Bringing Effortless Refinement of Data Layouts to COGENT

Liam O'Connor[1], Zilin Chen[1 2], Partha Susarla[2], Christine Rizkallah[1], Gerwin Klein[1 2], Gabriele Keller[1]

[1] UNSW Australia
Sydney, Australia
liamoc@cse.unsw.edu.au

[2] Data61, CSIRO (formerly NICTA)
Sydney, Australia
first.last@data61.csiro.au

Abstract. The language COGENT allows low-level operating system components to be modelled as pure mathematical functions operating on algebraic data types, which makes it highly suitable for verification in an interactive theorem prover. Furthermore, the COGENT compiler translates these models into imperative C programs, and provides a proof that this compilation is a refinement of the functional model. There remains a gap, however, between the C data structures used in the operating system, and the algebraic data types used by COGENT. This forces the programmer to write a large amount of boilerplate marshalling code to connect the two, which can lead to a significant runtime performance overhead due to excessive copying.

In this paper, we outline our design for a data description language and data refinement framework, called DARGENT, which provides the programmer with a means to specify how COGENT represents its algebraic data types. From this specification, the compiler can then generate the C code which manipulates the C data structures directly. Once fully realised, this extension will enable more code to be automatically verified by COGENT, smoother interoperability with C, and substantially improved performance of the generated code.

1 Introduction

In the context of end-to-end functional correctness verification of operating system components, the integration of modelling and programming presents a significant challenge.

Models are typically designed to enable concise specification and to reduce verification effort. In verifications using interactive proof assistants, such as that of the seL4 operating system[1], a *purely functional* model is ideal, as programs are modelled in terms of mathematical functions: objects for which proof assistants have significant built-in support and automation. On the other hand, programs are designed to be efficiently executable. Operating systems, and their kernels in particular, are usually written in relatively low-level languages such as C in order to achieve ideal performance and predictable run-time behaviour. Existing programming languages that support directly

[1] The seL4 microkernel: https://sel4.systems (accessed on August 31, 2018).

programming in a purely functional style such as HASKELL are not well-suited to systems programming, because of their reliance on extensive run-time support for memory management and evaluation.

The COGENT programming language [13] allows purely functional models to be compiled into efficient C code suitable for systems programming. It achieves this through the use of a sophisticated type system, which allows allocations to be replaced with efficient destructive update, and eliminates the need for a garbage collector. Furthermore, this compilation process is proven correct by translation validation — in addition to C code, the compiler generates a formal proof that any correctness theorem proven about the purely functional model also applies to the generated C code. Section 2 describes COGENT, its type system, and the associated verification framework in more detail.

COGENT programs do not exist in isolation, however. Typically, a COGENT program constitutes a component of a larger operating system, written in C. When integrating a COGENT component into this larger context, we see that there remains a gap between COGENT functional programming and typical systems programming used for the rest of the system.

In COGENT, programs are defined as pure functions operating on *algebraic data types*, such as product types (e.g. records, tuples) and sum types (also known as variants or tagged unions). The exact structure of these data types in memory is determined by the compiler. Many of the data structures in operating systems such as Linux could be represented as algebraic types, however their exact memory layout differs from that used by the COGENT compiler. Therefore, the file systems implemented in COGENT as a case study [1] must maintain a great deal of marshalling code to synchronise between the two copies of the same conceptual data structure. As COGENT code can only interact with the COGENT representation of the data, this synchronization code must be written in C. This code is tedious to write, wasteful of memory, prone to bugs, has a significant performance cost, and requires cumbersome manual verification at a low level of abstraction.

In this paper, we propose a new framework for data abstraction in COGENT programs. Rather than maintain two copies of data, we define a domain specific data description language, DARGENT, to describe the correspondence between COGENT algebraic data types and the bits and bytes of kernel data structures – what we call the *layout* of the data. As with COGENT itself, this framework reduces the gap between modelling and programming, in the sense that the programmer can write code as normal, manipulating ordinary COGENT data types, and after compilation the generated C code will manipulate kernel data structures directly, without constant copying and synchronisation at run-time. This will improve performance by eliminating redundant work, dramatically simplify the process of integrating C and COGENT code, and make it possible to verify more code with COGENT rather than using cumbersome C verification frameworks. Our vision for the DARGENT data description language is outlined in Section 3.

A number of extensions must also be made to COGENT itself to accommodate this new data refinement framework, outlined in Section 4:

1. The *type system* needs to be extended to incorporate these data layouts.

2. The *code generator* needs to compile each abstract read and write operation on COGENT data types to the equivalent concrete operation in C, according to DARGENT layouts.
3. The *verification framework* must be updated to once again automatically verify that the compiler output is a correct compilation of the compiler input.

At the time of writing, we have implemented the prototype data description language in the COGENT compiler, however the extensions to COGENT itself to take DARGENT layouts into account are still in development.

```
1  type Heap
2  type Bag = Ptr {count : U32, sum : U32}
3  newBag : Heap → ⟨Success (Heap, Bag) | Failure Heap⟩
4  freeBag : (Heap, Bag) → Heap
5  addToBag : (U32, Bag) → Bag
6  addToBag (x, b {count = c, sum = s}) =
7      b {count = c + 1, sum = s + x}
8  averageBag : Bag! → ⟨Success U32 | EmptyBag⟩
9  averageBag (b {count = c, sum = s}) =
10     if c == 0 then EmptyBag else Success (s / c)
11 type List a
12 reduce : ((List a) !, (a!, b) → b, b) → b
13 average : (Heap, (List U32)!) → (Heap, U32)
14 average (h, ls) = newBag h
15 | Success (h', b) →
16 |     let b' = reduce (ls, addToBag, b)
17 |     in averageBag b' !b'
18 |        | Success n → (freeBag (h', b'), n)
19 |        | EmptyBag → (freeBag (h', b'), 0)
20 | Failure h' → (h', 0)
```

Fig. 1. An example COGENT program, using heap-allocated data to compute the average of a list.

2 Overview of COGENT

COGENT is a purely functional programming language, in the tradition of languages such as HASKELL or ML. Programs are typically written in the form of mathematical functions operating on algebraic data types. Unlike HASKELL or ML, however, COGENT is designed for low-level operating system components, and therefore it does not require a garbage collector, and memory management is entirely explicit. Despite this, COGENT is able to guarantee memory safety through the use of a strict *uniqueness type* system. This static type discipline ensures that variables that contain references to heap

objects or other singular resources such as files and disks cannot be duplicated or disposed of implicitly. It achieves this with a cheap syntactic criteria — any variable of such a type must be used *exactly* once. This means that it is impossible to leak memory, as this would mean such a variable would be left unused; nor is it possible to access memory after it has been freed, as this would require accessing the same variable twice. A similar type discipline is used to ensure memory safety in the programming language RUST [2].

Figure 1 gives a full example of a COGENT program, computing the average of a list, storing the running total and count in a heap-allocated data structure called a Bag. Line 2 defines the Bag as a `Ptr` to a heap-allocated record (a *product type*) containing two 32-bit unsigned integers. Lines 3 and 4 introduce allocation and free functions for Bags. Definitions of types and of functions may be omitted from the COGENT source and provided externally via the C foreign function interface. Currently, heap memory management functions and loop iterators must be provided using this mechanism, although relaxing these requirements is part of ongoing work. The newBag function returns a *variant* (or *sum type*), indicating that *either* a bag and a new heap will be returned in the case of $\mathrm{Success}$, or, in the case of allocation $\mathrm{Failure}$, no new bag will be returned. The addToBag function (lines 5-7) demonstrates the use of pattern-matching to destructure the heap-allocated record to gain access to its fields, and update it with new values for each. The averageBag function (lines 8-10) returns if possible the average of the numbers added to the Bag. The input type $Bag!$ indicates that the input is a *read-only*, freely shareable view of a Bag, called an *observer* in COGENT or a *borrow* in RUST. An observer can be made of any variable using the ! notation, as seen on line 17 where averageBag is called. Lines 11 and 12 introduce a polymorphic abstract $List$ type, as well as a reduce function, which aggregates all data from the list using the provided function and identity element. Lastly, lines 13-20 define the overall average function, which creates a Bag with newBag, pattern matches on the result, and, if allocation was successful, adds every number in the given list to it, and then returns their average.

2.1 Uniqueness Types

Uniqueness types allow us to model imperative, stateful computations as pure mathematical functions, as the static type discipline ensures that each mutable heap object has exactly one usable reference at any point in time. This means that a well-typed program can be given two interpretations: an imperative *update* semantics that mutates heap objects and a "pure" *value* semantics with no notion of a heap, that treats all objects as immutable values. As it is impossible to alias mutating objects, the equational reasoning by which we would reason about the pure interpretation applies just as well to the imperative interpretation, as the lack of aliasing makes it impossible to observe the mutation of an object from any reference other than the one used to mutate it.

Such equational reasoning is highly suitable for verification in a proof assistant such as Isabelle/HOL, as the pure denotation of a COGENT function is simply a mathematical function – objects for which most proof assistants have significant built-in support.

[2] The Rust programming language: https://www.rust-lang.org (accessed on August 31, 2018).

2.2 The COGENT Verification Framework

In addition to a programming language, COGENT is also a verification framework realised in Isabelle/HOL, based on *certifying compilation*. This means that apart from compiling a COGENT program to C, the compiler also generates an Isabelle/HOL shallow embedding of the program in terms of simple functions, and a proof that the generated C code is a *refinement* of that embedding. This entails that any functional correctness theorem proven about the simple shallow embedding also applies to the generated C code.

In prior work [1], we developed two Linux file systems in COGENT, and proved key correctness theorems about one of them. The equational semantics drastically reduced the effort required to verify these systems, and their performance was comparable to other Linux file systems that were hand-written in C.

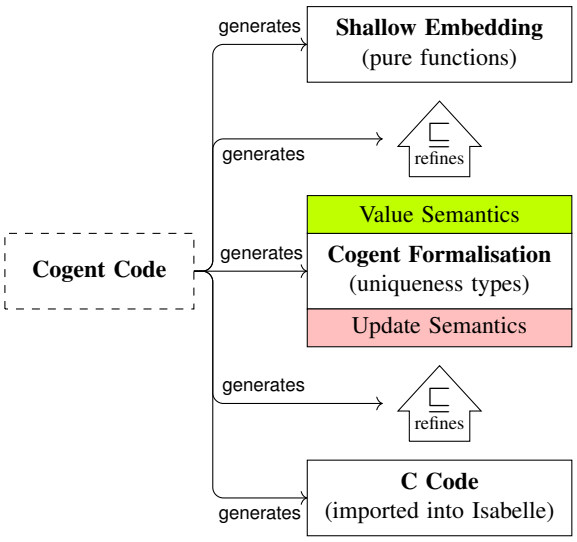


Fig. 2. The COGENT refinement framework. Boxes represent Isabelle/HOL embeddings of the COGENT program, arrows represent refinement proofs.

Figure 2 gives an overview of the COGENT framework. The overall proof that the C code refines the purely functional shallow embedding in Isabelle is broken into a number of sub-proofs and translation validation phases. Three embeddings are generated: a top-level shallow embedding in terms of pure functions; a deeply-embedded representation of the abstract syntax of the COGENT program, which can be interpreted using either the *value* or *update* semantics; and an Isabelle/HOL representation of the C code generated by the compiler, imported into Isabelle/HOL by the same C parser used in the seL4 project [11]. As can be seen in the diagram, the compiler also generates a proof that the pure interpretation of the deep embedding is a refinement of the top-level

shallow embedding, and that the C code is a refinement of the imperative interpretation of the deep embedding. These refinement proofs, along with the refinement theorem between the two semantic interpretations proven by [13], can be composed to produce a refinement proof stretching from the C code all the way up to the pure shallow embedding.

2.3 The C Model

To give a formal meaning to our C code in Isabelle/HOL, we make use of the same C-Parser framework [18] used by the seL4 project [10]. This framework imports a large subset of C99 code into HOL by translating it into the embedded language SIMPL [16], which has a full formal definition of its operational and axiomatic semantics for use in proofs. This semantics could be viewed denotationally as a *relation on states*, describing what final states could result from a given initial state. The exact structure used to represent a state is a *parameter* to the definition of SIMPL, and therefore the C-Parser is free to choose a structure that mirrors the C code closely. Typically, the state structure is an Isabelle/HOL *record* with fields for each of the stack-allocated local variables used in the code, along with a field for the heap, represented using the memory model of Tuch [17].

A consequence of this representation is that, while the heap memory model used allows for pointer arithmetic, unions, and type-casting of *heap* memory, the view of the *stack* is significantly abstracted. With this state definition, it is not well-defined to take a pointer to a stack-allocated variable, or to reinterpret stack memory as a different type. C code that performs such operations is rejected by the C-Parser.

2.4 Data Refinement

A program C *refines* a program A if every observation of C is also a possible observation of A. If $[\![\cdot]\!]$ maps programs to sets of observations, then the refinement of A by C is expressed by $[\![C]\!] \subseteq [\![A]\!]$.

Seeing as observations in our case take the form of relations on states, such a refinement would entail that any partial correctness property proven for all executions in $[\![A]\!]$ will also apply to those executions in $[\![C]\!]$.

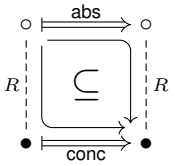
In our refinement proofs between the update semantics and C, however, the state spaces of the two programs are different, and thus the relations are not immediately comparable.

The COGENT memory model in the update semantics includes a mutable heap, however this heap contains values of *algebraic data types*, such as records (product types) and variants (sum types). The C memory model, on the other hand, is ultimately defined in terms of bits and bytes, not rich data types, to allow for pointer arithmetic and type-casting operations in C.

As our state spaces differ in this way, a simple subset relationship does not quite capture what we require of refinement; we need a notion of correspondence of states. We get this from some additional machinery from the world of *data refinement* [15]. We introduce a *refinement relation* R that relates abstract and concrete states, and show

that each step our program takes preserves this relation; the relation represents the desired correspondence. We must show that our abstract program behaves analogously to the concrete program given corresponding initial states. That is, if R relates our initial abstract and concrete states, then every final state of a concrete program conc will be related by R to a final state of a corresponding abstract program abs:

$$R; [\![\mathsf{conc}]\!] \subseteq [\![\mathsf{abs}]\!]; R$$
(where ; is composition of relations)



In our framework, the compiler generates the definition of this refinement relation by describing how the code generator lays out these rich data types in memory. Rizkallah et al. [14] describe in more detail the techniques we have developed to automatically prove these refinement theorems for COGENT programs.

These relation preservation proofs only imply refinement given the assumption that the relation R holds initially. A similar assumption is made for the verification of seL4, and proving this is the subject of ongoing research.

2.5 Memory Layouts in COGENT and Linux

Because of the restrictions on stack memory outlined in Section 2.3, the COGENT compiler chooses very straightforward memory layouts to represent algebraic data types. For record (product) types, each field is laid out in memory as a C `struct`. For variant (sum) types, a special value called the *tag*, which indicates the constructor used for the variant, is stored in a `struct` along with several sub-structures for the constructor parameters, only one of which will contain meaningful data.

The Linux kernel, on the other hand, typically chooses much more exotic data representations, using techniques such as:

Bitfields Several boolean flags are very often represented using the individual bits of a machine word.

Type tags The value of one part of a data structure can determine how to interpret/typecast another part of a data structure, for example with tagged unions or tagged void pointers.

Container pattern Kernel-defined structures are often nested within component-specific structures at offset zero. This means that a component-specific object can be safely cast to the more general kernel-defined type for use within the kernel, and then cast back to the component-specific type when returned to the component by the kernel.

Padding and Alignment Often, blank space is left in the object intentionally to account for architecture-specific alignment considerations.

Dynamically sized objects Kernel data types often contain values that determine the size of other objects. For example, an array is often paired with its length as an integer.

Seeing as none of these techniques can currently be used by the COGENT compiler when laying out types in memory, Linux types such as these are typically modelled as abstract "black box" types in COGENT. Conversion functions from Linux data structures and COGENT data types must be manually written in C and painstakingly verified in Isabelle. These functions are very tedious to write, frustratingly error prone, and have a negative impact on performance.

The focus of our present work is therefore to extend the compiler and verification framework of COGENT to better support these kinds of memory layouts, transparently representing them as algebraic data types.

3 DARGENT, the Data Description Language

We are in the process of designing a data description language, called DARGENT, that describes how a COGENT algebraic data type may be laid out in memory, down to the bit level. Data descriptions in this language will influence the definition of the refinement relation to C code generated by the compiler. Eventually, we hope to make this language flexible enough to accommodate any conventional representation of an algebraic data type in memory. Section 4 describes in more detail the extensions necessary to COGENT to accommodate DARGENT descriptions.

sizes	s	$::= n\mathrm{B} \mid n\mathrm{b} \mid n\mathrm{B} + m\mathrm{b}$	
layout expressions	ℓ	$::= s$	(block of memory)
		$\mid L$	(another layout)
		$\mid \ell$ at s	(offset operator)
		$\mid$ record $\{\overline{\mathrm{f}_i : \ell_i}\}$	(records)
		$\mid$ variant (ℓ) $\{\overline{\mathrm{A}_i\ (n_i) : \ell_i}\}$	(variants)
declarations	d	$::=$ **layout** $L = \ell$	
layout names	L		
record field names	f		
variant constructors	A		
numbers	n, m	$\in \mathbb{N}$	

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

Fig. 3. The grammar of our DARGENT prototype.

Figure 3 gives a grammar for the syntax of DARGENT descriptions, which are made of one or more **layout** declarations, which give names to particular *layout expressions*. Each such expression describes how to lay out a data type in memory. Primitive types such as integer types, pointers, and booleans are laid out as a contiguous block of memory of a particular size. For example, a layout consisting of a single 4 bytes would only be appropriate to describe COGENT types that occupy four contiguous bytes of memory, such as `U32` or pointer types.

$$\textbf{layout}\ \mathit{FourBytes} = 4\mathrm{B}$$

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

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

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 reserve overlapping blocks, the description is rejected by the compiler.

Layouts for variant types use the variant construct, which 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 provided in the variant type for that constructor.

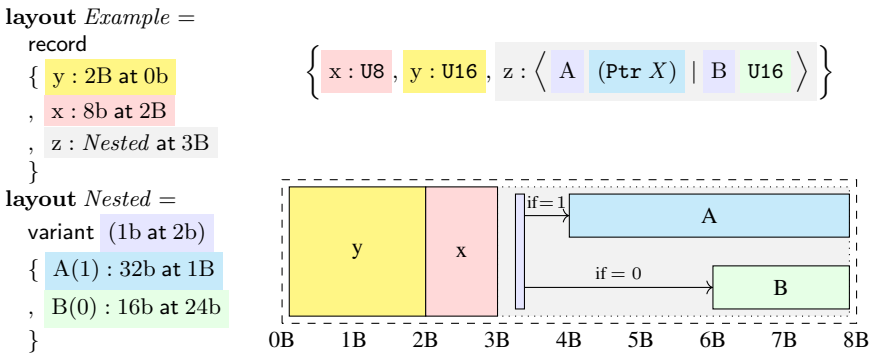


Fig. 4. A COGENT type (upper-right) laid out (lower-right) according to a DARGENT spec (left).

Figure 4 gives an illustrative example of a DARGENT description in our current prototype. We describe a memory layout for a COGENT record type containing two numbers and a variant, $\{\mathrm{x} : \mathtt{U8}, \mathrm{y} : \mathtt{U16}, \mathrm{z} : \langle \mathrm{A}\ (\mathtt{Ptr}\ X) \mid \mathrm{B}\ \mathtt{U16}\rangle\}$. As can be seen from the ordering of the fields y and x, fields may be placed in any order and at any location. This allows us to accommodate data layouts where certain parts of the data type must appear at particular offsets, such as with the container pattern mentioned in Section 2.5. It also makes it possible to leave unreserved space in between fields, accommodating data layouts which do this to respect padding or alignment constraints in the architecture.

The variant field z is represented according to the *Nested* description, offset by three bytes. That *Nested* description reserves the third bit of the first byte (the fourth byte of the original object) to determine which of the two constructors A and B is active. If the bit is 1, the constructor A is active, with the additional $\mathtt{Ptr}\ X$ payload stored at a one-byte offset (the fifth to eighth bytes of the original object). If the bit is 0, the constructor

B is active, with the U16 payload stored at a three byte offset (the seventh and eighth byte of the original object).

4 Extensions to COGENT Type system

A COGENT type τ represented on the heap using a particular DARGENT layout ℓ will result in different C code than the same type τ represented using a different layout ℓ'. While they are identical on the abstract level, on the concrete level they are not interchangeable. Therefore, we must extend the COGENT type system such that identical types represented differently are distinguished, by *tagging* types with their representation.

Seeing as our DARGENT descriptions only apply to objects allocated on the heap, and COGENT pointer types (`Ptr` τ) only ever point to objects on the heap (both due to the restrictions described in Section 2.3), the natural place to add these tags is to the pointer type itself. We will add an additional representation parameter ℓ to the pointer type, written `Ptr` τ ℓ. This type indicates that the heap object of type τ pointed at by the pointer is laid out in memory according to the description ℓ. The type checker is then responsible for ensuring two properties:

1. That the layout ℓ is well-formed, i.e. that it does not reserve overlapping blocks of memory, and that it does not reference any unknown **layout** declarations.
2. That the type τ can be represented according to the description ℓ. For example, a 32-bit word U32 or a pointer on a 32-bit architecture could both be represented by the description $FourBytes$ from Section 3, but a U64 value could not.

These descriptions are then used to generate the correspondence relation for each pointer type used in the program, which forms the basis of the refinement proof from COGENT to C.

4.1 Marshalling and Unmarshalling

COGENT programs often copy (parts of) heap allocated objects to the stack. For example, when pattern matching on particular fields of a heap allocated record, local variables are introduced for each field being matched, thus copying each matched field to the stack.

Without DARGENT, this copying was straightforward, as the memory representations on the heap and on the stack were the same. With DARGENT, however, the layouts may differ, and thus the compiler must also generate code to convert between the stack and heap representations of each heap-allocated type. We expect that this conversion code will be straightforward to generate as all the necessary information is provided to us by DARGENT data descriptions.

4.2 Extended Pattern Matching

Currently, to match on a heap-allocated variant type, COGENT programs first copy the variant to the stack, and then match on the stack-allocated variant. This is not ideal, as

large amounts of copying may be required if the variant contains a lot of data. To remedy this, we plan to make use of DARGENT descriptions to perform pattern matching on heap-allocated variant types in-place.

Additionally, we plan to extend the currently limited pattern matching features available in COGENT to allow for *nested* patterns, so that a variant can be matched in-place inside a heap-allocated record, for example.

By reducing the amount of copying to the stack that needs to take place, we expect well-designed COGENT programs augmented with DARGENT descriptions to require very little marshalling or unmarshalling at all, instead transparently manipulating the existing operating system data structures while viewing them as algebraic data types.

5 Future Work

The prototype data description language and framework envisioned here only scratches the surface of the potential use cases of DARGENT. In addition to several syntactic improvements, such as allowing offsets and sizes of objects to be computed relatively to the offsets and sizes of other objects, we plan to extend our initial prototype of DARGENT to support a number of additional semantic features.

5.1 Tighter C Integration

In our prototype, to access a data structure defined in C, one must define a highly platform-specific DARGENT layout that matches the alignment, padding, integer size, and pointer size of the architecture and C compiler being used.

Ideally, we would like to be able to automate this process, replicating the exact layout decisions made by the C compiler so that C definitions can automatically be converted into DARGENT for each compiler and architecture being used.

5.2 Constraints and Validation

Currently, data in memory is assumed to be *valid*, that is, to conform to the schema laid out by a DARGENT layout. Thus, to verify a COGENT program incorporating some external C code, it is neccessary to prove that the C code does not violate the invariants of the data structure. Therefore, our framework is designed to only deal with *trusted* data, where deserialisation is a total function. Extending the framework to deal with potentially-invalid data must be done with great care, as our data refinement theorem requires that the refinement relation maps *every* concrete state in the execution of our program to a corresponding abstract state.

Even with the relatively simple data descriptions in our prototype, however, it is possible to have invalid data: for example, if the tag field for a variant type contains a value that does not correspond to any of the constructors in the type.

To address this, we intend to add a new totally abstract pointer type to COGENT, written `APtr`, which does not have any associated type or layout information. This type will be used to represent data on the heap that is potentially invalid. We will add a language construct to operate on values of type `APtr`, written $\mathbf{validate}\ e\ \ell\ \tau$, which

performs a dynamic check that the given `APtr` value e can be represented as the type τ using the representation ℓ, with the following typing rule:

$$\frac{\ell \text{ matches } \tau \qquad e : \texttt{APtr}}{\mathbf{validate}\ e\ \ell\ \tau : \langle \text{Valid } (\texttt{Ptr}\ \tau\ \ell) \mid \text{Invalid } \texttt{APtr} \rangle}$$

This rule indicates that the **validate** operation returns a fully typed `Ptr` only if validation is successful. Because of the uniqueness type system, this conversion makes the previous `APtr` value inaccessible, making it impossible to interpret the same data structure in multiple ways. Thus, once validated, data will remain valid, even if mutated by COGENT programs.

This validation mechanism could be extended in a number of ways to support domain-specific error handling and validation constraints. For example, kernel data structures for file systems often include a "magic number" field that is hard-coded to be a specific unusual value, to detect buffer overruns early, and to debugging memory dumps easier. Some data structures, such as those found in file formats, also may include error detection mechanisms such as checksums.

We intend to allow the user to define these domain-specific constraints and attach them to a DARGENT descriptions using a subset of COGENT. Other embedded data description languages, such as PADS [6, 8], take a similar approach where the host programing language of the data description language is used to attach domain-specific constraints about the data descriptions.

5.3 Wire Formats

While COGENT was originally conceived to ease the verification of file systems [9], we have long considered network stacks as another systems area which could benefit from the COGENT approach to verification.

To better support network stacks, we intend to extend DARGENT to support *wire formats*, which describe the layout of data as it is transmitted over a network link. Typically, network protocols specify bit and byte-level endianness for all values, which may differ from the architecture-defined endianness used for in-memory data structures. We plan to incorporate endianness annotations to DARGENT descriptions to accommodate this.

Typically, network protocols also incorporate various error checking and correction mechanisms for data. There is a wealth of existing work on data description languages for network protocols (see Section 6), and we plan to draw on this work to inform our design for error-checking in DARGENT, potentially integrating it into our constraints and validation framework.

5.4 Dynamically Sized Data

Right now, our prototype only supports data structures of statically-known size. Dynamically sized arrays, for example, must be left as abstract types, manipulated by externally-provided functions.

Typically, a dynamically-sized data structure contains *header* information from which the size of the *payload* data structure can be computed. This is a natural fit for dependent types, which is why the core calculus of PADS, the Data Description Calculus, is based on a dependent type theory [7].

In a concurrent project, we are working on bringing simple indexed or dependent types to COGENT [4]. This would allow dynamically sized arrays to be given a safe interface where only indices that are provably in-range can be used. By making DARGENT descriptions dependent, just as the types are, we could support dynamic structures that carry their size as a separate field very naturally, in a manner similar to PADS.

5.5 Layout Polymorphism

By taking a COGENT program and adding DARGENT layout descriptions, we mix the abstract functional model with concrete implementation details. Thus we cannot simply run the same program with different heap layouts without changing either the program or the layout.

A COGENT program that does not make use of kernel APIs or foreign C functions can be defined *independently* of the layout used. For this reason, we plan to extend COGENT to support *layout polymorphism*. Such a feature would allow COGENT functions to be defined generically for any layout, and instantiated to particular layouts by the compiler, based on their call-sites.

6 Related Work

The are numerous data description languages which generate access, serialisation and de-serialisation code, as well as code to transparently access the data without translating between representations. However, to the best of our knowledge, apart from the seL4 bitfield generator [5], there are no systems which also provide a machine-checked correctness proof for the generated code. This bitfield generator is subsumed by our work, as bitfields are just one of the capabilities of DARGENT.

The aforementioned PADS family of languages [6], particularly PADS/Haskell [8], serve as inspiration for parts of DARGENT. These languages, unlike DARGENT, are aimed at non-binary formats such as ASCII encodings, and focus entirely on data marshalling: parsing and pretty printing. By contrast, DARGENT prefers to rely on transparent data refinement, only translating data between representations when absolutely necessary.

DataScript [2], like DARGENT, is targeted at binary formats. Rather than algebraic data types, however, it is designed to generate object-oriented class definitions in Java for a given binary format, along with marshalling routines.

PacketTypes [12] and Protege [19] are both data description languages for describing wire formats for network protocols. They are focused on C implementations, and therefore do not provide the sort of data abstraction we see in DARGENT. Instead, they generate C code to serialise to and deserialise from wire formats.

Nail [3] is a very powerful tool for parsing and generating data formatted according to a grammar. Using Nail's description language, the tool generates an in-memory

structure for the data, along with serialisation and deserialisation code. Nail supports a wide range of features, including recursive grammars, dependent fields, checksums, and dynamically sized data. Nail greatly reduces the effort in handling the conversion between data formats and their internal representations, backed by a semantic bijection. However, to the best of our knowledge, Nail lacks any formal semantics or proofs of correctness.

7 Conclusion

There is a fundamental tension between models, which are artefacts designed for *reasoning*, and programs, which are artefacts designed for *execution*. High level programming languages promise to ease this tension by allowing programs to resemble models more closely, and automating the process of refinement from model to executable.

Many traditional techniques for high level programming language implementation, such as run-time support for data representation and memory management, become unsuitable in the context of end-to-end formal verification. In order to minimise the assumptions made to support verification, we must adopt fully static, fully formalised techniques to describe the compilation process and the proof of its refinement.

At the same time, when developing operating system components, we need efficient and predictable performance — we must avoid sacrificing performance for verifiability.

COGENT, to a certain extent, already manages this balancing act, as it allows programs to be written in a purely functional style, while compiling to efficient C code without relying on any run-time support. The limitations of COGENT only begin to show once larger systems are implemented in it, where the mismatch of data representations can lead to severely under-performing systems.

The DARGENT data description language we have envisioned in this paper is the last piece in this puzzle. By allowing the programmer to control the compilation and refinement process, specifically in terms of data layouts, we widen the applicability of the COGENT framework to new domains such as network stacks, improve interoperability with existing systems code, and take another significant step towards achieving a *verified* unification of modelling and programming.

Bibliography

[1] Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., Heiser, G.: Cogent: Verifying high-assurance file system implementations. In: International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 175–188. Atlanta, GA, USA (Apr 2016)

[2] Back, G.: DataScript - a specification and scripting language for binary data. In: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering. pp. 66–77. Springer-Verlag, London, UK, UK (2002), http://dl.acm.org/citation.cfm?id=645435.652647

[3] Bangert, J., Zeldovich, N.: Nail: A practical tool for parsing and generating data formats. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. pp. 615–628. USENIX Association, Broomfield, CO (2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert

[4] Chen, Z.: COGENT$^{\Uparrow}$: Giving systems engineers a stepping stone (extended abstract). In: The 2017 ACM SIGPLAN Workshop on Type-driven Development (ICFP TyDe 17) (2017), https://www.cse.unsw.edu.au/~zilinc/tyde17.pdf

[5] Cock, D.: Bitfields and tagged unions in C — verification through automatic generation. In: Proceedings of the 5th International Verification Workshop. pp. 44–55. Sydney (Aug 2008)

[6] Fisher, K., Gruber, R.: PADS: a domain-specific language for processing ad hoc data. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 295–304. ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1065010.1065046

[7] Fisher, K., Mandelbaum, Y., Walker, D.: The next 700 data description languages. Journal of the ACM **57**(2), 10:1–10:51 (Feb 2010), http://doi.acm.org/10.1145/1667053.1667059

[8] Fisher, K., Walker, D.: The PADS project: an overview. In: Proceedings of the 14th International Conference on Database Theory. pp. 11–17. ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1938551.1938556

[9] Keller, G., Murray, T., Amani, S., O'Connor-Davis, L., Chen, Z., Ryzhyk, L., Klein, G., Heiser, G.: File systems deserve verification too! In: Workshop on Programming Languages and Operating Systems (PLOS). pp. 1–7. Farmington, Pennsylvania, USA (Nov 2013)

[10] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: ACM Symposium on Operating Systems Principles. pp. 207–220. ACM, Big Sky, MT, USA (Oct 2009)

[11] Klein, G., Sewell, T., Winwood, S.: Refinement in the formal verification of seL4, pp. 323–339. Springer (Mar 2010)

[12] McCann, P.J., Chandra, S.: PacketTypes: abstract specification of network protocol messages. In: Proceedings of the ACM Conference on Communications.

pp. 321–333. ACM, New York, NY, USA (2000), http://doi.acm.org/10.1145/347059.347563

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

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

[15] de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. No. 47 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, United Kingdom (1998)

[16] Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)

[17] Tuch, H.: Formal Memory Models for Verifying C Systems Code. Ph.D. thesis, UNSW, Sydney, Australia (Aug 2008)

[18] Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 97–108. ACM, Nice, France (Jan 2007)

[19] Wang, Y., Gaspes, V.: An embedded language for programming protocol stacks in embedded systems. In: Proceedings of the 20thACM SIGPLAN workshop on Partial evaluation and program manipulation. pp. 63–72. ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1929501.1929511