A Framework for the Automatic Formal Verification of Refinement from Cogent to C

Christine Rizkallah[1,2(✉)], Japheth Lim[1,2], Yutaka Nagashima[1],
Thomas Sewell[1,2], Zilin Chen[1,2], Liam O'Connor[1,2], Toby Murray[1,2],
Gabriele Keller[1,2], and Gerwin Klein[1,2]

[1] Data61 (formerly NICTA), Sydney, Australia
[2] UNSW, Sydney, Australia
christine.rizkallah@data61.csiro.au

Abstract. Our language Cogent simplifies verification of systems software using a certifying compiler, which produces a proof that the generated C code is a refinement of the original Cogent program. Despite the fact that Cogent itself contains a number of refinement layers, the semantic gap between even the lowest level of Cogent semantics and the generated C code remains large.

In this paper we close this gap with an automated refinement framework which validates the compiler's code generation phase. This framework makes use of existing C verification tools and introduces a new technique to relate the type systems of Cogent and C.

1 Introduction

In previous work, we designed a new language called Cogent [9] for easing the verification of certain classes of systems code such as file systems. Cogent is a linearly-typed, pure, polymorphic, functional language with a *certifying* compiler. We used it in separate work to write two Linux filesystems, `ext2` and `BilbyFs`, and achieved performance comparable to their native C implementations [2].

From a Cogent program the Cogent compiler produces three artefacts: C code, a shallow embedding of the Cogent program in Isabelle/HOL [8], and an Isabelle/HOL proof relating the two. The compiler certificate is a series of
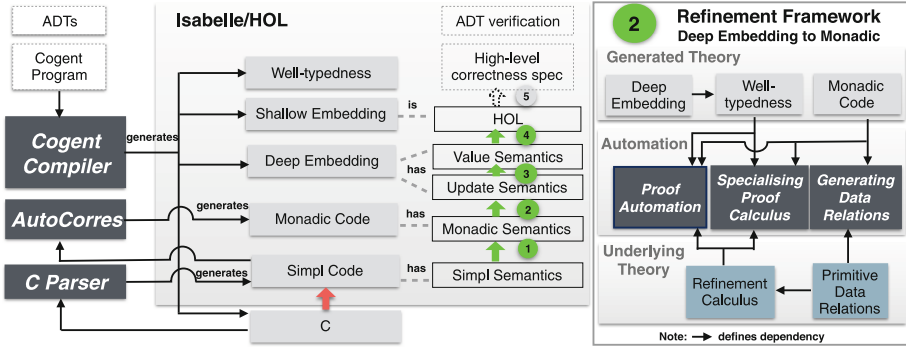
Fig. 1. An overview of the verification chain and our refinement framework.

language-level proofs and per-program translation validation phases that are combined into one top-level theorem in Isabelle/HOL. The most involved phase, and the phase we discuss in this paper, is the translation validation phase relating COGENT's imperative semantics to the generated C.

We present a refinement framework that enables the full automation of this phase of COGENT's certifying compilation. This framework has several components that relate COGENT values, states, types, and statements to their C counterparts. We put significant proof engineering work into enabling the framework to bridge the gap between the COGENT store and the C heap semantics. Moreover, we introduced the idea of *partial type erasure* to eliminate linearity information from a COGENT type in order to relate it to the corresponding C type. Furthermore, to relate COGENT and C statements, we developed a refinement calculus which contains a set of compositional proof rules. Given a program, our framework then customises the proof rules based on the values, types, and states that are used in this program. Finally, our refinement tactic applies the customised rules in a syntax-directed manner, certifying the refinement for this phase.

The method scales to significant COGENT code size, as demonstrated in the two Linux filesystems [2] mentioned above. A snapshot of our work is available online [1].

2 Overview and Background

This section explains the contribution of this paper within the broader COGENT project. The heart of the COGENT project is its certifying compiler. The certificate the compiler produces is a refinement theorem relating the generated shallow embedding and the generated C code. To ensure the C code is run correctly on the binary level, it can be compiled by CompCert [7].[1] It also falls into

---

[1] Mind the potential logical gap between our C parser's C semantics [13] and that of CompCert.

the subset of Sewell *et al.*'s gcc translation validator [12], which can be made to compose directly with our compiler certificate.[2]

The shallow Isabelle/HOL embedding is convenient for manual reasoning; however, the compiler additionally produces a deep embedding of each COGENT program, for the sake of structuring the generated certificate theorem and proof. There are two formal semantics for this deep embedding: (1) a functional *value semantics* where programs evaluate to values and (2) an imperative *update semantics* where programs manipulate references to mutable global state.

The left side of Fig. 1 summarises the generated program representations and the breakdown of the compiler certificate. The program representations are (from the bottom of Fig. 1): the C code, the semantics of the C code [13] expressed in Simpl [11], which is a generic imperative language inside Isabelle/HOL, the same expressed as a monadic program [4], an A-normal [10] deep embedding of the COGENT program, and a shallow embedding. Several theorems rely on the COGENT program being well-typed, which we prove automatically using type inference information from the compiler.

The labelled arrows and the arrow from C to Simpl represent refinement proofs and the arrow labels correspond to the numbers in the following description. The only arrow that is not verified is the one crossing from C code into Isabelle/HOL at the bottom of Fig. 1 — this is the C parser [13], which is a mature tool used in a number of large-scale verifications [5]. It could additionally be checked by Sewell *et al.*'s gcc translation validation tool.

We briefly describe each intermediate theorem, starting with Simpl at the bottom. For well-typed COGENT programs, we automatically prove the following four theorems, which together form the compiler certificate:

① The C parser's Simpl code corresponds to a monadic representation of the C code.

② The monadic code terminates and is a refinement of the update semantics of the COGENT deep embedding. To relate COGENT's linear type system to the monadic one, we introduce the reusable idea of *partial type erasure.*

③ If a COGENT deep embedding evaluates in the update semantics, it evaluates to the same result in the value semantics.

④ If the COGENT deep embedding evaluates in the value semantics then the COGENT shallow embedding evaluates to a corresponding shallow Isabelle/HOL value.

In order to prove high-level functional correctness, an additional step is necessary:

Arrow ⑤ indicates verification of user-supplied abstract data types (ADTs) implemented in C and manual high-level proofs on top of the shallow embedding. We demonstrated that this step is enabled by the previous steps for two real-world filesystems [2].

Step ③ is a consequence of linear types. It is a general property about the language and has been proven manually once and for all [9]. Steps ①, ②, and ④,

---

[2] COGENT's occasionally larger stack frames lead to `memcpy()` calls that, while conceptually straightforward, the translation validator does not yet cover.

as well as their respective proofs, are generated by our compiler for every program. The proof for step ① is generated by an adjusted version of the AutoCorres tool [4]. For steps ② and ④ we define compositional refinement calculi which enable the automation of the proofs. The most involved refinement proof is the one for step ② which we present in this paper. It took about three person years to develop tools for automating this proof. The calculus for step ④ is similar but much simpler, as at this stage one does not reason about the state. In comparison, its development only took a few person weeks.

The right side of Fig. 1 expands on the refinement framework used for proving step ②. The bottom layer represents the underlying theory we developed for defining primitive value and type relations which we use to create a refinement calculus between COGENT deeply embedded expressions and corresponding monadic statements. The middle layer represents the proof tools that automate the refinement proof on a per-program basis. These proof tools rely on the underlying theories about the language in general, and on compiler generated theories specific to the program. In particular, we have a tool for generating non-primitive data relations, one for specialising complex rules in the calculus to support automation, and finally a proof automation tactic which composes the proof rules to provide a fully-automatic refinement proof.

2.1 COGENT

COGENT is a restricted, polymorphic, higher-order, and purely functional language with *linear types*. The linear types ensure that resources such as memory are disposed of correctly without run-time support like garbage collection. Crucially for us, they also allow COGENT to be compiled into efficient C, including *destructive updates* to values rather than the repeated copying common in purely functional styles.

Variables of linear type must be used *exactly once*. This means each active mutable heap object has exactly one active pointer in scope at any point in the program. Hence, the difference between a destructive update and a pure copy-update is unobservable.

The COGENT compiler generates C code, a shallow embedding, and a collection of "hints" used by the proof tactic to certify the compilation. Importantly, the performance of the generated C is comparable to carefully handwritten C.

COGENT's certifying compilation makes the verification of filesystems more cost-effective, fully automating a significant part of the low-level proofs. We demonstrate this on two real-world COGENT filesystems, with a minimal TCB [2].

This paper focuses on the lower-level generated refinement proofs, which connect COGENT's *update semantics* to C. Figure 2 introduces a relevant fragment of COGENT. Many features of the full language are omitted here and described in detail elsewhere [9], including polymorphism, sum types, and the foreign function interface. The following gives a brief summary.

Much of the syntax presented in our fragment is standard for a functional language, such as handling control flow (**if**) and local bindings (**let**). The main point of difference is COGENT's record system: Some care is needed to reconcile

$$\boxed{\text{Statics}}$$

primops	o	$\in \{+, *, /, <=, ==, ||, <<, \ldots\}$
literals	ℓ	$\in \{123, \mathtt{True}, \mathtt{'a'}, \ldots\}$
expressions	e	$::= x \mid () \mid f \mid o(\overline{e}) \mid e_1\, e_2 \mid \ell$
		$\mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$
		$\mid \{\overline{\mathrm{f} = e}\} \mid \mathbf{put}\ e_1.\mathrm{f} := e_2 \mid \mathbf{take}\ x\ \{\mathrm{f} = y\} = e_1\ \mathbf{in}\ e_2 \mid \cdots$
fn. names		$\ni f, g$
variables		$\ni x, y$
record fields		$\ni \mathrm{f}, \mathrm{g}$
prim. types	t	$::= \mathtt{U8} \mid \mathtt{U16} \mid \mathtt{U32} \mid \mathtt{U64} \mid \mathtt{Bool}$
types	τ	$::= \alpha \mid t \mid \tau_1 \to \tau_2 \mid () \mid \{\overline{\mathrm{f} :: \tau^?}\}\, m \mid \cdots$
field types	$\tau^?$	$::= \tau \mid \mp$
modes	m	$::= \mathtt{r} \mid \mathtt{w} \mid \mathtt{u}$

$$\boxed{\text{Dynamics}}$$

update semantics values	u	$::= \ell\, r \mid () \mid \langle \lambda x.\, e \rangle \mid \{\overline{\mathrm{f} = u}\}\, r \mid p\, r \mid \cdots$
type representations	r	$::= t \mid () \mid \mathtt{Fun} \mid \{\overline{\mathrm{f} :: r}\} \mid \mathtt{Ptr}\, r \mid \cdots$
environments	U	$::= \overline{x \mapsto u}$ pointers p stores $\mu :\ p \rightharpoonup u$

Fig. 2. Definitions for COGENT fragment

record types and linear types. If a record contains at least one linear field, the whole record is of linear type. Otherwise, the linear field could be shared by sharing the record.

Accessing records becomes more complex as well. For instance, assume that `Object` is a type synonym for a record type containing an integer and two (linear) buffers, where `Object` = $\{\mathtt{size} :: \mathtt{U32}, \mathrm{b}_1 :: \mathtt{Buf}, \mathrm{b}_2 :: \mathtt{Buf}\}\, \mathtt{u}$. Let us say we want to extract the field b_1 from an `Object`. If we extract just a single `Buf`, we have implicitly discarded the other buffer b_2. However, we cannot return the entire `Object` along with `Buf`, as this would introduce aliasing. Our solution is to return along with `Buf` an `Object` where the field b_1 cannot be extracted again, and reflect this in the field's type, written as $\mathrm{b}_1 :: \overline{\mathtt{Buf}}$. This field extractor, whose general form is **take** $x\ \{\mathrm{f} = y\} = e_1\ \mathbf{in}\ e_2$, operates as follows: given a record e_1, it binds the field f of e_1 to the variable y, and the new record to the variable x in e_2. If that field is linear, it will then be marked as unavailable, or *taken*, in the type of the new record x.

Conversely, we also introduce a **put** operation, which, given a record with a taken field, allows a new value to be supplied in its place. The expression **put** $e_1.\mathrm{f} := e_2$ returns the record in e_1 where the field f has been replaced with the result of e_2. Unless the type of the field f allows it to be discarded, it must already be taken, to avoid accidentally destroying our only reference to a linear resource.

```
1 flip :: {f :: U8} w → {f :: U8} w
2 flip x =
3    take x' {f = y} = x
4    in if y == 0
5       then put x'.f := 1
6       else put x'.f := 0
```

Fig. 3. Example function in COGENT. *flip* updates a record on the heap in place.

We distinguish boxed records stored on the heap from unboxed records that are passed by value. Unboxed records can be created using a simple struct literal $\{\overline{f_i = e_i}\}$. Boxed records are created by invoking an externally-defined C allocator function. For these allocation functions, it is often convenient to allocate a record with all fields already taken, to indicate that they are uninitialised. That is, a function for allocating `Object`-like records might return values of type: $\{\text{size} :: \text{U32}, b_1 :: \text{Buf}, b_2 :: \text{Buf}\}$ w.

Also included in a record type is the *storage mode* of the type. A record is stored on the heap when its associated mode m is not unboxed. For boxed records, the storage mode distinguishes between those that are writable vs. read-only.

Example 1. Figure 3 defines a simple function in COGENT which, given a mutable record x, first **take**s the field f and, depending on its value, destructively updates the field with a new value, returning the updated record.

The details of COGENT's type system, semantics, and this proof are presented in [9], we only repeat the top-level concepts here.

The dynamic big step *update* semantics maps a triple of environment U, expression e, and mutable store μ to a result value u and a new mutable environment μ', written $U \vdash e \mid \mu \Downarrow_u u \mid \mu'$. The rules [9] for variables and **let** are straightforward. Functions are top-level functions in COGENT, and a function name simply evaluates to the lambda-expression it represents. The **take** and **put** rules evaluate as described above.

The static semantics include the standard typing judgement $\Gamma \vdash e : \tau$. Unlike conventional type systems, linear type systems are *substructural*, which means that the context Γ cannot be treated merely as a set of assumptions that always grows as one descends into the syntax tree. Instead, assumptions may also be removed from the context. This complication requires us to occasionally generalise the **corres** rules presented in Sect. 3.4 with multiple typing assumptions with different contexts.

To state *type preservation* for COGENT, we define the corresponding typing judgement for dynamic *values*, written $u \mid \mu : \tau$ and a generalisation of it to *environments* and *contexts*, written $U \mid \mu : \Gamma$. With this, we can prove the following (see also [9]).

Theorem 1 (type preservation). *For a program e, if $\Gamma \vdash e : \tau$ and $U \mid \mu : \Gamma$ and $U \vdash e \mid \mu \Downarrow_u u \mid \mu'$, then $u \mid \mu' : \tau$*

$$\boxed{\mathbf{repr}(\cdot) : \tau \to r}$$

$$\mathbf{repr}(()) = () \quad \mathbf{repr}(\{\overline{\mathbf{f} :: \tau^?}\}\ \mathbf{u}) = \{\overline{\mathbf{f} :: \mathbf{repr}(\tau)}\} \quad \mathbf{repr}(\tau \to \rho) \quad = \mathtt{Fun}$$
$$\mathbf{repr}(t) = t \quad \mathbf{repr}(\{\overline{\mathbf{f} :: \tau^?}\}\ \mathbf{r}) = \mathtt{Ptr}\ \{\overline{\mathbf{f} :: \mathbf{repr}(\tau)}\} \quad \mathbf{repr}(\{\overline{\mathbf{f} :: \tau^?}\}\ \mathbf{w}) = \mathtt{Ptr}\ \{\overline{\mathbf{f} :: \mathbf{repr}(\tau)}\}$$

Fig. 4. Partial type erasure of dynamic typing relation for update semantics

For a COGENT value to be well-typed, all accessible pointers in this value, e.g. a record, must be valid. This is important for proving safety, but becomes cumbersome when showing refinement to C as there exist values in the C code, such as those for taken fields, which may include temporarily invalid pointers. We therefore include additional information in each COGENT value, called its *representation*, which provides enough type information to determine the corresponding C type, without requiring recursive descent into the heap. In other words, the representation shown in Fig. 4 contains only the type information which is pertinent to C, with the linearity information erased. We call this technique *partial type erasure*. The value typing relation ensures that the representation information agrees with the value's type.

2.2   AutoCorres and C Monads in Isabelle/HOL

We use the C-to-Simpl [13] parser to provide a formal semantics for the generated C code. In principle, we could work from the C parser's output directly; however, this would mean dealing with the details of its low-level memory model. Instead, we opt to work with a *typed* heap model, provided by AutoCorres [4]. Specifically, the *state* of the AutoCorres monadic representation contains a set of *typed heaps*, each of type τ ptr $\Rightarrow \tau$, one for each type τ used on the heap in the C input program.

As AutoCorres was designed for human-guided verification, it uses many context-sensitive rules to simplify the generated code. As we aim to verify code automatically, we switch off most of these simplification stages in order to obtain predictable output.

AutoCorres generates shallow embeddings of code in the *nondeterministic state monad* of Cock *et al.* [3]. In this monad, computation is represented by functions of type *state* $\Rightarrow (\alpha \times state)$ *set* $\times$ *bool*. Here *state* is the global state of the monadic program, including global variables, while α is the return-type of the computation. A computation takes as input the global state and returns a set, *results*, of pairs with new state and result value. Additionally the computation returns a boolean, *failed*, indicating whether there potentially was undefined behaviour.

As C does not guarantee that all pointer locations are valid, AutoCorres emits *is-valid* guards before each memory access. When proving refinement between COGENT and monadic code, we need to discharge those guards using a state invariant (Sect. 3.2).

Figure 5 shows an example AutoCorres specification, using the following keywords:

do ...; ... **od**   sequence of statements
condition $cond\ e_1\ e_2$ run e_1 if $cond$ is true, otherwise run e_2
return v monadic return
gets f access part of monadic state given by f
modify h update part of monadic state given by h
guard G program fails if monadic state does not satisfy G

3 Refinement Framework

Recall that for a well-typed COGENT program, the compiler emits C code, a deep embedding of the program's semantics, and a proof that the C code correctly refines this embedding. We choose C as a compilation target because most existing systems code is written in C, and thanks to tools like CompCert and gcc translation validation, our C subset has formalised semantics and an existing formal verification infrastructure.

The right side of Fig. 1 provides an overview of the generation of our refinement proof. To phrase the refinement statement, we first define how deeply-embedded COGENT values relate to values in the monadic embedding (Sect. 3.2).

The C code generation is straightforward and this step itself does not perform global optimisations or transformations. Such transformations, for instance A-normalisation, are performed in earlier compiler phases. A-normalisation in particular is performed to simplify code generation, but it also simplifies our C refinement. Since it is performed early (and verified early on top of the shallow embedding [9]), it is sufficient for us to only consider COGENT expressions in A-normal form here, where nested subexpressions are replaced with explicit

```
1 flip :: {f :: U8} → {f :: U8}          1 flip_C :: rec_1 ptr ⇒ (rec_1 ptr, σ) nondet_monad
2 flip x =                                2 flip_C x = do
3    take x' {f = y} = x                  3    guard (λσ. is-valid σ x);
.                                         4    y ← gets (λσ. σ[r].f);
4    in let tmp_1 = 0                     5    tmp_1 ← return 0;
5       and tmp_2 = (y == tmp_1)          6    tmp_2 ← return bool (y = tmp_1);
6       in if tmp_2                       7    tmp_result ← condition (bool tmp_2 ≠ 0)
7          then let tmp_3 = 1             8       (do tmp_3 ← return 1;
8             and x'' = put x'.f := tmp_3 9          guard (λσ. is-valid σ x);
.                                         10         modify (λσ. σ[x].f := tmp_3);
9             in x''                      11         return x od)
10         else let tmp_4 = 0            12      (do tmp_4 ← return 0;
11            and x'' = put x'.f := tmp_4 13         guard (λσ. is-valid σ x);
.                                         14         modify (λσ. σ[x].f := tmp_4);
12            in x''                     15         return x od);
.                                        16    return tmp_result
.                                        17    od
```

Fig. 5. Intermediate representations of COGENT function from Fig. 3. Left: A-normalised source code, embedded into Isabelle/HOL. Right: AutoCorres monadic semantics for generated C code.

variable bindings. With this, the refinement calculus contains a set of compositional **corres** proof rules, typically one for each A-normal COGENT construct, which are applied automatically in a syntax-directed manner (Sect. 3.4).

The **corres** proof rules depend on preconditions about the expected state of the program, such as preconditions about the type and validity of pointers in the heap. We propagate the conditions similarly to the proof calculus of Cock *et al.* [3]. Our refinement theorem does not need an explicit assumption of well-typedness for the whole COGENT program — The proof tactic will simply fail for programs that are ill-typed.

Since our **corres** proof rules are specialised to COGENT and to the operation of the compiler, we can predict the form of their preconditions and design proof rules to combine them. This forms the basis for automation.

3.1   Refinement Statement

We define refinement generically between a monadic computation p_m and a COGENT expression e, evaluated under the update semantics. We denote the refinement predicate **corres**. The state relation R changes for each COGENT program, so we parametrise **corres** by an arbitrary state relation R. It is additionally parametrised by the typing context Γ and the environment U, as well as by the initial update semantics store μ and monadic shallow embedding state σ.

Definition 1 (correspondence)

$$
\begin{aligned}
&\textbf{corres } R \; e \; p_m \; U \; \Gamma \; \mu \; \sigma \stackrel{def}{=} \\
&\quad U \mid \mu : \Gamma \longrightarrow (\mu, \sigma) \in R \longrightarrow \\
&\quad\quad (\neg \; failed \; (p_m \; \sigma) \; \wedge \\
&\quad\quad (\forall v_m \; \sigma'. \; (v_m, \sigma') \in results \; (p_m \; \sigma) \longrightarrow \\
&\quad\quad\quad (\exists \mu' \; u. \; U \vdash e \mid \mu \Downarrow_u u \mid \mu' \wedge (\mu', \sigma') \in R \wedge val\text{-}rel \; u \; v_m)))
\end{aligned}
$$

Definition 1 states for well-typed stores μ that if the state relation R holds initially, then the monadic computation p_m cannot fail and, moreover, for all executions of p_m there must exist a corresponding execution under the update semantics of the expression e such that the final states are related by a state relation R and a value relation *val-rel* holds between the results of e and p_m.[3] We present the state and value relations in Sect. 3.2.

AutoCorres proves that if the monadic code never fails, then the C code is type- and memory-safe, and is free of undefined behaviour [4]. We prove non-failure as a side-condition of the refinement statement, essentially using COGENT's type system to guarantee C memory safety during execution. The **corres** predicate can compose with itself sequentially: it both assumes and shows the relation R, and the additional typing assumptions are preserved thanks to type preservation (Theorem 1).

---

[3] Although **corres** technically permits the monadic code to return no results, the code that we generate will additionally always return *results* $\neq \emptyset$ as long as it has not *failed*.

3.2   Data Relations

For each program, based on a library for primitive types, we generate a set of relations between the values, types and heaps of the COGENT and monadic code. We denote these as *val-rel*, *type-rel* and $\mathcal{R}$ respectively.

   We must give these relations separate definitions for each COGENT type, because each C struct type is embedded as a distinct Isabelle/HOL record. We use Isabelle's ad-hoc overloading mechanism for this.

   Recall that AutoCorres generates different typed heaps for each C type. The type relation *type-rel* is used by the state relation $\mathcal{R}$ to select the corresponding typed heap for each COGENT type. It is defined using the **repr** function (Fig. 4) which performs *partial type erasure*, unifying COGENT types that differ only in linear annotations in order to relate them to the same C type.

   Given *val-rel* and *type-rel* for a particular COGENT program, the *state relation* $\mathcal{R}$ defines the correspondence between the store μ over which the COGENT update semantics operates, and the state σ of the monadic shallow embedding. This relation is made into an invariant in **corres** (Sect. 3.1); it allows us to show that all C pointer accesses satisfy *is-valid*, whenever there are corresponding objects in the COGENT store μ.

Definition 2 (state relation). $(\mu, \sigma) \in \mathcal{R}$ *if and only if for all pointers p in the domain of μ, there exists a value v in the appropriate heap of σ (as defined by type-rel) at location p, such that val-rel $(\mu\,p)\;v$ holds.*

Generating Data Relations. We generate $\mathcal{R}$, *val-rel* and *type-rel* after obtaining the monadic program and its typed heaps from AutoCorres. Our COGENT compiler outputs a list of (COGENT, C) type pairs, which is used by an Isabelle/ML procedure to generate the needed relations.

Example 2. The program in Fig. 5 uses the types U8, Bool and {f :: U8}, which correspond to the C types word8, **bool** and rec_1, respectively. For *val-rel* and *type-rel*, the U8–word8 relation can be defined a priori, but **bool** and rec_1 are generated with the monadic program and their data relations are generated dynamically:

$$\text{(pre-defined)} \quad \textit{val-rel}\ (a :: \mathtt{U8})\ (a_C :: \mathtt{word8}) \overset{\text{def}}{=} (a = a_C)$$
$$\textit{val-rel}\ (a :: \mathtt{Bool})\ (a_C :: \mathbf{bool}) \overset{\text{def}}{=} (a = (\textit{bool}\ a_C \neq 0))$$
$$\textit{val-rel}\ (a :: \{\mathtt{f} :: \mathtt{U8}\})\ (a_C :: \mathrm{rec}_1) \overset{\text{def}}{=} \textit{val-rel}\ (a.\mathtt{f})\ (a_C.\mathtt{f})$$

Note that the *val-rel* definition for {f :: U8} depends on the definition for its field of type U8. The COGENT compiler always outputs the type list in dependency order, so this does not pose a problem.

   The state relation $\mathcal{R}$ cannot be overloaded in the same way as *val-rel* and *type-rel*, because it relates the heaps for every type simultaneously. We introduce an intermediate state relation, *heap-rel*, which relates a particular typed heap with a portion of the COGENT store. Like the other relations, this intermediate

relation can make use of type-based overloading. Following Definition 2, we define *heap-rel* for each type τ that appears on the heap as follows:

$$heap\text{-}rel\ \sigma_\tau\ \mu \stackrel{\mathrm{def}}{=} \forall p.\ \mu(p) \mapsto v \wedge type\text{-}rel\ (\mathbf{vrepr}(v))\ \tau \longrightarrow$$
$$is\text{-}valid\ \sigma_\tau\ p \wedge val\text{-}rel\ v\ \sigma_\tau[p]$$

where $\mathbf{vrepr}$ gives the partially-erased type for a value, similar to $\mathbf{repr}$. The state relation over all typed heaps σ_{τ_k} is $\mathcal{R}\ \sigma\ \mu \stackrel{\mathrm{def}}{=} (heap\text{-}rel\ \sigma_{\tau_1}\ \mu \wedge heap\text{-}rel\ \sigma_{\tau_2}\ \mu \wedge \dots)$.

3.3 Refinement Theorem

We state the overall top-level C refinement theorem below. In addition to the assumptions listed here, it also assumes that **corres** holds for all the foreign functions used in the program.

Theorem 2. *Let f be a* COGENT *function, with type τ and body e. Let p_m be the monadic embedding of its generated C code. Let u and v_m be arguments of appropriate type for f and p_m respectively. Then:*

$$\forall \mu\ \sigma.\ val\text{-}rel\ u\ v_m \longrightarrow \mathbf{corres}\ \mathcal{R}\ e\ (p_m\ v_m)\ (x \mapsto u)\ (x : \tau)\ \mu\ \sigma$$

Example 3. In Fig. 5, $f = flip$, $p_m = flip_C$, and $\tau = \tau' = \{\mathtt{f} :: \mathtt{U8}\}$.

3.4 Refinement Proof

This section describes the main components of the refinement proof automation, as shown in Fig. 1: the proof calculus used to relate COGENT and C programs, the generation of well-typedness theorems for COGENT, and the automated tactic that combines these two components to perform the overall refinement proof.

Refinement Calculus. Figure 6 depicts the **corres** rules in our calculus for variables, **let**, **if**, and for **take** and **put** expressions for boxed records. The full calculus is available online [1] under $\mathtt{c - refinement/COGENT\_Corres.thy}$. The proofs of the **corres** rules for compound expressions rely on Theorem 1 to infer value well-typedness.

The assumptions for these rules fall under three main groups:

1. Well-typedness assumptions; we generate typing theorems to discharge these.
2. Assumptions relating the values and mutable heaps of COGENT and C. Once a C program is read and concrete data relations (Sect. 3.2) are defined, we *specialise* the **corres** rules to simplify these assumptions.
3. **corres** assumptions on sub-expressions, discharged through our proof automation.

The rules VAR and LET correspond respectively to the two basic monadic operations **return**, which yields values, and **do** ...; ... **od**, for sequencing computations.

$$\frac{(x \mapsto v_u) \in U \qquad \textit{val-rel } v_u \, v_m}{\textbf{corres } R \; x \; (\textbf{return } v_m) \; U \; \Gamma \, \mu \, \sigma} \text{Var}$$

$$\frac{\begin{array}{c} \Gamma_1 \vdash e_1 : \tau \quad \textbf{corres } R \; e_1 \; e_1' \; U \; \Gamma_1 \, \mu \, \sigma \\ (\forall v_u \, v_m \, \mu' \, \sigma'. \; \textit{val-rel } v_u v_m \longrightarrow \textbf{corres } R \; e_2 \; (e_2' \; v_m) \; (x \mapsto v_u, U) \; (x : \tau, \Gamma_2) \; \mu' \sigma') \end{array}}{\textbf{corres } R \; (\textbf{let } x = e_1 \textbf{ in } e_2) \; (\textbf{do } e_1'; e_2' \textbf{ od}) \; U \; (\Gamma_1 \Gamma_2) \, \mu \, \sigma} \text{Let}$$

$$\frac{\begin{array}{c} \Gamma_1 \vdash c : \texttt{Bool} \qquad (\textit{bool } c' = 0 \vee \textit{bool } c' = 1) \qquad c \text{ is a Cogent boolean equal to } (\textit{bool } c' \neq 0) \\ \textbf{corres } R \; e_1 \; e_1' \; U \; \Gamma_2 \, \mu \, \sigma \qquad \textbf{corres } R \; e_2 \; e_2' \; U \; \Gamma_2 \, \mu \, \sigma \end{array}}{\begin{array}{c} \textbf{corres } R \; (\textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2) \\ (\textbf{do } x \leftarrow \textbf{condition } (\textit{bool } c' \neq 0) \; e_1' \; e_2'; \textbf{return } x \textbf{ od}) \; U \; (\Gamma_1 \Gamma_2) \, \mu \, \sigma \end{array}} \text{If}$$

$$\frac{\begin{array}{c} \exists \tau. \, (\Gamma_1 \Gamma_2) \vdash (\textbf{let } x = \textbf{put } e_1.\mathrm{f}_k := e_2 \textbf{ in } e_3) : \tau \qquad (\Gamma_1 \Gamma_2) \text{ and } \Gamma_1 \vdash e_1 : \{\overline{f_i :: \tau_i}, \mathrm{f}_k :: \tau_k^2\} \, \mathrm{w} \\ (\Gamma_1 \Gamma_2) \vdash (\textbf{put } e_1.\mathrm{f}_k := e_2) : \{\overline{f_i :: \tau_i}, \mathrm{f}_k :: \tau_k\} \, \mathrm{w} \qquad (e_1 \mapsto p\,(\texttt{Ptr } r)) \in U \qquad (e_2 \mapsto u_k') \in U \\ (\forall \, \overline{u_i} \, u_k. \, (\mu, \sigma) \in R \longrightarrow \mu(p) = \{\overline{\mathrm{f}_i = u_i}, \mathrm{f}_k = u_k\} \, r \longrightarrow p\,(\texttt{Ptr } r) \,|\, \mu : \{\overline{f_i :: \tau_i}, \mathrm{f}_k :: \tau_k^2\} \, \mathrm{w} \longrightarrow \\ p\,(\texttt{Ptr } r) \,|\, \mu(p := \{\overline{\mathrm{f}_i = u_i}, \mathrm{f}_k = u_k'\} \, r) : \{\overline{f_i :: \tau_i}, \mathrm{f}_k :: \tau_k^2\} \, \mathrm{w} \longrightarrow \\ (\textit{is-valid } \sigma \; p' \wedge (\mu(p := \{\overline{\mathrm{f}_i = u_i}, \mathrm{f}_k = u_k'\} \, r), \, h \, \sigma) \in R)) \\ (\forall \mu', \sigma'. \, \textbf{corres } R \; e_3 \; e_3' \; (e_1 \mapsto p\,(\texttt{Ptr } r), U) \; (e_1 \mapsto \{\overline{f_i :: \tau_i}, \mathrm{f}_k :: \tau_k\} \, \mathrm{w}, \Gamma_2) \; \mu' \; \sigma') \end{array}}{\begin{array}{c} \textbf{corres } R \; (\textbf{let } x = \textbf{put } e_1.\mathrm{f}_k := e_2 \textbf{ in } e_3) \\ (\textbf{do } \_ \leftarrow \textbf{guard } (\lambda \sigma. \, \textit{is-valid } \sigma \; p'); \_ \leftarrow \textbf{modify } h; \; e_3' \textbf{ od}) \; U \; (\Gamma_1 \Gamma_2) \, \mu \, \sigma \end{array}} \text{Put}$$

$$\frac{\begin{array}{c} (\Gamma_1 \Gamma_2) \vdash (\textbf{take } x \, \{\mathrm{f}_k = y\} = e_1 \textbf{ in } e_2) : \tau' \\ (\Gamma_1 \Gamma_2) \text{ and } \Gamma_1 \vdash e_1 : \{\overline{f_i :: \tau_i}, \mathrm{f}_k :: \tau_k\} \, \mathrm{w} \qquad (e_1.\mathrm{f}_k \mapsto \tau_k, e_1 \mapsto \{\overline{f_i :: \tau_i}, \mathrm{f}_k :: \tau_{\overline{k}}\} \, \mathrm{w}, \Gamma_2) \vdash e_2 : \tau' \\ p' \text{ has a C pointer type} \qquad (e_1 \mapsto p\,(\texttt{Ptr } r)) \in U \qquad \textit{val-rel } (p\,(\texttt{Ptr } r)) \; p' \\ (\forall \, \overline{u_i}, u_k. \, (\mu, \sigma) \in R \longrightarrow \mu(p) = \{\overline{\mathrm{f}_i = u_i}, \mathrm{f}_k = u_k\} \, r \longrightarrow \\ p\,(\texttt{Ptr } r) \,|\, \mu : \{\overline{f_i :: \tau_i}, \mathrm{f}_k :: \tau_k\} \, \mathrm{w} \longrightarrow \textit{is-valid } \sigma \; p' \wedge \textit{val-rel } u_k \, (f' \, \sigma)) \\ (\forall v_u \, v_m. \, \textit{val-rel } v_u \, v_m \longrightarrow \textbf{corres } R \; e_2 \; (e_2' \; v_m) \; (e_1.\mathrm{f}_k \mapsto v_u, e_1 \mapsto (p\,(\texttt{Ptr } r)), U) \\ (\mathrm{f}_k \mapsto \tau_k, e_1 \mapsto \{\overline{f_i :: \tau_i}, \mathrm{f}_k :: \tau_{\overline{k}}\} \, \mathrm{w}, \, \Gamma_2) \, \mu \, \sigma) \end{array}}{\begin{array}{c} \textbf{corres } R \; (\textbf{take } x \, \{\mathrm{f}_k = y\} = e_1 \textbf{ in } e_2) \\ (\textbf{do } \_ \leftarrow \textbf{guard } (\lambda \sigma. \, \textit{is-valid } \sigma \; p'); y' \leftarrow \textbf{gets } f'; \; e_2' \, y' \textbf{ od}) \; U \; (\Gamma_1 \Gamma_2) \, \mu \, \sigma \end{array}} \text{Take}$$

Fig. 6. Some of the important **corres** rules

Observe that Let is *compositional*: to prove that **let** $x = e_1$ **in** e_2 corresponds to **do** e_1'; e_2' **od**, we must prove that (1) e_1 corresponds to e_1' and (2) e_2 corresponds to e_2' when each are executed over corresponding results v_u and v_m (e.g. as yielded by e_1 and e_1' respectively). This compositionality, which is present in our whole calculus, significantly simplifies the automation of the refinement proof.

The If rule relates **if** c **then** e_1 **else** e_2 expressions to monadic **condition** $(\textit{bool } c' \neq 0)$ e_1' e_2' statements. It works similarly to Let, requiring an equivalence between c and $(\textit{bool } c' \neq 0)$, and correspondences between e_1 and e_1', and between e_2 and e_2'. Note that we represent booleans in C using a struct **bool** with an integer field named *bool*; we avoid C's builtin type _Bool because it may be an alias for an existing integer type like U8 and therefore indistinguishable from that integer type.

The more intricate rules in Fig. 6 are PUT and TAKE, which apply to **put** and **take** on boxed records (additional rules exist for unboxed records). Recall that boxed records are stored on the heap and are subject to the linear typing rules. These two rules are involved and contain many assumptions. They are mainly presented here to illustrate to the reader why we have a separate phase later on dedicated to simplifying them.

The PUT rule handles the correspondence between (**let** $x = $ **put** $e_1.f_k :=$ e_2 **in** e_3) expressions and (**do** _ $\leftarrow$ **guard** $(\lambda\sigma.\ \textit{is-valid}\ \sigma\ p')$; _ $\leftarrow$ **modify** h; e_3' **od**) statements. Note that unlike **let**, **if**, and **take**, **put** does not contain a continuation. Therefore, the compiler ensures that **put** expressions always appear within **let** expressions, which allows us to have a compositional rule for **put** in the same style as the other operators.

Recall that if e_1 is a pointer p, **put** updates the field f_k, of the record pointed to by p to the value of e_2. Similarly, the monadic code asserts that the corresponding p' is a valid pointer, then modifies the record at p' in h. At this stage h and *is-valid* are left unspecified, as these rules are defined generically regardless of type. Therefore, our PUT rule additionally includes a number of assumptions describing the expected properties of h and *is-valid*. In the next subsection, we specialise this rule to eliminate these assumptions.

TAKE is similar, it relates (**take** $x\ \{f_k\ = y\} = e_1$ **in** e_2) expressions and

$$(\textbf{do}\ \_ \leftarrow \textbf{guard}\ (\lambda\sigma.\ \textit{is-valid}\ \sigma\ p');\ y' \leftarrow \textbf{gets}\ f';\ e'\ y'\ \textbf{od})$$

statements. Recall that **take** removes the field f_k from e_1, binds it to a new variable y and runs e_2. The **corres** assumptions of TAKE are that (1) p' and e_1's value are related, and (2) given related values v_u and v_m, e_2 corresponds to $e_2'\ v_m$ under the extended value environment $(f_k \mapsto\ v_u, e_1 \mapsto p\,(\texttt{Ptr}\ r), U)$. We need to re-add e_1 to U because it is linear and cannot be reused.

Generating Specialised Rules. As mentioned earlier, we generate program-specific proof rules for operators involving specific C types, such as **take** and **put**. This is because the set of C types, different for each program, is shallowly embedded into Isabelle/HOL. Thus, the assumptions for rules involving those types can only be discharged once the C code has been parsed into Isabelle/HOL.

We could prove these assumptions while applying the **corres** rules, but this would be inefficient for rules that are applied many times. Thus, we generate specialised rules in a separate preprocessing phase. Implemented as an Isabelle/ML program, this phase reads the (COGENT, C) type list used for generating data relations to produce rules for the appropriate C and COGENT types.

Example 4. For the COGENT record $\{\texttt{f} :: \texttt{U8}\}$ in Fig. 5, we generate the following specialised rules for **take** and **put**:

$$(\Gamma_1\Gamma_2) \vdash (\textbf{take } x \; \{\texttt{f} = y\} = e_1 \textbf{ in } e_2) : \tau'$$
$$(\Gamma_1\Gamma_2) \vdash e_1 : \{\texttt{f}::\texttt{U8}\} \; \texttt{w} \qquad (y \mapsto \texttt{U8}, x \mapsto \{\texttt{f}::\texttt{U8}\} \; \texttt{w}, \Gamma_2) \vdash e_2 : \tau'$$
$$p' \text{ has type } rec_1 \text{ ptr} \qquad (e_1 \mapsto p\,(\texttt{Ptr } r)) \in U \qquad \textit{val-rel } (p\,(\texttt{Ptr } r))\; p'$$
$$\textit{type-rel } (\textbf{repr}(\texttt{U8})) \; \texttt{word8} \qquad \textit{type-rel } (\textbf{repr}(\{\texttt{f}::\texttt{U8}\}\; \texttt{w})) \; (rec_1 \text{ ptr})$$
$$(\forall v_u\, v_m.\; \textit{val-rel } v_u\, v_m \longrightarrow \textbf{corres } \mathcal{R}\, e_2\, (e_2'\, v_m)\, (y \mapsto v_u,\, x \mapsto (p\,(\texttt{Ptr } r)),\, U)$$
$$(y \mapsto \texttt{U8}, x \mapsto \{\texttt{f}::\texttt{U8}\}\; \texttt{w}, \Gamma_2)\; \mu\; \sigma)$$

$$\rule{10cm}{0.4pt}\; \text{Take}$$

$$\textbf{corres } \mathcal{R}\, (\textbf{take } x\; \{\texttt{f} = y\} = e_1 \textbf{ in } e_2)$$
$$(\textbf{do}\;\_\; \leftarrow \textbf{guard } (\lambda\sigma.\; \textit{is-valid } \sigma\; p');\; y' \leftarrow \textbf{gets } (\lambda\sigma.\; \sigma[p'].\texttt{f});\; e'\; y'\; \textbf{od})$$
$$U\; (\Gamma_1\Gamma_2)\; \mu\; \sigma$$

$$\exists\tau.\,(\Gamma_1\Gamma_2) \vdash (\textbf{let } x = \textbf{put } e_1.\texttt{f} := e_2 \textbf{ in } e_3) : \tau \qquad (\Gamma_1\Gamma_2) \vdash e_1 : \{\texttt{f}::\texttt{U8}\} \; \texttt{w}$$
$$\Gamma_1 \vdash (\textbf{put } e_1.\texttt{f} := e_2) : \{\texttt{f}::\texttt{U8}\}\; \texttt{w} \qquad (e_1 \mapsto p\,(\texttt{Ptr } r)) \in U \qquad (e_2 \mapsto v) \in U$$
$$\textit{val-rel } (p\,(\texttt{Ptr } r))\; p' \qquad \textit{type-rel } (\textbf{repr}(\{\texttt{f}::\texttt{U8}\}\; \texttt{w}))\; (rec_1 \text{ ptr}) \qquad \textit{val-rel } v\; v'$$
$$(\forall\mu', \sigma'.\; \textbf{corres } \mathcal{R}\, e_3\, e_3'\; (e_1 \mapsto p\,(\texttt{Ptr } r),\, U)\; (e_1 \mapsto \{\texttt{f}::\texttt{U8}\}\; \texttt{w}, \Gamma_2)\; \mu'\; \sigma')$$

$$\rule{10cm}{0.4pt}\; \text{Put}$$

$$\textbf{corres } \mathcal{R}\, (\textbf{let } x = \textbf{put } e_1.\texttt{f} := e_2 \textbf{ in } e_3)$$
$$(\textbf{do}\;\_\; \leftarrow \textbf{guard } (\lambda\sigma.\; \textit{is-valid } \sigma\; p');\; \_ \leftarrow \textbf{modify } (\lambda\sigma.\; \sigma[p'].\texttt{f} := v');\; e_3'\; \textbf{od})$$
$$U\; (\Gamma_1\Gamma_2)\; \mu\; \sigma$$

Note that the cumbersome record-update assumptions from Fig. 6 have been reduced to *val-rel* and *type-rel* statements. This is only possible after we obtain the concrete program and its data relations. We also instantiate the state relation $\mathcal{R}$ and show that **take** and **put** preserve it, allowing us to simplify the heap-update assumptions.

Well-Typedness. The COGENT compiler proves, via an automated Isabelle tactic, that the deep embedding of the input program is well-typed. Specifically, it shows for each function f with argument x, body e, and type $\tau_1 \rightarrow \tau_2$, that $x \mapsto \tau_1 \vdash e : \tau_2$.

Recall that the type system is substructural, and that proving refinement requires access to the typing judgements for each sub-expression of the program. To solve this, the COGENT compiler instructs Isabelle to store all intermediate typing judgements established during type checking. These theorems are stored in a tree structure, isomorphic to the COGENT program's type derivation tree. Each node is a typing theorem for a program sub-expression, and can be retrieved by the refinement proof tactic as it descends into the program.

Proof Automation. The core of our refinement prover is an Isabelle/ML tactic that proves the **corres** refinement theorem (Sect. 3.3) for each COGENT function in the program, by applying the **corres** rules previously proven, both generic and specialised (Sect. 3.4). This algorithm is straightforward as our rules are syntax-directed.

The tactic also expands definitions of *val-rel* and *type-rel* (Sect. 3.2) in order to discharge data relation assumptions in those **corres** rules, and retrieves the type derivation tree for the given COGENT function to discharge all well-typedness assumptions.

Example 5. For *flip* in Fig. 5, we wish to prove the refinement theorem

> **corres** $\mathcal{R}$ *flip* ($flip_C v_m$) ($x \mapsto u$) ($x : \{\texttt{f} :: \texttt{U8}\}$) $\mu\ \sigma$
>
> or after unfolding
>
> **corres** $\mathcal{R}$ (**take** x' $\{\texttt{f} = y\} = x$ **in** ...)
>   (**do guard** ($\lambda\sigma.$ *is-valid* $\sigma\ x$); $y \leftarrow$ **gets** ($\lambda\sigma.\ \sigma[r].$f); ... **od**)
>   ($x \mapsto u$) ($x : \{\texttt{f} :: \texttt{U8}\}$) $\mu\ \sigma$

The first step of the proof applies the specialised **take** rule for $\{\texttt{f} :: \texttt{U8}\}$ (Sect. 3.4). After discharging its typing and *val-rel* assumptions, we are left with a **corres** obligation on the remainder of the function, which can in turn be solved using the other proof rules.

Our tactic can be used easily for single functions, but extending it to whole programs required significant proof engineering effort, as we must handle function calls both to externally-defined C functions and to (potentially higher-order) COGENT functions.

Foreign functions. COGENT code depends on calls to foreign C functions to perform loops and I/O. Our framework requires these functions to be well-behaved, i.e. they respect COGENT's termination order and do not break the COGENT type system (e.g. by modifying variables they do not have access to).

Foreign functions are user-supplied and not verified automatically. Thus, when proving refinement theorems for COGENT code that calls these functions, we automatically insert assumptions that they are well-behaved. These assumptions remain until they are resolved by manual verification.

Whole-program refinement. COGENT is a total language and does not permit recursion, so we have, in principle, a well-ordering on function calls in any program. However, for higher-order functions, this well-ordering is non-obvious and difficult to work with.

In practice, most function calls in systems code are direct calls to first-order functions. For such functions, we can simply prove the **corres** theorems in bottom-up fashion, starting from the leaf functions and ending at the top-level functions.

There is one major exception: COGENT code cannot express loops using only first-order functions. Our COGENT programs use iteration *combinators*, which are second-order foreign functions that take a COGENT function pointer as the loop body (similar to the *map* or *fold* combinators in functional programming).

Therefore, our framework also supports second-order calls to foreign functions. Before assuming **corres** for these functions, we first prove **corres** for the argument function (i.e. the loop body).

This technique allows us to automate refinement for code with first- and second-order calls. While this restriction means that not all COGENT programs can be verified in our framework, we developed COGENT code for two file system drivers [2] in this fragment, demonstrating that substantial programs can be written in this subset.

4   Related Work

To date, the largest trustworthy compilation projects are the CompCert [7] C compiler and the CakeML [6] ML environment. In contrast to COGENT, they compile general-purpose programming languages and rely more heavily on verified compilation passes.

CompCert translates (a subset of) C to binary while our compiler translates the functional COGENT language to C. CompCert's core compilation process is verified and its optimisation passes are validated; the compiler executable itself is extracted from Coq into Caml. There is ongoing work to validate the Coq code extraction process and the Caml compiler for CompCert.

We chose to use certificates for most of COGENT's compiler passes, because our proof tools for C run in Isabelle directly, and our COGENT compiler is written in Haskell, which does not have a formal semantics nor a verified runtime at present. On one hand, processing the certificates is time-intensive. On the other hand, we do not need to trust the code extractor, nor the runtime for the extracted language. We do need to either trust the C compiler or use a verified one.

COGENT is closer to CakeML in that it is a high-level source language. However, COGENT targets a different application area. CakeML is a Turing-complete dialect of ML with complex semantics, and is suited for application code. On the other hand, COGENT is a restricted language of total functions with simple semantics that facilitate equational reasoning. COGENT avoids the need for a large runtime and a garbage collector so it can be used for embedded systems code, especially layered systems code with minimal sharing such as the control code of filesystems or network protocol stacks.

5   Take Away Lessons and Future Work

When designing the certifying compiler, we made a trade-off by writing the COGENT compiler tool-chain in Haskell, while the proof component was written in Isabelle's Standard ML environment. This divide allows the COGENT tool-chain to be used outside the theorem prover, and allows the proof tools to build on the existing C parser and AutoCorres framework.

On the other hand, this choice leads to complexity in designing the interface between these components. This is illustrated by our well-typedness proof of Sect. 3.4, where the COGENT compiler generates a certificate with the necessary type derivation hints. Initially, we used a naïve format consisting of the entire derivation tree, resulting in gigabyte-sized certificates. We implemented various compression techniques to reduce the certificates to a reasonable size (a few megabytes). It is possible to avoid these certificates entirely by duplicating the type inference algorithm in Isabelle/ML, but this would increase the code maintenance burden.

Even though reusing the C parser and AutoCorres is desirable, they take a long time to process our verbose generated C code. They take a total of 12 CPU

hours to translate the `ext2` filesystem into a monadic embedding and they take 32 CPU hours when applied to `BilbyFs`. Further proof optimisation is needed.

Optimisation of the generated code is another topic for future work. High-level COGENT-to-COGENT optimisations will be easy, as they can be verified over the shallow embedding of COGENT using equational rewriting. For instance, we verified A-normalisation using rewriting; while it is not an optimisation, it is an example of a code transformation that does not affect the COGENT-to-C proof. For low-level optimisations, we rely on the C compiler so as not to complicate our syntax-directed proof approach.

6   Conclusions

We developed a compositional refinement calculus and proof tools to create a fully automatic refinement certificate from COGENT's update semantics to C, including the use of *partial type erasure* to relate COGENT's expressive types to simpler C types. This refinement certificate is the most involved step in the full automation of the overall compiler certificate. Through the co-generation of code and proofs, our framework significantly reduces the cost of reasoning about efficient C code, by automatically discharging cumbersome safety obligations, and providing an embedding more amenable to verification. Our framework has been applied successfully to two real-world file-systems.

References

1. COGENT material (2016). https://github.com/NICTA/cogent/tree/itp_2016
2. Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., Heiser, G.: Cogent: Verifying high-assurance file system implementations. In: ASPLOS, pp. 175–188, April 2016
3. Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 167–182. Springer, Heidelberg (2008)
4. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: formal verification of C code without the pain. In: PLDI, pp. 429–439 (June 2014)
5. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: SOSP, pp. 207–220 (October 2009)
6. Kumar, R., Myreen, M., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: POPL, pp. 179–191, January 2014
7. Leroy, X.: Formal verification of a realistic compiler. CACM **52**(7), 107–115 (2009)
8. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
9. O'Connor, L., Chen, Z., Rizkallah, C., Amani, S., Lim, J., Murray, T., Nagashima, Y., Sewell, T., Klein, G.: Refinement through restraint: bringing down the cost of verification. In: ICFP (to appear, 2016)

10. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. SIGPLAN Lisp Pointers **V**(1), 288–298 (January 1992)
11. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
12. Sewell, T., Myreen, M., Klein, G.: Translation validation for a verified OS kernel. In: PLDI, pp. 471–481 (June 2013)
13. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: POPL, pp. 97–108 (January 2007)