Refinement Through Restraint: Bringing Down the Cost of Verification

Anonymous

Abstract

We present a framework aimed at significantly reducing the cost of verifying certain classes of systems software, such as file systems. Our framework allows for equational reasoning about systems code written in our new language, COGENT. COGENT is a restricted, polymorphic, higher-order, and purely functional language with linear types and without the need for a trusted runtime or garbage collector. Linear types allow us to assign two semantics to the language: one imperative, suitable for efficient C code generation; and one functional, suitable for equational reasoning and verification. As COGENT is a restricted language, it is designed to easily interoperate with existing C functions and to connect to existing C verification frameworks.

Our framework is based on *certifying compilation*: For a well-typed COGENT program, our compiler produces C code, a high-level shallow embedding of its semantics in Isabelle/HOL, and a proof that the C code correctly refines this embedding. Thus one can reason about the full semantics of real-world systems code productively and equationally, while retaining the interoperability and leanness of C. The compiler certificate is a series of language-level proofs and per-program translation validation phases, combined into one coherent top-level theorem in Isabelle/HOL.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software / Program Verification—Formal methods; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; D.3.2 [*Programming Languages*]: Language Classification—Applicative (functional) languages

Keywords verification; semantics; linear types; programming languages; file systems; Isabelle/HOL

1. Introduction

Imagine writing low-level systems code in a purely functional language and then reasoning about this code equationally and productively in an interactive theorem prover. Imagine doing this without the need for a trusted compiler, runtime or garbage collector and letting this code interoperate with native C parts of the system, including your own efficiently implemented and formally verified additional data types and operations.

Our verification framework achieves this goal through the certifying compiler of COGENT: a high-level, pure, polymorphic, functional language with linear types, specifically designed for certain classes of systems code such as file systems. For a given well-typed COGENT program, the compiler will produce a high-level shallow embedding of the program's semantics in Isabelle [Nipkow and Klein 2014], and a proof that connects this shallow embedding to the compiler generated C code: any property proved of the shallow embedding is guaranteed to hold for the generated C.

The compilation target is C, because C is the language most existing systems code is written in, and because with the advent of tools like CompCert [Leroy 2006, 2009] and gcc translation validation [Sewell et al. 2013], C is now a language with well understood semantics and existing formal verification infrastructure.

If C is so great, why not verify C systems code directly? After all, there is an ever growing list of successes [Beringer et al. 2015; Gu et al. 2015; Klein et al. 2009, 2014] in this space. The reason is simple: verification of manually written C programs remains expensive. Just as high-level languages increase programmer productivity, they should also increase verification productivity. COGENT is specifically designed with a verification-friendly high-level semantics. This makes the difference between imperative and functional verification. It is pointer fiddling and undefined behaviour guards in C versus abstract functional objects and equations in COGENT. An imperative VCG [Dijkstra 1997] for C must overwhelm the prover with detail, while the abstraction and type system of COGENT enable the use of far stronger existing automation for *high-level* proofs.

The state of the art for certifying compilation of functional languages is CakeML [Kumar et al. 2014], which covers an entire ML dialect. COGENT is targeted at a substantially different point in the design space. CakeML includes a verified runtime and garbage collector, while COGENT works hard to avoid these so it can be applicable to low-level embedded systems code. CakeML covers full Turing-complete ML with complex, stateful semantics, which works well for code written in theorem provers. COGENT is a restricted language of total functions with intentionally simple, pure semantics that are easy to reason about equationally. Even though COGENT is newer than CakeML, programs written in COGENT have already been verified, thanks to its simple semantics, whereas CakeML has only been used for small examples. CakeML is great for application code; COGENT is great for systems code, especially layered systems code with minimal sharing such as the control code of file systems or network protocol stacks. COGENT is not designed for systems code with closely-coupled, cross-cutting sharing, such as microkernels.

COGENT's main restrictions are the (purposeful) lack of recursion and iteration and its linear type system. The former ensures totality, which is important for both systems code correctness as well as for a simple shallow representation in higher-order logic. The latter is important for safe memory management and for enabling a transition from an imperative C-style semantics, suitable for code generation, to a functional semantics, suitable for equational reasoning and verification.

Even in the restricted target domains of COGENT, real programs will contain some amount of iteration. This is where COGENT's integrated foreign function interface comes in: the engineer provides her own verified data types and iterator interfaces in C and uses them seamlessly in COGENT, including in formal reasoning. Our proof guarantees that the verification of combined C-COGENT code bases has no room for unsoundness.

COGENT is restricted, but it is by no means a toy language. Amani et al. [2016] use COGENT to successfully implement two efficient

full-scale Linux file systems — the standard Linux `ext2` and the BilbyFs Flash file system [Keller et al. 2013], and prove two core functional correctness properties of BilbyFs. This illustrates that COGENT is suitable both for implementation and proofs; dramatically reducing the cost of verifying correctness of practical file systems. This is beneficial in its own right, as file systems constitute the second largest proportion of OS code, and have among the highest density of faults [Palix et al. 2011]. The benefits of this language-based approach for file system verification were conjectured by Keller et al. [2013] and are confirmed by our work. Our language is restricted, but not specific to the file systems domain. This leads us to believe that our language-based approach for simplifying verification will extend in the near future to other domains, either with COGENT directly, or with languages that make different trade-offs suitable for different types of software.

The contribution of this paper is a framework that significantly reduces the cost of formal verification for important classes of efficient systems code. It uses a language-based approach for automatically co-generating code and proofs. Specifically, our framework relies on the following contributions:

a) the COGENT language, its formal semantics and its certifying compiler;

b) the formal machine-checked proof for switching from imperative update semantics to functional value semantics for a full-featured functional language, justified by linear types (§3). We build upon well-known theoretical results about linear types, accounting for pointers and heap allocation;

c) the formal description of the assumptions required for C code imported via a foreign function interface to maintain the guarantees of the linear type system (§3.3.2);

d) the top-level compiler certificate (§4.1); and

e) the verification stages that make up the correctness theorem (§4), including automated refinement calculi, formally verified type checking, A-normalisation, and monomorphisation. One of these stages is presented in detail elsewhere [Anonymous 2016b], but summarised in §4.3.

We present two case studies which apply our verification framework to two file system implementations (from Amani et al. [2016]) as well as some of the lessons learned in this project in §5. The compiler, Isabelle proofs, and file systems are available online [Anonymous 2016a].

2. Overview

Our certifying compiler constructs a chain of proofs relating COGENT to efficient C code, such that a proof engineer can reason equationally about its semantics in Isabelle/HOL and apply the compiler theorem to derive properties about the generated C code. Formally, the certificate theorem is a refinement statement between the shallow embedding and the C code. This generated C code falls into the subset that can be compiled by CompCert, or validated by the gcc translation validation tool of Sewell et al. [2013], whose theorem would compose directly with our compiler certificate.[1]

Shallow embeddings are nice for humans, but do not provide much syntactic structure for constructing a compiler theorem. Therefore the compiler also generates a deep embedding for each COGENT program to use in the internal proof chain. There are two semantics for this deep embedding: (1) a formal functional *value semantics* where programs evaluate to values and (2) a formal imperative *update semantics* where programs manipulate references to mutable global state.

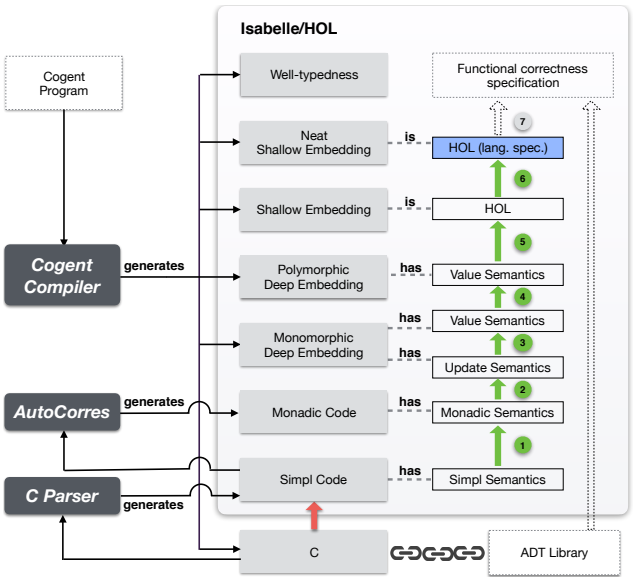

Figure 1: A detailed overview of the verification chain

Fig. 1 shows an overview of the program representations generated by the compiler and the break-down of the automatic refinement proof that makes up the compiler certificate. The program representations are, from the bottom of Fig. 1: the C code, the semantics of the C code expressed in Isabelle/Simpl [Schirmer 2006], the same expressed as a monadic functional program [Greenaway et al. 2012, 2014], a monomorphic A-normal [Sabry and Felleisen 1992] deep embedding of the COGENT program, a polymorphic A-normal deep embedding of the same, an A-normal shallow embedding, and finally a 'neat' shallow embedding of the COGENT program that is syntactically close to the COGENT source code. Several theorems rely on the COGENT program being well-typed, which we prove automatically using type inference information from the compiler.

The solid arrows on the right-hand side of the figure represent refinement proofs and the arrow labels correspond to the numbers in the following description. The only arrow that is not formally verified is the one crossing from C code into Isabelle/HOL at the bottom of Fig. 1 — this is the C parser, which is a mature verification tool used in a number of large-scale verifications [Klein et al. 2009]. As mentioned, it could be checked by translation validation.

We outline each intermediate theorem, starting with Simpl at the bottom. For well-typed COGENT programs, we prove:
(1) The Simpl code produced by the C parser corresponds to a monadic representation of the C code. The proof is generated using an adjusted version of AutoCorres [Greenaway et al. 2012].
(2) The monadic code terminates and is a refinement of the update semantics of the monomorphic COGENT deep embedding [Anonymous 2016b].
(3) If a COGENT deep embedding evaluates in the update semantics, it evaluates to the same result in the value semantics. We mechanise this consequence of linear type systems, significantly extending known theoretical results to accommodate heap data structures, on a real full-scale language.
(4) If a monomorphic COGENT deep embedding evaluates in the value semantics then the polymorphic deep embedding evaluates equivalently in the value semantics.

[1] At the time of writing, COGENT's occasionally larger stack frames lead to gcc emitting `memcpy()` calls that, while conceptually straightforward to handle, the translation validator does not yet cover.

(5) If the polymorphic COGENT deep embedding evaluates in the value semantics then the COGENT shallow embedding evaluates to a corresponding shallow Isabelle/HOL value.
(6) The shallow embedding is (extensionally) equal in Isabelle/HOL to a neater shallow embedding, more convenient for human reasoning. The neat shallow embedding corresponds to the COGENT code before being A-normalised.

In §4 we define in more detail the relations that formally link the values (and states, when applicable) that these programs evaluate to. Steps (3) and (4) are general properties about the language and we therefore prove them manually once and for all. Steps (1), (2), (5), and (6) are generated automatically for every program. The proof for step (1) is generated by AutoCorres. For steps (2) and (5) we define compositional refinement calculi that ease the automation of these proofs. Step (6), the correctness of A-normalisation, is straightforward to prove via rewriting because at this stage we can already use equational reasoning.

The refinement proofs state that every behaviour exhibited by the C code can also be exhibited by the COGENT code and, furthermore, that the C code is always well-defined, including that e.g. the generated C code never dereferences a null pointer, and never causes signed overflow. It also implies that the generated C code is type-safe and memory-safe, meaning the code will never try to dereference an invalid pointer, or try to dereference two aliasing pointers of incompatible types. In conjunction with the COGENT typing proofs, generated by the COGENT compiler for the input program, we get additional guarantees that the generated code handles all error cases, is free of memory leaks, and never *double-free*s a pointer. These points will be formally established in §3 and §4.

Amani et al. [2016] demonstrate how to reason on top of the COGENT formal specification in Isabelle/HOL (arrow (7) in Fig. 1), leveraging COGENT's purely functional semantics. Reasoning here is simpler than reasoning on C code directly because the COGENT semantics is represented as pure functions in HOL, rather that complex, stateful embeddings. This purely functional representation makes it possible to reason equationally about COGENT code via Isabelle's powerful rewriting engine. If desired, one could formalise domain-specific program logics, such as the Crash Hoare Logic of Chen et al. [2015], on top of our generated COGENT specification. This would enable further improvements to verification productivity by simplifying reasoning about domain-specific properties.

3. COGENT Language

Fig. 2 contains an excerpt of a COGENT `ext2` implementation, which demonstrates the use of most language features. Line 1 shows the COGENT side of the foreign function interface. It declares an abstract COGENT data type `ExSt`, implemented in C. Line 2 shows a parametric abstract type, and line 9 a corresponding abstract function `uarray_create()`, also implemented in C. `uarray_create()` is polymorphic, with a kind constraint `E` (see §3.1) on type argument `a`.
The integration of foreign functions is seamless on the COGENT side. It naturally puts requirements on the C code: it must respect the invariants of COGENT's linear type system, terminate, and implement the user-supplied semantics that appear in the corresponding shallow embedding of the COGENT program in Isabelle/HOL. Ideally the user provides a proof to discharge the corresponding assumption of the compiler certificate.

Abstract functions can be higher-order and provide the iteration constructs that are intentionally left out from core COGENT. E.g. line 20, `uarray_map_no_break()` implements a map iterator for arrays. Amani et al. [2016] found that for both of their file system implementations, a small common library of iterators for types like arrays and Linux's existing red-black tree implementation was sufficient.

```
1  type ExSt
2  type UArray a
3  type Opt a = <None () | Some a>
4  type Node = #{mbuf:Opt Buf, ptr:U32, fr:U32, to:U32}
5  type Acc = (ExSt, FsSt, VfsInode)
6  type Cnt = (UArray Node,
7    (U32, Node, Acc, U32, UArray Node) -> (Node, Acc))
8
9  uarray_create: all (a :< E). (ExSt, U32)
10   -> <Success (ExSt, UArray a) | Err ExSt>
11
12 ext2_free_branch: (U32, Node, Acc, U32)
13   -> (Node, Acc, <Expd Cnt | Iter ()>
14 ext2_free_branch (depth,nd,(ex,fs,inode),mdep) =
15   if depth < mdep
16     then uarray_create[Node] (ex,nd.to-nd.fr)!nd
17       | Success (ex, children) =>
18         let nd_t { mbuf } = nd
19         and (children, (ex, inode, _, mbuf)) =
20           uarray_map_no_break #{
21             arr  = children,
22             f    = ext2_free_branch_entry,
23             acc  = (ex, inode, node_t.fr, mbuf),
24             ... } !nd_t
25         and nd = nd_t { mbuf }
26         in (nd, (ex, fs, inode),
27           Expd (children, ext2_free_branch_cleanup))
28       | Err ex -> (nd, (ex,fs,inode), Iter ())
29     else ...
```

Figure 2: COGENT example

Lines 3–7 show basic type constructors and declarations of variants, records and tuples using type variables and the primitive type `U32`. E.g. type `Cnt` is defined as a pair of `UArray Node` and a function type. Types in COGENT are structural [Pierce 2002], i.e. types with the same structure are equal. Moreover, line 16 calls the abstract polymorphic function `uarray_create()`, instantiated with type argument `Node`. The `!nd` notation temporarily turns a linear object of type `Node` into a read-only one (see §3.2.1). The two basic, non-linear fields `to` and `fr` in type `Node` can directly be accessed read-only using projection functions. Line 17 and 28 are pattern matches on the result of the function invocation. Line 18 shows surface syntax for COGENT's linear **take** construct (see §3.2.3), accessing and binding the `mbuf` field of `nd` to the name `mbuf` (punning as in Haskell), as well as binding the rest of the record to the name `nd_t`. The linear type system tracks that the field `mbuf` is logically absent in `nd_t`. It also tracks that `nd` on line 18 was used, so cannot be accessed again. Thus the programmer is safe to bind a new object to the same name `nd` (on line 25) without worrying about name shadowing. Line 25 shows surface syntax for **put**, the dual to **take**, which re-establishes the `mbuf` fields in the example.

3.1 Types and Kinding

Wadler [1990] first noted that linear types can be used as a way to safely model mutable state and similar effects while maintaining a purely functional semantics. Hofmann [2000] later proved Wadler's intuition by showing that, for a linear language, imperative C code without heap-allocated data structures can implement a simple set-theoretic semantics.

Many languages such as Rust [Rust] and Vault [DeLine and Fähndrich 2001] use linear types to manage resources such as memory and eliminate the need for run-time support such as garbage collection.

We use linear types for both of these reasons: we do not require a garbage collector, and we assign to COGENT programs an equational, purely functional semantics implemented via mutable state internally. Unlike Hofmann, our mechanised proof of the correspondence between these two semantics takes heap-allocated data

$$
\begin{array}{llcl}
\text{prim. types} & t & ::= & \texttt{U8} \mid \texttt{U16} \mid \texttt{U32} \mid \texttt{U64} \mid \texttt{Bool} \\
\text{types} & \tau, \rho & ::= & \alpha \mid \alpha! \mid () \\
 & & \mid & t \mid \texttt{T}\ \overline{\tau}\ m \mid \tau \to \rho \\
 & & \mid & \langle \overline{\mathsf{C}\ \tau} \rangle \mid \{\overline{\mathrm{f}\ ::\ \tau^?}\}\ m \\
\text{field types} & \tau^? & ::= & \tau \mid \not\tau \\
\text{permissions} & \mathcal{P} & = & \{\mathsf{D}, \mathsf{S}, \mathsf{E}\} \\
\text{kinds} & \kappa & \subseteq & \mathcal{P} \\
\text{polytypes} & \pi & ::= & \forall (\overline{\alpha ::_{\mathrm{K}} \kappa}).\ \tau \\
\text{modes} & m & ::= & \texttt{r} \mid \texttt{w} \mid \texttt{u} \\
\text{type variables} & & \ni & \alpha, \beta \\
\text{abs. type names} & & \ni & \texttt{T}, \texttt{U} \\
\text{kind context} & \Delta & ::= & \overline{\alpha :_{\mathrm{K}} \kappa} \\
\text{type context} & \Gamma & ::= & \overline{x : \tau}
\end{array}
$$

$$
\boxed{\Delta \vdash \Gamma_1 \overset{\text{weak}}{\rightsquigarrow} \Gamma_2} \qquad \frac{\text{for each } i\text{: } \Delta \vdash \tau_i :_{\mathrm{K}} \{\mathsf{D}\}}{\Delta \vdash \overline{x_i : \tau_i}, \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma}
$$

$$
\boxed{\Delta \vdash \Gamma_1 \rightsquigarrow \Gamma_2 \boxplus \Gamma_3} \qquad \frac{\text{for each } i\text{: } \Delta \vdash \tau_i :_{\mathrm{K}} \{\mathsf{S}\}}{\Delta \vdash \overline{x_i : \tau_i}, \Gamma_1, \Gamma_2 \rightsquigarrow \overline{x_i : \tau_i}, \Gamma_1 \boxplus \overline{x_i : \tau_i}, \Gamma_2}
$$

(overbar indicates lists, i.e. zero or more)

Figure 3: Type Structure of Cogent & structural context operations

$$
\boxed{\Delta \vdash \tau :_{\mathrm{K}} \kappa}
$$

$$
\frac{}{\Delta \vdash () :_{\mathrm{K}} \kappa}\textsc{KUnit} \qquad \frac{}{\Delta \vdash t :_{\mathrm{K}} \kappa}\textsc{KPrim} \qquad \frac{}{\Delta \vdash \tau \to \rho :_{\mathrm{K}} \kappa}\textsc{KFun}
$$

$$
\frac{(\alpha :_{\mathrm{K}} \kappa') \in \Delta \quad \kappa \subseteq \kappa'}{\Delta \vdash \alpha :_{\mathrm{K}} \kappa}\textsc{KVar} \qquad \frac{(\alpha :_{\mathrm{K}} \kappa') \in \Delta \quad \kappa \subseteq \mathbf{bang}(\kappa')}{\Delta \vdash \alpha! :_{\mathrm{K}} \kappa}\textsc{KVar!}
$$

$$
\frac{\text{for each } i\text{: } \Delta \vdash \tau_i :_{\mathrm{K}} \kappa}{\Delta \vdash \langle \overline{\mathsf{C}_i\ \tau_i} \rangle :_{\mathrm{K}} \kappa}\textsc{KVariant}
$$

$$
\frac{\begin{array}{c} m :_{\mathrm{K}} \kappa' \qquad \kappa \subseteq \kappa' \\ \text{for each } i\text{: } \Delta \vdash \tau_i :_{\mathrm{K}} \kappa \end{array}}{\Delta \vdash \texttt{T}\ \overline{\tau_i}\ m :_{\mathrm{K}} \kappa}\textsc{KAbs} \qquad \frac{\begin{array}{c} m :_{\mathrm{K}} \kappa' \qquad \kappa \subseteq \kappa' \\ \text{for each } \tau_i \text{ not taken: } \Delta \vdash \tau_i :_{\mathrm{K}} \kappa \end{array}}{\Delta \vdash \{\overline{\mathrm{f}_i\ ::\ \tau_i^?}\}\ m :_{\mathrm{K}} \kappa}\textsc{KRec}
$$

$$
\boxed{m :_{\mathrm{K}} \kappa}
$$

$$
\texttt{r} :_{\mathrm{K}} \{\mathsf{D}, \mathsf{S}\} \qquad \texttt{w} :_{\mathrm{K}} \{\mathsf{E}\} \qquad \texttt{u} :_{\mathrm{K}} \{\mathsf{D}, \mathsf{S}, \mathsf{E}\}
$$

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

$$
\begin{array}{lcl}
\mathbf{bang}(\alpha) & = & \alpha! \\
\mathbf{bang}(\alpha!) & = & \alpha! \\
\mathbf{bang}(()) & = & () \\
\mathbf{bang}(t) & = & t \\
\mathbf{bang}(\texttt{T}\ \overline{\tau_i}\ m) & = & \texttt{T}\ \overline{\mathbf{bang}(\tau_i)}\ \mathbf{bang}(m) \\
\mathbf{bang}(\tau \to \rho) & = & \tau \to \rho \\
\mathbf{bang}(\langle \overline{\mathsf{C}_i\ \tau_i} \rangle) & = & \langle \overline{\mathsf{C}_i\ \mathbf{bang}(\tau_i)} \rangle \\
\mathbf{bang}(\{\overline{\mathrm{f}_i\ ::\ \tau_i^?}\}\ m) & = & \{\overline{\mathrm{f}_i\ ::\ \mathbf{bang}(\tau_i^?)}\}\ \mathbf{bang}(m)
\end{array}
$$

$$
\boxed{\mathbf{bang}(\cdot) : \kappa \to \kappa}
$$

$$
\mathbf{bang}(\kappa) = \begin{cases} \kappa & \text{if } \{\mathsf{D}, \mathsf{S}\} \subseteq \kappa \\ \{\mathsf{D}, \mathsf{S}\} & \text{otherwise} \end{cases}
$$

$$
\boxed{\mathbf{bang}(\cdot) : m \to m}
$$

$$
\begin{array}{lcl}
\mathbf{bang}(\texttt{r}) & = & \texttt{r} \\
\mathbf{bang}(\texttt{w}) & = & \texttt{r} \\
\mathbf{bang}(\texttt{u}) & = & \texttt{u}
\end{array}
$$

Figure 4: Kinding rules for Cogent types and the **bang**(·) operator

structures into account, using our *heap footprint* annotation technique (§3.3.1).

The type structure and associated syntax of Cogent is presented in Fig. 3. Our type system is polymorphic, but we restrict this polymorphism to be rank-1 and predicative, in the style of ML, to permit easy implementation by specialisation with minimal performance penalty.

To ease implementation, and to eliminate any direct dependency on a heap allocator, we require that all functions be defined on the top-level, disallowing closures. Any top-level function can be shared freely, as they cannot capture *any* local variables, let alone linear ones.

We include a set of primitive integer types (U8, U16, etc.). Records $\{\overline{\mathrm{f}\ ::\ \tau^?}\}\ m$ comprise (1) a sequence of fields $f\ ::\ \tau^?$, where $\not\tau$ is the type on an inaccessible field, and (2) a mode m (see §3.2.3 and §3.1.1 for a more detailed description). We also have polymorphic variants $\langle \overline{\mathsf{C}\ \tau} \rangle$, a generalised sum type in the style of OCaml , the mechanics of which are briefly described in §3.2.2. Abstract types $\texttt{T}\ \overline{\tau}\ m$ are also parametrised by modes. We omit product types from this presentation; they are desugared into unboxed records.

Similarly to other polymorphic, substructural type systems such as λ_{URAL} [Ahmed et al. 2005] and System $\mathcal{F}^\circ$ [Mazurak et al. 2010], we use *kinds* to determine if a type may be freely shared or discarded. Kinds in Cogent are sets of *permissions*, denoting whether a variable of that type may be discarded without being used (D), shared freely and used multiple times (S), or safely bound in a **let**! expression (E). A *linear* type, values of which must be used exactly once, has a kind that excludes D and S, and so forbids it being discarded or shared. We discuss **let**! expressions in §3.1.2.

We explicitly represent the context operations of weakening and contraction, normally relegated to structural rules, as explicit judgements: $\Delta \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'$ for weakening (discarding assumptions) and $\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$ for contraction (duplicating them). The rules for these judgements are presented in Fig. 3. For a typing assumption to be discarded (respectively duplicated), the type must have kind {D} (resp. {S}).

The full kinding rules for the types of Cogent are given in Fig. 4. Basic types such as () or U8, as well as functions, are simply passed by value and do not contain any heap references, so they may be given any kind. Kinding for structures and abstract functions is discussed shortly in §3.1.1.

A type may have multiple kinds, as a nonlinear type assumption may be used linearly, never being shared and being used exactly once. Therefore, a type with a permissive kind, such as {D, S}, would be an acceptable instantiation of a type variable of kind ∅, as we are free to *waive* permissions that are included in a kind. We can prove formally by straightforward rule induction:

Lemma 1 (Waiving rights). *If* $\Delta \vdash \tau :_{\mathrm{K}} \kappa$ *and* $\kappa' \subseteq \kappa$, *then* $\Delta \vdash \tau :_{\mathrm{K}} \kappa'$.

This result allows for a simple kind-checking algorithm, not immediately apparent from the rules. For example, the maximal kind of an unboxed structure with two fields of type τ_1 and τ_2 respectively can be computed by taking the intersection of the computed maximal kinds of τ_1 and τ_2. This result ensures that this intersection is also a valid kind for τ_1 and τ_2.

3.1.1 Kinding for Records and Abstract Types

Recall that COGENT may be extended with *abstract types*, implemented in C, which we write as $\mathtt{T}\ \overline{\tau_i}\ m$ in our formalisation. We allow abstract types to take any number of *type parameters* τ_i, where each specific instance corresponds to a distinct C type. For example, a `List` abstract type, parameterised by its element type, would correspond to a family of C `List` types, each one specialised to a particular concrete element type. Because the implementations of these types are user supplied, the user is free to specialise implementations based on these type parameters, for example representing an array of boolean values as a bitstring, so long as they can show that every different operation implementation is a refinement of the same user-supplied CDSL semantics for that operation.

Values of abstract types may be represented by references to heap data structures. Specifically, an abstract type or structure is stored on the heap when its associated *storage mode* m is not "`u`". For boxed records and abstract types, the storage mode distinguishes between those that are "`w`" vs. "`r`". The same is true for record types, written $\{\overline{\mathsf{f}\ ::\ \tau^?}\}\ m$, which are discussed in more detail in §3.2.3.

The storage mode m affects the maximal kind that can be assigned to the type. For example, an unboxed structure with two components of type `U8` is freely shareable, but if the structure is instead stored on the heap, then a writable reference to that structure must be linear. Thus, the type given to such references has the "`w`" mode, whose kind is $\{\mathsf{E}\}$, thereby preventing such a reference from being assigned a nonlinear kind such as $\{\mathsf{D}, \mathsf{S}\}$.

3.1.2 Kinding and bang

We allow linear values to be shared read-only in a limited scope, an idea first explored by Wadler [1990]. This is useful for practical programming in a language with linear types, as it makes our types more informative. For example, to write a function to determine the size of a (linear) buffer object, a naive approach would be to write a function:

$$\mathsf{size} : \mathtt{Buf} \to \mathtt{U32} \times \mathtt{Buf}$$

This function has a cumbersome additional return value just so that the linear argument is not discarded. Further, the type above does not express the fact that the input buffer and output buffer are identical — this would need to be established by additional proof. To address this problem, we include a type operator **bang**$(\cdot)$, in the style of Wadler's ! operator, which changes all writable modes in a type to read-only ones. The full definition of **bang**$(\cdot)$ is in Fig. 4. We can therefore write the type of our function as:

$$\mathsf{size} : \mathbf{bang}(\mathtt{Buf}) \to \mathtt{U32}$$

For any valid type τ, the kind of **bang**(τ) will be nonlinear, which means that our size function no longer needs to be encumbered by the extra return value. This kinding result is formally stated as:

Lemma 2 (Kinding for **bang**$(\cdot)$)**.** *For any type τ, if $\Delta \vdash \tau :_\mathrm{K} \kappa$ then $\Delta \vdash \boldsymbol{bang}(\tau) :_\mathrm{K} \boldsymbol{bang}(\kappa)$.*

To integrate this type operator with parametric polymorphism, we model our solution on Odersky's Observer types [Odersky 1992], and tag type variables that have been made read only, using the syntax $\alpha!$. Whenever a variable α is instantiated to some concrete type τ, we also replace $\alpha!$ with **bang**(τ). The lemma above ensures that our kinding rule for such tagged variables is sound, and enables us to prove the following:

Lemma 3 (Type instantiation preserves kinds)**.** *For any type τ, $\overline{\alpha_i :_\mathrm{K} \kappa_i} \vdash \tau :_\mathrm{K} \kappa$ implies $\Delta \vdash \tau[\overline{\rho_i}/\overline{\alpha_i}] :_\mathrm{K} \kappa$ when, for each i, $\Delta \vdash \rho_i :_\mathrm{K} \kappa_i$.*

primops	o	$\in$	$\{\mathtt{+}, \mathtt{*}, \mathtt{/}, \mathtt{<=}, \mathtt{==}, \mathtt{||}, \mathtt{<<}, \ldots\}$
literals	ℓ	$\in$	$\{123, \mathtt{True}, \mathtt{'a'}, \ldots\}$
expressions	e	$::=$	$x \mid () \mid f[\overline{\tau}] \mid o(\overline{e}) \mid e_1\ e_2$
		$\mid$	$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$
		$\mid$	$\mathbf{let!}(\overline{y})\ x = e_1\ \mathbf{in}\ e_2$
		$\mid$	$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$
		$\mid$	$\ell \mid \mathbf{cast}\ t\ e \mid \mathbf{promote}\ \langle \overline{\mathsf{C}\ \tau} \rangle\ e$
		$\mid$	$\mathbf{case}\ e_1\ \mathbf{of}\ \mathsf{C}\ x \to e_2\ \mathbf{else}\ y \to e_3$
		$\mid$	$\mathbf{esac}\ e \mid \mathsf{C}\ e$
		$\mid$	$\{\overline{\mathsf{f}\ = e}\} \mid e.\mathsf{f} \mid \mathbf{put}\ e_1.\mathsf{f} := e_2$
		$\mid$	$\mathbf{take}\ x\ \{\mathsf{f}\ = y\} = e_1\ \mathbf{in}\ e_2$
function def.	d	$::=$	$\langle f :: \pi, f\ x = e \rangle \mid \langle f :: \pi, \blacksquare \rangle$
programs	P	$::=$	$\overline{d}$
function names		$\ni$	f, g
variables		$\ni$	x, y
constructors		$\ni$	$\mathsf{A}, \mathsf{B}, \mathsf{C}$
record fields		$\ni$	f, g

primopType$(\cdot)$	:	$o \to \bar{t} \times t$	(primop types)
funDef$(\cdot)$	:	$f \to d$	(definition environment)
$\lvert\cdot\rvert$	:	$t \to \mathbb{N}$	(maximum value)

Figure 5: Syntax of COGENT programs (after desugaring)

3.2 Expressions and Typing

While COGENT features a rich surface syntax, due to space constraints, we only document the core language in Fig. 5 to which the surface syntax is desugared.

Fig. 6 shows the typing rules for COGENT expressions. Many of these are standard for any linear type system. We will discuss here the rules for **let!**, where we have taken a slightly different approach to established literature, and the rules for the extensions we have made to the type system, such as variants and record types.

3.2.1 Typing for let!

On the expression level, the programmer can use **let!** expressions, in the style of Wadler [1990], to temporarily convert variables of linear types to their read-only equivalents, allowing them to be freely shared. In this example, we wish to copy a buffer b_2 onto a buffer b_1 only when b_2 will fit inside b_1.

$$\mathbf{let!}(b_1, b_2)\ ok = (\mathsf{size}(b_2) < \mathsf{size}(b_1))\ \mathbf{in}$$
$$\mathbf{if}\ ok\ \mathbf{then}\ \mathsf{copy}(b_1, b_2)\ \mathbf{else}\ \ldots$$

Note that even though b_1 and b_2 are used multiple times, they are only used once in a linear context. Inside the **let!** binding, they have been made temporarily nonlinear. Our kind system ensures these read-only, shareable references inside **let!** bindings cannot "escape" into the outside context. For example, the expression $\mathbf{let!}(b)\ b' = b\ \mathbf{in}\ \mathsf{copy}(b, b')$ would violate the invariants of the linear type system, and ruin the purely functional abstraction that linear types allow, as both b and b' would refer to the same object, and a destructive update to b would change the shareable b'.

We are able to use the existing kind system to handle these safety checks with the inclusion of the E permission, for Escapable, which indicates that the type may be safely returned from within a **let!**. We ensure, via the typing rules of Fig. 6, that the left hand side of the binding (*ok* in the example) has the E permission, which excludes temporarily nonlinear references via **bang**$(\cdot)$ (see Fig. 4).

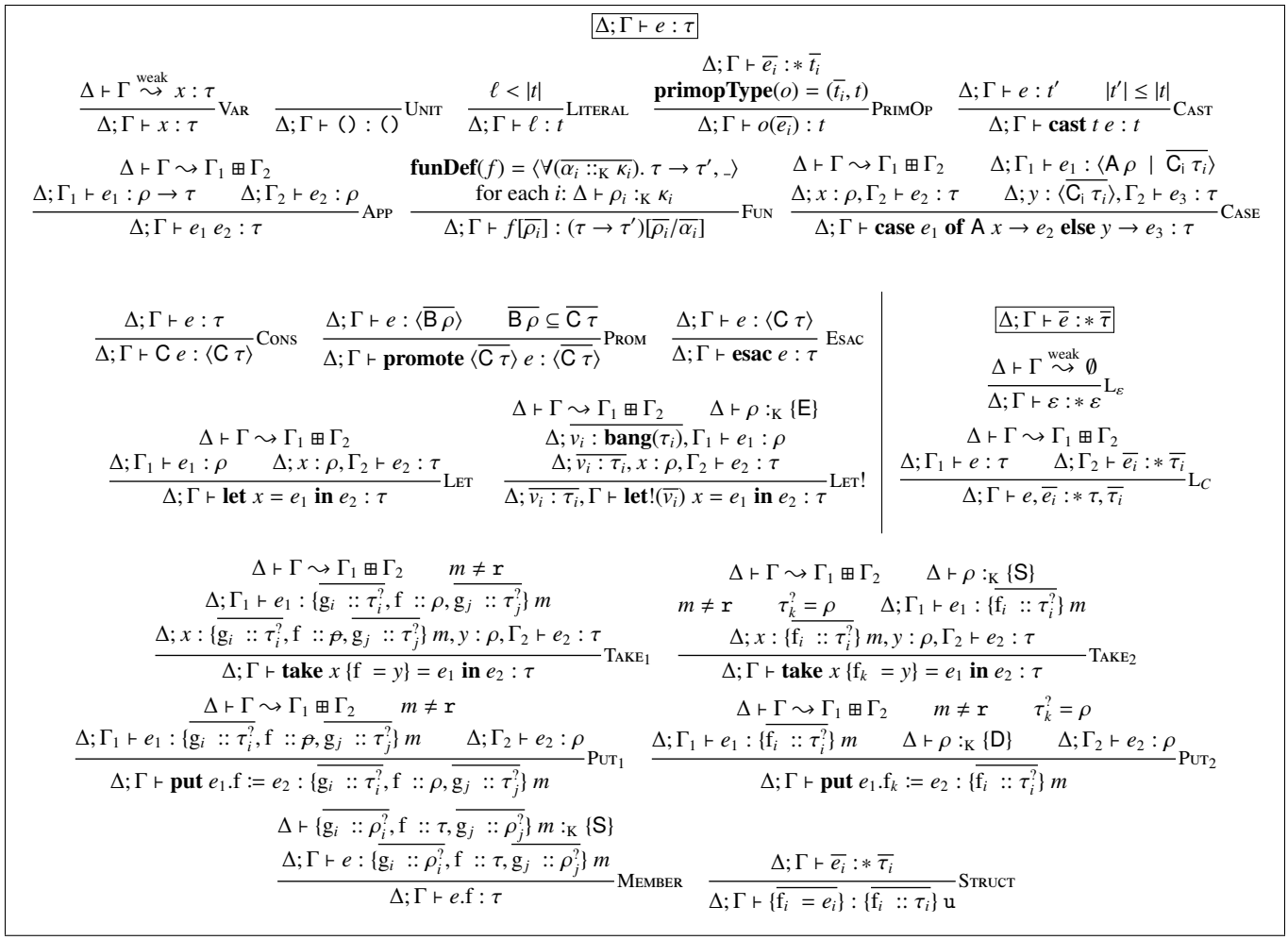

Figure 6: Typing rules for COGENT

3.2.2 Typing for Variants

A variant type $\langle \overline{\mathsf{C_i}\ \tau_i} \rangle$ is a generalised sum type, where each alternative is distinguished by a unique *data constructor* $\mathsf{C_i}$. The order in which the constructors appear in the type is not important. One can create a variant type with a single alternative simply by invoking a constructor, e.g. Some 255 might be given the type $\langle \mathsf{Some}\ \mathtt{U8} \rangle$. The original value of 255 can be retrieved using the **esac** construct. The set of alternatives is enlarged by using **promote** expressions that are automatically inserted by the type-checker of the surface language, which uses subtyping to infer the type of a given variant. A similar trick is used for numeric literals and **cast**.

In order to pattern match on a variant, we provide a **case** construct that attempts to match against one constructor. If the constructor does not match, it is *removed* from the type and the reduced type is provided to the **else** branch. In this way, a traditional multi-way pattern match can be desugared by nesting:

case x **of**		**case** x **of**
$\quad \mathsf{A}\ a \to e_a$		$\quad \mathsf{A}\ a \to e_a$
$\quad \mathsf{B}\ b \to e_b$	becomes	$\quad$ **else** $x' \to$ **case** x' **of**
$\quad \mathsf{C}\ c \to e_c$		$\quad\quad \mathsf{B}\ b \to e_b$
		$\quad\quad$ **else** $x'' \to$ **let** $c =$ **esac** x'' **in** e_c

Note that because the typing rule for **esac** only applies when only one alternative remains, our pattern matching is necessarily total.

3.2.3 Typing for Records

Some care is needed to reconcile record types and linear types. Assume that `Object` is a type synonym for an (unboxed) record type containing an integer and two (linear) buffers.

$$\mathtt{Object} = \{\mathrm{size} :: \mathtt{U32}, \mathrm{b}_1 :: \mathtt{Buf}, \mathrm{b}_2 :: \mathtt{Buf}\}\ \mathtt{u}$$

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. But, we can't return the entire `Object` along with the `Buf`, as this would introduce aliasing. Our solution is to return along with the `Buf` an `Object` where the field b_1 cannot be extracted again, and reflect this in the field's type, written as $\mathrm{b}_1 :: \not\mathtt{Buf}$. This field extractor, whose general form is **take** $x\ \{\mathrm{f} = y\} = e_1$ **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. Unless the type of the field f has kind $\{\mathsf{S}\}$, that field will 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 has kind $\{\mathsf{D}\}$, that field must already be taken, to avoid accidentally destroying our only reference to a linear resource.

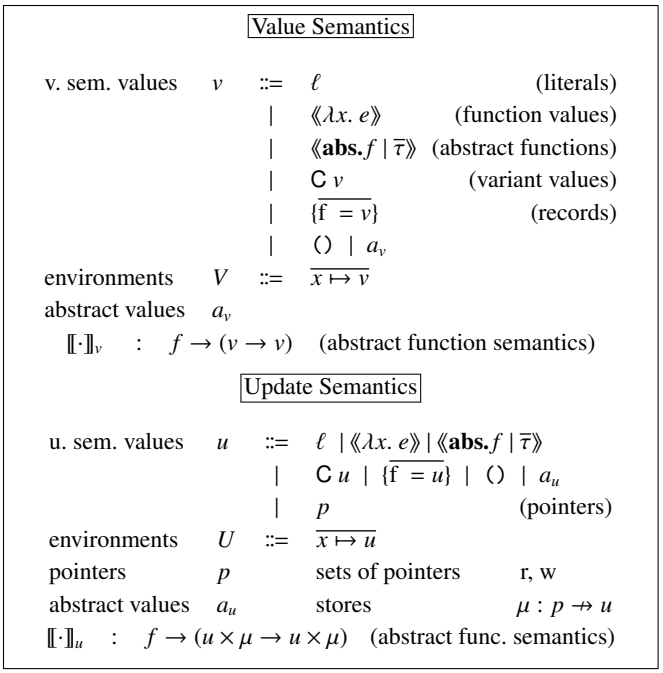

Figure 7: Definitions for Value and Update Semantics

Unboxed records can be created using a simple struct literal $\{\overline{\mathrm{f}_i = e_i}\}$. We also allow records to be stored on the heap to minimise unnecessary copying, as unboxed records are passed by value. These 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. Thus a function for allocating `Object`-like records might return values of type: $\{\mathrm{size} :: \sout{\texttt{U32}}, \mathrm{b}_1 :: \sout{\texttt{Buf}}, \mathrm{b}_2 :: \sout{\texttt{Buf}}\}$ `w`.

For any nonlinear record (that is, (1) read-only boxed records, which cannot have linear fields, as well as (2) unboxed records without linear fields) we also allow traditional member syntax $e.\mathrm{f}$ for field access. The typing rules for all of these expressions are given in Fig. 6.

3.2.4 Type Specialisation

As mentioned earlier, we implement parametric polymorphism by specialising code to avoid paying the performance penalties of other approaches such as boxing . This means that polymorphism in our language is restricted to predicative rank-1 quantifiers.

This allows us to specify dynamic objects, such as our value typing relations (see §3.3.1) and our dynamic semantics (see §3.3), in terms of simple monomorphic types, without type variables. Thus, in order to evaluate a polymorphic program, each type variable must first be instantiated to a monomorphic type. We show that typing of the instantiated program follows from the typing of the polymorphic program, if the type instantiation used matches the kinds of the type variables.

Lemma 4 (Type specialisation)**.** *$\overline{\alpha_i :_{\mathrm{K}} \kappa_i}; \Gamma \vdash e : \tau$ implies $\Delta; \Gamma[\overline{\rho_i}/\overline{\alpha_i}] \vdash e[\overline{\rho_i}/\overline{\alpha_i}] : \tau[\overline{\rho_i}/\overline{\alpha_i}]$ when, for each i, $\Delta \vdash \rho_i :_{\mathrm{K}} \kappa_i$.*

The above lemma is sufficient to show the monomorphic instantiation case, by setting $\Delta = \varepsilon$ (the empty context). This lemma is a key ingredient for the refinement link between polymorphic and monomorphic deep embeddings (See §4.4).

3.3 Dynamic Semantics

Fig. 8 includes most of the evaluation rules for the *value* semantics of Cogent, as a big-step relation $V \vdash e \Downarrow_v v$ from expressions to their values. These values are documented in Fig. 7. This semantics is typical of a purely functional language, by design, and we automatically produce a purely functional shallow embedding from it.

As functions must be defined on the top level, our function values $\langle\!\langle \lambda x.\, e \rangle\!\rangle$ consist only of an unevaluated expression, evaluated when the function is applied. Abstract function values, written $\langle\!\langle \mathbf{abs}.f \mid \overline{\tau} \rangle\!\rangle$, are instead passed indirectly, as a pair of the function name and a list of the types used to instantiate type variables. When an abstract function value $\langle\!\langle \mathbf{abs}.f \mid \overline{\tau} \rangle\!\rangle$ is applied, the user-supplied semantics $[\![f]\!]_v$ is invoked, which is just a function from input to output value.

The *update* semantics, by contrast, is rather imperative. An excerpt is presented in Fig. 8, with associated definitions in Fig. 7. This semantics is an evaluation semantics, written $U \vdash e \mid \mu \Downarrow_u u \mid \mu'$. Values in the update semantics may now be *pointers*, written p, to values in a mutable store or *heap* μ. This mutable store is modeled as a partial function from a pointer to an update semantics value.

Most rules in the update semantics only differ from the value semantics in that they thread the heap μ through the program evaluation, and so many of those rules have been omitted. However, key differences arise in the treatment of records and of abstract types, which may now be represented as *boxed* structures, stored on the heap. In particular, note that the rule $\textsc{Put}_2$ destructively updates the heap, instead of creating a new record value, and the semantics of abstract functions $[\![\cdot]\!]_u$ may also modify the heap.

3.3.1 Update-Value Refinement and Type Preservation

To show the update semantics is a refinement of the value semantics, we must exploit the information given by Cogent's linear type system. A typical refinement approach to relate the two semantics is to define a correspondence relation between update semantics states and value semantics values, and show that an update semantics evaluation implies a corresponding value semantics evaluation. However, such a statement is not true if aliasing exists, as a destructive update (from, say, **put**) would result in multiple values being changed in the update semantics but not necessarily in the value semantics. As our type system forbids aliasing of writable references, we must include this information in our correspondence relation. Written as $u \mid \mu : v : \tau\ [r * w]$, this relation states that the update semantics value u with heap μ corresponds to the value semantics value v, which both have the type τ. Unlike prior work [Hofmann 2000], we account for the heap by annotating this relation with the sets r and w, which contain all pointers accessible (transitively) from the value u that are read-only and writable respectively. We call these pointer sets the *heap footprint* of the value. By annotating the correspondence relation with the heap footprint, we can encode the uniqueness properties ensured by linear types as explicit non-aliasing constraints in the rules, given in Fig. 9. Read-only pointers may alias other read-only pointers, but writable pointers do not alias any other pointer, whether read-only or writable.

Because our correspondence relation includes types, it naturally implies a value typing relation for both value semantics (written $v : \tau$) and update semantics (written $u \mid \mu : \tau\ [r * w]$). In fact, the rules for both relations can be derived from the rules in Fig. 9 simply by erasing either the value semantics parts (highlighted like this) or the update semantics parts (highlighted like this). As we ultimately prove preservation for this correspondence relation across evaluation, this same erasure strategy can be applied to our proofs to produce a type preservation proof for either semantics.

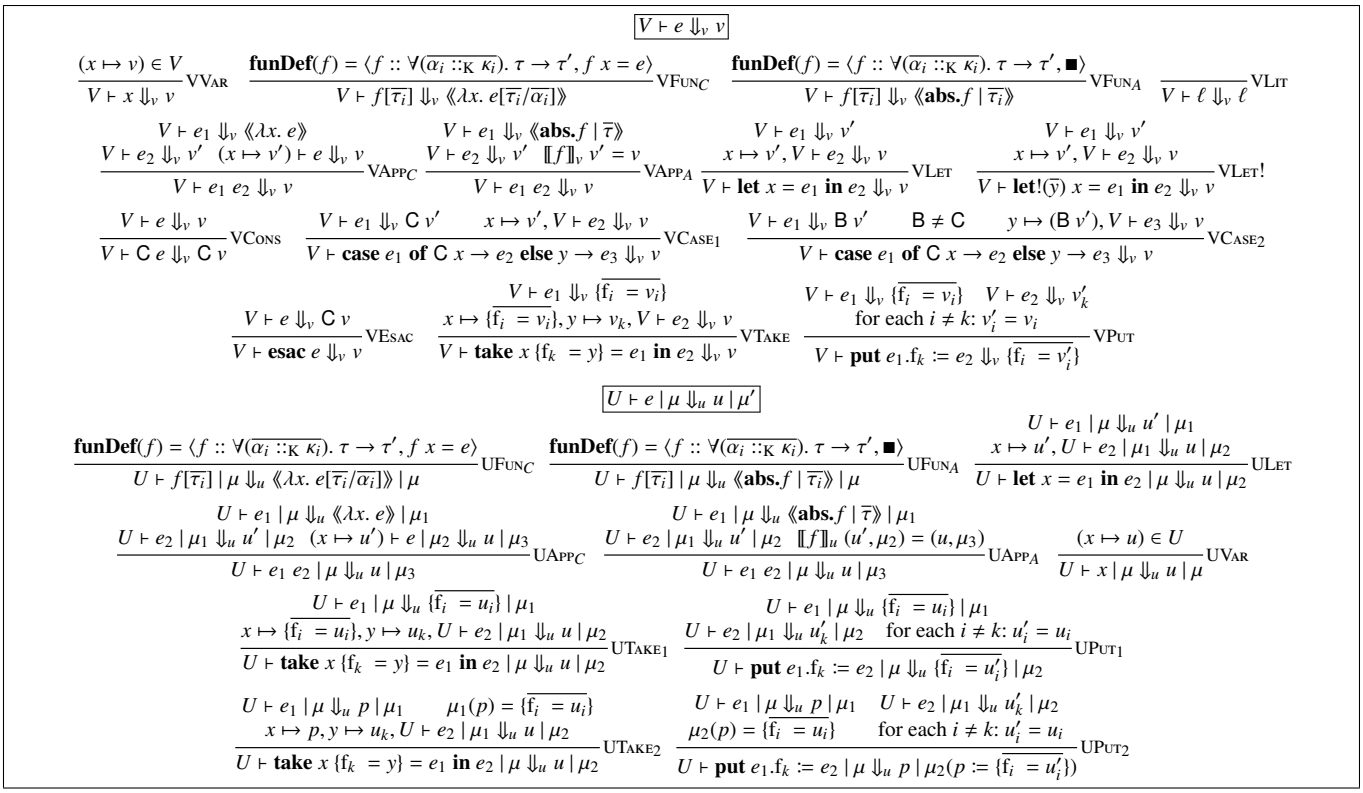

Figure 8: Some of the important rules for the two dynamic semantics of COGENT

Dealing with mutable state We define a *framing* relation which specifies exactly how evaluation may affect the mutable heap μ. Given an input set of writable pointers w_i, an input heap μ_i, an output set of pointers w_o and an output heap μ_o, the relation, written $w_i \mid \mu_i$ **frame** $w_o \mid \mu_o$, ensures three properties for any pointer p:

Inertia If $p \notin w_i \cup w_o$, then $\mu_i(p) = \mu_o(p)$.

Leak freedom If $p \in w_i$ and $p \notin w_o$, then $\mu_o(p) = \bot$.

Fresh allocation If $p \notin w_i$ and $p \in w_o$, then $\mu_i(p) = \bot$.

Framing implies that our correspondence relation, for both values and environments, is unaffected by updates to parts of the heap not mentioned in the heap footprint:

Lemma 5 (Unrelated updates)**.** *Assume two unrelated pointer sets* $w \cap w_1 = \emptyset$ *and that* $w_1 \mid \mu$ ***frame*** $w_2 \mid \mu'$*, then*

- *If* $u \mid \mu : v : \tau\ [r * w]$ *then* $u \mid \mu' : v : \tau\ [r * w]$ *and* $w \cap w_2 = \emptyset$.
- *If* $U \mid \mu : V : \Gamma\ [r * w]$ *then* $U \mid \mu' : V : \Gamma\ [r * w]$ *and* $w \cap w_2 = \emptyset$.

We are now able to prove refinement between the value and the update semantics. We first prove that the correspondence relation is preserved when both semantics evaluate from corresponding environments. By erasing one semantics, this becomes a type preservation theorem for the other. Full details of the proof are available in our Isabelle/HOL formalisation.

Theorem 1 (Preservation of types and correspondence)**.** *For any program e where* $\Gamma; e \vdash \tau :$*, if* $U \mid \mu : V : \Gamma\ [r * w]$ *and* $V \vdash e \Downarrow_v v$ *and* $U \vdash e \mid \mu \Downarrow_u u \mid \mu'$*, then there exists* $r' \subseteq r$ *and* w' *such that* $u \mid \mu' : v : \tau\ [r' * w']$ *and* $w \mid \mu$ ***frame*** $w' \mid \mu'$.

To prove refinement, we must show that every evaluation on the concrete update semantics has a corresponding evaluation in the abstract value semantics. While Theorem 1 gets us most of the way there, we still need to prove that the value semantics can evaluate whenever the update semantics does.

Theorem 2 (Upward-propagation of evaluation)**.** *For any program e where* $\Gamma; e \vdash \tau :$*, if* $U \mid \mu : V : \Gamma\ [r * w]$ *and* $U \vdash e \mid \mu \Downarrow_u u \mid \mu'$*, then there exists a v such that* $V \vdash e \Downarrow_v v$

Composing this lemma and Theorem 1, we can now easily prove our desired refinement statement.

Theorem 3 (Value ⇒ Update refinement)**.** *For any program e where* $\Gamma; e \vdash \tau :$*, if* $U \mid \mu : V : \Gamma\ [r * w]$ *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* $u \mid \mu' : v : \tau\ [r' * w']$ *and* $w \mid \mu$ ***frame*** $w' \mid \mu'$.

3.3.2 FFI requirements

Each of the above theorems makes certain assumptions about the the semantics given to abstract functions, $[\![\cdot]\!]_v$ and $[\![\cdot]\!]_u$. Specifically, we assume the preservation theorem above for these functions: for any abstract function f of type $\tau \to \rho$, if the argument values u and v correspond and are well typed i.e. $u \mid \mu : v : \tau\ [r * w]$ and $[\![f]\!]_u(u, \mu) = (u', \mu')$ and $[\![f]\!]_v\ v = v'$, then then $u' \mid \mu' : v' : \rho\ [r' * w']$ for some $r' \subseteq r$ and w' such that $w \mid \mu$ **frame** $w' \mid \mu'$. The well-typedness requirement ensures that the returned value does not include internal aliasing, and the **frame** requirements ensure that the abstract function does not manipulate any state other than values passed into it. These requirements are necessary to integrate C code with COGENT's linear type system. In order to show refinement, we must also assume that abstract functions evaluate in the value semantics whenever they evaluate in the update semantics, as in Theorem 2.

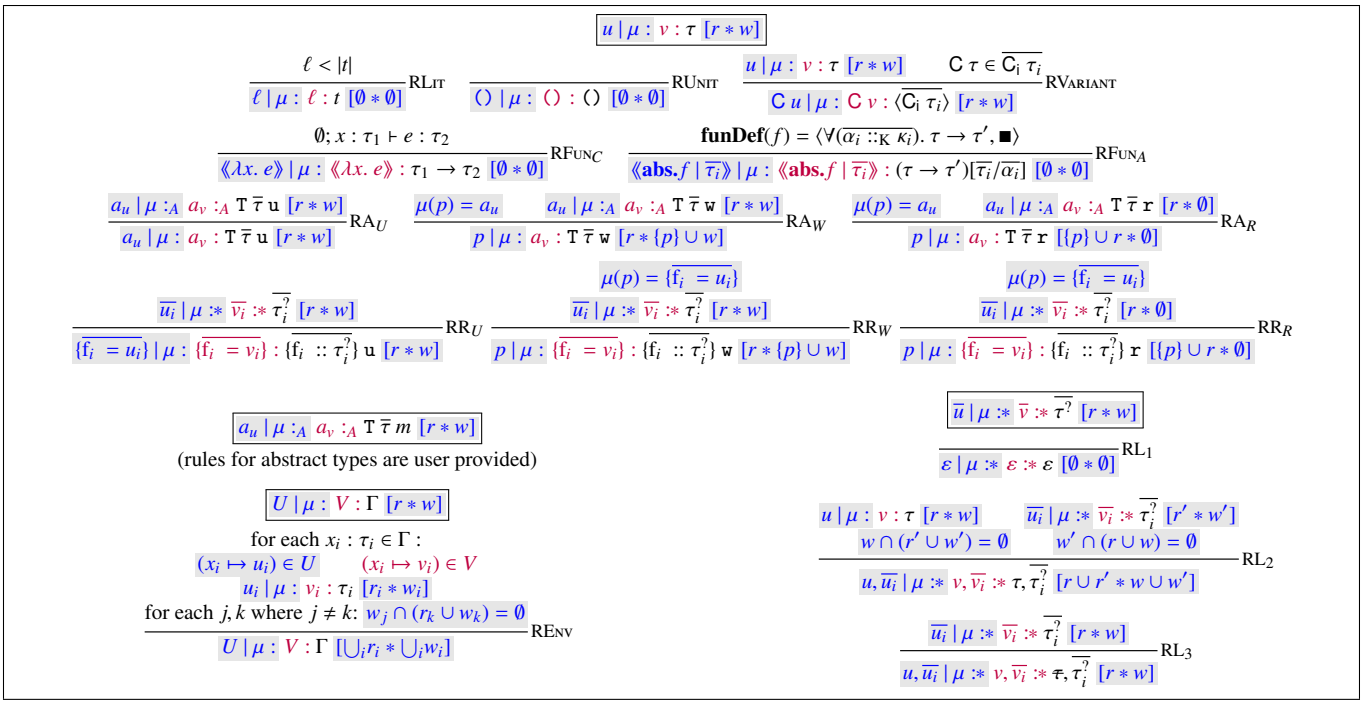

Figure 9: Value Typing and Refinement (For value typing rules, erase this text for value semantics, and this text for update semantics.)

4. Compiler Certificate

This section describes each of the proof steps that make up the compiler certificate, depicted in Fig. 1 in §2.

4.1 Top-Level Theorem

The top-level theorem is the overall certificate emitted by the compiler. We say a C program correctly implements its COGENT shallow embedding if: (i) it terminates with defined execution; and (ii) assuming that the initial C and COGENT stores are related and the program inputs are related, then their outputs are related.

In other words, the compiler theorem states that a *value relation* is preserved across evaluation. In §3.3.1, we present a value relation between update and value semantics. At every other refinement stage, we introduce a similar relation between values of the two respective programs. By composing these relations, we get the value relation $\mathcal{V}$ between a result v_m of a C program p_m and a shallow embedding s, going through intermediateupdate and value semantics results. The relation in §3.3.1 depends on a COGENT store μ, which is related to the C state using the *state relation* $\mathcal{R}$, defined in §4.3.

Let $\lambda e.\ \mathcal{M}_e\ r\ e$ and $\lambda v.\ \mathcal{M}_v\ r\ v$ (defined in §4.4) be two functions that monomorphise expressions and (function) values respectively, using a rename function r provided by the compiler. Further, let s be a shallow embedding, e a monomorphic deep embedding, p_m a C program, μ a COGENT store and σ a C state. We define the overall refinement predicate **correspondence** as follows:
If $(\exists r\ w.\ U \mid \mu : V :\ \Gamma\ [r * w])$ and $(\mu, \sigma) \in \mathcal{R}$, then p_m successfully terminates starting at σ; and for any resulting value v_m and state σ' of p_m, there exist μ', u, and v such that:

$$(\mu', \sigma') \in \mathcal{R} \ \wedge\ U \vdash e \mid \mu \Downarrow_u u \mid \mu' \ \wedge\ V \vdash e \Downarrow_v \mathcal{M}_v\ r\ v \wedge \mathcal{V}\ r\ \mu'\ v_m\ u\ v\ s$$

Intuitively, our top-level theorem states that for related input values, all programs in the refinement chain evaluate to related output values. This can of course be used to deduce that there exist intermediate programs through which the C code and its shallow embedding are directly related. The user does not need to care what those intermediate programs are.

Theorem 4. *Given a* COGENT *function f that takes x of type τ as input, let p_m be its generated C code, s its shallow embedding, and e its deep embedding. Let v_m be an argument of p_m, and u and v be the update and value semantics arguments, of appropriate type, for f.*

$$\forall \mu\ \sigma.\ \mathcal{V}\ r\ \mu\ v_m\ u\ v\ s \longrightarrow$$
$$\quad \mathbf{correspondence}\ r\ \mathcal{R}\ (s\ v_s)\ (\mathcal{M}_e\ r\ e)\ (p_m v_m)\ U\ V\ \Gamma\ \mu\ \sigma$$

where $U = (x \mapsto u)$, $V = (x \mapsto\ v)$, and $\Gamma = (x \mapsto \tau)$.

This refinement theorem additionally assumes abstract functions in the program adhere to their specification and their behaviour stays unchanged when monomorphised.

4.2 Well-typedness

We first mention the well-typing theorems used in refinement. The COGENT compiler proves, via an automated Isabelle tactic, that the monomorphic deep embedding of the input program is well-typed. Specifically, the compiler defines **funDef**$(\cdot)$ in Isabelle and proves that each COGENT function $f\ x\ =\ e$ matches the signature given by **funDef**(f). The typing proof for the polymorphic embedding is described in §4.4.

Theorem 5 (Typing)**.** *Let f be a (monomorphic)* COGENT *function, where* ***funDef****$(f) = \langle f :: \tau \to \tau', f\ x = e \rangle$. Then $\varepsilon; x : \tau \vdash e : \tau'$.*

As we will see in §4.3, proving refinement requires access to the typing judgements for program sub-expressions and not just for the top level, so 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 type derivation tree for the COGENT program. Each node is a typing theorem for a program sub-expression.

4.3 C to COGENT Monomorphic Deep Embedding

This section outlines the first three transformations in Fig. 1. Details of this proof automation can be found in Anonymous [2016b]. First, the C code is converted to a Simpl embedding by the C parserused in the seL4 project [Klein et al. 2009]. This step is simple makes no effort to abstract from low-level C semantics.

The second step applies a modified version of AutoCorres to produce a *monadic* shallow embedding of the C code, and to prove that the Simpl embedding is a refinement of the monadic shallow embedding. We modify AutoCorres to make its output more predictable by switching off its control-flow simplification and forcing it to output the shallow embedding in the *nondeterministic state monad* of Cock et al. [2008], where computation is represented by functions of type $state \Rightarrow (\alpha \times state)\ set \times bool$. Here *state* is the global state of the C 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. The computation also returns a boolean, *failed*, indicating the presence of undefined behaviour.

While AutoCorres was designed to ease manual reasoning about C, we use it as the foundation for automatically proving correspondence to the COGENT input program. One of the main benefits AutoCorres gives us is a *typed* memory model. Specifically, the *state* of the AutoCorres monadic representation contains a set of *typed heaps*, each of type $32\ \mathsf{word} \Rightarrow \alpha$, one for each type α used on the heap by the C input program.

If the monadic code never fails, then the C code is type- and memory-safe, and free of undefined behavior [Greenaway et al. 2014]. We prove non-failure as a side-condition of refinement, basically using COGENT's type system to guarantee C memory safety.

The third step in Fig. 1 is a refinement proof between deeply-embedded COGENT and a shallow, monadic embedding of C. To phrase the refinement statement we first define how deeply embedded COGENT values and types relate to their corresponding values in the monadic embedding. This value-mapping is captured by a value relation $val\text{-}rel_C$, generated in Isabelle automatically by the COGENT compiler, using *ad hoc* overloading. We must generate $val\text{-}rel_C$ separately for each COGENT program because the types used in the shallow embedding depend on those used in the deep. For example, e.g. C structs are represented directly as Isabelle records.

Given all the relation definitions for a particular COGENT program, the *state relation* $\mathcal{R}$ defines the correspondence between the store μ which the COGENT update semantics manipulates, and the state σ for the monadic shallow embedding.

Definition 1 (Monad-to-Update 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 corresponding heap of σ at location p, such that $val\text{-}rel_C\ \mu(p)\ v$ holds.*

With $\mathcal{R}$ and $val\text{-}rel_C$, 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**. Because $\mathcal{R}$ changes for each COGENT program, we parameterise **corres** by an arbitrary state relation R. It is parameterised also 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 2 (Monad-to-Update Correspondence)**.**

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

The definition states 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 R and $val\text{-}rel_C$ holds between their results. AutoCorres proves automatically that: $\neg\ failed\ (p_m\ \sigma) \longrightarrow results\ (p_m\ \sigma) \neq \emptyset$.

$$\frac{(x \mapsto u) \in U \qquad val\text{-}rel_C\ u\ v_m}{\textbf{corres}\ R\ x\ (\textbf{return}\ v_m)\ U\ \Gamma\ \mu\ \sigma}\ \textsc{Corres-Var}$$

$$\frac{\begin{array}{c}\vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad \varepsilon; \Gamma_1 \vdash a : \tau \quad \textbf{corres}\ R\ a\ a'\ U\ \Gamma_1\ \mu\ \sigma \\ (\forall v_u\ v_m\ \mu'\ \sigma'.\ val\text{-}rel_C\ v_u v_m \longrightarrow \\ \textbf{corres}\ R\ b\ (b'\ v_m)\ (x \mapsto v_u, U)\ (x : \tau, \Gamma_2)\ \mu'\sigma')\end{array}}{\textbf{corres}\ R\ (\textbf{let}\ x = a\ \textbf{in}\ b)\ (a'\ >>=\ b')\ U\ \Gamma\ \mu\ \sigma}\ \textsc{Corres-Let}$$

Figure 10: Two example **corres** rules

The refinement proof is automatic in Isabelle, driven by a set of syntax-directed **corres** rules, one for each COGENT construct. The proof procedure uses the fact that the COGENT terms are in A-normal form to reduce the number of cases that need considering and to simplify the higher-order unification problems that some of the proof rules pose to Isabelle.

Fig. 10 depicts two **corres** rules, one for expressions x that are variables and the other for **let** $x = a$ **in** b. These correspond respectively to the basic monadic operations **return**, which yields values, and >>=, for sequencing computations.

Observe that LET is *compositional*: Proving that **let** $x = a$ **in** b corresponds to $a'\ >>=\ b'$ requires proving correspondence for a to a' and b to b'. This compositionality significantly simplifies the automation of the refinement proof. The typing assumptions of LET are discharged by appealing to the compiler generated type theorem tree (see §4.2).

The rules for some of the other constructs, such as **take**, **put**, and **case**, contain non-trivial assumptions about $\mathcal{R}$ and about the types used in the program. Once a program and its $\mathcal{R}$ are fixed, a set of simpler rules is automatically generated by *specialising* the generic **corres** rules for each of these constructs to the particular $\mathcal{R}$ and types used in the input program. This in effect discharges the non-trivial assumptions of these rules once-and-for-all, allowing the automated proof of correspondence to proceed efficiently. This specialisation technique is documented in Anonymous [2016b].

Conceptually, the refinement proof proceeds bottom-up, starting with the leaf functions of the program and ending with the top-level entry points; **corres** results proved earlier are used to discharge **corres** assumptions for callees. The **corres** proof tactic thus follows the call-graph of the input program. Currently, the tactic is limited to computing call graphs correctly only for programs containing up to second-order functions. We have not needed any higher orders in our applications, but the tactic can be easily extended if needed.

The final refinement theorem for this stage assumes that **corres** holds for all the abstract functions used in the program.

Theorem 6. *Let f be a (monomorphic)* COGENT *function, such that* ***funDef****$(f) = \langle f :: \tau \to \tau', f\ x = e \rangle$. Let p_m be its monadic shallow embedding, derived from its generated C code. Let u and v_m be arguments of appropriate type for f and p_m respectively. Then:*

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

4.4 Monomorphic to Polymorphic Deep Embedding

To complete the refinement step from the update to the value semantics, the compiler just applies Theorem 3. We establish the correctness of the compiler's monomorphisation pass, moving upwards in Fig. 1 from a monomorphic to a polymorphic deep embedding in the value semantics.

The compiler generates a renaming function r that, for a polymorphic function name f_p and types $\overline{\tau}$, yields the specialised

monomorphic function name f_m. Just as we assume abstract functions are correctly implemented in C, we also assume that their behaviour remains consistent under r.

We write an Isabelle function to simulate the compiler monomorphisation, and prove that (1) the monomorphised program it produced is identical to that produced by the compiler, and (2) the monomorphised program is a correct refinement of the polymorphic one. We define two Isabelle functions parameterised by r: $\mathcal{M}_e$ for monomorphising expressions and $\mathcal{M}_v$ for monomorphising (function) values.

Step (1) is proved by straightforward rewriting, and is automated on a per-program basis. Step (2) is embodied in the following refinement theorem, which we prove, once and for all, by rule induction over the value semantics. The specialisation Lemma 4 of §3.2.4, is a key ingredient of this proof.

Theorem 7 (Monomorphisation)**.** *Let f be a (polymorphic)* Cogent *function whose definition given by* ***funDef***$(\cdot)$ *is $f\ x = e$. Let v be an appropriately-typed argument for f. Let r be a renaming function. Then:*

$$\forall v'.\ (x \mapsto \mathcal{M}_v\ r\ v) \vdash \mathcal{M}_e\ r\ e \Downarrow_v \mathcal{M}_v\ r\ v' \longrightarrow (x \mapsto v) \vdash e \Downarrow_v v'$$

As the compiler generates a well-typedness proof for a monomorphic deeply embedded program (§4.2), Theorem 7 implies well-typedness of the polymorphic deep embedding.

4.5 Deep to Shallow Embedding

In this section, the proof once again connects deep and shallow embeddings, where the shallow embedding is, this time, a pure function in Isabelle/HOL. This shallow embedding is still in A-normal form and is produced by the compiler: For each Cogent type, the compiler generates a corresponding Isabelle/HOL type definition, and for each Cogent function, a corresponding Isabelle/HOL constant definition. We drop the linear types and remain in Isabelle's simple types, because we have already used the linear types to justify our switch to the value semantics.

In addition to these definitions, the compiler produces a theorem that the deeply embedded polymorphic Cogent term under the value semantics correctly refines this Isabelle/HOL function. Refinement is formally defined here by the predicate **scorres** that defines when a shallowly embedded expression s is refined by a deeply embedded one e when evaluated under the environment V.

Definition 3 (Deep to Shallow Correspondence)**.**

$$\textbf{scorres}\ s\ e\ V\ \equiv\ \forall r.\ V \vdash e \Downarrow_v r \longrightarrow \mathit{val\text{-}rel}_S\ s\ r$$

That is, s corresponds to e under V if whenever e evaluates to an r under V, then s and v are in the value relation $\mathit{val\text{-}rel}_S$. Similarly to the proof from monadic C to update semantics, the value relation here is one polymorphic constant in Isabelle/HOL, defined incrementally via *ad hoc* overloading. The program-specific refinement theorem produced is:

Theorem 8 (Deep to Shallow Refinement)**.** *Let f be an A-normal* Cogent *function such that* ***funDef***$(f) = \langle f :: \pi, f\ x = e \rangle$*, and let s be f's shallow embedding. Then*
$\forall v_s\ v.\ \mathit{val\text{-}rel}_S\ v_s\ v \longrightarrow \textbf{scorres}\ (s\ v_s)\ e\ (x \mapsto v)$

$\mathit{val\text{-}rel}_S\ v_s\ v$ ensures that v_s and v are of matching type. Like the C refinement proof in §4.3, we have fully automated this proof using a specifically designed syntax-directed rule set.

4.6 Shallow Embedding to Neat Shallow Embedding

Fig. 11 depicts the top-level shallow embedding, only mildly polished for presentation, for the Cogent example of Fig. 2.
As Fig. 11 shows, the Isabelle definitions use the same names as the Cogent input program and keep the same structure, making it easy for the user to reason about.

```
1 ext2_free_branch (depth, nd (ex, fs, inode), mdep) ≡
2 if depth < mdep then
3 case uarray_create (ex, (to_f nd - fr_f nd)) of
4   R_11.Success (ex, children) ⇒
5     let (mbuf, nd_t) = take_G nd mbuf_f;
6         (children, ds_16) = take_G
7           (uarray_map_no_break
8             (ArrayMapP.mk children ext2_free_branch_entry
9               (ex, inode, (fr_f nd_t), mbuf) ...));  ...
```

Figure 11: Shallow embedding for the example from §2

The correctness statement for this phase is simple: it is pure Isabelle/HOL equality between the A-normal and neat shallow embedding for each function. For instance:

Shallow.ext2_free_branch = Neat.ext2_free_branch

The proof is simple as well. Since we can now use equational reasoning with Isabelle's powerful rewriter, we just unfold both sides, apply extensionality and the proof is automatic given the right congruence rules and equality theorems for functions lower in the call graph.

5. Discussion

Our aim is to significantly reduce the cost of verifying real-world systems software with minimal performance impact. In this section, we evaluate the performance and usability of Cogent in two realistic file system implementations, as well as discuss ways in which Cogent can be improved in the future.

Cogent as a Systems Programming Language Running IOZone [IOZone] microbenchmarks, Amani et al. [2016] show that the `ext2` implementation in Cogent exhibits a modest improvement compared to the C version in random- and sequential-writes throughput, with similar CPU usage at around 10%. For their own file system BilbyFs, the Cogent version performs slightly worse than the C version, with 10–20% less throughput and CPU usage of 20% compared to 10%. In the absence of any I/O disturbance, the Cogent implementation of `ext2` is also slightly slower than native `ext2fs` when performing random writes to a RAM disk. These slight performance overheads are due to the fact that the Cogent implementations tend to use data structures that strongly resemble those of the original C, rather than more idiomatic functional data structures, which are handled better by our compiler. For some operations, the C implementation would rely on unsafe features for performance, where the Cogent implementation uses slower, but easier to verify techniques. One way to improve this on a language level would be to include specialised constructs for these operations in the language, allowing for the generation of efficient, yet automatically verified code. Including support for such constructs is at the top of the priority list for future work on Cogent. These case studies are invaluable for identifying common patterns and guide this important next step.

Occasionally, some overheads are introduced in C code generation, as we rely on the C compiler for low-level optimisation. Generated code displays patterns which are uncommon in handwritten code and therefore might not be picked up by the C optimiser, even if they are trivial to optimise. For example, the generated code is already quite close to SSA form used by `gcc` internally for optimisation, however `gcc` does not recognize this and optimise accordingly. Generating a compiler's SSA representation directly, such as that of LLVM [LLVM], may eliminate these problems, however this would imply significant changes to our verification tool chain.

In addition to the results of Amani et al. [2016], we evaluated a set of standard applications on top of the C and Cogent BilbyFs implementations. The evaluation configuration is a

Application	C	COGENT
`git clone golang`	84 sec	88 sec
`make filebench`	48 sec	50 sec
`tar`	80 sec	96 sec
`du`	30 sec	31 sec

Figure 12: Running time for applications on top of BilbyFs

Mirabox [Mirabox] with 1 GiB of NAND flash, a Marvell Armada 370 single-core 1.2GHz ARMv7 processor and 1 GiB of DDR3 memory, running Linux kernel 3.17 from the Debian 6a distribution. Fig. 12 shows the running time of the applications on top of native C BilbyFs compared to COGENT BilbyFs. The first scenario clones the git repository of the language Go [Go], creating 580 directory and 5122 files, for a total of 152MB. The next test compiles the Filebench-1.4.9.1 source code [Filebench Project] with 44 C files for a total of 19K SLOC. The third application extracts a large tarball (an archlinux distribution), and the fourth measures the total size of a large directory structure. The `tar` command has a 20% slowdown, but all other scenarios show the formally verified COGENT version performing within 5% of the unverified native C implementation.

User Experience Amani et al. [2016] found that the BilbyFs proofs were significantly easier and less time consuming than direct C proofs; partly because the purely functional shallow embedding enabled a high degree of automation with Isabelle/HOL, and partly because the linear type system provided free theorems: e.g. operations that do not write to disk represent the disk as a shareable type, which automatically establishes that no writes are performed to it. This illustrates that COGENT is practical and usable for writing large amounts of code, that the generated C code is efficient, and that reasoning on top of the neat shallow embedding is much simpler than reasoning about C directly.

Effort and Size COGENT has been under development for over 2 years and has continually evolved as we have scaled it to ever larger applications. All up, the combined language development and certifying compiler took 5 person years. Engineering the COGENT compiler, excluding 33.5 person months (pm) spent on proof automation and proof framework development, consumed 10 pm. The remaining 18 pm were for the design, formalisation and proof of COGENT and its properties (e.g. the theorems of §3). The total size of the development in Isabelle is 16,423 LOC, which includes the once-and-for-all language proofs plus automated proof tactics to perform the translation validation steps, given appropriate hints from the COGENT compiler. The COGENT compiler, written in Haskell, is 11,456 SLOC. For 9,447 lines of `ext2` COGENT code the compiler generates 119,167 lines of Isabelle/HOL proofs and embeddings.

Optimisation The current COGENT compiler applies few optimisations when generating C code and leaves low-level optimisation to gcc or CompCert. Involved optimisations in the COGENT-to-C stage would complicate our current syntax-directed correspondence approach. COGENT-to-COGENT optimisations, however, are straightforward. The ease of proving A-normalisation correctness over the shallow embedding via rewriting suggests that this is the right approach in our context. In particular, some of the source-to-source optimisations discussed in Chlipala [2015] seem promising for COGENT.

6. Related Work

Like COGENT, HASP's systems language [HASP project 2010], Habit, is a functional language. It has a verified garbage collector [McCreight et al. 2010], but no full formal language semantics yet. Ivory [Pike et al. 2014] is a domain specific systems language embedded in Haskell. It generates well-defined, memory safe C code, but unlike COGENT it does not prove its correctness.

Linear types have been used in several general purpose imperative languages such as Vault [Fahndrich and DeLine 2002] and Rust. PacLang [Ennals et al. 2004] uses linear types to guide optimisation of packet processing on network processors. Similar substructural type systems, namely uniqueness types, have been integrated into the functional language Clean [Barendsen and Smetsers 1993]. However, the intent is only to provide an abstraction over effects, and thus Clean still depends on run-time garbage collection.

Hofmann [2000] proves, in pen and paper, the equivalence of the functional and imperative interpretation of a language with a linear type system. The proof is from a first order functional language to its translation in C, without any pointers or heap allocation. In contrast, COGENT is higher order, accommodates heap-allocated data, and its compiler produces a machine checked proof linking a purely functional shallow embedding to its C implementation.

As discussed in §1, Kumar et al. [2014] prove the correctness of the high-level language compiler CakeML. As it depends on run-time garbage collection, it is not suitable for our systems target. Furthemore, as it is an unrestricted, turing complete language with mutable state, its high-level semantics are not amenable to equational reasoning. By contrast, Neis et al. [2015] focus on a compositional approach to compiler verification for a relatively simple functional language, Pilsner, to an idealised assembly language. Like us, Charguéraud [2010, 2011] generate shallow embeddings to facilitate mechanical proofs, but unlike us they do not prove compilation correctness.

Chen et al. [2015] formally show in Coq [Bertot and Castéran 2004] full crash-resilience of FSCQ. FSCQ is smaller than `ext2` and BilbyFs, and an order of magnitude slower than asynchronous `ext4`. Its implementation relies on generating Haskell code from Coq, and executing that code with a full Haskell runtime in userspace. We focus on bridging high level and low level, on efficiency, and on providing a small trusted computing base, while Chen et al. [2015] assume all these are given and focus on crash resilience. The approaches are complementary, i.e. it would be straightforward to implement Crash Hoare Logic on top of Isabelle COGENT specifications, enabling a verification in the style of FSCQ.

7. Conclusions

We presented a framework for dramatically reducing the cost of formal verification of important classes of systems code. It relies on the COGENT language, its certifying compiler, their formal definitions and top-level compiler certificate theorem, and the correctness theorems for each compiler stage. COGENT targets systems code where data sharing is minimal or can be abstracted, performance and small memory footprint are requirements, and formal verification is the aim.

COGENT is a pure, total functional language to enable productive equational reasoning in an interactive theorem prover. It is higher-order and polymorphic to increase conciseness. It uses linear types to make memory management bugs compile time errors, and to enable efficient destructive in-place update. It avoids garbage collection and a trusted runtime to reduce footprint. It supports a formally modeled foreign-function interface to interoperate with C code and to implement additional data types, iterators and operations. It does all of these with full formal proof of compilation correctness and type-safety in Isabelle/HOL.

References

Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *10th ICFP*, pages 78–91, 2005.

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. Cogent: Verifying high-assurance file system implementations. In *ASPLOS*, Atlanta, GA, USA, Apr 2016.

Anonymous. Companion webpage: COGENT compiler, proofs, and file systems, 2016a. Link to the repository removed for double blind review, and is available in non-anonymous supplementary material.

Anonymous. A framework for the automatic formal verification of refinement from Cogent to C, 2016b. Submitted to ITP. Available as anonymous supplementary material.

Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *FSTTCS*, volume 761 of *LNCS*, pages 41–51, 1993.

Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security*, pages 207–221, Washington, DC, US, Aug 2015. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. 2004.

Arthur Charguéraud. Program verification through characteristic formulae. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 321–332, New York, NY, USA, 2010. URL http://doi.acm.org/10.1145/1863543.1863590.

Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 418–430, New York, NY, USA, 2011. URL http://doi.acm.org/10.1145/2034773.2034828.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *25th SOSP*, pages 18–37, Monterey, CA, Oct 2015.

Adam Chlipala. An optimizing compiler for a purely functional web-application language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 10–21, New York, NY, USA, 2015. URL http://doi.acm.org/10.1145/2784731.2784741.

David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In *21st TPHOLs*, pages 167–182, Montreal, Canada, Aug 2008.

Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 59–69, New York, NY, USA, 2001. URL http://doi.acm.org/10.1145/378795.378811.

Edsger Wybe Dijkstra. *A Discipline of Programming*. Upper Saddle River, NJ, USA, 1st edition, 1997.

Rob Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *13th ESOP*, volume 2986 of *LNCS*, pages 204–218, 2004.

Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 13–24, New York, NY, USA, 2002. URL http://doi.acm.org/10.1145/512529.512532.

Filebench Project. Filebench file system benchmark. http://sourceforge.net/projects/filebench. Accessed Nov 2015.

Go. The Go programming language. https://go.googlesource.com/go, 2015. Accessed Nov 2015.

David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of C. In *ITP*, pages 99–115, Princeton, New Jersey, USA, Aug 2012.

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *PLDI*, pages 429–439, Edinburgh, UK, Jun 2014.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *42nd POPL*, pages 595–608, 2015.

HASP project. The Habit programming language: The revised preliminary report. Technical Report http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf, Department of Computer Science, Portland State University, Portland, OR, USA, Nov 2010.

Martin Hofmann. A type system for bounded space and functional in-place update–extended abstract. In *ESOP*, volume 1782 of *LNCS*, pages 165–179, 2000.

IOZone. IOzone filesystem benchmark. http://www.iozone.org/. Accessed Mar 2015.

Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File systems deserve verification too! In *PLOS*, pages 1–7, Farmington, Pennsylvania, USA, Nov 2013.

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. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70, Feb 2014.

Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *POPL*, pages 179–191, San Diego, Jan 2014.

Xavier Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *33rd POPL*, pages 42–54, Charleston, SC, USA, 2006.

Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7): 107–115, 2009.

LLVM. The low-level virtual machine compiler. http://llvm.org/. Accessed Feb 2016.

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight Linear Types in System F°. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 77–88, New York, NY, USA, 2010. URL http://doi.acm.org/10.1145/1708016.1708027.

Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In *15th ICFP*, pages 273–284, 2010.

Mirabox. MiraBox development kit. https://www.globalscaletechnologies.com/p-58-mirabox-development-kit.aspx, 2015. Accessed Nov 2015.

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, Canada, August 31 September 2, 2015*, 2015.

Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. 2014.

Martin Odersky. Observers for linear types. In *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pages 390–407, February 1992. Lecture Notes in Computer Science 582.

Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: ten years later. In *16th ASPLOS*, pages 305–318, Newport Beach, CA, US, 2011.

Benjamin C. Pierce. *Types and Programming Languages*. 2002.

Lee Pike, Patrick Hickey, James Bielman, Trevor Elliott, Thomas DuBuisson, and John Launchbury. Programming languages for high-assurance autonomous vehicles: Extended abstract. In *2014PLPV*, pages 1–2, San Diego, California, USA, 2014.

Rust. The Rust programming language. http://rustlang.org, 2014. Accessed March 2015.

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *SIGPLAN Lisp Pointers*, V(1):288–298, Jan 1992.

Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL.* PhD thesis, Technische Universität München, 2006.

Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–481, Seattle, Washington, USA, Jun 2013.

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