Overcoming Restraint: Composing Verification of Foreign Functions with Cogent

Louis Cheung
University of Melborune
Melbourne, Victoria, Australia
lfcheung@student.unimelb.edu.au

Liam O'Connor
University of Edinburgh
Edinburgh, Scotland
l.oconnor@ed.ac.uk

Christine Rizkallah
University of Melbourne
Melbourne, Victoria, Australia
christine.rizkallah@unimelb.edu.au

Abstract

Cogent is a restricted functional language designed to reduce the cost of developing verified systems code. Because of its sometimes-onerous restrictions, such as the lack of support for recursion and its strict uniqueness type system, Cogent provides an escape hatch in the form of a foreign function interface (FFI) to C code. This poses a problem when verifying Cogent programs, as imported C components do not enjoy the same level of static guarantees that Cogent does. Previous verification of file systems implemented in Cogent merely assumed that their C components were correct and that they preserved the invariants of Cogent's type system. In this paper, we instead prove such obligations. We demonstrate how they smoothly compose with existing Cogent theorems, and result in a correctness theorem of the overall Cogent-C system. The Cogent FFI constraints ensure that key invariants of Cogent's type system are maintained even when calling C code. We verify reusable higher-order and polymorphic functions including a generic loop combinator and array iterators and demonstrate their application to several examples including binary search and the BilbyFs file system. We demonstrate the feasibility of verification of mixed Cogent-C systems, and provide some insight into verification of software comprised of code in multiple languages with differing levels of static guarantees.

CCS Concepts: • **General and reference** → **Verification**; • **Software and its engineering** → **Formal software verification**; Functionality; **Interoperability**; *Compilers*; • **Theory of computation** → Logic and verification.

Keywords: compilers, verification, type-systems, language interoperability, data-structures

CPP '22, January 17–18, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9182-5/22/01…$15.00
https://doi.org/10.1145/3497775.3503686

ACM Reference Format:
Louis Cheung, Liam O'Connor, and Christine Rizkallah. 2022. Overcoming Restraint: Composing Verification of Foreign Functions with Cogent. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '22), January 17–18, 2022, Philadelphia, PA, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3497775.3503686

1 Introduction

Cogent [16] is a restricted purely functional language with a *certifying compiler* [16, 21] designed to ease creating verified operating systems components [3]. It has a foreign function interface (FFI) that enables implementing parts of a system in C. Cogent's main restrictions are the purposeful lack of recursion or loops, which ensures totality, and its *uniqueness type system*, which enforces a *uniqueness invariant* that, among other benefits, guarantees memory safety.

Even in the restricted target domains of Cogent, real programs contain some amount of iteration, primarily over data structures such as buffers. This is achieved through Cogent's principled FFI: engineers provide data structures and their associated operations, including iterators, in a special dialect of C, and import them into Cogent, including in formal reasoning. This special C code, called *template C*, can refer to Cogent data types and functions, and is translated into standard C along with the Cogent program by the Cogent compiler. As long as the C components respect Cogent's foreign function interface — i.e., are correct, memory-safe and respect the uniqueness invariant — the Cogent framework guarantees that correctness properties proved on high-level specs also apply to the compiler output.

Two real-world Linux file systems have been implemented in Cogent — `ext2` and BilbyFs — and key operations of BilbyFs have been verified [3]. This prior work demonstrates Cogent's suitability as a systems programming language and as a verification framework that reduces the cost of verification. The implementations of these file systems import an external C library of data structures, which include fixed-length arrays and iterators for implementing loops, as well as Cogent stubs for accessing a range of the Linux kernel's internal APIs. This library was carefully designed to ensure compatibility with Cogent's FFI constraints, but was previously left unverified — only the Cogent parts of these file system operations were proven correct, and statements of the underlying C correctness and FFI constraints defining Cogent-C interoperability were left as assumptions.

To fully verify a system written in Cogent and C, one needs to provide manually-written abstractions of the C parts, and manually prove refinement through Cogent's FFI. The effort required for this manual verification remains substantial, but the reusability of these libraries allows this cost to be amortised across different systems.

In this paper, we eliminate several of these assumptions by verifying the array implementation and key iterators used in the BilbyFs file system (Section 3), and discharging the conditions imposed by Cogent's FFI (Section 5). We also verify a generic-loop combinator (Section 4) and its application to binary search. This demonstrates that it is possible and relatively straightforward for the C components of a real-world Cogent-C system to satisfy Cogent's FFI conditions. The compiler-generated refinement and preservation proofs compose with manual C proofs at each intermediate level up to Cogent's generated shallow embedding, and verify some example Cogent-C programs.

As arrays and loops are extremely common in Cogent programming, our proofs are highly reusable for verification of any future Cogent-C system. In addition, our proofs connect C arrays to plain old Isabelle/HOL lists and loops to an Isabelle/HOL repeat function that allows early termination. These proofs are reusable even beyond the context of Cogent in verification of C code. Our code and proofs are online [4].

Similar to many high-level languages, Cogent's foreign function interface connects a high-level language with strong static guarantees to an unsafe imperative language. This work provides our community with a case-study demonstrating how to equip such a foreign function interface with proof requirements such that those static guarantees are maintained for the overall system. In particular, our work supports, and is well-described by, Ahmed's claim that:

"Compositional compiler correctness is, in essence, a language interoperability problem: for viable solutions in the long term, high-level languages must be equipped with principled foreign-function interfaces that specify safe interoperability between high-level and low-level components, and between more precisely and less precisely typed code." [1]

Our approach to language interoperability does not rely on how the refinement theorems of the languages are obtained nor on whether they are manually or automatically proven. As such, we believe that this approach is likely reusable in the context of verified compilers.

2 Cogent

The Cogent language [15–17] was originally designed for the implementation of systems components such as file systems [3]. It is a purely functional language, but it is compiled into efficient C code suitable for systems programming[1].

The Cogent compiler produces three artefacts: C code, a shallow embedding of the Cogent code in Isabelle/HOL [14], and a formal refinement proof relating the two [16, 21]. The refinement theorem and proof rely on several intermediate embeddings also generated by the Cogent compiler, some related through language level proofs, and others through translation validation phases (Section 2.5). The compiler certificate guarantees that correctness theorems proven on top of the shallow embedding also hold for the generated C, which eases verification, and serves as the basis for further functional correctness proofs.

A key part of the compiler certificate depends on Cogent's *uniqueness type system*, which enforces that each mutable heap object has exactly one active pointer in scope at any point in time. This *uniqueness invariant* allows modelling imperative computations as pure functions: the allocations and repeated copying commonly found in functional programming can be replaced with destructive updates, and the need for garbage collection is eliminated, resulting in predictable and efficient code.

Well-typed Cogent programs have two interpretations: a purely functional *value semantics*, which has no notion of a heap and treats all objects as immutable values, and an imperative *update semantics*, describing the destructive mutation of heap objects. These two semantic interpretations correspond (Section 2.5), meaning that any correctness proofs about the value semantics also apply to the update semantics. As we shall see, this correspondence further guarantees that well-typed Cogent programs are memory safe.

2.1 Language Design and Examples

Cogent has unit, numeric, and boolean primitive types, as well as functions, sum types (i.e., variants) and product types (i.e., tuples and records). Users can declare additional *abstract* types in Cogent, and define them externally (i.e., in C). Abstract and record types may be *boxed*, that is, stored on the heap, in which case they are mutable and subject to the uniqueness restrictions of Cogent's type system. Cogent does not support closures, so partial application via currying is not common. Thus, functions of multiple arguments take a tuple or record of those arguments.

Figure 1 includes an example of Cogent signatures for an externally-defined array library interface, where array indices and length are unsigned 32-bit integers (`U32`).

[1] While Cogent is ideally suited for applications that involve minimal sharing and where efficiency matters, it is not specific to the systems domain.

```
type Array a
length : (Array a)! → U32
get : ((Array a)!, U32, a!) → a!
put : (Array a, U32, a) → Array a

map : (a → a, Array a) → Array a
fold : ((a!, b) → b, b, (Array a)!) → b
-----------------------------------------
add : (U32, U32) → U32
add (x, y) = x + y
sum : (Array U32) ! → U32
sum arr = fold (add, 0, arr)
```

Figure 1. A Cogent sum program that makes use of an abstract array type and operations.

```
length : (Array a)! → U32
mapAccum : ((a, b, c!) → (a, b), b, (Array a)!, U32, U32, c!)
           → (Array a, b)
fold : ((a, b, c!) → b, b, (Array a)!, U32, U32, c!)
-----------------------------------------
add : (U32, U32, ()) → U32
add (x, y, z) = x + y
sum : (Array U32)! → U32
sum arr = fold (add, 0, arr, 0, length arr, ())
```

Figure 2. The sum function now written against the interace from Cogent's C library.

Like ML, Cogent supports *parametric polymorphism* for top-level functions, and implements it via monomorphisation. For imported code, the compiler generates specialised C implementations from a polymorphic template, one for each concrete instantiation used in the Cogent code. Variables of polymorphic type are by default *linear*, which means they must be used exactly once [22]. Thus a polymorphic type variable may be instantiated to any type, including types that contain pointers, while preserving the uniqueness invariant.

As mentioned, types and functions provided in external C code are called *abstract* in Cogent. The Cogent compiler has infrastructure for linking the C implementations and the compiled Cogent code. Users write *template C* code, that can include embedded Cogent types and expressions via quasi-quotation, and the Cogent compiler translates the template C into ordinary C that it links with the C code generated from Cogent. To represent containers, abstract types may be given type parameters. These parameterised types, as well as polymorphic functions, are translated into a family of automatically generated C functions and types; one for each concrete type used in the Cogent program.

Though the *Array* type interface may appear purely functional, Cogent assumes that all abstract types are by default *linear*, ensuring that the uniqueness invariant applies to variables of type *Array*. Therefore, any implementation of the abstract put function is free to destructively update the provided array, without contradicting the purely functional semantics of Cogent.

When functions only need to read from a data structure, uniqueness types can complicate a program unnecessarily by requiring a programmer to thread through all state, even unchanged state. The !-operator helps to avoid this by converting *linear*, *writable* types to *read-only* types that can be freely shared or discarded. This is analogous to a *borrow* in Rust. The length and get functions, presented in Figure 1, can *read* from the given array, but may not *write* to it.

As Cogent does not support recursion, iteration is expressed using abstract *higher-order functions*, providing basic traversal combinators such as map and fold for abstract types, as can be seen in Figure 1. Note that map is passed a function of type $a \rightarrow a$. As such, map is able to destructively overwrite the array with the result of the function applied to each element.

While Cogent supports higher-order functions, it does not support nested lambda abstractions or closures, as these can require allocation if they capture variables. Thus, to invoke the map or fold functions, a separate top-level function must be defined, such as add in our example.

The array interface from Cogent's C library used in the implementation of the Cogent file systems, part of which is given in Figure 2, is more complex than that of Figure 1: The higher-order functions are given two additional index parameters to operate over only a subsection of an array; and instead of relying on closure captures, which are not available in Cogent, we provide alternative iterator functions which carry an additional *observer* read-only input (of type $c!$). In addition, the function mapAccum is a generalised version of map, which allows threading an accumulating argument through the map function, similar to the same function in Haskell. We present the verification of these iterators in Section 3.

2.2 Dynamic Semantics

Cogent's big-step *value* semantics is defined through the judgement $V \vdash e \Downarrow_{v} v$. This judgement states that the expression e under environment V evaluates to the value v. The environment V maps variables to their values. The imperative *update* semantics, which additionally may manipulate a mutable store μ, is defined through the judgement $U \vdash e|\mu \Downarrow_{u} u|\mu'$. This states that, starting with an initial store μ the expression e will evaluate under the environment U to a final store μ' and a result value u. Unlike the values in the value semantics, values in the update semantics may be *pointers* to locations in the store.

Both of these semantics are further parameterised by additional *functions* and types of *values* that are provided externally to Cogent, to model the semantics of abstract functions and types. More formally, the value semantics is parameterised by a function $\xi_v : f_{id} \to (v \to v)$ and the update semantics by a function $\xi_u : f_{id} \to (\mu \times u \to \mu \times u)$. Both of these are essentially an environment providing a pure HOL function on Cogent values (and stores, for the update semantics) for each abstract function. The definitions of values in the value semantics (v) and update semantics (u) are also extended with parameters a_v and a_u respectively which represent values of abstract types.

Along with C code for all abstract functions and types, the user must also manually provide Isabelle/HOL abstractions of this C code to instantiate these environments.

To verify Cogent systems, three main proof obligations must be discharged: *type preservation*, which ensures the uniqueness invariant is maintained; the *frame requirements*, which ensures that memory safety is maintained; and *refinement*, which ensures that functional correctness theorems are preserved down to the C level via the provided abstractions. Cogent proves all three of these requirements automatically for Cogent code: both type preservation and the frame requirements are simple corollaries of the key semantic correspondence theorem (Theorem 2.5) that makes up part of the Cogent refinement chain. For linked C code, however, the user must discharge these obligations manually. We discuss our verification of these requirements for C code in Section 3.

2.3 Typing and Type Preservation

Cogent's *static semantics* are defined through a standard typing judgement $A; \Gamma \vdash e : \tau$, which states that e has type τ under context Γ, with an additional context A that tracks assumptions about the linearity of type variables in τ. To accommodate abstract types, we allow the type system to be extended with types $\mathsf{A}\,\overline{\tau}$ and $(\mathsf{A}\,\overline{\tau})!$, referring to linear abstract types and read-only abstract types respectively, where A is a type constructor parameterised by zero or more type parameters $\overline{\tau}$.

Dynamic values in the value semantics are typed by the simple judgement $v : \tau$, whereas update semantics values must be typed with the store μ to type the parts of the value that are stored there. Update semantics values are typed by the judgement $u|\mu : \tau\ [r * w]$, which additionally includes the *heap footprint*, consisting of the sets of read-only (r) and writable (w) pointers the value can contain. We use the same notation for value typing on environments.

This heap footprint annotation is crucial to ensuring that Cogent maintains its uniqueness invariant, as it places constraints on the footprints of subcomponents of a value to rule out aliasing of live pointers. Thus, our theorem of type preservation across evaluation in the update semantics also shows that this invariant is preserved. More details on these constraints are discussed in earlier work [16, 17]. When the heap footprints are not relevant and merely existentially quantified, we shall omit them:

$$u|\mu : \tau \;\equiv\; \exists r\ w.\ u|\mu : \tau\ [r * w]$$

Theorem 2.1 (Type Preservation). *For both update and value semantics:*

- $A; \Gamma \vdash e : \tau \wedge U|\mu : \Gamma \wedge U \vdash e|\mu \Downarrow_{\mathrm{u}} u|\mu' \longrightarrow u|\mu' : \tau$
- $A; \Gamma \vdash e : \tau \wedge \ V : \Gamma \ \wedge \quad V \vdash e \Downarrow_{\mathrm{v}} v \qquad \longrightarrow v : \tau$

This states that the value typing relation for either semantics is preserved across the evaluation relation for well-typed expressions. Because the value typing relations of the update and value semantics are later combined into one refinement relation, which is shown to be preserved across evaluation in Theorem 2.5, type preservation is obtained by simply erasing one of the semantics from Theorem 2.5.

Because the set of types is extensible, the value-typing relation for both semantics must also be extensible. To ensure that the user's extensions to the value-typing relation do not violate the uniqueness invariant, Cogent places a number of proof obligations on abstract types that must be discharged by the user. These requirements are outlined in Section 2.7.

2.4 Frame Requirements

In addition to type preservation, which ensures that each Cogent value is well-formed and does not contain internal aliasing, we must also show that the mutable store μ is in good order throughout evaluation — memory should not be leaked, and programs should not write to memory to which they have no access. These *memory safety* requirements are summed up by Cogent's *frame* relation, which describes how a program may affect the store. Given an input set of writable pointers w_i, an input store μ_i, an output set of pointers w_o and an output store μ_o, the relation, $w_i \mid \mu_i\ \textbf{frame}\ w_o \mid \mu_o$, ensures three properties for any pointer p:

$$\begin{aligned} \text{inertia:} \quad & p \notin w_i \cup w_o \longrightarrow \mu_i(p) = \mu_o(p) \\ \text{leak freedom:} \quad & p \in w_i \longrightarrow p \notin w_o \longrightarrow \mu_o(p) = \bot \\ \text{fresh allocation:} \quad & p \notin w_i \longrightarrow p \in w_o \longrightarrow \mu_i(p) = \bot \end{aligned}$$

Inertia ensures that pointers not in the frame remain unchanged; *leak freedom* ensures that pointers removed from the frame no longer point to anything; and *fresh allocation* ensures that pointers added to the frame were not already used. The frame relation implies that any property of a given value is unaffected by updates to unrelated parts of the heap. The frame relation holds for all Cogent computations, ensuring memory safety along with type safety:

Theorem 2.2 (Preservation and Frame Relation).

$$\begin{aligned} & A; \Gamma \vdash e : \tau \ \wedge\ U|\mu : \Gamma\ [r * w] \ \wedge\ U \vdash e|\mu \Downarrow_{\mathrm{u}} u|\mu' \ \longrightarrow \\ & \quad \exists r'\ w'.\ r' \subseteq r \ \wedge\ w|\mu\ \textbf{frame}\ w'|\mu' \ \wedge\ u|\mu : \tau\ [r' * w'] \end{aligned}$$

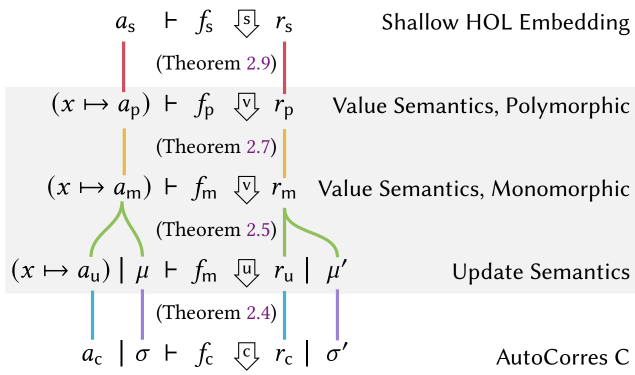

Figure 3. Cogent's semantic levels and refinement theorems.

This states that a well-typed program will evaluate in the update semantics to a well-typed value, *and* that the frame relation holds between the writable pointers of the input environment and the output value. Note that this theorem implies update semantics type preservation (Theorem 2.1), because this theorem too is a simplification of Theorem 2.5.

2.5 Refinement

The overall proof that the C code refines the purely functional shallow embedding in Isabelle/HOL is broken into a number of sub-proofs and translation validation phases. Figure 3 gives an overview of Cogent's refinement theorems for a function $f(x)$ applied to an argument a. The compiler generates four embeddings: a top-level shallow embedding in terms of pure functions; a polymorphic deep embedding of the Cogent program, which is interpreted using the *value* semantics; a monomorphic deep embedding of the Cogent program, which can be interpreted using either the *value* or *update* semantics; and an Isabelle/HOL representation of the C code generated by the compiler, imported into Isabelle/HOL by the C-parser [24] used in the seL4 project [8]. The C-parser generates a deep embedding of C in Isabelle/HOL, and, using AutoCorres [5, 6], is then abstracted to a corresponding state-monadic embedding of the C code in HOL.

Each of these semantic layers is connected by a refinement proof by *forward simulation*: Given a *refinement relation* that relates corresponding values between two layers, we prove that if the more *concrete* (lower in the hierarchy) layer evaluates, then the more *abstract* (higher in the hierarchy) layer, given corresponding inputs, will also evaluate to corresponding outputs. The composition of all these refinements means that any property preserved by refinement, such as functional correctness, proved about all executions of the most *abstract* embedding — the Shallow HOL embedding — will also apply to the most *concrete* embedding, i.e. the C code.

2.5.1 Update to C Refinement. In the first stage, Cogent proves refinement between the C implementation and the deep embedding in the update semantics. AutoCorres imports C as a nondeterministic state-monadic program shallowly embedded in Isabelle/HOL. To make our definitions more symmetrical with those of Cogent, we define a C evaluation relation as follows:

Definition 2.3 (C Evaluation Relation)**.**

$$a_c \vdash \sigma \mid f_c \Downarrow_{c} r_c \mid \sigma' \equiv (r_c, \sigma') \in \mathit{results}(f_c\ a_c\ \sigma) \wedge \neg \mathit{failed}(f_c\ a_c\ \sigma)$$

This states that given an input a_c and C heap σ, the C function f_c evaluates to r_c and an output heap σ' and that no undefined behaviour occurred (indicated by $\neg$*failed*).

The Cogent compiler additionally generates a *value relation* $\mathcal{V}^{\mathrm{u}}_{\mathrm{c}}$ and a *heap relation* $\mathcal{H}^{\mathrm{u}}_{\mathrm{c}}$, which together form the refinement relation for this refinement lemma. Using an automated technique described elsewhere [21], the Cogent proof then automatically discharges this proof obligation on a per-program basis via translation validation, while leaving open proof obligations for the user to discharge for abstract functions implemented in C.

Theorem 2.4 (Update $\Rightarrow$ C refinement)**.** *For any function $f(x)$ with monomorphic Cogent embedding f_m and C embedding f_c, given an argument represented in the update semantics of Cogent as a_u and in C as a_c, we have:*

$$\begin{aligned} &\mathcal{V}^{u}_{c}(a_c, a_u) \wedge \mathcal{H}^{u}_{c}(\sigma, \mu) \wedge a_c \vdash \sigma \mid f_c \Downarrow_{c} r_c \mid \sigma' \longrightarrow \\ &\quad \exists r_u\ \mu'.\ (x \mapsto a_u) \vdash f_m \mid \mu \Downarrow_{u} r_u \mid \mu' \\ &\qquad \wedge \mathcal{V}^{u}_{c}(r_c, r_u) \wedge \mathcal{H}^{u}_{c}(\sigma', \mu') \end{aligned}$$

This states that if the C embedding evaluates to a result, then the corresponding Cogent update semantics will, given corresponding input values and heaps, evaluate to corresponding output values and heaps.

2.5.2 Semantic Correspondence. For the second stage, we must bridge the gap between the *update* semantics and *value* semantics. This is accomplished by Cogent's proof for all well-typed programs that the two semantics correspond. As previously mentioned, this theorem combines both of the value typing relations from the two semantics $v : \tau$ and $u|\mu : \tau\ [r * w]$ into one one combined relation $u|\mu \overset{\mathcal{R}}{\sim} v : \tau\ [r * w]$. In addition to typing both values, u and v, this relation also requires that they represent the same conceptual value. Prior work [16, 17] proves that the update semantics refines the value semantics by proving that this relation is preserved across the evaluation of both, and furthermore that evaluation in the update semantics implies the evaluation in the value semantics. As mentioned in Section 2.4, this also simultaneously proves the frame requirements hold for Cogent code.

Theorem 2.5 (Value $\Rightarrow$ Update refinement)**.** *For any e where $A; \Gamma \vdash e : \tau$, if $U|\mu \overset{\mathcal{R}}{\sim} V : \Gamma\ [r * w]$ and $U \vdash e \mid \mu \Downarrow_{u} u \mid \mu'$, then there exists a value v and pointer sets $r' \subseteq r$ and w' such that $V \vdash e \Downarrow_{v} v$, and $u|\mu' \overset{\mathcal{R}}{\sim} v : \tau\ [r' * w']$ and $w \mid \mu$ **frame** $w' \mid \mu'$.*

This proof is parameterised by the assumption that the same holds for abstract functions. We discuss how to discharge this assumption in Section 3.

2.5.3 Monomorphisation. Recall that the Cogent compiler eliminates polymorphism by *monomorphising*, that is, replacing polymorphic functions with specialised copies, one for each type used in the program. Using template C, the Cogent compiler can do the same for the user-supplied C code. To prove that this elimination of polymorphism preserves correctness we must show that the monomorphic program refines the polymorphic program. This is accomplished by replicating the compiler's monomorphisation operations in Isabelle/HOL as functions on deep embeddings: $\mathcal{M}_e$ to monomorphise expressions and $\mathcal{M}_v$ to monomorphise values. Then, the refinement relation is simply defined:

Definition 2.6 (Monomorphisation Relation)**.**

$$\mathcal{R}^{\mathrm{p}}_{\mathrm{m}}(v_{\mathrm{m}}, v_{\mathrm{p}}) \equiv (v_{\mathrm{m}} = \mathcal{M}_v(N, v_{\mathrm{p}}))$$

where N is a *name mapping*, supplied by the compiler, indicating which set of monomorphic functions correspond to which polymorphic functions.

Proving refinement, then, proceeds on much the same lines as the other layers:

Theorem 2.7 (Polymorphic ⇒ Monomorphic refinement)**.** *For any function $f(x)$ with a polymorphic embedding f_p and argument a_p, let $f_m = \mathcal{M}_e(N, f_p)$ and $a_m = \mathcal{M}_v(a_p)$. Then we have:*

$$\mathcal{R}^{p}_{m}(a_m, a_p) \wedge (x \mapsto a_m) \vdash f_m \Downarrow_{\mathrm{v}} r_m \longrightarrow \exists r_p.\ (x \mapsto a_p) \vdash f_p \Downarrow_{\mathrm{v}} r_p \wedge \mathcal{R}^{p}_{m}(r_m, r_p)$$

2.5.4 Shallow Embedding. Having reached the top of the refinement tower, we must cross the bridge back to shallow embeddings, i.e., our pure HOL functions that serve as an executable specification. Because this embedding is just pure functions, our "evaluation" relation is just function application:

Definition 2.8 (Shallow Evaluation Relation)**.**

$$a_{\mathrm{s}} \vdash f_{\mathrm{s}} \Downarrow_{\mathrm{s}} r_{\mathrm{s}} \equiv (f_{\mathrm{s}}\ a_{\mathrm{s}} = r_{\mathrm{s}})$$

To show refinement, the compiler must once again connect deep and shallow embeddings, just as with the Cogent to C refinement (Theorem 2.4). Therefore, as with C, the compiler automatically produces a proof of this theorem on a per-program basis via translation validation. The compiler generates a refinement relation $\mathcal{R}^{\mathrm{s}}_{\mathrm{p}}$ for all types used in the program, and then proves:

Theorem 2.9 (Shallow ⇒ Polymorphic refinement)**.** *For any function $f(x)$ with a shallow embedding f_s and polymorphic deep embedding f_p, and arguments a_s and a_p respectively, we have:*

$$\mathcal{R}^{s}_{p}(a_p, a_s) \wedge (x \mapsto a_p) \vdash f_p \Downarrow_{\mathrm{v}} r_p \longrightarrow \exists r_s.\ a_s \vdash f_s \Downarrow_{\mathrm{s}} r_s \wedge \mathcal{R}^{s}_{p}(r_p, r_s)$$

2.6 Overall Refinement

To combine all of these refinement phases, we first define a refinement relation for all the layers of refinement:

Definition 2.10 (Combined Relation)**.**

$$\begin{aligned}&\mathcal{R}^{\mathrm{s}}_{\mathrm{c}}(v_{\mathrm{c}}, \sigma, v_{\mathrm{u}}, \mu, v_{\mathrm{m}}, v_{\mathrm{p}}, v_{\mathrm{s}}, \tau, r, w)\\ &\quad \equiv \mathcal{R}^{\mathrm{s}}_{\mathrm{p}}(v_{\mathrm{p}}, v_{\mathrm{s}}) \wedge \mathcal{R}^{\mathrm{p}}_{\mathrm{m}}(v_{\mathrm{m}}, v_{\mathrm{p}})\\ &\quad \wedge v_{\mathrm{u}}|\mu \overset{\mathcal{R}}{\sim} v_{\mathrm{m}} : \tau\ [r * w]\\ &\quad \wedge \mathcal{V}^{\mathrm{u}}_{\mathrm{c}}(v_{\mathrm{c}}, v_{\mathrm{u}}) \wedge \mathcal{H}^{\mathrm{u}}_{\mathrm{c}}(\sigma, \mu)\end{aligned}$$

Theorem 2.11 (Combined Refinement)**.** *Given a function $f(x)$ with embeddings f_c, f_m, f_p, f_s; argument value a_c, a_u, a_m, a_p, a_s; heap σ and store μ, we show:*

$$\begin{aligned}&\mathcal{R}^{s}_{c}(a_c, \sigma, a_u, \mu, a_m, a_p, a_s, \tau, r, w) \wedge a_c \vdash f_c \mid \sigma \Downarrow_{\mathrm{c}} v_c \mid \sigma' \longrightarrow\\ &\quad \exists\, r'\ w'\ \mu'.\ r' \subseteq r \wedge w|\mu\ \textbf{frame}\ w'|\mu'\\ &\quad\quad \wedge \exists\, v_u\ v_m\ v_p\ v_s.\\ &\quad\quad\quad \wedge (x \mapsto a_u) \vdash f_m|\mu \Downarrow_{\mathrm{u}} v_u|\mu'\\ &\quad\quad\quad \wedge (x \mapsto a_m) \vdash f_m \Downarrow_{\mathrm{v}} v_m\\ &\quad\quad\quad \wedge (x \mapsto a_p) \vdash f_p \Downarrow_{\mathrm{v}} v_p\\ &\quad\quad\quad \wedge a_s \vdash f_s \Downarrow_{\mathrm{s}} v_s\\ &\quad\quad\quad \wedge \mathcal{R}^{s}_{c}(v_c, \sigma', v_u, \mu', v_m, v_p, v_s, \tau', r', w')\end{aligned}$$

Because this theorem encompasses *all* levels of refinement as well as type preservation and the frame requirements, it is sufficient to prove this theorem about (each embedding of) each abstract function, where the semantics of the deep embeddings f_{p} and f_{m} is given by the user-supplied environments ξ_v and ξ_u. A proof of this theorem for an abstract function is sufficient to integrate its verification with the Cogent verification chain.

2.7 Requirements on Abstract Types

Theorem 2.11 encompasses all requirements the Cogent FFI places on abstract functions, but users can also provide abstract *types*, extending the dynamic value typing rules as they see fit. Therefore, Cogent imposes several constraints on the value typing judgements which ensure that key type system invariants such as memory safety are maintained for abstract types.

Consider an abstract type of the form "$\mathsf{A}\ \overline{\tau}$", where A is the name of the abstract type and $\overline{\tau}$ is a list a type parameters. Because it is not surrounded with the !-operator, it is *linear* and therefore writable. The requirements of the user-defined value typing relation are as follows.

Definition 2.12 (Value Typing Requirements)**.**

$$\begin{aligned}\text{bang}_{\mathrm{v}}:&\quad v : \mathsf{A}\ \overline{\tau} \longrightarrow v : (\mathsf{A}\ \overline{\tau!})!\\ \text{bang}_{\mathrm{u}}:&\quad u|\mu : \mathsf{A}\ \overline{\tau}\ [r * w] \longrightarrow u|\mu : (\mathsf{A}\ \overline{\tau!})\ [r \cup w * \emptyset]\\ \text{no-alias}:&\quad u|\mu : \mathsf{A}\ \overline{\tau}\ [r * w] \longrightarrow r \cap w = \emptyset\\ \text{valid}:&\quad u|\mu : \mathsf{A}\ \overline{\tau}\ [r * w] \longrightarrow \forall p \in r \cup w.\ \mu(p) \neq \bot\\ \text{frame}:&\quad u|\mu_i : \mathsf{A}\ \overline{\tau}\ [r * w] \longrightarrow w_i|\mu_i\ \textbf{frame}\ w_o|\mu_o\\ &\quad \longrightarrow (r \cup w) \cap w_i = \emptyset \longrightarrow u|\mu_o : \mathsf{A}\ \overline{\tau}\ [r * w]\end{aligned}$$

The *bang* rules ensure that abstract values respect the !-operator, i.e., when ! is applied to a value, the value becomes read-only; *no-alias* ensures that there is no aliasing of writable pointers by readable pointers within a value; *valid* enforces that all pointers are valid, i.e., point to values; and *frame* ensures that an abstract value is unchanged if it is not part of the store that is currently being modified. For a read-only type $(\mathsf{A}\ \overline{\tau})!$, the requirements are the same, save that the writable pointer set w must also be empty:

$$\text{read-only:} \quad u|\mu : (\mathsf{A}\ \overline{\tau})!\ [r * w] \longrightarrow w = \emptyset$$

2.8 Summary of Requirements

As can be seen from the previous sections, proof engineers must provide a number of implementations, abstractions, and proofs for each function and type they import. We briefly summarise these here. For each abstract type imported from C, the proof engineer simply needs to prove the requirements of Definition 2.12. For each imported foreign function f, proof engineers must define a version of the function for all semantic levels in the Cogent hierarchy. That is, they must define a C implementation, an update semantics ($\xi_{\mathsf{u}}\ f$), a monomorphic value semantics ($\xi_{\mathsf{m}}\ f$), a polymorphic value semantics ($\xi_{\mathsf{p}}\ f$), and a shallow embedding in HOL. Once all embeddings have been provided, overall refinement (Theorem 2.11) must be proven for the foreign function f, which necessitates proofs of all of our other preservation and refinement theorems:

- Type preservation in the update semantics (Theorem 2.2): required by all functions that call f to prove type preservation (Theorem 2.2), and by f and all functions that call it to prove refinement (Theorems 2.4 and 2.5).
- Type preservation in the monomorphic value semantics (Theorem 2.1): required by all functions that call f to prove type preservation (Theorem 2.1), and by f and all functions that call it to prove refinement (Theorems 2.5 and 2.7).
- Refinement from the update semantics to C (Theorem 2.4): required by all functions that call f to prove refinement (Theorem 2.4).
- Refinement from the value semantics to the update semantics (Theorem 2.5): required by all functions that call f to prove refinement (Theorem 2.5).
- Refinement from the polymorphic to monomorphic value semantics (Theorem 2.7): required by all functions that call f to prove refinement (Theorem 2.7).
- Refinement from the shallow embedding to the polymorphic value semantics (Theorem 2.9): required by all functions that call f to prove refinement (Theorem 2.9).

$$
\begin{aligned}
&\mathsf{length_s} : [a] \longrightarrow \mathtt{word32}\\
&\mathsf{length_s}\ xs = \mathsf{of\_nat}\ (\mathsf{List.length}\ xs)\\
&\mathsf{get_s} : [a] \longrightarrow \mathtt{word32} \longrightarrow a\\
&\mathsf{get_s}\ xs\ i\ d = \textbf{if}\ \mathsf{unat}\ i < \mathsf{List.length}\ xs\\
&\qquad\qquad \textbf{then}\ xs\ !\ (\mathsf{unat}\ i)\ \textbf{else}\ d\\
&\mathsf{put_s} : [a] \longrightarrow \mathtt{word32} \longrightarrow a \longrightarrow [a]\\
&\mathsf{put_s}\ xs\ i\ v = xs[\mathsf{unat}\ i := v]\\
&\mathsf{fold_s} : (b \longrightarrow a \longrightarrow c \longrightarrow b) \longrightarrow b \longrightarrow [a]\\
&\quad \longrightarrow \mathtt{word32} \longrightarrow \mathtt{word32} \longrightarrow c \longrightarrow b\\
&\mathsf{fold_s}\ f\ acc\ xs\ frm\ to\ obs =\\
&\quad \mathsf{List.fold}\ (\lambda acc\ el.\ f\ acc\ el\ obs)\ acc\ (\mathsf{slice}\ frm\ to\ xs)\\
&\mathsf{mapAccum_s} : (b \longrightarrow a \longrightarrow c \longrightarrow (a, b)) \longrightarrow b \longrightarrow [a]\\
&\qquad\qquad \longrightarrow \mathtt{word32} \longrightarrow \mathtt{word32} \longrightarrow c \longrightarrow ([a], b)\\
&\mathsf{mapAccum_s}\ f\ acc\ xs\ frm\ to\ obs =\\
&\quad \textbf{let}\ (xs', acc') = \mathsf{List.fold}\\
&\qquad (\lambda el\ (xs, acc).\ \textbf{let}\ (el', acc') = f\ acc\ el\ obs\\
&\qquad\qquad\qquad \textbf{in}\ (xs\ @\ [el'], acc'))\\
&\qquad (\mathsf{slice}\ frm\ to\ xs)\ ([], acc)\\
&\quad \textbf{in}\ (\mathsf{take}\ frm\ xs\ @\ xs'\ @\ \mathsf{drop}\ (\mathsf{max}\ frm\ to)\ xs, acc')
\end{aligned}
$$

Figure 4. Functional correctness specification of the array operations in Isabelle/HOL.

3 Arrays

Arrays in Cogent are stored on the heap with elements having any *unboxed* type. Arrays of pointers are a separate data structure in Cogent, as their interface is complicated by the uniqueness type system. We verify five array operations: the length, get, and put functions in Figure 1, and the fold and mapAccum iterators in Figure 2.

3.1 Specification and Implementation

Our operations on arrays are specified as Isabelle functions on lists. The specification for the array functions presented in Figures 1 and 2 is provided in Figure 4. Most array operations have obvious analogues in Isabelle/HOL's list library. The library functions unat and of-nat convert words and natural numbers, take *n xs* and drop *n xs* return and remove the first *n* elements of the list *xs* respectively, slice *n m xs* returns the sublist that starts at index *n* and ends at index *m* of the list *xs*, ! returns the i^{th} element of a list and @ appends two lists.

The operations get and mapAccum do not have straightforward analogues in the Isabelle library. mapAccum is not part of the library at all, and must be implemented in terms of fold, whereas get behaves differently to its corresponding list operation: our implementation of get is not undefined when the given index is out of bounds, but instead returns the provided default value.

In Figure 5, we present a version of the template C implementation of arrays with some syntactic simplifications for presentation. Quoted type parameters that are later instantiated to concrete Cogent types, such as **T** or **O**, are highlighted in blue. Generated C structure types for Cogent

```
struct WArrayT {
  u32 len;
  T* vals;
};
u32 length(WArrayT *arr) { return arr→len; }
T get(WArrayT *arr, u32 i, T def) {
  if(i < arr→len) { return arr→vals[i]; }
  return def;
}
WArrayT *put(WArrayT *arr, u32 i, T v) {
  if(i < arr→len) { arr→vals[i] = v; }
  return arr;
}
A fold(fid f, A acc, WArrayT *arr, u32 frm, u32 to, O obsv) {
  u32 i, e;
  e = arr→len;
  if(to < e) { e = to; }
  for(i = frm; i < e; i++)   {acc = dispatch_fold(f, arr→vals[i], acc, obsv); }
  return acc;
}
ArrayAcc mapAccum(fid f, A acc, WArrayT *arr, u32 frm, u32 to, O obsv) {
  u32 i, e;
  e = arr->len;
  if(to < e) { e = to; }
  for(i = frm; i < e; i++) {
    ElemAcc ea = dispatch_mapAccum(f, arr→vals[i], acc, obsv);
    arr→vals[i] = ea.elem;
    acc = ea.acc;
  }
  ArrayAcc ret = {.arr = arr, .acc = acc};
  return ret;
}
```

Figure 5. C implementation of key operations on arrays.

records and tuples such as *ArrayAcc*, which normally have compiler-generated names, are written with human-friendly names in grey. The *dispatch* functions highlighted in red are how Cogent deals with higher-order functions: Because the C-parser semantics do not accommodate function pointers, Cogent instead assigns a unique identifier to each function at compile time, and defines a dispatch function for each function type. This function takes a function identifier as an argument and calls the function that corresponds to that identifier.

3.2 Proving Refinement

As previously mentioned, to verify our abstract C library, we must additionally provide Isabelle/HOL abstractions of our C implementation that can connect with the automatically-generated Cogent embeddings. Specifically, we must provide Isabelle/HOL abstractions for each abstract function (i.e., put, get, etc.), as well as a combined value-update-type correspondence relation, analogous to the refinement relation for Theorem 2.5, for each abstract type (i.e., *WArray***T**).

3.2.1 Abstractions. We extend the definition of Cogent values with a new constructor for arrays. In the update semantics, arrays take the form UWA τ *len* p, where τ is the type of the array elements, *len* is the length of the array, and p is a pointer to the first element of the array. This is quite similar to the representation used in C, where a struct containing the length and a pointer is used (see Figure 5). We additionally store the type in the Cogent value so that the same form of Cogent value can be used for all the different structs generated by the Cogent compiler from the C template. In the value semantics, however, arrays take the form VWA τ xs, where xs is simply an Isabelle/HOL list of Cogent values, similar to the representation used in the shallow embedding (see Figure 4).

The various array function abstractions supplied to Cogent are defined as input-output relations that are later interpreted as functions, as shown in Figure 6. These abstractions are derived directly from the axiomatisation used in the verification of BilbyFs. In the update semantics (Figure 6a), the input is a pair of the Cogent store before the array operation and the argument(s) to the function, and the output is a pair of the store after the operation is complete and the return value.

Note that the definitions for fold and mapAccum are recursive in two ways. The most obvious is the direct structural recursion on the array indices, which is straightforward for Isabelle to show terminates. The other form of recursion is the mutual recursion with the Cogent semantics (note that fold and mapAccum invoke the Cogent semantics to evaluate the argument function). This is because the Cogent semantics is itself parameterised by an environment (ξ_u and ξ_v) containing the abstractions we are currently defining. We cannot define these embeddings and the Cogent semantics simultaneously as is normally done for mutual definitions, however, since our abstractions are defined by users later in the process after the semantics of Cogent have already been defined. Thus, we cannot close the recursion before users have provided these definitions. Therefore we must impose an additional constraint[2] that an n-order abstract function can only call a function of an order less than n. This forces us to prove refinement for all functions of order $n-1$ or less, before we can prove refinement for an abstract function of order n. By imposing this ordering, we can iteratively build up these environments in a staged way. This constraint is not burdensome, and is already satisfied by all Cogent codebases (see Rizkallah et al. [21]).

Since Cogent supports polymorphism, we in fact need to provide two abstractions for our operations in the value semantics corresponding to the two value semantics layers in our refinement chain, monomorphic and polymorphic. Because our abstractions (and even our implementation) are entirely parametric in the element type, the abstractions that are shown in Figure 6b can be used for both the monomorphic and the polymorphic layers, which, as we will see, trivialises the refinement proof for the monomorphisation layer of the hierarchy.

3.2.2 Value Typing. Unlike the approach for native Cogent values, where the value typing relations are defined as erasures of the value-update refinement relation, we shall

[2] This constraint is not a proof obligation, but merely a requirement for our framework to work automatically.

$$
\begin{aligned}
&\mathsf{length_u}\ (\mu, x)\ (\mu', y) = \\
&\quad \exists \tau\ len\ p.\ (\mu\ x = \mathtt{UWA}\ \tau\ len\ p) \land (y = len) \land (\mu = \mu') \\
&\mathsf{get_u}\ (\mu, (x, i, d))\ (\mu', y) = \\
&\quad \exists \tau\ len\ p.\ (\mu\ x = \mathtt{UWA}\ \tau\ len\ p) \\
&\qquad \land (i < len \longrightarrow \mu\ (p{+}(\mathsf{size}\ t){\times}i) = y) \\
&\qquad \land (i \geq len \longrightarrow y = d) \land (\mu = \mu') \\
&\mathsf{put_u}\ (\mu, (x, i, v))\ (\mu', y) = \\
&\quad \exists \tau\ len\ p.\ (\mu\ x = \mathtt{UWA}\ \tau\ len\ p) \\
&\qquad \land (i < len \longrightarrow \mu(p{+}(\mathsf{size}\ \tau){\times}i \mapsto v) = \mu') \\
&\qquad \land (i \geq len \longrightarrow \mu = \mu') \land (x = y) \\
&\mathsf{fold_u}\ (\mu_1, (f, acc, x, s, e, obs))\ (\mu_3, y) = \\
&\quad \exists \tau\ len\ p.\ (\mu_1\ x = \mathtt{UWA}\ \tau\ len\ p) \\
&\qquad \land (s < len \land s < e \longrightarrow \\
&\qquad\quad \exists v\ acc'\ \mu_2.\ \mu_1\ (p{+}(\mathsf{size}\ \tau){\times}s) = v \\
&\qquad\qquad \land [a \mapsto (v, acc, obs)] \vdash f\, a \mid \mu_1 \Downarrow_{u} acc' \mid \mu_2 \\
&\qquad\qquad \land \mathsf{fold_u}\ (\mu_2, (f, acc', x, s{+}1, e, obs))\ (\mu_3, y)) \\
&\qquad \land (s \geq len \lor s \geq e \longrightarrow \mu_1 = \mu_3 \land y = acc) \\
&\mathsf{mapAccum_u}\ (\mu_1, (f, acc, x, s, e, obs))\ (\mu_4, y) = \\
&\quad \exists \tau\ len\ p.\ (\mu_1\ x = \mathtt{UWA}\ \tau\ len\ p) \\
&\qquad \land (s < len \land s < e \longrightarrow \\
&\qquad\quad \exists v\ v'\ acc'\ \mu_2\ \mu_3.\ \mu_1\ (p{+}(\mathsf{size}\ \tau){\times}s) = v \\
&\qquad\qquad \land [a \mapsto (v, acc, obs)] \vdash f\, a \mid \mu_1 \Downarrow_{u} (v', acc') \mid \mu_2 \\
&\qquad\qquad \land \mu_3 = \mu_2(p{+}(\mathsf{size}\ t){\times}s \mapsto v') \\
&\qquad\qquad \land \mathsf{mapAccum_u}\ (\mu_3, (f, acc', x, s{+}1, e, obs))\ (\mu_4, y)) \\
&\qquad \land (s \geq len \lor s \geq e \longrightarrow \mu_1 = \mu_4 \land y = (x, acc))
\end{aligned}
$$

(a) C abstractions for the *update* semantics.

$$
\begin{aligned}
&\mathsf{length_v}\ x\ y = \\
&\quad \exists \tau\ xs.\ (x = \mathtt{VWA}\ \tau\ xs) \land (\mathsf{List.length}\ xs = \mathsf{unat}\ y) \\
&\mathsf{get_v}\ (x, i)\ y = \\
&\quad \exists \tau\ xs.\ (x = \mathtt{VWA}\ \tau\ xs) \\
&\qquad \land (\mathsf{unat}\ i < \mathsf{List.length}\ xs \longrightarrow xs\ !\ i = y) \\
&\qquad \land (\mathsf{unat}\ i \geq \mathsf{List.length}\ xs \longrightarrow y = 0) \\
&\mathsf{put_v}\ (x, i, v)\ y = \\
&\quad \exists \tau\ xs.\ (x = \mathtt{VWA}\ \tau\ xs) \land (y = \mathtt{VWA}\ \tau\ xs[\mathsf{unat}\ i := v]) \\
&\mathsf{fold_v}\ (f, acc, x, s, e, obs)\ y = \\
&\quad \exists \tau\ xs.\ (x = \mathtt{VWA}\ \tau\ xs) \\
&\qquad \land (\mathsf{unat}\ s < \mathsf{List.length}\ xs \land s < e \longrightarrow \\
&\qquad\quad \exists v\ acc'.\ (xs\ !\ s = v) \\
&\qquad\qquad \land [a \mapsto (v, acc, obs)] \vdash f\, a \Downarrow_{v} acc' \\
&\qquad\qquad \land \mathsf{fold_v}\ (f, acc', x, s{+}1, e, obs)\ y) \\
&\qquad \land (\mathsf{unat}\ s \geq \mathsf{List.length}\ xs \lor s \geq e \longrightarrow y = acc) \\
&\mathsf{mapAccum_v}\ (f, acc, x, s, e, obs)\ y = \\
&\quad \exists \tau\ xs.\ (x = \mathtt{VWA}\ \tau\ xs) \\
&\qquad \land (\mathsf{unat}\ s < \mathsf{List.length}\ xs \land s < e \longrightarrow \\
&\qquad\quad \exists v\ v'\ acc'\ x'.\ xs\ !\ s = v \\
&\qquad\qquad \land [a \mapsto (v, acc, obs)] \vdash f\, a \Downarrow_{v} (v', acc') \\
&\qquad\qquad \land x' = \mathtt{VWA}\ \tau\ xs[\mathsf{unat}\ s := v'] \\
&\qquad\qquad \land \mathsf{mapAccum_v}\ (f, acc', x', s{+}1, e, obs)\ y) \\
&\qquad \land (\mathsf{unat}\ s \geq \mathsf{List.length}\ xs \lor s \geq e \longrightarrow y = (x, acc))
\end{aligned}
$$

(b) C abstractions for the *value* semantics.

Figure 6. Cogent-compatible abstractions of C operations on arrays

define individual value-typing relations for arrays in the two semantics, and then combine them into a refinement relation in Definition 3.5

We define these typing relations with two equations, one for the writable arrays `Array` τ, and one for read-only arrays (`Array` τ)!. In the value semantics, these two types are identical, merely requiring that the list elements are well-typed:

Definition 3.1 (Array: Value Semantics Value Typing)**.**

$$
\begin{aligned}
&\mathtt{VWA}\ \tau\ xs : \mathtt{Array}\ \tau \quad\ \ \equiv (\forall i < \mathsf{List.length}\ xs.\ xs\ !\ i : \tau) \\
&\mathtt{VWA}\ \tau\ xs : (\mathtt{Array}\ \tau)! \equiv ((\mathtt{VWA}\ \tau\ xs) : \mathtt{Array}\ \tau)
\end{aligned}
$$

For the update semantics, we define an auxiliary predicate *okay*(`UWA` τ *len* p) which states that each of the values in the array (located at successive pointers starting at p) is well typed, as well as a necessary condition on *len* to ensure that our pointer arithmetic will not result in overflow. This predicate is used in the typing relation for both the writable and read-only array types. The heap footprint $[r * w]$ must consist of not just the pointer p but all of the successive pointers to each array element, because all of these memory locations are contained in the array, and ownership of all of them is passed along with the array. Because the array contains only unboxed types, we know that there are no other pointers in the heap footprint. For the read-only array type, r contains the heap footprint and w is empty, and vice versa for the writable array type.

Definition 3.2 (Array: Update Semantics Value Typing)**.**

$$
\begin{aligned}
&okay\ (\mathtt{UWA}\ \tau\ len\ p) \equiv (\mathsf{unat}\ len \times \mathsf{size}\ \tau \leq \mathtt{max\_word}) \\
&\land (\forall i < len.\ \exists v.\ \mu\ (p + \mathsf{size}\ \tau \times i) = v\ \ \land\ \ v|\mu : \tau\ [\emptyset * \emptyset]) \\
&\mathtt{UWA}\ \tau\ len\ p\ |\mu : (\mathtt{Array}\ \tau)!\ [r * w] \equiv okay\ (\mathtt{UWA}\ \tau\ len\ p) \\
&\qquad\qquad\qquad \land (r = \{p + i \mid \forall i.\ i < len\} \land w = \emptyset) \\
&\mathtt{UWA}\ \tau\ len\ p\ |\mu : (\mathtt{Array}\ \tau)\ [r * w] \equiv okay\ (\mathtt{UWA}\ \tau\ len\ p) \\
&\qquad\qquad\qquad \land (w = \{p + i \mid \forall i.\ i < len\} \land r = \emptyset)
\end{aligned}
$$

where `max_word` is $2^{32} - 1$ (as we are using 32-bit pointers).

Recall that these the value typing relations for abstract types must satisfy the constraints of Definition 2.12. Because arrays only have elements which are of unboxed types, these constraints are trivial to discharge.

3.2.3 Refinement Relations. Just as Cogent's typing relations and semantics are extended by our rules for arrays, so too are the various refinement relations in each layer of the semantics. Because, as previously mentioned, the C implementation of our arrays bears a strong resemblance to our update semantics values, and our value semantics values bear a strong resemblance to our Isabelle/HOL list representation, the value relations $\mathcal{V}^{u}_{c}$ and $\mathcal{R}^{s}_{p}$ for arrays are very simple. In the former, the two a related if the length values and the pointer values are equal. In the latter, the two are related if the length of the lists are the same and the elements are pairwise related.

Definition 3.3 (U32 Array: Update ⇒ C Value Relation).

$$\mathcal{V}^{\mathsf{u}}_{\mathsf{c}}(x_{\mathsf{c}}, \mathsf{UWA}\ \mathsf{U32}\ len_{\mathsf{u}}\ p_{\mathsf{u}}) \equiv (len_{\mathsf{u}} = \mathtt{len}_{\mathsf{c}}\ x_{\mathsf{c}}\ \wedge\ p_{\mathsf{u}} = \mathtt{arr}_{\mathsf{c}}\ x_{\mathsf{c}})$$

where $\mathtt{len}_{\mathsf{c}}$ and $\mathtt{arr}_{\mathsf{c}}$ are the struct projections generated by AutoCorres for the array type.

Definition 3.4 (Array: Shallow ⇒ Polymorphic Relation).

$$\mathcal{R}^{\mathsf{s}}_{\mathsf{p}}(xs_{\mathsf{s}}, \mathsf{VWA}\ \tau\ xs_{\mathsf{p}}) \equiv (\mathsf{length}\ xs_{\mathsf{p}} = \mathsf{length}\ xs_{\mathsf{s}}) \wedge (\forall i < \mathsf{length}\ xs_{\mathsf{p}}.\ \mathcal{R}^{\mathsf{s}}_{\mathsf{p}}(xs_{\mathsf{s}}\ !\ i, xs_{\mathsf{p}}\ !\ i))$$

Because we use the same abstractions for both monomorphic and polymorphic layers, the refinement relation $\mathcal{R}^{\mathsf{p}}_{\mathsf{m}}$ is just equality and its refinement theorem is trivial.

Lastly, it remains to define the value relation for arrays between the two Cogent semantics, update and value. As previously mentioned, we make use of the two value typing relations here with additional conditions to pairwise relate the corresponding elements of the two values:

Definition 3.5 (Array: Value (Mono) ⇒ Update Relation).

$$\begin{aligned}&\mathsf{UWA}\ t\ len_{\mathsf{u}}\ p_{\mathsf{u}} \mid \mu \overset{\mathcal{R}}{\sim} \mathsf{VWA}\ t\ xs_{\mathsf{m}} :\ \tau\ [r * w] \equiv \\ &\quad (\mathsf{unat}\ len_{\mathsf{u}} = \mathsf{length}\ xs_{\mathsf{m}}) \\ &\quad \wedge (\forall i < len_{\mathsf{u}}.\ \exists v_{\mathsf{u}}.\ \mu(p_{\mathsf{u}} + (\mathsf{size}\ t) \times i) = v_{\mathsf{u}} \\ &\qquad\quad \wedge v_{\mathsf{u}} \mid \mu \overset{\mathcal{R}}{\sim} (xs_{\mathsf{m}}\ !\ \mathsf{unat}\ i) :\ t\ [\emptyset * \emptyset]) \\ &\quad \wedge (\mathsf{VWA}\ t\ xs_{\mathsf{m}} : \tau)\ \wedge\ (\mathsf{UWA}\ t\ len_{\mathsf{u}}\ p_{\mathsf{u}} \mid \mu : \tau\ [r * w])\end{aligned}$$

Note the heap footprints of elements are always empty, as the array can only contain unboxed values.

3.2.4 Refinement.

Now that we have all our abstractions, value typing and refinement relations, we have all the ingredients we need to prove refinement for our array operations.

The theorems structurally resemble the refinement theorems presented in Section 2.5. For first order functions length, get and put, the proofs tend to follow easily from the definition of their abstractions, implementation, refinement relation and value typing relation — the creativity is largely in the definitions, not the proofs. This is because we want the proofs to be easily automatable in future. Nonetheless we shall sketch the proofs for our put operation, specifically for arrays of U32, as an illustrative example.

We first show that the update semantics abstraction is refined by the embedding of the C implementation that is automatically generated by AutoCorres. This appears similar to Theorem 2.4, but instead of invoking the Cogent update semantics we instead appeal to our abstract function from the environment $\xi_{\mathsf{u}}(\mathsf{put}_{\mathsf{U32}})$:

Theorem 3.6 (Verifying put: Update ⇒ C refinement). *Where the C embedding of* $\mathsf{put}_{\mathsf{U32}}$ *is* put_c*:*

$$\begin{aligned}&\mathcal{V}^{\mathsf{u}}_{c}(a_c, a_u) \wedge \mathcal{H}^{\mathsf{u}}_{c}(\sigma, \mu) \wedge a_c \vdash \sigma \mid put_c \overset{\text{c}}{\Downarrow} r_c \mid \sigma' \longrightarrow \\ &\quad \exists \mu'\ r_u.\ \xi_u(\mathsf{put}_{\mathsf{U32}})(\mu, a_u) = (\mu', r_u) \\ &\qquad \wedge \mathcal{V}^{\mathsf{u}}_{c}(r_c, r_u) \wedge \mathcal{H}^{\mathsf{u}}_{c}(\sigma', \mu')\end{aligned}$$

Proof. Recall that the argument to put for arrays of 32-bit words is a tuple that contains an index, the array, and a 32-bit word to write to it. We take cases on the index. In the case that the index is out of bounds, both the abstraction (Figure 6a) and the implementation (Figure 5) return the array unmodified with the store unmodified as well, and so the theorem is trivial. In the case where the index is within bounds, our value relation $\mathcal{V}^{\mathsf{u}}_{\mathsf{c}}$ on the arguments implies that the two argument words and indices are the same. Writing the same value to the same index within bounds only writes corresponding values to the same store locations, so it follows that our heap relation $\mathcal{H}^{\mathsf{u}}_{\mathsf{c}}$ is preserved. As it is destructively updated, the actual location of the array in memory is not changed, so the relation $\mathcal{V}^{\mathsf{u}}_{\mathsf{c}}$ is trivially preserved to the output array. □

For the next level up in the refinement hierarchy, we must show refinement from our value semantics abstraction (Figure 6b) to our update semantics abstraction (Figure 6a). Even though this refinement step is below monomorphisation in our hierarchy, our abstractions for put are agnostic to the element type of the array, so we can generalise the proof to arrays of any element type. The theorem resembles Theorem 2.5, but with the Cogent semantics replaced with our supplied abstractions in ξ_{v} and ξ_{u}.

Theorem 3.7 (Verifying put: Value ⇒ Update refinement). *For an element type* t*, if* $a_u|\mu \overset{\mathcal{R}}{\sim} a_m :\ (\mathtt{Array}\ t, \mathtt{U32}, t)\ [r * w]$ *and* $\xi_u(\mathsf{put}_t)(\mu, a_u) = (\mu', v_u)$*, then there exists a value* v_m *and pointer sets* $r' \subseteq r$ *and* w' *such that* $\xi_{\mathsf{v}}(\mathsf{put}_t)(a_m) = v_m$*, and* $v_u|\mu' \overset{\mathcal{R}}{\sim} v_m :\ \mathtt{Array}\ t\ [r' * w']$ *and* $w \mid \mu$ ***frame*** $w' \mid \mu'$*.*

Proof. We also prove this by cases on the index. In the case where the index is out of bounds, we trivially have correspondence. In the case where the index is within bounds, we prove that modifying the element at the given index from the pointer on the store is equivalent to modifying the element at the given index in the corresponding list. To prove this, we need to show that there is a one-to-one mapping between store addresses of elements to list indices. This is why we include in our typing relation that the array element addresses do not overflow the heap (Definition 3.2). Since this tells us the array cannot wrap around itself, each element in the array has a unique address, giving us our mapping. We also need to show that that the frame conditions are satisfied, but because our implementation is memory safe, these follow easily from our definitions. □

Next, we must show refinement from the polymorphic layer to the monomorphic layer, but because put is a first-order function (neither taking functions as arguments nor returning them), the value relations for its arguments and return values simplify to equality. Furthermore, as our Cogent abstractions for monomorphic and polymorphic layers are identical, the proof of refinement is trivialised to showing that identical functions will give equal results given equal inputs.

Monomorphisation thus easily dispatched, we must now make the final shift to the specification level (Figure 4). We must prove a theorem analogous to Theorem 2.9. Note that this theorem is also generic for any element type:

Theorem 3.8 (Verifying put: Shallow ⇒ Polymorphic Value)**.** *The shallow embedding of* put *is called* put_s. *Given arguments* a_s *and* a_p, *we have:*

$$\mathcal{R}^s_p(a_p, a_s) \wedge \xi_{\mathsf{v}}(\mathsf{put})(a_p) = r_p \longrightarrow \exists r_s.\ a_s \vdash \mathsf{put}_s \overset{s}{\Downarrow} r_s \wedge \mathcal{R}^s_p(r_p, r_s)$$

Proof. Follows from the definitions of $\mathsf{put}_{\mathsf{v}}$ (in ξ_{v}) and $\mathsf{put}_{\mathsf{s}}$, as well as the value relation from Definition 3.4. □

The above theorems are all we need to compose the correctness of put with Cogent's refinement hierarchy. For higher-order functions fold and mapAccum the proofs are broadly similar, but slightly complicated by the presence of functions as arguments. Our theorems assume that type preservation and refinement hold for all of their argument functions — an assumption that is discharged by the Cogent compiler (for Cogent functions) or by manual proofs (for C functions). As always with looping functions, the majority of the proof effort was concentrated on proving that loop invariants are maintained, and not on any aspect of the Cogent framework.

4 Generic Loops

Aside from arrays and their associated iterators and operations, the most commonly used library functions in BilbyFs are generic loop functions. These are used to write loops that do not simply iterate over a particular data structure, and are often used to accommodate non-standard search patterns over data structures. For example, in Section 5.2 we shall use such a function in a binary search.

We shall verify the function repeat, which is given the following type signature in Cogent:

$$\mathsf{repeat} : (\mathtt{U32}, (a, b!) \rightarrow \mathtt{Bool}, (a, b!) \rightarrow a, a, b!) \rightarrow a$$

The expression repeat *n stop step acc obs* operates on some mutable state *acc* (of linear type) and some observer data *obs* (of read-only type), and runs the loop body *step* on it at most *n* times, or until *stop* returns true. Figure 7 gives the Isabelle/HOL shallow embedding and Figure 8 gives the template C implementation. Figure 9 gives the Cogent-compatible embeddings for the environments ξ_u and ξ_{v}. Note that these must invoke the Cogent semantics twice, once to evaluate each of the argument functions. Discharging the required proof obligations connecting all of these embeddings and maintaining Cogent's invariants is even more straightforward than for other higher-order functions such as mapAccum, as here we do not even need to consider custom data structures such as arrays. This means that we can even define a polymorphic abstraction of the C code compatible with AutoCorres, enabling us to prove the entire refinement chain polymorphically and thereby largely eliminate the boilerplate of multiple type instantiations.

$$\begin{aligned}
&\mathsf{repeat_s} : \mathtt{nat} \rightarrow (a \rightarrow b \rightarrow \mathtt{bool}) \rightarrow (a \rightarrow b \rightarrow a) \\
&\qquad \rightarrow a \rightarrow b \rightarrow a \\
&\mathsf{repeat_s}\ 0\ _\ _\ acc\ _ = acc \\
&\mathsf{repeat_s}\ (\mathsf{Suc}\ n)\ f\ g\ acc\ obsv = \mathbf{if}\ (f\ acc\ obsv) \\
&\quad \mathbf{then}\ acc\ \mathbf{else}\ \mathsf{repeat_s}\ n\ f\ g\ (g\ acc\ obsv)\ obsv
\end{aligned}$$

Figure 7. Shallow embedding of the generic loop

5 Composing Verification of Cogent and C

Now that we have demonstrated how to prove the obligations placed on C code, we shall illustrate how to integrate these proofs with proofs about Cogent code. Firstly, as a simple example, we return to the sum function presented in Figure 2.

After compiling this program and linking it to our C library, the compiler produces a refinement theorem similar to Theorem 2.11, however the phases that are generated per-program via translation validation such as the final refinement to C leave open proof obligations for the user to discharge about abstract functions[3]. In our sum example, Cogent will generate obligations about length and about fold. The obligation about length is exactly our C refinement theorem for length (the length analogue of Theorem 3.6), and the obligation for fold is an instance of our theorem for fold: the obligation requires showing refinement under the assumption that the argument function is add, whereas our theorem is generically proven for any function that maintains Cogent's invariants and refinement. Because add is defined in Cogent, Cogent generates the required theorem for the argument function for us, allowing us to easily discharge this obligation.

We additionally must instantiate our sets of abstract values and types with arrays, and the environments ξ_{v} and ξ_{u} with our abstractions from Figure 6. This instantiation requires us to additionally provide proofs similar to that of Theorem 3.7 for each function, as well as proofs that all the type conditions from Section 2.7. These proofs ultimately connect our C code to the generated shallow embedding, which strongly resembles the original Cogent code:

$$\begin{aligned}
&\mathsf{add_s}\ x\ y = x + y \\
&\mathsf{sum_s}\ xs = \mathsf{fold_s}\ \mathsf{add_s}\ 0\ xs\ 0\ (\mathsf{length}_s\ xs)\ ()
\end{aligned}$$

[3]At the time of writing, Cogent's shallow phase (Theorem 2.9) implicitly assumes abstract function correctness rather than doing so explicitly as done in other phases. So for now, we just copy and discharge these.

```
A repeat(u32 n, fid f, fd g, A acc, O obsv) {
  for(u32 i = 0; i < n; i++)
    if (dispatch_f(f, acc, obsv)) break;
    else acc = dispatch_g(g, acc, obsv);
  return acc;
}
```

Figure 8. C implementation of the generic loop.

With all of these proofs in place, we get a refinement theorem like Theorem 2.11 for the function sum, which leaves no function unverified. This refinement theorem allows us to prove properties of sum_s just by equational reasoning, and have these proofs also apply to the C implementation.

5.1 Verifying C Parts of BilbyFs

As mentioned, the previous verification of the functional correctness of key operations in BilbyFs [3] simply assumed the correctness of abstract functions, including an axiomatisation of array operations [2]. We removed the axiomatisation for the five core array operations that we verified, and were able to show that the functional correctness proofs compose for the combined system. The array operations that remain unverified are functions like `create`, `set`, `copy`, `cmp`, that depend on platform-specific functions such as `malloc`.

5.2 Binary Search

Figure 10 gives the Cogent code for binary search using our previously defined and verified array functions and repeat. Note that while we specify the maximum number of iterations to repeat as the length of the array, the algorithm is still $O(\log n)$ as it will always exit early from the stop condition. This *fuel* argument is an easy way to ensure that Isabelle is convinced that our functions terminate.

We firstly prove that the generated shallow embedding, which strongly resembles the Cogent code, is correct:

Theorem 5.1 (Correctness of Cogent binary search)**.** *Let* i= unat (binary-search$_s$ (xs, v)) *in*

$$\text{sorted } xs \land \text{length } xs < 2^{32} \longrightarrow (i < \text{length } xs \longrightarrow xs \,!\, i = v) \land (\neg\, i < \text{length } xs \longrightarrow v \notin \text{set } xs)$$

where length, set, and sorted are Isabelle's list library functions that return the length of a list, turn a list to a set, and check whether a list is sorted, respectively. We denote search failure by returning an index that is out of bounds.

From our overall refinement theorem (Theorem 2.11), we can easily conclude that the C implementation is correct and remove any reference to Cogent:

Corollary 5.2 (Correctness of the C binary search)**.** *Let xs be the list abstraction of the array arr for C heap* σ*,* valid (σ, arr) *be the predicate that states that the array is valid, i.e., the array's size (in bytes) is less than the size of memory (2^{32} bytes) and is well-formed, and* same (σ, σ', arr) *be the predicate that states that an array is the same for the given C heaps:*

$$\text{sorted } xs \land \text{valid } (\sigma, arr) \land (arr, v) \vdash \sigma \mid \text{binary-search}_c \Downarrow_{\mathsf{c}} i \mid \sigma' \longrightarrow \text{valid } (\sigma', arr) \land \text{same } (\sigma, \sigma', arr) \land (\text{unat } i < \text{length } xs \longrightarrow xs \,!\, (\text{unat } i) = v) \land (\neg\, \text{unat } i < \text{length } xs \longrightarrow v \notin \text{set } xs)$$

This theorem depends on no additional assumptions about any functions or any other part of the Cogent framework. With this, we have shown that the Cogent framework enables the verification of combined Cogent-C systems.

6 Related Work

Compiler Correctness. Patterson and Ahmed [18] have defined a spectrum of compiler verification theorems focusing on compositional compiler correctness, extensible through linking. Cogent's certifying compiler does not neatly fall on this spectrum as the compiler itself does the linking of the Cogent-C system, and we are linking with manually verified C rather than other compiler outputs.

Like Cogent, the Cito language [23] allows cross-language linking and combined verification in a proof assistant. Unlike Cogent, Cito is a low-level C-like language without a sophisticated type system, so there are no FFI requirements to enforce static guarantees. Pit-Claudel et al. [19] use Cito as a target for verified compilation of relational queries, and support linking with foreign assembly code, but do not compose verification across languages for refinement from a common high-level spec, as we do.

An important part of verified compilation is how the semantics of linking is defined for the source language. The simplest way to define linking on the source language is to require that a program may only be linked if it refines a program definable in the source language. SepCompCert [7] and Pilsner [13] take this approach. This approach is suitable for languages which, unlike Cogent, are expressive enough to encompass all programs, but we require C functions to refine more expressive specifications than what is definable natively in Cogent. The Cogent compiler instead generates shallow embeddings of Cogent programs in Isabelle/HOL, that enable linking at the Isabelle specification level. Hence, Cogent supports source-independent linking insofar as we can embed all source languages inside Isabelle/HOL.

Verification Approach. Arrays are widely used data structures, particularly in low level systems programming. AutoCorres [5, 6] simplifies the verification of C code by abstracting it into a monadic embedding of C in Isabelle/HOL. The Cogent compiler further simplifies reasoning about the Cogent parts of a system by abstracting AutoCorres' monadic C into a purely functional Cogent embedding in Isabelle/HOL. For foreign functions, abstraction from AutoCorres' monadic C embedding to a purely functional embedding is manually

$$
\begin{array}{l}
\mathsf{repeat_u}\ (\mu_1, (n, f, g, acc, obs))\ (\mu_3, y) = \\
\quad (n > 0 \longrightarrow \\
\qquad \exists b.\ [a \mapsto (acc, obs)] \vdash f\ a \mid \mu_1 \Downarrow_{u} b) \mid \mu_1 \wedge \\
\qquad\quad (b \longrightarrow \mu_1 = \mu_3 \wedge y = acc) \wedge \\
\qquad\quad (\neg b \longrightarrow \exists \mu_2\ acc'. \\
\qquad\qquad [a \mapsto (acc, obs)] \vdash g\ a \mid \mu_1 \Downarrow_{u} acc' \mid \mu_2 \wedge \\
\qquad\qquad \mathsf{repeat_u}\ (\mu_2, (n\text{-}1, f, g, acc', obsv))\ (\mu_3, y))) \wedge \\
\quad (n = 0 \longrightarrow \mu_1 = \mu_3 \wedge y = acc)
\end{array}
$$

(a) C abstractions for the *update* semantics.

$$
\begin{array}{l}
\mathsf{repeat_v}\ (n, f, g, acc, obs)\ y = \\
\quad (n > 0 \longrightarrow \\
\qquad \exists b.\ [a \mapsto (acc, obs)] \vdash f\ a \Downarrow_{v} b) \wedge \\
\qquad\quad (b \longrightarrow y = acc) \wedge \\
\qquad\quad (\neg b \longrightarrow \exists acc'. \\
\qquad\qquad [a \mapsto (acc, obs)] \vdash g\ a \Downarrow_{v} acc' \wedge \\
\qquad\qquad \mathsf{repeat_v}\ (n\text{-}1, f, g, acc', obsv)\ y)) \wedge \\
\quad (n = 0 \longrightarrow y = acc)
\end{array}
$$

(b) C abstractions for the *value* semantics.

Figure 9. Cogent-compatible abstraction for the generic loop.

```
type Range = (U32, U32, Bool)
stop : (Range, ((Array U32)!, U32)) → Bool
stop ((l, r, b), (arr, v)) = b ∨ l ≥ r
search : (Range, ((Array U32)!, U32)) → Range
search ((l, r, b), (arr, v)) =
  let m = l + (r - l) ÷ 2 and
      x = get (arr, m, 0)
  in if | x < v → (m+1, r, b)
        | x > v → (l, m, b)
        | else → (m, r, True)
binary-search : ((Array U32)!, U32) → U32
binary-search (arr, v) =
  let len = length arr and
      (l, r, b) = repeat (len, stop, search, (0, len, False), (arr, v))
  in if b then l else → len
```

Figure 10. The binary search algorithm in Cogent.

verified through intermediate Cogent embeddings. AutoCorres comes with a number of examples including binary search and QuickSort on 32-bit word arrays which use the get and put operations. Their proofs for these operations (roughly 250 lines) are about half the size of ours but ours are reusable for arrays with elements of any unboxed type.

If we were only concerned with proving functional correctness of an array implementation, we could have taken a top down approach, i.e., generate an implementation from the specification, using existing tools or frameworks [9–11], rather than prove that an implementation refines its specification. While a top down approach would likely lessen the verification burden, our approach allows for more control over the implementation and is suitable for cases where an implementation already exists, as is the case for BilbyFs.

As mentioned in the introduction, Cogent purposely lacks recursion and iteration in order to ensure totality. There is an ongoing effort to add a limited form of recursion to Cogent [12] while retaining totality through a termination checker [12, 20]. Extending the proof infrastructure of Cogent to support recursive types and to certify termination for functions that pass the termination checker is non-trivial and still remains open. Additionally, we are exploring adding arrays as built in types in Cogent and extending the compiler certificate to account for a number of array operations. Even then, we believe that the language interoperability approach presented in this paper remains valuable. It serves as a reference for how Cogent users and system developers can contribute their own additional data structures with their preferred implementation to Cogent rather than relying solely on Cogent developers to extend the language with each desired additional data structure and data structure operation. Moreover, operations that are directly implemented in C and called through the FFI may occasionally be implemented more efficiently than if they were directly implemented in Cogent. This is because users can escape the Cogent type system when implementing operations in C and for instance use internal aliasing within a function's implementation as long as they can prove that the overall C implementation respects the frame invariant.

7 Conclusion

We have demonstrated our cross-language approach to proving software correct. Our systems mix Cogent, a safe functional language with a compiler that proves most of the required theorems automatically, and C, an unsafe imperative language with few guarantees. Specifically, we verified the array implementation and general loop iterators provided in Cogent's ADT library, which were used the implementation of real-world file systems, and we showed that they maintain the invariants required by Cogent. This enabled us to eliminate some key assumptions in the pre-existing verification of the BilbyFs file system in Cogent. These case-studies demonstrate that manual C verification can be straightforwardly composed with Cogent's refinement chain, leading to a top-level shallow embedding that can be seamlessly connected with functional correctness specifications to ensure the correctness of an overall Cogent-C system.

Acknowledgments

This research is partially supported by an Australian Government Research Training Program (RTP) Scholarship.

References

[1] Amal Ahmed. 2016. Compositional Compiler Verification for a Multi-Language World. In *1st International Conference on Formal Structures for*

Computation and Deduction (FSCD 2016) (Leibniz International Proceedings in Informatics (LIPIcs)), Delia Kesner and Brigitte Pientka (Eds.), Vol. 52. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:1. https://doi.org/10.4230/LIPIcs.FSCD.2016.1

[2] Sidney Amani. 2016. *A Methodology for Trustworthy File Systems*. Ph.D. Dissertation. University of New South Wales, Sydney, Australia.

[3] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computer Machinery, Atlanta, GA, USA, 175–188. https://doi.org/10.1145/2872362.2872404

[4] Cogent 2021. Cogent FFI Examples (Code and Proofs). https://github.com/au-ts/cogent/releases/tag/cpp2022

[5] David Greenaway, June Andronick, and Gerwin Klein. 2012. Bridging the Gap: Automatic Verified Abstraction of C. In *International Conference on Interactive Theorem Proving*. Springer Berlin Heidelberg, Princeton, New Jersey, USA, 99–115. https://doi.org/10.1007/978-3-642-32347-8_8

[6] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2014. Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Edinburgh, UK, 429–439. https://doi.org/10.1145/2594291.2594296

[7] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 178–190. https://doi.org/10.1145/2837614.2837642

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *ACM Symposium on Operating Systems Principles*. Association for Computer Machinery, Big Sky, MT, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[9] Peter Lammich. 2019. Generating Verified LLVM from Isabelle/HOL. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA (LIPIcs)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Portland, 22:1–22:19. https://doi.org/10.4230/LIPIcs.ITP.2019.22

[10] Peter Lammich. 2019. Refinement to Imperative HOL. *Journal of Automated Reasoning* 62, 4 (01 Apr 2019), 481–503. https://doi.org/10.1007/s10817-017-9437-1

[11] Peter Lammich and Andreas Lochbihler. 2010. The Isabelle Collections Framework. In *Interactive Theorem Proving*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 339–354. https://doi.org/10.1007/978-3-642-14052-5_24

[12] Emmet Murray. 2019. Recursive Types for Cogent. https://github.com/emmet-m/thesis. Accessed November 2021.

[13] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2784731.2784764

[14] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

[15] Liam O'Connor. 2019. *Type Systems for Systems Types*. Ph.D. Dissertation. University of New South Wales. https://www.unsworks.unsw.edu.au/permalink/f/a5fmj0/unsworks_61747

[16] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement Through Restraint: Bringing Down the Cost of Verification. In *International Conference on Functional Programming*. Nara, Japan. https://doi.org/10.1145/2951913.2951940

[17] Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. [n.d.]. Cogent: Uniqueness Types and Certifying Compilation. *Journal of Functional Programming* ([n. d.]). https://doi.org/10.1017/S095679682100023X To appear.

[18] Daniel Patterson and Amal Ahmed. 2019. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.* 3, ICFP (2019), 85:1–85:29. https://doi.org/10.1145/3341689

[19] Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 119–137. https://doi.org/10.1007/978-3-030-51054-1_7

[20] Lucy Qiu. 2020. Termination Checker for Recursive Types in Cogent.

[21] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. 2016. A Framework for the Automatic Formal Verification of Refinement from Cogent to C. In *International Conference on Interactive Theorem Proving*. Nancy, France. https://doi.org/10.1007/978-3-319-43144-4_20

[22] Philip Wadler. 1990. Linear types can change the world!. In *Programming Concepts and Methods*.

[23] Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler verification meets cross-language linking via data abstraction. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 675–690. https://doi.org/10.1145/2660193.2660201

[24] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. 2009. Mind the Gap: A Verification Framework for Low-Level C. In *International Conference on Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (Ed.). Springer, Munich, Germany, 500–515. https://doi.org/10.1007/978-3-642-03359-9_34