COMP30013 Research Project: Cedar, A Data Layout Description Language for C

George Anderson

18th November 2024

1 Abstract

In systems programming, precise control over data layouts is crucial for building reliable, performant, and secure software. This report introduces Cedar, a data layout description language designed to extend C, providing fine-grained control over memory layouts of C data types. Cedar is inspired by concepts from Cogent and Dargent. Where Cogent is a verified compiler outputting verified C code and Dargent, implemented in Cogent, enables verified bit-level data manipulations.

Partially implemented as a transpiler written in Haskell, Cedar translates high-level layout specifications into C code. Semantic analyis and lexing are implemented with concrete plans for code generation. Future development would see the completion of the transpiler.

The possibility of integrating Cedar into the C programming workflow would empower systems programmers with the tools to precisely control data layouts, facilitating interoperability and optimization without departing from the familiarity of C.

2 Introduction

This report discusses useful tools for systems programmers. Where systems programmers are programmers who use low-level programming languages like C to write software that typically serves as host to many other programs. One common example of this software would be an operating system. Due to the nature of systems code, programmers need their code to be reliable, performant and secure. Sometimes these programmers need to be able to make fiddly adjustments to their code to dictate how the binary representation (1s and 0s) is structured.

I would like to introduce Cogent and Dargent. Cogent is a language that compiles into verified C code [7, 6]. The verification means that the code has been mathematically proven to behave according to a certain set of specifications. An excellent guarantee for systems programmers who need as much reliability as possible. Dargent is effectively an add-on to Cogent that enables programmers to perform bit twiddling operations that are verified to be correct.

Dargent is called a data layout description language, one that is implemented in the Cogent programming language. Being a data layout description language is independent from the verification properties provided by Dargent. To summarise, Cogent is designed to enable systems programmers to write verified C code quickly and easily as proof certificates are automatically generated upon compilation. The main drawback of Cogent is that, in the interest of facilitating verification, only a subset of C can be written in Cogent.

Dargent grants programmers fine-grained control of the memory layout of data structures. This is a necessity for systems programmers for two reasons: firstly when using a well-defined software or hardware interface it is required for data to be delivered exactly how the interface expects. In systems code, there is a frequent requirement to make structs with fields that have non-standard bit widths, see the following example of a CAN driver which requires a field to have 29bits.

```c
struct can_id {
    uint32_t id:29;
    uint32_t exide:1;
    uint32_t rtr:1;
    uint32_t err:1;
};
```

Secondly, for performance reasons, it may be necessary to pack bits tightly together or perform other manipulations of the data structures. Perhaps, this all could be done separately with marshalling code, but a data layout language allows the solution to be achieved at a high conceptual level, for simplicity and efficiency.

A key weakness of marshalling code in the case of Cogent is the litany of errors that normally accompany such code and the work required to manually prove its veracity. Dargent crucially removes the need for marshalling code and due to its integration with Cogent, does not require additional manual proofs. It is mentioned in Dargent: A Silver Bullet for Verified Data Layout Refinement that while Dargent was created as a solution for a problem in Cogent, the general concept is applicable elsewhere.

To implement Dargent in another language we will require a compiler, which takes as input surface syntax and then typically outputs machine code that can be executed to run a program. In our case, due to the nature of Dargent as an add-on, if you will, to existing languages, we will consider transpilers. Transpilers output high-level human readable code instead of machine code.

In my research, I seek to apply the concepts behind Dargent directly to C in what I will call Cedar (C data layout description language). This will be implemented through the construction of a transpiler from Cedar into C code. A pen-and-paper implementation of a semantic analyser for the transpiler was described by Jakob Schuster in 2023, a foundtaional piece of work for this project [8]. Cedar is used for granting the powerful tool of layout specification. This is exceedingly relevant in a language such as C, which is ubiquitous in systems programming. The key advantage of Dargent for C over Dargent is that it is implemented for the whole of C, rather than just a subset of C like Cogent, which I mentioned earlier. It is currently not in scope to consider verification, however this could very well be a future development.

3 Preliminaries

Knowledge of context-free grammar is assumed along with knowledge of basis mathematical notation.

Lists are represented using overlines $\overline{x}$ to represent an list of elements.

Code snippets from Haskell are used frequently through this report. As such an understanding of functional programming is not necessary, but would assist in one's understanding of certain concepts.

The github repository containing my work may be found here: https://github.com/honours-theses/george-undergraduate.

4 Cedar

4.1 Transpiler Overview

Cedar is a high-level programming language for specifying the low-level layout of data in a C program. A complete implementation would manifest as a transpiler. Cedar surface syntax would be written into a transpiler and upon compilation C code would be generated. The components of the transpiler that have been currently implemented are lexical and semantic analysis. Both of these have been written in Haskell, as such it would make sense for the rest of the transpiler to be written in Haskell as well.

As seen in Figure 1, there are 4 stages that comprise the transpiler. Each stage takes their input from the previous stage and provides their output to the next stage. In lexical analysis the source code (Cedar code) is separated into atomic tokens that represent the smallest meaningful units. Syntactical analysis, sometimes called parsing, takes these tokens and generates an abstract syntax tree (AST). Semantic analysis checks that meaning of the source code adheres to the rules of the language. It intakes the AST and either finds and reports errors or allows progression to the next stage. Code generation constructs the output C code from the AST, in principle the generated code should have the same meaning as the source code.

4.2 Surface Syntax

The surface syntax defines how the source code can be written. The additional grammar for Cedar is represented in Figure 2 below:

The layouts are recursively defined as any permutation of composite layouts: arrays, structs and unions, eventually leading to a primitive layout. Memory size is represented by the number of bytes and/or bits that an object requires. Bits are typically a single octal digit. Order represents when a field is set at a relative offset to another field in a struct,

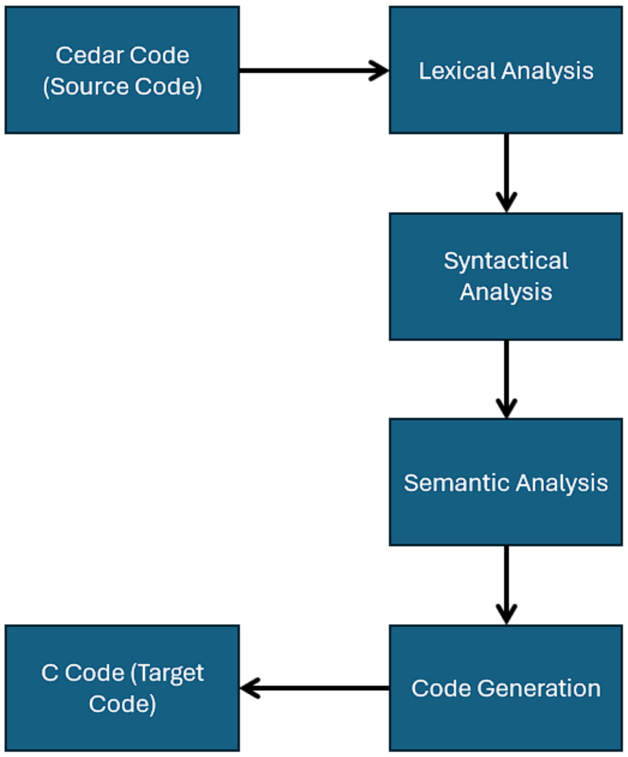

Figure 1: The Cedar Transpiler Pipeline

this can be either before or after by a memory size. This padding is often zero when we are trying to pack data tightly. Offset can either be specified absolutely by a memory size from the start of its current scope, or it can be set relative to another field, which can only occur inside of a struct. Endianess describes the orientation of the beginning of the data in binary representation. Typically this can be thought of as reading left to right (big endian) or right to left (little endian). Here big endian is BE and little endian is LE, ME is machine endian. This allows the compiler to decide on a default endian based on implementation context.

An example of surface syntax can be viewed in Figure 3. Here we can see the two new semantic cases that occur in Cedar, the definition of a layout and the association of a layout with a data structure. These two cases encapsulate the functionality added to C by the language Cedar. The semantic checking of them will be thoroughly addressed in subsection 4.3 Semantic Analysis. The keyword "layout" is an addition in Cedar, along with "after", "before" and "at" and others addressed in subsection 4.5 Lexing. Exactly how Cedar code is written syntactically is subject to change based on personal preference in implementation, unlike the grammar defined in Figure 2.

4.3 Semantic Analysis

The purpose of semantic analysis is to check the correctness of the code as rigorously as possible. It is not always possible to catch all errors, especially if the programmer has written code that is correct, but does not do what the programmer intends. The semantic analysis in

$$
\begin{array}{llll}
\text{Layouts} & \ell & ::= \mathsf{primitive}\,(\mathsf{mSize},\ \mathsf{ord},\ \omega) & \text{(primitive memory type)} \\
 & & \quad\mid\ \mathsf{array}\,(\mathsf{ord},\ \ell,\ n,\ \omega) & \text{(static array)} \\
 & & \quad\mid\ \mathsf{struct}\left(\mathsf{ord}, \overline{f : \ell}\right) & \text{(struct)} \\
 & & \quad\mid\ \mathsf{union}\left(\mathsf{ord}, \overline{f : \ell}\right) & \text{(tagged union)} \\
 & & & \\
\text{Memory Size} & \mathsf{mSize} & ::= \text{Byte } n \mid \text{Bit } n \mid \text{ByteBit } n\ n & \\
\text{Order} & \mathsf{ord} & ::= \text{Before } \mathsf{mSize} \mid \text{After } \mathsf{mSize} & \\
\text{Offset} & \delta & ::= \text{RelOffset } f\ \mathsf{ord} \mid \text{AbsOffset } \mathsf{mSize} & \\
\text{Endianness} & \omega & ::= \mathsf{BE} \mid \mathsf{LE} \mid \mathsf{ME} & \\
\text{Field names} & & \quad\ni\ f & \\
\text{Natural numbers} & & \quad\ni\ n &
\end{array}
$$

Figure 2: The syntax added in Cedar our layout language.

the Cedar transpiler is limited in scope to statements and expressions that involve layouts. This is more efficient as it delegates the activity to the C compiler.

Well-formedness is a concept that is mentioned often in this report, it refers to a set of rules that dictate the correct structure or behaviour of an object. Well-formedness is sometimes abbreviated to WF and it is seen explicitly in Figure 10 and Figure 13.

Semantic Analysis can be described by a pipeline visualised in Figure 4. Beginning with the surface syntax, the code is separated into that which describes data structures down the left-hand side into C and that which describes layouts down the right-hand side into L. The left-hand side begins in C after lexing and parsing takes place. C closely resembles the surface syntax, therefore it is reduced to CR which is the core data structure description. In this reduction anything that is not related to the memory size of the data structure is removed to simplify subsequent calculations and matching relations.

The right-hand side is a little bit more involved, it begins in L after lexing and parsing. L also closely mirrors the surface syntax, fields may still be represented as relative offsets so we are unable to make certain concrete calculations that would occur in LR. With the information that is available we are able to make an initial WF check. Contigent on the success of the WF check, the layout in L is reduced to a layout in LR. This primarily involves deriving the relative offsets present in L. Now inside of LR we are able to perform the final WF check based on the concrete memory information.

Once both the left and right side reach the matching relation stage, they are compared for compatibility. We need to confirm that the data structure from CR can be represented by the layout from LR. At any point in this pipeline the code could error out or succeed. Assuming the matching relation succeeds, we can then proceed to code generation, the implementation of which was out of scope for this report.

The representation of the C language found in the C stage of the pipeline was largely

```
// Layout Definition
layout Student_l = {
    name: 1B after grade;
    age: 1B before name;
    grade: {
        maths: 1B at 0B
        physics: 2B after maths;
    } at 0B;
};

// Associating a Layout with a Data Structure
typedef struct {
    char *name;
    uint8_t age;

    struct grade {
        uint8_t maths;
        uint16_t physics;
    } grade layout Student_l;
} Student;
```

Figure 3: Cedar Surface Syntax: C struct and valid layout.

informed by the CompCert project [5, 4]. The project was written in Coq, which is a formal proof management system and interactive theorem prover. Coq does share similarities with Haskell, as such it is possible to see the relation between CompCert in Figure 5 and Cedar in Figure 6. Notably we have changed the untagged unions into tagged unions in the interest of more efficient data structure usage.

As mentioned earlier CR (Figure 7) is simply a stripped down representation of C data structures with all non-memory related information removed. The ability to represent certain types such as functions as pointers allows for significant simplification. An additional module was utilised for ease of creation of the semantic analysis pipeline, BasicType. All of the primitive data types are in BasicType. It also has a configuration that allows for pointer size and long double size to be set as a compiler flag.

Simply put, Figure 7 shows us the code for the C to CR reduction. It reduces C to a core representation to make the later matching easier along with other operations.

Figure 9 shows us the Haskell definition of L. This shallow layout is a precursor to the core layout. It retains all of the detail of the surface syntax, but would be clunky to make core operations with. The similarity of this figure to Figure 2 should be noted.

In L there are also a number of initial well-formedness checks performed seen in Figure 10. Primarily infinite looping within relative offset references is detected. This would occur after either direct or indirect recursive referencing. A simple example of this would be when field 1 is defined to be after field 2 and field 2 is defined to be after field 1.

There other error checks implemented, including ArrayNonPositiveLength, which occurs when an array is defined with a non-positive length, SelfReferentialRelativeOffset, where a

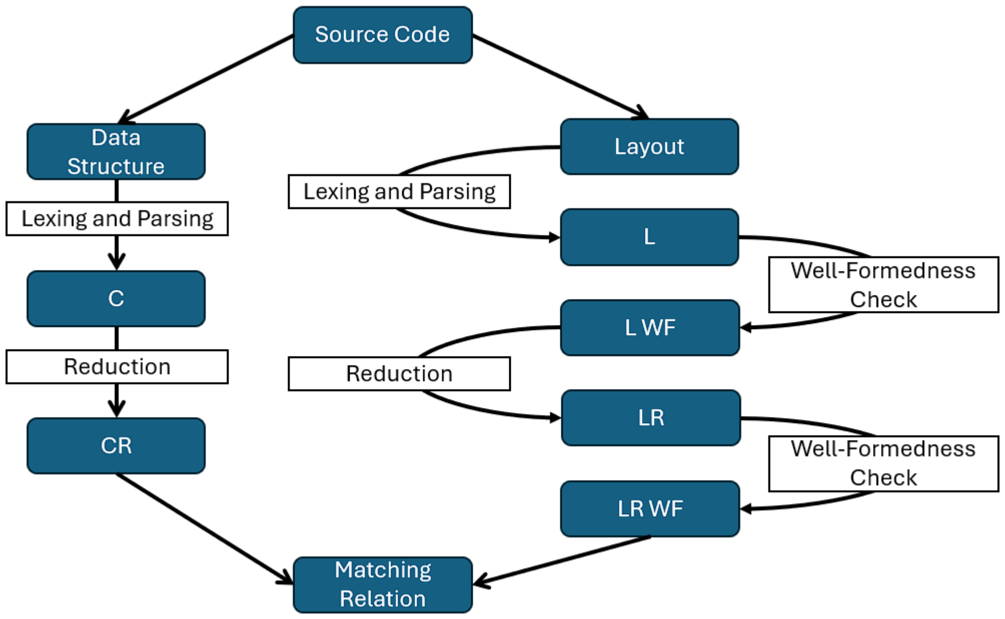

Figure 4: The Semantic Analysis Pipeline

relative offset refers to itself, FieldReferenceNotInScope, which arises when a field reference is not within the valid scope, NonUniqueFieldNames, where field names are not unique within a structure, NonStructRelativeOffset, indicating a relative offset applied to a non-struct entity, and EmptyCompositeLayout, which occurs when a composite layout has no defined fields or elements.

We can see that the definition of LR (Figure 11) is similar to that of L. The main difference lies in the fact that all of the values in the LR are concrete and therefore can be checked more thoroughly for correctness. The bits and bytes in L have been converted to bits and the relative offsets have been derived in the reduction (Figure 12).

The well-formedness checks in LR (Figure 13) comprise detecting all of the errors: NegativeOffset, NegativeSizeAllocation, StructOverlappingFields.

Finally the data structure and the layout are matched in the matching relation (Figure 14). The main consideration is the size of the layout, each primitive layout needs to be sufficiently large enough to carry the respective primitive data type. The layout and the data structure are matched by field name, mismatching field names will immediately error.

4.4 Code Generation

As stated earlier the transpiler intakes Cedar code which is a superset of C code, including sugared syntax that allows efficient writing of data layout description. It then outputs plain C code, crucially the C preprocessor has not been run on this and it is intelligible code that can be understood and altered as needed by systems programmers.

Focussing on the implementation of code generation, it will be trivial for the subset of Cedar that does not involve data layout specification as it can be output exactly the same as

CompCert C type definition:

```
Inductive type : Type :=
| Tvoid: type (* the void type *)
| Tint: intsize → signedness → attr → type (* integer types *)
| Tlong: signedness → attr → type (* 64-bit integer types *)
| Tfloat: floatsize → attr → type (* floating-point types *)
| Tpointer: type → attr → type (* pointer types (*ty) *)
| Tarray: type → Z → attr → type (* array types (ty[len]) *)
| Tfunction: typelist → type → calling_convention → type (* function types *)
| Tstruct: ident → attr → type (* struct types *)
| Tunion: ident → attr → type (* union types *)
with typelist : Type :=
| Tnil: typelist
| Tcons: type → typelist → typelist.
```

Figure 5: C Data Structure Haskell Code

it was input.

For the subset of Cedar that does involve data layout description, the data layouts may be output into C code as bit arrays with getters and setters. C does not natively support bit arrays but it is possible to artificially create them. It has been done before and instructions are widely avaiable to do so.

– bit array

4.5 Lexing and Parsing

Lexing and parsing entail a process of organising the surface syntax into a format that can be analysed and then manipluated to generate code.

The lexer has mostly been implemented in Haskell utilising the library Alex in accordance with the C standard ISO/IEC 9899:1999, Chapter 6 [3]. Additions have been made to the lexer that include Cedar keywords for layout description and memory size. This report will not go into detail about lexing since it is secondary in complexity and import to both semantic analysis and code generation. The lexer is available with the rest of the code referred to in this report and is well documented with the chapter and sections from the C standard specifically referenced throughout.

Parsing has not been implemented, but to integrate well with the existing of the transpiler components one would write it in Haskell with the Happy library. In a similar vein to lexing it is secondary to the main transpiler components.

C Data Structure:

```haskell
type FieldName = String
type TypeVar = String

data CType =
    Void
    | Int IntSize Signedness Attr
    | Long Signedness Attr
    | Float FloatSize Attr
    | Pointer CType Attr
    | Array CType Int Attr
    | Function [CType] CType CallingConvention
    | Struct [(FieldName, CType)]    -- differs to CompCert
    | Union [(FieldName, CType)]     -- differs to CompCert (C has untagged Unions)
```

Figure 6: C Data Structure Haskell Code

Basic Memory Type:

```haskell
data BasicMemType = Pointer | IBool | I8 | I16 | I32 | I64 | F32
    | F64 | LongDouble
```

CR Reduced Data Structure:

```haskell
import qualified Cedar.Semantic.BasicType as BMT

type FieldName = String
type Length = Int

data CRType =
    BasicMemType BMT.BasicMemType
    | Struct [(FieldName, CRType)]
    | Union [(FieldName, CRType)]
    | Array CRType Length
```

Figure 7: CR Reduced Data Structure and Basic Memory Type Haskell Code

The C to CR Reduction Function:

```haskell
import qualified Cedar.Semantic.C as C
import qualified Cedar.Semantic.CR as CR
import qualified Cedar.Semantic.BasicType as BMT

-- reduction from C to CR
reduction :: C.CType → CR.CRType
```

Figure 8: The C to CR reduction

The C to CR Reduction Function:

```haskell
data MemorySize = Byte Int | Bit Int | ByteBit Int Int
data Endianess = BE | LE | ME
data Order = After MemorySize | Before MemorySize
data Offset = RelOffset FieldName Order | AbsOffset MemorySize
type FieldName = String
type Length = Int

data Layout = Primitive MemorySize Offset Endianess
    | Array Offset Layout Length Endianess
    | Struct Offset [(FieldName, Layout)] Endianess
    | Union Offset [(FieldName, Layout)] Endianess
```

Figure 9: The C to CR reduction

The C to CR Reduction Function:

```haskell
-- Errors
data RawError =
    ArrayNonPositiveLength
    | RelativeOffsetInfiniteLoop
    | SelfReferentialRelativeOffset
    | FieldReferenceNotInScope
    | NonUniqueFieldNames
    | NonStructRelativeOffset
    | EmptyCompositeLayout
-- Result can accumulate multiple errors
data Result = NoError | Errors [RawError]

-- Main well-formedness function
wf :: Layout → Result
```

Figure 10: The C to CR reduction

The C to CR Reduction Function:

```

type MemorySize = Int -- Size in bits
type Offset = MemorySize
type Size = MemorySize
data Range = Range Offset Size
type FieldName = String
type Length = Int
data Endianess = BE | LE | ME -- ME, endianess of the target machine

data Layout = Primitive Range Endianess
    | Array Offset Layout Length Endianess
    | Struct Offset [(FieldName, Layout)] Endianess
    | Union Offset [(FieldName, Layout)] Endianess
```

Figure 11: The C to CR reduction

The C to CR Reduction Function:

```
-- | The main reduction function: converts L.Layout to LR.Layout
reduction :: L.Layout → LR.Layout
```

Figure 12: The C to CR reduction

The C to CR Reduction Function:

```
data RawError =
    NegativeOffset
    | NegativeSizeAllocation -- Could have a zero size union field (Just Nothing)
    | StructOverlappingFields

-- Result can accumulate multiple errors
data Result = NoError | Errors [RawError]
-- Well-formedness check with error reporting:

wf :: Layout → Result
```

Figure 13: The C to CR reduction

The C to CR Reduction Function:

```haskell
import qualified Cedar.Semantic.CR as CR
import qualified Cedar.Semantic.LR as LR
import qualified Cedar.Semantic.BasicType as BMT

-- | Matching relation: Determines if a CRType matches an LR Layout
matching :: BMT.Config → CR.CRType → LR.Layout → Bool
```

Figure 14: The C to CR reduction

5 Discussion

5.1 Testing

Various unit tests have been written to support the efficacy of the semantic analyser. The test cases themselves along with the error messages were incredibly helpful in diagnosing problems with the semantic analyser through the project.

5.2 Decisions

The main notable decision I had to make in the project was deciding between implementing a transpiler and implementing a compiler. I favoured the transpiler due to its ability to allow Cedar to integrate into legacy code, a significant issue considered C's ubiquity and age. It also would allow systems programmers to trust the generated code more, seeing as they would be able to understand and modify it themselves.

5.3 Future Work

There are a number of considerations for future work. Support of the C preprocessor would greatly increase the applicability and efficiency of Cedar. We have contemplated the result of runnning the preprocessor on the Cedar code before lexing and parsing. Aside from the work required to extend the C preprocessor to Cedar, we would also lose some of the abstraction by deriving everything. One idea is to run the preprocessor, perform everything up to semantic analysis, then restart with a copy of the original Cedar code and skip semantic analysis this time, assuming it succeeded on the first pass (Figure 15).

For code generation, tagged unions would need to be created artificially in C since they do not exist natively. Custom size ints would be incredibly useful to allow for more specific description of layouts. One may consider adding more operations to alter the layouts, currently we are able to manipulate offset and not much more. In future verification efforts could be made to increase the utility of Cedar and allow it to perform a role more similar to Dargent. Finally, simple quality of life improvements — adding more sugared syntax, speeding up the time taken to write code in Cedar would always be a good idea.

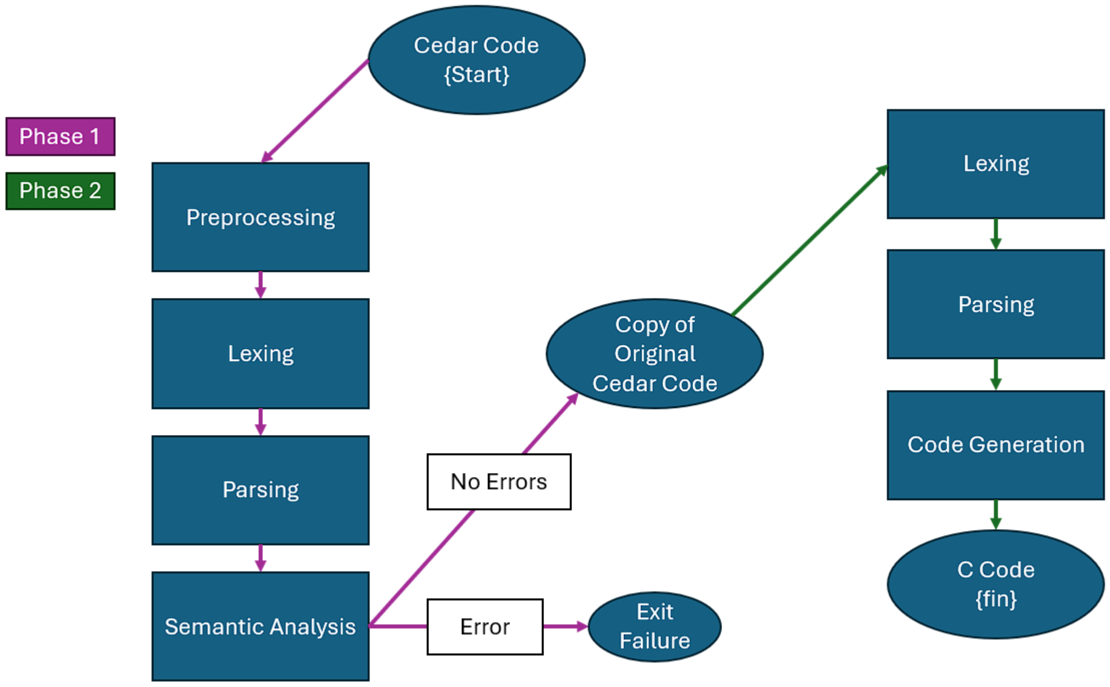

Figure 15: Another Cedar Transpiler Pipeline

6 Conclusion

This report introduced Cedar, a data layout description language designed to enhance the C programming language for systems programmers. Cedar empowers developers with precise control over memory layouts, enabling the creation of complex data structures in C. By building upon concepts from Cogent and Dargent—but applying them to the entire C language—Cedar addresses the need for fine-grained memory management without the constraints of a limited subset of C.

Implemented transpiler in Haskell, Cedar translates high level layout specifications into C code. The semantic analysis component ensures that layouts are well-formed and compatible with their corresponding data structures, reducing errors and increasing reliability. By eliminating the need for cumbersome marshalling code, Cedar simplifies interactions with hardware and low-level software interfaces that demand exact data representations.

While formal verification is not within the current scope, Cedar lays a solid foundation for future enhancements in this area. Potential developments include integrating the C preprocessor, code generation and capabilities to support features like tagged unions and custom-sized integers.

Acknowledgements

I would like to give a huge thanks Christine Rizkallah, my supervisor, for her guidance, support and patience throughout the semester. I was incredibly lucky to have her give me the opportunity to undertake this research.

7 Related Work

The field of data description that this report builds upon is densely populated with various languages [2, 11, 10, 9]. According to Zilin Chen et al.[1] the academic domain of describing low-level data layouts with high-level languages can be divided into two groups. program synthesis allows you to write specifically about low-level data representation through a high-level language, this allows you to write the specification quicker, more intuitively, it is made easier to verify and can be modified easily. program abstraction allows you to write code at a high-level while designating handling of the low-level representation entirely to the compiler. You may treat two different low level representations of the same high-level data structures the same which carries all of the benefits of abstraction — as mentioned in program synthesis.

Cedar would fall into the category of program abstraction. Allowing you to performance operations with a high-level data structure, without needing to directly consider the layout of the data. Effectively disentangling the data structure and memory layout. Of all of the languages Cedar is most similar to Dargent, by design [1]. Cedar separates itself from Dargent by its implementation in C, C is far more widely used than Cogent and it will also able to write the whole of the C language, not be limited to a subset of it.

In an undergradute report by Jakob Schuster, the foundations are set for the implementation of a semantic analyser for Cedar [8]. It is important to acknowledge the involvement of

Jakob's work in the implementation of the semantic analyser.

CompCert is a reputable project in the domain verified compilers [4]. In recognition of its contributions, CompCert received the 2021 ACM Software System Award. As such it was referred to for the implementation of C surface syntax and the C types.

References

[1] Zilin Chen et al. "Dargent: A Silver Bullet for Verified Data Layout Refinement". In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: 10.1145/3571240. URL: https://doi.org/10.1145/3571240.

[2] Marcell van Geest and Wouter Swierstra. "Generic Packet Descriptions: Verified Parsing and Pretty Printing of Low-Level Data". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development.* TyDe 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 30–40. DOI: 10.1145/3122975.3122979. URL: https://doi.org/10.1145/3122975.3122979.

[3] "International standard ISO / IEC 9899 Programming languages C - reference number ISO/IEC 9899:1999(E), Second Edition 1999-12-01". In: 1999. URL: https://api.semanticscholar.org/CorpusID:197662207.

[4] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115. URL: http://xavierleroy.org/publi/compcert-CACM.pdf.

[5] Xavier Leroy. *The CompCert C Verified Compiler: Documentation and User's Manual.* Intern report. hal-01091802v6. Inria, 2018, pp. 1–77.

[6] Liam O'Connor et al. *COGENT: Certified Compilation for a Functional Systems Language.* 2016. arXiv: 1601.05520 [cs.PL]. URL: https://arxiv.org/abs/1601.05520.

[7] LIAM O'CONNOR et al. "Cogent: uniqueness types and certifying compilation". In: *Journal of Functional Programming* 31 (2021), e25. DOI: 10.1017/S095679682100023X.

[8] Jakob Schuster. "Designing Data Layout Specification Languages". Unpublished Report. 2023.

[9] Michael Vollmer et al. "LoCal: A Language for Programs Operating on Serialized Data". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 48–62. DOI: 10.1145/3314221.3314631. URL: https://doi.org/10.1145/3314221.3314631.

[10] Yan Wang and Verónica Gaspes. "An Embedded Language for Programming Protocol Stacks in Embedded Systems". In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.* PEPM '11. Austin, TX, USA: Association for Computing Machinery, 2011, pp. 63–72. DOI: 10.1145/1929501.1929511. URL: https://doi.org/10.1145/1929501.1929511.

[11] Qianchuan Ye and Benjamin Delaware. "A Verified Protocol Buffer Compiler". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs.* CPP 2019. Cascais, Portugal: Association for Computing Machinery, 2019, pp. 222–233. DOI: 10.1145/3293880.3294105. URL: https://doi.org/10.1145/3293880.3294105.