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 8, 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.

Acknowledgements

Thanks to Kai, Liam and Christine for being great teachers and helping me break into the world of formal methods, without your guidance I would never have been able to work in such an exciting field! Thanks to my fantastic friends for supporting me and watching my health throughout my university years: Nick, James, James, Danni, Ofir, Namsu, Liana, the Coffee on Campus crew and many more! Finally, thanks to my Mum and Dad Cathy and Damian for helping me get to this point. Without their support and encouragement throughout all my years of schooling and university, without you both I couldn‘t have learned anything.

Contents

Chapter 1

Introduction

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

Background And Related Works

Our investigation of existing works will consider three domains: The existing types and features within Cogent, termination and recursive types and linear and uniqueness types.

2.1 Cogent Currently

Cogent's uniqueness type system features a variety of basic types as well as the more advanced variants and records.

2.1.1 Primitive Types

Cogent's primitive datatypes consist of Boolean types (`Bool`), and the four unsigned integer types `U8`, `U16`, `U32` and `U64`. Integer types can be upcast using the `upcast` keyword to convert a smaller integer type into a larger one (e.g. a `U8` into a `U32`). Details of these types can be seen in the syntax for Cogent's basic grammar in figure 2.1.

Each of these primitive types are of fixed size, and thus are *unboxed*, meaning they are stored on the stack and are not linear variables.

Cogent also features *tuples* (product types) through the standard tuple syntax across the ML family of languages: (a, b), and standard functions.

$$
\begin{array}{rlll}
\text{expressions} & e & ::= x \mid \ell & \text{(variables or literals)} \\
 & & \mid e_1 * e_2 & \text{(primitive operations)} \\
 & & \mid e_1\ e_2 & \text{(function application)} \\
 & & \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \text{(let statements)} \\
 & & \mid \textbf{if}\ e_1\ \ \textbf{then}\ e_2\ \textbf{else}\ e_3 & \text{(conditional)} \\
 & & \mid e_1\ e_2 & \text{(function application)} \\
 & & \mid e_1 :: \tau & \text{(type signatures)} \\
 & & \mid \ldots & \\
\text{types} & \tau & ::= a & \text{(type variables)} \\
 & & \mid \tau_1 \to \tau_2 & \text{(functions)} \\
 & & \mid \mathcal{T} & \\
 & & \mid \ldots & \\
\text{primitive types} & \mathcal{T} & ::= \ \mathrm{U8} \mid \mathrm{U16} \mid \mathrm{U32} \mid \mathrm{U64} \mid \textsc{Bool} & \\
\text{operators} & * & ::= + \mid \ \leq \ \mid \ \geq \ \mid \ \neq \ \mid \ldots & \\
\text{literals} & \ell & ::= \textsc{True} \mid \textsc{False} \mid \mathbb{N} &
\end{array}
$$

Figure 2.1: The basic syntax for Cogent [3]

$$
\begin{array}{rlll}
\text{expressions} & e & ::= \ldots \mid \kappa\ e & \text{(variant constructor)} \\
 & & \mid \textbf{case}\ e_1\ \textbf{of}\ \kappa\ e_2\ \ldots & \text{(pattern matching)} \\
\text{types} & \tau & ::= \ldots \mid \langle\overline{\kappa^n\ \tau}\rangle & \text{(variant types)} \\
\text{constructors} & \kappa & &
\end{array}
$$

Figure 2.2: The syntax for variant types within Cogent [3]

$$
\begin{array}{rlll}
\text{expressions} & e & ::= \ldots \mid \#\{\overline{f_i = e_i}\} & \text{(unboxed records)} \\
 & & \mid \textbf{take}\ x\ \{f = y\} = e_1\ \textbf{in}\ e_2 & \text{(record patterns)} \\
 & & \mid \textbf{put}\ e_1.f = e_2 & \text{(record updates)} \\
 & & \mid e_1.f & \text{(read record field)} \\
\text{types} & \tau & ::= \ldots \mid \{\overline{f_i^n\ \tau_i}\} & \text{(variant types)} \\
\text{field names} & f & &
\end{array}
$$

Figure 2.3: The syntax for record types within Cogent [3]

2.1.2 Variant Types

Cogent's variant types are inspired from the traditional sum types, where a variant type can contain one of many specified types, the syntax for which is defined in Figure 2.2.

Consider the following example, we can reconstruct Haskell's `Maybe` type using our variants:

$$\textbf{type}\ \mathit{Maybe}\ \text{a} = \langle\ \text{Nothing}\ ()\ \mid\ \text{Just a}\ \rangle$$

Or a type to represent a choice of colours:

$$\textbf{type}\ \mathit{RGB} = \langle\ \text{Red}\ ()\ \mid\ \text{Green}\ ()\ \mid\ \text{Blue}\ ()\ \rangle$$

Pattern matching on variants must be *complete*, i.e. you must give a case for every possible constructor of a variant, a constraint that helps keep functions total.

Variant types work well as a potential constructor for our recursive types. If we were to reference a recursive parameter inside a variant type, we would be able to create a recursive structure as in Haskell. However, as variants are *unboxed* (stored on the fixed size stack), we cannot allow for dynamically sized structures using only variants, so we must look elsewhere for a solution.

2.1.3 Record Types

Cogent's record types are objects that contain values via named fields. They come in two forms, *boxed* (stored dynamically on the heap) or *unboxed* (stored statically in the stack). The syntax of both is as in Figure 2.3.

For example, consider a record to bundle together user information:

```
type User = {
    name: String,
    age: U32,
    favouriteColour: RGB
}
```

Records allow for us to create dynamically sized objects, as we can used boxed records to chain together a combination of records on the heap. With the aid of variant types, records can allow us to construct our recursive types with a recursive parameter, using variants to give the differing construction cases for the type.

2.2 Termination And Recursive Types

Proving total correctness about the programs we write is a very desirable result. Computation performed by a program is useless if the program never returns the result of the computation. In a systems context, termination is especially desirable as an infinitely looping component of a system could cause it to hang, denying services to other parts of a system which could be core to system's function.

To deal with termination, we must consider the environment where we will prove termination for Cogent programs, the Isabelle embedding, and how we can make this process easier on the type level within Cogent.

2.2.1 Proving Termination in Isabelle

The official Isabelle tutorial [2] describes three methods of creating functions using the keywords **primrec**, **fun** and **function**. The first, **primrec**, allows one to create a *primitive recursive* function — one that returns a constant or removes a data type constructor from one of the arguments to the function in its body, 'decreasing' in size every time. These functions are total by construction, and therefore always terminate, removing the need of an explicit termination proof (which is required to reason inductively about any Isabelle function.) for all functions within Isabelle, unless they are defined to be partial, however as partial functions may not terminate we do not want to consider them using them for our verification. Primitive recursive functions however are limited in their expressiveness and are a subset of all computable functions, so we cannot rely on them for the general case.

In his tutorial, Krauss [4] discusses the details of the latter two of the three methods of creating functions in Isabelle. The **fun** keyword instructs Isabelle to try and solve all necessary termination proof obligations, rejecting the definition if it fails (either because the definition does not terminate or because Isabelle cannot prove it does alone). In contrast to **fun**, **function** requires that the termination proofs be solved manually by whoever is writing the function.

Due to their automatic termination proofs, we would like as many Cogent functions as possible to be primitive. For all others, we can achieve an embedding via **fun** to see if Isabelle can find a termination proof for us, and as a last resort use **function** and have the proof writer manually prove termination.

2.2.2 Strictly Positive Types

Adding recursive types to a type system allows for expressions that are potentially infinitely recursive, as discussed by Wadler [5], who explains the potential for recursive expressions to cause non-termination through polymorphic lambda calculus. In his paper, he discusses how this quality can be qualified with positive and negative data types.

Suppose a data type in its general form T and its data constructors $C_{1..n}$, each with a number of arguments $\tau_{i1}..\tau_{ik}$:

$$\begin{aligned} T = \; & C_1 \; \tau_{11} \; \tau_{12} \; \ldots \\ | \; & C_2 \; \tau_{21} \; \tau_{22} \; \ldots \\ | \; & \ldots \end{aligned}$$

Definition 2.2.1. A data type T is said to be in a *negative position* if T appears nested as an argument to a function an odd amount of times inside any τ_{ij}, and said to be in a *positive position* if T appears nested as an argument to a function an even amount of times inside τ_{ij}.

Definition 2.2.2. A data type T is a *negative* data type if it appears in a negative position in one of its constructors.

Definition 2.2.3. A data type T is a *positive* data type if only appears nested in a positive position in all of its constructors.

In simpler terms, if T appears to the left of a function arrow an odd amount of times, it is negative and if to the left an even amount of times then it is positive.

For example:

$$\begin{aligned} E &= C\ (\underline{E} \to E) \\ K &= D\ (\underline{(\underline{K} \to_1 Int)} \to_2 K) \end{aligned}$$

Here, the data type E is negative as it appears in a negative position (denoted here by an underline) to a function in the first argument of C. K however is positive as it appears only ever in a positive position as it is nested as an argument in function 1 ($\to_1$) and again in function 2 ($\to_2$) for a total of two times.

Allowing for negative types in our system allows for data structures that are infinitely recursive, which if iterated over would potentially cause non-termination. Consider the following example in Haskell:

```
data Bad = A (Bad → Bad)


g :: Bad → Bad
g (A f) = f (A f)


infiniteExpression = g (A g)
```

By our definition, we can see that our type *Bad* is a *negative* type and using it we were able to construct the infinitely recursive expression, `g (A g)` This is not an issue in Haskell due to its lazy evaluation and Haskell allows its programmers to write infinite loops freely, however in Cogent these expressions would be detrimental to our termination proofs as iterating over them potentially results in non-termination, and in this situation will hang when *infiniteExpression* is constructed. Although this example was constructed artificially, situations may arise where programmers may accidentally construct such an expression, so we must seek a way to eliminate them from our language.

Many theorem provers and dependently typed languages make use of *strictly positive* types, which prohibit the construction of infinitely recursive data structures, that under a dependant type system both negative and simple positive types allow. Agda[6], Coq[7] and even Isabelle[8] feature this exact constraint, as allowing for negative or non-strictly positive types introduce logical inconsistencies which can be used to prove false statements, something that is unacceptable for a theorem prover.

The definition of strictly positive is discussed by Conquand and Paulin [9], and is as follows:

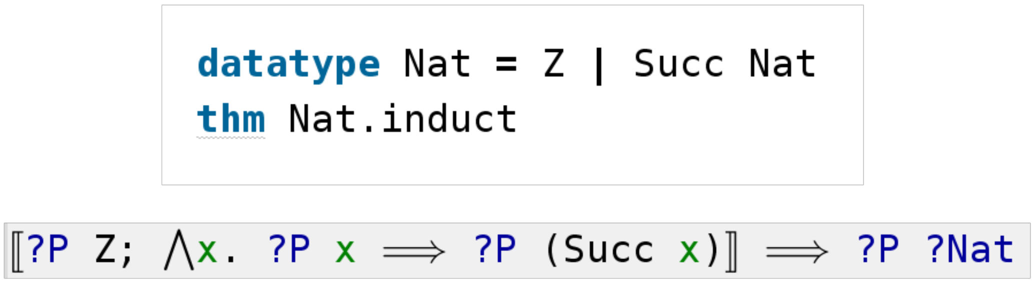
Definition 2.2.4. Given a data type T and its constructors $C_{1..n}$, for every argument τ_{ij} of any data constructor C_i where τ_{ij} is a function, T is said to be *strictly positive* if T does not occur as an argument to any τ_{ij}:

$$\forall\ \tau_{ij}.\ (\tau_{ij} = \phi_1 \to \cdots \to \phi_k) \implies T \notin \phi_{1..k-1}$$

Strictly positive types can also be defined as types where T appears in a negative or positive position exactly zero times (i.e. it does not appear to the left of any arrow).

In their paper, Conquand and Paulin further discuss the ability to produce an *eliminator* or a *fold* from any strictly positive type, which corresponds to an induction principle on the type.

Consider a type for natural numbers with two constructors for zero and successor:



```
datatype Nat = Z | Succ Nat
thm Nat.induct
```

```
⟦?P Z; ⋀x. ?P x ⟹ ?P (Succ x)⟧ ⟹ ?P ?Nat
```

Figure 2.4: A type for natural numbers defined in Isabelle, and the generated induction principle associated with the type.

$$Nat = \mathrm{Z} \mid \mathrm{S}\ Nat$$

We can see Nat is a strictly positive type and the induction principle it produces for any predicate over natural numbers, P, is:

$$\frac{P(Z) \qquad \forall(X : Nat).\ P(X) \implies P(S\ X)}{\forall(N : Nat).\ \ P(N)}$$

Where in order to prove our predicate P, we prove it for each case of how our type T could have been constructed, which each constructor for our our type supplies. That is, to prove any predicate P inductively over nats ($\forall(N : Nat).\ P(N)$) we prove it for the base (zero) case $P(Z)$ and then assuming the predicate holds for a natural number X, we prove it for its successor case $S\ X$: $P(X) \implies P(S\ X)$.

The interactive theorem prover Isabelle generates the same induction principle for any type created in Isabelle. We can get the same induction principle over natural numbers by redefining our Nat type in Isabelle, as in figure 2.4.

Considering our Cogent embedding will be within Isabelle, if we can embed our native Cogent types into an Isabelle type then we gain Isabelle's automatically generated induction principle over our Cogent types, allowing for much simpler reasoning about our Isabelle embedding.

$$\frac{\Gamma_2\Gamma_1 \vdash e : \tau}{\Gamma_1\Gamma_2 \vdash e : \tau}\ \textit{Exchange} \qquad \frac{\Gamma_1 \vdash e : \tau}{\Gamma_1\Gamma_2 \vdash e : \tau}\ \textit{Weakening} \qquad \frac{\Gamma_1\Gamma_1\Gamma_2 \vdash e : \tau}{\Gamma_1\Gamma_2 \vdash e : \tau}\ \textit{Contraction}$$

Figure 2.5: Structural typing rules

2.3 Linear And Uniqueness Types

Linear types are a kind of substructural type system as discussed by Walker [10]. Many standard programming languages such as C, JAVA and HASKELL feature three standard structural typing rules, described in figure Figure 2.5.

Exchange is the rule that states that the order in which we add variables in an environment is irrelevant. A conclusion of this is that if a term e typechecks under environment Γ, then any permutation of Γ will also typecheck e.

Weakening states that if a term e typechecks under the assumptions in Γ_1, then e will also typecheck if extra assumptions are added to the environment.

Contraction states that if we can typecheck a term e using two identical assumptions, then we are able to check e with just one of those two assumptions.

Substructural type systems however control access to information within the program by limiting which of the structural typing rules are allowed under certain contexts. Linear types ban the use of the contraction and weakening rules, which has the consequence that all linear variables must be used at least once (by lack of weakening) and at most once (by lack of contraction), hence exactly one time.

One powerful benefit that linear types allow is *static allocation* of objects, which Cogent features. Predicting when an object in a program will be last used (and after deallocated) is undecidable as it is a nontrivial semantic property by Rice's Theorem [11], however a linear type system can make it possible to deallocate this memory after its first use by the use once rule. The result is a language that does not require a garbage collector, can check that programs appropriately handle allocated resources statically as this can now be determined statically.

Wadler [12] also describes the performance benefits of destructive updates that linear types potentially grant. As we have a guarantee that no other part of a program is referencing a particular object (variables must be used exactly once), when performing an operation on an object, the resultant object can be our old object with the result of our operation performed in place (i.e. destructively mutated).

Consider the following program in Java:

```java
// A function that doubles all the elements of an input list
ArrayList<Integer> doubleList (ArrayList<Integer> input) {
    for (int i = 0; i < input.length; i++) {
        Integer n = input.get(i);
        input.set(i, n*2);
    }
    return input;
}
...
ArrayList<Integer> oldNumbers = ...;
ArrayList<Integer> copyOfNumbers = doubleList(oldNumbers);
// Mistake! oldNumbers has been updated in place,
// and copyOfNumbers and oldNumbers point to the same object!
```

In this example we attempt to double a copy of a list of numbers in place by use of the `doubleNumbers` function on `copyOfNumbers`, however by updating it in place has changed the original variable outside the function `oldNumbers`. If a programmer mistakenly uses `oldNumbers` again without realising that `doubleNumbers` has mutated it instead of a copy of it, it would most likely cause an error. In Java this kind of destructive update cannot be done safely whilst `oldNumbers` still exists, and we must resort to copying.

Linear types prevent this kind of mistake, as the duplicate reference that `oldNumbers` and `copyOfNumbers` share would be eliminated once `oldNumbers` is used once, which in turn allows for a destructive update on `oldNumbers` to take place with the result stored in copyOfNumbers.

Wadler however shows that mere linearity is not enough to guarantee safe destructive updates, as nonlinear variables with multiple references may be cast to linear ones, breaking the single reference guarantee for linear types. This is from the result that adding typecasting to and from linear variables grants controlled access to the contraction and weakening rules that linear types explicitly prohibit.

He further discusses separately [13] that with the removal of the ability to perform such an action, one can gain the *uniqueness* types that Cogent exhibits. With uniqueness types, it is truly impossible for linear variables to have multiple references, and thus destructive updates on these variables are safe.

All boxed types in Cogent are linear, and therefore must have at most one reference to each, however unboxed objects are only linear if they contain other linear values. With our implementation of recursive types, we must consider maintaining the linear and uniqueness constraints that Cogent features, and create these types in such a way that they integrate nicely with the existing system.

Chapter 3

Proposal and Plan

Considering our investigation, we seek to expand upon the existing Cogent infrastructure keeping in mind two major requirements: the necessity for our programs to be easily provably *total*, and the preservation of the benefits guaranteed by Congent's uniqueness types system.

Our proposed design will be implemented in *minigent*, a stripped down version of Cogent that features only the parsing, type checking and type inference components of Cogent in order to focus on integrating smoothly with the existing project. As the compilation of an Isabelle embedding and C code is out of scope for this project, using minigent will not hinder the potential of this project.

3.1 Structure And Syntax

We will extend our grammar with a recursive parameter operator **mu** as in Figure 3.1 to add recursive type parameters and use variant constructors to construct our types, which will be contained in boxed records.

We present the syntax for our new recursive types:

$$\textbf{type}\ \text{Recursive} = \textbf{mu}\ T\ \{\ \textit{field}\colon \langle\overline{\kappa^n\ \tau}\rangle\ \}$$

Where:

- **mu** $T.$ is a recursive parameter T that references the following type, It may be used in the body of the type

- { *field*: } is a boxed record with a field in it that has a valid field name
- $\langle \overline{\kappa^n\ \tau} \rangle$ is an alternate type with n constructors, with κ^i meaning constructor i in the series that takes one argument of type τ

Our decision to use Cogent's existing records to house recursive types comes from the fact that they are boxed — Cogent's verification framework does not allow for dynamic stack allocation, thereby prohibiting our dynamically sized recursive data structures on the stack. Hence we turn to the heap, where boxed records are stored. As records are currently implemented in Cogent, adding recursive types would not require new rules or a new implementation for static memory allocation which is already implemented for records.

Consider the following example recursive type for a list:

$$\textbf{type}\ \text{List a} = \textbf{mu}\ T\ \{\ \text{deref: } \langle\ \text{Nil ()} \mid \text{Cons}\ (a, T)\ \rangle\ \}$$

The recursive parameter T references the following record, and is used in the *Cons* data constructor to recursively reference the rest of the list. The variant type $\langle$ Nil | Cons (a, T) $\rangle$ we supply as the type of the field *deref* has two data constructors, *Nil* for the end of the list and *Cons* for an element followed by the remains of the list. Our list also takes a type parameter a, allowing our list to be generic.

We can define a recursive function to sum a U32 List by pattern matching on records as in Figure 3.2.

$$\begin{array}{llll} \text{types} & \tau ::= \ldots \mid \textbf{mu}\ T\ \tau & & \text{(recursive type declaration)} \\ & \quad\quad\ \mid T & & \text{(recursive type parameters)} \end{array}$$

Figure 3.1: Extending our syntax with the mu operator

```
sum : (List U32)! → U32
sum (r { deref }) =
   deref
      | Nil          → 0
      | Cons (x,r′) → x + sum r′
```

Figure 3.2: a recursive sum function

3.2 Totality

We will require that our recursive types be strictly positive, as in Agda, Coq and Isabelle in order to produce a cleaner and easier to reason about embedding within Isabelle. As our type system's constraints will be compatible with Isabelle's strict positivity constraint we can more easily achieve an embedding within Isabelle for our new recursive types, and potentially produce an induction principle within Isabelle to reason about these types directly.

Given our definition, we can check the strict positivity of a type with the following judgement:

$$\frac{\forall \tau.\ \Gamma \vdash (\tau \sqsubseteq \phi \rightarrow \psi) \implies T \notin \phi}{\Gamma \vdash \mu T.\ \{\ f : \langle \overline{\kappa^n\ \tau} \rangle\ \}}$$

Where:

- $\tau \sqsubseteq \phi \rightarrow \psi$ is the subtyping relation that τ is a subtype of $\phi \rightarrow \psi$ (i.e. τ has the 'shape' of a function, from type ϕ to ψ)
- $T \notin \phi$ is the constraint that the recursive parameter T does not exist anywhere in the type ϕ.

Whilst not in scope for this project, later implementations using our recursive types can write recursive functions such as the sum function in Figure 3.2. Due to our embedding considerations, we potentially can allow primitive recursive functions in Cogent to directly translate to a **primrec** function in Isabelle. For all other functions, proof writers can resort to **fun** and as a last resort **function**, however due to the high level and functional nature of the embedding of our recursive types and our potential induction principle over them, the cost of verifying these functions is minimised.

3.3 Preservation of Uniqueness

As boxed records are linear and enforce that the contents of the record are all linear, we can use Cogent's existing type system to provide us with the linear guarantees we wish to preserve. As we will only be able to create structures using Cogent's existing rules and as we introduce no new means of creating expressions, we will not break the existing rules either.

Destructive updates can occur on our recursive structures as the only references to them are recursively by other structures, where the topmost structure will be referenced by a single linear variable. As we will only allow these structures to be created with Cogent's existing rules, we prevent our uniqueness guarantees from being broken.

3.4 Potential Extensions

This project allows for multiple potential extensions after the initial core development has been completed. Some potential extensions if time allows are:

- Adding a check that functions are primitive recursive within minigent, introduce a new keyword to require that a function is primitive recursive in minigent (i.e. the same as Isabelle's **primrec**).
- Write a small standard library of recursive data types, e.g. lists, binary trees, sets.

3.5 Timeline

The timeline for the project in thesis B and C is outlined in Figure 3.3.

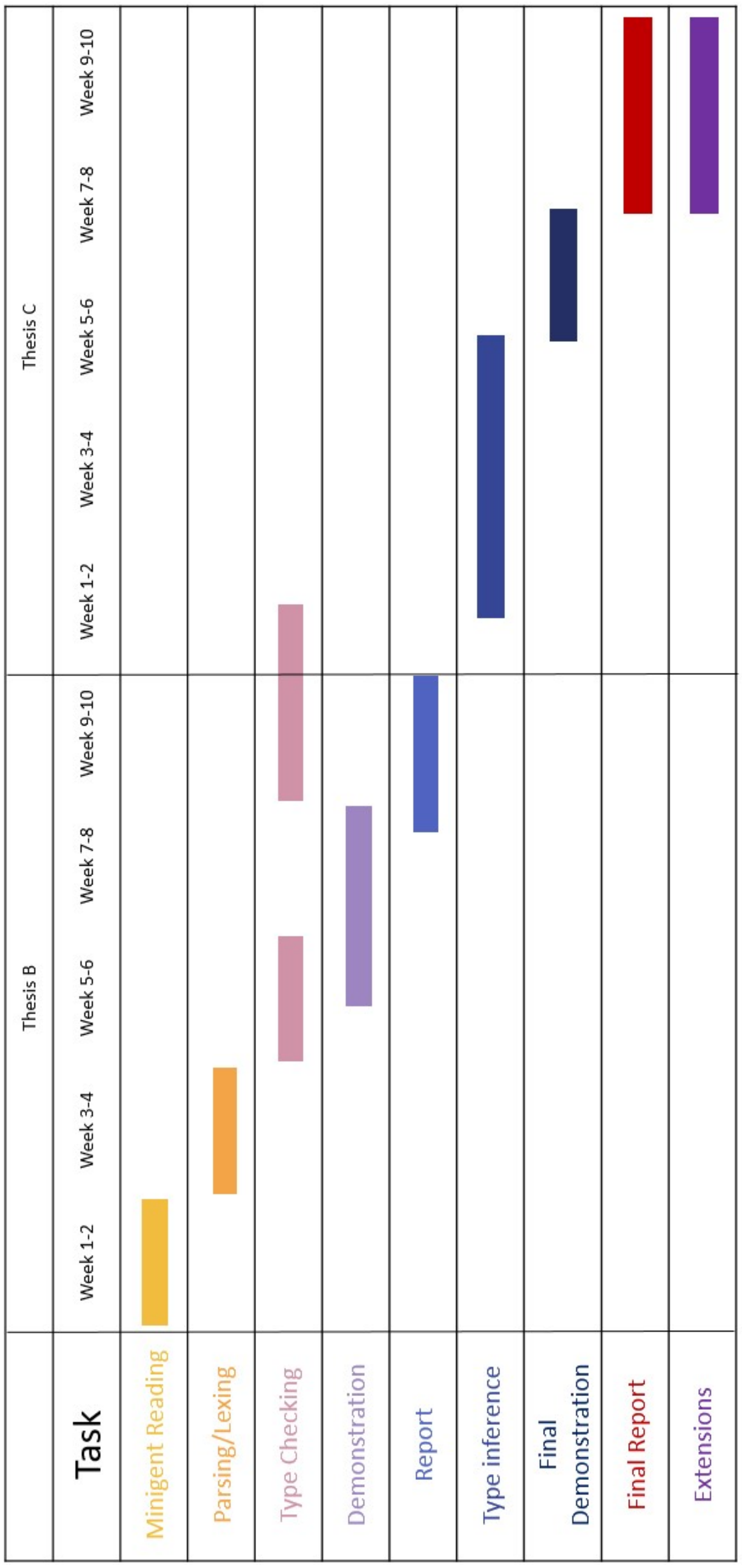


Figure 3.3: The timeline for the project

Chapter 4

Project Preparations

Prior to my thesis year, I took the subjects COMP2111, COMP3141, COMP3161, COMP3891 and COMP4161, all of which have provided me with knowledge I require to progress with my thesis project.

COMP2111 (System Modelling and Design) gave me adequate knowledge of the basis of verification — practice with developing mathematical models to describe the semantics of programs and their states by means of axiomatic semantics, Hoare logic and creating invariants for simple programs. I also learned about methods of data refinement, whereby abstract representations are refined to low level implementations via a specification.

In COMP3141 (Software System Design and Implementation) I was introduced to functional programming through Haskell, which is of great use to my progress with my project as Cogent is a functional programming language implemented in Haskell. As well as this I learned the basics of more advanced formal methods, for example the Curry-Howard Correspondence and working with types to ensure correctness of programs.

In COMP3161 (Concepts of Programming Languages), I was able to learn more Haskell and the basics of programming language theory and type theory by implementing the small programming language minhs in Haskell. As my project is heavily involved with programming language development, specifically in Cogent's type system, this subject is very relevant to my project and thus the knowledge I learned from it will be essential to my success.

In COMP3891 (Extended Operating Systems), I learned background knowledge about systems development, how lower level systems work and various implementation strategies for these systems. As Cogent is a domain specific language for systems development, this knowledge will benefit me throughout the project as it has throughout my research when understanding the context of Cogent.

In COMP4161 I learned more advanced methods in formal verification, and gained experience with using the theorem prover Isabelle. In addition to this, I gained greater experience in understanding the terminology and concepts behind various formal methods areas, such as reading natural deduction, understanding invariants and preservation of correctness, proofs about termination and totality of programs, and correctness with respect to specifications.

The skills I have gained in my university subjects so far have provided me with concrete knowledge on how to progress with my thesis project and understand the underlying theory and core concepts behind it.

Chapter 5

Project Progress

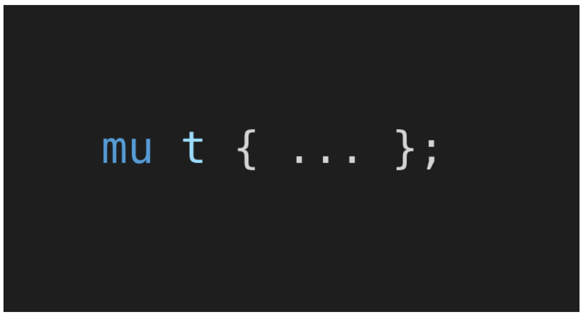
Previously in thesis A we elected Minigent as the place to design our new recursive types for Cogent. Our design was to introduce a new recursive parameter to Minigent's boxed records, in order to allow for recursively nested data structures. These data structures would be strictly positive as per definition 2.2.4, to prevent the programmer constructing infinitely recursive objects that would lead to termination issues, thus allowing for ease of verification in the Isabelle embedding.

Work has been completed to incorporate the design into the parser, lexer and reorganiser compiler phases of Minigent, and work has begun work on integrating updatin the type checking phase. In addition to the existing suite of tests in the Minigent compiler, additional tests have been added to test the newly added extension.

TODO: I'm starting to think this isn't very relevant to the thesis report, let alone the project. 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 3.3. 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 7.



```
mu t { ... };
```

Figure 5.1: The new recursive parameter grammar for boxed records

5.1 Parser and Lexer

The Minigent lexer and parser now accepts a new recursive parameter extension to boxed records, as seen in Figure 5.1.

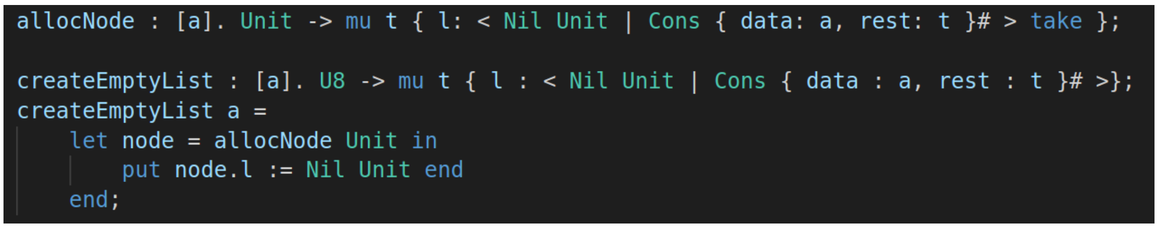
This new syntax is backwards compatible with the previous record syntax, so records without this new syntax in old code will not have to be changed in order for the change to be implemented.

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

5.2 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 positve 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.

Functions that also call themselves successfully typecheck as their type signature provides enough information to check the argument passed to the function, so basic recursion works as expected.



```
allocNode : [a]. Unit -> mu t { l: < Nil Unit | Cons { data: a, rest: t }# > take };

createEmptyList : [a]. U8 -> mu t { l : < Nil Unit | Cons { data : a, rest : t }# >};
createEmptyList a =
    let node = allocNode Unit in
        put node.l := Nil Unit end
    end;
```

Figure 5.2: Testing the construction of an empty list

TODO: Unused recursive parameter variables?

5.3 Type Checker

Work has begun on the type checker to check the types of records.

TODO: Talk about the stuff

5.4 Testing

A suite of tests has been added to the existing Minigent test suite in order to test the new extension's expected behaviour.

The tests aim to test the intended functionality of our recursive types, including constructing and manipulating recursive data structures (such as lists), as well as recursive functions that do not use our recursive types (such as functions recursing on U32 integers, etc.).

An example of such a test is seen in Figure 5.2, which constructs an empty list given the presence of an external allocation function.

These tests also test the intended effects of our new types, such as the strict positivity guarantee they provide and their interaction with the existing type system. Whilst the existing tests test the backwards compatability of record types, our new tests check that no recursive parameters are used non-strictly positive, that shadowing of recursive parameters behaves as intended, and that recursive parameters are seperate from quantified type variables.

TODO: talk about sp testing and such

5.5 Work efficiency

TODO: talk about syntax highlighting, but also potentially remove

Chapter 6

Reflection

TODO: reflect

Chapter 7

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.

[4] Alexander Krauss. Defining recursive functions in isabelle/hol.

[5] Philip Wadler. Recursive types for free! 1990.

[6] Ulf Norell, Andreas Abel, Nils Anders Danielsson, and Makoto Takeyama et al. Agda documentation, data types, strict positivity, 2016.

[7] Inria. Calculus of inductive constructions, 2018.

[8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelles logics: Hol, 2009.

[9] Thierry Coquand and Christine Paulin. Inductively defined types. 1988.

[10] David Walker. *Substructural Type Systems.* 04 2019.

[11] Michael Sipser. *Introduction to the Theory of Computation.* 1997.

[12] Philip Wadler. Linear types can change the world! 1990.

[13] Philip Wadler. Is there a use for linear logic? 1991.