DARGENT

A Silver Bullet for Data Layout Refinement

ANONYMOUS AUTHOR(S)

Systems programmers tend to need fine grained control over the memory layout of data structures to produce highly performant code and to be compliant to well-defined interfaces imposed by existing code, standardised protocols or hardware. These low-level representations and the code operating on it are hard to get right. Traditionally, this problem is addressed by implementing tedious marshalling code to convert between the loosely packed data types that the compiler chooses and the desired compact data formats. The marshalling code is error-prone and can lead to a significant runtime overhead due to excessive copying. There are many languages and systems that address the correctness issue by automating the generation and, in a few, the verification of the marshalling code. This still leaves programmers with the performance overhead introduced by the marshalling code. In particular for systems code, this overhead can be prohibitive. We address both the correctness and the performance problem.

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 has the desired memory layout, while retaining the formal proof that its 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). 2022. DARGENT: A Silver Bullet for Data Layout Refinement. In . ACM, New York, NY, USA, 25 pages.

1 INTRODUCTION

In the realm of software systems, such as device drivers, file systems, and network stacks, having complete control over the *data layout* of objects is crucial for compatibility and performance. Specifically, controlling the bit- and byte-level composition of objects can avoid the need for *deserialisation* between the on-medium and in-memory boundaries which often arise from interacting with standardised protocols or hardware. Systems code is often developed in the C language using low-level features to define and manipulate the data layouts of objects. Unfortunately, such C code may consist of subtle bit twiddling operations which, apart from being difficult to manually verify, are also tedious and error-prone to implement. Worst of all, such code is not centered around a conceptual structuring of the problem but rather the low-level details required to maintain good performance.

To maintain the higher-level structure of a program without sacrificing performance, we aim to use a high-level language with facilities for specifying the 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 [6, 24]. Nonetheless, these languages do not provide verified correctness of their translations.

In this paper, we propose DARGENT, a language for describing data layouts of high-level *algebraic datatypes* along with a data refinement framework for *automatically verifying* those layout descriptions. We build on the COGENT language and refinement framework [17] (Section 2). COGENT is

ICFP 2022, 11 – 16 September 2022 , Ljubljana, Slovenia
2022.

designed for implementing 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 [15] theorems witnessing a proof that the C program refines an Isabelle/HOL embedding of the COGENT source program.

The data layout descriptions used in the DARGENT language (Section 3) may look similar to those used in the line of work on *data description languages* [3, 9, 13, 19, 22, 23, 26, 27]. However, 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, 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., language 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 indented to solve a different problem. The DARGENT data layout descriptions grant programmers the ability to dictate the COGENT compiler how it should lay out algebraic data types. The compiler generates code that obeys the layout descriptions specified by the programmer as well as Isabelle/HOL proofs showing that it has done so correctly. In this setup, (de)serialisation becomes unnecessary removing both the performance as well as manual verification overhead.

This additional power in expressiveness can, therefore, be utilised to improve the performance of the compiled COGENT code, e.g., by having smaller memory footprints or 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.

Depending on the application, DARGENT can also be used to eliminate or reduce the need for data (de)serialisation code, be it manually written or automatically derived, when interacting with the external world. Instead of generating serialisers and deserialisers, like the aforementioned list of works do, DARGENT instructs the COGENT compiler to internally represent algebraic data types directly in their binary data formats. Therefore the programmer does not need to decode the raw data in order to operate on it algebraically, resulting in more concise and readable code, better performance, and easier informal and formal reasoning.

COGENT is readily amenable to an extension for prescribing data layouts and fine-tuning the compilation of algebraic data types by virtue of its non-existent runtime system and direct compilation to C; it currently adopts the layout conventions of the underlying C compiler. By introducing DARGENT into the framework (Section 3), we have been able to improve upon some outstanding inefficiencies in the prior design, including reducing reliance on *deserialisation* code within file system implementations [1] and directly representing device register formats (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 realizes the vision set out in 2018 [2]. 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 [15], 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 Other Supplement.

2 OVERVIEW OF COGENT

COGENT [17] 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 [4, 25] 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 [20] 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 [1]. 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 [25]. 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 retrive 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 unsigned *n*-bit integer types, U8, U16, U32 and U64, and booleans (Bool). 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 employes a *structural* type system, meaning that types, as well as subtyping relations, are defined solely on their structures. This is contrary to a *nominal* type system, where type names play a more important role. *Type synonyms* can be defined in COGENT, but they are only 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 sigil can either be a writable boxed sigil (`w`) or a read-only one (`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.

The COGENT compiler uses a pre-defined code generation algorithm to compile data types to C. A record type in COGENT is 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, is less obvious. COGENT's verification tool chain does not currently support C unions, therefore we generate variants as structs in C, which contains a field for the tag, and a field for each alternative's payload.

Applying a fixed code generation scheme is no surprise for a typical high-level functional language, whose implementation details are hidden from the language users. COGENT, besides being a functional language, it is also a systems language, which makes the low-level representation of data types relevant to the end users. This fixed code generation algorithm often results in suboptimal or undesired representations of COGENT types in C, and users of the COGENT language have 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 a COGENT program is developed to interoperate with pre-existing C components, say, when developing an operating system component that interfaces with the Linux kernel headers, some glue code is often required, resulting in development and run-time overhead. Also, problematically, as the COGENT compiler evolves, previously working code may break 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.

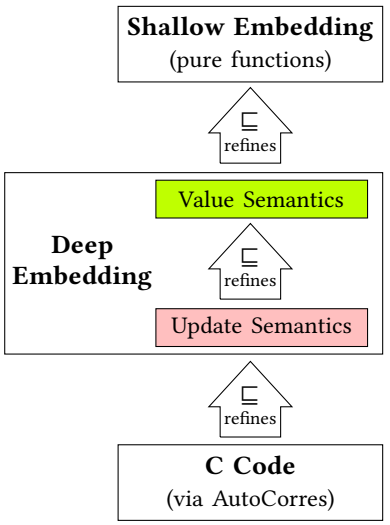

Fig. 1. The COGENT refinement framework.

The last item relies on the AutoCorres library [11] to generate a representation of the C code in Isabelle/HOL. More precisely, the AutoCorres library abstracts the C semantics via a SIMPL [21] 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 reasoning *equationally* about the HOL term and 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. To summarise, 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

```
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

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 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 specifies 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, while it is not associated to any particular algebraic data types. From this perspective, DARGENT descriptions are independent of COGENT types. But any COGENT type can only be laid out in a certain manner, therefore there are restrictions on what layout can be assigned to a COGENT type. To establish the connection between the two, a layout description needs to be attached to a COGENT type. We will see how to do it shortly.

When specifying a layout, two pieces of information are relevant: how much space a type occupies in memory and where it is placed in relation to the type that it is contained in. 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

[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 more on the implications of the choice of the word size later in Section 3.4.

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

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

Fig. 3. DARGENT syntax

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, block memory of word sizes (e.g. 1 byte, 2 bytes, 4 bytes and 8 bytes) can be given an endianness (BE or LE), with the using keyword.

A boxed type must be described as a pointer layout, and not as a chunk of memory. The use of this special layout is both for portability and readability of code. 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. Seeing as we can specify memory blocks down to the individual bits, we can naturally represent records of boolean values as a bitfield:

$$\textbf{layout}\ \mathsf{Bitfield} = \mathsf{record}\ \{\mathrm{x} : 1\mathsf{b}, \mathrm{y} : 1\mathsf{b}\ \mathsf{at}\ 1\mathsf{b}, \mathrm{z} : 1\mathsf{b}\ \mathsf{at}\ 2\mathsf{b}\}$$

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:

$$\textbf{layout}\ \mathsf{Bitfield} = \mathsf{record}\ \{\mathrm{x} : 1\mathsf{b}, \mathrm{y} : 1\mathsf{b}\ \mathsf{after}\ \mathrm{x}, \mathrm{z} : 1\mathsf{b}\ \mathsf{after}\ \mathrm{y}\}$$

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.

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*

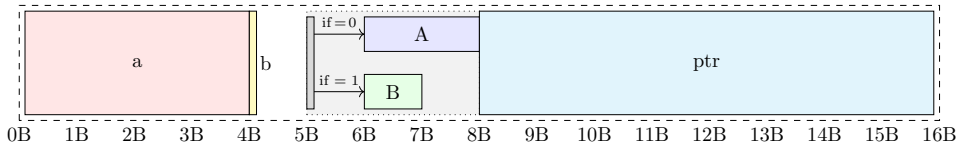

Fig. 4. The *ExampleLayout*, visualised.

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, in a similar manner in which 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 synonyms can be parametric.

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
}
```

A pictorial illustration of the memory layout is given in Figure 4. 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, e.g. it does not have overlapping fields, the tag values are distinct, etc. We can write Example **layout** ExampleLayout so that this layout is attached to the type Example, and the compiler will subsequently also check that it is an appropriate layout for the type it is describing. We will talk more about the wellformedness and matching rules later in the section (Section 3.3). To reduce the level of verbosity, a type synonym can be given to the layout-annotated type above.

A natural extension is to support *layout polymorphism* in COGENT functions. 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, as type variables. Constraints on layout variables are allowed. For now, there is only one type of constraint on layout variables: that a layout variable matches a type. [2] For

[2] A formal definition of "match" will be discussed later in Section 3.3.

example, in the code snippet below,

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

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 must 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. Additionally, what is not so explicit in the type signature is that any instantiation of ℓ also needs to ensure that all the layouts in the function 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 lay the snd field right after fst.

We make the design choice of having DARGENT layouts syntactically separate from the COGENT types, and relate them with the **layout** keyword in the language and with the layout-type matching rules (as we will see in Figure 6) in the type checker. This design may seem unintuitive and unproductive, as some common information is duplicated in both the types and the layouts, and matching them requires a set of dedicated typing rules. We make this design based on several concerns.

Firstly, the COGENT types are central to the semantics of a COGENT program, especially its functional semantics and the reasoning of the function correctness, whereas the layout of algebraic data types is an implementation detail, 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 are unimportant. Also, there is ongoing work towards adding refinement types to COGENT [18]; a refined type could possibly be laid out in a smaller memory area. For instance, $\{\nu : \mathtt{U16} | \nu < 2\}$ 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 architecure and environment, can decide the layouts and plug them into the COGENT programs.

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

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

Fig. 5. Syntax for DARGENT core language

Last, but not least, 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 compiler in the typechecker. The compiler does not only compiles the COGENT programs, but also generates many embeddings of the program in Isabelle/HOL and Haskell [5], proof tactics, etc. 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 irrevelant parts of the compiler.

Choosing to keep layouts and types separate, however, does not preclude us from implementing syntactic sugars to eliminate information duplication and improve the user experience of the language. We leave it as future work.

3.2 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.

In the core language, *bit ranges* are used to represent which bits in memory are used to store each piece of data. The core calculus is parametric over the bit range representation. Along the compilation pipeline, the bit ranges are represented differently, to suit the purpose of that particular phase of the compilation.

For primitive layouts in the core language, an endianness will be given explicitly. When the endianness is unspecified in the surface language, a *machine endianness* will be given by default, denoted as 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 first get 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 datatype* in which it is contained, as well as the range occupied (s bits). In BitRange (o, s), we require $s > 0$ for convenience, and introduce a dedicated empty layout $()$.

The bit range for layout variables is of the form $x\{o\}$ as we need to remember the offset to be applied to it. 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 boxed structure;
- 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:

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

We write $\mathtt{b}_\ell$ when we don't distinguish whether it is writable or readonly. Layout annotations can be left out and the compiler will use the default layout. In case a layout is omitted, the sigil is denoted as $\mathtt{b}$.

3.3 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 (see [16] for more details). In this section, we explain how these formal proofs and definitions are extended to take DARGENT into account.

The original standard typing judgement $\Delta; \Gamma \vdash e : \tau$ keeps track of the linearity of type variables in τ in an additional context Δ. COGENT also has a wellformedness judgement on types, $\Delta \vdash \tau\ \mathbf{wf}$, which ensures that types respect the linearity constraints. In the presence of DARGENT, the typing judgement must additionally ensure that COGENT types match their assigned layouts (Figure 6) and that the layouts of components of a type do not overlap (Figure 7). These additional constraints, however, cannot always be immediately ensured at the type checking phase as COGENT supports types that may contain polymorphic type variables, and their layouts may also be polymorphic. The typing judgement $\Delta; C; \Gamma \vdash e : \tau$ and wellformedness judgement $\Delta; C \vdash \tau\ \mathbf{wf}$ must, therefore, keep track of an additional set of layout constraints C. The set C tracks polymorphic types and their assigned layouts. When the types and layouts are specialised through substitution, the specialised constraints are checked.

As far as the matching relation between a type and a layout is concerned, linearity constraints are irrelevant. In fact, for the type wellformedness relation to be preserved by substitution (Lemma 3.1 below), type wellformedness must remain stable regardless of any linearity constraints. As such, the set C contains pairs of *simplified types* $\hat{\tau}$ and layouts ℓ, where $\hat{\tau}$ is essentially the type τ with the linearity information erased. Types are simplified by replacing all boxed components of the type with a newly introduced pointer type constant.

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

$$\frac{\overline{\mathrm{f}_i}\ \text{distinct} \qquad \Delta; C \vdash \tau_i\ \mathbf{wf} \qquad \ell\ \mathbf{wf} \qquad \ell \sim \text{record}\ \{\overline{\mathrm{f}_i : \hat{\tau}_i}\} \qquad (\ell, \text{record}\ \{\overline{\mathrm{f}_i : \hat{\tau}_i}\}) \in C}{\Delta; C \vdash \{\overline{\mathrm{f}_i :: \tau_i^?}\}\mathtt{b}_\ell\ \mathbf{wf}}\ \textsc{RecWf} \tag{1}$$

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

$$\frac{\text{bits}\ (T) = s \qquad T \text{ is a primitive type}}{\text{BitRange}\ (o, s) \sim T}\textsc{PrimTyMatch}$$

$$\frac{M \text{ is the pointer size}}{\text{BitRange}\ (o, M) \sim \text{pointer}}\textsc{BoxedTyMatch}$$

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

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

$$\frac{}{x\{o\} \sim \hat{t}}\textsc{VarMatch} \qquad \frac{}{\ell \sim \alpha}\textsc{VarTypeMatch}$$

$$\frac{}{() \sim ()}\textsc{UnitMatch} \qquad \begin{aligned} \text{bits}\ (\text{U8}) &= 8 \\ \text{bits}\ (\text{U16}) &= 16 \\ \text{bits}\ (\text{U32}) &= 32 \\ \text{bits}\ (\text{U64}) &= 64 \\ \text{bits}\ (\text{Bool}) &= 1 \end{aligned}$$

Fig. 6. Matching relation between layouts and types

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

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

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

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

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

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

Fig. 7. Wellformed layouts

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, ℓ **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.

3.3.2 Specialisation. COGENT's type preservation proof relies on type specialisation maintaining welltypedness [16, 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**, both $\ell[\overline{l_j/x_j}] \sim \hat{t}[\overline{\hat{\rho}_i/\alpha_i}]$ and $(\ell[\overline{l_j}/\overline{x_j}], \hat{\tau}[\overline{\hat{\rho}_i}/\overline{\alpha_i}]) \in C'$ hold.*

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.

3.3.3 Future extensions. Our type system currently does not check that the constraints in C are compatible. For example, nothing forbids that C contains both $(x\{o\}, \mathrm{U8})$ and $(x\{o\}, \mathrm{U16})$, although in that case the layout variable x can never be fully instantiated, because of the wellformed rule for substitution mentioned above. Note that this does not allow unsafe behaviour, since C code generation only involves completely monomorphised layouts and types. We leave for future work the design of a (possibly partial) validator that would deal with such situations and could raise a typing error as early as possible before instantiation.

3.4 Compiling records: custom getters and setters

Currently, records are the only types that our certifying compiler supports in COGENT that can be boxed. We therefore use records as an example to discuss how DARGENT layouts influence the generated C code. Similar techniques can be applied to other algebraic data types, such as variants and arrays, and we leave them as future work. The technical challenges in supporting these types are actually more on the uniqueness type system, which is chiefly an orthogonal matter than the data layout that we present in this paper.

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. This

does not have to be the case though. It can be easily made configurable to any word sizes (8, 16, 32 or 64 bits). It is worth mentioning that it does not mean that a layout has to be of multiples of words in the size. It is absolutely valid to have a record layout like record {a : 1B, b : 3b}, totally 11 bits. 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 having any implicit paddings. Only when a boxed type has an unaligned layout will the padding be added implicitly. This is in line with how C and the underlying architecture process data.

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 type 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. In future, we plan to extend COGENT's uniqueness type system to allow for pointer access to unboxed types, improving the run-time performance and reducing reliance on the C optimiser.

Invalid bit patterns. Calling a getter on an external word array can lead to unexpected behaviours when the data format is invalid. It 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) $\{\mathsf{A}(0) : 1B, \mathsf{B}(1) : 2\mathsf{B}, \mathsf{C}(2) : 4\mathsf{B}\}$ 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 is 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 §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.

3.5 Implementation

As mentioned in Section 3.2, 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 the both the formalisation of the type system and the C code generation of the compiler. However, it also affects the generated correspondence proof between the C code and the pure shallow embedding in Isabelle/HOL. Only the interface between Cogent's stateful update semantics and the generated C code needs to change. Above this low-level layer, the functional embeddings of Cogent still use 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 the Cogent types are laid out in C as per the Dargent specification. Again, all the results we present in this section are developed and checked in Isabelle/HOL.

Cogent's certifying compilation framework relies on a minimal trusted computing base (TCB) [16]. With Dargent, which instructs the compiler to directly produce the specified layout, we are able to remove the need for the data marshalling code, which typically involves manually written C code. This C code, until manually verified, will have to be trusted. Removing them will further reduce the TCB of the overall system.

Finally, let us mention that, although endianness annotations are fully taken into account by the compiler when generating C code, the automatic refinement framework has not yet been accordingly extended and will fail to generate an automatic refinement proof when these annotations are used. We plan to extend our refinement framework to produce proofs in the presence of endianness annotations in future work.

4.1 Relating record values

The correspondence [16, Definition 2] between Cogent update semantics and the compiled C program relies on a notion of relation between Cogent values (in the sense of the update semantics) and C values. Without Dargent, this value relation is straightforward: a Cogent record relates to a C structure with the same field number and names, and 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 and setter on the array. This requires an appropriate embedding of custom getters in Isabelle. To this end, we implemented an ML procedure that takes as input the *monadic* embeddings provided by the AutoCorres library, on which Cogent is based, and simplifies them to pure Isabelle functions. We call these simplified functions *direct getters* and *direct setters* in the following. These direct getters and setters allow a simple statement of the compositional properties between getters and setters, detailed in §4.3.

4.2 Relating record operations

Without Dargent, records directly compile to C structures. In the verification proofs, it is straightforward to relate Cogent record operations to their compiled C versions and show that they

correspond (in the sense that they produce related outputs when given related inputs). With layouts, the correspondence proof becomes more involved, since it relates getting or setting fields in COGENT to calling custom getters or setters in C. The adapted statements are proved automatically by tactics that exploit the nice compositional properties of getters and setters, as well as the relation between their monadic and direct embeddings.

4.3 Verifying the custom getters and setters

In the proof of correspondence relating record operations in COGENT and C, getters and setters are expected to obey certain composition properties, as schematically summarised below:

(1) 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`)`; and
(2) 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. The first one weakens the more intuitive `get_a (set_a arr` v`) = ` v, which does not always hold. Indeed, consider a boolean field. Since C does not natively provide such a 1-bit type, booleans are typically embedded using the type `U8`. But the custom getter for a boolean field would always return 0 or 1, picking the relevant bit as specified in the layout. We, thus, obtain a counter-example by taking $v = 3$.

4.4 Additional checks for custom getters and setters

As noted before, the refinement proof from C to the shallow embedding does not require custom getters and setters to respect the layout specification: only their compositional properties stated in the previous subsection matter. Writing a generic specification of getters and setters once and for all, that does not need to be generated for each program (and carefully checked by the user), requires a deep embedding of them. We thus provide a specification of getters in the deep embedding of COGENT, returning values in the update semantics from a word array, according to the layout. We designed tactics showing that generated getters respect their specification, using the value relations. The case of setters remain to be done, although the compositional properties together with the correctness of getters already provides some formal guarantees. Meanwhile, we designed automatic tactics to prove explicitly that the setters do not flip any bit that is outside the field layout.

5 APPLICATIONS

In this section, we showcase how DARGENT helps in developing and formally verifying systems code via some small examples, that are 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 consists in writing the DARGENT layout from the device specification.

The 32-bit register holds several pieces of information, each occupying a certain number of bits in the register. We can define the register as a record type in COGENT and use DARGENT to prescribe

[3] http://libopencm3.org/
[4] http://libopencm3.org/docs/latest/stm32g4/html/pwr_8c_source.html

the placement of each field. Once done, the functions in the power control code are merely setting 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, and DARGENT retains the low-level control of the implementation details that systems programmers desire.

Dealing with volatile registers. This example also demonstrates a current limitation of our system: the verification framework does not provide any guarantee on the exact access pattern of getters and setters. This is however crucial when dealing with hardware registers, since the write (or read) order matters. Consider, for example, the following COGENT function from the power control system.

```
pwr_enable_power_voltage_detect : (Cr2, Pvd_level) → Cr2
pwr_enable_power_voltage_detect (reg, pls) = reg { pls = pls, pvde = True }
```

This code actually compiles to two subsequent writes to the register CR2: it first sets the programmable voltage detector level `pls` and then enables the programmable voltage detector (by setting the boolean flag). Note that the reverse order would be problematic, as the detector would run with an unknown selection level between the two writes. To avoid this kind of unwanted behaviour, sequentialisation is usually avoided when dealing with hardware registers. In fact, the original C implementation performs a single write to the register:

```
void pwr_enable_power_voltage_detect(uint32_t pvd_level)
{
  uint32_t reg32;

  reg32 = PWR_CR2;
  reg32 &= ~(PWR_CR2_PLS_MASK << PWR_CR2_PLS_SHIFT);
  PWR_CR2 = (reg32 | (pvd_level << PWR_CR2_PLS_SHIFT) | PWR_CR2_PVDE);
}
```

In future work, we plan to extend COGENT so that it can perform multiple field updates/reads in a single operation, when possible.

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 [11] on which the COGENT verification framework is based, we can simulate this feature using DARGENT.

The last three fields are one bit long and can thus be handled with the COGENT boolean type. However there is no built-in type that can be used for the 29-bit field (recall the match rules in §3.3). Thankfully, COGENT provides a modular mechanism to introduce new types, as abstract types, whose definitions must be provided by the programmer in C. The verification framework must also be extended to take into account abstract values for these types. For example, in the shallow embedding, we embed U29 as the Isabelle type of natural numbers that fit in 29 bits.

By first declaring the abstract type U29, 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}
```

Abstract types are implicitly boxed, hence the additional unboxed annotation # in front of U29.

Abstract types alone are not very useful since no COGENT operation applies to them. We thus extend our example with back and forth casts between U29, and U32, declared in COGENT as *abstract functions*. This means that we implement them in C, and we also provide their semantics at each layer of the refinement proof, and prove properties about them [16, §3.3.2].

Although some manual verification effort is required (about 700 lines of Isabelle/HOL proof) on these irregular-sized integers and the cast functions, it is only a one-off effort, and they can be mechanically produced. We leave it as future work to have native support for these integer types from the COGENT language and the compiler.

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 the need for extending the verification tools to support the bit field feature in C.

5.3 Customising the encoding of records

In this section, we present a use case that is permitted thanks to the new structure of the verification framework, by abstracting the expected properties of getters and setters. As a consequence, we can use a representation of COGENT records in C different from what the compiler would generate (word arrays), as long as we provide getters and setters that compose well (see §4.3).

Consider, for example, some COGENT code that operates on a COGENT record

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

[5] https://github.com/seL4/camkes-vm-examples/blob/89f5d7b7ac373c8e9f000e80b91611e561358ef6/apps/Arm/odroid_vm/include/can_inf.h#L34

```
  value : U32
}
```

Suppose that actually, we would like the compiled code to operate on a more complicated C type where the two fields above are encoded somewhere. If we provide encoding / decoding functions for these fields, we can exploit Dargent and compile Cogent record operations to use them thanks to the following process.

(1) Force the generation and use of custom getters and setters (by assigning the `Entry` type a dummy layout).
(2) Replace in the compiled code the generated C type (word array) by the desired C structure.
(3) Replace the body of custom getters and setters with a call to the decoding / encoding functions.

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* a pre-defined C data structure such as an **`struct`** `ext2_inode` from the Linux ext2fs implementation and Cogent can omit fields which it does not care about, *e.g* certain file modes it does not support, or spinlocks. Without Dargent, Amani et al. [1] had to define their own `Ext2Inode` type in Cogent and implement marshalling code back-and-forth between the representations. In future work, we plan to improve the compiler to make this approach more user-friendly.

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 [7]. The formal verification is 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] consists of an interface for the so-called timers 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] https://github.com/seL4/util_libs/blob/c446df1f1a3e6aa1418a64a8f4db1ec615eae3c4/libplatsupport/src/plat/odroidc2/meson_timer.c

6.1 Using DARGENT

In the C implementation, operations on the timer registers are largely based on bit-wise operations, which is unintuitive to read, error-prone, and will not favour easy reasoning of the program's correctness. Modelling the timer registers as algebraic data types like records and sum types in a high-level programming language will make the program easy to read and potentially easy to reason about, but these high-level languages normally do not let the programmer to choose how to compile the algebraic data types down to the target language to meet the requirement from the drivers' end. DARGENT offers the capability of modelling the program in a high-level manner, whereas retaining low-level control in the implementation 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
}
```

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 showed that the shallow embedding of the COGENT driver refines it. Each of these two steps is about 150 lines long. The manual proof can be done straightforwardly using equational reasoning, and is much easier than reasoning about the bitwise operations as implemented in C. Layouts, in the COGENT version, are fully transparent on the shallow embedding level: COGENT records are encoded as Isabelle records in the same way 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.

```
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.

6.3 Discussion

Our formal verification provided us with an opportunity revealing implicit assumptions in the original C driver and make such assumptions explicit.

The first one is that the initialisation function `meson_init` enables 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 `meson_set_timeout`. 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. We introduced a specific invariant to make this assumption explicit.

Finally, we had to make an explicit assumption about the device state when verifying the function `meson_get_time`, namely that the timer value is not too big. Indeed, the function returns the time in nano-seconds, whereas the device provides it in micro-seconds. The conversion requires a multiplication by one thousand, possibly triggering an overflow (which however would not occur before 500 years).

6.4 Volatile behaviour

Although some of the device registers (such as the configuration register) behave as regular memory, some of them do not, in particular the value of timer E: reading it twice may yield different values. The original function `meson_get_time` exploits this behaviour to detect a possible overflow, when reading the lower bits of timer E. In this formalisation, we have not modelled these volatile features. As mentioned in Section 5.1, we plan to extend COGENT to handle them more carefully in future work.

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, languages such as the PADS family [9, 10, 12], PacketTypes [13], Protege [26], DataScript [3], the generic packet description by van Geest and Swierstra [23], verified Protocol Buffer [27], EverParse [19], and contiguity types [22] are concerned with synthesising a parser program (and also a pretty-printer for some of them) from a high-level specification of the data

format. On the other hand, the work by Diatchki et al. [8], LoCal [24], Floorplan [6] and DARGENT are concerned with providing high-level descriptions of layouts which are compiled into the low-level arithmetic required by the application.

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, converting data formats. It is an extension to COGENT for its compiler to fine-tune the target code generation so that compiled code is in a certain format that is suitable for systems software. In many cases, DARGENT can eliminate the need for such a data marshalling tool.

Of the existing literature we surveyed, LoCal is perhaps most comparable to DARGENT. Vollmer et al. [24] present 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 between our work and Vollmer et al. [24]'s since DARGENT can operate on mutable data by virtue of COGENT's linear types. On the other hand, due to COGENT's lack of full support for recursion we cannot yet define recursive layout descriptions but we believe this to be largely an engineering issue, c.f. [14].

The most significant difference is that DARGENT is a certifying data layout language with generated theorems that the translation is correct. Whereas, with LoCal, there are no verified guarantees regarding the final compiler output and the flexibility afforded to the compiler to modify data types and functions exacerbates the situation.

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 getting and setting specific bits in device registers. Instead, we enable programmers to provide declarative specifications of how their algebraic datatypes are laid out. This guides *certifying* compiler to generate code that manipulates C code directly along with proofs that its generated code is correct. We demonstrate DARGENT on a number of examples that show-case low-level systems features including a power control system, bit fields, and the formal verification of a timer device driver.

REFERENCES

[1] 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

[2] 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.

[3] 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

[4] 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.

[5] Zilin Chen, Liam O'Connor-Davis, 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

[6] 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

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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

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

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

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

[17] 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

[18] 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.

[19] 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/

[20] 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.

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

[22] 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

[23] 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

[24] 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

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

[26] 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

[27] 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