effectively, the entire `WITH-list-type` is embedded in a single `ConstType` constructor, and the covariance/contravariance proofs are trivial. And therefore, for defining subsumption of these non-dependent (ND) `funspec`s, we can use the simple `NDmk.funspec` constructor shown above. But really, `NDmk.funspec` is a shallowly embedded definition (in *Coq*) for an instance of the general (dependent) case, that is, the `mk.funspec` constructor; see the Appendix.

#### 4 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 4. As `fastpile.c` employs a different 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<sup>2</sup>; 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 as we mentioned in Section 2, the functions in `fastpile.c` can also be equipped with specifications that refer to a different representation predicate, `countrep` (see Figure 5). In reasoning about clients of this low-level interface, we do not need a notion of “sequence”—in contrast to `pilerep` in Fig. 4. The new specification is less abstract than the one in Fig. 4, and closer to the implementation. 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

<sup>1</sup> Bifunctor function-specs in *VST* were originally the work of Qinxiang Cao, Robert Dockins, and Aquinas Hobor, but were adapted to the new form of preconditions as part of the present work.

<sup>2</sup> 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, but is a key ingredient of an abstract component system that we are currently building on top of *VST*.

```

/* 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 pp, int n)
  {int s = pp->sum; if (0 ≤ n && n ≤ INT_MAX-s) pp->sum = s+n; }
int Pile_count(Pile pp) {return pp->sum;}
void Pile_free(Pile pp) {free(pp);}

(* spec_fastpile.v *)
Definition pilerep (σ: list Z) (p: val) : mpredef :=
  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.3 *)
Definition Pile_new_spec := (* looks identical to the one in fig.3 *)
Definition Pile_add_spec := (* looks identical to the one in fig.3 *)
Definition Pile_count_spec := (* looks identical to the one in fig.3 *)

```

**Fig. 4** 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 → val → mpredef`, even though the representation is a single number rather than a linked list.

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).<sup>3</sup> 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`.

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 4 *)
spec_fastpile_concrete.Pile_add_spec: ident*funspec (* in Figure 5 *)

```

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

<sup>3</sup> For example: in our proof of HMAC-DRBG [37], 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 3e61d29 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 55 lines of Coq (at line 42 of the same file).

(\* 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) PARAMS (p; Vint (Int.repr n))
  GLOBALS (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. 5** 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.

**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<sup>4</sup> rule of consequence

$$\frac{\vdash \{P'\}c\{Q'\}}{\vdash \{P\}c\{Q\}} \forall Z. \forall \sigma. PZ\sigma \rightarrow \forall \tau. \exists Z'. (P'Z'\sigma \wedge (Q'Z'\tau \rightarrow QZ\tau))$$

where assertions are binary predicates over auxiliary and ordinary states, and *Z*, *Z'* are the WITH values.<sup>5</sup> Our subsumption applies to function specifications, not arbitrary statements *c*. In the rule for function calls, it ensures that a

<sup>4</sup> Kleymann’s program logic, like ours, uses *auxiliary variables* (which we call WITH-lists) to relate the precondition to the postcondition. When auxiliary variables are used, one must be able to choose them freely to express this relation between pre and post. Two funspecs for the same function, related by `funspec_sub`, may have quite different auxiliary variables. This is the *parameter adaption* aspect of Kleymann’s system, and of ours. Kleymann pointed out that parameter adaption is necessary in order to achieve adaptation completeness, which is the property that if  $\forall c. \models \{P\}c\{Q\} \Rightarrow \models \{P'\}c\{Q'\}$  then one can derive that  $\vdash \{P\}\{Q\}$  implies  $\vdash \{P'\}\{Q'\}$ , independent of *c*.

<sup>5</sup> We give Kleymann’s rule for total correctness here. Kleymann’s partial-correctness adaptation rule cannot guarantee safety. That is: Kleymann’s total-correctness Hoare triple says, “If the start state satisfies *P*, then the command *c* will terminate, and will terminate in a state satisfying *Q*.” Kleymann’s partial-correctness Hoare triple says, “If the start state satisfies *P*, then if the command *c* terminates, then the final state satisfies *Q*.” The problem is that *c* might crash (or “get stuck” in operational-semantic terms), in which case Kleymann’s partial-correctness Hoare triple is still satisfied. For unsafe languages such as C, that

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:

```
Definition incr1_spec := DECLARE _incr1
WITH i: Z, a: val, π: share, private: list val
PRE [ tint, tptr tuint ]
  PROP (0 ≤ i < Int.max_signed; writable.share π)
  PARAMS (Vint (Int.repr i); a) GLOBALS ()
  SEP(data.at sh (tarray tuint 10) private a)
POST [ tint ]
  EX private': list val, PROP() RETURN(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)
  PARAMS (Vint (Int.repr i); a) GLOBALS () SEP()
POST [ tint ]
  PROP() RETURN(Vint (Int.repr (i + 1))) SEP()
```

is not a very useful Hoare triple, nor is his partial-correctness adaptation rule useful. `VST` is a logic for partial correctness, but its Hoare triple means, “If the start satisfies  $P$ , then it is safe to execute  $c$  ( $c$  will not crash);  $c$  will either infinite-loop, will safely exit by (e.g.) returning from the function, or will terminate in a state satisfying  $Q$ ”. This is useful for unsafe languages.



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 3, the specification of `pile_rep`, `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 `pile_rep'`. 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 `pile_rep` 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 `pile_rep'` 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.

## 5 Definitions of funspec subtyping

*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` ( $f_1 f_2 : \text{funspec}$ ) :=

**match**  $f_1, f_2$  **with**

`NDmk_funspec tsig cc1 A1 P1 Q1, NDmk_funspec (ptypes, rt) cc2 A2 P2 Q2 ⇒`

**let**  $\Delta := \text{rettype\_tycontext } rt$  **in**

`tsig = (ptypes, rt) ∧ cc1 = cc2 ∧ A1=A2 ∧`

`(∀x : A1, Δ, P2 nil x ⊢ P1 nil x) ∧`

`(∀x : A1, (ret0_tycon Δ), Q1 nil x ⊢ Q2 nil x)`

**end.**

We write  $\Delta, P_2 \text{ nil } x \vdash P_1 \text{ nil } x$ , where  $P_1$  and  $P_2$  are the preconditions of  $f_1$  and  $f_2$ ,  $\text{nil}$  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  $\text{ret0.tycon } \Delta$  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 [20, 29].

*Useful, ordinary funspec subtyping.* If  $\text{NDmk.funspec}$  were a constructor, we could define,

**Definition**  $\text{NDfunspec.sub } (f_1 f_2 : \text{funspec}) :=$

**match**  $f_1, f_2$  **with**

$\text{NDmk.funspec } \text{tsig } cc_1 A_1 P_1 Q_1, \text{NDmk.funspec } (p\text{types}, rt) cc_2 A_2 P_2 Q_2 \Rightarrow$

**let**  $\Delta := \text{retype.tycontext } rt$  **in**

$\text{tsig} = (p\text{types}, rt) \wedge cc_1 = cc_2 \wedge$

$\forall x_2 : A_2 \rho : \text{argsEnviron},$

$\Delta, P_2 \text{ nil } x_2 \rho \vdash$

$\text{EX } x_1 : A_1, \text{EX } F : \text{mpred}, (F * P_1 \text{ nil } x_1 \rho) \ \&\&$

$!! (\forall \tau. (\text{tc\_environ } \Delta), F * Q_1 \text{ nil } x_1 \tau \vdash Q_2 \text{ nil } x_2 \tau)$

**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  $\text{NDmk.funspec}$ , that is, for function specifications whose witness types  $A$  do not contain (co-variant or contravariant) occurrences of  $\text{mpred}$ . We present the general, dependent funspec-sub in the Appendix, with its constructor  $\text{mk.funspec}$ , and show the construction of  $\text{NDmk.funspec}$  as a derived form. And actually, since  $\text{NDmk.funspec}$  is not really a constructor (it is a function that applies the constructor  $\text{mk.funspec}$ ), we must define  $\text{NDfunspec.sub}$  as a pattern-match on  $\text{mk.funspec}$ ; see the Appendix.

## 6 The subsumption rules

The purpose of  $\text{funspec.sub}$  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  $\text{tycontext.sub } \Delta \Delta'$ . Again, this relation is reflexive and transitive.

**Definition** (*glob\_specs*) If  $i$  is a global identifier, write  $(\text{glob\_specs } \Delta)!i$  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  $(\text{glob\_specs } \Delta)!i = \text{Some } \phi$  then  $(\text{glob\_specs } \Delta')!i = \text{Some } \phi'$  and *funspec\_sub*  $\phi \phi'$ .

**Then**  $\Delta <: \Delta'$ .

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

**Theorem** (*semax\_Delta\_subsumption*)

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

*Proof.* Nontrivial. Because this is a logic of higher-order recursive function pointers, our Coq proof<sup>6</sup> in the modal step-indexed model uses the Löb rule to handle recursion, and unfolds our rather complicated semantic definition of the Hoare triple [5].

But this is not the only subsumption rule we desire. Because C has function-pointers, the general function-call rule is for  $\Delta \vdash \{P\}e_f(e_1, \dots, e_n)\{Q\}$  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:

$$\frac{(\text{glob\_specs } \Delta)!f = \text{Some } \phi \quad \Delta \vdash f \Downarrow v \quad \Delta \vdash \{\text{func\_ptr } v \phi \wedge P\}c\{Q\}}{\Delta \vdash \{P\}c\{Q\}} \quad \frac{\Delta \vdash e_f \Downarrow v \quad \Delta \vdash e_1 \Downarrow v_1 \dots \Delta \vdash e_n \Downarrow v_n \quad P * F \vdash \text{func\_ptr } v \phi \quad \phi(w) = \{P\}\{Q\}}{\Delta \vdash \{P * F\}e_f(e_1, e_2, \dots, e_n)\{Q * F\}}$$

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*  $\phi$ .

At right, 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  $Q$ ;

<sup>6</sup> See file `veric/semax_lemmas.v` in the VST repo.

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  $\phi <: \phi' \wedge \phi'(w) = \{P\}\{Q\}$ . That is,

$$\text{call-with-subsumption} \frac{\begin{array}{c} \Delta \vdash e_f \Downarrow v \\ \Delta \vdash e_1 \Downarrow v_1 \dots \Delta \vdash e_n \Downarrow v_n \\ P * F \vdash \text{func_ptr } v \phi \\ \phi <: \phi' \wedge \phi'(w) = \{P\}\{Q\} \end{array}}{\Delta \vdash \{P * F\}e_f(e_1, e_2, \dots, e_n)\{Q * F\}}$$

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

*Function-definition subsumption.* Recall that the proposition “function  $f$  satisfies its specification  $\phi$ ” is written,

$$\Gamma \vdash_{\text{semax\_body}} f : \phi$$

where  $\Gamma$  is the list of all funspecs of functions that the body of  $f$  might call. The proof is (typically) by proving the Hoare triple  $\forall x. \{P\}c\{Q\}$ , where  $x$  is the WITH-list of  $\phi$ , where  $P$  and  $Q$  are the precondition and postcondition of  $\phi$ , and  $c$  is the function body of  $f$ .

For a fully expressive notion of subsumption, one wants to apply it also at function definitions. Therefore, we have the rule,

$$\text{semax\_body\_funspec\_sub} \frac{\phi <: \phi' \quad \Gamma \vdash_{\text{semax\_body}} f : \phi}{\Gamma \vdash_{\text{semax\_body}} f : \phi'}$$

We found this rule almost essential for a fully expressive module system that can describe data abstraction; but (surprisingly) we could not prove it sound in our first-generation `funspec_sub` system [8]. Now that we use nameless formal parameters in funspecs, we have been able to prove this rule. As the  $\vdash_{\text{semax\_body}}$  judgment is defined in terms of VST's Hoare-logic judgment for C statements, it is perhaps not surprising that the proof of `semax_body_funspec_sub` utilizes a rule of consequence

$$\text{semax\_adapt\_frame} \frac{\Delta \vdash \{P'\}c\{Q'\} \quad \textit{SideCondition}}{\Delta \vdash \{P\}c\{Q\}}$$

with a side condition that—just like parameter adaption in `funspec_sub`—permits one to exploit the satisfaction of the conclusion's precondition,  $P$ , when proving the entailment between the postconditions and also includes framing. In slightly simplified form, the *Side Condition* is given by

$$\Delta, P \vdash (\text{EX } F. \text{!!}(\text{closed\_wrt\_modvars } c \ F) \ \&\& \ (P' * F) \ \&\& \ \text{!!}((Q' * F \vdash Q))).$$

However, the notation  $Q' * F \vdash Q$  here actually abbreviates four slightly different entailments, as statement-level postconditions in VST are quadruples of assertions, containing separate components for each possible control flow continuation of a code block: function return, `break`, `continue`, and fall-through.

## 7 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  $b : \text{bool}, \vec{x} : \vec{\tau}$ 
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 [...] 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 [...] 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  $\vec{x} : \vec{\tau}, i : \text{int}, \vec{y} : \vec{\sigma}$  PRE [...]  $P_1(\vec{x}, i, \vec{y})$  POST [...]  $Q_1(\vec{x}, i, \vec{y})$  and
WITH  $\vec{x} : \vec{\tau}, s : \text{string}, \vec{y} : \vec{\sigma}$  PRE [...]  $P_2(\vec{x}, s, \vec{y})$  POST [...]  $Q_2(\vec{x}, s, \vec{y})$ .
```

Generalizing to arbitrary index sets, we may—for a given function signature and calling convention—combine specifications into specification *families*, by lifting the dependent sum (i.e. sigma, `sigT` below) type construction from WITH-lists to function specifications:

```
Definition funspec_Sigma_ND  $tsig\ cc$  (I:Type) (A : I  $\rightarrow$  Type)
  (Pre: forall i, A i  $\rightarrow$  argsEnviron  $\rightarrow$  mpred):
  (Post: forall i, A i  $\rightarrow$  environ  $\rightarrow$  mpred): funspec :=
```

**Proof.**

```
  apply (NDmk.funspec sig cc (sigT A)).
  intros [i Ai] rho; apply (Pre _Ai rho).
  intros [i Ai] rho; apply (Post _Ai rho).
```

**Defined.**

This shows—using Coq's *definition-by-proof-script* feature—the nondependent (ND) case only, but our Coq development also contains the general case (all this in `veric/seplog.v`).

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

```
Lemma funspec_Sigma_ND_sub: forall  $tsig\ cc$  I A Pre Post i,
  funspec_sub (funspec_Sigma_ND  $tsig\ cc$  I A Pre Post)
    (NDmk.funspec  $tsig\ cc$  (A i) (Pre i) (Post i)).
```

**Lemma** `funspec_Sigma_ND_sub3`: forall *tsig cc* I A Pre Post g (i:I)  
 (HI: forall i, funspec\_sub g (NDmk\_funspec *tsig cc* (A i) (Pre i) (Post i))),  
 funspec\_sub g (funspec\_Sigma\_ND *tsig 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  $\frac{\forall i, \sigma <: \tau_i}{\sigma <: \wedge_{i \in I} \tau_i}$ , the specializations of which to the binary case appear on page 206 of TAPL [31]. We expect these rules to be helpful for formalizing Leavens and Naumann’s treatment of specification inheritance in object-oriented programs [22].

An *ad hoc* form of (binary) intersection was already used in our verification of the HMAC-DRBG cryptographic component [37]. A more general application of intersection occurs in a rule for composition of VST-verified compilation units that is part of a component layer on top of VST (under current development). Finally, we showed in previous work [7] how relational (2-execution) specifications can be encoded as unary VDM-style specifications. Intersection specifications may be seen as internalization of VDM’s “sets of specifications”.

## 8 Related work

There are other proof tools for languages such as C and Java, but none of them (to our knowledge) has a full calculus for function-specification subsumption. Instead of being embedded in a general proof assistant, most current verification tools embed their assertions directly in the program code and employ verification condition discharge using SMT solving. **VCC** [12] is a verifier for concurrent C programs, using the Boogie language-independent SMT-based verifier as a back end. **VeriFast** [15] is a separation logic for C, that can relate C programs to functional models with substantial automation based on SMT. **Frama-C** [19] is an analysis/verification tool that supports Hoare-logic verification of C programs with an assertion language called ACSL [13], which uses C-like syntax for its first-order assertion language. **Dafny** [23] is an SMT-based Hoare-logic verifier for a small but capable language (also called Dafny), with substantial proof automation using Boogie. **KeY** [1] is a verifier for Java, with specifications in JML and an assertion language based on *dynamic logic* [14]. It is an interactive theorem prover, programmable by “strategy macros” and with SMT plug-ins (in that sense, comparable to VST with Ltac programming and Coq’s linear-integer-arithmic plugin). A later extension [33] supports heap-modular reasoning using the theory of *dynamic frames* [18], backed up by a formalization in Isabelle/HOL [34] for a core calculus.

None of these verifiers has a machine-checked soundness proof of the tool or program logic, as VST does. All these verifiers have less expressive functional-modeling languages than VST, to varying degrees—because VST uses the full power of Coq’s dependently typed higher-order logic (CiC) for this purpose. All of them have weaker systems than VST for reasoning about logical properties of functional models (compared to the power of Coq’s tactical proof assistant, that VST uses for this).

VeriFast has a form of `funspec_sub`, called `produce_function_pointer_chunk`, that verifies a new specification for a function (with user assistance as for any VeriFast proof) by proving a function call based on the old specification. This supports framing and parameter adaption, but not transitivity of `funspec_sub`.<sup>7</sup>

As far as we can tell from reading the literature, VCC, Frama-C, Dafny, and KeY’s notion of `funspec_sub` are not formally equipped with a subsumption rule but directly integrated into proof rules for (virtual) methods. JML supports explicit intersection specifications using the keyword `also`, and these are implicitly employed in these tools’ implementations of behavioral subtyping. However, the user-interface design of VCC, Frama-C, Dafny, and KeY make `funspec_sub` and intersection less natural than they are in VST. This has little to do with VST’s more powerful (impredicative higher-order) semantic model: it is about *separating specifications from implementations*. That is, in VST the function specification is a self-contained syntactic unit, not intermingled in the same source file with the function body. In those other systems, assertions (preconditions, postconditions, loop invariants) are intermingled with the C or Java source code, as comments with a special form. Therefore, one typically has *the* specification of a function, mixed into the implementation; syntactically there is no room for more than one.<sup>8</sup> In VST it is much more natural—if we can write *one* `funspec` (in a different place from the function body), then it is easy enough to write more than one, to relate them, and to have proofs of the same function w.r.t. different specifications, perhaps with different loop invariants. Although this is not a fundamental, deep semantic aspect, we believe that fully supporting subsumption in these program verifiers may benefit from revisiting the design decision that mixes specifications into implementations.

The program logic framework that is most closely related to VST is Iris [17], which also implements a higher-order concurrent separation logic in Coq and provides a step-indexed model. At present, applications of Iris predominantly concern small-scale research languages. However, we anticipate that our development could be rather easily transferred to emerging applications of Iris to Rust [16] or Go [11], although neither of these developments are currently linked to formally verified compilers.

## 9 Conclusion

Even without `funspec` subtyping, separation logic easily expresses data abstraction [30]. 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

<sup>7</sup> Bart Jacobs, by e-mail, September 2020.

<sup>8</sup> VeriFast permits the function specification to be attached to the function *definition* in the `.c` file or to the function *declaration* in the `.h` file. This is a limited form of separating the specification from the implementation.

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 `onpile.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 our current proofs need to rerun the proof scripts on the modified 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`

`NDfunspec_sub` as introduced in Section 5 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/seplug.v`):

```
Definition funspec_sub_si (f1 f2 : funspec):mpred :=
match f1, f2 with
  mk_funspec tsig cc1 A1 P1 Q1 - -, mk_funspec (ptypes, rt) cc2 A2 P2 Q2 - -
  =>
    let Δ := rettype_tycontext rt in
      !(tsig = (ptypes, rt) ∧ cc1 = cc2) &&
        ! (ALL ts2:list Type, ALL x2: F A2, ALL gargs:argsEnviron,
          (!(tc_argseNV Δ ptypes gargs) && P2 ts2 x2 gargs)
          => EX ts1:list Type, EX x1: F A1, EX F,
            (F * P1 ts1 x1 gargs) &&
            ALL ρ:environ, !( (local (tc_enviroN 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:  $P \Rightarrow Q$  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  $\Rightarrow$  where they cannot tolerate  $\vdash$  [5, Chapter 17].

Second, both funspecs are constructors (`mk_funspec tsig cc A P Q - -`) as discussed in Section 5, but the two final arguments (the proofs that  $P$  and



$Q$  are super-nonexpansive) are irrelevant for the remainder of the definition and hence anonymous. We also abbreviate the `TypeTree`-interpreting operator alluded to in Section 3, `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 [5], 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”  
 $\Rightarrow$  “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  $Q_2$ .

Finally, note that the definition internally existentially quantifies over yet another `mpred`, the frame  $F$ .

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`,  
 !!( $v = \text{Vptr } b \text{ Ptrofs.zero}$ ) && 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`,  
 !!( $v = \text{Vptr } b \text{ Ptrofs.zero}$ ) && 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`  $f$   $v$   $\vdash$  `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`  $f$   $v$   $\vdash$  `func_ptr_si`  $g$   $v$ .

**Lemma** `func_ptr_mono`  $fs$   $gs$   $v$ : `funspec_sub`  $f$   $gs$   $\rightarrow$  (`func_ptr`  $f$   $v$   $\vdash$  `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.

**Acknowledgements** We are grateful to the members of the VST research group projects for their feedback, and we greatly appreciate the comments and suggestions made by the FM'19 program committee and by this journal's referees.

## References

1. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. *Deductive Software Verification—The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.
2. 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.
3. 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.
4. Andrew W. Appel, Lennart Beringer, Qinxiang Cao, and Josiah Dodds. Verifiable C: applying the Verified Software Toolchain to C programs. <https://vst.cs.princeton.edu/download/VC.pdf>, 2019.
5. 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.
6. Andrew W. Appel and David A. Naumann. Verified sequential malloc/free. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, pages 48–59, 2020.
7. Lennart Beringer. Relational decomposition. In *Interactive Theorem Proving (LNCS 6898)*, pages 39–54, Berlin, 2011. Springer.
8. Lennart Beringer and Andrew W. Appel. Abstraction and subsumption in modular verification of C programs. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Proceedings*, volume 11800 of *LNCS*, pages 573–590. Springer, 2019.
9. 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 Association, August 2015.
10. 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.
11. Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nikolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 243–258. ACM, 2019.
12. 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.
13. Jens Gerlach, Denis Efremov, Tim Sikatzi, Malte Brodmann, Jochen Burghardt, Andreas Carben, Robert Clausecker, Liangliang Gu, Kerstin Hartig, Timon Lapawczyk, Hans Werner Pohl, Juan Soto, and Kim Völlinger. ACSL by example: Towards a formally verified standard library, version 21.1.0. <https://github.com/fraunhoferfokus/acsl-by-example>, 2010.
14. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.
15. 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.
16. Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018.

17. 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.
18. Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.
19. 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.
20. Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Asp. Comput.*, 11(5):541–566, 1999.
21. 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.
22. 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.
23. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
24. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
25. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
26. 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.
27. 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.
28. David A. Naumann. Deriving sharp rules of adaptation for Hoare logics. Technical Report 9906, Department of Computer Science, Stevens Institute of Technology, 1999.
29. 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.
30. 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.
31. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Mass., 2002.
32. Cees Pierik and Frank S. de Boer. A proof outline logic for object-oriented programming. *Theor. Comput. Sci.*, 343(3):413–442, 2005.
33. Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in java dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2010.
34. Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in java dynamic logic – formalization and proofs. Technical Report 2010-11, KIT – Karlsruher Institut für Technologie, 2010.
35. Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. Certifying graph-manipulating C programs via localizations within data structures. *PACMPL*, 3(OOPSLA):171:1–171:30, 2019.

36. Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *PACMPL*, 4(POPL):51:1–51:32, 2020.
37. 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.