# Dargent

## A Silver Bullet for Data Layout Refinement

Anonymous Author(s)

## Abstract

The Cogent language is designed for implementing high-assurance low-level system components as pure mathematical functions operating on algebraic data types. Cogent's certifying compiler generates imperative C programs and provides a formal proof that its compilation is a refinement of the functional Cogent code. Systems programs often adhere to interfaces that are imposed by standardised protocols or by the underlying hardware. While Cogent systems are relatively easy to verify and while they can interoperate with surrounding C infrastructure, there remains a gap between the interfaces that systems programmers desire and the algebraic data types used by Cogent. As such, Cogent programmers are required to implement tedious marshalling code to convert between the two. This code is error-prone, tedious to verify, and can lead to a performance overhead due to excessive copying. To bridge this gap, we present a data layout description language and data refinement framework, called Dargent. Dargent provides Cogent programmers with a means to declaratively specify how algebraic data types are laid out in memory. Using the Dargent specification, the compiler generates C code that matches the desired interface, automatically generates code to directly manipulate the C data structures directly, and provides a formal proof that its generated code is correct. This added expressivity removes the need for implementing and verifying marshalling code, which in turn, decreases copying, smoothens interoperability with C, and increases trustworthiness of the overall system.

## 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, 19]. 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 *verifying* those layout descriptions. We build on the Cogent language and refinement framework [16] (Section 2) for co-generating a C program and Isabelle/HOL [14] theorems witnessing a proof that the C program refines an Isabelle/HOL embedding of the Cogent source program.

Cogent is readily amenable to an extension for describing and manipulating data layouts 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 *deserialisation* code within file system implementations [2] and directly representing device register formats (Section 5). 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 proof (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 efficient destructive updates.

### Contributions

This paper realizes the vision set out in 2018 [3]. 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, and discuss the compilation process;

*PLDI'22, Mon 20 – Fri 24 June 2022, San Diego, California, US*
2022.

- 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.

## 2 Overview of Cogent

Cogent [16] 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 [5, 20] 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 [17] 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. [2]. 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 [20]. 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.

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. *Type synonyms* to other types can be defined but such types have a *structural* interpretation, meaning the synonym is identical to the type it is aliasing. In general, two types in Cogent are identical if they have the same structure, after fully expanding any type synonyms.

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.

Cogent maps Cogent types to C types in a deterministic manner; this mapping is hard-coded in the compiler and users of the Cogent language have to learn about the code generation mechanism, in order to write Cogent programs that need to interface with C code. For example, a record type 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.

### 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 [10] to generate a representation of the C code in Isabelle/HOL. More precisely, the AutoCorres library abstracts the C semantics via a SIMPL [18] 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 programmer first needs to manually prove the desired properties about the Isabelle/HOL shallow embedding. This

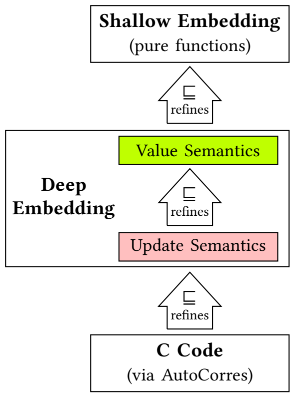

**Figure 1.** The Cogent refinement framework.

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 and 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 bytes. 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 below:

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

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 are the tag values for the variant constructors A and B. We will give a layout to this type after a short introduction to the Dargent language constructs.

In Figure 2 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.

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

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

**Figure 2.** Dargent syntax

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 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:

**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}

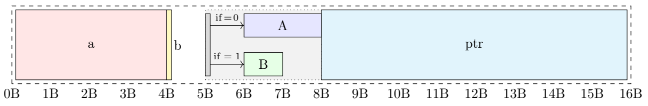

**Figure 3.** The ExampleLayout, visualised.

so that a later field are 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* data, which encodes the constructor in the variant which is being used. Then, for each constructor in the variant, a specific tag value is given as well as a layout expression for any additional data (i.e. the *payload*) provided in the variant type for that constructor.

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 (assuming a 64-bit architecture):

**layout** ExampleLayout = record {
  struct : record {a : 4B, b : 1b},
  ptr : pointer at 8B,
  sum : variant (1b at 5B)
    {A(0) : 2B at 6B, B(1) : 1B at 6B} at 0B
}

A pictorial illustration of the memory layout is given in Figure 3. 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. To reduce the 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

$$
\begin{array}{llcll}
\text{Bit ranges} & r & & & \\
\text{Offsets} & o & \in & \mathbb{N} & \\
\text{Layouts} & \ell & ::= & () & \text{(unit layout)} \\
 & & | & r_\omega & \text{(primitive layout)} \\
 & & | & x\{o\} & \text{(layout variable)} \\
 & & | & \mathsf{record}\ \{\overline{\mathrm{f}_i : \ell_i}\} & \\
 & & | & \mathsf{variant}\ (\ell)\ \{\overline{\mathrm{A}_i\ (n_i) : \ell_i}\} & \\
\text{Field names} & & \ni & \mathrm{f} & \\
\text{Constructors} & & \ni & \mathrm{A} & \\
\text{Endianness} & \omega & ::= & \mathsf{BE} \mid \mathsf{LE} \mid \mathsf{ME} & \\
\text{Layout contexts} & C & ::= & \overline{\ell_i \sim \tau_i} &
\end{array}
$$

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

**Figure 4.** Syntax for Dargent core language

variables are allowed. For now, there is only one type of constraint on layout variables: that a layout variable matches a type. [1] For example, in the code snippet below,

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

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, $\mathit{freePair}[\mathsf{U32}]\{\!\{\mathsf{FourBytes}\}\!\}$ instantiates $\tau$ to U32 and $\ell$ to FourBytes. If the type/layout application ends up with incompatibility, for instance $\mathit{freePair}[\mathsf{U8}]\{\!\{\mathsf{1b}\}\!\}$ 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 $\ell$ also needs to ensure that all the layouts in the function are well-formed. For example, if $\ell$ is instantiated with 8B, it will render LPair $\ell$ ill-formed, as the snd field will overlap with the fst. This can be rectified by using the after relative location (or leave it implicit) instead, which will automatically lay the snd field right after fst.

## 3.2 The Dargent core calculus

The Dargent surface language is desugared into a smaller core calculus, whose syntax is outlined in Figure 4. 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.

[1] A formal definition of "match" will be discussed later.

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 $\ell_\omega$.

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 $()$.

For layout variables, we need to remember the offset on variables, hence the bit range is of the form $x\{o\}$. 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:

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

We write $\mathsf{b}_\ell$ 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 b.

## 3.3 The static semantics with Dargent

Cogent's static semantics is defined through a standard typing judgement $\Delta; \Gamma \vdash e : \tau$ with an additional context $\Delta$ that tracks the linearity of type variables in $\tau$. 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 5) and that the layouts of components of a type do not overlap (Figure 6). 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 $\Delta; C \vdash \tau$ **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 $\ell$, where $\hat{\tau}$ is essentially the type $\tau$ 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.

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

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, $\ell$ **wf**, ensures that field layouts do not overlap (Figure 6). The last two premises require the type to match the layout (Figure 5) 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 [15, 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 $\Delta$.*

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

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

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

*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 $(\ell[\overline{l_j/x_j}], [\overline{\hat{\rho}_i/\alpha_i}]) \in C'$ hold.*

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

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

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

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

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

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

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

**Figure 5.** Matching relation between layouts and types

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

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

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

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

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

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

**Figure 6.** Wellformed layouts

In this lemma, $-[\overline{\rho_i/\alpha_i}, \overline{l_j/x_j}]$ denotes the simultaneous substitution replacing both type variables $\alpha_i$ and layout variables $x_j$, and $\Gamma$ 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 $\Delta', C'$. 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 $\Delta$), 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 $(x\{o\}, \text{U8})$ and $(x\{o\}, \text{U16})$, although in that case the layout variable $x$ can never be fully instantiated, because of the wellformed rule for substitution mentioned above.

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

Without custom layouts, a Cogent record is simply 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.

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 any available Cogent operation on boxed records. Assigning a custom layout $\ell$ 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 getters and setters are then generated according to the layout. For a field a : $A$ in $T$, 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 or the type of the second argument of the setter, does not involve the layout. The purpose of the getter, indeed, is to reconstruct a standard representation out of the custom representation induced by the layout.

***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:

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 details in §4.3. It is worth mentioning that extending the verification framework to take Dargent into account does not require to prove that the generated getters and setters are putting and getting 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 representation, however, do not necessarily help code generation. Therefore we have another step to convert each bit range into a list of *aligned bit ranges* tailored for C code generation.

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 essentially 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 operators. Each such C statement is generated according to one aligned bit range. This one-to-one mapping simplifies C code generation, and matches the way machine instructions work, which is important for high performance.

## 4 Verification framework with Dargent

Dargent affects the formalisation of the type system, and beyond C code generation, it also affects the generated correspondence proof with the pure shallow embedding in Isabelle, but only at the interface between Cogent's stateful update semantics and the generated C code. Indeed, above this low-level layer, the functional embeddings of Cogent still use algebraic types, regardless of the specified layouts. We discuss in the following subsections the changes to the verification framework regarding:

1. the correspondence between Cogent record values to C flat arrays;
2. the correspondence between record operations in Cogent and C;
3. the compositional properties of generated getters and setters.

In a last subsection, we describe some sanity checks about getters and setters.

Finally, let us mention that our extension to the verification framework does not yet tackle the endianness annotations.

### 4.1 Relating record values

The correspondence [15, 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 should relate to the result of applying the relevant custom getter on the array.

This requires an appropriate embedding of custom getters in Isabelle. To this end, we use a ML procedure that takes as input the *monadic* embeddings provided by the AutoCorres library, on which Cogent is based, and simplifies them to get pure Isabelle functions. We call them *direct getters* in the following. We similarly infer *direct setters* that allow us to state nicely the compositional properties detailed in §4.3.

## 4.2 Relating record operations

Without Dargent, records compile directly to C structures. Therefore, it is straightforward to relate Cogent record operations to their compiled versions and show that they correspond, in the sense that they produce related outputs when given related inputs. With layouts, the correspondence becomes more involved, since it relates getting or setting fields in Cogent and calling custom getters or setters in C. These adapted statements are proved automatically by tactics that exploit the good 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 compose well, as schematically summarised below:

1. if a Cogent value $x$ relates to a C value $v$, then it also relates to `get_a (set_a arr` $v$`)`
2. `get_a (set_b arr v) = 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. Consider indeed a boolean field. Since C does not natively provides such a 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. Nonetheless, we also designed automatic tactics to prove that field getters depend only on the bits taken by the field layout, or that field setters do not flip any bit that is outside the field layout. In future work, we plan to write a generic specification of these getters and setters and prove them correct automatically.

# 5 Applications

In this section, we showcase how Dargent helps in developing and verifying systems code via some small examples.

## 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[2], an open-source low-level hardware library for ARM Cortex-M3 microcontrollers. The original C file[3] 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 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 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.

## 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[4] 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;
```

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

```
  };
```

Although this is not supported by the AutoCorres library 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 [15, §3.3.2].

Although some manual verification effort is required 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,
    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. [2] 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 more user-friendly.

## 6 Verification of a timer device driver

In this section, we present a formally verified Cogent timer driver, that we ran successfully on an odroid hardware. Our formalisation took advantage of the fact that the shallow embedding 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 [7] that consists of an interface for the so-called timers A and E provided by the device. 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 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 7.

## 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. The manual proof can be done straightforwardly using equational reasoning, and is much easier than reasoning about the bit-wise 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 }
```

**Figure 7.** 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 Result of verification

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 Future extensions: volatile variables

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. This volatile behaviour can be modelled by introducing a layer of non-determinism in the shallow embedding, when necessary. The Cogent file system BilbyFs has an example of how non-determinism can be modelled on top of the shallow embedding [1]. One can explore dealing with volatile registers in a similar manner.

## 7 Related work

Data description languages are a rich area of research which 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 [8, 9, 11], PacketTypes [12], Protege [21], and DataScript [4] are concerned with synthesising a parser program from a high-level specification of the data format. On the other hand, LoCal [19], 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.

Of the existing literature we surveyed, LoCal is perhaps most comparable to Dargent. Vollmer et al. [19] 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.'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, *cf.* Murray [13]. 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 writing the *glue* code 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. 2016. *A Methodology for Trustworthy File Systems.* Ph.D. Dissertation. CSE, UNSW, Sydney, Australia.

[2] 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

[3] 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.

[4] 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

[5] 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.

[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] CSIRO Data61. 2021. A C timer driver for odroid. https://github.com/seL4/util_libs/blob/c446df1f1a3e6aa1418a64a8f4db1ec615eae3c4/libplatsupport/src/plat/odroidc2/meson_timer.c.

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] Emmet Murray. 2019. Recursive Types for Cogent. https://github.com/emmet-m/thesis. Accessed November 2021.

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

[15] 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.

[16] 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

[17] 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.

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

[19] 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

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

[21] 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