A Data Layout Description Language for Cogent

(Extended Abstract)

Zilin Chen[1 2] Matthew Di Meglio[2] Liam O'Connor[2] Partha Susarla[1]
Christine Rizkallah[2] Gabriele Keller[3]

[1] Data61, CSIRO
Sydney, Australia

[2] UNSW
Sydney, Australia

[3] Utrecht University
The Netherlands

1 Introduction

While purely functional languages allow for reasoning about code equationally and productively, they often do not give systems programmers sufficiently fine-grained control for achieving the level of efficiency they desire. Our aim is to reduce the effort required for producing reliable, efficient systems.

Cogent [5] is a restricted uniqueness-typed purely functional language for writing high-assurance systems code [1]. Cogent has a certifying compiler that generates efficient C code [5, 7] by making use of Cogent's uniqueness types.

Cogent programs do not exist in isolation. Typically, a Cogent program constitutes a component of a larger system, written in C, which is connected to the Cogent program using a foreign function interface (FFI). The aim, when building a system using Cogent, is to write as much of it in Cogent as possible, because the effort needed to verify low level imperative C code is significantly higher than that needed to verify Cogent code. Cogent programs are defined as pure functions operating on *algebraic data types*. The exact layout of these data types in memory is determined by the Cogent 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 our Cogent compiler.

Therefore, the systems written in Cogent must maintain a great deal of glue code to synchronise between two copies of the same conceptual data structure [1]. As Cogent code can only interact with the Cogent data representation, this glue code is currently 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. To solve these issues we want to be able to write this type of code in Cogent.

To do so, we propose a new framework that allows for data abstraction in Cogent programs. Rather than maintaining two copies of data, we define a 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. With Dargent, the programmer can write code as usual, manipulating ordinary Cogent data types, and after compilation the generated C code will manipulate kernel data structures directly, without extensive copying and synchronisation at run-time.

This will improve performance by eliminating redundant code, simplifying the integration of C and Cogent code, and enabling users to write and verify more Cogent code rather than reasoning about cumbersome C code. Dargent eliminates the need for a standalone language for marshalling and unmarshalling data (such as PADS [3], Nail [2]). Moreover, it allows programmers a level of fine-grained control over memory usage similar to that provided by C.

So far, we have designed Dargent (Section 2), implemented some of the compilation phases, and formalised our Dargent prototype design in Agda. We have also implemented some essential extensions to the Cogent language (Section 3) to accommodate the data layout descriptions.

2 The Dargent Language

Dargent describes how a Cogent algebraic data type may be laid out in memory, down to the bit level. Data descriptions in Dargent will influence the generated definitions and proofs that constitute the compilation certificate between Cogent and the generated C code.

Figure 1 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, $\{\mathsf{x} : \mathsf{U8},\ \mathsf{y} : \mathsf{U16},\ \mathsf{z} : \langle \mathsf{A}\ X \mid \mathsf{B}\ \mathsf{U16}\rangle\}$. 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 accommodating data layouts where certain parts of the data type must appear at particular offsets, such as with the container pattern. [1] 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.

In the record, the field y starts at bit 0 (0b), occupying 2 bytes (2B); the field x comes right after y, starting at byte 2, and takes 8 bits (8b). The variant field z is represented according to the *Nested* description, offset by three bytes

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/kernel.h?h=v4.19-rc8#n995. Accessed on November 21, 2018.

```
layout Example  = record  { y : 2B at 0b
                          , x : 8b at 2B
                          , z : Nested at 3B }

layout Nested   = variant (1b at 2b)
                          { A(1) : 32b at 1B
                          , B(0) : 16b at 24b }
```

```
{ x : U8 , y : U16 , z : ⟨ A X | B U16 ⟩ }
```

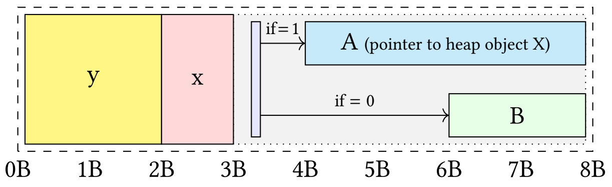

Figure 1. A Cogent type (upper-right) laid out (lower-right) according to a Dargent spec (left).

(3B). 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 pointer-to-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).

3 Extensions to Cogent

A number of extensions must be made to Cogent to accommodate our new data layout description framework.

Type system The type system needs to incorporate Dargent data layouts. In Cogent, there are two types of records: (heap-allocated) boxed records and (stack-allocated) unboxed records. Previously, we proposed an explicit pointer type to carry data layout information [6]. However, such a pointer type would merely be a special case of a boxed record with only one field. Therefore, boxed records are the only top-level heap-allocated datatypes that need to be laid out. Thus Dargent descriptions should be attached to boxed records.

Code generator The code generator needs to compile each abstract read and write operation on Cogent data types to the equivalent concrete operation in C using information provided by Dargent. Furthermore, because Cogent programs often copy (parts of) heap allocated objects to the stack, the compiler must also generate code to convert between the stack and heap representations of each heap-allocated type.

Verification framework Cogent has a certifying compiler [5, 7] which in addition to C code, produces a shallow embedding of Cogent in Isabelle/HOL and a formal proof that the generated C code refines the shallow embedding. Any correctness theorem proven about the shallow embedding also applies to the generated C. We plan to update our verification framework to account for Dargent. The C semantics [4, 8] our Cogent compiler relies on does not allow a reinterpretation or multiple interpretations of heap objects. This limitation has an inevitable impact on our Dargent design and on our Dargent compiler implementation. To account for this, the compiler explicitly generates setter and getter functions for abstract datatypes embedded in a composite type, such as records. When we update our verification framework, we will have to axiomatize the correctness of these setter and getter functions. While they cannot be verified in our C framework, their implementations should be so trivial that their correctness can be established by inspection.

4 Extensions to Dargent

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, we plan to extend our initial prototype of Dargent to support a number of additional semantic features.

Tighter C Integration 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.

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.

Expressiveness Our Dargent prototype currently supports a limited form of layouts. We plan to extend Dargent to allow for defining bit- and byte-level endianness, necessary for implementing network protocols, dynamically sized data, dependent fields, and constraints on permissible values.

References

[1] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 175–188.

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

[3] Kathleen Fisher and David Walker. 2011. The PADS project: an overview. In *Proceedings of the 14th International Conference on Database Theory*. ACM, New York, NY, USA, 11–17. http://doi.acm.org/10.1145/1938551.1938556

[4] David Greenaway, June Andronick, and Gerwin Klein. 2012. Bridging the Gap: Automatic Verified Abstraction of C. In *International Conference on Interactive Theorem Proving*. Springer, Princeton, New Jersey, USA, 99–115.

[5] 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.

[6] Liam O'Connor, Zilin Chen, Partha Susarla, Christine Rizkallah, Gerwin Klein, and Gabriele Keller. 2018. Bringing Effortless Refinement of Data Layouts to Cogent. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. To appear.

[7] 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.

[8] Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, Bytes, and Separation Logic. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Nice, France, 97–108.