DARGENT

A Silver Bullet for Data Layout Refinement

ANONYMOUS AUTHOR(S)

Systems programmers need fine grained control over the memory layout of data structures, both to produce performant code and to comply with well-defined interfaces imposed by existing code, standardised protocols or hardware. Code that manipulates these low-level representations in memory is hard to get right. Traditionally, this problem is addressed by the implementation of tedious marshalling code to convert between compiler-selected data representations and the desired compact data formats. Such marshalling code is error-prone and can lead to a significant runtime overhead due to excessive copying. While there are many languages and systems that address the correctness issue, by automating the generation and, in some cases, the verification of the marshalling code, the performance overhead introduced by the marshalling code remains. In particular for systems code, this overhead can be prohibitive. In this work, we address both the correctness and the performance problems.

We present a data layout description language and data refinement framework, called DARGENT, which allows programmers to declaratively specify how algebraic data types are laid out in memory. Our solution is applied to the COGENT language, but the general ideas behind our solution are applicable to other settings. The DARGENT framework generates C code that manipulates data directly with the desired memory layout, while retaining the formal proof that this generated C code is correct with respect to the COGENT functional semantics. This added expressivity removes the need for implementing and verifying marshalling code, which eliminates copying, smoothens interoperability with surrounding systems, and increases the trustworthiness of the overall system.

Additional Key Words and Phrases: Certifying compiler, data refinement, systems programming

ACM Reference Format:
Anonymous Author(s). 2023. DARGENT: A Silver Bullet for Data Layout Refinement. *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2023), 25 pages.

1 INTRODUCTION

In the realm of software systems, such as device drivers, file systems, and network stacks, precise control over the *data layout* of objects is crucial for compatibility and performance. Specifically, controlling the composition of objects in memory on a bit- and byte-level can avoid the need for translation or *deserialisation* at the boundaries between on-medium and in-memory data which frequently arise from interacting with standardised protocols or hardware. These systems are often implemented in the C language, in part because it offers low-level features to give fine-grained control over data layout. Unfortunately, to maintain good performance, the C programmer must throw away the conceptual abstraction of the data type, and instead focus on the low-level details of bits and bytes. This low-level code contains many subtle bit-twiddling operations which, apart from being difficult to manually verify, are also tedious and error-prone to implement.

To maintain the higher-level structure of a program without sacrificing performance, we want to use a language with a high-level semantics, but with facilities for specifying the low-level memory layout of heap-allocated objects. While most high-level languages use fixed heap layouts, there has been recent progress on language-based support for user-defined memory layouts [Cronburg and Guyer 2019; Vollmer et al. 2019]. These languages, however, do not provide verified correctness of their translations.

In this paper, we present DARGENT, a language for describing data layouts of high-level algebraic datatypes along with a data refinement framework for *automatically verifying* those layout

2023. 2475-1421/2023/1-ART1 $15.00
https://doi.org/

descriptions (Section 3). We build on the COGENT language and refinement framework [O'Connor et al. 2021] (Section 2). COGENT is designed for the implementation of high-assurance low-level systems components as pure mathematical functions operating on algebraic data types. COGENT's certifying compiler co-generates a C program and Isabelle/HOL [Nipkow et al. 2002] theorems witnessing a proof that the C program refines an Isabelle/HOL embedding of the COGENT source program [Rizkallah et al. 2016].

While there is a very long line of prior work on *data description languages* [Back 2002; Fisher and Gruber 2005; McCann and Chandra 2000; Ramananandro et al. 2019; Slind 2021; van Geest and Swierstra 2017; Wang and Gaspes 2011; Ye and Delaware 2019], and the data layout descriptions used in DARGENT do indeed look similar to those used in such languages, there is a fundamental difference: These languages are designed for synthesising data (de)serialisation functions (also sometimes referred to as data marshalling/unmarshalling functions, encoders/decoders, or parsers and pretty-printers for low-level data), which convert data stored in a low-level, sequential format in some storage medium to a high-level, structured representation in memory and vice versa. They are primarily used for the interaction and communication between different programming languages (e.g., a foreign function interface) or systems (e.g., data transmission over the network). In this context, code to transform back-and-forth between the two representation is still necessary.

DARGENT, on the other hand, is intended to solve a different problem. The DARGENT data layout descriptions grant programmers the ability to dictate to the COGENT compiler how it should lay out the algebraic data types used by the COGENT program itself. The compiler generates code that works directly with data laid out according to the programmer's specifications, as well as Isabelle/HOL proofs showing that it has done so correctly.

Depending on the application, DARGENT therefore can either eliminate entirely or reduce the need for data (de)serialisation code, be it manually written or automatically derived, when interacting with the external world. Because algebraic data types can be represented directly in their binary data formats with DARGENT, the programmer does not need to decode raw data first into some other in-memory representation in order to operate on it as a data type. Eliminating these (de)serialisation steps naturally results in more concise and readable code, better performance, and easier informal and formal reasoning.

This additional power in expressiveness can also be used to improve the performance of the compiled COGENT code, e.g., by having smaller memory footprints or a specialised memory layout optimal for the underlying architecture, independent of the interoperation between languages or systems. It also enables COGENT programmers to directly write code that is binary-compatible with native C programs.

COGENT is readily amenable to an extension for prescribing data layouts and fine-tuning the compilation of algebraic data types by virtue of its lack of a runtime system and direct compilation to C. Before DARGENT, it simply adopted the layout conventions of the underlying C compiler. The introduction of DARGENT (Section 3) into the framework enabled improvements to some outstanding inefficiencies in the prior design, such as a reduced reliance on deserialisation code within file system implementations [Amani et al. 2016] and directly representing device register formats as data types (Sections 5.1 and 6). Furthermore, we have extended the COGENT compiler so as to preserve the benefits of COGENT's high-level type system and semantics. The upshot is our compiler automatically translates read and write operations on heap-allocated objects to take account of their particular data layout. The translation's correctness is guaranteed by the enhanced data refinement theorem (Section 4). To our knowledge, this is the first *certifying* framework to be extended with data layout descriptions supporting mutable data, thanks to COGENT's uniqueness type system which allows for efficient destructive updates.

Contributions

This paper realises the vision set out in 2018 [Anonymous 2018]. We make the following contributions:

- The design and implementation of DARGENT (Section 3), a *data layout* language for controlling the memory layout of algebraic data types, down to the bit level. We formalise a core calculus and its static semantics in Isabelle/HOL [Nipkow et al. 2002], and discuss the compilation process;
- An extension to the COGENT verification framework (Section 4) to *automatically verify* the translation of high-level read/write accesses (known as *getters* and *setters*) to explicit offsets within a well-defined memory region;
- An extended suite of examples (Sections 5 and 6) demonstrating the utility of DARGENT in the context of device drivers.

All the results described in this paper, including the case studies we present, are associated with formal proofs in Isabelle/HOL. All the code and proofs are publicly available in the project repository: provided as Anonymized Supplementary Material.

2 OVERVIEW OF COGENT

COGENT [O'Connor et al. 2021] is a higher-order, polymorphic, purely functional programming language, in the tradition of the pure subsets of languages such as HASKELL or ML. Programs are expressed as mathematical functions operating on algebraic data types. Unlike HASKELL or ML, however, COGENT is designed for implementing low-level systems code where manual memory management is essential for performance reasons. As such, COGENT has no garbage collection mechanism and heap (de)allocations are explicit in the language.

The COGENT language is equipped with a uniqueness type system [Barendsen and Smetsers 1993; Wadler 1990] enabling a seamless transition from a purely functional semantics to an imperative semantics and a compilation strategy to efficient low-level C code. The certifying compiler automatically produces a formal proof [Rizkallah et al. 2016] that its generated C code refines an Isabelle/HOL embedding of COGENT's functional semantics. COGENT was used to implement two real-world file systems and verify the correctness of key file system operations [Amani et al. 2016]. This section summarises aspects of COGENT and its verification framework that are relevant to our work.

2.1 Uniqueness types system

Uniqueness type systems ensure that each *linear* object in memory is uniquely referenced. Consequently, updates to these objects in a purely functional language can be compiled as in-place destructive updates, without the need for copying [Wadler 1990]. In COGENT, we call the type of objects that are subject to the uniqueness restrictions *linear types*, and the rest *non-linear types*. Roughly speaking, any objects that reside in the heap, or contain pointers to other heap-objects are linear (with one exception, explained later), otherwise they are non-linear. COGENT's verification framework depends on the AutoCorres library, which does not support stack pointers, therefore all pointers address heap memory. In short: a linear object is behind a pointer and/or contains pointers. Later we will see that the linearity of types are closely correlated to the DARGENT layouts.

As a simple COGENT example, consider the following code.

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

addToBag : (U32, Bag) → Bag
```

```
addToBag (x, b {count = c, sum = s}) = b {count = c + 1 , sum = s + x }
```

The first line declares a linear record type `Bag`, which is a record comprised of two 32-bit unsigned words. In the `addToBag` function, we retrieve the value of the two fields by pattern matching and update both fields of the record according to the given input. It is compiled to a C function that takes as input a pointer to a bag structure and updates it *in place*, because the uniqueness type system ensures that the `Bag` object is uniquely referenced.

COGENT *primitive* types include the n-bit unsigned integer types, Un (where $n = 1 \ldots 64$), and booleans (Bool). Among the Un types, we call them *word-sized integers* (or simply *words*) when $n \in \{8, 16, 32, 64\}$. Algebraic data types can be formed using *record* and *variant* types (aka. tagged unions). Furthermore, COGENT supports declaring *abstract types* with their definitions provided in C. COGENT employs a structural type system, meaning that compound types, as well as their subtyping relations, are defined solely by their structure and the structure of their components. Names given to types such as `Bag` above are type synonyms—mere abbreviations for better cosmetics.

The compilation process translates COGENT algebraic data types into C structs. The COGENT type system distinguishes between *boxed* and *unboxed* types through a *sigil* annotation on the type. A boxed type (sigil `b`) resides on the heap and is accessed by reference through its unique pointer. An unboxed type (sigil `u`) is accessed by value and either resides on the stack or is inlined inside a larger data structure on the heap. In the latter case, when the object is accessed, it will be copied by-value to a stack variable.

Records and abstract types can be either boxed or unboxed. Primitive types and variant types, however, can only be unboxed. For this reason, primitives and variants do not come with a sigil. For a boxed type, COGENT allows further fine-grained control of accessibility: a boxed type can either be writable (with sigil `w`) or read-only (with sigil `r`). When a type has a read-only sigil, even though it is behind a pointer or contains pointers, it becomes non-linear — this is the exception we mentioned at the beginning of this section. This read-only access is analogous to the concept of "borrowing" in the type system of Rust.

Before DARGENT, the COGENT compiler used a pre-defined code generation algorithm to compile data types to C. A record type in COGENT was mapped to a C struct, with the fields laid out in the order in which they are declared in the type. The mapping for variant types, on the other hand, was less direct: COGENT's verification tool chain does not support C unions, so variants were also represented as structs in C, containing a field for a tag, and a field for each alternative's payload.

Applying such a fixed code generation scheme is no surprise for a typical high-level functional language, whose implementation details are hidden from the language users. But COGENT is not just a functional language, it is also a systems language where the exact low-level representation of data types is relevant to programmers. This fixed code generation algorithm often resulted in suboptimal or undesired representations of COGENT types in C, and users of the COGENT language had to know about the implementation details of the code generator in order to write C code that directly interfaces with COGENT programs. In situations where the COGENT program needs to interoperate with pre-existing C components, say, when developing an operating system component that interfaces with the Linux kernel headers, glue code was required to translate between representations, resulting in development and run-time overhead. Also, problematically, as this glue code depended on the representation choices made by the compiler, future versions of the COGENT compiler could break previously working code due to changes in the code generation scheme.

2.2 Verification framework

In addition to a programming language, COGENT is also a verification framework realised in Isabelle/HOL, based on *certifying compilation*. Compiling a COGENT program results in multiple components:

(1) a C program,
(2) an Isabelle/HOL *shallow embedding* of the COGENT program,
(3) an Isabelle proof of refinement between the C program and the Isabelle shallow embedding.

The last item relies on the AutoCorres library [Greenaway et al. 2014] to generate a representation of the C code in Isabelle/HOL. More precisely, the AutoCorres library abstracts the C semantics via a SIMPL [Schirmer 2005] formal language into a monadic embedding in Isabelle/HOL.

This compilation process provides an indirect way of formally verifying properties about the generated C program. The user first proves the desired properties about the Isabelle/HOL shallow embedding. This manual proof should follow by simple equational reasoning about HOL terms by applying term rewriting tactics provided by the theorem prover. Then, the automatic refinement proof between the C code and the shallow embedding transports the proven properties to the C program. In short, COGENT's verification framework reduces complicated low-level verification on the C program to a simple high-level equational proof.

The compiler-generated refinement proof between the C code and the Isabelle/HOL shallow embedding is composed of several smaller correspondence proofs. When the compiler generates the shallow embedding of the COGENT program in Isabelle/HOL, it also generates a *deep embedding* representing the abstract syntax of the COGENT program. Two semantics are assigned to the deep embedding: a purely functional *value semantics*, and a stateful *update semantics*, with pointers, memory states, and in-place field updating. It is proved once-for-all that these two semantics are equivalent for any well-typed COGENT program. As part of the certifying compilation, the compiler produces a refinement proof between the shallow embedding and the deep embedding with value semantics, and a refinement proof between the deep embedding with update semantics and the monadic C embedding obtained from AutoCorres. Chaining the three correspondence lemmas results in a correspondence between the shallow embedding and the C program, stating that the C program is a refinement of the COGENT program's semantics (see Figure 1).

3 DARGENT

We have designed and implemented a data layout description language called DARGENT, which describes how a COGENT algebraic data type may be laid out in memory, down to the bit level. Layout descriptions in DARGENT are transparent to the shallow embedding of COGENT's semantics, but they influence the definition of the refinement relation to C code generated by the compiler. In Section 4, we describe in more detail the extensions to our verification framework to accommodate the DARGENT layout descriptions. Here, we focus on the language definition. Firstly, we give an informal overview of DARGENT's language features.

3.1 An informal introduction to DARGENT

DARGENT offers the possibility to assign any boxed COGENT type a custom layout describing how the data should be stored in the heap. A boxed type assigned with a custom layout is compiled to a C struct with a single field: an array of 32-bit words.[1] This array *represents* the COGENT type: it can be deemed as untyped in C, but the DARGENT description contains enough information to access

[1]Throughout the paper, unless we explicitly specify the size, the term "word" always refers to unsigned integer types of 1 byte, 2 bytes, 4 bytes or 8 bytes and the actual size is usually less relevant in the discussion. It does not necessarily imply pointer-sized words. We discuss the implications of the choice of the word size later in Section 3.5.

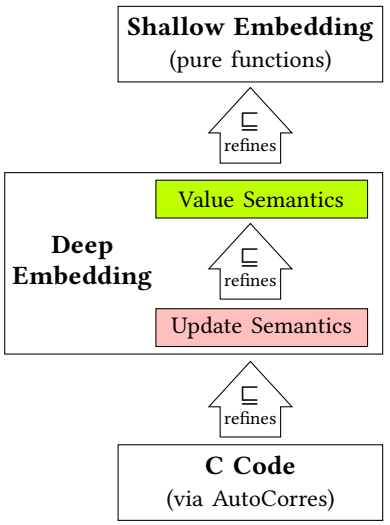

Fig. 1. The COGENT refinement framework.

```
type Example = {
  struct : #{a : U32, b : Bool},   -- nested embedded record
  ptr : {c : U8},                  -- pointer to another record
  sum : ⟨A U16 | B U8⟩             -- variant type
}
```

Fig. 2. COGENT type example

individual parts of the type correctly. How data is laid out in memory is purely a low-level concern, and it does not affect the functional semantics of a COGENT program in any way. In other words, COGENT functions are parametric over the layouts of types. Under the hood, the COGENT compiler generates custom getters (and setters) in C, retrieving (and setting) the relevant parts of the type from the representing array.

As an example, consider the COGENT type in Figure 2. This record consists of three fields: struct, ptr and sum. struct is an unboxed record, denoted by the leading # symbol. This field is embedded inside the parent Example record. The ptr field is a boxed record, and is stored somewhere else in the heap, referenced by a pointer in the Example type. The last field is a variant type, with two alternatives tagged A and B respectively. This variant is unboxed (recall that there is no boxed variant in COGENT) and is stored inside the parent record.

A layout for this record type must specify where each field is located in the word array. Overlapping is not allowed, except for the payloads of the two constructors of the variant type, since only one of them is relevant at each time, depending on the tag value. The layout must also specify what the tag values for the variant constructors A and B are. We will give a layout to this type after a short introduction to the DARGENT language constructs.

In Figure 3 we present the surface syntax for DARGENT. A *layout expression* is a description of the usage of some (heap) memory. It only describes the low-level view of a memory region—it is not associated to any particular algebraic data type. From this perspective, DARGENT descriptions are

Sizes	s	::=	$n\mathsf{B} \mid n\mathsf{b} \mid s + s$
Layout expressions	ℓ	::=	s (block of memory)
		$\mid$	x (layout variable)
		$\mid$	pointer (pointer layout)
		$\mid$	$L\ \overline{\ell_i}$ (another layout)
		$\mid$	ℓ at s (offset operator)
		$\mid$	ℓ after f (relative location)
		$\mid$	ℓ using ω (endianness)
		$\mid$	record $\{\overline{\mathrm{f}_i : \ell_i}\}$
		$\mid$	variant (ℓ) $\{\overline{\mathrm{A}_i\ (n_i) : \ell_i}\}$
Declarations	d	::=	**layout** $L\ \overline{x_i} = \ell$
Layout names		$\ni$	L
Endianness	ω	::=	BE $\mid$ LE
Field names		$\ni$	f
Constructors		$\ni$	A
Natural numbers		$\ni$	n, m

(lists are represented by overlines)

Fig. 3. DARGENT syntax

independent of COGENT types. As we will shortly see, however, a given COGENT type can only be laid out in certain ways, which places restrictions on which layouts can be assigned to a given type.

When specifying a layout, two pieces of information are relevant: how much space a component occupies in memory and where it is placed in relation to the overall heap object in which it is contained. Primitive types, such as integer types and booleans, are laid out as a contiguous block of memory of a particular size. For example, a contiguous 4-byte block would be an appropriate layout for the 32-bit word type `U32`. A block of memory is specified as a size expression, which can be in bytes (B), bits (b), or additions of smaller sizes. Additionally, memory blocks of word size (e.g. 1 byte, 2 bytes, 4 bytes and 8 bytes) can be given an endianness (BE or LE), with the using keyword.

Components of boxed type are represented as a pointer, and thus must be described with the special pointer layout, and not as a chunk of memory. This special layout improves readability of code, and also allows for some portability: A pointer layout will have different sizes according to the host machine's architecture.

Layouts for record types use the record construct, which contains sub-expressions for the memory layout of each field. As we can specify memory blocks down to the individual bits, we can naturally represent records of boolean values as a bitfield:

layout Bitfield = record {x : 1b, y : 1b at 1b, z : 1b at 2b}

Here the at operator is used to place each field at a different bit offset, so that they do not overlap. If two record fields have overlapping layouts, the description is rejected by the compiler.

The at operator can be applied to any layout expressions, which will shift the entire expression by the specified amount. Alternatively, the after operator can be used in a record layout:

layout Bitfield = record {x : 1b, y : 1b after x, z : 1b after y}

so that a later field is placed right after the previous one. This saves the programmer from calculating the concrete offset. When no offset (at or after) is given, it will by default place the field after the previous one.

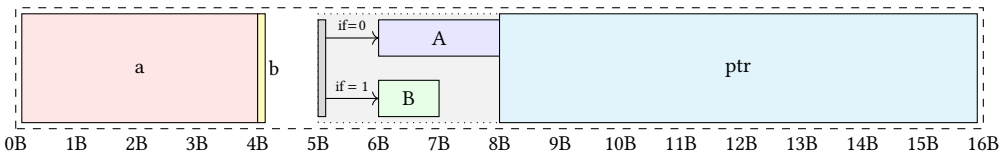

Fig. 4. The *ExampleLayout*, visualised.

Layouts for variant types use the variant construct. It firstly requires a layout expression for the *tag*. Then, for each constructor in the variant, a specific tag value needs to be assigned, followed by a layout expression for the *payload* of that constructor. When one alternative is taken, the memory used by other alternatives becomes irrelevant, which is why the memory for the payloads can overlap. Additionally, Dargent allows for zero-sized payloads. For instance, the Maybe *a* type, defined as ⟨Just *a* | Nothing ()⟩, may be given a layout in which the payload for constructor Nothing does not occupy any memory.

Similar to Cogent types, Dargent expressions are also structural. Layout synonyms can be defined using the `layout` keyword, just as Cogent type synonyms are defined using the `type` keyword. For example,

layout FourBytes = 4B

defines a layout synonym FourBytes, which is definitionally equal to 4B on the right hand side. Layout and type synonyms can take parameters.

We can now give a Dargent description to the Example type in Figure 2 (assuming a 64-bit architecture):

```
layout ExampleLayout = record {
  struct : record {a : 4B, b : 1b},
  ptr : pointer at 8B,
  sum : variant (1b)
    {A(0) : 2B at 1B, B(1) : 1B at 1B} at 5B
}
```

Figure 4 gives a pictorial illustration of this memory layout. In the layout above, it is worth noting that the at 1B offsets for the two payloads of A and B are in relation to the beginning of the sum field, which is 5 bytes (5B) from the beginning of the top-level structure. We can equivalently write variant (1b at 5B) {A(0) : 2B at 6B, B(1) : 1B at 6B} at 0B for the sum field without changing its layout.

At this point, this layout is still independent of the Cogent type, and the compiler will only check that this layout definition is well-formed: that it does not have overlapping fields, that the tag values are distinct, and so on. To associate a layout to a type, we add a **layout** keyword to the type language of Cogent. For this example, the type Example **layout** ExampleLayout describes the type Example laid out according to the description in ExampleLayout. The compiler will check that ExampleLayout is an appropriate layout for the Cogent type Example. We will talk more about the wellformedness and matching rules later in Section 3.4. To eliminate verbosity, a type synonym can be given to the layout-annotated type above.

We make the design choice of having Dargent layouts be defined independently of Cogent types, and only relating them afterwards with the **layout** keyword. This design may seem sub-optimal, as some common information is duplicated in both the types and the layouts, and matching

them requires a set of dedicated typing rules (as we will see in Figure 6). We make this design decision for several reasons.

Firstly, while the COGENT type structure is central to the functional semantics of any COGENT program and the reasoning of its function correctness, the exact layout of algebraic data types is just an implementation detail, totally irrelevant to the top-level HOL embedding of the COGENT program, and whose correctness is guaranteed by the automatic C refinement proof that the COGENT compiler produces. These two constructs are conceptually separate.

Secondly, this approach leaves more flexibility and is amenable to future extensions. Currently, as we will see in Figure 6, the layout-type matching is fairly restricted: e.g. a `U16` type has to occupy 2 bytes (2B) in memory, and a record type has to be laid out in accordance with a record layout. In the future, several extensions can be made to relax this matching relation. For example, it is technically valid to store a smaller type in a larger memory area, say, a `U16` type in a memory region of 4 bytes, or a record type in a contiguous chunk of memory that is big enough. Some heuristics can be implemented in the compiler to decide how to arrange under-specified layouts. This feature can be useful in improving programmers' productivity, because the programmers do not always need to fully specify the layout when the layout details of parts of a type that are unimportant. Also, as there is ongoing work towards adding refinement types to COGENT [Paradeza 2020], a refined type could possibly be laid out in a smaller memory area. For instance, $\{\nu : \texttt{U16} \mid \nu < 2\}$ can be laid out in a memory area as small as 1 bit. None of these extensions would be easy to implement if the layout information was baked into the types.

Thirdly, separating layouts and types encourages modularity. Developers using the COGENT language can write programs without needing to worry about the detailed layout of types, and are still able to prove functional correctness of their code against some high-level specification and ship their code to end users. The end users, with the knowledge of the particular target architecture and environment, can decide the layouts and plug them into the COGENT programs.

Finally, this design decision enables easier compiler engineering. Even though our design requires a dedicated set of rules for checking the layout-type matching, it actually significantly simplifies the compiler engineering in the long term. The COGENT compiler is very large, and the layout-type matching checker only constitutes a small part of the typechecker. The compiler not only compiles COGENT programs to C, but also generates information for various Isabelle proof tactics, many embeddings of the program in Isabelle/HOL and Haskell [Chen et al. 2017], and so on. A lot of these embeddings are only concerned with types, and not layouts. If the layouts and types were merged, any changes to the layout implementation would require changes to many irrelevant parts of the compiler.

3.2 Layout Polymorphism

We also extend COGENT's existing parametric polymorphism mechanism to support *layout polymorphism*. This feature is handy when we want to abstract over the layout we would like to assign to a certain type, when targeting different architectures, for instance. In a COGENT function signature, *layout variables* can be universally quantified, just as with type variables. These layout variables may be constrained, similarly to type constraints in HASKELL, which require that a layout variable matches a type. [2] For example, in the code snippet below,

$$\begin{aligned}
&\textbf{type}\ \mathsf{Pair}\ t = \{\mathsf{fst} : t, \mathsf{snd} : t\}\\
&\textbf{layout}\ \mathsf{LPair}\ l = \mathsf{record}\ \{\mathsf{fst} : l, \mathsf{snd} : l\ \mathsf{at}\ 4\mathsf{B}\}\\
&\mathit{freePair} : \forall(\tau, \ell :\sim \tau).\ \mathsf{Pair}\ \tau\ \textbf{layout}\ \mathsf{LPair}\ \ell \to ()
\end{aligned}$$

[2] A formal definition of layout matching is discussed in Section 3.4.

Bit ranges	r		
Offsets	o	$\in$	$\mathbb{N}$
Layouts	ℓ	::=	$()$ (unit layout)
		|	r_ω (primitive layout)
		|	$x\{o\}$ (layout variable)
		|	record $\{\overline{\mathsf{f}_i : \ell_i}\}$
		|	variant (ℓ) $\{\overline{\mathsf{A}_i\ (n_i) : \ell_i}\}$
Field names		$\ni$	f
Constructors		$\ni$	A
Endianness	ω	::=	BE | LE | ME

(lists are represented by $\overline{\text{overlines}}$)

Fig. 5. Syntax for Dargent core language

we define a parametric type synonym Pair t and layout synonym LPair l, and an abstract polymorphic function that operates on such a pair. In the function's type, we require that the layout l be able to describe the layout of type t, so that the LPair l is always a valid layout expression for type Pair t. Layout-polymorphic functions may be explicitly applied to layouts, akin to explicit type applications. For example, *freePair*[U32]{{FourBytes}} instantiates τ to U32 and ℓ to FourBytes. If the type/layout application ends up with incompatibility, for instance *freePair*[U8]{{1b}} in which 1b is not big enough to store the U8, the typechecker will reject such a program. The typechecker also ensures that any instantiation of ℓ produces layouts that are well-formed. For example, if ℓ is instantiated with 8B, it will render LPair ℓ ill-formed, as the snd field will overlap with the fst. This can be rectified by using the after relative location (or leaving the location implicit) instead, which will automatically place the snd field right after fst.

3.3 The Dargent core calculus

The Dargent surface language is desugared into a smaller core calculus, whose syntax is outlined in Figure 5. The core calculus is the language on which the verification is based. As can be seen in Figure 5, layouts consist fundamentally of *bit ranges*, which describe which bits in memory are used to store each piece of data. The definition of bit ranges is omitted, because our core calculus is parametric over the exact bit range representation—at different points in the compilation pipeline, bit ranges are represented differently, to suit the purpose of that particular phase of the compilation.

Bit ranges are annotated with an endianness to form a primitive layout in our core language. When the endianness is not specified in the surface language, a *machine endianness* will be given by default, written ME in core. When C code is generated, a subroutine will be invoked to determine the machine endianness, so that the C code works as intended. In the paper, when the endianness is unimportant, we will leave the subscript out from ℓ_ω.

When the surface layout expressions are first desugared, the bit ranges are represented as BitRange (o, s), which is a pair of natural numbers, indicating the offset (o bits) from *the beginning of the top-level heap object* in which it is contained, as well as the range occupied (s bits). In BitRange (o, s), we require $s > 0$ for convenience. For zero-sized layouts, the empty layout $()$ can be used instead.

Layout variables are given the form $x\{o\}$, as we additionally need to remember the offset to be applied to it when it is instantiated—when x gets instantiated, it will be shifted to the right by o bits. The other core layouts are very similar to their counterparts in the surface language.

In summary, the desugarer converting the surface layout expressions to the core calculus must perform the following tasks:

- expanding layout names to layout definitions;
- computing size expressions into bit ranges;
- computing offsets relative to the beginning of the top-level heap object;
- inserting explicit layout applications to any layout-polymorphic function calls.

Recall that in COGENT, the boxedness of a type is denoted by a sigil. The surface language of DARGENT assigns layouts to any COGENT type. However, since we only allow layouts to be specified on boxed types, in the formal definition layouts are attached to the boxed sigil, describing the layout of the datatype behind the pointer:

$$\begin{array}{rrcll} \text{Sigils} & s & ::= & \mathsf{w}_\ell & \text{(writable sigil)} \\ & & \mid & \mathsf{r}_\ell & \text{(readonly sigil)} \\ & & \mid & \mathsf{u} & \text{(unboxed sigil)} \end{array}$$

In this work we also have a sigil b_ℓ as a notational convenience when we do not wish to distinguish whether it is writable or readonly. Layout annotations can be left out and the compiler will use the default layout.

3.4 The static semantics with DARGENT

COGENT's static semantics, along with some meta-properties of the type system such as type preservation, are formalised in Isabelle/HOL (O'Connor et al. [2016, 2021] give more details). In this section, we explain how these formal proofs and definitions are extended to take DARGENT into account.

In the original COGENT typing judgement $\Delta; \Gamma \vdash e : \tau$, we have the set of polymorphic type variables Δ in the context. Each entry in Δ also includes a set of *constraints* for that type variable, which can require that, for example, a type variable represent only types that are shareable. When these type variables are instantiated, these constraints are checked. We also have a wellformedness judgement on types, $\Delta \vdash \tau\ \mathbf{wf}$, which ensures that all type variables are in scope in Δ and that the type structure of τ respects the linearity constraints in Δ.

With DARGENT, the typing judgement must additionally ensure that COGENT types match their assigned layouts (Figure 6) and that the layouts are themselves wellformed (Figure 7). Because COGENT has polymorphic type and layout variables, however, these additional constraints cannot always be immediately ensured when type-checking locally. Thus, we extend the COGENT typing judgement $\Delta; C; \Gamma \vdash e : \tau$ and wellformedness judgement $\Delta; C \vdash \tau\ \mathbf{wf}$ to include an additional set of *layout* constraints C. The set C tracks the relationship between polymorphic type variables and their assigned layouts. It is a set of pairs $(\ell, \hat{\tau})$ denoting that the layout ℓ must match the type $\hat{\tau}$. When the types and layouts are instantiated, these matching constraints are then checked.

The type in the context C is *simplified*, denoted as $\hat{\tau}$, because any type structure beyond the single heap object described by the layout ℓ is irrelevant. This means that all boxed types, which are all uniformly represented as a pointer to another heap object, can be treated simply as a primitive machine word when matching with layouts. Types are simplified by replacing all boxed components of the type with the newly introduced pointer type constant.

$$\boxed{\ell \sim \hat{t}}$$

$$\frac{\text{bits}\ (T) = s \qquad T \text{ is a primitive type}}{\text{BitRange}\ (o, s) \sim T}\textsc{PrimTyMatch} \qquad \begin{array}{l}\text{bits}\ (\text{U}n) \quad = n \\ \text{bits}\ (\text{Bool}) \quad = 1\end{array}$$

$$\frac{M \text{ is the pointer size}}{\text{BitRange}\ (o, M) \sim \text{pointer}}\textsc{BoxedTyMatch} \qquad \frac{}{x\{o\} \sim \hat{t}}\textsc{VarMatch}$$

$$\frac{}{() \sim ()}\textsc{UnitMatch} \qquad \frac{\text{for each } i\text{: } \ell_i \sim \hat{t}_i}{\text{record}\ \{\overline{\text{f}_i : \ell_i}\} \sim \text{record}\ \{\overline{\text{f}_i : \hat{t}_i}\}}\textsc{URecordMatch}$$

$$\frac{\text{for each } i\text{: } \ell_i \sim \hat{t}_i}{\text{variant}\ (s)\ \{\overline{\text{A}_i\ (v_i) : \ell_i}\} \sim \text{variant}\ \{\overline{\text{A}_i\ \hat{t}_i}\}}\textsc{VariantMatch}$$

Fig. 6. Matching relation between layouts and types

$$\boxed{\ell\ \mathbf{wf}}$$

$$\frac{}{\text{BitRange}\ (o, s)\ \mathbf{wf}}\textsc{BitRangeWF} \qquad \frac{}{()\ \mathbf{wf}}\textsc{UnitWF}$$

$$\frac{\begin{array}{c}\text{for each } i\text{: } \ell_i\ \mathbf{wf} \\ \text{for each } i \neq j\text{: } \text{taken}(\ell_i) \cap \text{taken}(\ell_j) = \emptyset\end{array}}{\text{record}\ \{\overline{\text{f}_i : \ell_i}\}\ \mathbf{wf}}\textsc{URecordWF}$$

$$\frac{\begin{array}{c}\text{for each } i\text{: } \ell_i\ \mathbf{wf} \qquad v_i < 2^s \qquad \text{taken}(\ell_i) \cap \text{taken}(\text{BitRange}\ (o, s)) = \emptyset \\ \text{for each } i \neq j\text{: } v_i \neq v_j\end{array}}{\text{variant}\ (\text{BitRange}\ (o, s))\ \{\overline{\text{A}_i\ (v_i) : \ell_i}\}\ \mathbf{wf}}\textsc{VariantWF}$$

$$\frac{}{x\{o\}\ \mathbf{wf}}\textsc{VarWF}$$

where $\text{taken}(\ell) \in \mathbb{N}$ returns the set of bit positions that ℓ occupies (defined recursively).

Fig. 7. Wellformed layouts

3.4.1 Typing rule for records with layouts. We are now ready to present the main typing rule that is added for DARGENT: wellformedness of boxed records with custom layouts.

$$\frac{\begin{array}{c}\overline{\text{f}_i}\ \text{distinct} \qquad \Delta; C \vdash \tau_i\ \mathbf{wf} \qquad \ell\ \mathbf{wf} \\ \ell \sim \text{record}\ \{\overline{\text{f}_i : \hat{\tau}_i}\} \qquad (\ell, \text{record}\ \{\overline{\text{f}_i : \hat{\tau}_i}\}) \in C\end{array}}{\Delta; C \vdash \{\overline{\text{f}_i :: \tau_i^?}\} b_\ell\ \mathbf{wf}}\textsc{RecWf} \tag{1}$$

The first two premises in this rule are similar to those in COGENT, the next three premises are added to account for DARGENT. The layout wellformedness judgement, $\ell\ \mathbf{wf}$, ensures that field layouts do not overlap (Figure 7). The last two premises require the type to match the layout (Figure 6) and ensure that this is tracked in the set of constraints, respectively. We in fact use a variant of the rule RECWF where the layout constraint is only tracked in C if the type or the layout are polymorphic.

3.4.2 Specialisation. COGENT's type preservation proof relies on type specialisation maintaining welltypedness [O'Connor et al. 2016, Lemmas 3 and 4].

With DARGENT, we need to adapt these statements as follows.

LEMMA 3.1 (SPECIALISATION LEMMAS). *Let $\overline{\rho_i}$ be a list of types and $\overline{l_j}$ be a list of layouts. Let $\overline{\alpha_i}$ denote the list of type variables declared in Δ.*

*Then, $\Delta; C \vdash \tau$ **wf** and $\Delta; C; \Gamma \vdash e : t$ respectively imply*

$$\Delta'; C' \vdash \tau[\overline{\rho_i}/\overline{\alpha_i}, \overline{l_j}/\overline{x_j}] \ \mathbf{wf};$$

$$\Delta'; C'; \Gamma[\overline{\rho_i}/\overline{\alpha_i}, \overline{l_j}/\overline{x_j}] \vdash e[\overline{\rho_i}/\overline{\alpha_i}, \overline{l_j}/\overline{x_j}] : t[\overline{\rho_i}/\overline{\alpha_i}, \overline{l_j}/\overline{x_j}],$$

on the following conditions:

- *for each i, $\Delta', C' \vdash \rho_i$ **wf**;*
- *for each pair $(\ell, \hat{t}) \in C$ such that $\Delta \vdash \hat{t}$ **wf**, the following statements hold:*
  - *$\ell[\overline{l_j}/\overline{x_j}]$ **wf**;*
  - *$\ell[\overline{l_j}/\overline{x_j}] \sim \hat{t}[\overline{\hat{\rho}_i}/\overline{\alpha_i}]$;*
  - *$(\ell[\overline{l_j}/\overline{x_j}], \hat{\tau}[\overline{\hat{\rho}_i}/\overline{\alpha_i}]) \in C'$.*

In this lemma, $-[\overline{\rho_i}/\overline{\alpha_i}, \overline{l_j}/\overline{x_j}]$ denotes the simultaneous substitution replacing both type variables α_i and layout variables x_j, and Γ is a typing context environment.

Informally, the two conditions mean that the pair $(\overline{\rho_i}, \overline{l_j})$ of type and layout lists defines a valid substitution from $\Delta; C$ to Δ', C'. They also appear in the typing rule for specialising polymorphic functions. The second one is new and enforces that for each pair $(\ell, \hat{t}) \in C$ that is wellformed (in the sense that type variables appearing in $\hat{t}$ are in Δ), the substituted constraint is satisfied and remembered in the set of constraints C'. This last requirement can in fact be dropped if the substituted constraint is closed.

3.5 Compiling records: custom getters and setters

Currently, records are the only built-in types in COGENT that can be boxed. We therefore use records as an example to discuss how DARGENT layouts influence the generated C code.

Without custom layouts, a COGENT record is directly compiled to a C struct with as many fields as the original record: if $T = \{a : A, b : B\}$, then $[\![T]\!]$ is a pointer type to `struct` $\{[\![A]\!]\ a; [\![B]\!]\ b;\}$, where $[\![\cdot]\!]$ denotes the compilation of COGENT types to C types.

The DARGENT extension to the compiler relies on the observation that we are free to choose what a COGENT boxed record is compiled to as long as we provide getters and setters for each field, since they account for all the available COGENT operations on boxed records. Assigning a custom layout ℓ to a COGENT record T results in a C type that we denote by $[\![T]\!]_\ell$, consisting of a C struct with a fixed sized array of words as a single field. The implementation chooses the word size to be 32 bits, primarily because most COGENT programs we develop are targeting 32-bit embedded systems, but this is not fundamental to the design and can be easily made configurable to any word size. It is worth mentioning that this does not mean that a layout has to be a neat multiple of 32 bits in size. It is absolutely valid to have a record layout like record {a : 1B, b : 3b}, 11 bits in total. When this layout is embedded in another layout, e.g. record {x : record {a : 1B, b : 3b}, y : 2B}, the remaining bits after the field b will be used by the following field y, without any implicit padding.

The getters and setters for each field are generated according to the layout. If a top-level boxed record T contains a field $a : A$, the C prototypes are

$$[\![A]\!]\ \texttt{get\_a}\ ([\![T]\!]_\ell\ t);$$

$$\texttt{void set\_a}\ ([\![T]\!]_\ell\ t, [\![A]\!]\ a);$$

Note that $[\![A]\!]$, the return type of the getter (and similarly for the second argument of the setter function) does not involve the layout ℓ. If A is a boxed type, then $[\![A]\!]$ is a pointer to the type A,

whose layout is dictated by the layout information stored in A's sigil, independent of ℓ. If A is unboxed, the getter function is the point at which we convert the low-level custom representation governed by the layout ℓ into a standard representation of A, so that the value of the field a can be inspected by the rest of the program. As an example, consider the `struct` field in Figure 2. Roughly, the generated C getter has prototype `struct { U32 a; bool b; } get_struct (Example * t)`, where `Example` is a C structure with a single field consisting of a fixed sized array spanning 16 bytes.

Getters and setters are generated recursively following the structure of the involved field types. The process typically involves generating auxiliary "nested setters and getters" that directly manipulate the data array based on the value of the nested field (such as a or b in the example of Figure 2), and similarly for getters. For example, the getter and setter for the `struct` field of Figure 2 are roughly implemented as follows.

```
// the data array
typdef struct Example { U32[4] data } Example;

struct field_struct { U32 a; bool b; };

// setter for the struct field
void set_struct(Example * d, struct field_struct v)
{
  // calls to nested setters
  set_struct_a(d, v.a);
  set_struct_b(d, v.b);
}

// getter for the struct field
struct field_struct get_struct(Example * d)
{
  // calls to nested getters
  return (struct field_struct){.a = get_struct_a(d), .b = get_struct_b(d)};
}
```

As can be seen, a getter function (or similarly a setter) incurs a data format conversion: it turns the low-level data format into a high-level typed value in C. Therefore getting and passing around a large unboxed types can be expensive at run-time. In practice though, if the program is carefully designed and implemented, and the programmer only passes the minimal structures needed to external functions, there is a good chance that an optimising C compiler is able to eliminate unnecessary data conversions. COGENT allows programmers to annotate functions with a `CINLINE` pragma, so that the `inline` modifier is generated in the C functions, exposing more optimisation opportunities.

Invalid bit patterns. Calling a getter on an external word array can lead to unexpected behaviours when the data format is invalid. This can happen when variant types are involved, if the value in the tag part does not match any tag values declared in the DARGENT layout. Any undefined tag values will be treated as the last constructor's tag. For example, if the layout of a variant is variant (2b) $\{\mathrm{A}(0) : 1B, \mathrm{B}(1) : 2\mathrm{B}, \mathrm{C}(2) : 4\mathrm{B}\}$ and the tag value seen in the data format is 3, which

does not match any defined tag values but also fits in the 2-bit space reserved for the tag, it will be deemed as the last alternative, namely C.

DARGENT is not a low-level data parsing language, nor an interface language, therefore the generated C getters do not check for validity, and assume that the input is valid. Users of the language are responsible for checking the validity of any incoming data.

Required properties of getters and setters. As mentioned above, the compilation of a COGENT record type T to C can be arbitrary as long as getters and setters are provided. When it comes to formally verifying the compiled C program, these getters and setters are expected to compose well:

(1) setting a field does not affect the result of getting another field;
(2) getting a set field should return the set value (or at least an equivalent value).

These properties are discussed in more detail in Section 4.3. It is worth mentioning that, somewhat unintuitively, extending the verification framework to take DARGENT into account does not require us to prove that the generated getters and setters are accessing data at the locations specified by the layout.

3.6 Implementation

As mentioned in Section 3.3, the DARGENT core calculus initially represents a bit range as a pair of integers denoted by BitRange (o, s), where o indicates the offset (in bits) to the beginning of the top-level datatype in which it is contained, and s indicates how many bits it occupies. This representation is concise and easy to work with when typechecking the core language. Such representations, however, do not necessarily lend itself to an easy code generation algorithm. Therefore we have another step to convert each bit range into a list of *aligned bit ranges* tailored for C code generation. Each aligned bit range is essentially a word.

An aligned bit range AlignedBitRange (w, o, s) is a triple of integers, where w is the word-offset to the top-level datatype, o is the bit-offset inside this word, and s is the number of bits occupied. Each aligned bit range contains the information of how the bits in each word is used. The generated C getter/setter consists of a series of statements, each operating on one word via bitwise operations. Each such C statement is generated according to one aligned bit range. This one-to-one mapping simplifies C code generation. Because of the deployment of aligned bit ranges, it is easy for us to choose an appropriate word size (namely the size of the aligned bit ranges) specifically tailored for the application in question. This flexibility is important for low-level programming (see Section 5.1 for a concrete example) and for high performance.

4 VERIFICATION FRAMEWORK WITH DARGENT

As we have demonstrated, DARGENT affects both the formalisation of the type system and the C code generation of the compiler. Moreover, it also affects the generated correspondence proof between the C code and the pure shallow embedding in Isabelle/HOL. For this proof, only the refinement between COGENT's stateful update semantics and the generated C code must change. Above this low-level layer, the functional embeddings of COGENT still use high-level algebraic types, regardless of the specified layouts. In the following subsections, we discuss the extensions to the verification framework regarding:

(1) the correspondence between COGENT record values and C flat arrays;
(2) the correspondence between record operations in COGENT and C;
(3) the compositional properties of generated getters and setters;
(4) the correspondence between generated getters/setters and the specified layout.

These extensions to the verification framework not only ensure that the C code generated by the compiler is a refinement of the COGENT code, but also prove that COGENT types are laid out in C correctly according to the programmer's DARGENT specification. Again, all the results we present in this section are developed and checked in Isabelle/HOL.

The verification of any software system must assume the correctness of a *trusted computing base* (TCB). In COGENT systems, this TCB is largely comprised of externally provided C code [O'Connor et al. 2016]. While the COGENT framework allows for manual verification of this C code [Cheung et al. 2022], with DARGENT we eliminate large swathes of this C code entirely, specifically the so-called "glue code" which translates between data formats. This reduces the size of the TCB without imposing any additional verification burden.

4.1 Relating record values

The correctness of compilation is justified by a *refinement proof* that states that a simulation relation, called a *refinement relation*, holds between the source language and the target language. The refinement relation from the COGENT update semantics and the compiled C program is defined in terms of a relation between COGENT values (in the sense of the update semantics) and C values. Without DARGENT, this value relation is straightforward for records: a COGENT record relates to a C structure with the same fields, such that each COGENT field is itself value-related to the corresponding C field.

With layouts, a COGENT record now relates to (essentially) a flat word array, and each COGENT field relates to the result of applying the relevant custom getter to the array. This requires an appropriate embedding of our getters in Isabelle. To this end, we have implemented an Isabelle/ML procedure that imports the C getters and setters generated for each field using the AutoCorres library, which produces a monadic, shallow HOL embedding that models the semantics of the C functions. Our ML procedure then simplifies these embeddings to pure Isabelle functions. We call these simplified functions *direct getters* and *direct setters*. These direct getters and setters allow a simple statement of the compositional properties between getters and setters, detailed in Section 4.3.

4.2 Relating record operations

Without DARGENT, records directly compile to C structures, so it is trivial to relate COGENT record operations to their compiled C versions and show that they correspond. With layouts, the correspondence proof becomes slightly more involved, since it relates getting or setting fields in COGENT to calling custom getters or setters in C. These correspondence statements are proved automatically by custom tactics we have developed which exploit the compositional properties of getters and setters, detailed in the next section, as well as the relation between their monadic and direct embeddings.

4.3 Verifying the custom getters and setters

To relate record operations in COGENT and in C, getters and setters are expected to obey certain compositional laws, schematically summarised below. Given an array `arr` which models our record type, we have:

- **Roundtrip**: If a COGENT value x relates to a C value v, then it must also relate to the C value `get_a (set_a arr v)`. This is a weakening of the more intuitive roundtrip statement `get_a (set_a arr v) = v`, which does not always hold: As a counterexample, consider a boolean field. Since C does not natively provide such a 1-bit type, booleans are typically represented using a single byte `U8`. But a custom getter for a boolean field would always return 0 or 1,

picking the relevant bit as specified in the layout. Thus, we obtain a counter-example by taking $v = 3$.

- **Frame**: For distinct fields `a` and `b`, `get_a (set_b arr v)` is equal to `get_a arr`.

These statements are subject to typing constraints that we do not detail for brevity.

4.4 Ensuring getters and setters comply with layouts

As noted before, our compiler correctness certificate does not require that our generated getters and setters actually respect the layout specification—only that they satisfy the compositional properties stated previously. Thus, it remains to ensure that the generated getters and setters comply with data layouts specified by the user.

To this end, we generate specifications for getters in the deep embedding of COGENT, returning COGENT values (in the update semantics) from a word array, according to the layout. We designed tactics to automatically prove that generated getters respect this specification, using the value relations detailed previously. We have not yet done the same for setters, although the compositional properties we verify, together with the correctness of getters, already provides some formal guarantees. Additionally, we also automatically prove explicitly that setters do not change any bit that is outside the field layout.

4.5 Endianness

Since COGENT relies on the AutoCorres library, which does not support big-endian architectures yet, we assume that the machine is little-endian. Therefore, only big-endian annotations need to be seriously accounted for: big-endian fields of type `U16`, `U32`, and `U64` require reversing the bytes on retrieval and setting. This reversal is performed in dedicated optimised functions. As an example, here is the code for reversing the bytes in a `U32`:

```
static inline u32 swap_u32(u32 v) {
  v = ((v << 8) & 0xFF00FF00) | ((v >> 8) & 0xFF00FF);
  return (v << 16) | (v >> 16);
}
```

This additional reversal step does not require any extension to the proof tactics for the compositional properties of getters and setters described in Section 4.3, as they proceed first by unfolding all involved function definitions and applying an automated simplifier for bitwise operations, which can automatically prove that these swap operations are an involution. Recall that the value relation described in Section 4.1 is defined based on our generated custom getters, which already take endianness into account, so the value relation also naturally takes endianness into account.

The proof described in Section 4.4, that our getters respect the given layout, also must accommodate endianness. Specifically, we use a purely functional HOL embedding of the swap function to specify getters of big-endian values. Because our HOL embedding follows the C implementation closely, our proof tactics also need no extension for big-endian fields to prove correctness of getters.

We also prove as a sanity check that these swap functions are correct, in the sense that they reverse the byte order.

5 APPLICATIONS

In this section, we showcase how DARGENT helps in developing and formally verifying systems code via some small examples, the full source code of which is available in the supplementary material.

5.1 Power control system

To demonstrate the improved readability of programs that Dargent offers, we reimplemented the power control system for the STM32G4 series of ARM Cortex microcontrollers by ST Microelectronics, based on a C implementation from LibOpenCM3[3], an open-source low-level hardware library for ARM Cortex-M3 microcontrollers. The original C file[4] is about one hundred lines long, and consists of eight functions of a few lines each that perform some bitwise operations on the device register. The Cogent implementation is a bit smaller as each function is then one line long. The most involved part is the Dargent layout itself, which is derived from the hardware specifications of the device.

The 32-bit power control register holds several pieces of information, each occupying a certain number of bits in the register. We define the register as a record type in Cogent and use Dargent to prescribe the placement of each field. With a Dargent layout, the functions in the power control code no longer involve bitwise operations, but merely set values to individual fields. For example, the function to set the voltage output scaling bit in C:

```
void pwr_set_vos_scale(enum pwr_vos_scale scale) {
  uint32_t reg32;

  reg32 = PWR_CR1 & ~(PWR_CR1_VOS_MASK << PWR_CR1_VOS_SHIFT);
  reg32 |= (scale & PWR_CR1_VOS_MASK) << PWR_CR1_VOS_SHIFT;
  PWR_CR1 = reg32;
}
```

translates to the Cogent function setting the vos field in the record:

```
pwr_set_vos_scale : (Cr1, Vos_scale) → Cr1
pwr_set_vos_scale (reg, scale) = reg { vos = scale }
```

It can be seen from this example that Cogent allows systems programmers to write code on an abstract level, while retaining low-level control of the implementation details with Dargent.

5.2 Bit fields

In systems code, there is a regular pattern consisting in declaring a structure whose fields have non-standard bit widths, as in the following example taken from a CAN driver[5] where the first field is an identifier on 29 bits.

```
struct can_id {
  uint32_t id:29;
  uint32_t exide:1;
  uint32_t rtr:1;
  uint32_t err:1;
};
```

Although this is not supported by the AutoCorres library [Greenaway et al. 2014] on which the Cogent verification framework is based, we can simulate this feature using Dargent.

[3] http://libopencm3.org/
[4] http://libopencm3.org/docs/latest/stm32g4/html/pwr_8c_source.html
[5] https://github.com/seL4/camkes-vm-examples/blob/89f5d7b7ac373c8e9f000e80b91611e561358ef6/apps/Arm/odroid_vm/include/can_inf.h#L34

The last three fields are one bit long and can thus be handled with a COGENT boolean type. In order to cover the 29-bit field, we can use a primitive 29-bit integer U29. These non-word-size integers are a new extension we added to COGENT for this purpose, and consisted only of a dozen changed lines of code. We compile such integers to the smallest standard integer type that can contain the type, since the C language does not natively support such types. For instance, U7 is compiled to U8, while U20 is compiled to U32. Thanks to the extension of the value relation (see Section 4.1) to those new types, compiled COGENT programs maintain the invariant that C values always fit in the narrower bit-width.

With a DARGENT layout, we can define a COGENT version of the above C structure:

```
type CanId = { id : U29, exide : Bool, rtr : Bool, err : Bool }
  layout record { id : 29b, exide : 1b, rtr: 1b, err:1b}
```

Bit fields can play an important role in systems programming. This example demonstrates that DARGENT not only allows programmers to store data in a compact manner, but also provides a principled way to reason about C bit fields without needing to extend our C verification tools to support the bit field feature in C.

5.3 Custom getters and setters

Suppose we are writing a COGENT system that must work on a complicated, externally defined C structure that involves many fields, but the COGENT component is only concerned with a few specific fields.

Because DARGENT makes all record accesses go through getter and setter functions, we can reuse this mechanism by providing *custom* getter and setter functions in C, that extract (or write to) the relevant fields inside the large C structure. Then, we can represent this large C structure transparently as a simple COGENT record with only the relevant fields:

```
type Entry = {
  id : U32,
  value : U32
}
```

The custom getters and setters provided by the user must compose well (see Section 4.3).

We have successfully applied this approach on a small example, where the C structure has the same fields as the original COGENT type, extended with an additional field. Whilst this example is contrived, this feature does have real application: it makes it possible to *inherit* pre-defined C data structures. For example, in a previous COGENT implementation of a Linux `ext2fs` driver, Amani et al. [2016] had to define their own COGENT `Ext2Inode` type which corresponds to the standard C **`struct`** `ext2_inode` type, but did not include fields that were irrelevant to the COGENT implementation, such as certain unsupported file modes and spinlocks. This required tedious glue code to marshal back-and-forth between the representations. Using this technique, the standard C **`struct`** `ext2_inode` can be used directly as the representation of the `Ext2Inode` type, thus eliminating this glue code entirely.

6 VERIFICATION OF A TIMER DEVICE DRIVER

In this section, we present a formally verified COGENT timer driver (available in the supplementary material), that we ran successfully on an ODROID hardware, based on the seL4 operating system [seL4 Developers 2022]. The formal verification was conducted in Isabelle/HOL. Our formalisation took advantage of the fact that the shallow embedding of the COGENT program in

Isabelle/HOL remains simple, as the layouts are fully transparent to the functional semantics of the COGENT program.

Our COGENT implementation is based on a C driver[6] consisting of an interface for two timers, called A and E, provided by the device. Both implementations are about the same size (around 60 LoC, excluding type and layout declarations).

The driver can be used to:

- measure elapsed time since its initialisation, using the device timer E,
- generate an interrupt at the end of a (possibly periodic) countdown, using the device timer A.

The driver consists of four functions:

- `meson_init` initialises the device register;
- `meson_get_time` returns the elapsed time since the initialisation (in nanoseconds);
- `meson_set_timeout` sets the countdown value, making it possibly periodic;
- `meson_stop_timer` stops the countdown.

The driver state is passed around as a C structure called `meson_timer_t`, consisting of the location of the device registers in memory as well as a boolean flag remembering whether the countdown is enabled.

6.1 Using DARGENT

In the original C implementation, operations on the timer registers are largely based on bit-wise operations, which is unintuitive to read, error-prone, and more difficult to prove correct. Modelling the timer registers as algebraic data types, like records and sum types, makes the program easier to read and to reason about, while DARGENT still allows us control over low-level representation details.

With DARGENT, the timer register can be defined as:

```
type Meson_timer_reg = {
   timer_a_en : Bool,
   timer_a : U32,
   timer_a_mode : Bool,
   timer_a_input_clk : Timeout_timebase,
   timer_e : U32,
   timer_e_hi : U32,
   timer_e_input_clk : Timestamp_timebase
} layout record {
     timer_a_mode : 1b at 11b,
     timer_a_en : 1b at 15b,
     timer_a_input_clk : LTimeout_timebase,
     timer_e_input_clk : LTimestamp_timebase,
     timer_a : 4B at 4B,
     timer_e : 4B at 72B, -- 4*18,
     timer_e_hi : 4B at 76B
}
```

[6] https://github.com/seL4/util_libs/blob/c446df1f1a3e6aa1418a64a8f4db1ec615eae3c4/libplatsupport/src/plat/odroidc2/meson_timer.c

```
timer->regs->mux =
   TIMER_A_EN
 | (TIMESTAMP_TIMEBASE_1_US << TIMER_E_INPUT_CLK)
 | (TIMEOUT_TIMEBASE_1_MS << TIMER_A_INPUT_CLK);
```

```
regs {
   timer_a_en = True
 , timer_a_input_clk = TIMEOUT_TIMEBASE_1_MS
 , timer_e_input_clk = TIMESTAMP_TIMEBASE_1_US }
```

Fig. 8. The C version (above) and the COGENT version (below). The code enables the timer A, and selects the time units for the timers A and E. `TIMEOUT_TIMEBASE_1_MS` and `TIMESTAMP_TIMEBASE_1_US` are constant macro definitions in C, whereas in COGENT they can be modelled as constructors of a variant type, but compiled to 2-bit integers.

Contrary to the C definition, which is mostly type-less and relies on macros and bit-shifting to denote the semantics of the bits in use, the COGENT definition comes with a lot more type information, and also concisely prescribes how bits are used and placed. A typical set-value operation in C can thus be rewritten in a more abstract way in COGENT as record field updates, shown in Figure 8.

6.2 Verification

To formally verify the driver, we first wrote a purely functional specification of the driver. Then, we proved that the shallow embedding of the COGENT driver refines it. Both the specification and the manual functional correctness proof are approximately 150 lines each. The manual proof is accomplished straightforwardly by equational reasoning—much easier than reasoning about the bitwise operations as implemented in C. Layouts are fully transparent on the shallow embedding level: COGENT records are encoded as Isabelle records just as if no layout were specified.

This manual functional correctness proof composed with the automatically generated compiler certificate establishes the correctness of the compiler generated C code. All the tedium in the layout details is successfully hidden by our automatically generated compiler proofs.

6.3 Discussion

Even though this timer driver is a short program, our formal verification nonetheless uncovered several bugs or implicit assumptions that had been made in the original C driver.

Firstly, the original implementation of the initialisation function `meson_init` enabled the countdown timer A, without setting a starting value for it. The behaviour of the timer device in this case is unspecified. In fact, this countdown timer should only be enabled when used, i.e., in the `meson_set_timeout` function. Another related issue is that the initialisation function does not ensure that the disable flag of the driver state is synchronised with the enable flag of the device register, but rather assumes such. We introduced a specific invariant to the functional correctness specification of this function, to make this assumption explicit.

Additionally, we had to make explicit an assumption about the device state when verifying the function `meson_get_time`, namely that the timer value is not too big. While the device provides the timer value in micro-seconds, the function is specified to return the time in nano-seconds. The conversion requires a multiplication by one thousand, possibly triggering an overflow if the timer value is larger than approximately 500 years. Thus, we must add a precondition to rule out these cases.

6.4 Volatile behaviour

Although some device registers (such as the configuration register) behave as regular memory, other registers are volatile. In particular, the value of timer E may change between reading, so reading it twice may yield different values. The original C implementation of `meson_get_time` exploits this behaviour to detect a possible overflow when reading the lower bits of timer E.

Such volatile memory can be modelled by adapting the getter functions on those memory locations to return non-deterministic values. While COGENT functions are deterministic, non-determinism can be expressed by threading an additional abstract value through a deterministic version of the function, then abstracting that function to a non-deterministic one without that additional value. In particular, the getter function of a volatile register can be replaced by a function that takes an additional abstract type as input and returns a value of that type along with the value of the register. The shallow embedding of this getter function can then be abstracted to a function that ignores that additional input returns a non-deterministic value representing the true behaviour of the volatile register. We worked out a small example that sums two random numbers to clearly demonstrate this idea. The verification of this example illustrates that summing two random numbers can yield any value (not just even values).

7 RELATED WORK

The idea of describing low-level data layout with high-level languages is not new. The rich area of research makes it challenging to fully contextualise our work within the space. Simplifying our analysis a bit, we can roughly bifurcate the literature into research on *program synthesis* and *program abstraction*.

For instance, the PADS family of languages [Fisher and Gruber 2005; Fisher and Walker 2011; Mandelbaum et al. 2007], PacketTypes [McCann and Chandra 2000], Protege [Wang and Gaspes 2011], DataScript [Back 2002], Nail [Bangert and Zeldovich 2014], the generic packet description by van Geest and Swierstra [2017], the verified Protocol Buffer [Ye and Delaware 2019] built upon the NARCISSUS framework [Delaware et al. 2019], EverParse [Ramananandro et al. 2019], and contiguity types [Slind 2021] are all concerned with synthesising a parser program (and also a pretty-printer for some of them) from a high-level specification of the data format.

DARGENT's primary focus is on the data refinement of algebraic data types rather than securely operating on wire formats. Even though our technology shares a lot in common, the problem we try to solve is very different. In particular, DARGENT is not a language for parsing or converting between data formats. It is an extension to COGENT for its compiler to fine-tune the target code generation so that compiled code is already in the desired format that is suitable for systems software. In many cases, DARGENT can eliminate the need for such a data marshalling tool entirely.

Together with DARGENT in the program abstraction camp, LoCal [Vollmer et al. 2019], SHAPE [Franco et al. 2017, 2019] and `hobbit` [Diatchki and Jones 2006; Diatchki et al. 2005] are concerned with compiling data structures in a program into specific layouts dictated by the user. Programmers can therefore still work with high-level source code, while the compiler does the heavy-lifting to generate the low-level mechanisms, retaining the separation of program logic from low-level concerns.

LoCal [Vollmer et al. 2019] is a compiler for a first-order pure functional language which can operate on recursive serialised data by translation into an intermediate *location* calculus, LoCal, mapping pointer indirections of the high-level language to pointer arithmetic calculations on a base address. The final compiler output is C code which, interestingly, preserves the asymptotic complexity of the original recursive functions, although this property is implementation-defined and not assured by any formal theorem. Nonetheless, LoCal's type safety theorem does ensure a

form of memory safety: each location is initialised and written to exactly once. The latter property is a key difference to our work, since DARGENT can operate on mutable data by virtue of COGENT's linear types. On the other hand, COGENT is a total language and purposely lacks full support for recursion, we therefore do not yet support recursive layout descriptions. Primitive recursive types for COGENT are under development and use records (see Murray [2019]). Since we already support layout descriptions on records, we believe, once recursive types are supported, adding support for recursive layouts would be a straightforward engineering task. As a systems language, COGENT code often uses abstract types such as arrays and iteration constructs over such types. Arrays and iteration constructs over arrays were recently verified through COGENT's FFI [Cheung et al. 2022]. We have ensured these proofs work with our DARGENT extensions. The most significant difference with LoCal is that DARGENT is a *certifying* data layout language, with generated theorems that the translation is correct, whereas LoCal offers no verified guarantees regarding the final compiler output.

SHAPE [Franco et al. 2017, 2019] is an extension to an object-oriented language to fine-tune the layout of objects of classes for better cache performance. It allows users to define layout-unaware classes and specify what layout to use at object instantiation time. This class parameterisation mechanism shares some similarity with DARGENT's layout polymorphism. The layouts that SHAPE concerns with are primarily arrays of values, which are key to better cache locality but are not how compilers of managed languages natively represent data in memory. Their layouts are not down to the bit-level, but rather on the level of record fields. In contrast, DARGENT's layouts are lower-level and more flexible, and are less tailored for a specific optimisation. SHAPE's type system maintains memory safety properties of the program when it splits and lays out boxed data types. Therefore it also plays a similar role as COGENT's uniqueness type system. In our work, these two aspects are independently managed: DARGENT does not directly interfere with memory safety properties guaranteed by COGENT's type system.

The `hobbit` interpreter [Diatchki et al. 2005] extends a Haskell-like functional language with first-class support for bit-level types and operations (e.g. bit concatenation and splitting), supporting external representations of bit-level structures. Their work initially focused on bit-data that can be stored within a single register and later gets extended to memory areas realised as arrays [Diatchki and Jones 2006]. Instead of assigning a high-level type and a low-level layout to an object in memory, their types already prescribe the layouts, by virtue of the first-class bit-data support in the language. In that sense, it is more comparable to the bit-fields feature in the C language, or to COGENT if the DARGENT layout descriptors were subsumed by the COGENT types. Their research novelty also lies in using advanced type system features that are readily available in Haskell to encode the new language constructs and to perform sophisticated typechecking, which is arguably an orthogonal matter to data layouts.

Floorplan [Cronburg and Guyer 2019] is also somewhat relevant to DARGENT, but hardly fits in either category. It is a memory layout specification language for declaratively describing the structure of a heap as laid out by a memory manager. It therefore chiefly serves the implementors of memory managers rather than systems developers and users in general, and the abstraction it provides does not necessarily extends to algebraic types of heap objects. The compiler follows the specification to generate memory safe Rust code to perform common tasks that are needed in the implementation of a memory manager. The semantics of a heap layout specification is denoted by the set of values that the heap can take. In contrast, the semantics of a DARGENT layout is characterised by the getter and the setter functions.

8 CONCLUSION

Systems code must adhere to stringent requirements on data representation to achieve efficient, predictable performance and avoid costly mediation at abstraction boundaries. In many cases, these requirements result in code that is error prone and tedious to write, ugly to read, and very difficult to verify.

We are not without hope, however, as we have demonstrated in this paper. By using Dargent, we can avoid the need for having the *glue* code (be it manually written or synthesised) that marshals data from one format into another, and eliminate error-prone bit-twiddling operations for manipulating specific bits in device registers. Instead, we enable programmers to provide declarative specifications of how their algebraic datatypes are laid out. Given these specifications, our *certifying* compiler generates corresponding C code that operates on these data types directly, along with proofs that the generated code is functionally correct. We have shown the applicability of Dargent on a number of examples showcasing its support for low-level systems features including the formal verification of a timer device driver.

REFERENCES

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 175–188. https://doi.org/10.1145/2872362.2872404

Anonymous. 2018. Bringing Effortless Refinement of Data Layouts to Cogent. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, 134–149.

Godmar Back. 2002. DataScript - A Specification and Scripting Language for Binary Data. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GPCE '02)*. Springer-Verlag, London, UK, UK, 66–77. http://dl.acm.org/citation.cfm?id=645435.652647

Julian Bangert and Nickolai Zeldovich. 2014. Nail: A Practical Tool for Parsing and Generating Data Formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 615–628. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert

Erik Barendsen and Sjaak Smetsers. 1993. Conventional and Uniqueness Typing in Graph Rewrite Systems. In *Foundations of Software Technology and Theoretical Computer Science (Lecture Notes in Computer Science, Vol. 761)*. 41–51.

Zilin Chen, Liam O'Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. 2017. The Cogent Case for Property-Based Testing. In *Workshop on Programming Languages and Operating Systems (PLOS)* (2017-10-28). ACM, Shanghai, China, 1–7. https://doi.org/10.1145/3144555.3144556

Louis Cheung, Liam O'Connor, and Christine Rizkallah. 2022. Overcoming Restraint: Composing Verification of Foreign Functions with Cogent. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. New York, NY, USA, 13–26. https://doi.org/10.1145/3497775.3503686

Karl Cronburg and Samuel Z. Guyer. 2019. Floorplan: Spatial Layout in Memory Management Systems. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Athens, Greece) *(GPCE 2019)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3357765.3359519

Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats. *Proc. ACM Program. Lang.* 3, ICFP, Article 82 (jul 2019), 29 pages. https://doi.org/10.1145/3341686

Iavor S. Diatchki and Mark P. Jones. 2006. Strongly Typed Memory Areas Programming Systems-Level Data Structures in a Functional Language. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell* (Portland, Oregon, USA) *(Haskell '06)*. Association for Computing Machinery, New York, NY, USA, 72–83. https://doi.org/10.1145/1159842.1159851

Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. 2005. High-Level Views on Low-Level Representations. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. Association for Computing Machinery, New York, NY, USA, 168–179. https://doi.org/10.1145/1086365.1086387

Kathleen Fisher and Robert Gruber. 2005. PADS: a domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 295–304. https://doi.org/10.1145/1065010.1065046

Kathleen Fisher and David Walker. 2011. The PADS project: an overview. In *Proceedings of the 14th International Conference on Database Theory* (Uppsala, Sweden) *(ICDT '11)*. ACM, New York, NY, USA, 11–17. https://doi.org/10.1145/1938551.1938556

Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. 2017. You Can Have It All: Abstraction and Good Cache Performance. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 148–167. https://doi.org/10.1145/3133850.3133861

Juliana Franco, Alexandros Tasos, Sophia Drossopoulou, Tobias Wrigstad, and Susan Eisenbach. 2019. Safely Abstracting Memory Layouts. https://doi.org/10.48550/ARXIV.1901.08006

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

Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. 2007. PADS/ML: a functional data description language. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Nice, France) *(POPL '07)*. ACM, New York, NY, USA, 77–83. https://doi.org/10.1145/1190216.1190231

Peter J. McCann and Satish Chandra. 2000. PacketTypes: abstract specification of network protocol messages. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Stockholm, Sweden) *(SIGCOMM '00)*. ACM, New York, NY, USA, 321–333. https://doi.org/10.1145/347059.347563

Emmet Murray. 2019. *Recursive Types for Cogent*. Bachelor's Thesis. https://github.com/emmet-m/thesis Accessed November 2021.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg.

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

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming* 31 (2021), e25. https://doi.org/10.1017/S095679682100023X

Blaise Paradeza. 2020. *Refinement Types for Cogent*. Bachelor's Thesis. https://people.eng.unimelb.edu.au/rizkallahc/theses/blaise-paradeza-honours-thesis.pdf Accessed Feb 2022.

Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *USENIX Security*. USENIX. https://www.microsoft.com/en-us/research/publication/everparse/

Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. 2016. A Framework for the Automatic Formal Verification of Refinement from Cogent to C. In *International Conference on Interactive Theorem Proving*. Nancy, France.

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

seL4 Developers. 2022. *The seL4 Microkernel*. https://sel4.systems/

Konrad Slind. 2021. Specifying Message Formats with Contiguity Types. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:17. https://doi.org/10.4230/LIPIcs.ITP.2021.30

Marcell van Geest and Wouter Swierstra. 2017. Generic Packet Descriptions: Verified Parsing and Pretty Printing of Low-Level Data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development* (Oxford, UK) *(TyDe 2017)*. Association for Computing Machinery, New York, NY, USA, 30–40. https://doi.org/10.1145/3122975.3122979

Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: A Language for Programs Operating on Serialized Data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 48–62. https://doi.org/10.1145/3314221.3314631

Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*.

Yan Wang and Verónica Gaspes. 2011. An embedded language for programming protocol stacks in embedded systems. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation* (Austin, Texas, USA) *(PEPM '11)*. ACM, New York, NY, USA, 63–72. https://doi.org/10.1145/1929501.1929511

Qianchuan Ye and Benjamin Delaware. 2019. A Verified Protocol Buffer Compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 222–233. https://doi.org/10.1145/3293880.3294105