



**School of Computer Science and Engineering**

**Faculty of Engineering**

**The University of New South Wales**

# **Recursive Types For Cogent**

by

**Emmet Murray**

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: August 7, 2019

Student ID: z5059840

Supervisor: Christine Rizkallah

# Abstract

Cogent is a linearly-typed functional programming language written in Haskell for writing trustworthy and efficient systems code. It has a certifying compiler that produces C code, a shallow embedding in the Isabelle/HOL theorem prover, and an Isabelle/HOL proof that the C code refines the shallow embedding. Cogent's linear type system ensures desired properties such as memory-safety and it also allows the compiler to generate efficient C code.

Cogent currently has no support for recursion; instead it has a foreign function interface (FFI) to C. Data-types, as well as iterators over these types, are implemented in C and called by Cogent code using the FFI.

The main task of this project is to add a restricted form of recursion to Cogent while keeping the language total. These recursive types would enable defining more data-types as well as iterators over these data-types directly in Cogent rather than resorting to C.

# Contents

|          |                            |          |
|----------|----------------------------|----------|
| <b>1</b> | <b>Project Description</b> | <b>1</b> |
| <b>2</b> | <b>Progress Report</b>     | <b>3</b> |
| 2.1      | Parser and Lexer . . . . . | 4        |
| 2.2      | Reorganiser . . . . .      | 5        |
| 2.3      | Type Checker . . . . .     | 5        |
| 2.4      | Testing . . . . .          | 5        |
| 2.5      | Work efficiency . . . . .  | 5        |
| <b>3</b> | <b>Reflection</b>          | <b>6</b> |
| <b>4</b> | <b>Revised Plan</b>        | <b>7</b> |

# Chapter 1

## Project Description

**TODO: Revise this**

Formal verification is the field of computer science that explores the methods that allow us to reason rigorously about the functional correctness of programs we write. One of the benefits of doing so is a proof of correctness for programs with respect to a specification, and which for well specified programs aids in eliminating bugs and unexpected program behaviour. Much effort has specifically been put into the verification of low level systems code which is critical to the operation of any computer. The presence of bugs in such a system can lead to security vulnerabilities, system crashes and invalid system behaviour, which for mission critical systems is unacceptable and for everyday use causes frustration for end users.

Using C to implement this code is a very popular choice in the systems community, and there have been many attempts to verify systems code written in C using tools such as AutoCorres [1], which takes parsed C code and produces a *shallow embedding* inside the theorem prover Isabelle/HOL [2]. This embedding is a representation of the semantics of the C code within the theorem prover, however due to the nature of the C language many difficulties arise when trying to reason about its functional properties, due to its lack of memory and type safety and its mutable state.

Cogent [3], is a domain specific language that was introduced to replace C as a systems implementation language. It is a functional, high level language with uniqueness types and a certifying compiler that produces a shallow embedding in Isabelle/HOL as well as low level efficient C code; the semantics of which correspond to the the Isabelle embedding. Due to the functional, high level nature of the embedding which is

designed to be reasoned about equationally as well as its resemblance to higher order logic, Cogent allows for a much less taxing process of verifying low level systems code.

Cogent is also suitable for low level systems development in contrast to many existing functional languages which operate on layers of abstractions away from the system, as its uniqueness types allow for both efficient destructive updates as well as static memory allocation. In addition to the benefits of uniqueness types, Cogent presents a C foreign function interface (FFI) allowing existing C programs to interact with Cogent code, without forcing teams to abandon a project already written in C and already verified.

However, Cogent currently has no support for recursion or iteration. Currently, any data type that can be iterated over and its iterators have to be defined externally in C, and included in the Cogent program via its Cogent's C FFI. Proving totality about existing code is a guarantee that a system will not hang, or deny services to other systems, which is a great benefit. However the cost of reasoning about a particular program's termination is exacerbated with the overhead of handwritten C code, due to iteration being an external construct to the language. This forces the the use of low level C code in the verification process, which Cogent strives to avoid.

This thesis aims to introduce recursive types to Cogent's type system, allowing internal iteration over internal data structures without the involvement of handwritten C code. While providing this benefit we must also respect the existing guarantees that Cogent enjoys, in particular, simple reasoning about functional correctness, totality, its static memory allocation, destructive updates, all while keeping in mind a possible efficient C representation for the later implementation of the compilation of Cogent code to C code.

# Chapter 2

## Progress Report

Previously in thesis A we elected Minigent as the place to design our new recursive types for Cogent. Our design was to introduce a new recursive parameter to Minigent's boxed records, in order to allow for recursively nested data structures. These data structures would be strictly positive as per the definition in **TODO: insert definition**, to prevent the programmer constructing infinitely recursive objects that would lead to termination issues, thus allowing for ease of verification in the Isabelle embedding.

I have completed work to incorporate the design into the parser, lexer and reorganiser compiler phases of Minigent, and I have begun work on beginning integration into the type checking phase. I have also added a test suits to test the correctness of my implementation, in addition to the existing suite of tests in the Minigent compiler.

**TODO: I'm starting to think this isn't very relevant to the thesis report. Remove?** In order to increase my work efficiency, I created an extension in VSCode that provides syntax highlighting for both Cogent and Minigent files, which allowed for an easier experience when constructing Minigent programs, and hence made it easier to write tests for new features.

My work has progressed according to my thesis A schedule, as in Figure 2.1. However, due to new discoveries **TODO: <- rephrase that** for the work on the type checker, my plan has changed for thesis C to allow more time for extensions, which are outlined in chapter 4.










| Task                | Thesis B  |          |          |          |           | Thesis C  |          |          |          |           |
|---------------------|---|----------|----------|----------|-----------|---|----------|----------|----------|-----------|
|                     | Week 1-2  | Week 3-4 | Week 5-6 | Week 7-8 | Week 9-10 | Week 1-2  | Week 3-4 | Week 5-6 | Week 7-8 | Week 9-10 |
| Minigent Reading    |  |          |          |          |           |   |          |          |          |           |
| Parsing/Lexing      |  |          |          |          |           |   |          |          |          |           |
| Type Checking       |  |          |          |          |           |   |          |          |          |           |
| Demonstration       |  |          |          |          |           |   |          |          |          |           |
| Report              |  |          |          |          |           |   |          |          |          |           |
| Type inference      |   |          |          |          |           |   |          |          |          |           |
| Final Demonstration |   |          |          |          |           |  |          |          |          |           |
| Final Report        |   |          |          |          |           |  |          |          |          |           |
| Extensions          |   |          |          |          |           |  |          |          |          |           |

Figure 2.1: The previous plan timeline, outlined in thesis A.

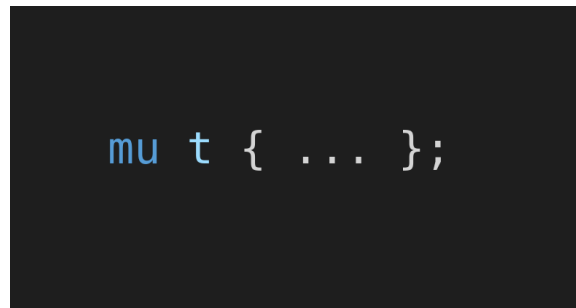


Figure 2.2: The new recursive parameter grammar for boxed records

## 2.1 Parser and Lexer

The Minigent lexer and parser now accepts a new recursive parameter extension to boxed records, as seen in Figure 2.2.

This new syntax is backwards compatible with the previous record syntax, so records without this new syntax in old code will not have to be changed in order for the change to be implemented.

**TODO: Should I talk about the code I wrote here?**

## 2.2 Reorganiser

In the reorganiser, a strict positivity check has been added to prevent the construction of infinitely recursive objects. This check recursively evaluates all of the functions in a Minigent program, and checks that their types do not contain a non-strictly positive type. This check also accounts for the shadowing of recursive parameter variables, in the event a nested boxed record declaration's recursive parameter shadows the parent record's parameter. These new recursive parameters are now distinguished from regular type variables, and the reorganiser no longer counts them as type variables.

**TODO: Functions calling themselves** **TODO: Unused recursive parameter variables**

## 2.3 Type Checker

**TODO: Talk about the stuff**

## 2.4 Testing

**TODO: talk about the tests**

## 2.5 Work efficiency

**TODO: talk about syntax highlighting, but also potentially remove**



# Chapter 3

## Reflection

TODO: reflect

# Chapter 4

## Revised Plan

TODO: The plan

# Bibliography

- [1] David Greenaway. Automated proof-producing abstraction of c code. 2014.
- [2] A proof assistant for higher-order logic, 2018.
- [3] 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. 2016.