UNSW
AUSTRALIA

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: December 4, 2019 Student ID: z5059840

Supervisor: Christine Rizkallah

Abstract

Cogent is a functional programming language with uniqueness types 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 uniqueness 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. Datatypes, 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 datatypes as well as iterators over these datatypes directly in Cogent rather than resorting to C. This would enable programmers to write more programs in Cogent without using external C, thus easing the verification effort by allowing more use of Cogent's shallow embedding.

Acknowledgements

Thanks to Liam, Kai 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 again to Liam and Christine, as well as Vincent and Zilin for continuously helping me work on my thesis, I've learned a lot from all of you and I'm extremely thankful to have had the opportunity to learn beside you.

Thanks to my fantastic friends for supporting me and watching my health throughout my university years: Nick, James Treloar, James Dowers, Danni, Ofir, Namsu, Andrew and many more!

Thanks especially to Liana for looking after me during the rough patches and supporting me throughout the year, your support means the world to me.

Finally, thanks to my Mum and Dad for helping me get to this point. Without their support, encouragement, and especially their wisdom throughout all my years of schooling and university, I wouldn't have been able to learn as much as I did.

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 verification 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 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/HOL 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.

In contrast to many existing functional languages, which operate on layers of abstractions away from the system Cogent is suitable for low level systems development, 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 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 Cogent's C FFI. Proving totality of code is necessary to ensure desired properties, for instance that a system will not hang or deny services to other systems. The cost of reasoning about a particular program's termination is exacerbated, however, by 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 programmers should strive 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 and benefits 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

O'Connor [4] describes in his master thesis Cogent's uniqueness type system, which features typing constructs that facilitate the memory safety and totality guarantees that Cogent enjoys, as well as the features of Cogent's type checking system which employs constraint generation and constraint solving techniques to show type correctness.

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 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.

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

Figure 2.1: The basic syntax for Cogent [4]

$$
\begin{array}{rrll}
\text{expressions} & e & ::= \ldots \mid \kappa\ e & \text{(variant constructor)} \\
& & \mid \mathsf{case}\ e_1\ \mathsf{of}\ x.\ \kappa\ e_2\ \mathsf{else}\ y.\ e_3 & \text{(pattern matching)} \\
& & \mid \mathsf{case}\ e_1\ \mathsf{of}\ x.\ \kappa & \text{(irrefutable match)} \\
\text{types} & \tau & ::= \ldots \mid \langle \overline{\kappa^n\ \tau} \rangle & \text{(variant types)} \\
\text{constructors} & \kappa & &
\end{array}
$$

Figure 2.2: The syntax for variant types within Cogent [4]. An $\overline{\text{overline}}$ indicates a set of 1 or more options

$$
\begin{array}{rrll}
\text{expressions} & e & ::= \ldots \mid \#\{\overline{f_i = e_i}\} & \text{(unboxed records)} \\
& & \mid \mathsf{take}\ x\ \{f = y\} = e_1\ \mathsf{in}\ e_2 & \text{(record patterns)} \\
& & \mid \mathsf{put}\ e_1.f = e_2 & \text{(record updates)} \\
& & \mid e_1.f & \text{(read record field)} \\
\text{types} & \tau & ::= \ldots \mid \{\overline{f_i^n\ \tau_i}\}\ s & \text{(record types)} \\
\text{sigils} & s & ::= \textcircled{w} & \text{(writable)} \\
& & \mid \quad \textcircled{r} & \text{(readable)} \\
& & \mid \quad \textcircled{u} & \text{(unboxed)} \\
& & \mid \quad \alpha & \text{(unknown)} \\
\text{field names} & f & &
\end{array}
$$

Figure 2.3: The syntax for record types within Cogent [3]. An $\overline{\text{overline}}$ indicates a set of 1 or more options

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.

$$\frac{G_1 \vdash e_1 : \alpha \to \tau \rightsquigarrow G_2 \mid C_1 \quad G_2 \vdash e_2 : \alpha \rightsquigarrow G_3 \mid C_2}{G_1 \vdash e_1\ e_2 \rightsquigarrow G_3 \mid C_1 \wedge C_2}\ \text{CG-App}$$

Figure 2.4: The function application constraint generation rule

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.

2.1.4 Constraint Based Type Inference and Type Checking

Cogent features a constraint based type checker, which checks the type correctness of programs by first generating constraints based on the body of each defined function, and then solving those constraints in the constraint solver. A constraint can be any requirement on the types featuring in a Cogent program, for example type equality ($\tau_1 \approx \tau_2$) and subtyping ($\tau_1 \sqsubseteq \tau_2$).

The constraint generation component has a defined rule for which constraints to generate

based on which expression is encountered in the program. For example, the function application constraint generation rule in Figure 2.4, which starts with the function application expression $e_1\ e_2$, and generates the constraints that the function expression e_1 must have type $\alpha \to \tau$ for some α and τ, and that the argument expression e_2 must have the type of α.

The solver component has 5 phases; the *simplifier*, the *normaliser*, the *join/meet*, the *sink/float* and the *unifier* phases. Each phase contains various rules that rewrite constraints and the types that feature in constraints. The solver feeds the set of all generated constraints through each phase until one phase rewrites one of the given constraints. It then restarts the solver phase until all constraints are eliminated or no changes can be made to any constraint, which correspond to the program being type correct or type incorrect respectively.

Take for example a type equality constraint between two function types, $\tau_1 \to \tau_2 \eqsim \rho_1 \to \rho_2$. The simplifier phase contains a rule that this constraint can be simplified into solving two smaller constraints:

$$\tau_1 \to \tau_2 \eqsim \rho_1 \to \rho_2 \quad \overset{\text{simp}}{\hookrightarrow} \quad \tau_1 \eqsim \rho_1, \tau_2 \eqsim \rho_2$$

2.2 Termination and Recursive Types

Proving total correctness about the programs we write is a very desirable result, as 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 the system's function.

To deal with termination, we must consider the environment where we will prove termination for Cogent programs — the Isabelle/HOL embedding and how we can make this process easier on the type level within Cogent. We then seek a way to facilitate checking the termination of functions in Cogent.

2.2.1 Proving Termination in Isabelle

The official Isabelle/HOL 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 primitive recursive functions are total by construction and therefore always terminate, removing the need for an explicit termination proof. This is required to reason inductively about any Isabelle/HOL function, 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 computable functions, so we cannot rely on them for the general case.

In his tutorial, Krauss [5] discusses the details of creating functions using the **fun** and **function** keywords in Isabelle. The **fun** keyword instructs Isabelle/HOL 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/HOL cannot prove it). In contrast to **fun**, **function** requires that the termination proofs be provided 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** in the hope that Isabelle/HOL can find a termination proof. As a last resort we use **function** and have the proof engineer manually give a proof of 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 [6]. Wadler 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 $\tau_{i1}..\tau_{ik}$:

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

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 it 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 number of times, it is negative, and if it appears to the left an even amount of times then it is positive.

For example:

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

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 is positive as it only every appears 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 may 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 definition, we can see that the 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's permissiveness of general unbounded recursion, 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 evaluated. Although this example was constructed artificially, situations may arise where programmers may 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 regular types allow. Agda [7], Coq [8] and even Isabelle [9] feature this exact constraint, as allowing for negative or non-strictly positive types introduces logical inconsistencies, something that is unacceptable for a theorem prover.

The definition of strictly positive is given by Coquand and Paulin [10] 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:

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

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 natural numbers ($\forall(N : Nat).\ P(N)$) 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 $S\ X$: $P(X) \implies P(S\ X)$.

Coquand and Paulin [10] describe the eliminator for natural numbers, a means to realise a predicate over natural numbers, as follows:

$$eliminator : P \to P(0) \to (x : Nat \to P(x) \to P(S(x))) \to P(Nat)$$

Which, given a predicate P, a proof of the predicate for 0, $P(0)$ and the proof of the predicate for the successor of any natural number $x \to P(x) \to P(S(x))$, we receive a proof of the given predicate P for all natural numbers. This eliminator is indeed similar to the induction principle we previously defined, which is no coincidence due to their correspondence.

The interactive theorem prover Isabelle/HOL generates the same induction principle for any type created. We can get the same induction principle over natural numbers by redefining our Nat type in Isabelle, as in 2.5.

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

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

Figure 2.5: A type for natural numbers defined in Isabelle/HOL and the generated induction principle associated with the type.

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

2.2.3 Termination Checking

Many languages feature language based termination checking, a process where a language compiler has the ability to check if a user-written function terminates. For many theorem provers such as Isabelle, termination is a requirement as permitting non-terminating functions can introduce logical inconsistencies in the theorem prover.

One technique to identify if a function terminates is to check if it belongs to a subclass of functions that are known to terminate. One such subclass is *primitive recursive* functions, which take away one constructor from a data structure in a recursive call, and gradually approach the 'bottom' of the data structure, where they terminate.

Colson [11], in his paper on primitive recursive algorithms describes an inductive definition of primitive recursive functions:

Definition 2.2.5. A function is *primitive recursive* if it can be constructed using the following combinators:

- O, the *null* or *constant* combinator that takes zero arguments.
- Succ, the *successor* combinator that takes one argument, and returns the successor of that argument.

- π_i^n, the *projection* combinator that takes $n \geq 1$ arguments and returns the i'th argument, where $1 \leq i \leq n$.
- $S_m^n(f, g_1, g_2, \ldots, g_n)$, the *composition* combinator, where f and $g_{1..n}$ are primitive recursive combinators that take n and m arguments respectfully, such that

$$S_m^n(f, g_1, g_2, \ldots, g_n) = f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$

- $Rec(b, s)$, where b and s are primitive recursive combinators that take $n+1$ arguments and $n+2$ arguments respectfully is the *recursion* combinator with base case b and recursive step n.

Expressing a function in terms of these combinators shows that it is primitive recursive, and hence terminating. To make use of these combinators for termination checking in Cogent, we seek a means to translate Cogent functions to a combination of the above combinators, and map expressions to natural numbers that these combinators operate on.

The primitive recursive subclass however cannot express more complex functions such as the *Ackermann function* [12], so using this technique limits termination checking in Cogent to functions that perform simple iteration or non-recursive functions.

Alternatively, we may consider using structurally recursive methods as discussed by Downen, Johnson-Freyd, and Ariola [13] and Abel and Altenkirch [14]. In this case our goal is to find a *termination order* upon which our functions terminate — An ordering where each recursive call operates on arguments that are 'smaller' than the previous, and prove that our recursion gradually approaches termination for a given input.

A language with strictly positive types can rely on the fact that functions that recurse structurally on data structures terminate, as all strictly positive datatypes cannot be infinite.

Abel and Altenkirch [14] talk in particular about *structural ordering*, by measuring term size in terms of constructors. They describe two axioms to measure this order, the first

being given a term e and a constructor C, the transitive closure of:

$$e < C(\ldots, e, \ldots)$$

And for measuring the size of function expressions, given a function f of type $\alpha \longrightarrow \tau$, and an argument a of type α:

$$f\ a \leq f$$

Abel [15] also discusses seperately termination checking via means of creating a call graph between variables featured in recursive calls. He creates a matrix of relations between function argument variables and variables that feature in the recursive call of a function. These relations are relations on the structural size of the variables, which take the form of inequality ($<$), equality ($=$) and unknown ($?$) sizes, and are generated during language expressions such as constructor elimination.

Take for example the following function which operates on two Peano numbers:

$$\begin{array}{l} f(x, y) = \\ \quad \mathsf{case}\ x\ \mathsf{of} \\ \quad\quad Z \to Z \\ \quad\quad S\ x' \to f(x', y) \\ \quad \mathsf{end} \end{array}$$

For the recursive call in the function, we produce the following relation matrix:

	x	y
x'	$<$	$?$
y	$?$	$=$

Which shows that $x' < x$ (via constructor elimination), $y = y$ (as y is unchanged), and $x\ ?\ y$ and $x'\ ?\ y$ (no relation between x', x and y).

By inspecting the diagonal column of the matrix, the respective relation between variables in the same argument position, we can show the function terminates by finding a

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

Figure 2.6: Structural typing rules

lexicographical ordering between the arguments.

This method shows promise for Cogent, as our language features many ways to produce size relations, such as take, put, let and case. In addition to this, if we find such a lexicographical ordering we may be able to produce an Isabelle/HOL proof of our function termination automatically, assisting the ease of verifying termination.

2.3 Linear and Uniqueness Types

Linear types are a kind of substructural type system as discussed by Walker [16]. Many standard programming languages such as C, Java and Haskell feature three standard structural typing rules, described in Figure 2.6.

The *exchange* rule 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.

The *weakening* rule 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.

The *contraction* rule 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 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 afterwards deallocated) is undecidable as it is a nontrivial semantic property by Rice's Theorem [17], however as a uniqueness types system only ever allows a single pointer to a given allocation, we can check that programs appropriately handle allocated resources statically.

Wadler [18] 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:

```
1   // A function that doubles all the elements of an input list
2   ArrayList<Integer> doubleList (ArrayList<Integer> input) {
3       for (int i = 0; i < input.length; i++) {
4           Integer n = input.get(i);
5           input.set(i, n*2);
6       }
7       return input;
8   }
9   ...
10  ArrayList<Integer> oldNumbers = ...;
11  ArrayList<Integer> copyOfNumbers = doubleList(oldNumbers);
12  // Mistake! oldNumbers has been updated in place,
13  // 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, 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 that with the removal of the ability to perform such an action, one can gain the *uniqueness* types that Cogent exhibits. As uniqueness types prohibit multiple references to allocated objects, destructive updates upon these object become 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

Completed Work

We have expanded upon the existing Cogent infrastructure keeping in mind three major requirements: the ability for programmers to create recursive iterable data structures via recursive types, the necessity for our programs to be easily proven *total*, and the preservation of the benefits guaranteed by Cogent's uniqueness type system.

Our proposed design has been implemented in *Minigent*, a stripped down version of Cogent that features only the type checking component of Cogent. This allows us to focus on the design of our recursive types without having to change the compiler's C code generation, Isabelle/HOL embedding and refinement proof linking the two.

We have incorporated our proposed design into the parsing, lexing, reorganising and type-checking compiler phases of Minigent. Moreover we provide a formalisation of termination checking for primitive recursive functions. In addition to ensuring that the existing test suite in the Minigent typechecker still works, we have added additional tests to test the newly added recursive types.

$$\begin{aligned} \text{types } \tau ::= \ldots \mid \mu\ \{\overline{f_i^u\ \tau_i}\} \\ \mid t_r \\ \text{recursive parameters } \mu ::= \mathbf{mu}\ t \mid \varepsilon \end{aligned}$$

Figure 3.1: Extending our record syntax with recursive parameters

```
type List = mu t {
      l: ⟨
            Cons (t, U32)
          | Nil ()
      ⟩
}
```

Figure 3.2: Constructing the List datatype using recursive parameters

3.1 Grammar and Syntax

We have extended the existing record grammar with an optional recursive parameter **mu** presented in Figure 3.1 to add recursive type parameters to Cogent's boxed records, this enables the use of recursive parameter variables in the language.

Boxed records now feature an optional recursive type variable bound by the **mu** keyword, which may optionally feature in a record type in which they are bound. This also ensures our new record syntax is backwards compatible with the previous record syntax, enabling existing Cogent programs to remain unchanged.

Figure 3.2 demonstrates the use of our new grammar to construct an integer list datatype, with the use of a variant type in a recursive record field l that is either Nil, the end of the list, or Cons, the current integer and the rest of the list. Further examples are discussed in section 3.5 at the end of the chapter.

3.2 Strictly Positive Types

Checking that all types only occur *strictly positive* has been implemented in the reorganisation phase of the compiler instead of during type checking. This allows for a simple one-pass algorithm that can analyse all the types in the program at once, and a simpler constraint generation and solving phases.

Figure 3.3 describes the strictly positive check over Cogent's types. This algorithm keeps track of a set of in scope bound recursive parameters $\mathcal{B}$ and a set of recursive parameters currently appear in a non-strictly positive position $\mathcal{S}$.

$$\mathbf{RP}(\mathbf{mu}\ t) = \{t\}$$
$$\mathbf{RP}(\varepsilon) = \{\}$$

$$\frac{\text{for all } i;\ \mathcal{S} \setminus \mathbf{RP}(\mu); \mathcal{B} \cup \mathbf{RP}(\mu) \vdash \tau_i\ \mathbf{SP}}{\mathcal{S}; \mathcal{B} \vdash \mu\ \{\overline{f_i : \tau_i}\}\ \mathbf{SP}}\ \text{SP-REC} \qquad \frac{\text{for all } i;\ \mathcal{S}; \mathcal{B} \vdash \tau_i\ \mathbf{SP}}{\mathcal{S}; \mathcal{B} \vdash \langle \overline{\kappa_i : \tau_i} \rangle\ \mathbf{SP}}\ \text{SP-VAR}$$

$$\frac{\mathcal{S} \cup \mathcal{B}; \mathcal{B} \vdash \tau_1\ \mathbf{SP} \quad \mathcal{S}; \mathcal{B} \vdash \tau_2\ \mathbf{SP}}{\mathcal{S}; \mathcal{B} \vdash \tau_1 \to \tau_2\ \mathbf{SP}}\ \text{SP-FUNC}$$

$$\frac{t_r \notin \mathcal{S}}{\mathcal{S}; \mathcal{B} \vdash t_r\ \mathbf{SP}}\ \text{SP-RVAR} \qquad \frac{}{\mathcal{S}; \mathcal{B} \vdash \mathcal{T}\ \mathbf{SP}}\ \text{SP-PRIM} \qquad \frac{}{\mathcal{S}; \mathcal{B} \vdash a\ \mathbf{SP}}\ \text{SP-TVAR}$$

Figure 3.3: The rules for checking that Cogent types are strictly positive

Rule SP-REC describes checking a record bound with a recursive parameter t, where parameter is added to the bound set $\mathcal{B}$, and removed from the non-strictly positive set $\mathcal{S}$ (as these variables can shadow previous variables), where $\mathbf{RP}(\mu)$ returns the recursive parameter variable in μ if the μ has one.

Rule SP-FUNC describes checking a function type, where checking the argument position τ_1 adds all currently bound ($\mathcal{B}$) parameters to the non-strictly positive set ($\mathcal{S}$), and then checking the result position τ_2.

Rule SP-RVAR describes checking a recursive parameter variable t_r, requiring that it is not currently in a non-strictly positive position (i.e. not in the $\mathcal{S}$ set).

SP-VAR merely descends on the types nested within a variant, SP-PRIM allows for primitive types ($\mathcal{T}$) and SP-TVAR allows for type variables (a).

Once strictly positive typing is checked, the reorganiser will embed all used recursive parameter variables with a reference to the recursive boxed record that they were bound to, which is used during the constraint solving phase.

3.3 Typing and Constraint Generation Rules

Our typing rules and Constraint generation rules have only been changed only to check one extra constraint on records - that recursive parameters are only used on boxed records.

$$\frac{\mu = \varepsilon}{\mu\ \textcircled{u}\ \textbf{UnboxedNotRecursive}}$$

$$\frac{s \neq \textcircled{u}}{(\textbf{mu}\ t)\ \text{s}\ \textbf{UnboxedNotRecursive}}$$

Figure 3.4: A constraint checking that recursive parameters are used only with boxed records

$$\frac{\begin{array}{c} A;\Gamma \vdash e2 : \tau_k \qquad A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad s \neq \textcircled{r} \\ A;\Gamma_1 \vdash e_1 \ : \mu\ \{\overline{f_i^u : \tau_i}, f_k^\bullet : \tau_k\}\ s \\ A \vdash \mu\ s\ \textbf{UnboxedNotRecursive} \end{array}}{A;\Gamma \vdash \mathsf{put}\ e_1.f_k = e_2 : \ \mu\ \{\overline{f_i^u : \tau_i}, f_k^\circ : \tau_k\} s}\ \textsc{Put}$$

Figure 3.5: The updated typing rule for PUT

$$\frac{\begin{array}{c} A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad A;\Gamma_1 \vdash e_1 \ : \mu\ \{\overline{f_i^u : \tau_i}, f_k^\circ : \tau_k\}\ s \\ A \vdash \ \mu\ \ s\ \textbf{UnboxedNotRecursive} \qquad s \neq \textcircled{r} \\ A; r : \mu\ \{\overline{f_i^u : \tau_i}, f_k^\bullet : \tau_k\}\ s,\ y : \tau_k,\ \Gamma_2 \vdash e_2 : \tau \end{array}}{A;\Gamma \vdash \mathsf{take}\ r\ \{f_k = y\} = e1\ \mathsf{in}\ e_2 : \tau}\ \textsc{Take}$$

Figure 3.6: The updated typing rule for TAKE

$$\frac{A;\Gamma \vdash e : \mu\ \{\overline{f_i^\bullet : \tau_i}, f_k^\circ : \tau_k\}\ s \qquad A \vdash \mu\ s\ \textbf{UnboxedNotRecursive}}{A;\Gamma \vdash e.f_k : \tau}\ \textsc{Member}$$

Figure 3.7: The updated typing rule for MEMBER

$$\frac{\begin{array}{c} \alpha, \beta, \gamma\ \text{fresh} \qquad G \vdash e : \gamma\ \{f^\circ : \tau | \alpha\}\ \beta \rightsquigarrow G' \mid C_1 \\ C_2 = \gamma\ \{f^\bullet : \tau | \alpha\}\ \beta\ \textbf{Drop} \qquad C_3 = \gamma\ \beta\ \textbf{UnboxedNotRecursive} \end{array}}{G \vdash e.f : \tau \rightsquigarrow G' \mid C_1 \wedge C_2 \wedge C_3}\ \text{CG-}\textsc{Member}$$

Figure 3.8: The updated constraint generation rule for MEMBER

$$\frac{\begin{array}{c}\alpha, \beta, \gamma, \delta \text{ fresh} \quad G_1 \vdash e : \delta\ \{f^\circ : \beta \mid \alpha\}\ \gamma \rightsquigarrow G_2 \mid C_1 \\ x :_{\langle 0 \rangle} \delta\{f^\bullet : \beta|\alpha\}\gamma,\ y :_{\langle 0 \rangle} \beta,\ G_2 \vdash e_2 : \tau \rightsquigarrow x :_{\langle n \rangle} \delta\{f^\bullet : \beta|\alpha\}\gamma,\ y :_{\langle m \rangle} \beta,\ G_3 \mid C_2 \\ C_3 = \text{if } n = 0 \text{ then } \delta\ \{f^\bullet : \beta \mid \alpha\}\ \gamma\ \textbf{Drop} \text{ else } \top \\ C_4 = \text{if } m = 0 \text{ then } \beta\ \textbf{Drop} \text{ else } \top \\ C_5 = \gamma \neq \textcircled{r} \qquad C_6 = \delta\ \gamma\ \textbf{UnboxedNotRecursive}\end{array}}{G_1 \vdash \mathsf{take}\ x\ \{f = y\} = e_1\ \mathsf{in}\ e_2 : \tau \rightsquigarrow G_3 | \bigwedge_{k \in 1..6} C_k}\ \text{CG-Take}$$

Figure 3.9: The updated constraint generation rule for Take

$$\frac{\begin{array}{c}\alpha, \beta, \gamma, \delta \text{ fresh} \qquad G_1 \vdash e_1 : \delta\ \{f^\bullet : \tau|\alpha\}\ \gamma \rightsquigarrow G_2 \mid C_1 \\ G_2 \vdash e_2 : \beta \rightsquigarrow G_3 \mid C_2 \qquad C_3 = \delta\ \{f^\circ : \tau|\alpha\}\ \gamma \sqsubseteq \tau \qquad C_4 = \gamma \neq \textcircled{r} \\ C_5 = \delta\ \gamma\ \textbf{UnboxedNotRecursive}\end{array}}{G_1 \vdash \mathsf{put}\ e_1.f = e_2 : \tau \rightsquigarrow G_3 \mid \bigwedge_{k \in 1..5} C_k}\ \text{CG-Put}$$

Figure 3.10: The updated constraint generation rule for Put

The check for this new constraint is outlined in Figure 3.4

We modify the original Take, Put and Member typing rules defined by O'Connor [4] to include this constraint as presented in Figure 3.5, Figure 3.6 and Figure 3.7, with our differences highlighted in blue.

Next, we modify the CG-Member, CG-Take and CG-Put to account for constraint generation. Our new constraints will be generated as defined by Figure 3.9, Figure 3.10 and Figure 3.8, to also include this new constraint, with our differences again highlighted in blue.

Each of these rules additionally now includes an extra fresh unification variable, which is used to infer the recursive parameter of a given record. These variables are substituted for real recursive parameters during the constraint solver's *unification* phase.

$$t_r \sqsubseteq \tau \overset{\text{simp}}{\hookrightarrow} \textbf{unroll}\ t_r \sqsubseteq \tau \tag{3.1}$$

$$\tau \sqsubseteq t_r \overset{\text{simp}}{\hookrightarrow} \tau \sqsubseteq \textbf{unroll}\ t_r \tag{3.2}$$

$$t_r \eqsim \tau \overset{\text{simp}}{\hookrightarrow} \textbf{unroll}\ t_r \eqsim \tau \tag{3.3}$$

$$\tau \eqsim t_r \overset{\text{simp}}{\hookrightarrow} \tau \eqsim \textbf{unroll}\ t_r \tag{3.4}$$

$$t_r \eqsim k_r \overset{\text{simp}}{\hookrightarrow} \textbf{unroll}\ t_r = \textbf{unroll}\ k_r \tag{3.5}$$

$$t_r \sqsubseteq k_r \overset{\text{simp}}{\hookrightarrow} \textbf{unroll}\ t_r = \textbf{unroll}\ k_r \tag{3.6}$$

$$(\textbf{mu}\ t)\ s\ \textbf{UnboxedNotRecursive} \overset{\text{simp}}{\hookrightarrow} s \neq \textcircled{u} \tag{3.7}$$

$$\varepsilon\ \textcircled{u}\ \textbf{UnboxedNotRecursive} \overset{\text{simp}}{\hookrightarrow} \varepsilon \tag{3.8}$$

Figure 3.11: The new simplification rules

$$\textbf{unroll}\ t_r = \textbf{mu}\ t\ \{\overline{f_i^u : \tau_i}\}$$

Figure 3.12: the **unroll** operator, which expands a recursive parameter to its recursive reference

3.4 Constraint Solver Rules

The constraint solver now contains additional rules in order to reason about recursive records, which have changed the *simplification* and *unification* constraint solving phases.

3.4.1 Simplification Phase

In the simplification phase, the rules described in Figure 3.11 allow the solver to reason about recursive types that feature in the comparison constraints subtyping ($\sqsubseteq$) and type equality ($\eqsim$). Equation 3.1 through to Equation 3.4 state that when a recursive type t_r is directly compared against a type τ, the solver can **unroll** t_r, as defined in Figure 3.12. Additionally, Equation 3.5 and Equation 3.6 cover the case where we compare two recursive parameters t_r and k_r, where **unroll**ing both parameters results in the same type via meta equality.

Equation 3.7 and Equation 3.6 allow for the elimination of **UnboxedNotRecursive**

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

Figure 3.13: A polymorphic list datatype in Minigent

constraints given that either the recursive parameter is ε and the sigil is unboxed (ⓤ), or that the recursive parameter is some **mu** t and the accompanying sigil s is readonly (ⓡ) or writable (ⓦ).

3.4.2 Unification Phase

The unification phase now unifies unknown recursive parameters that are embedded on records. Given constraints of the form $\alpha\ \{\dots\} \sqsubseteq \mu\ \{\dots\}$ or of the form $\alpha\ \{\dots\} \approx \mu\ \{\dots\}$, where α is a unification variable and μ is a concrete recursive parameter, we can simply replace all α with μ without an occurs check, and add the substitution $\alpha := \mu$ to our set of assignments that the solver will output. This is true symmetrically, when the left and right hand sides of the above constraints are swapped.

The unification phase also unifies recursive parameters in an **UnboxedNotRecursive** constraint to ε if the sigil in that constraint has been unified to an unboxed sigil (ⓤ). If this unification variable was meant to be a recursive parameter t_r, then the constraint solving will fail elsewhere due to this substitution, preserving the correctness of the solver.

3.5 Example Usage and Testing

We present a list datatype in Figure 3.13, which was implemented in Minigent during the course of testing our implmenetation. As Minigent's syntax is not as rich as the full compiler's, our definition differs from that of Figure 3.2, as we are unable to create type aliases (i.e. **type** List = ...) and unable to use tuples, which are replaced with an

```
sumList : mu t { l :< Nil Unit | Cons { data : U32, rest : t! }# >}! → U32;
sumList r =
  take r' {l = l} = r in
    case l of
      Nil u → 0
      | v2 →
        case v2 of
          Cons s →
            take s' {rest = x} = s in
              s.data + sumList x
            end
        end
    end
  end
```

Figure 3.14: A function that sums a list of 32 bit integers

unboxed record instead.

A simple use of this list can be seen in Figure 3.14, which sums a list of 32 bit integers (U32). Our recursive call is on a variable x, which has the linear type $t!$, which the compiler will unroll and then propagate the bang (!) through the unrolled type via the normaliser.

We can write common functions that operate on lists, such as map, as in Figure 3.15. While our implementation is very verbose in Minigent, the full compiler's syntactic sugar would allow for a much more simplified implementation with tuples and more compact case statements.

It can be seen in our recursive call to map, we pass *remaining2.rest* and *fun* as our arguments, being the tail and mapping function for the list respectively. As *remaining2.rest* has the type of the recursive parameter for the list t, our solver unrolls this variable as described in order to ensure the recursive call has been given expressions of the correct type.

Furthermore, we can create a tree datatype in Minigent as shown in Figure 3.16. The tree type resembles our list datatype but with two fields that have the type of our mu quantified recursive parameter t, the left and right branches of the tree.

```
alloc : [x]. Unit → mu t { l :< Nil Unit | Cons { data : x, rest : t }# > take };

map : [a, b].
        { list : mu t { l :< Nil Unit | Cons { data : a, rest : t }# >}!, f : (a- > b) }#
        → mu t { l :< Nil Unit | Cons { data : a, rest : t }# >};
map l =
  take l2 { list = node } = l in
    take l3 { f = fun } = l2 in
      take node2 { l = head }  = node in
        case head of
          Nil u →
            let newNode = alloc Unit in
              put newNode.l := Nil Unit end
            end
          | v2 →
            case v2 of
              Cons remaining →
                take remaining2 { data = x } = remaining in
                  let newNode = alloc Unit in
                    put newNode.l :=
                          Cons {
                            data = funx,
                            rest = map { list = remaining2.rest, f = fun}
                          } end
                    end
                  end
        end
      end
    end
  end;
```

Figure 3.15: Map for lists implemented in Minigent

```
[a]. mu t {
      f :<
            Leaf Unit
          | Node { left : t, right : t, data : a}#
      >
}
```

Figure 3.16: A polymorphic tree datatype in Minigent

3.6 Termination

A primitive recursion detection scheme has been designed to check the termination of functions that structurally recurse on our new recursive types. This introduces a new phase into the compiler; the *termination checker*.

3.6.1 Design

Our new termination checker works by generating assertions about the structural size of variables passed to the function, similar to the approach taken by Abel [15]. When recursive calls are made, the termination checker uses these assertions to check that the structural size of the argument to the recursive call is strictly smaller than that of the function argument. Hence, the argument given to a recursive call will only ever grow smaller, and therefore the function will eventually reach the *bottom* of the structure and terminate. As all recursive types constructed must be finite due to our use of strictly positive types and Cogent's linear type system (i.e. no infinitely looping structures), all primitive recursive calls will terminate.

Our size assertions take the form of two relations between variables, size inequality ($\prec$) and size equality ($\approx$). $x \prec y$ is the assertion that a variable x is strictly structurally smaller than a variable y and, $x \approx y$ is the assertion that a variable x is structurally equal in size to a variable y.

The checker starts with a set of known termination assertions G, and then adds extra assertions by analysing the expression body of a function to produce a stronger set of assertions G', as well as a set of variables that are arguments to recursive calls in the function, C, which we call *goal* variables. We wish to show that every argument to a recursive function call (i.e. every variable in C, our goals) is smaller than the argument to the function. Figure 3.21 describes the condition for soundness of our termination checker.

For our check to be *complete*, our checker would need to show all non-terminating functions are invalid. As a consequence of the halting problem, however, this completeness is

$$
\begin{aligned}
E ::= x \to \alpha,\ E \quad & \text{(variable mapping)} \\
\mid \varepsilon \quad & \text{(empty)}
\end{aligned}
$$

Figure 3.17: The syntax of our environments

impossible so our checker only attempts to check for primitive recursion.

We generate such assertions by looking at Cogent's expression language where any form of structural size is added or taken away, namely take, put and case expressions. Figure 3.19 describes this assertion generation.

However, if a user were to shadow a variable in a let expression they may create a false assertion. Consider the expression let $x = K\ x$ in $\ldots$, which would produce the assertion $x \approx K\ x$, which by our structural size rules can be rewritten as $x \prec x$, which is clearly false.

We prevent this from occurring by using a fresh variable name for every introduced variable in scope, and use these fresh names in our assertions. We also store a mapping of program variables to fresh variables names in a environment E, which we update as we encounter new variable definitions in the function expression. The syntax of our environment is as presented in Figure 3.17, where $E[x]$ denotes the lookup of a variable x in E, which returns the first mapping of x found in E.

With our current method of size relations, we are unable to reason about variables compared against any general expression. For example, suppose the assertion $x \prec K\ y$. This assertion is useless to us as we have no way of reasoning about x and y directly, and we are unable to remove the constructor K as we have no information about the gap in size between x and y. We could have $x \prec y$ if x is more than one layer of structure smaller than y, or $x \approx y$ if x is only one layer of structure smaller.

We seek a way to eliminate such assertions from being generated. To do this, we define a function get in Figure 3.18 that maps an expression to a fresh variable name if the expression is a program variable and otherwise ε. Whenever we generate an assertion against an expression e, we use the result of $\mathsf{get}(E, e)$ in the assertion, which will also

$$\mathsf{get}(E, e) = \mathsf{if}\ e = t\ \mathsf{then}\ E[t]\ \mathsf{else}\ \varepsilon$$

Figure 3.18: The definition of get

insert the correct fresh variable name in the assertion.

After generating all of our assertions, we filter out any assertions that include an ε, as we cannot reason about them in any way, or fail if any goal C contains an ε. While this solution may not be as complete as it possibly could be, we may still use our remaining assertions to prove termination in some cases.

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

$$\frac{\alpha, \beta \text{ fresh} \quad E; \alpha \prec \mathsf{get}(E, e_1),\ \mathsf{get}(E, e_1) \approx \beta,\ G_1 \vdash e_1 \rightsquigarrow G_2 \mid C_1 \quad E, x \to \alpha, r' \to \beta; G_2 \vdash e_2 \rightsquigarrow G_3 \mid C_2}{E; G_1 \vdash \mathsf{take}\ r'\ \{f = x\} = e_1\ \mathsf{in}\ e_2 \rightsquigarrow G_3 \mid C_1 \cup C_2}\ \text{T-Take}$$

$$\frac{\alpha, \beta \text{ fresh} \quad E; \alpha \prec \beta,\ \beta \approx \mathsf{get}(E, e_1),\ \alpha \approx \mathsf{get}(E, e_2),\ G_1 \vdash e_1 \rightsquigarrow G_2 \mid C_1 \quad E; G_2 \vdash e_2 \rightsquigarrow G_3 \mid C_2}{E; G_1 \vdash \mathsf{put}\ e_1.f := e_2 \rightsquigarrow G_3 \mid C_1 \cup C_2}\ \text{T-Put}$$

$$\frac{\alpha, \beta, \gamma \text{ fresh} \quad E;\ G_1,\ \beta \approx \mathsf{get}(E, e_1) \vdash e_1 \rightsquigarrow G_2 \mid C_1 \quad E, \alpha \to x;\ G_2, \alpha \prec \beta \vdash e_2 \rightsquigarrow G_3 \mid C_2 \quad E, \gamma \to y;\ G_3, \gamma \approx \beta \vdash e_3 \rightsquigarrow G_4 \mid C_3}{E; G_1 \vdash \mathsf{case}\ e_1\ \mathsf{of}\ \kappa\ x.\ e_2\ \mathsf{else}\ y.\ e_3 \rightsquigarrow G_4 \mid C_1 \cup C_2 \cup C_3}\ \text{T-Case}$$

$$\frac{\alpha, \beta \text{ fresh} \quad E;\ G_1,\ \beta \approx \mathsf{get}(E, e_1) \vdash e_1 \rightsquigarrow G_2 \mid C_1 \quad E, \alpha \to x;\ G_2, \alpha \prec \beta \vdash e_2 \rightsquigarrow G_3 \mid C_2}{E; G_1 \vdash \mathsf{case}\ e_1\ \mathsf{of}\ \kappa\ x.\ e_2\ \rightsquigarrow G_3 \mid C_1 \cup C_2}\ \text{T-CaseIrr}$$

$$\frac{\alpha \text{ fresh} \quad E; G_1 \vdash e_1 \rightsquigarrow G_2 \mid C_1 \quad E, x \to \alpha; \alpha \approx \mathsf{get}(E, e_1),\ G_2 \vdash e_2 \rightsquigarrow G_3 \mid C_2}{E; G_1 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow G_3 \mid C_1 \cup C_2}\ \text{T-Let}$$

Figure 3.19: Termination assertion generation rules

$$\frac{E; G_1 \vdash e_1 \rightsquigarrow G_2 \mid C}{E; G_1 \vdash e.f \rightsquigarrow G_2 \mid C}\ \textsc{T-Member}$$

$$\frac{E; G_1 \vdash e_1 \rightsquigarrow G_2 \mid C}{E; G_1 \vdash \kappa\ e \rightsquigarrow G_2 \mid C}\ \textsc{T-Con} \qquad \frac{}{E; G \vdash f[\vec{\tau_i}] \rightsquigarrow G \mid E \mid \varnothing}\ \textsc{T-TApp}$$

$$\frac{E; G_1 \vdash e_1 \rightsquigarrow G_2 \mid C_1 \quad E_2; G_2 \vdash e_2 \rightsquigarrow G_3 \mid C_2}{E; G_1 \vdash e_1\ e_2 \rightsquigarrow G_3 \mid C_1 \cup C_2 \cup \{\mathsf{get}(E, e_2)\}}\ \textsc{T-FApp}$$

$$\frac{}{E; G \vdash x \rightsquigarrow G \mid E \mid \varnothing}\ \textsc{T-TVar} \qquad \frac{}{E; G \vdash \ell \rightsquigarrow G \mid E \mid \varnothing}\ \textsc{T-Lit}$$

$$\frac{E; G_1 \vdash e_1 \rightsquigarrow G_2 \mid C_1 \quad E_2; G_2 \vdash e_2 \rightsquigarrow G_3 \mid C_2}{E; G_1 \vdash e_1 * e_2 \rightsquigarrow G_3 \mid C_1 \cup C_2}\ \textsc{T-PrimOp}$$

$$\frac{\begin{array}{c} E; G_1 \vdash e_1 \rightsquigarrow G_2 \mid C_1 \\ E; G_2 \vdash e_2 \rightsquigarrow G_3 \mid C_2 \quad E; G_3 \vdash e_3 \rightsquigarrow G_4 \mid C_3 \end{array}}{E; G_1 \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rightsquigarrow G_4 \mid C_1 \cup C_2 \cup C_3}\ \textsc{T-If}$$

$$\frac{E; G_1 \vdash e \rightsquigarrow G_2 \mid C}{E; G_1 \vdash e :: \tau \rightsquigarrow G_2 \mid C}\ \textsc{T-Tysig}$$

$$\frac{\text{for all } i \in \{1, \ldots, n\} \quad E; G_i \vdash e_i \rightsquigarrow G_{i+1} \mid C_i}{E; G_1 \vdash \{\overline{f_i = e_i}\} \rightsquigarrow G_{n+1} \mid \bigcup_{k=1}^{n} C_k}\ \textsc{T-Struct}$$

Figure 3.20: Termination assertion generation rules, continued from Figure 3.19

$f\ x = e$	Given a function definition
$\implies\ \alpha \leftarrow x, \varepsilon;\ \varnothing \vdash e \rightsquigarrow G' \mid C$	And a set of assertions and goals generated by the termination checker from an initial environment and empty set of assertions
$\implies\ G' \vdash \forall \beta \in C.\ \beta \prec \alpha$	The function terminates if every goal expression produced is smaller than the function argument under the produced assertions

Figure 3.21: The termination checker soundness condition

```
sumList r =                                          E1 = α ← r; G1 = ∅
                                                                (Initialisation)

  take r' {l = l} = r in                             E2, = E1, β ← r', γ ← l;
                                                       G2 = G1, α ≈ β, γ ≺ β
                                                                (T-Take)

    case l of
      Nil u → 0
      | v2 →                                         E3 = E2, u ← δ, η ← v2;
                                                       G3 = G2, δ ≺ γ, η ≈ γ
                                                                (T-Case)

        case v2 of
          Cons s →                                     E4 = E3, φ ← s
                                                       G4 = G3, φ ≺ η
                                                                (T-CaseIrr)

            take s' {rest = r} = s in                  E5 = E4, ψ ← s', π ← r;
                                                       G5 = G4, ψ ≈ φ, π ≺ ψ
                                                                (T-Take)

              s.data + sumList r                       C = {π}
                                                                (T-FApp)

            end
        end
    end
end;
```

Final termination assertion generator output:

$$G' = \{\ \alpha \approx \beta,\ \gamma \prec \beta,\ \delta \prec \gamma,\ \eta \approx \gamma,\ \phi \prec \eta,\ \psi \approx \phi,\ \pi \prec \psi\ \}$$

$$C = \{\pi\}$$

Figure 3.22: An example of generated termination assertions on a program that sums a list of integers

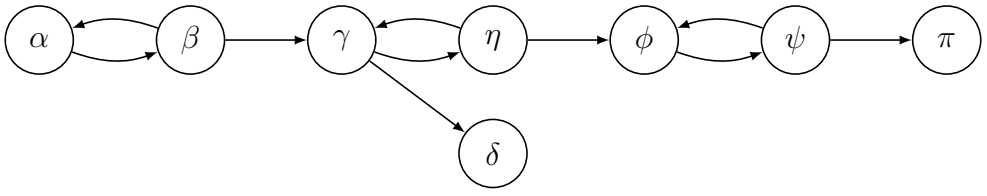

Figure 3.23: The graph constructed from the assertions generated in Figure 3.22

To determine whether our function terminates, we construct a directed graph from our assertions, where variable names are vertices and edges are the relations between them. For example, $\alpha \prec \beta$ would produce a directed edge from β to α, and $\alpha \approx \beta$ would produce a bidirectional edge between α and β. To show termination, we show that there is a one-way path from the argument node to every goal node.

As an example, consider our earlier list sum example in Figure 3.22 that features inline generated termination constraints. Note that our program also shadows the variable r in the last `take` statement, which the termination rules generate a different fresh name for. Using our final set of assertions G' and our goal set C we must show that $\pi \prec \alpha$. Using our assertions we construct the graph in Figure 3.23 and observe the following path from α to π:

$$\alpha \to \beta \to \gamma \to \eta \to \phi \to \psi \to \pi$$

We also observe that there are no outgoing edges from π, therefore there is no return path to α and hence our path is one-way. Thus, our example function terminates.

3.6.2 Possible Future Improvements

Earlier, we discussed the problem of only being able to reason about expressions that are program variables. Our partial solution currently simply eliminates the assertions that do not reason solely about program variables, however we seek a more complete way to solve this problem.

One potential idea is to eliminate structural size inequality ($\prec$), and preserve relative size information by using only discrete size changes. Consider our earlier example $x \prec K\ y$, where we were unable to remove the constructor C from y. If earlier when generating this assertion we wrote $x - 1 \approx K\ y$, where $x - 1$ means '1 structural size less than x', we would be able to know exactly how much structurally smaller x is than y. If we wanted to simplify this assertion, we could do so by removing the constructor K from y, and then

taking away one structural size from the LHS of the assertion, leaving $x - 2 \approx y$.

Not only does this allow us to use extra assertions, but we may still determine if our function terminates by showing existence of a one-way path from the input argument to recursive call arguments. We simply normalise all equations into the form $x - C \approx y$, where x and y are variables and C is a constant, and then create a directed path from y to x if $C \neq 0$, and a bidirectional path between x and y if $C = 0$.

Unfortunately, Cogent integer literals U8, U16, U32 and U64 are not structural types, and hence terminating recursion on integers cannot be detected via our current method. Furthermore, as unsigned integer overflow is permitted in our Cogent programs, even basic structural integer recursion could produce a non-terminating program, as a program that takes away from an integer recursively with no base case can infinitely underflow.

A potential solution to this problem could be to map unsigned integers and literals to a Peano natural number representation, and require that each program has an expression with a base case for any recursive call on integers, and performs recursion by taking only one constructor away per recursive call, so it does not *skip* a base case.

As a thesis extension, we have defined termination checking and provided a pen-and-paper formalisation. As future work, termination checking can be the adapted and implemented in the full compiler and in Minigent, and then the generation of an Isabelle/HOL termination proof from the results of the termination checker.

3.7 Conclusion

Our proposed work has satisfied our three major requirements: type-safe recursive types, maintaining the existing type system benefits that Cogent's type system provides, and ease of showing termination.

Our design allows for the flexible creation of recursive types on top of Cogent's existing boxed records, which prevent type-incorrect programs from being permitted by Cogent's constraint solver. Our examples show that we can create data structures such as lists

and trees, and implement functions such as `map` to operate on them. Cogent's uniqueness type guarantees are maintained as our recursive types do not alter the existing constraint generation rules or solver rules, but rather only add more constraints. This will in future allow the optimisations and high level embedding that Cogent currently has in the main compiler to remain unchanged when porting recursive types from Minigent.

Our extension work for formalising a termination check within Cogent has set the groundwork to ensure that Cogent programs are total and terminating. While the design has room for improvement, the current formalised rules already provide the basis for an implementation in Minigent and then in the full compiler. Using an opt-out mechanism in the language, programmers would be allowed to express more complex functions if they choose while being reminded of potential non termination by the compiler, giving them an increased awareness of the termination proof awaiting in the embedding.

Bibliography

[1] David Greenaway. "Automated proof-producing abstraction of C code". PhD thesis. University of New South Wales, Sydney, Australia, 2014. URL: http://handle.unsw.edu.au/1959.4/54260.

[2] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *A Proof Assistant for Higher-Order Logic*. 2018. URL: https://isabelle.in.tum.de/doc/tutorial.pdf.

[3] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby C. Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. "Refinement through restraint: bringing down the cost of verification". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 2016, pp. 89–102. DOI: 10.1145/2951913.2951940. URL: https://doi.org/10.1145/2951913.2951940.

[4] Liam O'Connor. *Type Systems for Systems Types*. 2019.

[5] Alexander Krauss. *Defining Recursive Functions in Isabelle HOL*. URL: https://isabelle.in.tum.de/doc/functions.pdf.

[6] Philip Wadler. "Recursive types for free!" In: (1990). URL: https://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt.

[7] Ulf Norell, Andreas Abel, Nils Anders Danielsson, and Makoto Takeyama et al. *Agda Documentation, Data Types, Strict Positivity*. 2016. URL: https://agda.readthedocs.io/en/v2.5.2/language/data-types.html#strict-positivity.

[8] Inria. *Calculus of Inductive Constructions*. 2018. URL: https://coq.inria.fr/doc/language/cic.html#strict-positivity.

[9] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle's Logics: HOL*. 2009. URL: https://isabelle.in.tum.de/website-Isabelle2009-1/dist/Isabelle/doc/logics-HOL.pdf.

[10] Thierry Coquand and Christine Paulin. "Inductively defined types". In: *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*. 1988, pp. 50–66. DOI: 10.1007/3-540-52335-9_47. URL: https://doi.org/10.1007/3-540-52335-9%5C_47.

[11] Loic Colson. "About Primitive Recursive Algorithms". In: *Theor. Comput. Sci.* 83.1 (1991), pp. 57–69. DOI: 10.1016/0304-3975(91)90039-5. URL: https://doi.org/10.1016/0304-3975(91)90039-5.

[12] Wilhelm Ackermann. "Zum Hilbertschen Aufbau der reellen Zahlen". In: *Mathematische Annalen* 99.1 (Dec. 1928), pp. 118–133. ISSN: 1432-1807. DOI: 10.1007/BF01459088. URL: https://doi.org/10.1007/BF01459088.

[13] Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola. "Structures for structural recursion". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 2015, pp. 127–139. DOI: 10.1145/2784731.2784762. URL: https://doi.org/10.1145/2784731.2784762.

[14] Andreas Abel and Thorsten Altenkirch. "A predicative analysis of structural recursion". In: *J. Funct. Program.* 12.1 (2002), pp. 1–41. DOI: 10.1017/S0956796801004191. URL: https://doi.org/10.1017/S0956796801004191.

[15] Andreas Abel. "foetus – Termination Checker for Simple Functional Programs". Programming Lab Report. 1998. URL: http://www.tcs.informatik.uni-muenchen.de/%5C~%7B%7Dabel/foetus/.

[16] David Walker. *Substructural Type Systems*. Apr. 2019. URL: http://mitp-content-server.mit.edu:18180/books/content/sectbyfn?collid=books_pres_0&id=1104&fn=9780262162289_sch_0001.pdf.

[17] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Trans. Amer. Math. Soc.* 74 (1953), pp. 358–366.

[18] Philip Wadler. "Linear Types can Change the World!" In: *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*. 1990, p. 561.