UNSW
AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Extending Cogent's Data-Description Language with Layout Polymorphism

by

Sahan Fernando

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: December 4, 2019 Student ID: z5113187

Supervisor: Christine Rizkallah

Abstract

Cogent is a linearly-typed functional programming language for writing trustworthy and efficient systems code. To allow easier interoperability with common systems level data structures, a data-description language extension (Dargent) was proposed to allow the schematic declaration of the in-memory layouts of data-types. Unfortunately, when types have multiple distinct layouts, they essentially need to inhabit multiple nearly identical (but entirely distinct to Cogent's type system) types that only differ by assigned layout.

This usually necessitates a lot of unnecessary code repetition and verification overhead, since functions operating on these types have to be manually duplicated for each layout by the programmer.

The project detailed in this report aims to extend the data-description language to support polymorphic layouts by introducing layout variables as well as constraints on the variables to ensure they can fit a given type. This project will also extend the type checker to be able to deal with these additional variables.

Acknowledgements

I want to thank Kai, Liam, Christine and Aleks for being amazing teachers, and for introducing and guiding me through a myriad of incredibly interesting areas of computer science.

I'd also like to thanks to my friends: Andrew, Paul, Minjie, Greg, Emmet, Conrad, Clement and countless more for all the support they've provided me and all the good times we've had during my time at UNSW.

Finally, I want to thank my parents Mangalee and Anand, and my brother Lasath for always supporting and encouraging me, without them I would never have made it where I am today.

Contents

1 Introduction **1**

1.1 Cogent 2

1.2 Compatibility with compact and custom representations of data 3

1.3 Dargent 4

1.4 Layout Polymorphism 5

2 Background **6**

2.1 Cogent 6

2.1.1 Uniqueness typing 7

2.1.2 Dargent 8

2.2 Data Description Languages 9

2.3 Intensional Type Analysis 10

2.4 Formal Semantics for Data Description Languages 11

3 Completed Work **12**

3.1 Methodology 12

3.1.1 Pen-and-Paper Formalization of Dargent Layout Polymorphism 12

3.1.2 Compiler augmentation 13

4 Results: Formalization **15**

4.1 Formalization Strategy 15

4.2 Dargent Formalization 16

4.2.1 Abstract Grammar for Dargent 16

4.2.2 Memory reservation rules 18

4.2.3 How Dargent stores layouts 19

4.2.4 Rules for matching types and layout 21

4.2.5 Modified Cogent typing rules regarding sigils 23

4.2.6 Constraint generation/solving rules 25

4.3 Layout Polymorphism formalization 26

4.3.1 Constraints 26

4.3.2 Constraint generation rules 27

4.3.3 Constraint solving rules 28

5 Conclusion **29**

5.1 Future work 30

List of Figures

1.1 Layout of a TCP header 3
1.2 Bit layouts of ARGB and RGBA pixel format 5

2.1 Example Cogent code 7
2.2 An example Dargent layout, with in-memory layout on right, image source: [1] 8

3.1 Cogent Compiler Pipeline 13

4.1 Abstract grammar for Dargent 17
4.2 Memory reservation rules for Dargent 18
4.3 Cogent definition of sigils 20
4.4 Dargent definition of sigils 20
4.5 Memory reservation rules for primitive types 21
4.6 Memory reservation rules for compound types 22
4.7 Well-formedness rules for types 23
4.8 Some Cogent typing rules for the bang operator 24
4.9 Corresponding Dargent typing rules for the bang operator . . 25
4.10 Modified constraints needed for layout polymorphism 26
4.11 Constraint generation rule for layout polymorphism 27

4.12 Constraint solving rule for well-formedness rule 28

Chapter 1

Introduction

While end-to-end verification of code via formal methods is considered the gold standard for high assurance software projects, this approach is usually considered prohibitively expensive for most projects due to both development time and cost. This view is especially prevalent in the systems programming world, where efficiency concerns often cause developers to prefer languages that are 'close to the metal', like C and C++, over languages that are further abstracted from their compilation targets, at the expense of safety, reliability and fault tolerance.

This has resulted in multiple critical systems software components being effectively riddled with correctness bugs. This problem is worrying due to the effect this has on the correctness of higher level programs, since most higher level applications implicitly assume the correctness of the lower level components they are built upon in their own correctness models.

Many efforts have been made to make formal methods more accessible to offset this, from model-checking tools to formally verified programming

languages, but none so far have been particularly effective in the systems space.

1.1 Cogent

Cogent[2] is a restricted uniqueness-typed functional programming language designed for writing efficient, provably-correct systems code. With a syntax similar to Haskell, it attempts to provide the features and safety of higher level languages, whilst still having the portability and performance of C.

It is however designed to complement rather than replace C, allowing major parts of application logic to be written in it while relying on C for anything the language cannot express. For this reason, Cogent has been designed to seamlessly integrate with C, having a powerful FFI, no dependency on a garbage collector and minimal runtime costs, and generating C code as its output.

The Cogent compiler can also generate shallow embeddings of Cogent programs in Isabelle/HOL, the language of a powerful interactive theorem prover, as well as a proof that the generated C code refines this embedding. This, combined with the equational reasoning the purely functional nature of the language, makes Cogent a great language for writing end-to-end verified software.

16bit source port		16bit dest port
32bit sequence number		
32bit acknowledgment number		
Header length	Reserved	Flags	16bit window size
16bit checksum		16bit urgent pointer

Figure 1.1: Layout of a TCP header

1.2 Compatibility with compact and custom representations of data

A significant amount of systems codebases employ techniques such as bit-packing and unaligned fields to more efficiently represent data-types, with common examples being page table entries in an operating system's virtual memory subsystem, and TCP packet headers, as shown above. It becomes apparent that to be able to effectively interact with most systems libraries, we need some way of handling data held in these representations from within Cogent.

Cogent operates on algebraic data-types, which while powerful enough to represent most systems data-types, have fixed layouts determined by the Cogent compiler. This unfortunately results in a pattern where C data-structures need to repeatedly be converted into a form that cogent can understand, operated on and then converted back.

This pattern adds a lot of unneeded complexity to the code, and increases both execution times and verification costs.

1.3 Dargent

Dargent[3] is a proposed extension for Cogent that allows specification of the in-memory layouts of data structures. Dargent allows us to declare specifications of the layouts of data structures, then attach these layouts to existing Cogent types, which the Cogent compiler will then use for all reads and writes to values of that type. This allows Cogent to operate on the data-types directly, without requiring externally defined functions marshalling the data between different forms.

Unfortunately, this approach somewhat breaks down when multiple representations of a data structure needed to be used simultaneously. Each type is associated with exactly one layout, with a default layout being generated if none are specified, so types that have multiple layouts are inconvenient to express in Cogent. For instance, consider **??**. When dealing with 32bit colours, two commonly used encodings are Red-Green-Blue-Alpha (RGBA) and Alpha-Red-Green-Blue (ARGB). Both of these encodings encode their colour channels sequentially in 8bit values in the order determined by their respective names. These two formats only differ by the relative position of the alpha channel, yet would need different types to represent in Cogent since they have different layouts.

The first half of this project provides a comprehensive pen-and paper formalization of Dargent. This is required because there isn't any existing literature pinning Dargent's semantics or typing rules, the only existing definitive source on how it behaves is its implementation. This would be quite problematic when attempting to extend it, not only is it difficult to

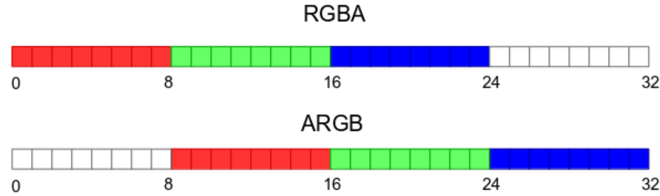

Figure 1.2: Bit layouts of ARGB and RGBA pixel format

gauge the intended semantics of a language from its implementation, but it is hard to determine what changes would break expected behaviour, and going forward what behaviours have been implicitly assumed by its users. To mitigate this, this formalization has been created.

1.4 Layout Polymorphism

Functions on these types may need to be duplicated by the programmer for each layout, and conversion routines may need to be manually created between these types. This causes a lot of unneeded complexity, and also unnecessarily increases verification costs.

The second half of this project proposes and implements an extension of Cogent's type system to allow functions to be polymorphic over layouts. By allowing data-types to have multiple layouts and functions to be polymorphic over layouts, we eliminate the need for these duplicate functions, and make the language more ergonomic overall.

Chapter 2

Background

2.1 Cogent

Cogent is a systems-level functional programming language, designed to be a good compromise between ease of verification and performance. It is, however, also a restricted language, being (purposefully) incapable of recursion, iteration and having no support for recursive types (such as lists or trees). This is because by not having any of these features, the language becomes much easier to reason about and verify — since recursion and iteration are impossible, all programs written in it are guaranteed to terminate, which greatly simplifies verification.

Cogent code operates on algebraic data-types. These data-types can easily map to a lot of systems data structures, with product types representing structs, sum types representing tagged unions, and boxed types representing heap allocated values.

The example code in 2.1 shows off many of the features of Cogent. The

```
type Opt a = <None () | Some a>
type LookupTable
type ContainedValue = Bool
type Entry = <Value U16 | Empty ContainedValue>

getEntry: (LookupTable, U32) -> Entry

getBoundedValue: (LookupTable, U32, U16) -> Opt U16
getBoundedValue (lookUpTable, key, bound) =
    let entry = getEntry (lookUpTable, key) in
        entry | Value x -> if x <= bound
                        then Some x
                        else None
              | Empty _ -> None
```

Figure 2.1: Example Cogent code

example program defines a function called getBoundedValue. This function takes in a lookup table (which is an externally defined data-type, implemented in C), an unsigned 32bit key value and an unsigned 16bit threshold. When executed, it uses the externally defined C function getEntry to retrieve the entry for that key, which is either a 16bit unsigned value, or a boolean value representing whether or not that slot contained a value in the past. If it resolves to a value that is not greater than the threshold, the function returns the value, otherwise it return nothing.

2.1.1 Uniqueness typing

A key issue with purely functional languages in the systems world is that values can not be updated in place without violating referential transparency. This is because all values have an unknown number of outstanding refer-

ences to them, so they cannot be modified without potentially effecting other computations.

This means that most functional languages implement updates by allocating new boxed values, and unfortunately tracking these allocations usually requires some method of garbage collection. Garbage collection is undesired in systems environments however, as it can have a noticeable performance overhead, so purely functional languages are usually not widely used in the systems space.

Uniqueness type systems, like the one in Cogent allow values to be mutated destructively[4]. This is because a uniqueness type system allows the compiler to static assert that all modifying accesses to a value can only occur when there are no outlying references to it, by ensuring this, we can ensure that we don't modify a value that is being used elsewhere. This allows us to achieve performance comparable to C without sacrificing our equational semantics, and to not require a garbage collector.

2.1.2 Dargent

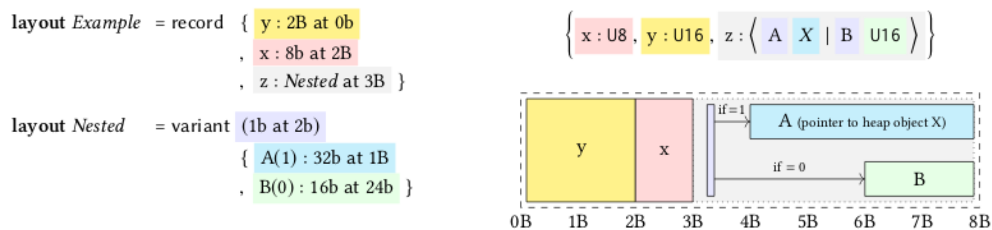

Figure 2.2: An example Dargent layout, with in-memory layout on right, image source: [1]

Dargent[3] is a proposed extension to Cogent that enables programmatic

prescription of the bit-level representation of Cogent algebraic data-types. The extension essentially defines a Domain Specific Language (DSL) that can be used to specify layouts, abstract mappings of variables to their concrete positions in a chunk of memory.

The example in Figure 2.2 shows the layout of the type **nested**, as specified by a Dargent layout. The type consists of an 8bit unsigned integer x, a 16bit unsigned integer y, and a tagged union consisting of either a reference to a value of type **X**, or an unsigned 16 bit integer. The tagged union is discriminated by a single specified bit.

2.2 Data Description Languages

PADS[5] is a Data Description Language (DDL) that allows programmatic specification of the physical layout of data sources. Given a schema of a data format, the PADS compiler can generate C libraries to facilitate parsers and serializers for files adhering to it. The language is motivated as being designed for dealing with ad-hoc data sources, which are commonly error prone and inconsistent. The language does seem to focus on text based formats, so it's less relatable to our work.

Datascript[6] is another DDL, but instead of generating a parser from a specification, it instead generates a Java class definition. It also generates a visitor type for the class. This seems somewhat close to what we want to implement, but unfortunately, a visitor pattern would not work particularly well in Cogent.

Unfortunately, neither of these works are very applicable to our project.

While both languages use methods that abstract away the layout of a datatype from its contents and semantics, neither use methods of abstracting over data sources that are particularly efficient or automated, so we cannot draw inspiration from either of them.

It seems that most research into DDLs focuses on approaches that use separate serialization, processing and de-serialization stages. The likely cause of this is that in most cases outside of formal verification, there is no real need to programmatically generate bindings to data structures, since most programs interface with internal data structures through well defined interfaces, and have no pressing need to validate these structures. From this, it becomes apparent that some degree of original work will be needed to design our layout polymorphism extension.

2.3 Intensional Type Analysis

Cogent has an unusual approach to polymorphism. The normal method used for polymorphism in functional languages is to use boxed pairs of values and method-dictionaries to represent values of the unknown type, specifying all required methods by constraining the type variables, and looking up these methods in the attached dictionary at run-time when they are needed. Cogent implements its polymorphism in a method more reminiscent of C++. What it does is that it instead determines statically all the concrete types that the polymorphic types resolve to through a method called monomorphisation, then it generates separate code paths for each of these concrete types.

Intensional type analysis[7] is a set of methods for formalizing polymorphic languages of this nature. The paper rigorously describes the translation process from a polymorphic variant of Mini-ML to a monomorphic language based on Girard's F_ω.

2.4 Formal Semantics for Data Description Languages

DDC [8] is a calculus designed to effectively model the core features of most DDLs. It was primarily designed around PADS, with minor inspiration from other languages like PACKETTYPES and DataScript. This calculus gives us a good reference for elaborating Cogent's type system with typing and inference rules for layouts.

Chapter 3

Completed Work

The goal of our project is to extend Dargent with layout polymorphism. This project had two components, a research component and an engineering component. The research component entailed determining how to extend the type inferencing algorithm and typing rules to allow types to be polymorphic over layouts. This was done through pen and paper methods, since a rigorous computer verifiable formalization is far beyond the scope of this project. The engineering component consists of the modifications to the Cogent compiler needed to implement layout polymorphism.

3.1 Methodology

3.1.1 Pen-and-Paper Formalization of Dargent Layout Polymorphism

The purpose of the formalization, in regards to the original goal of implementing layout polymorphism, is to give us a well defined foundation and

definition of how polymorphism over layouts would work. Since there is no existing literature on the subject of layout polymorphism, some careful thought was needed to ensure our approach wouldn't create soundness holes or bugs in the Cogent compiler. During the thesis, the scope of this component was extended massively to include formalizing Dargent as a whole, due to the lack of any definitive source on Dargent's behaviours and semantics. The formalization of Dargent ended up being a significant outcome of this thesis, with the formalization of layout polymorphism ending up relatively small in comparison.

3.1.2 Compiler augmentation

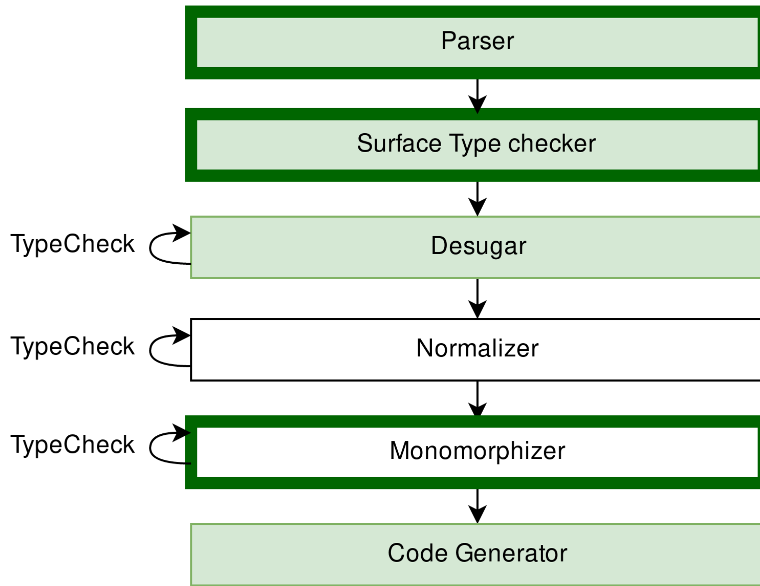

Figure 3.1: Cogent Compiler Pipeline

Cogent has two languages internally, the surface language (which is what the programmer writes), which contains all the nice features that Cogent

supports, and the core language, a stripped down variant of the language lacking most of the niceties of the surface language, but which as a result is much easier for the compiler to process. Most of the phases of the compiler pipeline from parsing up to monomorphisation (where all polymorphism related information is processed then erased) needed at least some modification to accommodate layout polymorphism.

In the above diagram, the phases of Cogent that will need to be modified have been outlined in dark green, and the phases that of Cogent that Dargent's implementation modified have been highlighted in light green.

Since types will now be representable through multiple distinct layouts, we now have to track pairings of types with layouts individually, instead of knowing globally which layout is associated with which type. Internally, the compiler marks types with *sigils* to track whether variables are writable and/or boxed. Our approach modified this sigils to track layout information, since they are propagated through many phases of the compiler pipeline. This coupled with some changes to the syntax, and the addition of a few additional constraints, was sufficient to implement a rudimentary version of layout polymorphism.

Chapter 4

Results: Formalization

The original plan was to have a simple formalization along with most of the implementation done by now. However, the formalization of layout polymorphism heavily depends on the formalization of the core Dargent language itself, which was non-existent at the offset of this project. As a result, the scope of this formalization has been greatly increased, to include formalizing the core Dargent language itself, along with the changes to Cogent's semantics needed for Dargent to operate. This chapter details the results of this formalization.

4.1 Formalization Strategy

This formalization was done in an entirely offline pen-and-paper manner; no proof code was written, so it isn't directly machine checkable, though it will be a great support for any future attempts to write a machine-checkable formalization. The formalization works by first specifying rules for checking

the well-formedness of layouts, then defining when it is legal to match a type with a layout, and finally specifying the constraints that need to be generated (and rules for solving them) to check that a Cogent program can associate a layout with a type. The formalization operates entirely on the core language, since most of the steps will occur in the core language.

We use the notation $\{\!\{x\}\!\}$ to denote multisets. For example, $\{\!\{a, a, b\}\!\}$ is defined as a multiset containing a twice and b once, and we use a modified set builder notation $\{\!\{a \mid a < 5 \wedge a \in \mathbb{N}\}\!\}$ to represent the numbers $0 - 4$, each with multiplicity 1.

Also, we use overlines to denote lists similar to the Cogent formalization, e.g. $\overline{(\mathbb{R}, \mathbb{R})}$ denotes a list of pairs of reals numbers.

4.2 Dargent Formalization

4.2.1 Abstract Grammar for Dargent

In this formalization, we refer to Dargent layouts as expressions following the abstract grammar defined in Figure 4.1.

We define the layouts recursively as being one of five types: sized blocks, offset layouts, record layouts and variant layouts. Sized blocks (**Sized** n) are defined entirely based off of their size, which is in bits. Offset layouts (**At** ℓ o) are essentially other layouts at a bit offset, and are parameterized by the inner layout and the offset. Record layouts (**LRecord** $\{\overline{(\mathrm{f}, \ell)}\}$) simply contain a list of fieldname-layout pairs. Similarly, variants (**LVariant** ℓ $\{\overline{(\mathsf{K}, t, \ell)}\}$) contain a list of tuples of variant constructor names, variant tag values (which are used by Cogent to discriminate the different variants), and lay-

Size	n	::=	$\mathbb{N}$
Offset	o	::=	$\mathbb{N}$
Tag	t	::=	$\mathbb{N}$
Variant Constructor Name	K	::=	Σ^*
Record Field Name	f	::=	Σ^*
Layout	ℓ	::=	$\textbf{Sized}\ n$
		|	$\textbf{At}\ \ell\ o$
		|	$\textbf{LRecord}\ \{\overline{(\mathrm{f}, \ell)}\}$
		|	$\textbf{LVariant}\ \ell\ \{\overline{(\mathsf{K}, t, \ell)}\}$
		|	α

Figure 4.1: Abstract grammar for Dargent

outs, along with a single layout for the tag itself. Finally, we include layout variables (α), which are type variables that only match against layouts, and are included to support layout polymorphism.

There are some simplifying generalizations made between the concrete and abstract grammars. In the concrete grammar, layouts are assigned names, and layouts can reference other names internally to allow more structured layout definitions (although self reference isn't allowed, since that would require recursive types). In the abstract grammar, we assume that each layout has been expanded, i.e. we assume that all references to layout names have been replaced by the layouts themselves, which removes the need for modelling names.

Also, in the concrete grammar, sizes and offsets can be represented in either byte-counts, bit-counts or an arbitrarily and unboundedly nested summation of the two. In the abstract grammar, we assume that all sizes and offsets are specified as bit-counts.

4.2.2 Memory reservation rules

We start by defining the memory reservation judgement (denoted $\ell \asymp s$), which associates each valid layout ℓ with the set of bits s that layout would use (as bit offsets from the origin).

This also gives us a trivial way of determining if a layout is well-formed, since a layout ℓ is well-formed iff there exists some s such that $\ell \asymp s$. This is denoted by the shorthand 'ℓ **Ok**'.

$$\boxed{\ell \asymp s} \qquad \ell\ \textsf{\textbf{Ok}} \stackrel{\text{def}}{=} \exists\, s\,.\,\ell \asymp s$$

$$\frac{}{\textbf{Sized}\ n \asymp \mathbb{N}_{<n}}\,\mathbf{I_{Sized}} \qquad \frac{\textbf{At}\ \ell\ o \qquad \ell \asymp s}{\textbf{At}\ \ell\ o \asymp \{x + o \mid x \in s\}}\,\mathbf{I_{At}}$$

$$\frac{\begin{array}{c} S = \{\!\{ s \mid \ell \asymp s \wedge (\mathrm{f}, \ell) \in M \}\!\} \\ \text{for each } (\mathrm{f}, \ell) \in M.\, \ell\ \textsf{\textbf{Ok}} \qquad S\ \textsf{\textbf{Disjoint}} \qquad \{\!\{ \mathrm{f} \mid (\mathrm{f}, \ell) \in M \}\!\}\ \textsf{\textbf{Disjoint}} \end{array}}{\textbf{LRecord}\ M \asymp \{x \mid x \in s \wedge s \in S\}}\,\mathbf{I_{LRecord}}$$

$$\frac{\begin{array}{c} s = \{x \mid x \in s \wedge \ell \asymp s \wedge (\mathsf{K}, t, \ell) \in V\} \qquad \ell' \asymp s' \\ s \cap s' = \emptyset \qquad \text{for each } (\mathsf{K}, t, \ell) \in V.\, \ell\ \textsf{\textbf{Ok}} \qquad \{\!\{ \mathsf{K} \mid (\mathsf{K}, t, \ell) \in V \}\!\}\ \textsf{\textbf{Disjoint}} \\ \{\!\{ t \mid (\mathsf{K}, t, \ell) \in V \}\!\}\ \textsf{\textbf{Disjoint}} \qquad \ell' \stackrel{p}{\asymp} n \qquad \forall (\mathsf{K}, t, \ell) \in V.\, t < 2^n \end{array}}{\textbf{LVariant}\ \ell'\ V \asymp s \cup s'}\,\mathbf{I_{LVariant}}$$

Figure 4.2: Memory reservation rules for Dargent

We start with the base case, sized blocks (**Sized** n). A sized block of size n reserves the set of bits from 0 to $n - 1$ inclusive on both ends, as shown by the rule $\mathbf{I_{Sized}}$ in Figure 4.2.

For an offset layout (**At** ℓ o), we simply shift all the bits reserved by ℓ upwards by o bits, as shown by the rule $\mathbf{I_{At}}$.

Record layouts (**LRecord** M) are a bit more complex. We first have to ensure that all inner layouts in M are well-formed, and that all field names are distinct. We then have to check that all inner layouts in M occupy pairwise disjoint sets of memory. If both of these conditions are true, then the layout occupies the union of the reserved memory of all of the inner layouts, as shown by the rule $\mathbf{I_{LRecord}}$.

Finally, variant layouts (**LVariant** ℓ' V) require three checks, as shown by the rule $\mathbf{I_{LVariant}}$. We first ensure that none of the inner layouts $((_, _, \ell) \in V)$ overlap with the tag layout (ℓ'). Then we ensure that all variant constructor names (K) and tag values (t) in V are disjoint. Finally, we then check that every tag value (t) can fit in the tag layout (ℓ').

Note that we do not include a rule for the fifth layout type: layout variables. This is because a layout variable represents an unknown layout, which we can't match until we have unified it with a concrete layout.

4.2.3 How Dargent stores layouts

In Cogent, every abstract type and record type contains a sigil (as defined in Figure 4.3), which is a small tag specifying whether the type is unboxed, writable or read-only. These sigils are used to allow multiple immutable temporary references to boxed record types, which can only be facilitated by allowing the sharability of said type to be temporarily changed.

Dargent only permits boxed records (and their contents) to have a specified layout. This is sufficient since Cogent's memory model only permits custom layouts for values on the heap, which is currently only possible through a boxed record. As a result, Dargent stores data layouts in the sigils of the

$$
\begin{array}{lcll}
\text{Sigil} & s & ::= & \text{ⓤ} \\
 & & | & \text{ⓦ} \\
 & & | & \text{ⓡ} \\
 & & | & \alpha
\end{array}
$$

Figure 4.3: Cogent definition of sigils

boxed record types they are applied to.

This differs from Cogent's definition of sigils, so we use a new definition for sigils (Figure 4.4). The only difference here is that we use a subscript on the sigil to specify the associated layout. We notate the compiler's default layout for a type using ℓ_0.

$$
\begin{array}{lcll}
\text{Sigil} & s & ::= & \text{ⓤ} \\
 & & | & \text{ⓦ}_\ell \\
 & & | & \text{ⓡ}_\ell \\
 & & | & \alpha
\end{array}
$$

Figure 4.4: Dargent definition of sigils

This provides us with a bit of a problem though. The types that we wish to match must contain the layouts we are trying to match them against. This means that we no longer have a matching relation between the set of types and the set of layouts, instead we have a well-formedness predicate on types (to make sure they match their contained layouts). We also now have the issue that all the rules which reference sigils in the old manner need to be modified slightly, this will be elaborated in later sections of this document.

4.2.4 Rules for matching types and layout

Now that we have a rule for the well-formedness of a layout, we can define rules for matching types with layouts. We now define the judgement 'matches' ($\tau \succ \ell$), which determines if the layout ℓ can legally match the type τ. This relation is not defined for boxed records. We then define our well-formedness predicate on types, which recursively checks types to ensure that all boxed records match their layouts.

To help with this, we first define two helper judgements (Figure 4.5). $\ell \overset{p}{\asymp} n$ determines the size of a primitive layout (i.e. a layout that consists of a single contiguous chunk of memory) in bits.

$$\boxed{\ell \overset{p}{\asymp} n}$$

$$(\mathbf{Sized}\ n) \overset{p}{\asymp} n \mathbf{P_{Sized}} \qquad \frac{\ell \overset{p}{\asymp} n}{(\mathbf{At}\ \ell\ o) \overset{p}{\asymp} n}\mathbf{P_{At}}$$

Figure 4.5: Memory reservation rules for primitive types

From those two primitive rules, we can determine if a primitive type matches a primitive layout through $\mathbf{F_{Prim}}$ (Figure 4.6), which is the base case for the 'matches' judgement. To do this, we simply check if the size the primitive layout can fit is greater than the required size for the primitive type.

From this, we can inductively check composite types. To start, we first state that if a τ matches ℓ, then τ matches ℓ at any arbitrary offset. This

$$\boxed{\tau \succ \ell}$$

$$\frac{\ell \overset{p}{\succeq} n \qquad n \geq \text{primSize}(\tau)}{\tau \succ \ell}\mathbf{F_{Prim}}$$

$$\frac{\tau \succ \ell}{\tau \succ \mathbf{At}\ \ell\ o}\mathbf{F_{At}} \qquad \frac{\ell \overset{p}{\succeq} n \qquad n \geq \mathit{PtrSize}}{\{\overline{\mathrm{f} : \tau}\}\ s_{\ell'} \succ \ell}\mathbf{F_{BRecord}}$$

$$\frac{\text{for each } i, \exists(\mathrm{f}_i, \ell) \in M.\, \tau_i \succ \ell \qquad (\mathbf{LRecord}\ M)\ \textsf{Ok}}{\{\overline{\mathrm{f}_i : \tau_i}\}\ ⓤ \succ \mathbf{LRecord}\ M}\mathbf{F_{Record}}$$

$$\frac{\text{for each } i, \exists(\mathsf{K}_i, t, \ell) \in V.\, \tau_i \succ \ell \qquad (\mathbf{LVariant}\ \ell'\ V)\ \textsf{Ok}}{\langle\overline{\mathsf{K}_i\ \tau_i}\rangle \succ \mathbf{LVariant}\ \ell'\ V}\mathbf{F_{Variant}}$$

Figure 4.6: Memory reservation rules for compound types

may seem superflous due to $\mathbf{P_{At}}$, but it is needed, since records and variants can also be placed at offsets.

We then define the case for boxed records ($\{\overline{\mathrm{f} : \tau}\}\ s_\ell$). All interior boxed records are pointers, so we have to check whether ℓ can fit a pointer of the desired architecture. We also have to ensure that the boxed record type is well-formed i.e. that it matches its internal layout, but this is given by the recursive nature of the well-formedness judgement.

For unboxed records, we have to check both that the layout is well-formed and that each field in the type has a corresponding field in the layout that can match that field's type. We match variants essentially the exact same way.

Now, we can finally define our well-formedness rule for types (Figure 4.7).

There are a lot of loose well-formedness conditions for types from the formulation of Cogent, for our purposes we will only consider conditions relevant to layout matching. The rule is very simple, if we have a type that is not a boxed record, we know it is well-formed as long as all interior types (which are checked recursively) are well formed. If the type is a boxed record, then it must contain a layout that it must match against as if it were an unboxed record ($\mathbf{Ok_{BRecord}}$). Since our layout matching relation requires the well-formedness of all boxed records inside the type it matches, we know that all inner types are well-formed too.

$$\boxed{\tau\ \mathsf{Ok}}$$

$$\frac{}{\mathrm{T}\ \mathsf{Ok}}\mathbf{Ok_{Lit}} \qquad \frac{\tau\ \mathsf{Ok} \qquad \tau'\ \mathsf{Ok}}{\tau\ \rightarrow\ \tau'\ \mathsf{Ok}}\mathbf{Ok_{Fun}}$$

$$\frac{\text{for each } i.\,\tau_i\ \mathsf{Ok}}{\langle\overline{\mathsf{K} : \tau_i}\rangle\ \mathsf{Ok}}\mathbf{Ok_{Variant}} \qquad \frac{\text{for each } i.\,\tau_i\ \mathsf{Ok}}{\{\overline{\mathrm{f}_i : \tau_i}\}\ \textcircled{u}\ \mathsf{Ok}}\mathbf{Ok_{Record}}$$

$$\frac{\text{for each } i.\,\tau_i\ \mathsf{Ok} \qquad \{\overline{\mathrm{f}_i : \tau_i}\}\ \textcircled{u} \succ \ell}{\{\overline{\mathrm{f}_i : \tau_i}\}\ s_\ell\ \mathsf{Ok}}\mathbf{Ok_{BRecord}}$$

Figure 4.7: Well-formedness rules for types

4.2.5 Modified Cogent typing rules regarding sigils

As mentioned previously, we need to adapt a few of Cogent's typing rules to match our new definition for sigils. These fall into two major categories, those that destructure sigils and operate differently based on the (the rules regarding equality between sigils) and those that transform them (the rules

regarding the bang operator). We detail the changes needed for both of these categories. Rules that simply pass sigils through without modification or observation do not need modification. This includes sub-typing rules for records.

Bang operator

The bang operator is a type level operator that temporarily converts writable types into read-only types. Figure 4.8 lists all the normalization rules related to this rule that need to be modified to match our new definition.

$$\boxed{\tau \hookrightarrow \tau}$$

$$\begin{array}{lcl}
\mathbf{bang}\,(\mathrm{A}\ \overrightarrow{\tau_i}\ \text{ⓦ}) & \hookrightarrow & \mathrm{A}\ \overrightarrow{\mathbf{bang}\,(\tau_i)}\ \text{ⓡ} \\
\mathbf{bang}\,(\mathrm{A}\ \overrightarrow{\tau_i}\ \text{ⓡ}) & \hookrightarrow & \mathrm{A}\ \overrightarrow{\mathbf{bang}\,(\tau_i)}\ \text{ⓡ} \\
\mathbf{bang}\left(\{\overline{\mathrm{f}\,{}^{u}_{i} : \tau_i}\}\ \text{ⓦ}\right) & \hookrightarrow & \{\overline{\mathrm{f}\,{}^{u}_{i} : \mathbf{bang}\,(\tau_i)}\}\ \text{ⓡ} \\
\mathbf{bang}\left(\{\overline{\mathrm{f}\,{}^{u}_{i} : \tau_i}\}\ \text{ⓡ}\right) & \hookrightarrow & \{\overline{\mathrm{f}\,{}^{u}_{i} : \mathbf{bang}\,(\tau_i)}\}\ \text{ⓡ} \\
 & \ldots &
\end{array}$$

Figure 4.8: Some Cogent typing rules for the bang operator

The changes here are fairly straight forward, since we only want to make the type readable without modifying the layout. Hence, we simply note that sigils on both sides of the normalization rule require the same layout.

$$\boxed{\tau \hookrightarrow \tau}$$

$$
\begin{array}{lcl}
\textbf{bang}\,(\mathrm{A}\ \overrightarrow{\tau_i}\ ⓦ_\ell) & \hookrightarrow & \mathrm{A}\ \overrightarrow{\textbf{bang}\,(\tau_i)}\ ⓡ_\ell \\
\textbf{bang}\,(\mathrm{A}\ \overrightarrow{\tau_i}\ ⓡ_\ell) & \hookrightarrow & \mathrm{A}\ \overrightarrow{\textbf{bang}\,(\tau_i)}\ ⓡ_\ell \\
\textbf{bang}\left(\{\overline{\mathrm{f}\,{}^{u}_{i} : \tau_i}\}\ ⓦ_\ell\right) & \hookrightarrow & \{\overline{\mathrm{f}\,{}^{u}_{i} : \textbf{bang}\,(\tau_i)}\}\ ⓡ_\ell \\
\textbf{bang}\left(\{\overline{\mathrm{f}\,{}^{u}_{i} : \tau_i}\}\ ⓡ_\ell\right) & \hookrightarrow & \{\overline{\mathrm{f}\,{}^{u}_{i} : \textbf{bang}\,(\tau_i)}\}\ ⓡ_\ell \\
 & \dots &
\end{array}
$$

Figure 4.9: Corresponding Dargent typing rules for the bang operator

Equality between sigils

There are other rules that actually observe the sigils and have requirements of or different behaviours based on their values. It can be noted (through exhaustively checking each one) that none of these rules try to insert or construct new sigils into a type. In particularly, all of these rules only have constraints of the form $s \neq s'$ where s' is some literal sigil. Hence, it suffices to simply change our definition of this inequality to there being no layout we can inject into the literal s' (if needed) such that $s = s'$.

4.2.6 Constraint generation/solving rules

In the existing implementation of Dargent, we don't need to generate or solve any constraints when checking types. All type checks regarding layouts are done on concrete types and layouts layouts and types, and as a result, in core Dargent, we don't need our well-formedness rule to itself be a constraint.

This is no longer the case after we include layout polymorphism, however.

4.3 Layout Polymorphism formalization

Now that we have a well defined basis for reasoning about Dargent, we can start defining how layout polymorphism will work. The general idea is that we will make our well-formedness rule a constraint, and allow users to guard their polymorphic functions with these constraints. We also allow layout variables to appear in the quantifiers of these functions. This should allow our functions to be polymorphic over layouts. The issue in doing this is that we have to determine rules for solving these constraints, determine where these well-formedness constraints need to be generated, and modify our type application rule to handle these changed rules.

4.3.1 Constraints

We define a new constraint for our well-formedness rule.

$$\begin{array}{lcll} \text{constraint} & C & ::= & \tau\ \textsf{\textbf{Consistent}} \\ & & | & \ldots \end{array}$$

Figure 4.10: Modified constraints needed for layout polymorphism

In the actual Dargent concrete grammar, this constraint looks a bit different. It appears as a constraint of the form `t :~ l`, where `t` has to be a boxed record type with the compiler default layout (that may contain type variables) and `l` has to be a layout variable. This can trivially be converted into our formalized version of the constraint. Unfortunately, more complicated constraints (such as those containing a type variable on the left, or a concrete layout on the right) cannot be handled at this time, as they can re-

sult in some degenerate matching cases which our model of Dargent cannot handle.

4.3.2 Constraint generation rules

Cogent's constraint generation rules roughly follow the schema: Given an algorithmic context G, attempting to type an expression e with type τ will generate a new algorithmic context G', and a set of new constraints C. We need to modify some constraint generation rules to both ensure that all types are consistent and we can instantiate layout-polymorphic functions.

We need to generate this constraint essentially for all types that are mentioned anywhere in a cogent program. Luckily, we can assure this by checking for well-formedness whenever a new type is created (in a way that might associate a layout), through the programmer's input. The only place where this can happen is in the type annotation rule.

$$\boxed{\mathsf{G} \vdash e : \tau \rightsquigarrow \mathsf{G}' \mid C}$$

$$\frac{\mathsf{G}_1 \vdash e : \tau' \rightsquigarrow G_2 \mid C}{\mathsf{G}_1 \vdash e :: \tau' : \tau \rightsquigarrow G_2 \mid C \ \wedge \ \tau' \ \sqsubseteq \ \tau \wedge \tau \ \textbf{Consistent}} \mathbf{CG_{Sig}}$$

$$\frac{\overrightarrow{\gamma_k}\ \mathsf{fresh} \qquad \mathsf{typeOf}(f) = \forall \overrightarrow{a_i}\, \overrightarrow{b_j}\, \overrightarrow{c_k}.\, C \Rightarrow \rho \qquad C' = C[\overrightarrow{\tau_i}/\overrightarrow{a_i}][\overrightarrow{\ell_j}/\overrightarrow{b_j}][\overrightarrow{\gamma_k}/\overrightarrow{c_k}]}{\mathsf{G} \vdash f[\overrightarrow{\tau_i}][\overrightarrow{\ell_j}] : \tau \rightsquigarrow \mathsf{G} \mid \rho[\overrightarrow{\tau_i}/\overrightarrow{a_i}][\overrightarrow{\ell_j}/\overrightarrow{b_j}][\overrightarrow{\gamma_k}/\overrightarrow{c_k}] \ \sqsubseteq \ \tau \wedge C'} \mathbf{CG\text{-}TL_{App}}$$

Figure 4.11: Constraint generation rule for layout polymorphism

These rules were changed as described in Figure 4.11. Whenever a type is explicitly provided through a type signature, said type signature must

be checked, which requires a change to $\mathbf{CG_{Sig}}$. We also have to modify our type application rule to also apply layouts, which requires a change to $\mathbf{CG\text{-}T_{App}}$, now called $\mathbf{CG\text{-}TL_{App}}$.

4.3.3 Constraint solving rules

We also need to add some rules to allow the constraint solver to solve the new type of constraint. Our well-formedness constraint doesn't interact with other constraints, only with the substitutions made through the constraint solving process. Since constraint solving is run to a fixed point, we can do all solving for this rule in the simplification stage. The solver rule we use is in Figure 4.12.

$$\boxed{C \overset{simp}{\hookrightarrow} C}$$

$$\frac{\overline{\tau_i\ \mathsf{Ok}} \vdash \tau\ \mathsf{Ok}}{\tau\ \mathsf{Consistent} \overset{simp}{\hookrightarrow} \overline{\tau_i\ \mathsf{Consistent}}}$$

Figure 4.12: Constraint solving rule for well-formedness rule

While this step seems non-trivial for the simplification phase, it is fairly straight-forward and easy to implement. All it does is convert a well-formedness constraint on a type into all the well-formedness constraints that it depends on, using our well-formedness rule, which we have already defined algorithmically.

Chapter 5

Conclusion

Whilst this project didn't achieve all of the goals in its original scope, it has nonetheless provided some valuable contributions. The pen and paper formalization provides a concrete definition of how Dargent should behave, which is arguably more important that polymorphism, since it provides value to all future developments regarding Dargent, whereas layout polymorphism is only truly useful in a handful of scenarios.

We also have a good idea of how layout polymorphism should behave in Dargent, while the rules detailed in this report aren't the most flexible, they are more than sufficient to handle most common usecases of layout polymorphism.

The implementation of Layout polymorphism was attempted with serious headway made, unfortunately however, it cannot be called complete. While the basic implementation was written, there were some elusive bugs that couldn't been ironed out in the time remaining after completing the Dargent formalization. The progress made will hopefully be merged into a

separate branch in Dargent, so that it can be fixed and completed later.

5.1 Future work

Our model of Layout polymorphism currently only support matching constraints for boxed records being matched against layout variables. This could be generalized to type variables with layout variables, type variables with concrete layouts, and even types containing variables with layouts containing variables. Implementing this would require some additional constraints to be introduced to Cogent's type system, particularly ones matching non-boxed types with layouts (using our $\asymp$ relation), and one specifying that a group of layouts don't overlap.

Finally, it might be worthwhile to support parametric layout synonyms, especially for types like iterators. This would be especially useful when combined with the previous proposed extension.

Bibliography

[1] Zilin Chen, Matt Di Meglio, Liam O'Connor, Partha Susaria, Christine Rizkallah, and Gabi Keller. A data layout description language for Cogent, January 2019. at PriSC.

[2] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. *SIGPLAN Not.*, 51(9):89–102, September 2016.

[3] Liam O'Connor, Zilin Chen, Partha Susaria, Christine Rizkallah, Gerwin Klein, and Gabi Keller. Bringing effortless refinement of data layouts to Cogent. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 134–149, Limassol, Cyprus, November 2018. Springer.

[4] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[5] Kathleen Fisher and Robert Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pages 295–304, New York, NY, USA, 2005. ACM.

[6] Godmar Back. DataScript- a specification and scripting language for binary data. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, pages 66–77, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[7] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM.

[8] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. *SIGPLAN Not.*, 41(1):2–15, January 2006.