Designing Data Layout Specification Languages

Jakob Schuster

October 2023

1 Introduction

Porgrammers writing software often enjoy the conceptual abstraction of types and values, without considering the actual layout of their values in memory, on a bit- and byte-level. The layout of types is typically reserved for the compiler, handled under the programmers' feet. However, some low-level programmers require control over how their objects are layed out. When writing device drivers, file systems, and network software, systems programmers need to closely design their data layout for compatibility or performance reasons. In these situations, they must typically throw away the abstraction of the data type, writing code which performs complex bit-twiddling operations with tedious and error-prone implementations. We desire a best-of-both-worlds solution; where the programmer can design the layout of their values, without giving up safety to manage the getting and setting of their structures by hand.

One solution to this is the idea of data layout specification, as a language feature or extension. In this approach, the programmer dictates to the compiler how it should lay out values of a particular type, and the compiler generates appropriate getter and setter functions reliably following their specification. Dargent is one leading example, applying this idea to the functional language Cogent and verifying its marshalling operations ([1]). Another similar approach by Baudon et al. introduces Ribbit, generating code targeting LLVM IR ([2]).

In this report, we contribute to the development of data layout languages and discuss their application and formalisation. First, we extend Dargent to allow for the laying out of recursive types. Then, we propose a data layout language for C, giving a static semantics of its core syntax to define layout well-formedness and a matching relation between layouts and types. In doing this, we give an account of how C automatically generates layouts for its types. In the design and formalisation of this language, we follow the example of Dargent, essentially reproducing the steps of Chen's formalisation in a new language setting ([3]). Finally, we discuss more broadly the relationship of layouts and types, speculating on how programming languages could treat data layouts more thoroughly to allow for greater portability and interoperability.

2 Preliminaries

Notation

We assume the knowledge of basic set theory concepts, and reserve this section for specific notation that appears throughout this report.

Ordered Lists We use overlines $\overline{x_i}$ to represent an ordered list of elements. Often, we treat lists like $Q = \overline{k : v}$ as *environments*, where looking up a key k returns its value v. This lookup operation is notated as $Q.k = v$.

Inference Rules Most static semantics are given through inference rules, where premises are written above the line and the conclusion is written below the line. The rules below provide an example.

$$\frac{}{0 \ \mathsf{nat}}\textsc{ZeroNat} \qquad \frac{n \ \mathsf{nat}}{\mathrm{s}\,(n) \ \mathsf{nat}}\textsc{SuccNat}$$

Denotation Brackets Where we give denotational semantics, we use denotation brackets $[\![\cdot]\!]$ around the syntactic construct.

Cogent and Dargent

Cogent is a functional programming language which is transpiled to C. It is designed with systems programming in mind, and features a uniqueness type system which allows for two different semantics: one imperative semantics useful in C code generation, and one purely functional semantics which simplifies the process of verifying correctness properties. These aspects of Cogent are central to its purpose, but they will not be the subject of our discussion in this report. We will consider Cogent in the limited capacity of being the host language to Dargent, a data layout language which allows Cogent programmers to dictate to the compiler how their types should be layed out. Dargent's design is declarative; memory specifications are written once by the programmer, with no need to write getters and setters or worry about low-level memory details anywhere else in the program. The actual implementation and management of data with custom layouts is handled by the compiler. Dargent also fits into the Cogent verification framework, generating correctness proofs.

$$\begin{array}{llcll}
\text{Types} & \tau & ::= & T & \text{(primitive types)}\\
 & & | & () & \text{(unit)}\\
 & & | & \alpha & \text{(type variables)}\\
 & & | & t_r & \text{(recursive type variables)}\\
 & & | & \mu\ \{\overline{f_i : \tau_i}\}\ s & \text{(recursive records)}\\
 & & | & \langle\overline{A_i\ \tau_i}\rangle s & \text{(variants)}\\
 & & | & \mathsf{K}\ \overline{\tau_i}\ s & \text{(abstract types)}\\
 & & | & \ldots & \\
\text{Primitive types} & T & ::= & \mathtt{U}n \mid \mathtt{Bool} & \text{(n = 1 \ldots 64)}\\
\text{Field names} & & \ni & \mathrm{f} & \\
\text{Constructors} & & \ni & \mathrm{A} & \\
\text{Sigils} & s & ::= & \textcircled{b} \mid \textcircled{u} & \\
 & \textcircled{b} & ::= & \textcircled{w} \mid \textcircled{r} & \\
\text{Recursive parameters} & \mu & ::= & \mathbf{mu}\ t. \mid \epsilon &
\end{array}$$

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

Figure 1: The syntax of Cogent types, with recursive parameters.

3 Extending Dargent to Recursive Types

With the addition of recursive types (μ-types), records can be optionally prefixed with a recursive parameter (Figure 1). Recursive parameter variables t_r can occur within μ-types. The approach introduced by Murray lays out μ-types using pointers, as opposed to a 'flattened' memory layout ([4]). As a result, an intuitive Dargent layout can be written using pointers; explicit recursion need not be present in the memory layout of a μ-type.

```
layout Number = record {
  l : record {
    variant (1b) { Zero(0) : 0B, Succ(1) : pointer }
  }
}
```

To allow for this new use of syntax, we modify Dargent's main typing rule ([1]), with our modifications highlighted :

$$\frac{\begin{array}{c}\overline{\mathrm{f}_i}\ \text{distinct} \qquad \text{for each } i\text{:}\ \Delta; C \vdash \tau_i\ \mathbf{wf}\\ \ell\ \mathbf{wf} \qquad \ell \sim \mu\,\{\overline{\mathrm{f}_i : \hat{\tau}_i}\} \qquad (\ell,\ \mu\,\{\overline{\mathrm{f}_i : \hat{\tau}_i}\}) \in C\end{array}}{\Delta; C \vdash \mu\,\{\overline{\mathrm{f}_i :: \tau_i}\} b_\ell\ \mathbf{wf}}\textsc{RecWf}$$

Clearly, the only change is the insertion of μ-parameters. We also tweak the matching relation between layouts and types. We account for the new record syntax in URecordMatch, and we capture the relation of μ-type parameter variables t_r as pointers in the MuParamMatch rule. (Figure 2). Since the surface syntax of Dargent needs no alteration, we can leave untouched the layout-wellformedness and code generation aspects of Dargent.

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

$$\frac{\text{bits}\ (T) = s}{\mathsf{BitRange}\ (o, s) \sim T}\textsc{PrimTyMatch} \qquad \begin{array}{ll} \text{bits}\ (\mathsf{U}n) & = n \\ \text{bits}\ (\mathsf{Bool}) & = 1 \end{array}$$

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

$$\frac{}{x\{o\} \sim \hat{\tau}}\textsc{VarMatch}$$

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

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

Figure 2: Matching relation between layouts and types, extended to μ-types.

4 Dargent for C

The idea of surface syntax to control the layout of types can be applied to languages beyond COGENT. In this section, we apply it to the widely-used systems programming language C, as it is a language in which programmers often apply non-intuitive bit-twiddling approaches to manually manage the memory of their data. We formalise a syntax and its associated static semantics to afford C programmers control over the memory layout of their types.

In our syntax, each type can be associated with a desired memory representation using the as keyword (Figure 3). The position of each field of a struct can be moved to a position in memory, either absolutely using at, or relative to another field using after/before. As shown in the example above, this positioning system allows programmers to break C's traditional padding specifications and pack structs to meet their needs. In the following sections, we formalise this language extension.

Bits and Bytes

First, we define a generic language of bits and bytes, and operations thereupon, which we will use across our later language definitions. Memory quantities (typically representing either the size or offset of some object) are expressed as quantities of *bits* $m\mathsf{b}$ and *bytes* $n\mathsf{B}$.

$$\begin{array}{lll} \text{Memory} & o, s ::= & n\mathsf{B} \mid m\mathsf{b} \mid n\mathsf{B}\ m\mathsf{b} \\ \text{Naturals} & \ni & n, m \end{array}$$

Conversion functions are defined between sizes and the number of bits they denote in $\mathbb{N}$, allowing us to define addition, subtraction, scalar multiplication and the modulo operation on sizes.

```
struct {
    struct { int a ; _Bool b ; } fst;
    struct { char c; }* ptr ;
    union { char d ; char e [3]; } sum ;
} as {
    fst: { a : 4B after b; b : 1B at 0B; };
    ptr : 8B at 8B;
    sum : 3B at 5B;
} example;
```

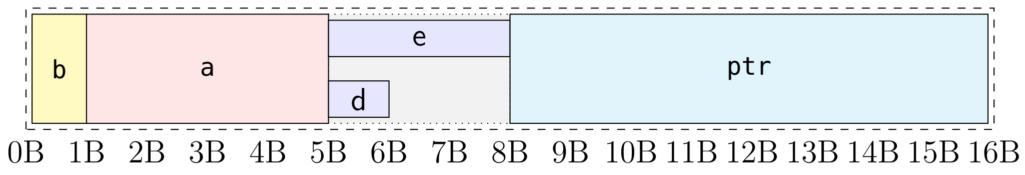

Figure 3: A demo of the C layout language.

$$
\begin{array}{lcc}
\mathrm{bits}\,(n\mathsf{B}\ m\mathsf{b}) & = & \mathrm{bits}\,(n\mathsf{B}) + \mathrm{bits}\,(m\mathsf{b}) \\
\mathrm{bits}\,(n\mathsf{B}) & = & 8 \cdot n \\
\mathrm{bits}\,(m\mathsf{b}) & = & m \\
\\
\mathrm{size}\,(n) & = & \lfloor \frac{n}{8} \rfloor \mathsf{B}\ (n \bmod 8)\mathsf{b} \\
\\
s_1 + s_2 & = & \mathrm{size}\,(\mathrm{bits}\,(s_1) + \mathrm{bits}\,(s_2)) \\
s_1 - s_2 & = & \mathrm{size}\,(\mathrm{bits}\,(s_1) - \mathrm{bits}\,(s_2)) \quad [s_1 \geq s_2] \\
s \cdot n & = & \mathrm{size}\,(\mathrm{bits}\,(s) \cdot n) \\
\frac{s_1}{s_2} & = & \frac{\mathrm{bits}(s_1)}{\mathrm{bits}(s_2)} \\
s \bmod n & = & \mathrm{size}\,(\mathrm{bits}\,(s) \bmod n) \\
\\
\mathrm{aligned}\,(o, s) & = & s \cdot \left\lceil \frac{o}{s} \right\rceil
\end{array}
$$

We define a function aligned (o, s) to handle C's notion of alignment, finding the next alignment after o which is a multiple of an alignment size s. For example, aligned $(3\mathsf{B}\ 7\mathsf{b}, 2\mathsf{B}) = 4\mathsf{B}$; three *padding bits* are required to align $3\mathsf{B}\ 7\mathsf{b}$ to the nearest $2\mathsf{B}$. We also define a *range* $(\!|\ o, s\ |\!)$ to represent a region of bits starting at offset o with size s.

$$\text{Ranges } r ::= (\!|\ o, s\ |\!)$$

We also have a domain of *fields*, shared across all of the language fragments we discuss.

$$\text{Fields} \ni f$$

Types in C

Next, we give the type system of C. C's surface syntax as outlined in the specification makes it difficult to separate type information from the other language features of C; in our formalisation, taking strong inspiration from CompCert ([5]), the types have been isolated for simplicity (Figure 4). One C feature worth discussing is the assignment of attributes α, allowing the programmer to add special behaviours to variables, such as aligning one type as though it were another in memory.

Types	τ	$::=$	void	(the void type)
		$\mid$	$\mathsf{char}(\sigma)$	(character types)
		$\mid$	$\mathsf{int}(\sigma, s_i)$	(integer types)
		$\mid$	float	(floating-point)
		$\mid$	$\mathsf{double}(s_d)$	(double-precision int)
		$\mid$	$\mathsf{array}\,(\tau, n)$	(array of τ)
		$\mid$	$\mathsf{struct}\,\left(\overline{f_i : \tau_i}\right)$	(structure)
		$\mid$	$\mathsf{union}\,\left(\overline{f_i : \tau_i}\right)$	(union)
		$\mid$	$\mathsf{pointer}\,(\tau)$	(pointer to τ)
		$\mid$	$\mathsf{enum}\,\left(\overline{f_i}\right)$	(enumeration type)
		$\mid$	β	(type variable)
		$\mid$	$\mathsf{fun}\,(\overline{\tau_i}, \overline{\tau_j}, \tau_r)$	(function with $\overline{\tau_i}$ man, $\overline{\tau_j}$ opt args)
		$\mid$	$\mathsf{bitfield}\,\left(\tau, \overline{f_i : r}\right)$	(bitfield with basic type τ, fields $\overline{f_i}$, ranges $\overline{r_i}$)
		$\mid$	$\tau\,(\alpha)$	(τ with attribute α)
		$\mid$	τ as l	(τ layed out as l)
Signedness	σ	$::=$	$\mathsf{signed} \mid \mathsf{unsigned}$	
Int sizes	s_i	$::=$	$\mathsf{bool} \mid \mathsf{short} \mid \mathsf{default}$	(valid integer sizes)
		$\mid$	$\mathsf{long} \mid \mathsf{long\ long}$	
Double sizes	s_d	$::=$	$\mathsf{default} \mid \mathsf{long}$	
Attributes	α	$::=$	$\mathsf{align\ as}\ \tau \mid \mathsf{volatile} \mid \cdots$	(optionally, align the type as though it were τ)
Type variables		$\ni$	β	

Figure 4: Type system of C, adapted from the CompCert formalisation, with some omissions to focus on memory-relevant properties.

Bit-fields

Another C type worth discussing is the bit-field, a language feature with highly particular memory behaviour. A bit-field can be thought of as a region whose size is a multiple of some basic type τ (usually an integer), with specific subsections of this region given names, and others marked as empty padding (Figure 5). Special syntax `:0` pads the bit-field up to the next multiple of its basic type. The bit-field is C's way of making bit-packing more ergonomic, a goal we accomplish with layout specifications. In our formalisation, each individual field of the bit-field does not stand alone as its own type; rather, we capture them all as a list of field-range pairs under one type, $\mathsf{bitfield}\left(\tau, \overline{f_i : r_i}\right)$.

```
struct {
    int a : 8; int : 2; int b : 12; int : 0;
    int c : 4; int : 0;
    int d : 20;
} str;
```

$$\mathsf{bitfield}(\ a : \llparenthesis\ 0\mathsf{b}, 8\mathsf{b}\ \rrparenthesis\ ,\ b : \llparenthesis\ 10\mathsf{b}, 12\mathsf{b}\ \rrparenthesis\ ,\ c : \llparenthesis\ 1\mathsf{B}, 4\mathsf{b}\ \rrparenthesis\ ,\ d : \llparenthesis\ 8\mathsf{B}, 20\mathsf{b}\ \rrparenthesis\ )$$

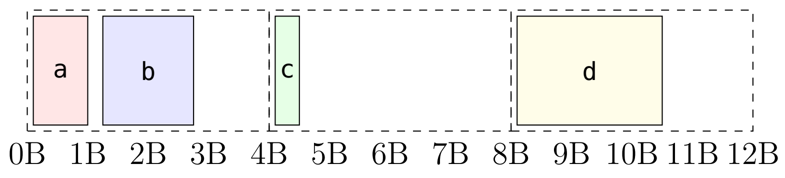

Figure 5: A bit-field, specified in C, shown in memory, and captured by our formalisation.

Reducing to CR

Even this stripped-back formalisation includes more information than we need for reasoning about memory. We introduce a reduced type system CR (Figure 6), for two reasons: first, to minimize the complexity of our matching relation later, and second, to give the reader an intuitive sense of what should matter to us about these C types, and what can be discarded. In particular, note that CR features a set of basic memory constants ρ which are not arbitrarily sized.

Now we establish a reduction $\rightsquigarrow$ from C to CR (Figure 7). Some reductions are trivial, but others provide insight. For example, VoidRed codifies that, in accordance with the language specification, void types are layed out using the char memory representation. Many complex types drop information; for example, it doesn't matter what type a $\mathsf{pointer}\,(\tau)$ is pointing to, since all pointers use the same constant representation pointer. The arguments and return type of a $\mathsf{fun}\,(\overline{\tau_i}, \overline{\tau_j}, \tau_r)$ are also not important to us, as all functions are simply represented as pointers. It is important to remember that we are not checking the correctness of these types, which is what allows us to discard so much.

$$
\begin{array}{llll}
\text{Basic memory types} & \rho & ::= \mathsf{pointer} \mid \mathsf{char} & \text{(memory constants)} \\
& & \mid\ \mathsf{bool} \mid \mathsf{short} \mid \mathsf{int} \mid \mathsf{long} \mid \mathsf{long\ long} & \\
& & \mid\ \mathsf{float} \mid \mathsf{double} \mid \mathsf{long\ double} & \\
\text{Types} & \tau' & ::= \rho & \text{(basic type } \rho \text{ aligned as } \tau'\text{)} \\
& & \mid\ \tau'\,[n] & \\
& & \mid\ \left\{\overline{f_i : \tau'_i}\right\} & \\
& & \mid\ \left\langle\overline{f_i : \tau'_i}\right\rangle & \\
& & \mid\ \left|\tau', \overline{f_i : r_i}\right| & \\
& & \mid\ \tau'_1\ \mathsf{aligned\ like}\ \tau'_2 & \\
& & \mid\ \mathsf{as}\ \ell' &
\end{array}
$$

Figure 6: Reduced type system CR, containing only the information necessary for matching against memory layouts.

Layouts in L

We now introduce a syntax for memory layouts in C (Figure 8). We'll refer to this layout language as L. It has slightly less syntax than Dargent, as unions in C are not tagged, allowing them to be handled as simple memory sizes. One difference between L and Dargent is the inclusion of the basic types ρ as layouts, useful due to C's large number of primitive memory types. Also, in addition to placing one field after another, L includes syntax to place one field *before* another. Aside from these minor differences, its language design is very similar to Dargent.

Reducing to LR

Similar to the approach taken by Chen ([3]), we establish a core layout language LR on which we apply our semantics. Layouts in LR fundamentally consist of bit ranges describing the size and position of each piece of data, as well as structs containing a number of named fields. Endianness is still present in LR, with the endianness of the target machine represented by ME.

Reduction from L to LR (Figure 10) takes place in a context where:

- Q is the environment of layout variables $\overline{l : \ell'}$.
- L contains the layouts of the other fields present in the struct $\overline{f : \ell}$, used for relative positioning.

These rules simply desugar the syntax of L, reducing it to bit ranges. In AtRed, the absolute position is translated to an offset which is applied to the layout. In BeforeRed and AfterRed, the name of the field f is looked up in L, and the offset o is calculated to position the layout either directly before or directly after the other field.

$$\boxed{B, Q \vdash \tau \rightsquigarrow \tau'}$$

$$\frac{B, Q \vdash \tau \rightsquigarrow \tau' \qquad B, Q \vdash \tau_a \rightsquigarrow \tau'_a}{B, Q \vdash \tau\ (\mathsf{aligned\ like}\ \tau_a) \rightsquigarrow \tau'\ \mathsf{aligned\ like}\ \tau'_a}\ \textsc{AlignAttrRed}$$

$$\frac{B, Q \vdash \tau \rightsquigarrow \tau' \qquad \alpha \neq \mathsf{aligned\ like}\ \tau_a}{B, Q \vdash \tau\ (\alpha) \rightsquigarrow \tau'}\ \textsc{DropAttrRed}$$

$$\frac{}{B, Q \vdash \mathsf{void} \rightsquigarrow \mathsf{char}}\ \textsc{VoidRed} \qquad \frac{}{B, Q \vdash \mathsf{char}(\sigma) \rightsquigarrow \mathsf{char}}\ \textsc{CharRed}$$

$$\frac{}{B, Q \vdash \mathsf{int}(\sigma, \mathsf{bool}) \rightsquigarrow \mathsf{bool}}\ \textsc{BoolRed} \qquad \frac{}{B, Q \vdash \mathsf{int}(\sigma, \mathsf{short}) \rightsquigarrow \mathsf{short}}\ \textsc{ShortRed}$$

$$\frac{}{B, Q \vdash \mathsf{int}(\sigma, \mathsf{default}) \rightsquigarrow \mathsf{int}}\ \textsc{IntRed} \qquad \frac{}{B, Q \vdash \mathsf{int}(\sigma, \mathsf{long}) \rightsquigarrow \mathsf{long}}\ \textsc{LongRed}$$

$$\frac{}{B, Q \vdash \mathsf{int}(\sigma, \mathsf{long\ long}) \rightsquigarrow \mathsf{long\ long}}\ \textsc{LongLongRed}$$

$$\frac{}{B, Q \vdash \mathsf{float} \rightsquigarrow \mathsf{float}}\ \textsc{FloatRed}$$

$$\frac{}{B, Q \vdash \mathsf{double}(\mathsf{default}) \rightsquigarrow \mathsf{double}}\ \textsc{DoubleRed}$$

$$\frac{}{B, Q \vdash \mathsf{double}(\mathsf{long}) \rightsquigarrow \mathsf{long\ double}}\ \textsc{LongDoubleRed}$$

$$\frac{\tau \rightsquigarrow \tau'}{B, Q \vdash \mathsf{array}\ (\tau, n) \rightsquigarrow \tau'\ [n]}\ \textsc{ArrayRed}$$

$$\frac{\text{for each } i\text{: } \tau_i \rightsquigarrow \tau'_i}{B, Q \vdash \mathsf{struct}\ \left(\overline{f_i : \tau_i}\right) \rightsquigarrow \left\{\overline{f_i : \tau'_i}\right\}}\ \textsc{StructRed} \qquad \frac{\text{for each } i\text{: } \tau_i \rightsquigarrow \tau'_i}{B, Q \vdash \mathsf{union}\ \left(\overline{f_i : \tau_i}\right) \rightsquigarrow \left\langle\overline{f_i : \tau'_i}\right\rangle}\ \textsc{UnionRed}$$

$$\frac{}{B, Q \vdash \mathsf{pointer}\ (\tau) \rightsquigarrow \mathsf{pointer}}\ \textsc{PointerRed} \qquad \frac{}{B, Q \vdash \mathsf{enum}\ \left(\overline{f_i}\right) \rightsquigarrow \mathsf{int}}\ \textsc{EnumRed}$$

$$\frac{B.\beta \rightsquigarrow \tau'}{B, Q \vdash \beta \rightsquigarrow \tau'}\ \textsc{TyVarRed} \qquad \frac{}{B, Q \vdash \mathsf{fun}\ (\overline{\tau_i}, \overline{\tau_j}, \tau_r) \rightsquigarrow \mathsf{pointer}}\ \textsc{FunRed}$$

$$\frac{}{B, Q \vdash \mathsf{bitfield}\ \left(\tau, \overline{f_i : r_i}\right) \rightsquigarrow \left|\tau', \overline{f_i : r_i}\right|}\ \textsc{BitFieldRed}$$

$$\frac{}{B, Q \vdash \tau\ \mathsf{as}\ l \rightsquigarrow \mathsf{as}\ Q.l}\ \textsc{AsRed}$$

Figure 7: Rules for reducing C to CR.

$$\begin{array}{llll}
\text{Layouts} & \ell & ::= s & \text{(memory blocks)} \\
 & & \mid\ \rho & \text{(memory constants)} \\
 & & \mid\ \ell \ \mathsf{at}\ s & \text{(absolute offsets)} \\
 & & \mid\ \ell \ \mathsf{after}\ f \mid \ell \ \mathsf{before}\ f & \text{(relative offsets)} \\
 & & \mid\ \ell \ \mathsf{using}\ \omega & \text{(endianness)} \\
 & & \mid\ l & \text{(variables)} \\
 & & \mid\ \left\{\overline{f_i : \ell_i}\right\} & \text{(structs)} \\
\text{Endianness} & \omega & ::= \mathsf{BE} \mid \mathsf{LE} &
\end{array}$$

Figure 8: The syntax of our C layout language, L.

$$\begin{array}{llll}
\text{Layouts} & \ell' & ::= \ r_{\omega'} & \text{(primitive layouts)} \\
 & & \mid\ \ \left\{\overline{f_i : \ell'_i}\right\} & \text{(structs)} \\
\text{Endianness} & \omega' & ::= \ \mathsf{BE} \mid \mathsf{LE} \mid \mathsf{ME} & \\
\text{Field names} & & \ni \ \ f &
\end{array}$$

$$\text{start, end, len} : \text{Layout} \to \text{Sizes}$$

$$\text{start}\left(\llparenthesis\, o, s \,\rrparenthesis_{\omega'}\right) = o \qquad \text{end}\left(\llparenthesis\, o, s \,\rrparenthesis_{\omega'}\right) = o + s$$

$$\text{start}\left(\left\{\overline{f_i : \ell'_i}\right\}\right) = \min\left(\bigcup_i \text{start}\left(\ell'_i\right)\right) \qquad \text{end}\left(\left\{\overline{f_i : \ell'_i}\right\}\right) = \max\left(\bigcup_i \text{end}\left(\ell'_i\right)\right)$$

$$\text{len}\left(\ell'\right) = \text{end}\left(\ell'\right) - \text{start}\left(\ell'\right)$$

$$\text{offset} : \text{Layouts} \to \text{Layouts}$$

$$\text{offset}\left(o, \llparenthesis\, o', s \,\rrparenthesis_{\omega'}\right) = \llparenthesis\, o + o', s \,\rrparenthesis_{\omega'}$$

$$\text{offset}\left(o, \left\{\overline{f_i : \ell'_i}\right\}\right) = \left\{\overline{f_i : \text{offset}\left(o, \ell'_i\right)}\right\}$$

Figure 9: Our core layout language, LR, and some basic operations.

Static Semantics

We now consider the semantics of LR, starting with well-formedness rules (Figure 11). Here, we think of each layout as denoting a particular *allocation*, a set of bits which it occupies. If a layout considers the same bit to be a part of multiple fields, it is ill-formed. If an allocation can be found for a layout, it is well-formed; that is, $l \ \mathbf{wf} \overset{\text{def}}{=} \exists a. \ell' \asymp a$. Like Chen ([3]), we leave out the scope checking of layout variables.

Importantly, a layout's well-formedness does not guarantee that it will be able to match with any particular type. For this, we establish the *matching relation*. Before we can do this, we need to know the size of C's types, which is not a straightforward calculation.

To understand C sizes and alignment requirements, we formalise C's default scheme for laying out a particular type, which we call the *auto-matching relation*. The auto-matching

$$\boxed{Q, L \vdash \ell \rightsquigarrow \ell'}$$

$$\frac{0\mathsf{b} \vdash \rho \mathrel{\tilde{\sim}} \ell'}{Q, L \vdash \rho \rightsquigarrow \ell'}\textsc{BasicTypeRed} \qquad \frac{}{Q, L \vdash s \rightsquigarrow (\!|\, 0\mathsf{b}, s \,|\!)_{\mathsf{ME}}}\textsc{SizeRed}$$

$$\frac{Q, L \vdash \ell \rightsquigarrow \ell'}{Q, L \vdash \ell \;\mathsf{at}\; s \rightsquigarrow \mathrm{offset}\,(s, \ell')}\textsc{AtRed} \qquad \frac{}{Q, L \vdash l \rightsquigarrow Q.l}\textsc{VarRed}$$

$$\frac{\begin{array}{c} Q, L \vdash L.f \rightsquigarrow \ell'_f \\ Q, L \vdash \ell \rightsquigarrow \ell' \qquad o = \left[\mathrm{end}\left(\ell'_f\right) + 1\right]\mathsf{b} \end{array}}{Q, L \vdash \ell \;\mathsf{after}\; f \rightsquigarrow \mathrm{offset}\,(o, \ell')}\textsc{AfterRed}$$

$$\frac{\begin{array}{c} Q, L \vdash L.f \rightsquigarrow \ell'_f \\ Q, L \vdash \ell \rightsquigarrow \ell' \qquad o = \left[\mathrm{start}\left(\ell'_f\right) - \mathrm{len}\,(\ell')\right]\mathsf{b} \end{array}}{Q, L \vdash \ell \;\mathsf{before}\; f \rightsquigarrow \mathrm{offset}\,(o, \ell')}\textsc{BeforeRed}$$

$$\frac{\text{for each } i\text{: } Q, \overline{f_i : \ell_i} \vdash \ell_i \rightsquigarrow \ell'_i}{Q, L \vdash \left\{\overline{f_i : \ell_i}\right\} \rightsquigarrow \left\{\overline{f_i : \ell'_i}\right\}}\textsc{StructRed}$$

Figure 10: Rules for reducing layouts to core layouts.

$$\boxed{\ell' \asymp a}$$

$$\frac{}{(\!|\, o, s \,|\!)_{\omega'} \asymp \{\, \text{bits}\,(o)\,, \text{bits}\,(o) + 1, \cdots, \text{bits}\,(o+s)\,\}}\textsc{RangeWF}$$

$$\frac{\text{for each } i\text{: } \ell'_i \asymp a_i \qquad \overline{a_i} \text{ disjoint}}{\left\{\overline{f_i : \ell'_i}\right\} \asymp \bigcup_i a_i}\textsc{StructWF}$$

Figure 11: Well-formedness on core layouts.

relation $\ell' \mathrel{\bar{\sim}} \tau'$, means that, without our intervention, C's compiler will lay out a type τ' as ℓ' (Figure 12). These automatic layouts have some implementation-specific behaviour; in particular, the sizes and alignment restrictions on basic memory types ρ vary between compilers, and often between target machines. Implementation-dependent behaviours have been highlighted in the figure. Because certain types will only align to multiples of certain sizes, the auto-matching relation is parameterised over o, the current offset of the type within the struct.

The fields of a struct are automatically layed out one after the other, and the whole struct is aligned according to the requirements of their strictest-aligned field (a complex alignment requirement represented in the rule by $\max\left(\cup_j o_j\right) \vdash \ell'_1 \mathrel{\bar{\sim}} \tau'_1$). Unions similarly use the strictest alignment among their fields, and they allocate as much space as their largest field requires. We simply treat them as a range,

Now we can establish the matching relation $\ell' \sim \tau'$, stating more generally that a layout ℓ' *can be* applied to a type τ' (Figure 13). According to RangeMatch, the alignment of types is overruled by our custom layout language, allowing any type to be placed anywhere by the user, provided that the layout itself is well-formed. In lieu of the tags present in Cogent, unions are simply matched with RangeMatch. Similarly, RangeMatch captures aligned types, and types that have been assigned a layout. Structs match when each field's type corresponds with a matching layout field. For bit-fields, the size of each field must be a match, but the fields can be positioned in any well-formed way.

$$\boxed{o \vdash \ell' \mathrel{\bar{\sim}} \tau'}$$

$$\frac{}{o \vdash \llparenthesis\, \text{aligned}\left(o,\ \mathsf{8B}\right),\ \mathsf{8B}\ \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \mathsf{pointer}}\ \textsc{PointerAuto}$$

$$\frac{}{o \vdash \llparenthesis\, \text{aligned}\left(o,\ \mathsf{1B}\right),\ \mathsf{1B}\ \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \mathsf{char}}\ \textsc{CharAuto}$$

$$\frac{}{o \vdash \llparenthesis\, \text{aligned}\left(o,\ \mathsf{2B}\right),\ \mathsf{2B}\ \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \mathsf{short}}\ \textsc{ShortAuto}$$

$$\frac{}{o \vdash \llparenthesis\, \text{aligned}\left(o,\ \mathsf{4B}\right),\ \mathsf{4B}\ \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \mathsf{int}}\ \textsc{IntAuto}$$

$$\frac{}{o \vdash \llparenthesis\, \text{aligned}\left(o,\ \mathsf{4B}\right),\ \mathsf{4B}\ \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \mathsf{long}}\ \textsc{LongAuto}$$

$$\frac{}{o \vdash \llparenthesis\, \text{aligned}\left(o,\ \mathsf{8B}\right),\ \mathsf{8B}\ \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \mathsf{long\ long}}\ \textsc{LongLongAuto}$$

$$\frac{}{o \vdash \llparenthesis\, \text{aligned}\left(o,\ \mathsf{8B}\right),\ \mathsf{8B}\ \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \mathsf{float}}\ \textsc{FloatAuto}$$

$$\frac{}{o \vdash \llparenthesis\, \text{aligned}\left(o,\ \mathsf{8B}\right),\ \mathsf{8B}\ \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \mathsf{double}}\ \textsc{DoubleAuto}$$

$$\frac{}{o \vdash \llparenthesis\, \text{aligned}\left(o,\ \mathsf{8B}\right),\ \mathsf{12B}\ \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \mathsf{long\ double}}\ \textsc{LongDoubleAuto}$$

$$\frac{o \vdash \ell' \mathrel{\bar{\sim}} \tau'}{o \vdash \llparenthesis\, \text{start}\,(\ell'), \text{len}\,(\ell') \cdot n\ \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \tau'\,[n]}\ \textsc{ArrayAuto} \qquad \frac{}{o \vdash \ell' \mathrel{\bar{\sim}} \mathsf{as}\ \ell'}\ \textsc{AsAuto}$$

$$\frac{\begin{array}{c} o \vdash \ell'_1 \mathrel{\bar{\sim}} \tau'_1 \qquad o \vdash \ell'_2 \mathrel{\bar{\sim}} \tau'_2 \\ p = \text{start}\,(\ell'_2) - \text{start}\,(\ell'_1) \end{array}}{o \vdash \text{offset}\,(p, \ell'_2) \mathrel{\bar{\sim}} \tau'_1\ \mathsf{aligned\ like}\ \tau'_2}\ \textsc{AlignedAuto}$$

$$\frac{\begin{array}{c} \{\text{for each } j\text{: } o \vdash \ell'_j \mathrel{\bar{\sim}} \tau'_j \qquad \text{let } o_j = \text{start}\left(\ell'_j\right)\} \\ \max\left(\bigcup_j o_j\right) \vdash \ell'_1 \mathrel{\bar{\sim}} \tau'_1 \qquad \text{for all } i > 1\text{: end}\left(\ell'_{i-1}\right) \vdash \tau'_i \mathrel{\bar{\sim}} \ell'_i \end{array}}{o \vdash \left\{\overline{f_i : \ell'_i}\right\} \mathrel{\bar{\sim}} \left\{\overline{f_i : \tau'_i}\right\}}\ \textsc{StructAuto}$$

$$\frac{\text{for all } i\text{: } o \vdash \ell'_i \mathrel{\bar{\sim}} \tau'_i}{o \vdash \llparenthesis\, \max\left(\bigcup_i \text{start}\left(\overline{\ell'_i}\right)\right), \max\left(\bigcup_i \text{len}\left(\overline{\ell'_i}\right)\right) \rrparenthesis_{\mathsf{ME}} \mathrel{\bar{\sim}} \left\langle \overline{f_i : \tau'_i} \right\rangle}\ \textsc{UnionAuto}$$

$$\frac{}{\left\{\overline{f_i : (r_i)_{\mathsf{ME}}}\right\} \mathrel{\bar{\sim}} \left|\tau', \overline{f_i : r_i}\right|}\ \textsc{BitfieldAuto}$$

Figure 12: Auto-matching relation $\bar{\sim}$ between CR and LR.

$$\boxed{\ell' \sim \tau'}$$

$$\frac{\ell' \bar{\sim} \tau'}{\llparenthesis\, o, \mathrm{len}\,(\ell')\, \rrparenthesis_{\omega'} \sim \tau'}\textsc{RangeMatch}$$

$$\frac{\text{for each } i\text{: } \ell'_i \sim \tau'_i}{\left\{\overline{f_i : \ell'_i}\right\} \sim \left\{\overline{f_i : \tau'_i}\right\}}\textsc{StructMatch}$$

$$\frac{}{\left\{\overline{f_i : \llparenthesis\, o_i, \mathrm{len}\,(r_i)\, \rrparenthesis_{\mathsf{ME}}}\right\} \sim \left|\tau', \overline{f_i : r_i}\right|}\textsc{BitfieldMatch}$$

Figure 13: Matching relation between reduced layouts and CR types.

Layout Variable Declaration

We have not addressed the declaration of layout variables; that is, the addition of new $l : \ell'$ entries into our layout variable environment Q. For this, we insert layout variable declarations into C's statement syntax. We don't give a full account of the C statement syntax since it comprises a full chapter in the specification ([6]), but we simply append the syntax as follows:

$$\text{Statements } s ::= \cdots \mid\ l = \ell;$$

Next we give a denotational semantics of these layout variable declarations. Layout variable declarations are thought of as an insertion into the environment Q, where the reduction to a core layout ℓ' happens at the point of declaration, taking in the layout variable context Q, and the empty set for the field context L (since it is populated within the reduction, by the StructRed rule).

$$\begin{aligned} Q\ \llbracket\, l = \ell;\, \rrbracket &= l : \ell' \\ &\text{where } Q, \{\} \vdash \ell \rightsquigarrow \ell' \end{aligned}$$

Implementation in C

The focus of this work is purely pen-and-paper, exploring the feasibility of layout specifications interacting with C's type system. As such, we do not thoroughly explore implementation options. Dargent, as an extension of Cogent, is implemented as a transpilation to C, with custom layouts in Dargent transpiling to singleton C structs, with generated getter and setter code ([1]). A similar approach could be taken here, with an extended language C+L transpiling down to C, generating low-level bit-twiddling code to get and set values with custom layouts. A more direct approach would be to include layout specification as a part of the C language, with compilers handling the layouts in their compilation. The language design implications of this would be significant.

5 Discussion

The following observations arose from studying the C and HASKELL language specifications in preparation for writing this report, research which was more difficult than expected. In the case of C, an understanding of type layout comes partially from the C standard, and partially from GCC documentation. For HASKELL, the language specification leaves memory layout very open - GHC documentation is the primary resource, although conclusive information about even the basic memory layout strategies employed by GHC is buried deep within compiler wikis. The spuriousness and ambiguity of the current state of memory layout formalisation is the context for this extended line of inquiry.

Limitations of C Types

Although our layout language for C provides a good basis for future work in this direction, its limitations are clear. Our attempts to create freedom of type representation ran into limitations imposed by the standard. The C standard sits halfway between enforcing restrictions on memory layout so that programmers can write bit-manipulating code, and leaving the details open to implementation so that compilers can optimise. This former part is problematic for us; we want programmers to have control over the bit representation not by simply formalising one fixed representation, but rather through flexibly-represented types which they can specify themselves - made impossible when the language design restricts memory layout. The latter part is also problematic, since leaving the details of memory layout completely unwritten means we rely on compiler documentation, which is often informal or incomplete. Then, any code we write will depend on the behaviour of some particular compiler or machine - behaviour which could change under our feet.

One immediate solution would be to add new types to C whose length is user-specified, like the $\mathsf{U}n$ family of types in COGENT. That way, fields could be created in structs which match custom-size layouts. However, with this solution not only do the type signature and the layout specification contain duplicate information, but it also progresses us further down the path of inseperability of types from their memory representations. To approach the problem differently, we must consider the semantics of types.

Disentangling Language and Memory

Ideally, an object's type conveys semantic information - an integer or a boolean is called as such not because of the way it looks in memory, but because of the *range of things it represents*. Thus, a language should be formalisable in layout-agnostic terms, so that it is truly portable and maximally flexible. Reynolds gives an account of types discussing their semantics in these terms, with type semantics like $[\![\ \mathsf{Int} \ ]\!] = \mathbb{Z}_\perp$ ([7]).

The question then arises: how can programmers write code which takes advantage of facts about memory representation, if it isn't given by the language specification? Here, we need new specification. In particular, it is necessary for language implementers to formalise their treatment of memory for any given type, and for this implementation specification to modularly extend or refine the language semantics, allowing for different compilers to be properly compared. What follows from this is a disentanglement of language semantics from memory.

We will go through one toy example to communicate the idea. Suppose a language A has a type SmallNat. The language specification gives a domain for this type as follows:

$$[\![\text{SmallNat}]\!] = [0, 255]$$

Note the domain of SmallInt is *not* the entire domain of $\mathbb{N}$, and that this design decision is informed by practicality of implementation - however, it still stops short of laying out memory. What is important is that the domain is semantic, referring to mathematical objects outside of the language.

The implementation specification then describes read and write functions for the type - that is, functions from the domain of the type in A, to a domain made up of memory units which are, in this case, binary.

$$\text{Unit} = \mathbb{B}$$

$$\text{read} : \mathbb{B}^8 \to [0, 255] \qquad\qquad \text{write} : [0, 255] \to \mathbb{B}^8$$

$$\text{read}\,(b) = \sum_{i=0}^{7} b_i \cdot 2^i \qquad\qquad \text{write}\,(n) = \text{write}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \langle n \bmod 2 \rangle$$

We could instead specify read and write functions for a most-significant bit ordering, an equivalently compact representation on the same hardware but with different read-write semantics:

$$\text{read} : \mathbb{B}^8 \to [0, 255] \qquad\qquad \text{write} : [0, 255] \to \mathbb{B}^8$$

$$\text{read}\,(b) = \sum_{i=0}^{7} b_i \cdot 2^{7-i} \qquad\qquad \text{write}\,(n) = \langle n \bmod 2 \rangle + \text{write}\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

Suppose, for some reason, we want to compile A on a machine which stores its memory in ternary units, $\mathbb{T} = \{\, 0, 1, 2 \,\}$. We can still establish functions for this without changing the language design, using six trits instead of eight bits:

$$\text{Unit} = \mathbb{T}$$

$$\text{read} : \mathbb{T}^6 \rightharpoonup [0, 255] \qquad\qquad \text{write} : [0, 255] \to \mathbb{T}^6$$

$$\text{read}\,(b) = \sum_{i=0}^{5} b_i \cdot 3^{5-i} \qquad\qquad \text{write}\,(n) = \langle n \bmod 3 \rangle + \text{write}\left(\left\lfloor \frac{n}{3} \right\rfloor\right)$$

The point of this example is how the same language specification can be refined to suit several different implementation contexts without adjustment, because it is defined without reference to memory. Thus, anything proven about A holds across these different environments, with no danger of undefined behaviour.

From this foundation, we could equip the programmer with surface syntax exposing the read and write behaviour of types, allowing them to reason about abstract datatypes while retaining arbitrarily fine control over how those types are represented, going beyond just the limited degree of control we offer in our proposed language L. A programmer in special circumstances may need to represent their integers using an entirely different representation, such as a different bit ordering; with this proposed approach, they would simply need to write reliable read and write functions, and then the compiler would allow them to use their deeply-customised integers as though they were regular integers (albeit potentially with less efficient performance, depending on their representation).

If formally describing the memory representation of every type sounds too time-consuming, consider that once basic types as well as product and sum types have been specified, many other commonly-used types will immediately follow. Importantly, if an existing language is to be ported to new hardware, its language semantics do not need to change, in the way that C's semantics have needed to - famously, advancements in processor hardware motivated the introduction of the `long` type because it was the simplest way to avoid changing specification of the `int` type ([6]). Many of C's complexities may have been avoided had the designers separated layout from their core language and treated it as an implementation detail, to be formalised elsewhere in a modular fashion. A universal, machine-readable language of memory layout specification could be established, allowing language implementations to draw on common libraries of memory representations. This research direction has a lot of promise; further investigation should be done into comparative study of memory layouts in various languages, to establish a common language of memory representation and finally disentangle types from layout.

6 Conclusion

Our investigations into data layout languages have resulted in a successful extension to Dargent's formalisation, accounting for recursive types and treating them as pointers. In addition, we have developed an entirely new layout syntax for C, allowing for the manipulation of struct fields, enabling programmers to easily rearrange and pack their data. Aside from following the steps established by Chen in layout language design ([3]), we also outline C's automatic approach to layout, which provides an understanding of the its type system specificities, particularly with regards to padding and sizing. Finally, we outline possible research directions in the separation of a memory layouts from language semantics, as a new project pursuing portability and reliability.

Acknowledgements

I would like to thank Dr Christine Rizkallah, for supervising me and guiding me through this process with curiosity and expertise. Also, the programming languages and verification reading group, whose conversations are always lively and have taught me much by osmosis. I would also like to thank my parents, who are always eager to support my research - though strangely, they tune out when I talk about type theory. Finally, I thank my partner Bronte, who gives me more than could ever be quantified, whose patience and faith powers me through every challenge.

References

[1] Chen, Z., Lafont, A., O'Connor, L., Keller, G., McLaughlin, C., Jackson, V., and Rizkallah, C. (jan, 2023) Dargent: A Silver Bullet for Verified Data Layout Refinement. *Proc. ACM Program. Lang.,* **7**(POPL).

[2] Baudon, T., Radanne, G., and Gonnord, L. (aug, 2023) Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types. *Proc. ACM Program. Lang.,* **7**(ICFP).

[3] Chen, Z. Towards A Practical High-Assurance Systems Programming Language PhD thesis UNSW Sydney (2023).

[4] Murray, E. Recursive types for cogent PhD thesis Honours thesis, UNSW Sydney, Computer Science & Engineering (2019).

[5] Leroy, X. (2009) Formal verification of a realistic compiler. *Communications of the ACM,* **52**(7), 107–115.

[6] ISO (June, 2018) ISO/IEC 9899:2018 Information technology — Programming languages — C, , fourth edition.

[7] Reynolds, J. C. What do types mean? — From intrinsic to extrinsic semantics pp. 309–327 Springer New York New York, NY (2003).