JFP, 67 pages, 2020.
doi:10.1017/xxxxx

Cogent: Uniqueness Types and Certifying Compilation

LIAM O'CONNOR
University of Edinburgh, Scotland
(e-mail: l.oconnor@ed.ac.uk)

ZILIN CHEN
UNSW Sydney, Australia

CHRISTINE RIZKALLAH
University of Melbourne, Australia

VINCENT JACKSON
UNSW Sydney, Australia

SIDNEY AMANI
Canva, Australia

GERWIN KLEIN
Proofcraft and UNSW Sydney, Australia

TOBY MURRAY
University of Melbourne, Australia

THOMAS SEWELL
University of Cambridge, United Kingdom

GABRIELE KELLER
Utrecht University, The Netherlands

Abstract

This paper presents a framework aimed at significantly reducing the cost of proving functional correctness for low-level operating systems components. The framework is designed around a new functional programming language, Cogent. A central aspect of the language is its uniqueness type system, which eliminates the need for a trusted runtime or garbage collector while still guaranteeing memory safety, a crucial property for safety and security. Moreover, it allows us to assign two semantics to the language: The first semantics is imperative, suitable for efficient C code generation, and the second is purely functional, providing a user friendly interface for equational reasoning and verification of higher level correctness properties. The refinement theorem connecting the two semantics allows the compiler to produce a proof via translation validation certifying the correctness of the generated C code with respect to the semantics of the Cogent source program. We have demonstrated the effectiveness of our framework for implementation and for verification through two file system implementations.

1 Introduction

The correctness of any application critically depends on the correctness of the systems on which it relies. Proving the correctness of systems code, however, is particularly challenging because it is usually written in low-level languages such as C, which provide fine grained control over the program execution, but few abstraction mechanisms and static guarantees. The verification framework we present in this paper addresses this problem. It enables the programmer to write low-level systems code in Cogent, a purely functional language with a strong static type system. It facilitates simpler verification of code via equational reasoning in the interactive theorem prover Isabelle/HOL (Nipkow *et al.*, 2002), through a *certifying compiler* from Cogent to efficient C code. This compiler, given a well-typed program, produces a high-level shallow embedding of the program's semantics in Isabelle/HOL, suitable for equational reasoning, as well as a proof that connects this shallow embedding to the compiler-generated C code. As a consequence, any functional correctness property proved of the shallow embedding is guaranteed to hold for the generated C. Because the code generated by Cogent is within the subset of the binary verification tool of Sewell *et al.* (2013), it is possible in principle to extend this compilation certificate all the way down to the binary level.

The compilation target of our compiler is C, because it is the language in which most existing systems code is written, and because with the advent of tools like CompCert (Leroy, 2009*b*) and gcc translation validation (Sewell *et al.*, 2013), large subsets of C now have a formalised semantics and an existing formal verification infrastructure. Why, then, do we not opt to verify C systems code directly? After all, there is an ever growing list of successes (Beringer *et al.*, 2015; Gu *et al.*, 2016; Klein *et al.*, 2009) 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: the proof engineer faces 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.

In contrast to CakeML (Kumar *et al.*, 2014), which is the state of the art for certifying compilation of general purpose functional languages, Cogent is targeted at a substantially different application area and 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 the implementation of theorem provers. Cogent is a restricted language of total functions with intentionally simple, pure semantics that are easy to reason about equationally. 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.

The main restrictions of Cogent are the (purposeful) lack of built-in iteration or recursion, and its uniqueness 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.

The lack of recursion in Cogent is not much of a problem in practice, given the target domain, but iteration over finite structures is of course necessary. This is where Cogent's integrated foreign function interface (FFI) comes in: engineers can provide their own verified data types and iterator interfaces in C and use them seamlessly in Cogent, including in formal reasoning. Our framework guarantees that the verification of combined C-Cogent code bases has no room for unsoundness.

To evaluate the suitability of the framework, we performed two major case studies and implemented two full-scale Linux file systems (Amani *et al.*, 2016)—the standard Linux `ext2` and the BilbyFs Flash file system (Keller *et al.*, 2013) and prove two core functional correctness properties of BilbyFs. These file systems were competitive in performance with their C counterparts. This illustrates that Cogent is suitable both for implementation and proofs, dramatically reducing the cost of verifying correctness of practical file systems. These case studies are beneficial in their 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*a*). The benefits of this language-based approach for file system verification were conjectured by Keller *et al.* (2013) and are confirmed by our work.

Cogent is restricted, but it is 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. Our main contribution is the framework for significantly reducing the cost of formal verification for important classes of systems code, using this language-based approach for automatically co-generating code and proofs.

This paper is the consolidation of a long research programme consisting of several conference papers and a PhD thesis (O'Connor, 2019). Specifically, this paper presents:

- The Cogent language (§2), its certifying compiler and static semantics (§3). The version of the type system featured in this paper includes a new *subtyping* feature that was not present in its initial presentation (O'Connor *et al.*, 2016). We present our formalisation of this feature in Section 3 and discuss its impact in Section 5.
- The formal semantics of Cogent, as well as a machine-checked proof for switching from imperative update semantics to functional value semantics for a full-featured functional language, justified by uniqueness types (§4). We build upon well-known theoretical results about linear types, accounting for pointers and heap allocation. We also formally specify the assumptions required for C code imported via the FFI to maintain the guarantees of the uniqueness type system and the overall refinement certificate. This work was originally presented by O'Connor *et al.* (2016).
- The top-level compiler certificate, and the verification stages that make up the compiler correctness theorem (§5), including automated refinement calculi, formally verified type checking, A-normalisation, and monomorphisation. The final stage

connecting Cogent and C code relies on a sophisticated refinement calculus, which is summarised in Section 5 with more technical details available from Rizkallah *et al.* (2016).

The implementation and verification of our case study file systems are discussed in detail in other work (Amani *et al.*, 2016; Amani, 2016), and briefly summarised here. These case studies demonstrate Cogent's suitability for systems programming, as well as its potential to reduce the cost of functional correctness verification for real world systems.

2 Cogent

Before we discuss the formalisation of the static and dynamic semantics, and the verification of Cogent software, let us first examine Cogent as a programming language. In this section, we will give a short tutorial on Cogent, and briefly discuss the experience of writing systems software in Cogent with reference to our two case studies.

Cogent is a functional language, with syntax resembling ML or Haskell:

```
add : (U32, U32) → U32
add (x, y) = x + y
```

Arithmetic operations (e.g. +) are overloaded to be used on any numeric type (e.g. U8), as long as the arguments both have the same numeric type. Cogent does not presently support closures, so partial application via currying is also not common. As a consequence, multi-parameter functions typically take tuples of their arguments.

Cogent supports conditionals, non-recursive let-bindings, and pattern matching. The syntax of the latter is more lightweight than in Haskell or ML, because pattern matching is used in Cogent for error-handling situations that would make use of exceptions in Haskell or ML. Also unlike those languages, our patterns must also be *exhaustive*. Omitting a case is not just a warning but a compile error. To match on an expression, a series of vertically-aligned pipe characters (|) are placed after the expression, one for each case, rendered in this paper as a solid vertical line.

Consider the following function add′, which again adds to unsigned 32-bit numbers, but this time using pattern matching to check for and handle overflow:

```
add′ : (U32, U32) → U32
add′ (x, y) =
  let out = x + y
  in out < x ‖ out < y
    | True → 0
    | False ⇒ out
```

The programmer can convey optimisation information to the underlying C compiler to determine the likelihood of each branch by choosing different arrow symbols, → (->) and ⇒ (=>) in the example, for normal and likely branches respectively.

2.1 Variant types

Known in other languages as a *tagged union* or a *sum type*, a variant type describes values that may be one of several types, disambiguated by a *tag* or *constructor*. For example, a value of type ⟨Failure U16 | Success U8⟩ may contain *either* an eight-bit or sixteen-bit unsigned integer, depending on which constructor (Success or Failure) is used.

Using the unit type (written ()), the type with a single trivial inhabitant (also written ()), we can also use variant types to construct the familiar `Option` or `Maybe` types from ML or Haskell:

$$\textbf{type}\ \mathit{Option}\ a = \langle \text{None}\ () \mid \text{Some}\ a \rangle$$

Here we have used the syntax for *type synonyms* in Cogent. While *Option* U8 is easier for humans to write, the Cogent type system makes absolutely no distinction between *Option* U8 and ⟨None () | Some U8⟩.

A variant type may include any number of constructors:

$$\textbf{type}\ \mathit{CarState} = \langle \text{Drive U32} \mid \text{Neutral}\ () \mid \text{Reverse U32} \rangle$$

Variant types are deconstructed via pattern matching, and constructed by simply typing a constructor name followed by its parameter. Constructor names are required to begin with a capital letter, so that they can be disambiguated from variables and functions.

To ease implementation, compatibility with Isabelle, and to avoid costly backtracking, we require that the pattern for a constructor's argument be *irrefutable* (i.e. a pattern like (Some (Some v)) is not allowed). An irrefutable pattern will always successfully match against any well-typed value.

2.2 Subtyping and variant types

Our requirement that pattern matching may not fail could, with a simplistic type system, lead to a significant amount of dead code. For example, the caller of the accelerate function has to handle the case for the Neutral constructor, despite the fact that this case will never be executed.

```
accelerate : (CarState, U32) → CarState
accelerate (st, δ) =
  st
    | Drive vel → Drive (vel + δ)
    | Neutral () → Drive δ
    | Reverse vel → Reverse (vel + δ)
```

One simple way to solve this problem is to have a version of *CarState* without the Neutral constructor:

$$\textbf{type}\ \mathit{CarState}' = \langle \text{Drive U32} \mid \text{Reverse U32} \rangle$$

We can then use this to give the above function a more precise type:

$$\text{accelerate} : (\mathit{CarState}, \text{U32}) \to \mathit{CarState}'$$

While this would type-check, the representation of *CarState′* is completely independent of *CarState*. This means that, even though a trivial injection exists from *CarState′* to *CarState*, any function that makes use of *CarState* cannot accept a *CarState′* without a potentially expensive copy.

We address this problem by allowing the programmer to specify that certain constructors of a variant type are statically known not to be present, using the **take** keyword:

$$\mathsf{accelerate} : (\mathit{CarState}, \mathtt{U32}) \to \mathit{CarState}\ \textbf{take}\ \text{Neutral}$$

Unlike *CarState′*, the type *CarState* **take** Neutral has the same run-time representation as *CarState*, and thus can be trivially coerced into the broader type. Indeed, the type checker will automatically perform such coercions via subtyping.

Such additional static information also becomes useful when default cases are used in pattern matching, for example:

```
bounce : CarState → CarState take Drive
bounce
  | Drive vel → Reverse vel
  | st → st
```

Note that the local variable *st* here is of type *CarState* **take** Drive because the Drive constructor has already been matched.

2.3 Abstract types and functions

By omitting the implementations of functions and type definitions, we declare them to be *abstract*. Abstract types and functions are defined outside of Cogent. Typically, an implementation is provided in C. The Cogent compiler includes powerful infrastructure for compiling C implementations along with Cogent code, including the embedding of Cogent types and expressions inside C code using quasi-quotation.

```
type Buffer

poke : (Buffer, U32, U8) → Buffer
```

Outwardly, the interface of this *Buffer* type seems purely functional, however Cogent assumes by default that all abstract types are *linear*. This means that any variable of type *Buffer*, or any compound type such as a variant that could potentially contain a *Buffer*, must be used exactly once. This scheme of *uniqueness types* ensures that there is only one active reference to a given *Buffer* object at any given time. Therefore, the C implementation of poke is free to destructively update the provided *Buffer* without contradicting the purely functional semantics of Cogent.

```
hello : Buffer → Buffer
hello buf =
  let buf = poke (buf, 0, 'H')
  and buf = poke (buf, 1, 'e')
  and buf = poke (buf, 2, 'l')
  and buf = poke (buf, 3, 'l')
  and buf = poke (buf, 4, 'o')
  in buf
```

In the above example, while it would appear that many intermediate buffers are created, the real implementation is merely a series of destructive updates to the same buffer.

2.4 Suspending uniqueness

When we are only reading from a data structure, uniqueness types complicate a program unnecessarily, as the structure would have to be threaded through the program. For example, a simple peek function to read from a buffer would, if *Buffer* were linear, have to have this cumbersome type:

$$\mathsf{peek}' : (\mathit{Buffer}, \mathtt{U32}) \to \langle \mathrm{Err}\ \mathit{Buffer} \mid \mathrm{Ok}\ (\mathtt{U8}, \mathit{Buffer}) \rangle$$

The ! type operator helps to avoid this problem. This operator converts any *linear*, *writable* type to a *read-only* type that can be freely shared or discarded. This is analogous to a shared reference in the type system of Rust. A function that takes a value of type `Buffer!` is free to *read* from the buffer, but is unable to *write* to it.

$$\mathsf{peek}' : (\mathit{Buffer}!, \mathtt{U32}) \to \langle \mathrm{Err}\ () \mid \mathrm{Ok}\ \mathtt{U8} \rangle$$

A value of type *Buffer* can be temporarily converted to a *Buffer*! using the expression-level ! construct. By placing a ! followed by a variable name after any let binding, match scrutinee or **if** condition, the variable will be made temporarily read-only for the duration of that expression.

For example, a function that writes a character to the address specified at the beginning of a buffer combines both read-only and writable uses of the same buffer:

```
writeChar : (U8, Buffer) → ⟨Err Buffer | Ok Buffer⟩
writeChar (c, buf) =
  peek' (buf, 0) !buf
  | Ok i ⇒ Ok (poke (buf, i, c))
  | Err () → Err buf
```

Here the use of the ! post-fix on the third line allows the *buf* variable to be used both in a read-only way as an argument to peek' and in a writable way as an argument to poke.

To ensure that read-only references are never simultaneously live with writable references, we require that any such !-annotated expression must not contain any use of the ! operator in its type. This restriction prevents types with the ! operator from escaping the scope in which they are used, which is necessary to be able to reason equationally about Cogent programs and to preserve the refinement theorem connecting the two semantics.

2.5 Higher-order functions

As Cogent does not support recursion, iteration is expressed through the use of abstract *higher-order functions*, providing basic functional traversal combinators such as map and fold for abstract types. For example, the *Buffer* type described above could have a map function like:

$$\mathsf{map} : (\mathtt{U8} \to \mathtt{U8}, \mathit{Buffer}) \to \mathit{Buffer}$$

Here our map function is able to destructively overwrite the buffer with the results of the function applied to each byte.

While Cogent does support higher-order functions (functions that accept functions as arguments or return functions), it does not yet support nested lambda abstractions or closures, as these can require allocation if they capture variables. Thus, to invoke this map function, a separate top-level function must be defined for its argument.

2.6 Polymorphism

Cogent also supports *parametric polymorphism*. Our compiler generates multiple specialised C implementations from a polymorphic C template, one for each concrete instantiation used in the Cogent code.

Polymorphic functions can be instantiated to concrete types using square brackets. This type application syntax is not always necessary — the type checker can often infer the omitted types.

$$\mathsf{foldBuf} : \forall a.\ (\mathit{Buffer}!, (\mathtt{U8}, a) \to a,\ a) \to a$$

$$\mathsf{sumBuf} : \mathit{Buffer}! \to \mathtt{U32}$$
$$\mathsf{sumBuf}\ \mathit{buf} = \mathsf{foldBuf}[\mathtt{U32}]\ (\mathit{buf}, \mathsf{sumHelper}, 0)$$

$$\mathsf{sumHelper} : (\mathtt{U8}, \mathtt{U32}) \to \mathtt{U32}$$
$$\mathsf{sumHelper}\ (x, y) = (\textbf{upcast}\ x) + y$$

As in ML, polymorphic functions are not first class — we only allow polymorphic definitions on the top-level. Variables of polymorphic type are by default treated as *linear* — they must be used exactly once — this allows the polymorphic type variable to be instantiated to any type, shareable or not. Additional *constraints* can be placed on the type variable (before the $\Rightarrow$ symbol, as in Haskell) to restrict the possible instantiations to those that can be shared:

$$\mathsf{dup} : \forall a.\ (\mathrm{Share}\ a) \Rightarrow a \to (a, a)$$
$$\mathsf{dup}\ a = (a, a)$$

In addition to Share constraints, we also allow Drop constraints, which require instantiations to be discardable without being used, as well as Escape constraints, which require instantiations to be safe to return from a !-annotated expression. Multiple constraints can be specified as follows:

$$\mathsf{dupOrDrop} : \forall a.\ (\mathrm{Drop}\ a, \mathrm{Share}\ a) \Rightarrow (\mathit{Bool}, a) \to \langle \mathrm{Drop}\ () \mid \mathrm{Dup}\ (a, a) \rangle$$
$$\mathsf{dupOrDrop}\ (b, a) = \textbf{if}\ b\ \textbf{then}\ \mathrm{Dup}\ (a, a)\ \textbf{else}\ \mathrm{Drop}$$

Abstract types may be given type parameters also, such as in the *Array* type given below. As with abstract functions, this will correspond to a family of automatically generated C types for each concrete type used in the Cogent code. The type-level ! operator can also be applied to type variables, as shown in the abstract fold function below, for abstract arrays:

$$\textbf{type}\ \mathit{Array}\ a$$

$$\mathsf{fold} : ((\mathit{Array}\ a)!, (a!, b) \to b, b) \to b$$

```
type Heap

type Bag = {count : U32, sum : U32}

newBag : Heap → ⟨Failure Heap | Success (Bag, Heap)⟩

freeBag : (Heap, Bag) → Heap

addToBag : (U32, Bag) → Bag
addToBag (x, b {count = c, sum = s}) =
    b {count = c + 1, sum = s + x}


averageBag : Bag! → ⟨EmptyBag | Success U32⟩
averageBag (b {count, sum}) =
    if count == 0 then EmptyBag else Success (sum / count)


type List a

reduce : ∀a b. (List a!, (a!, b) → b, b) → b

average : (Heap, List U32!) → (Heap, U32)
average (h, ls) =
    newBag h
    | Success (bag, h') → let bag' = reduce (ls, addToBag, bag)
        in averageBag bag'  !bag'
        | Success n → (freeBag (h', bag'), n)
        | EmptyBag → (freeBag (h', bag'), 0)

    | Failure h' → (h', 0)
```

Fig. 1. Average using a Bag of numbers

2.7 Records

Cogent also supports *records*, which may be heap-allocated (and thus linear) or stack-allocated. Like variants, certain fields of a record can be statically marked unavailable in a type using the **take** keyword. We describe our record system in more detail in Section 3.

Figure 1 contains an example of a complete Cogent program, including the use of records. Assuming an abstract *List* data structure with a reduce function (which aggregates a *List* using a given aggregation function and identity element), the function average computes the average of a list of 32-bit unsigned integers. It accomplishes this by storing the running total and count in a heap-allocated data structure called a *Bag*. We define the *Bag* as a heap-allocated record containing two 32-bit unsigned integers, and introduce allocation and free functions for *Bag*s. The newBag function returns a variant, indicating that either a bag and a new heap will be returned in the case of Success, or, in the case of allocation Failure, no new bag will be returned. The addToBag function demonstrates the use of pattern-matching to destructure the heap-allocated record to gain access to its fields, and update it with new values for each. The averageBag function returns, if possible, the average of the numbers added to the *Bag*. The input type *Bag*! indicates that the input is a read-only,

freely shareable view of a *Bag*. This view of the *Bag* is made with the ! notation in the average function, which creates a *Bag* with newBag, pattern matches on the result, and, if allocation was successful, adds every number in the given list to it, and returns their average.

2.8 Cogent for systems programming

In our previous work, we conducted a case study into the implementation and verification of software systems written in Cogent (Amani *et al.*, 2016). Two file systems were implemented by systems programmers who were not Cogent developers but were experienced in functional programming. The first is an almost feature-complete implementation of the `ext2` revision 1 file system, passing the POSIX File System Test Suite (ntfs3g, *n.d.*) for all implemented features. Its performance is comparable to the implementation of `ext2` that is included as part of the Linux Kernel. The second is a flash file system BilbyFs, designed from the ground up to be easy to verify (Keller *et al.*, 2013).

The Cogent implementations of `ext2` and BilbyFs share a common C library of abstract data types that includes fixed-length arrays for words and structures, simple iterators for implementing loops, and Cogent stubs for accessing a range of Linux APIs such as the buffer cache and its native red-black tree implementation. The interfaces exposed by this library are carefully designed to ensure compatibility with Cogent's uniqueness type system.

The `ext2` implementation demonstrates Cogent's ability to enable re-engineering of existing file systems, and thus its potential to provide an incremental upgrade path to increase the reliability of existing systems code. BilbyFs, on the other hand, provides a glimpse of how to design and engineer new file systems that are not only performant, but amenable to being verified as correct against a high-level specification of file system correctness.

2.9 Experience with Cogent

In order to shed light on Cogent's usability as a systems programming language, we briefly describe the experience of developing the `ext2` and BilbyFs implementations. In both cases, a manually-written C implementation was used as a starting point: In the case of `ext2`, this was Linux's `ext2fs` implementation; for BilbyFs, it was our own implementation of the file system that was used to prototype its design (Keller *et al.*, 2013). The two file systems were written by separate developers, but in the case of BilbyFs, the same developer wrote both the C and Cogent implementations. Both developers were already familiar with functional programming.

Naturally, Cogent itself evolved in the process — at the time of the initial implementations, the language had uniqueness types, but no polymorphism nor higher-order functions. The developers jointly wrote the shared C library, and the `ext2` developer spent considerable time assisting with Cogent tool-chain design and development. Unfortunately, this makes it infeasible to give accurate effort estimates for how long each file system would have taken to write had the language and tool-chain been stable, as they are now. Having to adopt Cogent's functional style was not a major barrier for either developer; indeed one reported that Cogent's use of **let**-expressions for sequencing and pattern matching for error handling aided his understanding of the potential control paths of his code. While both had to get used

	Original C	Cogent	Generated C
ext2	4, 077	2, 789	12, 066
BilbyFs	4, 021	4, 643	18, 182

(Generated line counts include C library.)

Table 1. Implementation source lines of code, measured with `sloccount`.

to the uniqueness type system, both reported that this happened quite quickly and that the type system generally did not impose much of a burden when writing ordinary Cogent code. Both developers noted the usefulness of Cogent's uniqueness types for tracking memory allocation and catching memory leaks. Uniqueness types were reported to cause some friction when having to design the shared C library interfaces to respect the constraints of the type system.

Both developers reported that the strong type system provided by Cogent decreased the time they usually would have spent debugging, which is to be expected. Of course, logic bugs which cannot be captured by the static semantics could remain in Cogent code. Such bugs are harder to debug than in a comparable C implementation, because of the lack of debugging tool support for Cogent. The developers, however, found comparatively few bugs in the Cogent code; the vast majority of bugs were in the C code that accompanies it.

Table 1 shows the source code sizes of the two systems. For the original `ext2` system (i.e. the Linux code) we exclude code that implements features that the Cogent implementation does not support. We can see that for the `ext2` system, the Cogent implementation is about two-thirds the size of C.

BilbyFs' Cogent implementation is larger than `ext2`'s, relative to their respective original C implementations. This is because BilbyFs makes heavier use of the various abstract data types available in the C library, some of which present fairly verbose client interfaces in their current implementation.

The blowout in size of the generated C code is mostly a result of normalisation steps applied by the Cogent compiler, most of which is easily optimised away by the C compiler. The performance of these file systems is generally competitive with their C counterparts (Amani *et al.*, 2016), however we found that `gcc`'s optimiser does an unsatisfactory job of optimising operations on large structs, resulting in some unnecessary copy operations left in the code. Our new subtyping feature helps to reduce the amount of copying in the generated C code, however more work needs to be done to generate C code that is more in line with the expectations of the C optimiser.

3 Static semantics

The central feature of the Cogent language is its system of *uniqueness types* (de Vries *et al.*, 2008). It is this feature that allows it to be interpreted simultaneously (and *equivalently*) as both purely functional and fully imperative – combining destructive updates with equational reasoning. This semantic coincidence, discussed in Section 4, is the foundation of the overall refinement certificate of Section 5. In this section, we will formally describe the type system for a minimal version of Cogent. The process of type inference and elaboration from the surface level language to this core language is detailed by O'Connor (2019), and is outside the scope of this paper.

expressions	e	$::=$	$x \mid \ell$	
		$\mid$	$e_1 \wr e_2$	(*primops*)
		$\mid$	$e_1\ e_2 \mid f\,[\vec{\tau_i}]$	(*applications*)
		$\mid$	$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	
		$\mid$	$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	
		$\mid$	$e :: \tau$	(*type signatures*)
		$\mid$	$\cdots$	
types	τ, ρ	$::=$	$a \mid \tau_1 \to \tau_2 \mid T \mid \cdots$	
prim. types	T	$::=$	`U8` $\mid$ `U16` $\mid$ `U32` $\mid$ `U64` $\mid$ `Bool`	
operators	$\wr$	$::=$	$+ \mid \leq \mid \neq \mid \wedge \mid \cdots$	
literals	ℓ	$::=$	`True` $\mid$ `False` $\mid \mathbb{N}$	
constraints	C	$::=$	$\tau_1 \sqsubseteq \tau_2$	(*subtyping*)
		$\mid$	$\tau\ \mathbf{Share} \mid \tau\ \mathbf{Drop}$	(*contract/weaken*)
contexts	Γ	$::=$	$\overrightarrow{x : \tau}$	
axiom sets	A	$::=$	$\overline{a_i\ \mathbf{Drop}}, \overline{b_j\ \mathbf{Share}}$	
polytypes	π	$::=$	$\forall \vec{a}.\ C \Rightarrow \tau$	
type vars	a, b, c			
variables	x, y, z			

$\overrightarrow{\text{Harpoons}}$ indicate a list of zero or more.
$\overline{\text{Overlines}}$ indicate a set, i.e. order is not important.

(Continued in Figures 6, 9, and 12)

Fig. 2. Syntax of the basic fragment of Cogent

$$\boxed{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}{A \vdash x : \tau, \Gamma \rightsquigarrow x : \tau, \Gamma_1 \boxplus \Gamma_2}\,\mathrm{L} \qquad \frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}{A \vdash x : \tau, \Gamma \rightsquigarrow \Gamma_1 \boxplus x : \tau, \Gamma_2}\,\mathrm{R}$$

$$\frac{}{A \vdash \varepsilon \rightsquigarrow \varepsilon \boxplus \varepsilon} \qquad \frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad A \vdash \tau\ \mathbf{Share}}{A \vdash x : \tau, \Gamma \rightsquigarrow x : \tau, \Gamma_1 \boxplus x : \tau, \Gamma_2}\,\mathrm{C}$$

$$\boxed{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'}$$

$$\frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'}{A \vdash x : \tau, \Gamma \overset{\text{weak}}{\rightsquigarrow} x : \tau, \Gamma'}\,\mathrm{K} \qquad \frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma' \qquad A \vdash \tau\ \mathbf{Drop}}{A \vdash x : \tau, \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'}\,\mathrm{D}$$

$$\frac{}{A \vdash \varepsilon \overset{\text{weak}}{\rightsquigarrow} \varepsilon}$$

Fig. 3. Context relations

For Cogent, typical ML-style type systems are simultaneously too rich, as they support local polymorphic bindings which Cogent disallows; and somewhat deficient, as they assume that the theory includes a *structural context*. That is, they accept the following *structural* laws implicitly:

$$\frac{\Gamma_1\Gamma_2 \vdash e : \tau}{\Gamma_2\Gamma_1 \vdash e : \tau}\,\textsc{Exchange} \qquad \frac{\Gamma_1 \vdash e : \tau}{\Gamma_1\Gamma_2 \vdash e : \tau}\,\textsc{Weakening} \qquad \frac{\Gamma_1\Gamma_1\Gamma_2 \vdash e : \tau}{\Gamma_1\Gamma_2 \vdash e : \tau}\,\textsc{Contraction}$$

These laws, which respectively state that we may swap, drop, or duplicate assumptions whenever necessary, allow the typing context to be treated as a *set*. Indeed, in many such calculi the rule for variables is presented as

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\,\textsc{Var}$$

where the WEAKENING rule is implicitly used to discard unneeded assumptions, rather than the more precise version of the rule:

$$\frac{}{x : \tau \vdash x : \tau}\,\textsc{Var}$$

As Cogent makes use of a *substructural* type system, specifically uniqueness types, we must be substantially more precise when dealing with contexts. We do not accept the rules of CONTRACTION and WEAKENING universally. Admitting CONTRACTION for any type would allow multiple references to a mutable object to be accessible at one time, thus breaking the semantic correspondence Cogent enjoys. Admitting WEAKENING for any type would allow resources to be discarded without being properly disposed.[1] Rather than a set, a context is now a *multiset*, where each assumption about a variable is viewed as a one-use *permission* to type that variable.

Of course, not *all* types benefit from such linearity restrictions. For example, it would be most inconvenient if one was forced to use a variable of type `Bool` exactly once. Thus, it becomes beneficial to allow contraction and weakening for some types, but not others.

To cleanly accomplish this, we move the manipulation of contexts out of the structural rules; instead reifying them as the explicit relations given in Figure 3. We define a *context-splitting* operation, used for typing the *branches* of the abstract syntax tree, which, given assumptions A about the linearity of polymorphic type variables, splits a context Γ into two sub-contexts Γ_1 and Γ_2. Each assumption from Γ must be put into either Γ_1 or Γ_2. An assumption may only be distributed into *both* sub-contexts if it is *shareable*, i.e. it contains no unique references. We also define a *weakening* relation, used for typing the *leaves* of the abstract syntax tree, which, under assumptions A, weakens a context Γ into a smaller context Γ', where each discarded assumption must have a *discardable* type. The specifics of what makes a type *shareable* or *discardable* are encapsulated by the **Share** and **Drop** judgements respectively, definitions of which are provided later in Figure 5. The fragment of Cogent defined in Figure 2 contains only primitive types, however, which are all freely shareable and discardable.

[1] Although it is possible to statically insert destructor code as linear variables go out of scope, giving an *affine* type system, this complicates implementation and is omitted for now.

$$\boxed{A; \Gamma \vdash e : \tau}$$

$$\frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} x : \tau}{A; \Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{A; \Gamma \vdash e : \tau}{A; \Gamma \vdash e :: \tau : \tau}\ \textsc{Sig}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A; \Gamma_1 \vdash e_1 : \tau_1 \to \tau_2 \quad A; \Gamma_2 \vdash e_2 : \tau_1}{A; \Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

$$\frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \varepsilon \quad \mathit{typeOf}(f) = \forall \vec{a_i}.\ C \Rightarrow \tau \quad A \vdash C \left[\vec{\tau_i}/\vec{a_i}\right]}{A; \Gamma \vdash f\,[\vec{\tau_i}] : \tau \left[\vec{\tau_i}/\vec{a_i}\right]}\ \textsc{TApp}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A; \Gamma_1 \vdash e_1 : \tau_1 \quad A; x : \tau_2, \Gamma_2 \vdash e_2 : \tau_2}{A; \Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ \textsc{Let}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A; \Gamma_1 \vdash e_1 : \texttt{Bool} \quad A; \Gamma_2 \vdash e_2 : \tau \quad A; \Gamma_2 \vdash e_3 : \tau}{A; \Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}\ \textsc{If}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad T \neq \texttt{Bool} \quad \wr \in \{+, -, \times, \div, \dots\} \quad A; \Gamma_1 \vdash e_1 : T \quad A; \Gamma_2 \vdash e_2 : T}{A; \Gamma \vdash e_1 \wr e_2 : T}\ \textsc{IOp}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad T \neq \texttt{Bool} \quad \wr \in \{=, \neq, <, >, \leq, \geq\} \quad A; \Gamma_1 \vdash e_1 : T \quad A; \Gamma_2 \vdash e_2 : T}{A; \Gamma \vdash e_1 \wr e_2 : \texttt{Bool}}\ \textsc{COp}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad \wr \in \{\wedge, \vee\} \quad A; \Gamma_1 \vdash e_1 : \texttt{Bool} \quad A; \Gamma_2 \vdash e_2 : \texttt{Bool}}{A; \Gamma \vdash e_1 \wr e_2 : \texttt{Bool}}\ \textsc{BOp}$$

$$\frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \varepsilon \quad \ell \in \mathbb{N} \quad \ell < |T|}{A; \Gamma \vdash \ell : T}\ \textsc{ILit} \qquad \frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \varepsilon \quad \ell \in \{\texttt{True}, \texttt{False}\}}{A; \Gamma \vdash \ell : \texttt{Bool}}\ \textsc{BLit}$$

(Continued in Figures 7, 11, and 13)

Fig. 4. Some basic typing rules

$$\boxed{A \vdash C}$$

$$\frac{}{A \vdash \tau \sqsubseteq \tau}\ \textsc{Refl} \qquad \frac{C \in A}{A \vdash C}\ \textsc{Asm} \qquad \frac{A \vdash \rho_1 \sqsubseteq \tau_1 \quad A \vdash \tau_2 \sqsubseteq \rho_2}{A \vdash (\tau_1 \to \tau_2) \sqsubseteq (\rho_1 \to \rho_2)}\ \textsc{Fun}$$

$$\frac{}{A \vdash \tau_1 \to \tau_2\ \textbf{Share}}\ \textsc{Fun-S} \qquad \frac{}{A \vdash \tau_1 \to \tau_2\ \textbf{Drop}}\ \textsc{Fun-D}$$

$$\frac{}{A \vdash T\ \textbf{Share}}\ \textsc{Prim-S} \qquad \frac{}{A \vdash T\ \textbf{Drop}}\ \textsc{Prim-D}$$

(Continued in Figures 8, 10, and 14)

Fig. 5. Constraint semantics

Figure 4 contains the typing rules for this elementary fragment of Cogent: just variables (VAR), literals (ILIT and BLIT), binary operators (IOP, BOP and COP), conditionals (IF), and local monomorphic bindings (LET). For simplicity, Cogent does not currently include lambda abstractions or local polymorphism. Thus, all function definitions or polymorphic definitions must occur on the top-level. We assume the existence of a global environment $typeOf(\cdot)$ that includes the complete types of all top-level definitions so far. The rule TAPP allows these top-level polymorphic definitions to be used and instantiated.

3.1 Variant types

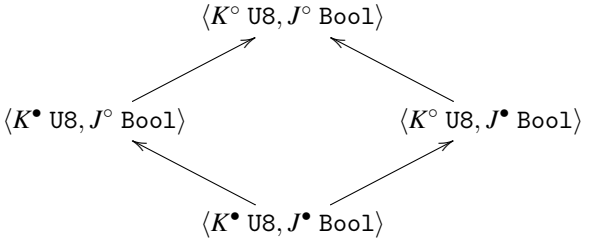
Variants in Cogent are an anonymous n-ary sum type consisting of a set of *constructor* names paired with types. The syntax for variants is given in Figure 6. Users may construct a value of variant type by invoking a constructor, as in

$$K\ 42 : \langle K^{\circ}\ \texttt{U8}, J^{\bullet}\ \texttt{Bool}\rangle$$

We also tag each constructor with a usage tag, either $\bullet$ or $\circ$, for use in exhaustivity checking for pattern matching. These usage tags are how we represent the type-level **take** annotations on variant types seen in Section 2 in our core language. A constructor is marked with $\bullet$ if it is statically known that this constructor is *not* the one actually used to construct the value. In this way, we can ensure exhaustivity by only permitting irrefutable patterns when *every* other constructor is marked with $\bullet$. Unfortunately, this necessitates the addition of subtyping to the type system. Take this simple example:

$$\begin{array}{l}\textbf{if}\ (condition)\ \textbf{then}\\ \quad K\ 42 : \langle K^{\circ}\ \texttt{U8}, J^{\bullet}\ \texttt{Bool}\rangle\\ \textbf{else}\\ \quad J\ \texttt{True} : \langle K^{\bullet}\ \texttt{U8}, J^{\circ}\ \texttt{Bool}\rangle\\ : \langle K^{\circ}\ \texttt{U8}, J^{\circ}\ \texttt{Bool}\rangle\end{array}$$

Note that the types for the two branches of the conditional differ only in the static knowledge we have of the constructor. The two types have the same run-time representation, so it is safe to *discard* information in order to type the expression. Fortunately, this subtyping is fairly well behaved, forming a complete lattice for each variant type:



This subtyping feature is new to Cogent when compared to previously published work (O'Connor *et al.*, 2016), and significantly simplified the shallow embeddings produced by the compiler. The impact of this feature is discussed in more detail in Section 5.6.

expressions	e	$::=$	$\cdots \mid K\ e$	(*variant constructor*)
		$\mid$	$\mathbf{case}\ e_1\ \mathbf{of}\ K\ x.\ e_2\ \mathbf{else}\ y.\ e_3$	(*pattern matching*)
		$\mid$	$\mathbf{case}\ e_1\ \mathbf{of}\ K\ x.\ e_2$	(*irrefutable match*)
types	τ, ρ	$::=$	$\cdots \mid \langle \overline{K^{\mathrm{u}}\ \tau} \rangle$	(*variant types*)
usage tags	u	$::=$	$\circ$	(*unused*)
		$\mid$	$\bullet$	(*used*)
constructors	K			

Fig. 6. Syntax for variants

$$\boxed{A; \Gamma \vdash e : \tau}$$

$$\cdots$$

$$\frac{A; \Gamma \vdash e : \tau' \qquad A \vdash \tau' \sqsubseteq \tau}{A; \Gamma \vdash e : \tau}\ \textsc{Sub} \qquad \frac{A; \Gamma \vdash e : \tau}{A; \Gamma \vdash K\ e : \langle K^{\circ}\ \tau, \overline{K_i^{\bullet}\ \tau_i} \rangle}\ \textsc{vCon}$$

$$\frac{\begin{array}{c} A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad A; \Gamma_1 \vdash e_1 : \langle K^{\circ}\ \rho, \overline{K_i^{\mathrm{u}}\ \tau_i} \rangle \\ A; x : \rho, \Gamma_2 \vdash e_2 : \tau \qquad A; y : \langle K^{\bullet}\ \rho, \overline{K_i^{\mathrm{u}}\ \tau_i} \rangle, \Gamma_2 \vdash e_3 : \tau \end{array}}{A; \Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ K\ x.\ e_2\ \mathbf{else}\ y.\ e_3 : \tau}\ \textsc{Case}$$

$$\frac{\begin{array}{c} A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad A; \Gamma_1 \vdash e_1 : \langle K^{\circ}\ \rho, \overline{K_i^{\bullet}\ \tau_i} \rangle \\ A; x : \rho, \Gamma_2 \vdash e_2 : \tau \end{array}}{A; \Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ K\ x.\ e_2 : \tau}\ \textsc{Irref}$$

Fig. 7. Typing Rules for variants

Typing rules for all expressions dealing with variants are given in Figure 7.

Pattern matching on variants is accomplished in our core language with two primitive forms (**case** expressions). The first is for a *refutable* match (i.e. when the pattern in question is not statically known to match the value), and it includes a default alternative in case the match fails. The second is for *irrefutable* matches, and is only well-typed when the pattern match can be shown statically to succeed.

Typically, a long chain of patterns is desugared into a nested chain of refutable **case** expressions, with a final irrefutable match when the chain of patterns is exhaustive:

$$\begin{array}{l} \mathbf{case}\ x\ \mathbf{of} \\ \quad K_1\ a \to e_1 \\ \quad K_2\ b \to e_2 \\ \quad K_3\ c \to e_3 \end{array} \quad \text{becomes} \quad \begin{array}{l} \mathbf{case}\ x\ \mathbf{of} \\ \quad K_1\ a.\ e_1 \\ \quad x'.\ \mathbf{case}\ x'\ \mathbf{of} \\ \qquad K_2\ b.\ e_2 \\ \qquad x''.\ \mathbf{case}\ x''\ \mathbf{of} \\ \qquad\quad K_3\ c.\ e_3 \end{array}$$

$$\boxed{\tau \hookrightarrow \tau}$$

$$\langle K^{\circ}\ \tau, \overline{K_i^{\mathrm{u}}\ \rho_i}\rangle\ \mathbf{take}\ K \quad \hookrightarrow \quad \langle K^{\bullet}\ \tau, \overline{K_i^{\mathrm{u}}\ \rho_i}\rangle$$

$$\boxed{A \vdash C}$$

$$\cdots$$

$$\frac{\text{for each } i\text{: } A \vdash \tau_i \sqsubseteq \rho_i \qquad \text{for each } j\text{: } A \vdash \tau_j \sqsubseteq \rho_j}{A \vdash \langle \overline{K_i^{\bullet}\ \tau_i}, \overline{K_j^{\mathrm{u}}\ \tau_j}\rangle \sqsubseteq \langle \overline{K_i^{\circ}\ \rho_i}, \overline{K_j^{\mathrm{u}}\ \rho_j}\rangle}\ \textsc{VarSub}$$

$$\frac{\text{for each } i\text{: } A \vdash \tau_i\ \mathbf{Share}}{A \vdash \langle \overline{K_i^{\circ}\ \tau_i}, \overline{K_j^{\bullet}\ \rho_j}\rangle\ \mathbf{Share}}\ \textsc{VarShare} \qquad \frac{\text{for each } i\text{: } A \vdash \tau_i\ \mathbf{Drop}}{A \vdash \langle \overline{K_i^{\circ}\ \tau_i}, \overline{K_j^{\bullet}\ \rho_j}\rangle\ \mathbf{Drop}}\ \textsc{VarDrop}$$

Fig. 8. Constraint semantics for variants

types	τ, ρ	::=	$\cdots \mid \mathtt{A}\ \vec{\tau}\ s$	(*abstract types*)
		$\mid$	$a!$	(*observer types*)
		$\mid$	$\mathbf{bang}(\tau)$	(*observation operator*)
constraints	C	::=	$\cdots \mid \tau\ \mathbf{Escape}$	(*escape analysis*)
expressions	e	::=	$\cdots \mid \mathbf{let!}\ (\overline{y_i})\ x = e_1\ \mathbf{in}\ e_2$	(*observation*)
sigils	s	::=	ⓦ	(*writable*)
		$\mid$	ⓡ	(*read-only*)
		$\mid$	ⓤ	(*unboxed*)

Fig. 9. Syntax for abstract and observer types

3.2 Abstract and observer types

An *abstract type* is a type whose full definition must be provided in imported C code. Values of abstract type must be constructed (and, if necessary, destroyed) by imported C functions, and all operations on them must also be defined in C. Nevertheless, they must be explicitly declared when used in Cogent code. An abstract type declaration consists of a type name and a series of parameters, without any definition provided.

In our core type system, an abstract type is represented as $\mathtt{A}\ \vec{\tau}\ s$, where $\mathtt{A}$ is the type name, $\vec{\tau}$ is the list of type parameters, and s is a *sigil*, which determines which constraints are satisfied by the abstract type. There are three forms of sigil:

- *Read-only* sigils (ⓡ), indicating that the value is represented as a pointer that can be freely shared or dropped, as the value cannot be written to during the lifetime of this pointer.
- *Writable* sigils (ⓦ), indicating that the value is represented as a pointer, and must be linear, as the value may be destructively updated.

$$\boxed{\tau \hookrightarrow \tau}$$

$$\begin{array}{lcl}
\mathbf{bang}(\langle \overline{K_i^{\mathrm{u}}\ \rho_i} \rangle) & \hookrightarrow & \langle \overline{K_i^{\mathrm{u}}\ \mathbf{bang}(\rho_i)} \rangle \\
\mathbf{bang}(\tau \to \rho) & \hookrightarrow & \tau \to \rho \\
\mathbf{bang}(\mathtt{A}\ \vec{\tau_i}\ \text{Ⓦ}) & \hookrightarrow & \mathtt{A}\ \overline{\mathbf{bang}(\tau_i)}\ \text{Ⓡ} \\
\mathbf{bang}(\mathtt{A}\ \vec{\tau_i}\ \text{Ⓡ}) & \hookrightarrow & \mathtt{A}\ \overline{\mathbf{bang}(\tau_i)}\ \text{Ⓡ} \\
\mathbf{bang}(\mathtt{A}\ \vec{\tau_i}\ \text{Ⓤ}) & \hookrightarrow & \mathtt{A}\ \overline{\mathbf{bang}(\tau_i)}\ \text{Ⓤ} \\
\mathbf{bang}(a) & \hookrightarrow & a! \\
\mathbf{bang}(a!) & \hookrightarrow & a! \\
\mathbf{bang}(T) & \hookrightarrow & T
\end{array}$$

$$\boxed{A \vdash C}$$

$$\cdots$$

$$\frac{A \vdash C[\tau] \qquad \tau \hookrightarrow \rho}{A \vdash C[\rho]}\ \textsc{Norm}$$

$$\frac{s \neq \text{Ⓦ} \qquad \text{for each } i\text{: } A \vdash \tau_i\ \mathbf{Drop}}{A \vdash \mathtt{A}\ \vec{\tau_i}\ s\ \mathbf{Drop}}\ \textsc{AbsDrop} \qquad \frac{s \neq \text{Ⓦ} \qquad \text{for each } i\text{: } A \vdash \tau_i\ \mathbf{Share}}{A \vdash \mathtt{A}\ \vec{\tau_i}\ s\ \mathbf{Share}}\ \textsc{AbsShare}$$

$$\frac{s \neq \text{Ⓡ} \qquad \text{for each } i\text{: } A \vdash \tau_i\ \mathbf{Escape}}{A \vdash \mathtt{A}\ \vec{\tau_i}\ s\ \mathbf{Escape}}\ \textsc{AbsEsc} \qquad \frac{}{A \vdash \tau \to \rho\ \mathbf{Escape}}\ \textsc{FunEsc} \qquad \frac{}{A \vdash T\ \mathbf{Escape}}\ \textsc{PrimEsc}$$

$$\frac{\text{for each } i\text{: } A \vdash \rho_i\ \mathbf{Escape}}{A \vdash \langle \overline{K_i^{\mathrm{u}}\ \rho_i} \rangle\ \mathbf{Escape}}\ \textsc{SumEsc} \qquad \frac{}{A \vdash a!\ \mathbf{Drop}}\ \textsc{ObsDrop} \qquad \frac{}{A \vdash a!\ \mathbf{Share}}\ \textsc{ObsShare}$$

Fig. 10. Constraint semantics for abstract and observer types

- *Unboxed* sigils (Ⓤ), indicating that the value is not represented as a pointer at all,[2] and may be freely shared or dropped.

Note that, according to the constraint semantics given in Figure 10, an abstract type can only satisfy the **Share** and **Drop** constraints *if the sigil is not* Ⓦritable. Thus these writable abstract types are the first of the types we have introduced to be *linear*.

3.2.1 Observation and escape analysis

In our core language, the expression-level ! construct that allows linear values to be temporarily shared within a limited scope is desugared into the syntactic form **let**! $(\overline{y_i})\ x = e_1$ **in** e_2. This form is similar to a **let** expression, except that the variables $\overline{y_i : \rho_i}$ are temporarily retyped during the typing of e_1 as $\overline{y_i : \mathbf{bang}(\rho_i)}$, where $\mathbf{bang}(\cdot)$ is a type operator that changes all linear Ⓦritable sigils in a type to shareable Ⓡead-only ones. The

[2] Or, as a pointer to which no Ⓦritable pointer will ever exist.

$$\boxed{A; \Gamma \vdash e : \tau}$$

$$\ldots$$

$$\frac{A; \overline{y_i : \mathbf{bang}(\rho_i)}, \Gamma_1 \vdash e_1 : \tau' \qquad A \vdash \tau' \ \mathbf{Escape} \qquad A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad A; x : \tau', \overline{y_i : \rho_i}, \Gamma_2 \vdash e_2 : \tau}{A; \overline{y_i : \rho_i}, \Gamma \vdash \mathbf{let!}\ (\overline{y_i})\ x = e_1\ \mathbf{in}\ e_2 : \tau}\ \textsc{Let!}$$

Fig. 11. Typing rules for **let!**

typing rules for **let**! expressions are given in Figure 11. We provide *normalisation rules* for this type operator, starting in Figure 10: Written $\tau \hookrightarrow \tau'$, these rules describe how types may be rewritten to eliminate the type operators. In the compiler implementation, this normalisation is also used to handle features such as type synonyms.

To handle polymorphic type variables, which may be instantiated to types containing Ⓦritable sigils, we introduce another kind of polymorphic type variable, written $a!$, which becomes $\mathbf{bang}(\tau)$ under the application of the substitution $[\tau/a]$. Furthermore, for a type variable a, we define $\mathbf{bang}(a) \mapsto a!$. In this way, we can guarantee that $\mathbf{bang}(\tau)$ is always non-linear *regardless* of τ, as no Ⓦritable sigils will remain in the type. This technique is originally due to Odersky (1992).

Theorem 3.1 (`bang_non_linear`)**.** *For all types τ and assumptions A, if no unification variables occur in τ we have $A \vdash \mathbf{bang}(\tau)$* **Share** *and $A \vdash \mathbf{bang}(\tau)$* **Drop**.

Proof By structural induction on τ. ■

The dynamic *uniqueness property*, introduced informally in Section 2 and formally in Section 4, can be stated as:

> *No Ⓦritable pointer can be aliased by any other pointer. A ⓡead-only pointer may be aliased by any number of other ⓡead-only pointers.*

We prove in Section 4 that this property is maintained as a dynamic invariant as a consequence of the static semantics (making Ⓦritable pointers linear). A naïve implementation of the **let**! feature, however, can easily lead to this invariant being violated:

$$\mathbf{let!}\ (x)\ y = x\ \mathbf{in}\ (x, y)$$

In this example, the freely shareable ⓡead-only pointer x is bound to y, and thus aliases the Ⓦritable pointer x in the returned tuple. Therefore, to maintain the invariant, we must prevent the ⓡead-only pointers available in a **let**! from escaping their scope. The first formulation to include a **let**! feature is that of Wadler (1990), which imposes a type-based safety check on the type of the binding in a **let**!, essentially requiring that the type of the binding and the type of the temporarily non-linear variables have no components in common. We adopt a slightly different approach which originated from Odersky (1992), although it differs in presentation.

We introduce a new type constraint, written τ **Escape**, that states that τ can be safely bound by a **let**! expression. Crucially, it does *not* hold if any ⓡ sigils appear in the type. This means that read-only pointers cannot be bound in a **let**! expression, but writable, linear

$$
\begin{array}{llll}
\text{types} & \tau, \rho & ::= & \cdots \mid \{\overline{\mathtt{f}_i^{\mu} : \tau_i}\}\ s \qquad (\textit{record types}) \\
\text{expressions} & e & ::= & \cdots \mid \sharp\{\overline{\mathtt{f}_i = e_i}\} \qquad (\textit{unboxed allocation}) \\
 & & \mid & \textbf{take}\ x\ \{\mathtt{f} = y\} = e_1\ \textbf{in}\ e_2 \qquad (\textit{record patterns}) \\
 & & \mid & \textbf{put}\ e_1.\mathtt{f} = e_2 \qquad (\textit{record updates}) \\
 & & \mid & e_1.\mathtt{f} \qquad (\textit{record field read}) \\
\text{field names} & \mathtt{f} & &
\end{array}
$$

Fig. 12. Syntax for records

pointers and unboxed values can be bound without a type error. Figure 10 contains full definitions for this **Escape** judgement.

Both methods, our own and that of Wadler (1990), are sound, type-based over-approximations of *escape analysis*. Fruitful avenues for further research may be to incorporate more sophisticated analysis techniques to improve the flexibility and predictability of this feature. One possible method may be the use of region types (Tofte & Talpin, 1994) to track the provenance of pointer variables more precisely, which Rust uses to great effect in its similar type system.

3.3 Record types

Lastly, we must formalise the typing rules for *record types* or products. The syntax for record types is given in Figure 12. A record, written $\{\overline{\mathtt{f}_i^{\mu} : \tau_i}\}\ s$, consists of one or more *fields* ($\mathtt{f}_i$). Due to the additional properties maintained by our type system, record types in Cogent are structured slightly differently to more traditional programming languages. Suppose we wish to access a particular field `f` of a record r. An expression like $r.\mathtt{f}$ would be problematic, as this *uses* the variable r, so any non-`f` fields in r would need to satisfy **Drop**. If this were the only way to access the fields of a record, any record with two linear fields would be unusable.

If instead we imagine a pattern-matching expression that reintroduces the record as a new variable name, like so:

$$\textbf{let}\ r'\ \{\mathtt{f} = x\} = r\ \textbf{in}\ \cdots$$

Then this violates the uniqueness property that our type system purports to maintain, as the field `f` could be accessed from the resultant record r' as well as by the new variable x. To solve this problem, any field that is extracted via pattern matching is marked as *unavailable* in the type of the resultant record, by changing the usage tag associated with each of the extracted fields to be $\bullet$. This pattern matching is desugared into one or more **take** expressions, written $\textbf{take}\ x\ \{\mathtt{f} = y\} = e_1\ \textbf{in}\ e_2$. Note that the typing rules in Figure 13 requires that the field being taken is available (tagged with $\circ$), and ensures that the field is no longer available in e_2 (tagged with $\bullet$). Conversely, record assignment expressions are desugared into **put** expressions, of the form $\textbf{put}\ e_1.\mathtt{f} = e_2$. The typing rule for this expression ensures that the field being overwritten has already been extracted ($\bullet$), and makes the field available again in the resultant record ($\circ$).

$$\boxed{A;\Gamma \vdash e : \tau}$$

$$\ldots$$

$$\frac{A;\Gamma \vdash e : \{\overline{\mathtt{f}_i^{\bullet} : \tau_i}, \mathtt{f}^{\circ} : \tau\}\ s}{A;\Gamma \vdash e.\mathtt{f} : \tau}\ \textsc{Member} \qquad \frac{A;\Gamma \vdash \overline{e_i : \tau_i}}{A;\Gamma \vdash \sharp\{\overline{\mathtt{f}_i = e_i}\} : \{\overline{\mathtt{f}_i^{\circ} : \tau_i}\}\ \textcircled{u}}\ \textsc{Struct}$$

$$\frac{\begin{array}{c} A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ A;\Gamma_1 \vdash e_1 : \{\overline{\mathtt{f}_i^{u} : \tau_i}, \mathtt{f}^{\bullet} : \tau\}\ s \\ s \neq \textcircled{r} \qquad A;\Gamma_2 \vdash e_2 : \tau \end{array}}{A;\Gamma \vdash \textbf{put}\ e_1.\mathtt{f} = e_2 : \{\overline{\mathtt{f}_i^{u} : \tau_i}, \mathtt{f}^{\circ} : \tau\}\ s}\ \textsc{Put}$$

$$\frac{\begin{array}{c} A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ A;\Gamma_1 \vdash e_1 : \{\overline{\mathtt{f}_i^{u} : \tau_i}, \mathtt{f}^{\circ} : \rho\}\ s \qquad s \neq \textcircled{r} \\ A;x : \{\overline{\mathtt{f}_i^{u} : \tau_i}, \mathtt{f}^{\bullet} : \rho\}\ s, y : \rho, \Gamma_2 \vdash e_2 : \tau \end{array}}{A;\Gamma \vdash \textbf{take}\ x\ \{\mathtt{f} = y\} = e_1\ \textbf{in}\ e_2 : \tau}\ \textsc{Take}$$

$$\boxed{A;\Gamma \vdash \overline{e_i : \tau_i}}$$

$$\frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \varepsilon}{A;\Gamma \vdash \varepsilon}\ \textsc{Empty} \qquad \frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad A;\Gamma \vdash e : \tau \qquad A;\Gamma \vdash \overline{e_i : \tau_i}}{A;\Gamma \vdash e : \tau, \overline{e_i : \tau_i}}\ \textsc{Cons}$$

Fig. 13. Typing rules for records

$$\boxed{\tau \hookrightarrow \tau}$$

$$\ldots$$

$$\begin{array}{lcl} \textbf{bang}(\{\overline{\mathtt{f}_i^{u} : \tau_i}\}\ \textcircled{w}) & \hookrightarrow & \{\overline{\mathtt{f}_i^{u} : \textbf{bang}(\tau_i)}\}\ \textcircled{r} \\ \textbf{bang}(\{\overline{\mathtt{f}_i^{u} : \tau_i}\}\ \textcircled{r}) & \hookrightarrow & \{\overline{\mathtt{f}_i^{u} : \textbf{bang}(\tau_i)}\}\ \textcircled{r} \\ \textbf{bang}(\{\overline{\mathtt{f}_i^{u} : \tau_i}\}\ \textcircled{u}) & \hookrightarrow & \{\overline{\mathtt{f}_i^{u} : \textbf{bang}(\tau_i)}\}\ \textcircled{u} \end{array}$$

$$\boxed{A \vdash C}$$

$$\ldots$$

$$\frac{\text{for each } i\text{: } A \vdash \tau_i\ \textbf{Drop} \qquad \text{for each } i\text{: } A \vdash \tau_i \sqsubseteq \rho_i \qquad \text{for each } j\text{: } A \vdash \tau_j \sqsubseteq \rho_j}{A \vdash \{\overline{\mathtt{f}_i^{\circ} : \tau_i}, \overline{\mathtt{f}_j^{u} : \tau_j}\}\ s \sqsubseteq \{\overline{\mathtt{f}_i^{\bullet} : \rho_i}, \overline{\mathtt{f}_j^{u} : \rho_j}\}\ s}\ \textsc{RecSub}$$

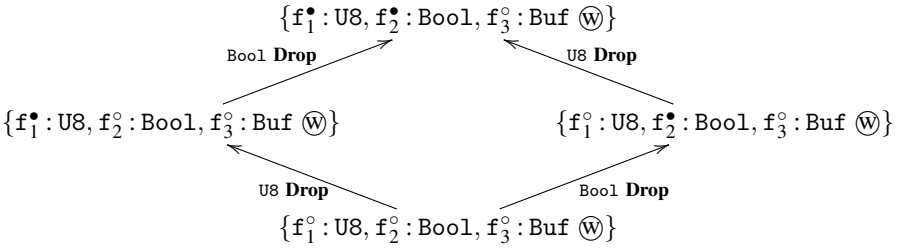
$$\frac{s \neq \textcircled{w} \qquad \text{for each } i\text{: } A \vdash \tau_i\ \textbf{Share}}{A \vdash \{\overline{\mathtt{f}_i^{\circ} : \tau_i}, \overline{\mathtt{f}_j^{\bullet} : \tau_j}\}\ s\ \textbf{Share}}\ \textsc{RecShare} \qquad \frac{s \neq \textcircled{w} \qquad \text{for each } i\text{: } A \vdash \tau_i\ \textbf{Drop}}{A \vdash \{\overline{\mathtt{f}_i^{\circ} : \tau_i}, \overline{\mathtt{f}_j^{\bullet} : \tau_j}\}\ s\ \textbf{Drop}}\ \textsc{RecDrop}$$

$$\frac{s \neq \textcircled{r} \qquad \text{for each } i\text{: } A \vdash \tau_i\ \textbf{Escape}}{A \vdash \{\overline{\mathtt{f}_i^{\circ} : \tau_i}, \overline{\mathtt{f}_j^{\bullet} : \tau_j}\}\ s\ \textbf{Escape}}\ \textsc{RecEscape}$$

Fig. 14. Constraint semantics for records

Like abstract types, records may be stored on the heap and passed around by reference, in which case we must track uniqueness of each pointer to the record. For this reason, record types are tagged with a *sigil s*, which, much as with abstract types, allows records to be declared *read-only* (r), where they are stored on the heap and passed by a read-only, shareable pointer; *writable* (w), where they are stored on the heap and passed by a writable, linear pointer; or *unboxed* (u), where they are typically represented on the stack or as a flat structure. Tuples are desugared in our core language as unboxed record types with two fields. As can be seen in Figure 14, the **bang** operator interacts with these sigils in much the same way as with abstract types. The **Share**, **Drop**, and **Escape** constraints place the same constraints on the sigils as with abstract types, with the added requirement that the type of each available field also satisfies the constraint in question.

If we wish to **put** a new value into a field that is marked as available (○) in the original record, the typing rules seem to indicate that we would have to **take** the field out, discard it, and **put** in a new value. To avoid having to explicitly **take** out every field we wish to discard, we allow fields that satisfy **Drop** to be automatically discarded from a record via subtyping — almost a dual of the subtyping relation used for variants:



3.4 Polymorphism

The polymorphism in Cogent is carefully restricted. As polymorphism is only permitted on the top level functions in prenex position, we can statically determine all instantiations that are needed for a polymorphic function, and can generate specialised functions at compile time. Because all polymorphism is top-level, our typing rules can simply treat type variables as concrete types. The assumptions A for the constraint semantics and typing rules indicate which uniqueness constraints (of **Share**, **Drop**, and **Escape**) are satisfied by these type variables.

We prove that instantiating polymorphic type variables does not make well-typed terms ill-typed, nor does it make satisfiable type constraints unsatisfiable. These theorems are useful for showing type preservation in Section 4, and for the correctness of the monomorphisation phase of our refinement framework described in Section 5.

Theorem 3.2 (Instantiating Type Variables in Constraints)**.**

Given a constraint that holds under assumptions A,		$A \vdash C$
and a substitution to type variables that satisfies A,	$\wedge$	$\forall C_i \in A.\ \varepsilon \vdash \sigma(C_i)$
then the substituted constraint also holds.	$\Rightarrow$	$\varepsilon \vdash \sigma(C)$

Proof Straightforward rule induction on the assumption $A \vdash C$. Wherever the rule ASM is used, we refer to the second assumption to justify the validity of the substituted constraint. Whenever observer type variables ($a!$) are substituted, we make use of Theorem 3.1. ■

Theorem 3.3 (Instantiating Type Variables — `instantiation`)**.**

Given a term typed under requirements A,		$A; \Gamma \vdash e : \tau$
and a substitution to type variables that satisfies A,	$\wedge$	$\forall C_i \in A.\ \varepsilon \vdash \sigma(C_i)$
then the substituted term is also well-typed.	$\Rightarrow$	$\varepsilon; \sigma(\Gamma) \vdash \sigma(e) : \sigma(\tau)$

Proof Rule induction, using Theorem 3.2. ■

3.5 Subtyping

Subtyping could be viewed as a partial order on types, as we have seen in our typing rules, or as a partial lattice with greatest lowest bound (glb) $\sqcap$ and least upper bound (lub) $\sqcup$ operations. These operations are partial as, for instance, $(\tau_1, \tau_2) \sqcap (\rho_1 \to \rho_2)$ is not defined.

Each of the two views of subtyping is convenient in different contexts. The order view is more convenient for proofs, as it is a simple inductive relation on two types, whereas the glb and lub operations are mutually inductive. The lattice view is more convenient for type inference, as it provides a direct way to find common supertypes or subtypes (see O'Connor (2019)).

To bridge the gap between these two interpretations, we have formalised both views of subtyping in Isabelle/HOL, and proven the standard equivalences:

Theorem 3.4 (Equivalence of two subtyping notions)**.**

The notions that τ is a subtype of ρ;		$A \vdash \tau \sqsubseteq \rho$
that τ is the glb of ρ and τ; and	$\Leftrightarrow$	$A \vdash \tau \sqcap \rho = \tau$
that ρ is the lub of ρ and τ are all equivalent.	$\Leftrightarrow$	$A \vdash \tau \sqcup \rho = \rho$

4 Dynamic semantics

It has long been understood that linear and uniqueness type systems can be used to provide a purely functional interface to mutable state and side-effects (Wadler, 1990). This intuition follows from the *uniqueness property* mentioned in Section 2, that each live mutable object is referenced by exactly one variable at a time: If a function has a reference to a mutable object, no other references must exist. Therefore, destructive update is indistinguishable from the traditional purely functional copy-update idiom, as no aliases exist to observe the change.

Despite this result, many languages with uniqueness types, such as Rust (Rust, 2014) or Vault (DeLine & Fähndrich, 2001), only make use of such type systems to reduce or eliminate the need for run-time memory management, and to facilitate informal reasoning about the provenance of pointers. The functional language Clean (Barendsen & Smetsers, 1993) makes use of uniqueness types to abstract over effects, but it still has need for a

$$\boxed{A \vdash \tau \sqcap \rho = \gamma}$$

$$\frac{\tau \in \{\texttt{Bool}, \texttt{U8}, \texttt{U16}, \texttt{U32}, \texttt{U64}\}}{A \vdash \tau \sqcap \tau = \tau}\ \textsc{Prim-}\sqcap \qquad \frac{A \vdash \tau_1 \sqcup \rho_1 = \gamma_1 \qquad A \vdash \tau_2 \sqcap \rho_2 = \gamma_2}{A \vdash (\tau_1 \to \tau_2) \sqcap (\rho_1 \to \rho_2) = (\gamma_1 \to \gamma_2)}\ \textsc{Function-}\sqcap$$

$$\frac{A \vdash (\tau_1 \sqcap \rho_1) = \gamma_1 \qquad A \vdash (\tau_2 \sqcap \rho_2) = \gamma_2}{A \vdash (\tau_1 \times \tau_2) \sqcap (\rho_1 \times \rho_2) = (\gamma_1 \times \gamma_2)}\ \textsc{Tuple-}\sqcap \qquad \frac{}{A \vdash \mathtt{A}\ \overline{\tau_i} \sqcap \mathtt{A}\ \overline{\tau_i} = \mathtt{A}\ \overline{\tau_i}}\ \textsc{Abstract-}\sqcap$$

$$\frac{\text{for each } i\text{: } A \vdash \tau_i \sqcap \rho_i = \gamma_i \quad \text{for each } j\text{: } A \vdash \tau_j \sqcap \rho_j = \gamma_j \quad \text{for each } k\text{: } A \vdash \tau_k \sqcap \rho_k = \gamma_k}{A \vdash \langle \overline{K_i^\bullet\ \tau_i}, \overline{K_j^\circ\ \tau_j}, \overline{K_k^{\mathrm{u}}\ \tau_k} \rangle \sqcap \langle \overline{K_i^\circ\ \rho_i}, \overline{K_j^\bullet\ \tau_j}, \overline{K_k^{\mathrm{u}}\ \rho_k} \rangle = \langle \overline{K_i^\bullet\ \gamma_i}, \overline{K_j^\bullet\ \gamma_j}, \overline{K_k^{\mathrm{u}}\ \gamma_k} \rangle}\ \textsc{Variant-}\sqcap$$

$$\frac{\text{for each } i\text{: } A \vdash \tau_i \sqcap \rho_i = \gamma_i \quad \text{for each } j\text{: } A \vdash \tau_j \sqcap \rho_j = \gamma_j \quad \text{for each } k\text{: } A \vdash \tau_k \sqcap \rho_k = \gamma_k}{A \vdash \{\overline{\mathtt{f}_i^\circ : \tau_i}, \overline{\mathtt{f}_j^\bullet : \tau_j}, \overline{\mathtt{f}_k^{\mathrm{u}} : \tau_k}\}\ s \sqcap \{\overline{\mathtt{f}_i^\bullet : \rho_i}, \overline{\mathtt{f}_j^\circ : \tau_j}, \overline{\mathtt{f}_k^{\mathrm{u}} : \rho_k}\}\ s = \{\overline{\mathtt{f}_i^\circ : \rho_i}, \overline{\mathtt{f}_j^\circ : \tau_j}, \overline{\mathtt{f}_k^{\mathrm{u}} : \rho_k}\}\ s}\ \textsc{Record-}\sqcap$$

$$\boxed{A \vdash \tau \sqcup \rho = \gamma}$$

$$\frac{\tau \in \{\texttt{Bool}, \texttt{U8}, \texttt{U16}, \texttt{U32}, \texttt{U64}\}}{A \vdash \tau \sqcup \tau = \tau}\ \textsc{Prim-}\sqcup \qquad \frac{A \vdash \tau_1 \sqcap \rho_1 = \gamma_1 \qquad A \vdash \tau_2 \sqcup \rho_2 = \gamma_2}{A \vdash (\tau_1 \to \tau_2) \sqcup (\rho_1 \to \rho_2) = (\gamma_1 \to \gamma_2)}\ \textsc{Function-}\sqcup$$

$$\frac{A \vdash (\tau_1 \sqcup \rho_1) = \gamma_1 \qquad A \vdash (\tau_2 \sqcup \rho_2) = \gamma_2}{A \vdash (\tau_1 \times \tau_2) \sqcup (\rho_1 \times \rho_2) = (\gamma_1 \times \gamma_2)}\ \textsc{Tuple-}\sqcup \qquad \frac{}{A \vdash \mathtt{A}\ \overline{\tau_i} \sqcup \mathtt{A}\ \overline{\tau_i} = \mathtt{A}\ \overline{\tau_i}}\ \textsc{Abstract-}\sqcup$$

$$\frac{\text{for each } i\text{: } A \vdash \tau_i \sqcup \rho_i = \gamma_i \quad \text{for each } j\text{: } A \vdash \tau_j \sqcup \rho_j = \gamma_j \quad \text{for each } k\text{: } A \vdash \tau_k \sqcup \rho_k = \gamma_k}{A \vdash \langle \overline{K_i^\bullet\ \tau_i}, \overline{K_j^\circ\ \tau_j}, \overline{K_k^{\mathrm{u}}\ \tau_k} \rangle \sqcup \langle \overline{K_i^\circ\ \rho_i}, \overline{K_j^\bullet\ \tau_j}, \overline{K_k^{\mathrm{u}}\ \rho_k} \rangle = \langle \overline{K_i^\circ\ \gamma_i}, \overline{K_j^\circ\ \gamma_j}, \overline{K_k^{\mathrm{u}}\ \gamma_k} \rangle}\ \textsc{Variant-}\sqcup$$

$$\frac{\begin{array}{c}\text{for each } i\text{: } A \vdash \tau_i \textbf{ Drop} \qquad \text{for each } i\text{: } A \vdash \tau_i \sqcup \rho_i = \gamma_i \\ \text{for each } j\text{: } A \vdash \rho_j \textbf{ Drop} \qquad \text{for each } j\text{: } A \vdash \tau_j \sqcup \rho_j = \gamma_j \qquad \text{for each } k\text{: } A \vdash \tau_k \sqcup \rho_k = \gamma_k\end{array}}{A \vdash \{\overline{\mathtt{f}_i^\circ : \tau_i}, \overline{\mathtt{f}_j^\bullet : \tau_j}, \overline{\mathtt{f}_k^{\mathrm{u}} : \tau_k}\}\ s \sqcup \{\overline{\mathtt{f}_i^\bullet : \rho_i}, \overline{\mathtt{f}_j^\circ : \tau_j}, \overline{\mathtt{f}_k^{\mathrm{u}} : \rho_k}\}\ s = \{\overline{\mathtt{f}_i^\bullet : \rho_i}, \overline{\mathtt{f}_j^\bullet : \tau_j}, \overline{\mathtt{f}_k^{\mathrm{u}} : \rho_k}\}\ s}\ \textsc{Record-}\sqcup$$

Fig. 15. The rules of the subtyping relation viewed as a lattice

garbage collector, and it does not prove, on paper nor in a machine-checked proof script, the semantic coincidence that results from the type system.

The proof of this semantic coincidence is more than just a curiosity for Cogent, as it forms a key part of the compiler certificate used to show refinement from an Isabelle/HOL shallow embedding of the Cogent code all the way to an efficient C implementation, the details of which are discussed in Section 5.

Hofmann (2000) first formalised this intuition by providing both a set-theoretic denotational semantics and a compilation to C for a functional language, and demonstrating that these two semantics coincide in a pen-and-paper proof. The language in question, however, was extremely minimal, and did not involve heap-allocated objects or pointers, merely mutable stack-allocated integers.

In this respect, the machine-checked proof of semantic coincidence for Cogent represents a significant advancement in the state of the art, as Cogent is a higher-order language with full support for compound types and heap-allocated objects, necessitating a more intricate formulation of the uniqueness property, outlined in Section 4.2. Cogent also integrates with C code called via the foreign function interface, which necessitates a formal treatment of the boundary between these languages. Specifically, we must characterise the obligations the C code must meet in order to maintain our uniqueness invariant (see Section 4.2.4).

Each of the theorems presented in this section are formalised and machine-checked in Isabelle/HOL, as they form a vital part of our overall refinement certificate. Each theorem includes the corresponding name (written in `typewriter typeface`) of the equivalent theorem in Isabelle/HOL formalisation of Cogent (*n.d.*).

4.1 A Tale of Two Semantics

As previously mentioned, we assign two dynamic semantics to Cogent terms. The first is the functional *value semantics*, which is suitable for equational reasoning, and can be easily connected to an Isabelle/HOL shallow embedding. The second, the *update semantics*, is more imperative in flavour, where values may take the form of *pointers* to a mutable *store*.

Figure 16 describes the syntax of values and their environments for our two dynamic semantics. Both semantics definitions are parameterised by a set of *abstract values*, a_{V} and a_{U} respectively, which denote values of abstract types defined in C. They are also parameterised by functional abstractions of any C foreign functions used in the Cogent code, manually written and supplied by the programmer. For an abstract function f, the value semantics abstraction $[\![f]\!]_{\mathrm{V}}$ must be a pure function, and the update semantics abstraction $[\![f]\!]_{\mathrm{U}}$ must be a refinement of $[\![f]\!]_{\mathrm{V}}$ which respects the invariants of our type system. The exact proof obligations placed on these functions are outlined in Section 4.2.4. The Cogent refinement framework described in Section 5 is additionally parameterised by refinement proofs between these purely functional abstractions and their C implementation. If full end-to-end verification of all components of the system is desired, the user must additionally prove this refinement, and compose this proof with our framework.

4.1.1 Value semantics

The rules for the value semantics are given in Figure 17. Specified as a big-step evaluation relation $V \vdash e \Downarrow_{\mathrm{V}} v$, these rules describe the evaluation of an expression e to a single result value v with the environment V containing the values of all variables in scope. In many ways, these semantics are entirely typical of a λ-calculus or other purely functional language: all values are self-contained, there is no notion of sharing or references. Therefore, other than the values of all available variables, there is no need for any context to evaluate an expression. The rules can be viewed as an evaluation algorithm, as they are entirely syntax-directed — exactly one rule specifies the evaluation for each form of expression. Syntactic constructs which only exist to aid the uniqueness type system have no impact on the dynamic semantics. For example, the **let!** construct behaves identically to **let**.

Just as in Section 3, where we assumed the existence of a global type environment for top-level definitions called *typeOf*$(\cdot)$, we include a global definition environment *defnOf*$(\cdot)$ that, given a function name, provides either:

1. a transparent definition, written $\Lambda\vec{a}.\ \lambda x.\ e$, which denotes a Cogent function returning e, parametric for type variables $\vec{a}$ and a single value argument x; or
2. a black box (■), which indicates that the function's definition is *abstract*, i.e. provided externally in C.

The rule VTAPP describes how non-abstract functions are evaluated to function values. 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 parameterised by a value, evaluated when the function is applied, thereby supplying the argument value. There is no need to define closures or environment capture, as top-level functions cannot capture local bindings. Abstract function values, written $\langle\!\langle \mathbf{abs}.\ f \mid \vec{\tau} \rangle\!\rangle$, are passed indirectly, as a pair of the function name and a list of the

Value Semantics

value semantics values	v	$::=$	ℓ	(*literals*)
		$\mid$	$\langle\!\langle \lambda x.\ e \rangle\!\rangle$	(*function values*)
		$\mid$	$\langle\!\langle \mathbf{abs}.\ f \mid \vec{\tau} \rangle\!\rangle$	(*abstract functions*)
		$\mid$	$K\ v$	(*variant values*)
		$\mid$	$\{\overline{\mathtt{f} \mapsto v}\}$	(*records*)
		$\mid$	a_{V}	(*abstract values*)
environments	V	$::=$	$\overline{x \mapsto v}$	
abstract values	a_{V}			
abstract function semantics	$[\![\cdot]\!]_{\mathrm{V}}$	$:$	$f \to v \to v$	

Update Semantics

update semantics values	u	$::=$	ℓ	(*literals*)
		$\mid$	$\langle\!\langle \lambda x.\ e \rangle\!\rangle$	(*function values*)
		$\mid$	$\langle\!\langle \mathbf{abs}.\ f \mid \vec{\tau} \rangle\!\rangle$	(*abstract functions*)
		$\mid$	$K\ u$	(*variant values*)
		$\mid$	$\{\overline{\mathtt{f} \mapsto u}\}$	(*records*)
		$\mid$	a_{U}	(*abstract values*)
		$\mid$	p	(*pointers*)
environments	U	$::=$	$\overline{x \mapsto u}$	
abstract values	a_{U}			
pointers	p			
sets of pointers	r, w			
mutable stores	μ	$:$	$p \rightharpoonup u$	
abstract function semantics	$[\![\cdot]\!]_{\mathrm{U}}$	$:$	$f \to \mu \times u \to \mu \times u$	

primop semantics	$[\![\wr]\!]$	$:$	$v \times v \to v$	
function defn. env.	$defnOf(\cdot)$	$:$	$f \to D$	
function defn.	D	$::=$	$\blacksquare$	(*abstract functions*)
		$\mid$	$\Lambda\vec{a}.\ \lambda x.\ e$	(*function definitions*)

Fig. 16. Syntax for both dynamic semantics interpretations.

types used to instantiate type variables. When an abstract function value $\langle\!\langle \mathbf{abs}.\ f \mid \vec{\tau} \rangle\!\rangle$ is applied to an argument, the user-supplied purely functional abstraction of the C semantics $[\![f]\!]_{\mathrm{V}}$ is invoked—merely a mathematical function from the argument value to the output value.

4.1.2 Update semantics

Similarly to the value semantics, the update semantics is specified as a big-step evaluation relation, however unlike the value semantics, a *mutable store* is included as an input to

$$\boxed{V \vdash e \downarrow_{\mathrm{v}} v}$$

$$\frac{x \mapsto v \in V}{V \vdash x \downarrow_{\mathrm{v}} v}\ \textsc{vVar} \qquad \frac{}{V \vdash \ell \downarrow_{\mathrm{v}} \ell}\ \textsc{vLit} \qquad \frac{V \vdash e_1 \downarrow_{\mathrm{v}} v_1 \qquad V \vdash e_2 \downarrow_{\mathrm{v}} v_2}{V \vdash e_1 \wr e_2 \downarrow_{\mathrm{v}} v_1 \,[\![\wr]\!]\, v_2}\ \textsc{vOp}$$

$$\frac{V \vdash e \downarrow_{\mathrm{v}} \texttt{True} \qquad V \vdash e_1 \downarrow_{\mathrm{v}} v}{V \vdash \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \downarrow_{\mathrm{v}} v}\ \textsc{vIf-T} \qquad \frac{V \vdash e \downarrow_{\mathrm{v}} \texttt{False} \qquad V \vdash e_2 \downarrow_{\mathrm{v}} v}{V \vdash \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \downarrow_{\mathrm{v}} v}\ \textsc{vIf-F}$$

$$\frac{V \vdash e_1 \downarrow_{\mathrm{v}} \langle\!\langle \lambda x.\ e \rangle\!\rangle \qquad V \vdash e_2 \downarrow_{\mathrm{v}} v \qquad x \mapsto v \vdash e \downarrow_{\mathrm{v}} v'}{V \vdash e_1\ e_2 \downarrow_{\mathrm{v}} v'}\ \textsc{vApp} \qquad \frac{V \vdash e_1 \downarrow_{\mathrm{v}} \langle\!\langle \textbf{abs}.\ f \mid \overrightarrow{\tau} \rangle\!\rangle \qquad V \vdash e_2 \downarrow_{\mathrm{v}} v \qquad v' = [\![f]\!]_{\mathrm{v}}\ v}{V \vdash e_1\ e_2 \downarrow_{\mathrm{v}} v'}\ \textsc{vApp-A}$$

$$\frac{defnOf(f) = \Lambda \overrightarrow{a_i}.\ \lambda x.\ e}{V \vdash f\,[\overrightarrow{\tau_i}] \downarrow_{\mathrm{v}} \langle\!\langle \lambda x.\ e \left[{}^{\overrightarrow{\tau_i}}/_{\overrightarrow{a_i}} \right] \rangle\!\rangle}\ \textsc{vTApp} \qquad \frac{defnOf(f) = \blacksquare}{V \vdash f\,[\overrightarrow{\tau_i}] \downarrow_{\mathrm{v}} \langle\!\langle \textbf{abs}.\ f \mid \overrightarrow{\tau_i} \rangle\!\rangle}\ \textsc{vTApp-A}$$

$$\frac{V \vdash e_1 \downarrow_{\mathrm{v}} v' \qquad x \mapsto v', V \vdash e_2 \downarrow_{\mathrm{v}} v}{V \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \downarrow_{\mathrm{v}} v}\ \textsc{vLet} \qquad \frac{V \vdash e_1 \downarrow_{\mathrm{v}} v' \qquad x \mapsto v', V \vdash e_2 \downarrow_{\mathrm{v}} v}{V \vdash \textbf{let!} \ (\overline{y})\ x = e_1 \textbf{ in } e_2 \downarrow_{\mathrm{v}} v}\ \textsc{vLet!}$$

$$\frac{V \vdash e \downarrow_{\mathrm{v}} \{\mathtt{f} \mapsto v, \overline{\mathtt{f}_i \mapsto v_i}\}}{V \vdash e.\mathtt{f} \downarrow_{\mathrm{v}} v}\ \textsc{vMember} \qquad \frac{\text{for each } i, V \vdash e_i \downarrow_{\mathrm{v}} v_i}{V \vdash \sharp\{\overline{\mathtt{f}_i = e_i}\} \downarrow_{\mathrm{v}} \{\overline{\mathtt{f}_i \mapsto v_i}\}}\ \textsc{vStruct}$$

$$\frac{V \vdash e_1 \downarrow_{\mathrm{v}} \{\mathtt{f} \mapsto v, \overline{\mathtt{f}_i \mapsto v_i}\} \qquad x \mapsto \{\mathtt{f} \mapsto v, \overline{\mathtt{f}_i \mapsto v_i}\}, y \mapsto v, V \vdash e_2 \downarrow_{\mathrm{v}} v'}{V \vdash \textbf{take } x\ \{\mathtt{f} = y\} = e_1 \textbf{ in } e_2 \downarrow_{\mathrm{v}} v'}\ \textsc{vTake}$$

$$\frac{V \vdash e_1 \downarrow_{\mathrm{v}} \{\mathtt{f} \mapsto v, \overline{\mathtt{f}_i \mapsto v_i}\} \qquad V \vdash e_2 \downarrow_{\mathrm{v}} v'}{V \vdash \textbf{put } e_1.\mathtt{f} = e_2 \downarrow_{\mathrm{v}} \{\mathtt{f} \mapsto v', \overline{\mathtt{f}_i \mapsto v_i}\}}\ \textsc{vPut}$$

$$\frac{V \vdash e \downarrow_{\mathrm{v}} v}{V \vdash K\ e \downarrow_{\mathrm{v}} K\ v}\ \textsc{vCon} \qquad \frac{V \vdash e \downarrow_{\mathrm{v}} K\ v' \qquad x \mapsto v', V \vdash e' \downarrow_{\mathrm{v}} v}{V \vdash \textbf{case } e \textbf{ of } K\ x.\ e' \downarrow_{\mathrm{v}} v}\ \textsc{vIrref}$$

$$\frac{V \vdash e \downarrow_{\mathrm{v}} K\ v' \qquad x \mapsto v', V \vdash e_1 \downarrow_{\mathrm{v}} v}{V \vdash \textbf{case } e \textbf{ of } K\ x.\ e_1 \textbf{ else } y.\ e_2 \downarrow_{\mathrm{v}} v}\ \textsc{vCase-M}$$

$$\frac{V \vdash e \downarrow_{\mathrm{v}} K'\ v' \qquad K \neq K' \qquad y \mapsto K'\ v', V \vdash e_2 \downarrow_{\mathrm{v}} v}{V \vdash \textbf{case } e \textbf{ of } K\ x.\ e_1 \textbf{ else } y.\ e_2 \downarrow_{\mathrm{v}} v}\ \textsc{vCase-N}$$

Fig. 17. The value semantics evaluation rules.

$$\boxed{U \vdash e \mid \mu \overset{\text{U}}{\rightsquigarrow} u \mid \mu}$$

$$\frac{x \mapsto u \in U}{U \vdash x \mid \mu \overset{\text{U}}{\rightsquigarrow} u \mid \mu}\ \textsc{UVar} \qquad \frac{}{U \vdash \ell \mid \mu \overset{\text{U}}{\rightsquigarrow} \ell \mid \mu}\ \textsc{ULit} \qquad \frac{U \vdash e_1 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u_1 \mid \mu_2 \quad U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u_2 \mid \mu_3}{U \vdash e_1 \mathbin{\wr} e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u_1 \mathbin{[\![\wr]\!]} u_2 \mid \mu_3}\ \textsc{UOp}$$

$$\frac{U \vdash e \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} \texttt{True} \mid \mu_2 \quad U \vdash e_1 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}{U \vdash \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}\ \textsc{UIf-T} \qquad \frac{U \vdash e \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} \texttt{False} \mid \mu_2 \quad U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}{U \vdash \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}\ \textsc{UIf-F}$$

$$\frac{U \vdash e_1 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} \langle\!\langle \lambda x.\ e \rangle\!\rangle \mid \mu_2 \quad U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3 \quad x \mapsto u \vdash e \mid \mu_3 \overset{\text{U}}{\rightsquigarrow} u' \mid \mu_4}{U \vdash e_1\ e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u' \mid \mu_4}\ \textsc{UApp} \qquad \frac{U \vdash e_1 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} \langle\!\langle \textbf{abs}.\ f \mid \overrightarrow{\tau} \rangle\!\rangle \mid \mu_2 \quad U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3 \quad (u', \mu_4) = [\![f]\!]_{\text{U}}\ (u, \mu_3)}{U \vdash e_1\ e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u' \mid \mu_4}\ \textsc{UApp-A}$$

$$\frac{\mathit{defnOf}(f) = \Lambda \overrightarrow{a_i}.\ \lambda x.\ e}{U \vdash f\,[\overrightarrow{\tau_i}] \mid \mu \overset{\text{U}}{\rightsquigarrow} \langle\!\langle \lambda x.\ e\ \left[\overrightarrow{\tau_i}/\overrightarrow{a_i}\right] \rangle\!\rangle \mid \mu}\ \textsc{UTApp} \qquad \frac{\mathit{defnOf}(f) = \blacksquare}{U \vdash f\,[\overrightarrow{\tau_i}] \mid \mu \overset{\text{U}}{\rightsquigarrow} \langle\!\langle \textbf{abs}.\ f \mid \overrightarrow{\tau_i} \rangle\!\rangle \mid \mu}\ \textsc{UTApp-A}$$

$$\frac{U \vdash e_1 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u' \mid \mu_2 \quad x \mapsto u', U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}{U \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}\ \textsc{ULet} \qquad \frac{U \vdash e_1 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u' \mid \mu_2 \quad x \mapsto u', U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}{U \vdash \textbf{let!}\ (\overline{y})\ x = e_1\ \textbf{in}\ e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}\ \textsc{ULet!}$$

$$\frac{U \vdash e_1 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} \{\texttt{f} \mapsto u, \overline{\texttt{f}_i \mapsto u_i}\} \mid \mu_2 \quad x \mapsto \{\texttt{f} \mapsto u, \overline{\texttt{f}_i \mapsto u_i}\}, y \mapsto u, U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u' \mid \mu_3}{U \vdash \textbf{take}\ x\ \{\texttt{f} = y\} = e_1\ \textbf{in}\ e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u' \mid \mu_3}\ \textsc{UTake}$$

$$\frac{U \vdash e_1 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} \{\texttt{f} \mapsto u, \overline{\texttt{f}_i \mapsto u_i}\} \mid \mu_2 \qquad U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u' \mid \mu_3}{U \vdash \textbf{put}\ e_1.\texttt{f} = e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} \{\texttt{f} \mapsto u', \overline{\texttt{f}_i \mapsto u_i}\} \mid \mu_3}\ \textsc{UPut}$$

$$\frac{U \vdash e \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} \{\texttt{f} \mapsto u, \overline{\texttt{f}_i \mapsto u_i}\} \mid \mu_2}{U \vdash e.\texttt{f} \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_2}\ \textsc{UMem} \qquad \frac{U \vdash \overline{e_i} \mid \mu_1 \overset{\text{U}}{\rightsquigarrow}^{\star} \overline{u_i} \mid \mu_2}{U \vdash \sharp\{\overline{\texttt{f}_i = e_i}\} \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} \{\overline{\texttt{f}_i \mapsto u_i}\} \mid \mu_2}\ \textsc{UStruct}$$

$$\frac{U \vdash e \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_2}{U \vdash K\ e \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} K\ u \mid \mu_2}\ \textsc{UCon} \qquad \frac{U \vdash e \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} K\ u' \mid \mu_2 \quad x \mapsto u', U \vdash e' \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}{U \vdash \textbf{case}\ e\ \textbf{of}\ K\ x.\ e' \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_2}\ \textsc{UIrref}$$

$$\frac{U \vdash e \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} K\ u' \mid \mu_2 \qquad x \mapsto u', U \vdash e_1 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}{U \vdash \textbf{case}\ e\ \textbf{of}\ K\ x.\ e_1\ \textbf{else}\ y.\ e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}\ \textsc{UCase-M}$$

$$\frac{U \vdash e \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} K'\ u' \mid \mu_2 \qquad K \neq K' \quad y \mapsto K'\ u', U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}{U \vdash \textbf{case}\ e\ \textbf{of}\ K\ x.\ e_1\ \textbf{else}\ y.\ e_2 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u \mid \mu_3}\ \textsc{UCase-N}$$

(Continued in Figure 19)

$$\boxed{U \vdash \overline{e} \mid \mu \overset{\text{U}}{\rightsquigarrow}^{\star} \overline{u} \mid \mu}$$

$$\frac{}{U \vdash \varepsilon \mid \mu \overset{\text{U}}{\rightsquigarrow}^{\star} \varepsilon \mid \mu}\ \textsc{UNil} \qquad \frac{U \vdash e_0 \mid \mu_1 \overset{\text{U}}{\rightsquigarrow} u_0 \mid \mu_2 \qquad U \vdash \overline{e_i} \mid \mu_2 \overset{\text{U}}{\rightsquigarrow}^{\star} \overline{u_i} \mid \mu_3}{U \vdash e_0\ \overline{e_i} \mid \mu_1 \overset{\text{U}}{\rightsquigarrow}^{\star} u_0\ \overline{u_i} \mid \mu_3}\ \textsc{UCons}$$

Fig. 18. The straightforward update semantics evaluation rules.

$$\boxed{U \vdash e \mid \mu \overset{\text{U}}{\Downarrow} u \mid \mu}$$

$$\frac{U \vdash e_1 \mid \mu_1 \overset{\text{U}}{\Downarrow} p \mid \mu_2 \qquad \mu_2(p) = \{\mathtt{f} \mapsto u, \overline{\mathtt{f}_i \mapsto u_i}\} \qquad x \mapsto p, y \mapsto u, U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\Downarrow} u' \mid \mu_3}{U \vdash \mathbf{take}\ x\ \{\mathtt{f} = y\} = e_1\ \mathbf{in}\ e_2 \mid \mu_1 \overset{\text{U}}{\Downarrow} u' \mid \mu_3}\ \textsc{UTake-B}$$

$$\frac{U \vdash e_1 \mid \mu_1 \overset{\text{U}}{\Downarrow} p \mid \mu_2 \qquad \mu_2(p) = \{\mathtt{f} \mapsto u, \overline{\mathtt{f}_i \mapsto u_i}\} \qquad U \vdash e_2 \mid \mu_2 \overset{\text{U}}{\Downarrow} u' \mid \mu_3}{U \vdash \mathbf{put}\ e_1.\mathtt{f} = e_2 \mid \mu_1 \overset{\text{U}}{\Downarrow} p \mid \mu_3(p := \{\mathtt{f} \mapsto u', \overline{\mathtt{f}_i \mapsto u_i}\})}\ \textsc{UPut-B}$$

$$\frac{U \vdash e \mid \mu_1 \overset{\text{U}}{\Downarrow} p \mid \mu_2 \qquad \mu_2(p) = \{\mathtt{f} \mapsto u, \overline{\mathtt{f}_i \mapsto u_i}\}}{U \vdash e.\mathtt{f} \mid \mu_1 \overset{\text{U}}{\Downarrow} u \mid \mu_2}\ \textsc{UMem-B}$$

Fig. 19. The update semantics evaluation rules concerning pointers.

and output of an expression's evaluation, and values may be represented as pointers to locations in that mutable store. Written $U \vdash e \mid \mu \overset{\text{U}}{\Downarrow} u \mid \mu'$, this evaluation relation specifies that, given an environment U of values that may contain pointers into a mutable store μ, the evaluation of the expression e will result in the value u and a final store μ'. Figure 18 outlines the straightforward rules for this evaluation relation. The majority of these are very similar to their value-semantics equivalents, save that they thread the mutable store through the evaluation. The mutable store is specified as a partial mapping from pointers (written p) to values. The exact content of pointer values is left abstract: our semantics merely requires that they be enumerable and comparable. In Section 5, we instantiate p to a concrete set to prove refinement to the C implementation. Like in the value semantics, the semantics of foreign functions are provided externally, this time permitting modifications to the mutable store in addition to returning a value.

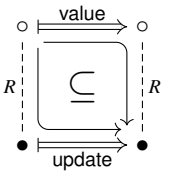
Unlike the value semantics, the update semantics distinguishes between *boxed* and *unboxed* records. For *unboxed* records, which are stack-allocated and passed by value, the rules for **take**, **put** etc. resemble their value-semantics counterparts. *Boxed* records, however, are represented as a pointer — the rule for **take** must consult the heap, and the rule for **put** *mutates* the heap, destructively updating the record. The rules that involve the mutable heap are specified in Figure 19.

4.2 Refinement and type preservation

To show that the update semantics refines the value semantics, the typical approach from data refinement (de Roever & Engelhardt, 1998) is to define a *refinement relation* R between values in the value semantics and states in the update semantics, and show that any update semantics evaluation has a corresponding R-preserving value semantics evaluation. When the semantics are viewed as binary relations from initial to final states (outputs), this requirement can be succinctly expressed as a commutative diagram. For example, with respect to an externally defined function f, we relate the user-provided value semantics $[\![f]\!]_{\text{V}}$ and update semantics $[\![f]\!]_{\text{U}}$ as follows:

$$R; [\![f]\!]_{\mathrm{U}} \subseteq [\![f]\!]_{\mathrm{V}}; R$$

(where ; is forward composition of relations)



Assuming that the relation holds initially, we can conclude from such a proof that any execution in our update semantics interpretation has a corresponding execution in our value semantics interpretation, and thus any functional correctness property we prove about all our value semantics executions applies also to our update semantics executions.

The relation R must relate value semantics values (v) to update semantics states ($u \times \mu$). A plausible definition would be as an abstraction function, which eliminates pointers from each update semantics value u in the state by following all pointers from the value u in the store μ, collapsing the pointer graph structure into a self-contained value v in the value semantics.

Such a relation, however, is not preserved by evaluation in the presence of aliasing of mutable data, as a destructive update (such as a **put**) to a location in the store aliased by two variables would affect the value of both variables in the update semantics, but only one of them in the value semantics. Therefore, the refinement relation must additionally encode the uniqueness property ensured by our type system, which rules out not just *direct* aliasing, where two separate variables refer to the same data structure on the heap, but also *internal* aliasing, where a single data structure contains two or more aliasing pointers.

4.2.1 A typed refinement relation

The rules in Figure 20 define our refinement relation, extended to take into account the type system and aliasing of pointers.

Because our relation relates both update semantics and value semantics to types, we can derive a value-typing relation for either semantics by creatively erasing part of the rules. Erasing all the update semantics parts (highlighted like this) leaves a value-typing relation definition for the value semantics, and erasing all the value semantics parts (highlighted like this) gives a state-typing relation definition for the update semantics. As we ultimately prove preservation for this refinement relation across evaluation, the same erasure strategy can be applied to the proofs to produce a typing *preservation* proof for either semantics — a key component of type safety. Written $u \mid \mu : v : \tau \ [r * w]$, this judgement states that:

1. Transitively following all the pointers from u in the store μ results in the self-contained value v,
2. Both u and v have the type τ,
3. The set r contains all *read-only* pointers (according to the type τ) transitively accessible from u,
4. The set w contains all *writable* pointers transitively accessible from u, and
5. The value u contains no internal aliasing of any of the writable pointers in w, whether by read-only or writable pointers.

$$\boxed{u \mid \mu : v : \tau \quad [r * w]}$$

$$\frac{\ell < |T|}{\ell \mid \mu : \ell : T \quad [\emptyset * \emptyset]}\ \textsc{RLit} \qquad \frac{\varepsilon; x : \tau \vdash e : \tau'}{\langle\!\langle \lambda x.\, e \rangle\!\rangle \mid \mu : \langle\!\langle \lambda x.\, e \rangle\!\rangle : \tau \to \tau' \quad [\emptyset * \emptyset]}\ \textsc{RFun}$$

$$\frac{typeOf(f) = \forall \overrightarrow{a_i}.\, C \Rightarrow \tau}{\langle\!\langle \mathbf{abs}.\, f \mid \overrightarrow{\tau_i} \rangle\!\rangle \mid \mu : \langle\!\langle \mathbf{abs}.\, f \mid \overrightarrow{\tau_i} \rangle\!\rangle : \tau \left[\overrightarrow{\tau_i}/\overrightarrow{a_i}\right] \quad [\emptyset * \emptyset]}\ \textsc{RAFun}$$

$$\frac{u \mid \mu : v : \tau \quad [r * w]}{K\ u \mid \mu : K\ v : \langle K^{\circ}\ \tau, \overline{K_i^{\mathrm{u}}\ \tau_i} \rangle \quad [r * w]}\ \textsc{RVariant}$$

$$\frac{\mu(p) = a_{\mathrm{U}} \qquad a_{\mathrm{U}} \mid \mu :_A a_{\mathrm{V}} :_A \mathtt{A}\ \overrightarrow{\tau_i}\ \textcircled{r} \quad [r * \emptyset]}{p \mid \mu : a_{\mathrm{V}} : \mathtt{A}\ \overrightarrow{\tau_i}\ \textcircled{r} \quad [\{p\} \cup r * \emptyset]}\ \textsc{RAbsR} \qquad \frac{\mu(p) = a_{\mathrm{U}} \qquad a_{\mathrm{U}} \mid \mu :_A a_{\mathrm{V}} :_A \mathtt{A}\ \overrightarrow{\tau_i}\ \textcircled{w} \quad [r * w]}{p \mid \mu : a_{\mathrm{V}} : \mathtt{A}\ \overrightarrow{\tau_i}\ \textcircled{w} \quad [r * \{p\} \cup w]}\ \textsc{RAbsW}$$

$$\frac{a_{\mathrm{U}} \mid \mu :_A a_{\mathrm{V}} :_A \mathtt{A}\ \overrightarrow{\tau_i}\ \textcircled{u} \quad [r * w]}{a_{\mathrm{U}} \mid \mu : a_{\mathrm{V}} : \mathtt{A}\ \overrightarrow{\tau_i}\ \textcircled{u} \quad [r * w]}\ \textsc{RAbsU}$$

$$\frac{\begin{array}{c} u = \{\overline{\mathtt{f}_i \mapsto u_i}, \overline{\mathtt{f}_k \mapsto u_k}\} \qquad v = \{\overline{\mathtt{f}_i \mapsto v_i}, \overline{\mathtt{f}_k \mapsto v_k}\} \\ \text{for each } i,\ u_i \mid \mu : v_i : \tau_i \quad [r_i * w_i] \\ \text{for each } \mathtt{f}_j \in \overline{\mathtt{f}_i} \text{ where } i \neq j,\ w_i \cap (r_j \cup w_j) = \emptyset \end{array}}{u \mid \mu : v : \{\overline{\mathtt{f}_i^{\circ} : \tau_i}, \overline{\mathtt{f}_k^{\bullet} : \tau_k}\}\ \textcircled{u} \quad [\textstyle\bigcup_i r_i * \bigcup_i w_i]}\ \textsc{RRecU}$$

$$\frac{\begin{array}{c} \mu(p) = \{\overline{\mathtt{f}_i \mapsto u_i}, \overline{\mathtt{f}_k \mapsto u_k}\} \qquad v = \{\overline{\mathtt{f}_i \mapsto v_i}, \overline{\mathtt{f}_k \mapsto v_k}\} \\ \text{for each } i,\ u_i \mid \mu : v_i : \tau_i \quad [r_i * \emptyset] \end{array}}{p \mid \mu : v : \{\overline{\mathtt{f}_i^{\circ} : \tau_i}, \overline{\mathtt{f}_k^{\bullet} : \tau_k}\}\ \textcircled{r} \quad [\{p\} \cup \textstyle\bigcup_i r_i * \emptyset]}\ \textsc{RRecR}$$

$$\frac{\begin{array}{c} \mu(p) = \{\overline{\mathtt{f}_i \mapsto u_i}, \overline{\mathtt{f}_k \mapsto u_k}\} \qquad v = \{\overline{\mathtt{f}_i \mapsto v_i}, \overline{\mathtt{f}_k \mapsto v_k}\} \\ \text{for each } i,\ u_i \mid \mu : v_i : \tau_i \quad [r_i * w_i] \\ \text{for each } \mathtt{f}_j \in \overline{\mathtt{f}_i} \text{ where } i \neq j,\ w_i \cap (r_j \cup w_j) = \emptyset \end{array}}{p \mid \mu : v : \{\overline{\mathtt{f}_i^{\circ} : \tau_i}, \overline{\mathtt{f}_k^{\bullet} : \tau_k}\}\ \textcircled{w} \quad [\textstyle\bigcup_i r_i * \{p\} \cup \bigcup_i w_i]}\ \textsc{RRecW}$$

$$\boxed{a_{\mathrm{U}} \mid \mu :_A a_{\mathrm{V}} :_A \mathtt{A}\ \overrightarrow{\tau_i}\ s \quad [r * w]}$$

(abstract types are user-provided)

$$\boxed{U \mid \mu : V : \Gamma \quad [r * w]}$$

$$\frac{\begin{array}{c} \text{for each } x_i : \tau_i \in \Gamma, \\ x_i \mapsto u_i \in U \qquad x_i \mapsto v_i \in V \qquad u_i \mid \mu : v_i : \tau_i \quad [r_i * w_i] \\ \text{for each } x_j \in \overline{x_i} \text{ where } i \neq j,\ w_i \cap (r_j \cup w_j) = \emptyset \end{array}}{U \mid \mu : V : \Gamma \quad [r * w]}\ \textsc{REnv}$$

Fig. 20. The value typing and update/value refinement rules.

We call the sets r and w the *footprint* of the value u. By annotating the relation in this way, we can insert the required non-aliasing requirements into the rules for compound values such as records. Read-only pointers may alias other read-only pointers, but writable pointers may not alias any other pointer, whether read-only or writable.

4.2.1.1 Polymorphism. As mentioned in Section 2, we implement parametric polymorphism by specialising code to avoid paying the performance penalties of other approaches such as boxing. This means that polymorphism in Cogent is restricted to predicative rank-1 quantifiers, in the style of ML. This allows us to specify dynamic objects, such as our values and their typing and refinement relations, in terms of simple monomorphic types, without type variables. Thus, to evaluate a polymorphic program, each type variable must first be instantiated to a monomorphic type. Theorem 3.3 shows that any valid instantiation of a well-typed polymorphic program is well-typed, which implies the monomorphic specialisation case when all variables are instantiated. Thus, our results about our refinement relation can safely assume the well-typedness of the monomorphic specialisation of the program which is being evaluated.

4.2.1.2 Environments. Figure 20 also defines the refinement relation for environments and type contexts, written $U \mid \mu : V : \Gamma \ [\, r * w \,]$. Just as our original refinement relation enforces our uniqueness requirements inside a single value, the refinement relation for environments requires that the values of all variables in Γ meet the uniqueness requirements, such that no available variable will contain an alias of a writable pointer in any other available variable. Because this relation is only concerned with *available* variables, we can show that the context-splitting relation (given in Figure 3), which partitions the available variables into two sub-contexts, also neatly bifurcates the associated pointer sets r and w, such that the same environment viewed through either of the sub-contexts does not alias the other sub-context's writable pointers:

Lemma 4.1 (Splitting contexts splits footprints — `u_v_matches_split`)**.**

$$
\begin{array}{lll}
\textit{If an environment corresponds to a context,} & & U \mid \mu : V : \Gamma \ [\, r * w \,] \\
\textit{and that context is split in two, then} & \wedge & \varepsilon \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\
\textit{the heap footprint may also be split into} & \Rightarrow & \exists r_1\ w_1\ r_2\ w_2. \\
\textit{two sub-footprints, where each sub-footprint} & \wedge & r = r_1 \cup r_2 \wedge w = w_1 \cup w_2 \\
\textit{does not contain any aliases to writable} & \wedge & w_1 \cap (w_2 \cup r_2) = \emptyset \\
\textit{pointers in the other footprint, and where} & \wedge & w_2 \cap (w_1 \cup r_1) = \emptyset \\
\textit{the refinement relation holds for each} & \wedge & U \mid \mu : V : \Gamma_1 \ [\, r_1 * w_1 \,] \\
\textit{sub-footprint and sub-context respectively.} & \wedge & U \mid \mu : V : \Gamma_2 \ [\, r_2 * w_2 \,]
\end{array}
$$

4.2.1.3 Abstract Values. Because the representation of abstract values is defined externally to Cogent, the corresponding refinement relation $a_U \mid \mu :_A a_V :_A \mathtt{A}\ \vec{\tau_i}\ s \ [\, r * w \,]$ is defined externally also.

To ensure that our uniqueness invariant is maintained, certain requirements are placed on the pointer sets r and w in the user-supplied definition, depending on the sigil s:

- If the sigil s is ⓡ, then the set w must be empty. This is because abstract, read-only values are assumed to be shareable in Cogent's type system (see Figure 10), and therefore must not contain any writable pointers.
- If the sigil s is ⓤ, then both sets must be empty. Abstract, unboxed values meet the **Share**, **Drop** *and* **Escape** constraints. Therefore, w must be empty to avoid violating uniqueness directly, and r must be empty to prevent uniqueness violations in **let!** expressions.
- If the sigil s is ⓦ, then we only require that the sets r and w must be disjoint.

These pointer sets need not include all pointers contained within the data structure, but merely those pointers to *Cogent values* that are accessible via the interface exposed to Cogent. This allows data structures that rely on sharing, or would otherwise violate the uniqueness property of the type system, to be safely imported and used by Cogent functions. Similarly, the requirements of the frame relation here only apply to those pointers accessible from the Cogent side. Thus, during the execution of an imported C function, the uniqueness and framing conditions need not be adhered to — only the *interface* with Cogent needs to satisfy these requirements. The exact requirements of the Cogent interface are summarised in Section 4.2.4.

4.2.2 Framing

If the inputs to a Cogent program have a footprint $[\, r * w \,]$, then it is reasonable to require that no live objects in the store other than those referenced in w will be modified or affected by the evaluation of the program. In this way, two subprograms that affect different parts of the store may be evaluated independently. We formalise this requirement as a *framing* relation, which states exactly how evaluation may affect the mutable store.

Definition 4.1 (Framing Relation)**.** *Given an input set of writable pointers w_i to a store μ_i, and an output set of writable pointers w_o to a store μ_o, the framing relation $w_i \mid \mu_i$ **frame** $w_o \mid \mu_o$ ensures three properties for any pointer p:*

- Inertia*: Any value outside the footprint is unaffected, i.e. if $p \notin w_i \cup w_o$ then $\mu_i(p) = \mu_o(p)$.*
- Leak freedom*: Any value removed from the footprint must be freed, i.e. if $p \in w_i$ and $p \notin w_o$, then $\mu_o(p) = \bot$.*
- Fresh allocation*: Any value added to the footprint must not overwrite anything else, i.e. if $p \notin w_i$ and $p \in w_o$ then $\mu_i(p) = \bot$.*

If a program's evaluation meets the requirements specified in the framing relation, we can directly prove that our refinement relation is unaffected by any updates to the store outside the footprint:

Lemma 4.2 (Unrelated updates — `upd_val_rel_frame`)**.**

Assuming two unrelated pointer sets, where $w \cap w_1 = \emptyset$
one set is part of a value's footprint, and the $\wedge$ $u \mid \mu : v : \tau \; [\, r * w \,]$
other is the frame of a computation; then the $\wedge$ $w_1 \mid \mu$ **frame** $w_2 \mid \mu'$
refinement relation is re-established for the $\Rightarrow$ $u \mid \mu' : v : \tau \; [\, r * w \,]$
resultant store of that computation.

This result also generalises smoothly to our refinement relation for environments and contexts:

Lemma 4.3 (Unrelated updates for environments — `upd_val_rel_frame_env`)**.**

Assuming two unrelated pointer sets, where $w \cap w_1 = \emptyset$
one set is part of an environment's footprint, $\wedge$ $U \mid \mu : V : \Gamma \ [r * w]$
and the other is the frame of a computation; $\wedge$ $w_1 \mid \mu$ **frame** $w_2 \mid \mu'$
then the refinement relation is re-established $\Rightarrow$ $U \mid \mu' : V : \Gamma \ [r * w]$
for the resultant store of that computation.

The frame relation allows us to address the well known *frame problem* in verification and logic. Using these results along with Lemma 4.1, we can show (in the proof of Theorem 4.1) that evaluating one sub-expression does not affect any part of the store other than those mentioned in the heap footprint for the corresponding sub-context, and therefore that the refinement relation is preserved for the evaluation of subsequent sub-expressions.

4.2.3 Proving refinement

To prove our desired refinement statement, we must show that every evaluation in the update semantics has a corresponding evaluation in the value semantics that preserves our refinement relation. We decompose this into two main theorems: one to show general *preservation* of the refinement relation, and one to show *upward-propagation* of evaluation.

As previously mentioned, the *preservation* theorem can, with the right kind of selective vision, be viewed as a type preservation theorem for either semantics. Viewed in its entirety, it states that our refinement relation is preserved by any pair of evaluations for a well typed expression. Note that we relate the writable component of the footprints with the frame relation, and we require that the read-only component of the output to be a subset of the input. This means that a Cogent program can only read from pointers that are in its input footprint, an important aspect of memory safety crucial for security.

Theorem 4.1 (Preservation of Refinement/Typing Relation — `correspondence`)**.**

For a well-typed expression which evaluates $A; \Gamma \vdash e : \tau$
in the value semantics from environment V*,* $\wedge$ $V \vdash e \Downarrow_{\mathrm{V}} v$
and in the update semantics from U*:* $\wedge$ $U \vdash \mu \mid e \Downarrow_{\mathrm{U}} u \mid \mu'$
If V *and* U *correspond with some footprint,* $\wedge$ $U \mid \mu : V : \Gamma \ [r * w]$
then there exists another footprint $\Rightarrow$ $\exists\, r' \subseteq r.\ \exists\, w'.$
which results from the initial footprint, $w \mid \mu$ **frame** $w' \mid \mu'$
such that the result values correspond. $\wedge$ $u \mid \mu : v : \tau \ [r' * w']$

Proof By rule induction on the update semantics evaluation. For expressions which involve more than one sub-expression, we use Lemma 4.1 to establish that each sub-expression has a non-overlapping footprint. Then, from the inductive hypothesis, we know that the frame relation holds for each of these footprints. Then we use Lemma 4.2 and Lemma 4.3 to demonstrate that the evaluation of the first expression still preserves the refinement relation for the unrelated second expression.

To obtain this inductive hypothesis, we must additionally prove for each case that the frame relation holds for each evaluation. This is relatively simple, as the requirements of the frame relation are all straightforward consequences of our uniqueness type system. ■

Because it assumes the existence of an evaluation on both the update *and* value semantics levels, this preservation theorem is not sufficient to show refinement by itself. We still need to show that the value semantics evaluates whenever the update semantics does. This is where our *upward propagation* theorem comes in, proven by straightforward rule induction:

Theorem 4.2 (Upward evaluation propagation — `val_executes_from_upd_executes`)**.**

$$\begin{array}{lll} \textit{For a well-typed expression } e \textit{ which evaluates} & & A; \Gamma \vdash e : \tau \\ \textit{in the update semantics from } U\textit{, if } U \textit{ has a} & \wedge & U \vdash \mu \mid e \Downarrow_{\mathrm{U}} u \mid \mu' \\ \textit{corresponding value semantics environment,} & \wedge & U \mid \mu : V : \Gamma \ \ [\, r * w \,] \\ \textit{then } e \textit{ also evaluates in the value semantics.} & \Rightarrow & \exists v.\ V \vdash e \Downarrow_{\mathrm{V}} v \end{array}$$

With this result, the overall refinement of the value semantics to the update semantics is a simple corollary:

Theorem 4.3 (Value → Update refinement)**.**

$$\begin{array}{lll} \textit{If a typed expression } e\textit{, under environment} & & A; \Gamma \vdash e : \tau \\ U\textit{, evaluates to } u \textit{ in the update semantics,} & \wedge & U \vdash \mu \mid e \Downarrow_{\mathrm{U}} u \mid \mu' \\ \textit{and } U \textit{ corresponds to environment } V\textit{, then} & \wedge & U \mid \mu : V : \Gamma \ \ [\, r * w \,] \\ e \textit{ evaluates to some } v \textit{ under } V \textit{ in the value} & \Rightarrow & \exists v.\ V \vdash e \Downarrow_{\mathrm{V}} v \\ \textit{semantics, and there exists a footprint that} & & \wedge \exists\, r' \subseteq r.\ \exists\, w'. \\ \textit{results from the original footprint such that} & & \quad w \mid \mu \ \mathbf{frame}\ w' \mid \mu' \\ u \textit{ corresponds to } v\textit{.} & & \wedge\, u \mid \mu : v : \tau \ \ [\, r' * w' \,] \end{array}$$

This theorem forms an essential component of our overall compiler certificate, the construction of which is outlined in Section 5.

4.2.4 *Foreign functions*

Each of the above theorems makes certain assumptions about the semantics given to abstract functions, $[\![\cdot]\!]_{\mathrm{U}}$ and $[\![\cdot]\!]_{\mathrm{V}}$. Specifically, we must assume that the two semantics are *coherent*, in that they evaluate in analogous ways; that they respect the requirements of the frame relation to maintain our memory invariants; and that they do not introduce any observable aliasing, which would violate the uniqueness requirement of our type system.

These three properties are ensured by an assumption similar in format to the two lemmas used for the proof of refinement, Theorems 4.1 and 4.2. Specifically, we assume for a foreign function f of type $\tau \to \rho$ that, given input values u and v that correspond, i.e. $u \mid \mu : v : \tau \ \ [\, r * w \,]$,

1. If the update semantics evaluates, i.e. $[\![f]\!]_{\mathrm{U}}(\mu, u) = (\mu', u')$, then the value semantics evaluates, i.e. $[\![f]\!]_{\mathrm{V}}(v) = v'$;
2. Their results correspond, i.e. $u' \mid \mu' : v' : \rho \ \ [\, r' * w' \,]$ for some $r' \subseteq r$ and w'; and
3. The frame relation holds, i.e. $w \mid \mu \ \mathbf{frame}\ w' \mid \mu'$.

These assumptions directly satisfy any obligations about foreign functions that arise in the proofs of Theorems 4.1 and 4.2, thus providing all the necessary ingredients to prove refinement in the presence of foreign functions.

The refinement theorem between our two semantic interpretations, vital to our overall framework, is only possible because Cogent is a significantly restricted language, disallowing aliasing of writable pointers. This *semantic shift* refinement has been proven in a mechanical theorem prover, definitively confirming the intuition of Wadler (1990), and extending existing pen-and-paper theoretical work (Hofmann, 2000) to apply to real-world languages with heap-allocated objects and pointers.

5 Refinement framework

The refinement proof from value to update semantics presented in Section 4 is only one piece, albeit a crucial one, of the overall refinement chain from the Isabelle/HOL embedding of the Cogent code down to the generated C.

Both below and above this semantic shift, specialised tactics in Isabelle/HOL generate numerous refinement proofs, which mirror each transformation made by the Cogent compiler. These refinement proofs are combined into a proof of a top-level refinement theorem that connects the semantics of the C code with a higher-order logic (HOL) embedding of the Cogent code. The proof structure of this refinement framework is outlined in Figure 21, and involves a number of different embeddings: *shallow embeddings*, where the program is represented as a semantically equivalent HOL term, and also *deep embeddings*, where the program is represented as an abstract syntax tree in HOL.

As shallow embeddings have a direct semantic interpretation in HOL, they are easier to reason about concretely: that is, individual shallowly embedded programs are mere mathematical functions, and are therefore amenable to verification using standard theorem prover definitions and tactics. This is why the more abstract embeddings at the top of the chain are all shallow, as these embeddings are used for further functional correctness verification, connecting to a higher level abstract specification written specifically for the program under examination.

On the other hand, shallow embeddings make it very difficult to prove results for *all* programs, such as the refinement theorem between update and value semantics in Section 4. In such situations, *deep* embeddings are preferred, where the program terms are represented as an abstract syntax tree, and separate evaluation relation(s) are defined to provide semantics, such as those in Section 4. This allows us to perform induction on program terms, exhaustively verifying a property for *every program*. Furthermore, this decoupling of term structure and semantics allows us to define multiple semantic interpretations for the same set of terms. We need both of these advantages to prove theorems like Theorem 4.3, which justify the semantic shift from value semantics to update semantics. The embeddings in the middle of the refinement chain are all therefore deep, as this is where Theorem 4.3 is used.

The lower-level embeddings closer to C code are also shallowly-embedded. This is because the Cogent verification framework builds on two existing mature verification tools for C software in Isabelle/HOL: The C→SIMPL Parser used in the seL4 project, and the automatic C abstraction tool AutoCorres (Greenaway *et al.*, 2014, 2012). As both of these

are designed for manual verification of specific C programs, they choose to represent C code using shallow embeddings, suitable for human consumption. The C Parser imports C code into the Isabelle-embedded language SIMPL (Schirmer, 2005) extended with the memory model of Tuch *et al.* (2007); while AutoCorres abstracts this SIMPL code into HOL terms involving the *non-deterministic state monad* first described in Cock *et al.* (2008).

Each of the refinement proofs presented in Figure 21 is established via *translation validation* (Pnueli *et al.*, 1998*b*). That is, rather than *a priori* verification of phases of the compiler, specialised Isabelle tactics and proof generators are used to establish a refinement proof *a posteriori*, relating the input and output of each compiler phase after the compiler has executed. For the most part, this is because these refinement stages involve shallow embeddings, which do not allow the kind of term inspection needed to directly model a compiler phase and prove it correct. It also has the advantage of allowing us some flexibility in implementation, as the *post-hoc* generated refinement proof is not dependent on the exact implementation of the compiler.

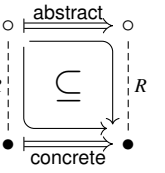
This approach is not without its drawbacks, however. Chief among these is the lack of a completeness guarantee: While we know that the compiler acted correctly if Isabelle/HOL validates the generated refinement proof, there is no way to establish any formal guarantee that Isabelle/HOL will always validate the generated proof if the compiler acts correctly. In a verified compiler, proofs need to be checked only once, thus indicating that the compiler is trustworthy; but with translation validation, proofs must be checked after each compilation.

5.1 Refinement and forward simulation

As mentioned in Section 4, each of our refinement proofs is based on the *forward simulation* technique for data refinement, an idea independently discovered by many people but crystallised by de Roever & Engelhardt (1998). This technique involves defining a *refinement relation* R that connects *abstract* states (for example in the HOL embedding) to corresponding *concrete* states (for example in the C code). Then, assuming R holds for initial states, we must prove that every possible concrete evaluation can be matched by a corresponding abstract execution, resulting in final states for which R is re-established:

$$R; \langle \mathsf{abstract} \rangle \subseteq \langle \mathsf{concrete} \rangle; R$$

(where ; is forward composition of relations)



These relation preservation proofs only imply refinement given the assumption that the relation R holds initially. This means that the two semantics must evaluate from comparable environments. A similar assumption is made for the verification of seL4 (Klein *et al.*, 2009). Bridging this remaining gap in the verification chain must be made on a case-by-case basis, and is the subject of further research.

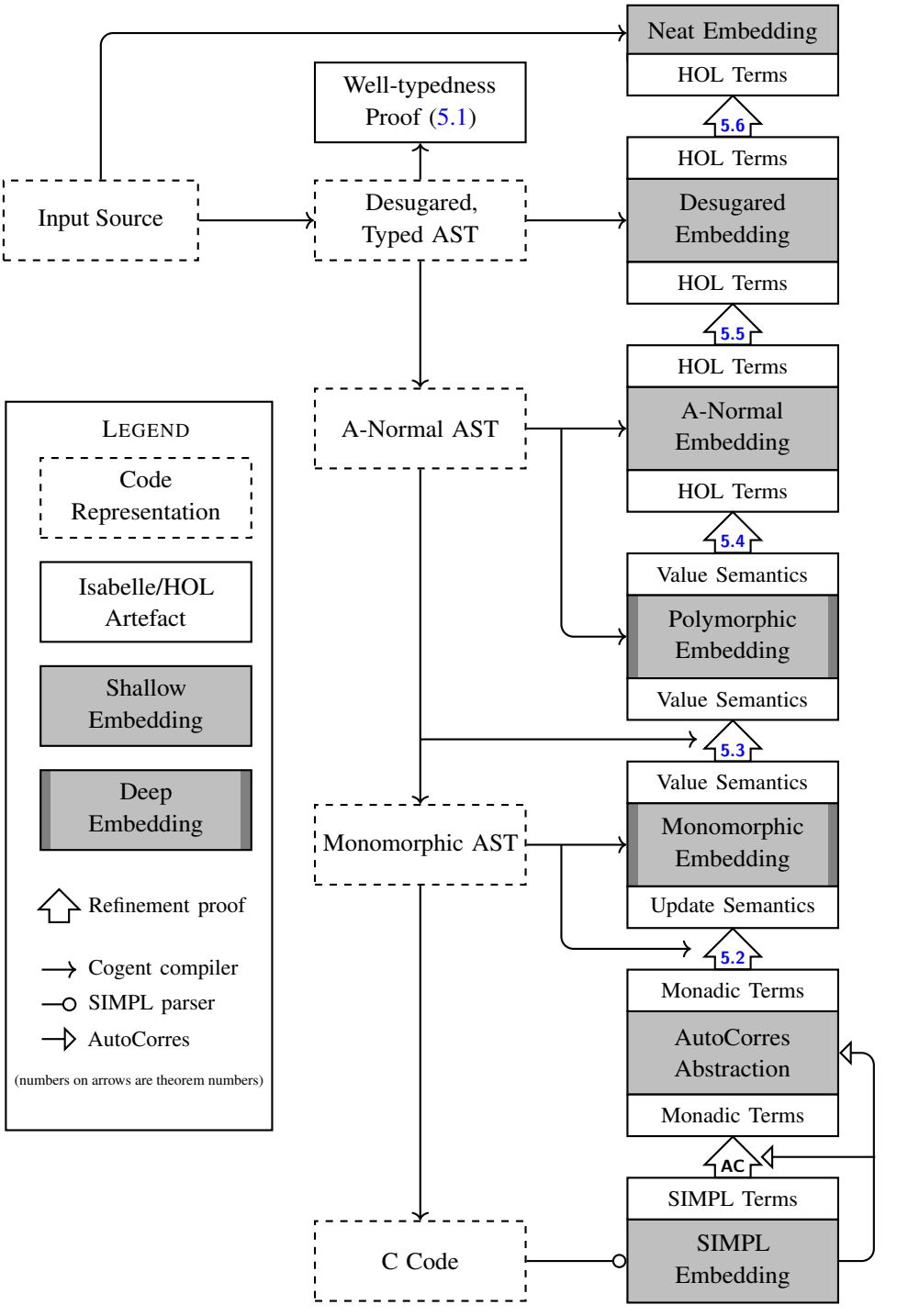


Fig. 21. Refinement phases of Cogent (arrows indicate refinement theorems)

5.2 Well-typedness proof

The refinement theorems concerning the monomorphic deep embedding, such as our semantic shift refinement relation in Section 4, assume that the Cogent program is well-typed. Therefore, it is necessary to prove in Isabelle/HOL that the generated monomorphic deep embedding is well-typed.

Specifically, the compiler will generate Isabelle/HOL definitions of the $defnOf(\cdot)$ and $typeOf(\cdot)$ environments (described in Section 3) for the monomorphised version of the Cogent program, and then prove the following theorem via a custom Isabelle tactic:

Generated Theorem 5.1 (Typing)**.** *Let f be the name of a monomorphic Cogent function, where $defnOf(f) = \lambda x.\ e$ and $typeOf(f) = \tau \to \rho$. Then, $x : \tau \vdash e : \rho$.*

Because the typing rules we have presented are not algorithmic, we require additional information from the Cogent compiler to produce an efficient deterministic algorithm that synthesises a proof of this theorem. There are a number of sources of non-determinism in these typing rules:

1. The use of the context-splitting relation in the typing rules means that a naïve algorithm for proof synthesis could necessitate traversing over every sub-expression to determine which variables are used in each split. The compiler eliminates the need for this by emitting a table of *hints* that informs the proof-synthesis tactic on how each context is split, indicating which variables are used in each sub-expression.
2. As the subsumption rule of subtyping is not syntax-directed, it could potentially be used at any point in the typing derivation. To eliminate non-determinism resulting from such potential upcasts, the compiler includes special **promote** syntax nodes in the generated deep embedding, which indicate precisely where in the syntax tree subsumption has been used.
3. Integer literals are overloaded in the Cogent syntax, which can make their typing ambiguous. The compiler resolves this simply by annotating all literals with their precise inferred type in the generated deep embedding.

Armed with this additional information from the compiler, our proof synthesis tactic proceeds by merely applying each of the typing rules from Section 3 as introduction rules. The choice of which rule to apply, and which instantiations of type variables to use, is now entirely unambiguous.

Because HOL is a *proof-irrelevant* logic, once we prove the top-level typing theorem for a function, we lose access to the typing lemmas for each of the sub-expressions that make up the function's body. While it is true that well-typedness of an expression implies well-typedness of its subexpressions, we specifically need access to the theorems (and the instantiations of metavariables) that are used to construct the overall well-typedness theorem. As theorems do not contain any information or structure beyond their provability, we cannot precisely extract these lemmas from the theorem. As we will see in Section 5.3.1, our synthesised refinement proof from the monomorphic deep embedding to the AutoCorres embedding needs access to all of these typing lemmas. For this reason, our tactic remembers each intermediate typing derivation in a tree structure as it proves the top-level typing

theorem. This tree structurally matches the derivation tree for the typing theorem itself: each node contains the intermediate theorem for that part of the typing derivation.

5.3 Refinement phases

The only synthesised proof artefacts in our framework aside from the proof of well-typedness are the six refinement theorems presented in Figure 21. While they are all refinement theorems proven by translation validation, the exact structure of the theorem and the mechanism used to prove them differs in each case.

Each of the generated embeddings correspond to the parts of the program written in Cogent. As mentioned in Section 2, many functions in Cogent software are *foreign*, i.e. written externally in C. Each of the refinement certificates presented here assume similar refinement statements for each of the foreign functions. Therefore, to fully verify Cogent software, a proof engineer must provide manually-written abstractions of C code, and manually prove the refinement theorems that are automatically generated for Cogent code. As demonstrated in Section 2, these foreign functions tend to be reusable library functions. Thus, the cost in terms of verification effort of these functions can be amortised by reusing these manually-verified libraries in multiple systems.

5.3.1 SIMPL and AutoCorres

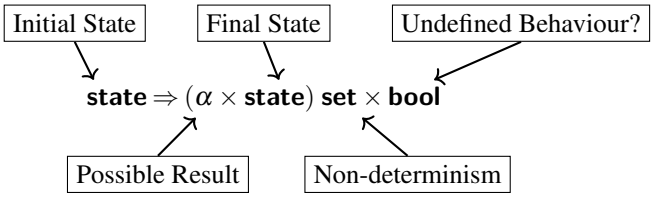
As previously mentioned, we assign a formal semantics to C code using the C→SIMPL parser also used in the verification of seL4 and other projects. SIMPL is an imperative language embedded in Isabelle/HOL with straightforward semantics designed by Schirmer (2005), intended for use with program logics such as Hoare Logic for software verification. The language semantics is parameterised by a type used to model all mutable state used in the program. The C→SIMPL parser instantiates this parameter with a generated Isabelle record type containing a field for each local variable in the program, along with a special field for the C heap using the memory model of Tuch *et al.* (2007).

While we could, in principle, work with the SIMPL code directly, its memory model treats the heap essentially as a large collection of bytes: It does not make use of any of the information from C's type system to automatically abstract heap data structures. This is, in part, due to the nature of manually-written C code, where programmers often subvert the type system using potentially unsafe casts, reinterpreting memory based on dynamic information. Because our code is automatically generated, and does not rely on dynamically reinterpreting memory, we can abstract away from the bits and bytes of the C heap to a higher-level, typed representation — this is where AutoCorres comes in.

AutoCorres (Greenaway *et al.*, 2014, 2012) is a tool intended to reduce the cost of manually verifying C programs in Isabelle/HOL. It works by automatically abstracting the SIMPL interpretation of the C code into a shallow embedding using the non-deterministic state monad of Cock *et al.* (2008). In this monad, computations are represented using the following HOL type:

do $\cdots$; $\cdots$ **od**	sequence of statements
$x \leftarrow P$	monadic binding
condition c P_1 P_2	run P_1 if c is true, else run P_2
return v	monadic return
gets f	return the part of the state given by f
modify h	update the state using function h
guard g	program fails if g is false
P >>= Q	monadic bind (desugared)

Fig. 22. The monadic embedding do-notation.



Here, **state** represents all the global state of the C program, including any global variables, and a set of *typed heaps*, one for each C type used on the heap in the C program. A typed heap for a particular type τ is modelled as a function τ **ptr** $\Rightarrow \tau$.

Given an input **state**, the computation will produce a set called *results*, consisting of the possible return value and final **state** pairs, as well as a flag called *failed*, which indicates when undefined behaviour is possible.

In the generated embedding, each access to a typed heap is protected by a *guard* that ensures that the given pointer is valid, to ensure that the heap function is defined for that particular input. Proving that these guards always hold is therefore essential for showing that the program is free of undefined behaviour. When proving refinement from Cogent code, we discharge these obligations by appealing to a globally-invariant *state relation* that implies the validity of all pointers in scope.

Figure 23 shows a very simple Cogent program that negates the boolean interpretation of an unsigned integer inside a boxed record. To simplify code generation to C, the Cogent compiler first transforms the program into *A-normal form*, an intermediate representation first developed by Sabry & Felleisen (1992). This form ensures that a unique variable binding is made for each step of the computation, making it easier to convert an expression-oriented language like Cogent to a statement-oriented language like C. This A-normal form also simplifies the refinement tactic used to connect the AutoCorres-abstracted C code to the Cogent deep embedding, described in the next section. As shown in Figure 23, the monadic embedding of the C code has a strong resemblance to the A-normal form of the Cogent program. Figure 22 describes the notation used in HOL for the monadic embedding, inspired by the **do**-notation of Haskell (Marlow, 2010). Because AutoCorres is designed for human-guided verification, it includes a number of context-sensitive rules to simplify the resulting monadic embedding. For example, it includes features which can simplify reasoning about machine words into reasoning about natural numbers, if it can prove that no overflow occurs. Because we are using AutoCorres as part of an automated framework,

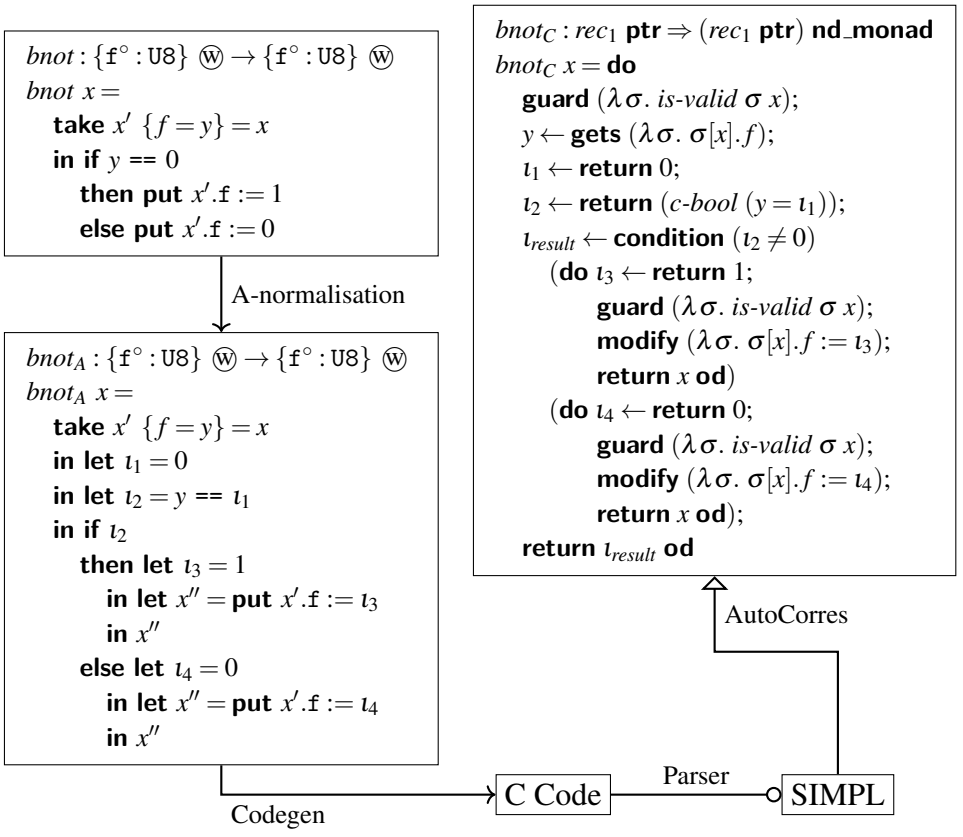


Fig. 23. An example program, its A-normalisation, and monadic embedding.

most of these abstraction and simplification features are disabled to give highly predictable output. The only significant feature used is the abstraction to the typed heap model.

As can be seen in Figure 21, AutoCorres synthesises a refinement proof, showing that the monadic embedding is a true abstraction of the imported SIMPL code. While this refinement proof forms a part of our overall compiler certificate, this proof is entirely internal to AutoCorres, and the SIMPL embedding is not exposed. Therefore, our combined refinement theorem, documented in Section 5.3.6, treats the AutoCorres-generated monadic shallow embedding as the most concrete representation in our overall refinement statement.

5.3.2 *AutoCorres and Cogent*

While AutoCorres provides some much-needed abstraction on top of C code, the monadic embedding still resembles the generated C code far more than the Cogent code from which it was generated. We still need a technique to validate the code generation phase of the compiler, and synthesise a refinement proof to connect the semantics of Cogent to this monadic embedding (Rizkallah *et al.*, 2016).

The C code generation phase of the compiler proceeds relatively straightforwardly, and does not perform global optimisations or code transformations. Transformations such as

$$
\begin{array}{lrcll}
\text{representations} & \delta & ::= & T & (\textit{primitives}) \\
& & | & \textbf{Fun} & (\textit{functions}) \\
& & | & \textbf{Abstract}\ \mathtt{A} & (\textit{abstract types}) \\
& & | & \{\overline{\mathtt{f} : \delta}\} & (\textit{records}) \\
& & | & \langle\overline{K\ \delta}\rangle & (\textit{variants}) \\
& & | & \textbf{Ptr}\ \delta & (\textit{boxed types})
\end{array}
$$

$$
\boxed{\mathit{erase}(\cdot) : \tau \to \delta}
$$

$$
\begin{array}{lcl}
\mathit{erase}(T) & = & T \\
\mathit{erase}(\tau \to \rho) & = & \textbf{Fun} \\
\mathit{erase}(\mathtt{A}\ \overline{\tau_i}\ \textcircled{u}) & = & \textbf{Abstract}\ \mathtt{A} \\
\mathit{erase}(\mathtt{A}\ \overline{\tau_i}\ \textcircled{r}) & = & \textbf{Ptr}\ (\textbf{Abstract}\ \mathtt{A}) \\
\mathit{erase}(\mathtt{A}\ \overline{\tau_i}\ \textcircled{w}) & = & \textbf{Ptr}\ (\textbf{Abstract}\ \mathtt{A}) \\
\mathit{erase}(\{\overline{\mathtt{f}_i^u : \tau_i}\}\ \textcircled{u}) & = & \{\mathtt{f}_i : \mathit{erase}(\tau_i)\} \\
\mathit{erase}(\{\overline{\mathtt{f}_i^u : \tau_i}\}\ \textcircled{r}) & = & \textbf{Ptr}\ \{\overline{\mathtt{f}_i : \mathit{erase}(\tau_i)}\} \\
\mathit{erase}(\{\overline{\mathtt{f}_i^u : \tau_i}\}\ \textcircled{w}) & = & \textbf{Ptr}\ \{\overline{\mathtt{f}_i : \mathit{erase}(\tau_i)}\} \\
\mathit{erase}(\langle\overline{K_i^u\ \tau_i}\rangle) & = & \langle\overline{K_i\ \mathit{erase}(\tau_i)}\rangle
\end{array}
$$

Fig. 24. Partial type erasure to determine C representation

the aforementioned A-normalisation occur in earlier compiler phases and are verified at a higher level in the overall refinement certificate. As all terms are in A-normal form at this stage, nested sub-expressions are replaced with explicit variable bindings. The refinement framework consists of a series of compositional rules designed to prove refinement in a syntax-directed way, one for each A-normal expression.

5.3.2.1 Refinement Relations. While our high level view of refinement from de Roever & Engelhardt (1998) defines just a single *refinement relation R* that relates abstract and concrete states, three relations must be defined when proving refinement from the Cogent deep embedding (with the update semantics) to the AutoCorres monadic embedding. The Cogent compiler generates each of these relations after obtaining the monadic shallow embedding and the definitions of its typed heaps from AutoCorres:

1. A *value relation*, written $\mathscr{R}_{\text{val}}$, that relates Cogent update-semantics values (defined in Figure 16) to monadic C values. Because AutoCorres generates separate Isabelle types for each C type, this value relation is defined for each generated type using Isabelle's ad-hoc overloading features. Morally, this relation asserts the equality of the two values. For example, the record type in the example in Figure 23 would cause the following definitions to be generated:

$$
\begin{array}{llcll}
(\ell, & v_c :: 8\ \textbf{word}) & \in \mathscr{R}_{\text{val}} & \Leftrightarrow & (\ell = v_c) \\
(\{\mathtt{f} \mapsto u\}, & v_c :: rec_1) & \in \mathscr{R}_{\text{val}} & \Leftrightarrow & (u, v_c.f) \in \mathscr{R}_{\text{val}} \\
(p, & v_c :: rec_1\ \textbf{ptr}) & \in \mathscr{R}_{\text{val}} & \Leftrightarrow & (p = v_c)
\end{array}
$$

Note that the definition for the C structure type rec_1 depends on the definition for 8-bit words. The compiler always outputs these definitions in dependency order to ensure that this does not pose a problem.

2. A *type relation*, written $\mathscr{R}_{\text{type}}$, which allows us to determine which AutoCorres heap to select for a given Cogent type. As with the value relation, the type relation is defined using ad-hoc overloading. It does not relate Cogent types directly to AutoCorres-generated types, but rather a Cogent *representation*, as defined in Figure 24. A representation, written as δ, is a partially-erased Cogent type, which contains all the necessary information to determine which C type is used to represent it. Therefore, the usage tags on taken fields and constructors, type parameters in abstract values, the read-only status of sigils, and other superfluous information is discarded. The function $erase(\cdot)$ describes how to convert a type to its representation.
   The reasoning behind the decision to relate *representations* instead of Cogent types to C types is quite subtle: Unlike in C, for a Cogent value to be well-typed, all accessible pointers in the value must be valid (i.e. defined in the store μ) and the values those pointers reference must also, in turn, be well-typed. For taken fields of a record, however, no typing obligations are required for those values, as they may include invalid pointers (see the update semantics erasure of the rules in Figure 20). In C, however, taken fields must still be well-typed, and values can be well-typed even if they contain invalid pointers. Therefore, it is impossible to determine from a Cogent value alone what C type it corresponds to, making the overloading used for these relations ambiguous.
   To remedy this, we additionally include the representation of a value's type inside each update-semantics value u in our formalisation, although this detail is not shown in Figure 16. This means that we can determine which C type corresponds to a Cogent value simply by extracting the relevant representation, without requiring recursive descent into the heap or unnecessary restrictions on taken fields.
3. A *state relation*, written $\mathscr{R}$, which relates a Cogent store μ to a collection of AutoCorres heaps σ. We define $(\mu, \sigma) \in \mathscr{R}$ if and only if for all pointers p in the domain of μ, there exists a value v in the appropriate heap of σ (selected by $\mathscr{R}_{\text{type}}$) at location p such that $(\mu(p), v) \in \mathscr{R}_{\text{val}}$.
   The state relation cannot be overloaded in the same way as $\mathscr{R}_{\text{val}}$ and $\mathscr{R}_{\text{type}}$, because it relates the heaps for every type simultaneously. We introduce an intermediate state relation, $\mathscr{R}_{\text{heap}}$, 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. We define $\mathscr{R}_{\text{heap}}$ for each C type τ_C that appears on the heap as follows:

$$(\mu, \sigma_{\tau_C}) \in \mathscr{R}_{\text{heap}} \quad \Leftrightarrow \quad \forall p.\ \mu(p) = u \wedge (repr(u), \tau_C) \in \mathscr{R}_{\text{type}} \\ \Rightarrow \mathit{is\text{-}valid}\ \sigma_{\tau_C}\ p \wedge (u, \sigma_{\tau_C}[p]) \in \mathscr{R}_{\text{val}}$$

   where *repr* gives the representation for a value, and $\mathit{is\text{-}valid}\ \sigma\ p$ is true iff the pointer p points to a valid object in the heap σ. The state relation $\mathscr{R}$ over all typed heaps is defined to be merely the conjunction of every $\mathscr{R}_{\text{heap}}$ for each C type used in the program:

$$(\mu, \sigma) \in \mathscr{R} \ \Leftrightarrow\ (\mu, \sigma_{\tau_1}) \in \mathscr{R}_{\text{heap}} \wedge (\mu, \sigma_{\tau_2}) \in \mathscr{R}_{\text{heap}} \wedge \cdots$$

$$\frac{(x \mapsto u) \in U \qquad (u, v_C) \in \mathscr{R}_{\text{val}}}{\textbf{corres}\ \mathscr{R}\ x\ (\textbf{return}\ v_C)\ U\ \Gamma\ \mu\ \sigma}\ \text{C-VAR}$$

$$\frac{\begin{array}{c}\vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau \qquad \textbf{corres}\ \mathscr{R}\ e_1\ P_1\ U\ \Gamma_1\ \mu\ \sigma \\ \forall u\ v_C\ \mu'\ \sigma'.\ (u, v_C) \in \mathscr{R}_{\text{val}} \Rightarrow \textbf{corres}\ \mathscr{R}\ e_2\ (Q\ v_C)\ (x \mapsto u, U)\ (x : \tau, \Gamma_2)\ \mu'\ \sigma'\end{array}}{\textbf{corres}\ \mathscr{R}\ (\textbf{let}\ x = e_1\ \textbf{in}\ e_2)\ (P\ \texttt{>>=}\ Q)\ U\ \Gamma\ \mu\ \sigma}\ \text{C-LET}$$

$$\frac{\begin{array}{c}\vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Gamma_1 \vdash x : \texttt{Bool} \qquad (x \mapsto \ell) \in U \qquad (\ell = \texttt{True}) \Leftrightarrow (c \neq 0) \\ \textbf{corres}\ \mathscr{R}\ e_1\ P_1\ U\ \Gamma_2\ \mu\ \sigma \qquad \textbf{corres}\ \mathscr{R}\ e_2\ P_2\ U\ \Gamma_2\ \mu\ \sigma\end{array}}{\textbf{corres}\ \mathscr{R}\ (\textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ (\textbf{condition}\ c\ P_1\ P_2)\ U\ \Gamma\ \mu\ \sigma}\ \text{C-IF}$$

Fig. 25. Some example **corres** rules

5.3.2.2 Correspondence. We define refinement generically between a monadic C computation P and a Cogent expression e, evaluated under the update semantics. We denote refinement with a predicate **corres**, similar to the refinement calculus of Cock *et al.* (2008). The state relation $\mathscr{R}$ changes for each Cogent program, so we parameterise **corres** by an arbitrary state relation $\mathscr{R}$. It is additionally parameterised by the typing context Γ and the environment U, as well as by the initial update semantics store μ and typed heaps σ:

Definition 5.1 (Cogent→C correspondence)**.**

$$\begin{array}{lcl}\textbf{\textit{corres}}\ \mathscr{R}\ e\ P\ U\ \Gamma\ \mu\ \sigma & = & (\exists r\ w.\ U \mid \mu : \Gamma\ [\, r * w\,]) \wedge (\mu, \sigma) \in \mathscr{R} \\ & & \Rightarrow\ \neg \mathit{failed}\ (P\ \sigma) \\ & & \wedge\ \ \forall (v_C, \sigma') \in \mathit{results}\ (P\ \sigma).\ \\ & & \quad \exists \mu'\ u.\ \ U \vdash \mu \mid e \Downarrow_U u \mid \mu' \\ & & \quad\ \wedge\ \ (\mu', \sigma') \in \mathscr{R} \wedge (u, v_C) \in \mathscr{R}_{val}\end{array}$$

This definition states that, for well-typed stores μ where the state relation R holds initially, the monadic embedding of the C program P will not exhibit any undefined behaviour and, moreover, for all executions of P there must exist a corresponding execution under the update semantics of the expression e such that the final states are related by the state relation $\mathscr{R}$, and the returned values are related by the value relation $\mathscr{R}_{val}$.

AutoCorres proves that if *failed* is false for a given program, then the C code is type and memory-safe, and is free of undefined behaviour (Greenaway *et al.*, 2014). 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 $\mathscr{R}$, and the additional typing assumptions are preserved thanks to update-semantics type preservation corollary of Theorem 4.1.

Figure 25 shows some of the simpler **corres** rules used by our Isabelle tactic to automatically prove refinement. The rule C-VAR for variables, relating them to a monadic **return** operation; the rule C-LET for **let** bindings, relating them to the monadic bind operator >>=; and the rule C-IF for conditional expressions, relating them to the **condition** operation from Figure 22. Note that in the rule C-IF, we can assume that the condition expression x is

a variable, as the Cogent code is already in A-normal form. In our Isabelle formalisation, we have defined many **corres** rules which validate the entire Cogent language, however they all follow the same basic format as the rules presented in Figure 25. The assumptions for these rules fall into three main groups:

1. Each rule for compound expressions includes well-typedness assumptions about some sub-expressions. Theorem 4.1, used to discharge value-typing assumptions in the **corres** definition, also has well-typedness assumptions. Our automated tactic therefore needs access to all of the typing derivations used to construct the overall typing theorem for a program. A mere top-level well-typedness theorem is not sufficient to discharge these obligations. This is why we store each intermediate typing theorem as a tree in Isabelle/ML, as previously mentioned in Section 5.2.
2. Expressions which interact with the heap, such as **take** and **put** for boxed records, must maintain the relation $\mathscr{R}$ between the Cogent store and the AutoCorres typed heaps. Because the definitions of the typed heaps and the definition of *is-valid* are not provided until after we import the C program, we define these rules generically, parameterised by these AutoCorres-provided definitions. Then, after importing the C program, our framework automatically generates and proves *specialised* versions of the rule for the specific program at hand. This specialisation technique is documented in detail by Rizkallah *et al.* (2016).
3. Expressions such as **take** and **let** which are not made into leaves of the syntax tree by A-normalisation typically have recursive **corres** assumptions for each sub-expression, resolved by recursively applying our tactic. Because each rule is defined for exactly one A-normal Cogent expression, these proofs are syntax-directed and can be resolved by recursive descent without ambiguity or back-tracking.

Cogent is a total language and does not permit recursion, so we have, in principle, a well-ordering on function calls in any program. Therefore, our tactic proceeds by starting at the leaves of the call graph, proving **corres** theorems bottom-up until refinement is proven for the entire program.[3]

Generated Theorem 5.2 (Update Semantics $\sqsubseteq$ Monadic Embedding)**.** *Let f be the name of a monomorphic and A-normal Cogent function, where $\mathit{defnOf}(f) = \lambda x.\ e$ and $\mathit{typeOf}(f) = \tau \to \rho$. Let P be the monadic shallow embedding derived from the generated C code for f. Then, for any corresponding arguments u and v_C of the appropriate type, we have:*

$$\forall \mu\ \sigma.\ (u, v_C) \in \mathscr{R}_{\text{val}} \Rightarrow \textbf{corres}\ \mathscr{R}\ e\ (P\ v_C)\ (x \mapsto u)\ (x : \tau)\ \mu\ \sigma$$

This picture is complicated somewhat by the presence of higher order functions in Cogent, which are commonly used for loops and iteration. When higher order functions are involved, the call graph is no longer so clear, as it cannot be strictly determined syntactically. Our framework supports second-order functions by first proving **corres** for all argument functions (e.g. the loop body) before establishing **corres** for the second-order function (e.g. the loop combinator), a kind of defunctionalisation where we need consider only higher-order

[3] There are options to achieve this in the presence of recursion. Primitive or structural recursion *a la* Coquand & Paulin (1988) is one such option.

functions applied to specific function arguments[4]. We could straightforwardly extend this framework to any higher-order functions, but second-order functions were sufficient to cover our case-study file system implementations (Amani *et al.*, 2016).

5.3.3 Monomorphisation

The next refinement step that is established by translation validation is monomorphisation. The monomorphisation proof shows that the supplied polymorphic Cogent program is an abstraction of the monomorphised equivalent produced by the compiler. At this point, we can operate freely in value semantics without concern for mutable state, as the semantic shift occurs on the monomorphic deep embedding, justified by Theorem 4.3.

The Cogent compiler converts polymorphic programs into monomorphic ones by generating monomorphic specialisations of polymorphic functions based on each type argument used in the program, *a la* Harper & Morrisett (1995). Inside our framework, the compiler generates a renaming function θ that, for a polymorphic function name f_p and types $\vec{\tau}$, yields a specialised monomorphic function name f_m. Just as we assume that foreign functions are correctly implemented in C, we also assume that their behaviour remains consistent under θ. We write two main Isabelle/HOL functions to simulate this compiler monomorphisation phase, each defined in terms of an arbitrary renaming function θ: An *expression* monomorphisation function, $\mathscr{M}_\theta(\cdot)$, which applies θ to any type applications in the expression; and a *value* monomorphisation function, $\mathscr{M}^{\mathrm{v}}_\theta(\cdot)$ which applies the expression monomorphisation function $\mathscr{M}_\theta$ to each expression inside a value (i.e. in a function value). Then, we generate a proof which shows that the monomorphised program the Isabelle function produces is identical to that produced by the compiler. If the programs are not structurally identical, this indicates a bug in the compiler.

Generated Theorem 5.3 (Monomorphisation)**.** *Let θ be the generated renaming function and f be a polymorphic function where $\mathit{defnOf}(f) = \lambda x.\ e$. Let f_m be a monomorphised version of f generated by the compiler. Then, $\mathit{defnOf}(f_m) = \lambda x.\ \mathscr{M}_\theta(e)$.*

Then, it remains to prove that the monomorphic program is a refinement of the polymorphic one:

Theorem 5.1 (Monomorphisation Refinement)**.** *Let f be a (polymorphic) Cogent function and $\mathit{defnOf}(f) = \lambda x.\ e$. Let v be an appropriately-typed argument for f. Let θ be any renaming function. Then for any v', if $(x \mapsto \mathscr{M}^{\mathrm{v}}_\theta\ v) \vdash \mathscr{M}_\theta\ e \Downarrow_{\mathrm{v}} \mathscr{M}^{\mathrm{v}}_\theta\ v'$, then $(x \mapsto v) \vdash e \Downarrow_{\mathrm{v}} v'$.*

Proof This is proven once and for all by rule induction over the value semantics relation, with appropriate assumptions being made about foreign functions. Typing assumptions are discharged via Theorem 3.3. ■

5.3.4 A-normal and deep embeddings

Above the semantic shift and monomorphisation stages of our refinement chain, we no longer have any use for deep embeddings. As we are now in the value semantics, shallow

[4] This defunctionalisation is merely an implementation detail of the proof framework and doesn't affect one's ability to reason about higher order functions on the level of Cogent specifications.

embeddings are preferred, as Isabelle's simplifier can work wonders on pure HOL terms. Therefore, as with Section 5.3.1, we must connect a shallow embedding to a deep embedding. However, this time the deep embedding is the *bottom* of the refinement, and the shallow embedding is comprised of simple pure functions, rather than procedures in a state monad.

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 erase usage tags, sigils and other type system features used for uniqueness type checking, converting the Cogent program to a simple pure term in the fragment of System $\mathscr{F}$ (Girard, 1971; Reynolds, 1974) supported by Isabelle/HOL. As we have already made use of the type system to justify our semantic shift, we no longer need these type system features in the value semantics.

In addition to these definitions, we automatically prove a theorem that each generated HOL function refines to its corresponding deeply embedded polymorphic Cogent term under the value semantics. Refinement is formally defined here by the predicate **scorres**, which relates a shallowly embedded expression s to a deeply embedded one e when evaluated under the environment V:

Definition 5.2 (Shallow→Deep correspondence)**.**

$$\textbf{\textit{scorres}}\ s\ e\ V \quad = \quad \forall v.\ V \vdash e \downarrow_{\mathsf{v}} v \Rightarrow (s, v) \in \mathscr{R}_{\mathsf{S}}$$

Here, $\mathscr{R}_{\mathsf{S}}$ is a value relation*, much like the value relation $\mathscr{R}_{val}$ for* ***corres*** *refinement, connecting HOL and Cogent values. Just as with the* ***corres*** *refinement, the relation $\mathscr{R}_{\mathsf{S}}$ is defined incrementally, using Isabelle's ad-hoc overloading mechanism. The automated tactic for* ***scorres*** *theorems is substantially simpler than the tactic for* ***corres****, as* ***scorres*** *rules do not require well-typedness, nor do they involve any mutable state or the state relation $\mathscr{R}$. The tactic proceeds simply by applying specially-crafted introduction rules one by one, which correspond exactly to each form of A-normal Cogent syntax.*

The program-specific refinement theorem produced by our tactic is:

Generated Theorem 5.4 (Shallow to Deep refinement)**.** *Let f be the name of an A-normal Cogent function where $defnOf(f) = \lambda x.\ e$ and let s be the shallow embedding of f. Then, for any $(v_s, v) \in \mathscr{R}_{\mathsf{S}}$, we have* **scorres** $(s\ v_s)\ e\ (x \mapsto v)$*. The definition of $\mathscr{R}_{\mathsf{S}}$ ensures that v_s and v are of matching types.*

5.3.5 Desugared and neat embeddings

Figure 26 depicts the top-level *neat* embedding for the example presented previously in Figure 23. As can be seen, the Isabelle definitions use the same names and structure as the original Cogent program, making it easy for the user to reason about. In addition to the neat embedding, the compiler also produces a *desugared* shallow embedding, which does not resemble the input program as closely. For example, pattern matching is split into a series of binary **case** expressions. Lastly, the compiler also produces an A-normal shallow embedding, which resembles the A-normal intermediate representation of the code, as seen in Figure 23.

```
record α T =
  f :: α
definition
  bnot :: (8 word) T ⇒ (8 word) T
where
  bnot x =
    let (x', y) = take_f x
    in if (y = 0)
      then x' (| f = 1 |)
      else x' (| f = 0 |)
```

Fig. 26. "Neat" shallow embedding of the program from Figure 23.

Because we are now on the level of purely functional shallow embeddings, the proofs connecting the neat embedding to desugared embedding, and the desugared embedding to the A-normal equivalent, are significantly stronger than refinement — Instead, we prove equality. In Isabelle/HOL, equality is defined based on $\alpha\beta\eta$-equivalence, which means that this notion of equality admits the principle of functional extensionality.

Generated Theorem 5.5 (Neat and A-Normal equality)**.** *Let s_D be the desugared shallow embedding and s_A be the A-normal shallow embedding of a Cogent function. Then $s_D \overset{\alpha\beta\eta}{=} s_A$.*

Generated Theorem 5.6 (Neat and Desugared equality)**.** *Let s_N be the neat shallow embedding and s_D be the desugared shallow embedding of a Cogent function. Then $s_N \overset{\alpha\beta\eta}{=} s_D$.*

The proofs of these theorems are simple to generate. Since we can now use equational reasoning with Isabelle's powerful rewriter, we just unfold definitions on both sides, apply extensionality, and the rest of the proof is automatic given the right congruence rules and equality theorems for functions lower in the call graph.

5.3.6 Combined predicate for full refinement

To show that the top-level neat shallow embedding is a valid abstraction of the C code, the individual refinement certificates presented in the previous sections (Generated Theorems 5.2, 5.3, 5.4, 5.5 and 5.6) are not sufficient. We must also show that the individual refinement relations for each of these stages compose together, producing an overall proof of refinement across the entire chain.

We define our combined predicate **correspondence** connecting a top-level shallow embedding s, a monomorphic deep embedding e of type τ, and an AutoCorres-produced monadic embedding P. It is also parameterised by the C state relation $\mathscr{R}$, the monomorphisation renaming function θ, the update and value semantics environments U and V for the deeply embedded expression e, as well as its typing context Γ, the Cogent store μ and the AutoCorres state σ.

Definition 5.3 (Correspondence)**.**

$$
\begin{aligned}
&\textbf{\textit{correspondence}}\ \theta\ \mathscr{R}\ s\ e\ \tau\ P\ U\ V\ \Gamma\ \mu\ \sigma = \\
&\quad (\exists r\ w.\ U \mid \mu : V : \Gamma\ \ [\, r * w \,]) \wedge (\mu, \sigma) \in \mathscr{R} \\
&\Rightarrow \neg \mathit{failed}\ (P\ \sigma) \wedge\ \ \forall (v_C, \sigma') \in \mathit{results}\ (P\ \sigma). \\
&\qquad\qquad \exists \mu'\ u\ v.\quad U \vdash e \mid \mu \Downarrow_u u \mid \mu' \\
&\qquad\qquad\qquad \wedge\quad V \vdash e \Downarrow_v \mathscr{M}_\theta^v\ v \\
&\qquad\qquad\qquad \wedge\quad (\mu', \sigma') \in \mathscr{R} \wedge (u, v_C) \in \mathscr{R}_{val} \\
&\qquad\qquad\qquad \wedge\quad (\exists r\ w.\ u \mid \mu' : \mathscr{M}_\theta^v\ v : \tau\ \ [\, r * w \,]) \\
&\qquad\qquad\qquad \wedge\quad (s, v) \in \mathscr{R}_{\mathsf{S}}
\end{aligned}
$$

Observe that this definition is essentially the combination of our semantic shift preservation theorem (i.e. Theorem 4.1) with the refinement predicates ***corres*** *(Definition 5.1) and* ***scorres*** *(Definition 5.2).*

Intuitively, our top-level theorem states that for related input values, all programs in the refinement chain evaluate to related output values, propagating up the chain according to the intuitive forward-simulation method of de Roever & Engelhardt (1998). 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.

Generated Theorem 5.7 (Overall Refinement)**.** *For a Cogent function f, let $\mathit{defnOf}(f) = \lambda x.\ e$ and s be the shallow embedding of f. Let f_m be the monomorphised version of f according to renaming function θ, where $\mathit{typeOf}(f_m) = \tau \to \rho$, and P is the monadic embedding of the generated C for f_m.*

Then, we can show that for related input values v_S, v, u and v_C for the pure shallow embedding, value semantics, update semantics and monadic embedding respectively, our ***correspondence*** *predicate holds:*

$$
\begin{aligned}
\forall \mu\ \sigma.\quad & (v_S, v) \in \mathscr{R}_{\mathsf{S}} \\
\wedge\quad & (\exists r\ w.\ u \mid \mu : \mathscr{M}_\theta^v\ v : \tau\ \ [\, r * w \,]) \\
\wedge\quad & (u, v_C) \in \mathscr{R}_{\mathrm{val}} \\
\Rightarrow\quad & \textbf{\textit{correspondence}}\ \theta\ \mathscr{R}\ (s\ v_S)\ (\mathscr{M}_\theta\ e)\ \rho\ (P\ v_C)\ (x \mapsto u)\ (x \mapsto v)\ (x : \tau)\ \mu\ \sigma
\end{aligned}
$$

The automatic proof of this theorem is straightforward, merely unfolding the definitions of **corres** and **scorres** in Generated Theorems 5.2 and 5.4, applying Generated Theorem 5.3 to establish the equivalence of the definition of f_m with $\mathscr{M}_\theta\ e$, and applying Theorem 4.3 to connect the value and update semantics.

Generated Theorems 5.6 and 5.5 show equality, not mere refinement, and thus they implicitly apply to our overall theorem, extending it to cover these high-level embeddings.

As previously mentioned, this theorem assumes that foreign functions adhere to their user-provided specification and their behaviour is unchanged when monomorphised. To fully verify a system implemented in Cogent and C, one needs to provide abstractions of the C code and manually prove that the C code respects the frame conditions similar to those ensured by Cogent's type system as well as refinement statements similar to those generated by the Cogent compiler. Cheung *et al.* (2021) provide such proofs for the C implementation of *fixed-length word arrays* used in the `ext2` and BilbyFS implementations.

Word arrays are specified as Isabelle/HOL lists and *iterators over the array* are specified as *map accumulate* and *fold* functions over lists. These abstractions and proofs are used to discharge assumptions about foreign functions generated by the Cogent compiler. This demonstrates that Cogent's foreign function interface provides a modular cross-language approach to proving refinement between Cogent, a safe functional language, and C, an unsafe imperative language. Cogent's foreign function interface ensures safe and correct interoperability between the two languages.

5.4 Connecting to abstract specifications

Generated Theorem 5.7 shows that, assuming that the refinement relation holds initially, that the C functions are appropriately verified, and that our SIMPL C semantics accurately capture the semantics of the executed code, any functional correctness property we prove about the neat shallow embedding applies just as well to our C implementation. We stipulate *functional correctness* properties here, as other properties, such as security or timing properties, are not necessarily preserved by refinement.

To prove functional correctness, we must first define a functional correctness specification. This specification can take a variety of forms, but must essentially capture the externally observable correctness requirements of the program, without concern for implementation details or performance. Typically, this specification is highly *non-deterministic*, to allow for abstraction from operational details of the program. For example, the seL4 refinement proof contains a number of layers of specification, where non-determinism increases in each layer up the refinement chain (Klein *et al.*, 2009). The Cogent file system verification of Amani *et al.* (2016) specifies each file system operation as a program in a set monad to model this non-determinism.

5.5 Example: verification of the `sync()` function in BilbyFs

To demonstrate how this high-level specification facilitates further formal reasoning at much reduced effort compared to traditional functional correctness verification as typified by e.g. seL4 (Klein *et al.*, 2009), we show the manual functional correctness proof of the `sync()` function in BilbyFs (Amani *et al.*, 2016). For more complete and detailed reports on the file system's design, verification, and our experience, we refer interested readers to our previous publications (Amani *et al.*, 2016; Amani, 2016; Amani & Murray, 2015).

5.5.1 Functional correctness specification

The goal is to show that the BilbyFs `sync()` operation implemented in Cogent is *functionally correct*, meaning that it behaves correctly in accordance with a *top-level*, abstract specification for this operation. We call this specification a *functional correctness specification*. It is short enough that a human can audit it to ensure that it accurately captures the intended behaviour.

The top-level specifications for `sync()` is depicted in Figure 27. `sync()` implements the corresponding functions expected of the Linux's virtual file system (VFS) layer. It synchronises the current in-memory state of the file system to physical storage. As BilbyFs

```
1  afs_sync afs ≡
2   if is_readonly afs then
3    return (afs, Error eRoFs)
4   else do
5    n ← select{0..length (updates afs)};
6    let updates = updates afs;
7      (toapply, rem) = (take n updates, drop n updates);
8      afs = (afs(|med := apply_updates toapply (med afs),
9                  updates := rem|));
10   in if rem = [] then
11    return (afs, Success ())
12   else do
13    e ← select {eIO, eNoMem, eNoSpc, eOverflow};
14    return (afs(|is_readonly := (e = eIO)|), Error e)
15   od
16  od
```

Fig. 27. Functional correctness specifications for `sync()`

buffers pending writes in memory for better performance, the in-memory state may be temporarily out-of-sync with the physical state. The top-level abstract file system (AFS) specification for `sync()`, afs_sync, operates over the abstract file system state *afs*. The *afs* state is a record, consisting of (1) a map from inode numbers to inode objects, which tracks the state of the physical storage medium (med); (2) a list of functions modifying the physical storage for pending in-memory medium updates (updates); (3) and a Boolean flag indicating whether the file-system is currently read-only (is_readonly). The specification says that `sync()` first checks whether the file system is read-only, in which case an appropriate Error code is returned with the file system state unchanged (lines 2 and 3). Otherwise, it applies the in-memory updates to the physical medium, with each update modelled as a function modifying the physical medium.

The specification is sufficiently non-deterministic to capture the behaviour of a correct file system under the situation when the in-memory updates are only partially applied, perhaps because of a flash device failure part-way through. For this reason, the specification allows any number of updates *n* (line 5) to succeed, between 0 and the total number of updates currently in-memory (i.e. length (updates *afs*)). It then (lines 8 and 9) applies the first *n* updates (*toapply*) to the physical medium med *afs*, and remembers the updates that remain to be applied *rem*. If all updates were applied, it returns Successfully, yielding the new file system state (lines 10 and 11). Otherwise (lines 12 to 14), it returns an appropriate error code, selected non-deterministically because the specification abstracts away from the precise reason *why* the failure might have occurred. In case of an I/O error (eIO), the file system is also put into read-only mode.

lemma *refine_sync* :
assumes *ref* : "*afs_fsop_rel afs fs_st*"
shows
" ⋀ *ex*. *cogent_corres rsync_res* (*afs_sync afs*) (*fsop_sync_fs* (*fs_st*))"

Fig. 28. Refinement lemma for the `sync()` operation.

5.5.2 Functional correctness proof

We prove the correctness of the BilbyFs implementation of the `sync()` operation in a modular fashion against the top-level specifications in Figure 27. BilbyFs is designed with formal verification in mind and features a highly modular design. Thus the proof can follow the modular decomposition of its implementation. To prove the `sync()` implementation refines its functional correctness specification, we make assumption about each of the modules `sync()` depends on. These assumptions form an *axiomatic* specification of the respective module, and serves as a compact representation of its correctness that abstracts away its implementation details.

For `sync()`, one such interfacing module is ObjectStore, which keeps track of a mapping from object identifiers to generic file system objects. To prove ObjectStore correct, we follow the same approach by proving it correct with respect to its top-level abstract specification based on the assumptions on its dependencies. The proof eventually bottoms out at components that are entirely abstract, only captured by an axiomatic specification. The validity of the entire functional correctness proof then rests on the validity of the these axioms. In our file systems, these are hardware or other trusted components of the operating system.

In the Isabelle/HOL development, the refinement (by forward simulation) lemma Figure 28 for the `sync()` function is very standard (c.f. Section 5.1): if the correspondence relation *afs_fsop_rel* holds between the abstract file system state *afs* and the Cogent state *fs_st*, then after applying the abstract function *afs_sync* and the Cogent function *fsop_sync_fs* respectively, then the results are also related by *rsync_res*.

With the modular verification strategy, our proof script for `sync()` in Isabelle/HOL consists of approximately 60 lines of code, among which most are simply unfolding definitions and simplification rules, which is exactly what is expected of equational reasoning. [5]

5.5.3 Proving invariants

As we have demonstrated above, these proofs are far simpler than e.g. the comparable functional correctness proofs of seL4 (Klein *et al.*, 2009), which establish similar properties. Just as with seL4, the functional correctness proof here requires us to establish global invariants about the abstract specification and its implementation. We now showcase the proof of the invariants that are needed in the refinement proof above.

The lemma *afs_inv_steps_updated_afsD* in Figure 29 states that if the invariants (*afs_inv*) hold when any prefix of the list of pending updates in *afs* is applied (*afs_inv_steps*), then they also hold when the list of updates are fully applied. In this case, the invariants include

[5] The line count does not include generic lemmas from HOL or those about common data structures, nor does it include the few auxiliary lemmas needed for *afs*, which can be proved by the Isabelle/HOL's simplifer (e.g. Section 5.5.3).

```
definition
 "afs_inv_steps afs ≡
  (∀n ≤ length (a_medium_updates afs).
    afs_inv (a_afs_updated_n n (a_medium_afs afs) (a_medium_updates afs)))"

lemma afs_inv_steps_updated_afsD : "afs_inv_steps afs ⟹ afs_inv (updated_afs afs)"
 apply (simp add : updated_afs_def)
 apply (simp add : afs_inv_steps_def)
 apply (erule_tac x = "length (a_medium_updates afs)" in allE)
 apply (fastforce simp add : a_afs_updated_def)
done
```

Fig. 29. Invariants that need to be maintained before and after applying the pending updates.

e.g. the absence of link cycles, dangling links and the correctness of link counts, as well as the consistency of information that is duplicated in the file system for efficiency.

Importantly, unlike with seL4, none of the invariants have to include that in-memory objects do not overlap, or that object-pointers are correctly aligned and do point to valid objects. All of these properties are ensured automatically by Cogent's type system, and justified by Generated Theorem 5.7. Even better, when proving that the file system correctly maintains its invariants, we get to reason over pure, functional specifications of the Cogent code. Because they are pure functions, these specifications do not deal with mutable state (as e.g., the seL4 ones do). The proof can be done simply by unfolding definitions (as shown in Figure 29 by the simp add : $*\_def$ rules).

5.6 Subtyping and refinement framework

The introduction of subtyping to Cogent's type system required adapting our original Cogent compiler, formalisation, and refinement framework (O'Connor *et al.*, 2016) to account for this addition. As mentioned in Section 3.1, due to subtyping the compiler no longer needs to generate separate data types for each narrowing of a variant type. Thus, this feature drastically reduced the number of data types in the generated C code and the generated shallow embedding. For the the Bilby file system, Cogent previously generated 49 separate data type definitions; with subtyping, this has been reduced to 7.

This makes the automated refinement proof of correspondence between Cogent code and the generated shallow embedding much clearer, and also simplifies the shallow embedding. In addition conversion functions between variants are no longer generated in the C code and shallow embedding. Re-proving the correctness of the BilbyFs operations that we previously verified on top of the new shallow embedding did not require much effort. The manual proofs on top of the shallow embedding have not increased in complexity as a result of our change.

6 Conclusions, evaluation, and future work

Our work has already shown promising results, both as a systems programming language and a verification target, however the file system implementations and verification conducted as a case-study (Amani, 2016; Amani *et al.*, 2016) bring several opportunities into focus for future improvements to our framework.

6.1 The Cogent toolchain

Our decision to write the Cogent compiler tool-chain in Haskell but the refinement framework and proof tactics in Isabelle/ML allows the Cogent tool-chain to be used outside the theorem prover, while still allowing our refinement framework to build on the existing C and AutoCorres framework available in Isabelle/HOL.

On the other hand, this choice leads to some complexity in designing the interface between these components. This is illustrated by the well-typedness proof in Section 5.2, where the Cogent compiler generates a certificate tree with the necessary type derivation hints. Initially, a naïve format consisting of the entire derivation tree was used, resulting in gigabyte-sized certificates. Various compression techniques reduced this to a reasonable size (a few megabytes), but these certificates still take some time to process. It would be possible to avoid these certificates entirely by duplicating the entire type inference algorithm from the compiler in Isabelle/ML, but this would increase the code maintenance burden significantly.

The use of pre-existing mature tools to give C code a semantics in Isabelle/HOL, namely the C→SIMPL parser and AutoCorres, is a pragmatic choice aimed at reducing the effort required to build our refinement framework, ensuring that our C semantics lines up with other large-scale C verification projects, and enabling integration with the seL4 verification specifically. Unfortunately, however, these tools are particularly time-consuming when processing Cogent-generated C code. For the file system implementations of Amani *et al.* (2016), these tools take anywhere from 12 to 32 CPU hours to generate the monadic embedding of the generated C code. While the time taken to establish our refinement certificate does not endanger the trustworthiness of Cogent software, it does make our automatic verification framework less useful as a debugging tool. Future work involves integrating robust specification-based testing tools in the style of QuickCheck (Claessen & Hughes, 2000) to Cogent (Chen *et al.*, 2017), to improve turn-around time for debugging and to allow verification to be attempted only after developers are confident that the code is indeed correct.

6.2 Verification effort

Klein *et al.* (2009) report that approximately one third of the overall verification effort for seL4 went into the second refinement step, connecting the intermediate executable specification to the C code. This estimation is not including the effort that went into developing re-usable libraries and frameworks. Our Generated Theorem 5.7 encompasses this step and more, because, as previously discussed, our intermediate executable specification (the neat embedding) is significantly more high-level. Therefore, we can confidently predict that,

where Cogent can be used to implement a system, our refinement framework will reduce the effort of verifying that system by at least a third, relative to existing C verification techniques. Because our neat embedding is higher-level than the intermediate executable specification of seL4, the savings are possibly even greater.

In the course of the verification of two file system operations, we found six defects in our already-tested file system implementations (Amani *et al.*, 2016). The effort for verifying the complete file system component chain for these operations was roughly 9.25 person months, and produced roughly 13,000 lines of proof for the 1,350 lines of Cogent code. This compares favourably with traditional C-level verification as for instance in seL4, which spent 12 person years with 200,000 lines of proof for 8,700 source lines of C code. Roughly 1.65 person months per 100 C source lines in seL4 are reduced to ≈0.69 person months per 100 Cogent source lines with our framework. We are in the process of implementing a data-description language as an extension to Cogent (O'Connor *et al.*, 2018; Chen *et al.*, 2019), which will automate the functional correctness verification of approximately 850 lines of deserialisation and serialisation code in these file system implementations. These 850 lines of Cogent code required ≈4000 lines of proof to verify, taking approximately 4.5 person months. With this added automation, the cost of verification can be reduced even further. This data description language also enables us to reduce the amount of marshalling and unmarshalling code required to pass data structures between C and Cogent, as we can ensure that data is laid out in the same way in both languages. This can eliminate copies and improve efficiency of the generated code.

Another possible avenue to reduce the cost of functional correctness verification is to provide stronger static guarantees from the type system. The more properties we can encode in the type system and check automatically, the less will have to be manually established by a proof engineer in post-hoc verification.For this purpose, we plan to explore adding *refinement types* (Freeman & Pfenning, 1991) to Cogent. Refinement types allow specifying propositions on types and may therefore help us track that array indices are within bounds and thus memory safety once arrays are introduced to Cogent. More generally, refinement types have the potential to drastically reduce verification effort, depending on the expressive power of the refinements, as well as potentially reducing debugging turn-around time, as SMT push-button verification is faster than manual proof in an interactive theorem prover, although less powerful.

6.3 Safety and Security

Cogent's certificate goes beyond certifying the correctness of the generated C code relative to the HOL embedding. Even systems programmers *without any formal verification expertise* can statically eliminate whole classes of common errors that lead to security vulnerabilities (Amani *et al.*, 2016): The language is type safe. The compiler and type system in turn automatically ensure *memory safety* (de Amorim *et al.*, 2018), which ensures the absence of any undefined behaviour on the C level, null pointer dereferences, buffer overflows, memory leaks, and pointer mis-management in error handling.

File systems, which motivated the inception of Cogent, constitute the largest fraction of code in Linux after device drivers and have among the highest defect density of Linux kernel code (Palix *et al.*, 2011*b*). The mismanagement of pointers in error-handling code

is a wide-spread problem in Linux file systems specifically (Rubio-González & Liblit, 2011), and Saha *et al.* (2011) shows that file systems have among the highest density of error-handling code in Linux.

This problem isn't local to file systems: the "goto-fail" defect in Apple's SSL/TLS implementation was an error-handling problem obscured by `goto`s in an `if`-cascade (OpenSSL, 2014), and the memory leak in Android Lollipop also was part of error-handling code (Lollipop, 2014). By requiring programs to be expressed as pure and total functions, and enforcing correct cleanup and management of resources with uniqueness types, Cogent ensures that errors are correctly handled and all pointers and resources are correctly disposed of in an error scenario.

More generally, one of the most common sources of security vulnerabilities in software implemented in low-level languages such as C is memory corruption bugs (Szekeres *et al.*, 2013). The infamous Heartbleed bug for instance, was a buffer overflow (Heartbleed, 2014). All such vulnerabilities relating to the absence of memory safety are prevented on the language level by Cogent and are enforced by its certifying compiler.

Memory safety is intimately tied to the notion of *noninterference* (Goguen & Meseguer, 1982) which requires showing that programs can neither affect nor be affected by unreachable parts of the state (de Amorim *et al.*, 2018). The formal notion of memory safety defined by de Amorim *et al.* (2018) is similar to ours in that it supports local reasoning about state. In the case of Cogent, this requirement is demonstrated by the frame relation and other restrictions on the heap footprints of programs, key to proving that Cogent's imperative semantics can be abstracted by its functional semantics. While Cogent does not make many guarantees about foreign C code that is unverified, our frame relation already places verification requirements on foreign C code that enforce a kind of integrity: Any objects to which a function is not explicitly given access (i.e. any pointer outside the heap footprint) may not be modified by Cogent-compliant C code.

While Cogent's static guarantees rule out a large class of security vulnerabilities, Cogent does not provide constructs for defining and tracking security levels and for reasoning about *information-flow control*. One extension of Cogent that is under development is Flogent, which is a type system extension to Cogent that leverages uniqueness types to establish information-flow control (Dang, 2020). In the future, we plan to investigate preserving information flow security through compilation in the style of Covern (Sison & Murray, 2019).

6.4 Optimisations

Adding optimisation passes to the Cogent compiler would improve performance, but presents a verification challenge. Cogent-to-Cogent optimisations are straightforward to verify— the ease of proving A-normalisation correctness over the shallow embedding via rewriting suggests that this is the right approach in our context. Many optimisations are described as equational rewrites for functional languages, for example stream fusion (Coutts *et al.*, 2007). In particular, some of the source-to-source optimisations discussed by Chlipala (2015) seem promising for Cogent. However, introducing significant optimisations to the Cogent-to-C stage of our framework will complicate the syntax-directed correspondence approach described in Section 5.

Currently, the Cogent compiler relies primarily on the underlying C compiler for optimisations. 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, due to the A-normal representation used by the Cogent compiler, the generated C code is already quite close to single static assignment (SSA) form used internally by C compilers `gcc` and `clang`, however these compilers do not always recognise this and optimise accordingly. Generating a compiler's SSA representation directly, such as LLVM IR, may eliminate these problems. Shang (2020) has recently implemented a prototype of an LLVM backend for a core subset of Cogent. We plan to extend the LLVM backend to the full Cogent language and certify this compilation. Projects to verify subsets of LLVM IR exist for us to target (Zhao *et al.*, 2012; Lammich, 2019), however such an endeavor would imply significant and fundamental changes to our verification infrastructure. Thus, we plan to use this as opportunity to explore a new approach for establishing compiler assurance.

There are two main approaches to establishing compiler assurance (Leroy, 2009*a*): *compiler verification* (Kumar *et al.*, 2014; Leroy, 2009*b*), and *translation validation* (Pnueli *et al.*, 1998*a*) through either a *verified validator* (Rideau & Leroy, 2010) or by synthesising a proof of correctness in a theorem prover after each translation, as we do in Cogent. Compiler verification is difficult to establish and costly to maintain, and, as previously mentioned, our current approach requires time-consuming proof checking for every compilation (Rizkallah *et al.*, 2016). Translation validation through a verified validator provides no guarantees when the validator rejects (Leroy, 2009*a*). An ideal approach would combine several of the benefits of the existing approaches. For our LLVM backend, we plan to explore a new verification approach that draws inspiration from a line of work on creating trustworthy *certifying algorithms* (Blum & Kannan, 1995; Sullivan & Masson, 1990; McConnell *et al.*, 2011) using *verified checkers* (Bright *et al.*, 1997; Alkassar *et al.*, 2014; Noschinski *et al.*, 2014; Rizkallah, 2015).

6.5 Language features

We are currently working to remove limitations of the language, to make Cogent more convenient to use, and to enable more kinds of code to be written and verified with Cogent. One of the most glaring limitations of Cogent is the intentional absence of recursion, to ensure that all Cogent programs terminate. We are working on relaxing these limitations by introducing recursive types and functions (Murray, 2019) as well as a limited form of arrays to Cogent. To maintain our termination guarantee, we are implementing termination checking algorithms in the compiler front-end.

Another clear avenue for extension is support for concurrency. Session types (Dezani-Ciancaglini & de'Liguoro, 2010), intended to describe concurrent systems, could be cleanly integrated into Cogent's uniqueness type system. Verifying a concurrent Cogent would also necessitate a verified concurrent semantics for each level in our refinement chain, including for C. While there is a concurrent version of SIMPL, based on the foundational Owicki/Gries method (Owicki & Gries, 1976), called COMPLX (Amani *et al.*, 2017), it is intended for verification of low-level, potentially racy code and may therefore not be suitable for our purpose. Connecting to this low-level semantics, or developing a higher-level semantics based on more recent methods will be a significant endeavour.

We are also still investigating improvements to our type inference algorithm (see O'Connor (2019) for an initial formalisation) as well as proving soundness of this type inference process in Isabelle/HOL. As Cogent involves a number of language features that complicate type inference (subtyping, uniqueness types, and our unique structural record and variant types), our algorithm is quite involved and its verification is ongoing work.

6.6 Related Work

6.6.1 Safe languages

Formally verified full-scale language implementations include the verified C compiler CompCert (Leroy, 2009*b*), the verified ML compiler CakeML (Kumar *et al.*, 2014), Verisoft's implementation language C0 (Leinenbach, 2010), and Dafny (Rustan & Leino, 2010). All of these verified language implementations do not provide the functional abstraction over mutation that Cogent does, which is crucial for providing a clean equational interface for reasoning about high-level specifications. Additionally, they either come with a run-time and garbage collector (as in CakeML or Dafny), or they provide only weak type system guarantees (as in C or C0). The key novelty in Cogent is that its verification gives high-level type system benefits and a strong theorem proving interface without the need to include a garbage collector in low-level systems code.

Similarly, while type safe C dialects such as Cyclone (Jim *et al.*, 2002) or CCured (Necula *et al.*, 2005), and programming languages like Rust (Rust, 2014), can guarantee memory safety using types without depending on a garbage collector, they do not raise the level of abstraction as Cogent does. As these are imperative programming languages, they do not use their type systems to provide a functional abstraction of mutation, but merely as a means to track the lifetime of heap objects. Therefore, these type systems are less restrictive than that of Cogent. Rust and Cyclone additionally have a much more fine-grained notion of lifetimes than Cogent, similar to region types (Tofte & Talpin, 1994), which may be worth integrating into Cogent in future.

There are many tools to generate shallow embeddings from functional code, such as CFML (Charguéraud, 2010, 2011) and `hs-to-coq` (Spector-Zabusky *et al.*, 2018; Breitner *et al.*, 2021). Like us, these generate shallow embeddings to facilitate mechanical proofs, but unlike us they do not prove correctness of compilation.

The Ivory language (Pike *et al.*, 2014) is a strongly-typed domain-specific language embedded in Haskell for systems programming, in particular for writing programs that interact directly with hardware and do not require dynamic memory allocation. It presently has a formal semantics, but no compiler correctness proof. Its intended domain is close but separate from Cogent. Cogent is less aimed at interacting directly with hardware, but more for high-level control code and does support dynamic memory allocation and tracking of dynamic memory with its linear type system.

Like Cogent, the programming language developed as part of the HASP project, Habit (HASP project, 2010), is a functional systems language. It has a verified garbage collector (McCreight *et al.*, 2010), but no formal language semantics or compilation certificate.

The Rosette language (Torlak & Bodik, 2014) is a rapid prototyping environment for domain-specific languages, including support for solver-aided verification and synthesis, but not as such for efficient and verified compilation to standalone systems code.

Linear types have seen growing use, with extensions being developed for Haskell (Bernardy *et al.*, 2017) and Idris (Brady, 2013). PacLang (Ennals *et al.*, 2004) uses linear types to guide optimisation of packet processing on network processors. Uniqueness types are integrated into the functional language Clean (Barendsen & Smetsers, 1993), although Clean still depends on run-time garbage collection.

Section 4 mentions that 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.

6.6.2 Safe file systems

Functional verification of file systems belongs to systems verification in general. Klein *et al.* (2017) gives a more thorough overview of the work in this area. Some major achievements are the comprehensive verification of the seL4 microkernel (Klein *et al.*, 2009), the verification stack of the Verisoft project (Alkassar *et al.*, 2009, 2010), the increase of verification productivity in CertiKOS (Gu *et al.*, 2011, 2015), and the full end-to-end application verification in Ironclad (Hawblitzel *et al.*, 2014), which builds on a modified verified Verve kernel (Yang & Hawblitzel, 2010) and the aforementioned Dafny language.

Early Z specifications of file systems are those by Morgan & Sufrin (1984) for UNIX, and Bevier *et al.* (1995) for a custom file system. Arkoudas *et al.* (2004) verify two key operations on the block-level of a file system, but the result remains partial and the authors even argue that system components such as file systems will probably always remain beyond the reach of full correctness proofs.

There is plenty of previous work containing proofs about high-level abstractions of file systems. Our work closes the gap from these high level abstractions to code. For instance, Hesselink and Lali (Hesselink & Lali, 2009) manage to prove a file refinement stack that goes down to a formal model, but assumes an infinite storage device and other simplifications, and does not end in code. The Event B refinement proof by Damchoom et al. (Damchoom *et al.*, 2008) similarly does not end in code. In theory, Event B can generate code from low-level models, but neither of these verifications are close enough to achieve usable file system implementations, let alone high performance.

The most realistic high-level Flash file system verification work to date is conducted using the KIV tool (Reif *et al.*, 1998) and goes from the Flash device layer up to a Linux VFS implementation (Schierl *et al.*, 2009; Ernst *et al.*, 2013; Schellhorn *et al.*, 2014). The verification work is still in progress and the current code generation from low-level models targets Scala running on a Java Virtual Machine, which implies run-time overheads and dependency on a large language run-time. It may be fruitful to investigate a Cogent backend for this work.

Maric & Sprenger (2014) investigate the issue of crash tolerance in file systems, and previously Andronick (2006) formally analysed similar issues for tearing in smart cards with persistent storage. Cogent does not provide any special handling for crash tolerance, but the generated executable specifications are detailed enough to facilitate reasoning about it. Chen *et al.* (2015) take crash tolerance to the level of a complete file system in a proof that includes functional correctness of an implementation in Coq. 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 specification and low-level implementation, 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 potentially straightfoward to implement Crash Hoare Logic on top of Isabelle Cogent executable specifications, enabling a verification of crash tolerance for Cogent file systems.

Another stream of work in the literature focuses on more automatic techniques such as model checking and static analysis (Yang *et al.*, 2006; Gunawi *et al.*, 2008; Rubio-González & Liblit, 2011). While in theory these techniques could be used to provide similar guarantees as Cogent, this has not yet been achieved in practice. Instead of providing guarantees, such analyses are more useful as tools for efficiently finding defects in existing implementations. They also do not provide a path to further higher-level reasoning.

6.7 Conclusion

Cogent has achieved its stated goal: To reduce the cost of formally verifying functional correctness of low-level operating systems components. It achieves this by allowing users to write code at a high level of abstraction, in the native language of interactive proof assistants — purely functional programming.

Functional programmers have long recognised, and advocated for, the benefits afforded by reasoning over pure functions. For the first time, Cogent allows these benefits to be enjoyed by proof engineers verifying low-level systems, without depending on a run-time system or enlarging the trusted computing base. Building on the key refinement theorem given to us by our uniqueness type system (Theorem 4.3), our refinement framework makes use of multiple translation validation techniques to establish a long refinement chain. This allows engineers to reason about Cogent code on a high level in Isabelle/HOL and have confidence that their reasoning applies just as well to the C implementation we generate.

To interact with existing systems and to enable greater expressivity, we include a foreign function interface that allows the programmer to mix Cogent code with C code. It is possible to verify C code and compose these proofs with the proofs generated by the Cogent framework.

Our two case study file systems serve to validate our approach, with key operations of one file system verified for functional correctness. The results from these studies confirm our hypothesis: the language ensures a greater degree of reliability by default compared to C programming, verification effort is reduced by at least one third, and the performance of the generated code is, while slower, still comparable to native C implementations, and acceptable for realistic file system implementations.

Cogent not only allows non-experts in formal verification to write provably-safe code, it is also a key step towards lowering the effort and complexity for the full mechanical verification of operating system components against high-level formal specifications. It is a significant milestone, bringing the grand goal of affordable, verified, high-assurance systems one step closer to reality.

Acknowledgements

We are indebted to Amos Robinson for his extensions to our type inference algorithm and compiler; to Yutaka Nagashima and Japheth Lim for their help with our refinement framework; to Joseph Tuong and Sean Seefried for their work on generating Isabelle proofs from Haskell; and to Partha Susarla, Peter Chubb and Alex Hixon for their work on systems programming in Cogent.

Conflicts of Interest

None.

References

Alkassar, E., Hillebrand, M., Leinenbach, D., Schirmer, N., Starostin, A. and Tsyban, A. (2009) Balancing the load — leveraging a semantics stack for systems verification. *Journal of Automated Reasoning: Special Issue on Operating System Verification* **42, Numbers 2–4**:389–454.

Alkassar, E., Paul, W., Starostin, A. and Tsyban, A. (2010) Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. *Verified Software: Theories, Tools and Experiments*. Lecture Notes in Computer Science 6217, pp. 71–85. Springer.

Alkassar, E., Böhme, S., Mehlhorn, K. and Rizkallah, C. (2014) A framework for the verification of certifying computations. *Journal of Automated Reasoning* **52**(3):241–273.

Amani, S. (2016) *A methodology for trustworthy file systems*. PhD thesis, University of New South Wales.

Amani, S. and Murray, T. (2015) Specifying a realistic file system. *Workshop on Models for Formal Analysis of Real Systems* pp. 1–9.

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. and Heiser, G. (2016) Cogent: Verifying high-assurance file system implementations. *Architectural Support for Programming Languages and Operating Systems* pp. 175–188.

Amani, S., Andronick, J., Bortin, M., Lewis, C., Rizkallah, C. and Tuong, J. (2017) COMPLX: a verification framework for concurrent imperative programs. *Certified Programs and Proofs* pp. 138–150. ACM.

Andronick, J. (2006) Formally Proved Anti-tearing Properties of Embedded C Code. *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* pp. 129–136. Invited Speaker.

Arkoudas, K., Zee, K., Kuncak, V. and Rinard, M. C. (2004) Verifying a file system implementation. *Conference on Formal Engineering Methods*. Lecture Notes in Computer Science 3308, pp. 373–390. Springer.

Barendsen, E. and Smetsers, S. (1993) Conventional and uniqueness typing in graph rewrite systems. *Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science 761, pp. 41–51.

Beringer, L., Petcher, A., Ye, K. Q. and Appel, A. W. (2015) Verified correctness and security of OpenSSL HMAC. *Security Symposium* pp. 207–221. USENIX.

Bernardy, J.-P., Boespflug, M., Newton, R. R., Peyton Jones, S. and Spiwack, A. (2017) Linear Haskell: Practical linearity in a higher-order polymorphic language. *Principles of Programming Languages* pp. 5:1–5:29. ACM.

Bevier, W., Cohen, R. and Turner, J. (1995) *A Specification for the Synergy File System*. Tech. rept. Technical Report 120. Computational Logic Inc., Austin, Texas, USA.

Blum, M. and Kannan, S. (1995) Designing programs that check their work. *Journal of the ACM* **42**(1):269–291.

Brady, E. (2013) Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* **23**(9):552–593.

Breitner, J., Spector-Zabusky, A., Li, Y., Rizkallah, C., Wiegley, J., Cohen, J. and Weirich, S. (2021) Ready, Set, Verify! Applying hs-to-coq to real-world Haskell code. *Journal of Functional Programming* **31**.

Bright, J. D., Sullivan, G. F. and Masson, G. M. (1997) A formally verified sorting certifier. *IEEE Transactions on Computers* **46**(12):1304–1312.

Charguéraud, A. (2010) Program verification through characteristic formulae. *International Conference on Functional Programming* pp. 321–332. ACM.

Charguéraud, A. (2011) Characteristic formulae for the verification of imperative programs. *International Conference on Functional Programming* pp. 418–430. ACM.

Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F. and Zeldovich, N. (2015) Using Crash Hoare logic for certifying the FSCQ file system. *Symposium on Operating Systems Principles* pp. 18–37. ACM.

Chen, Z., O'Connor, L., Keller, G., Klein, G. and Heiser, G. (2017) The Cogent case for property-based testing. *Programming Languages and Operating Systems* pp. 1–7. ACM.

Chen, Z., Di Meglio, M., O'Connor, L., Susarla Ajay, P., Rizkallah, C. and Keller, G. (2019) *A Data Layout Description Language for Cogent (extended abstract)*. http://ssrg.nicta.com/publications/csiro_full_text/Chen_DOSRK_19.pdf. At Principles of Secure Compilation (PriSC).

Cheung, L., O'Connor, L. and Rizkallah, C. (2021) Overcoming restraint: Modular refinement using cogent's principled foreign function interface. *CoRR* **abs/2102.09920**. Under submission.

Chlipala, A. (2015) An optimizing compiler for a purely functional web application language. *International Conference on Functional Programming* pp. 10–21. ACM.

Claessen, K. and Hughes, J. (2000) Quickcheck: A lightweight tool for random testing of haskell programs. *International Conference on Functional Programming* pp. 268–279. ACM.

Cock, D., Klein, G. and Sewell, T. (2008) Secure microkernels, state monads and scalable refinement. *International Conference on Theorem Proving in Higher Order Logics*, vol. 5170, pp. 167–182. Springer-Verlag.

Cogent. *Cogent source code and Isabelle/HOL formalisation*. https://github.com/NICTA/cogent. Accessed: 2021-03-11.

Coquand, T. and Paulin, C. (1988) Inductively defined types. *International Conference on Computer Logic* pp. 50–66.

Coutts, D., Leshchinskiy, R. and Stewart, D. (2007) Stream fusion: From lists to streams to nothing at all. *International Conference on Functional Programming* pp. 315–326. ACM.

Damchoom, K., Butler, M. and Abrial, J.-R. (2008) Modelling and proof of a tree-structured file system in Event-B and Rodin. *Conference on Formal Engineering Methods*. Lecture Notes in Computer Science 5256, pp. 25–44. Springer.

Dang, V. (2020) *Flogent: An Information Flow Security Feature for Cogent*. Honours thesis, UNSW Sydney, Computer Science & Engineering.

de Amorim, A. A., Hritcu, C. and Pierce, B. C. (2018) The meaning of memory safety. Bauer, L. and Küsters, R. (eds), *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Lecture Notes in Computer Science 10804,

pp. 79–105. Springer.
de Roever, W. P. and Engelhardt, K. (1998) *Data Refinement: Model-oriented Proof Theories and their Comparison*. Cambridge Tracts in Theoretical Computer Science, vol. 46. Cambridge University Press.
de Vries, E., Plasmeijer, R. and Abrahamson, D. M. (2008) Uniqueness typing simplified. *Implementation and Application of Functional Languages*.
DeLine, R. and Fähndrich, M. (2001) Enforcing high-level protocols in low-level software. *Programming Language Design and Implementation* pp. 59–69.
Dezani-Ciancaglini, M. and de'Liguoro, U. (2010) Sessions and session types: An overview. *Web Services and Formal Methods* pp. 1–28. Springer.
Dijkstra, E. W. (1997) *A Discipline of Programming*. 1st edn. Prentice Hall PTR.
Ennals, R., Sharp, R. and Mycroft, A. (2004) Linear types for packet processing. *European Symposium on Programming* pp. 204–218. Springer.
Ernst, G., Schellhorn, G., Haneberg, D., Pfähler, J. and Reif, W. (2013) Verification of a virtual filesystem switch. *Verified Software: Theories, Tools and Experiments*. Lecture Notes in Computer Science 8164, pp. 242–261. Springer.
Freeman, T. and Pfenning, F. (1991) Refinement types for ml. *Programming Language Design and Implementation* pp. 268–277. ACM.
Girard, J.-Y. (1971) Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. *Scandinavian Logic Symposium* pp. 63–92. North-Holland.
Goguen, J. A. and Meseguer, J. (1982) Security policies and security models. *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982* pp. 11–20. IEEE Computer Society.
Greenaway, D., Andronick, J. and Klein, G. (2012) Bridging the gap: Automatic verified abstraction of C. *International Conference on Interactive Theorem Proving* pp. 99–115. Springer.
Greenaway, D., Lim, J., Andronick, J. and Klein, G. (2014) Don't sweat the small stuff: Formal verification of C code without the pain. *Programming Language Design and Implementation* pp. 429–439. ACM.
Gu, L., Vaynberg, A., Ford, B., Shao, Z. and Costanzo, D. (2011) CertiKOS: A certified kernel for secure cloud computing. *Asia-Pacific Workshop on Systems*.
Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X. N., Weng, S., Zhang, H. and Guo, Y. (2015) Deep specifications and certified abstraction layers. *Principles of Programming Languages* pp. 595–608. ACM.
Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V. and Costanzo, D. (2016) CertiKOS: An extensible architecture for building certified concurrent OS kernels. *Operating Systems Design and Implementation*. ACM.
Gunawi, H. S., Rajimwale, A., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. (2008) Sqck: A declarative file system checker. *Operating Systems Design and Implementation* p. 131–146. USENIX Association.
Harper, R. and Morrisett, G. (1995) Compiling polymorphism using intensional type analysis. *Principles of Programming Languages* pp. 130–141. ACM.
HASP project. (2010) *The Habit Programming Language: The Revised Preliminary Report*. Tech. rept. http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf. Department of Computer Science, Portland State University, Portland, OR, USA.
Hawblitzel, C., Howell, J., Lorch, J. R., Narayan, A., Parno, B., Zhang, D. and Zill, B. (2014) Ironclad apps: End-to-end security via automated full-system verification. *Operating Systems Design and Implementation* pp. 165–181.
Heartbleed. (2014) *The Heartbleed Bug*. http://heartbleed.com, https://www.openssl.org/news/secadv_20140407.txt. Accessed March 2015.
Hesselink, W. H. and Lali, M. I. (2009) Formalizing a hierarchical file system. *BCS-FACS Refinement Workshop*. Electronic Notes in Theoretical Computer Science 259, pp. 67–85.
Hofmann, M. (2000) A type system for bounded space and functional in-place update. *European Symposium on Programming*. Lecture Notes in Computer Science 1782, pp. 165–179.

Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J. and Wang, Y. (2002) Cyclone: A safe dialect of c. *USENIX Annual Technical Conference* pp. 275–288. USENIX.

Keller, G., Murray, T., Amani, S., O'Connor, L., Chen, Z., Ryzhyk, L., Klein, G. and Heiser, G. (2013) File systems deserve verification too! *Programming Languages and Operating Systems* pp. 1–7.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S. (2009) seL4: Formal verification of an os kernel. *Symposium on Operating Systems Principles* pp. 207–220. ACM.

Klein, G., Andronick, J., Keller, G., Matichuk, D., Murray, T. and O'Connor, L. (2017) Provably trustworthy systems. *Philosophical Transactions of the Royal Society A* **375**(September):1–23.

Kumar, R., Myreen, M., Norrish, M. and Owens, S. (2014) CakeML: A verified implementation of ML. *Principles of Programming Languages* pp. 179–191. ACM.

Lammich, P. (2019) Generating verified LLVM from Isabelle/HOL. *International Conference on Interactive Theorem Proving*. LIPIcs 141, pp. 22:1–22:19. Schloss Dagstuhl.

Leinenbach, D. (2010) *Compiler verification in the context of pervasive system verification*. PhD thesis, Saarland University.

Leroy, X. (2009a) Formal verification of a realistic compiler. *Communications of the ACM* **52**(7):107–115.

Leroy, X. (2009b) A formally verified compiler back-end. *Journal of Automated Reasoning* **43**(4):363–446.

Lollipop. (2014) *Memory Leak in Android Lollipop's rendering engine*. https://code.google.com/p/android/issues/detail?id=79729#c177. Accessed March 2015.

Maric, O. and Sprenger, C. (2014) Verification of a transactional memory manager under hardware failures and restarts. *International Symposium on Formal Methods*. Lecture Notes in Computer Science 8442, pp. 449–464. Springer.

Marlow, S. (2010) *Haskell 2010 Language Report*.

McConnell, R. M., Mehlhorn, K., Näher, S. and Schweitzer, P. (2011) Certifying algorithms. *Computer Science Review* **5**(2):119–161.

McCreight, A., Chevalier, T. and Tolmach, A. (2010) A certified framework for compiling and executing garbage-collected languages. *International Conference on Functional Programming* pp. 273–284. ACM.

Morgan, C. and Sufrin, B. (1984) Specification of the UNIX filing system. *IEEE Transactions on Software Engineering* **10**(2):128–142.

Murray, E. (2019) *Recursive Types For Cogent*. Honours thesis, UNSW Sydney, Computer Science & Engineering.

Necula, G. C., Condit, J., Harren, M., McPeak, S. and Weimer, W. (2005) CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems* **27**(3):477–526.

Nipkow, T., Wenzel, M. and Paulson, L. C. (2002) *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag.

Noschinski, L., Rizkallah, C. and Mehlhorn, K. (2014) Verification of certifying computations through autocorres and simpl. *NASA Formal Methods*. Lecture Notes in Computer Science 8430, pp. 46–61. Springer.

ntfs3g. *POSIX Filesystem Test Project*. http://sourceforge.net/p/ntfs-3g/pjd-fstest/ci/master/tree/. Retrieved Jan., 2016.

O'Connor, L. (2019) *Type Systems for Systems Types*. PhD thesis, University of New South Wales.

O'Connor, L., Chen, Z., Rizkallah, C., Amani, S., Lim, J., Murray, T., Nagashima, Y., Sewell, T. and Klein, G. (2016) Refinement through restraint: Bringing down the cost of verification. *International Conference on Functional Programming*.

O'Connor, L., Chen, Z., Susaria, P., Rizkallah, C., Klein, G. and Keller, G. (2018) Bringing effortless refinement of data layouts to Cogent. *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* pp. 134–149. Springer.

Odersky, M. (1992) Observers for linear types. *European Symposium on Programming* pp. 390–407. Springer-Verlag.

OpenSSL. (2014) *Apple's SSL/TLS bug*. https://www.imperialviolet.org/2014/02/22/applebug.html. Accessed March 2015.

Owicki, S. and Gries, D. (1976) An axiomatic proof technique for parallel programs. *Acta Informatica* **6**(4):319–340.

Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G. (2011a) Faults in Linux: Ten years later. *Architectural Support for Programming Languages and Operating Systems* pp. 305–318. ACM.

Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. L. and Muller, G. (2011b) Faults in Linux: ten years later. Gupta, R. and Mowry, T. C. (eds), *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011* pp. 305–318. ACM.

Pike, L., Hickey, P., Bielman, J., Elliott, T., DuBuisson, T. and Launchbury, J. (2014) Programming languages for high-assurance autonomous vehicles: Extended abstract. *Programming Languages Meets Program Verification* pp. 1–2. ACM.

Pnueli, A., Siegel, M. and Singerman, E. (1998a) Translation validation. *Tools and Algorithms for the Construction and Analysis of Systems* pp. 151–166. Springer.

Pnueli, A., Shtrichman, O. and Siegel, M. (1998b) Translation validation for synchronous languages. *International Colloquium on Automata, Languages and Programming* pp. 235–246. Springer-Verlag.

Reif, W., Schellhorn, G., Stenzel, K. and Balser, M. (1998) Structured specifications and interactive proofs with KIV. *Automated Deduction – A Basis for Applications*. Applied Logic Series 9, pp. 13–39. Springer.

Reynolds, J. C. (1974) Towards a theory of type structure. *Programming Symposium, Proceedings Colloque Sur La Programmation* pp. 408–423. Springer-Verlag.

Rideau, S. and Leroy, X. (2010) Validating register allocation and spilling. *International Conference on Compiler Construction* pp. 224–243. Springer.

Rizkallah, C. (2015) *Verification of program computations*. PhD thesis, Saarland University.

Rizkallah, C., Lim, J., Nagashima, Y., Sewell, T., Chen, Z., O'Connor, L., Murray, T., Keller, G. and Klein, G. (2016) A framework for the automatic formal verification of refinement from Cogent to C. *International Conference on Interactive Theorem Proving*.

Rubio-González, C. and Liblit, B. (2011) Defective error/pointer interactions in the Linux kernel. *International Symposium on Software Testing and Analysis* pp. 111–121. ACM.

Rust. (2014) *The Rust Language*. http://rust-lang.org. Accessed: 2014-10-05.

Rustan, K. and Leino, M. (2010) Dafny: An automatic program verifier for functional correctness. *Logic for Programming, Artificial Intelligence, and Reasoning* pp. 348–370. Springer.

Sabry, A. and Felleisen, M. (1992) Reasoning about programs in continuation passing style. *Conference on LISP and Functional Programming* pp. 288–298. ACM.

Saha, S., Lawall, J. L. and Muller, G. (2011) An approach to improving the structure of error-handling code in the Linux kernel. Vitek, J. and Sutter, B. D. (eds), *Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems, LCTES 2011, Chicago, IL, USA, April 11-14, 2011* pp. 41–50. ACM.

Schellhorn, G., Ernst, G., Pfähler, J., Haneberg, D. and Reif, W. (2014) Development of a verified flash file system. *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Lecture Notes in Computer Science 8477, pp. 9–24. Springer.

Schierl, A., Schellhorn, G., Haneberg, D. and Reif, W. (2009) Abstract specification of the UBIFS file system for flash memory. *World Congress on Formal Methods*. Lecture Notes in Computer Science 5850, pp. 190–206. Springer.

Schirmer, N. (2005) A verification environment for sequential imperative programs in Isabelle/HOL. *Logic for Programming, AI, and Reasoning* pp. 398–414. Springer.

Sewell, T., Myreen, M. and Klein, G. (2013) Translation validation for a verified OS kernel. *Programming Language Design and Implementation* pp. 471–481. ACM.

Shang, Z. (2020) *An LLVM Backend of the Cogent Compiler*. Project report, UNSW Sydney, Computer Science & Engineering.

Sison, R. and Murray, T. (2019) Verifying that a compiler preserves concurrent value-dependent information-flow security. Harrison, J., O'Leary, J. and Tolmach, A. (eds), *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*. LIPIcs 141, pp. 27:1–27:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Spector-Zabusky, A., Breitner, J., Rizkallah, C. and Weirich, S. (2018) Total haskell is reasonable coq. *Certified Programs and Proofs* p. 14–27. ACM.

Sullivan, G. F. and Masson, G. M. (1990) Using certification trails to achieve software fault tolerance. *Fault-Tolerant Computing Symposium* pp. 423–431. IEEE Computer Society.

Szekeres, L., Payer, M., Wei, T. and Song, D. (2013) Sok: Eternal war in memory. *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* pp. 48–62. IEEE Computer Society.

Tofte, M. and Talpin, J.-P. (1994) Implementation of the typed call-by-value λ-calculus using a stack of regions. *Principles of Programming Languages* pp. 188–201. ACM.

Torlak, E. and Bodik, R. (2014) A lightweight symbolic virtual machine for solver-aided host languages. *Programming Language Design and Implementation* p. 530–541. ACM.

Tuch, H., Klein, G. and Norrish, M. (2007) Types, bytes, and separation logic. *Principles of Programming Languages* pp. 97–108. ACM.

Wadler, P. (1990) Linear types can change the world! *Programming Concepts and Methods*.

Yang, J. and Hawblitzel, C. (2010) Safe to the last instruction: automated verification of a type-safe operating system. *Programming Language Design and Implementation* pp. 99–110. ACM.

Yang, J., Sar, C. and Engler, D. (2006) EXPLODE: a lightweight, general system for finding serious storage system errors. *Operating Systems Design and Implementation* pp. 131–146.

Zhao, J., Nagarakatte, S., Martin, M. M. and Zdancewic, S. (2012) Formalizing the LLVM intermediate representation for verified program transformations. *Principles of Programming Languages* pp. 427–440. ACM.