School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Termination Checker for Recursive Types in Cogent

by

Lucy Qiu

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: December 10, 2020

Supervisor: Christine Rizkallah

Student ID: z5113955

Acknowledgements

Thank you to Liam O'Connor and Christine Rizkallah for being great educators and introducing me to formal methods. This thesis is inspired from previous work on the COGENT language by Emmet Murray; thank you for your optimism and support throughout my thesis.

I would like to express my deepest gratitude to Christine Rizkallah and Vincent Jackson for their guidance, insights, ideas and feedback throughout the year. This thesis would not have been completed without you.

Thank you to my wonderful friends for their support and companionship throughout the year; Lucy Qu, Blaise, Nancy, Yuseph, Mark, Flora, Rui and many others. Thanks especially to Lucy and Blaise for their empathy, humour and encouragement.

Finally, I would like to express my eternal gratitude to my parents for their warmth, kindness and guidance.

Abstract

COGENT is a linearly-typed functional programming language developed in Haskell for implementing trustworthy and efficient systems code. Recently, a restricted form of recursion was added to COGENT. To ensure that COGENT only permits terminating functions, there exists a basic termination checker that permits a small subset of terminating recursive functions. This project aims to implement a more permissive termination checker for recursive types in COGENT using techniques of structural recursion and lexicographic ordering.

Contents

Chapter 1

Introduction

COGENT (O'Connor, 2019) is a purely functional programming language designed for low level systems programming (Amani et al., 2016). The high-level COGENT language is compiled to a low-level language (C) with an automatically generated proof of correctness (O'Connor et al., 2016).

COGENT has a linear and uniqueness type system that guarantees memory safety and, due to exhaustive pattern matching and termination checking, is total by design. MINIGENT is a stripped down version of COGENT used for experimental purposes.

This thesis aims to improve the termination checker implemented by Murray (2019) inside MINIGENT, enabling it to recognise a larger subset of terminating recursive functions. Termination checking for functional languages relies on finding a measure on function arguments where successive recursive calls decrease in size. Prevalent termination checking approaches include structural recursion (Abel and Altenkirch, 2000) and type annotation (Abel, 2010). The former searches for a termination ordering where the recursive argument decreases in structural size from the function argument. The latter includes size information as part of the language itself, and solves termination constraints in the type checker to determine termination.

This thesis presents a solution based on structural recursion with lexicographic ordering. It improves upon the previous termination checker with the introduction of *measures* that can

be applied to function arguments to analyse structural size. From the generated measures, we search for a lexicographic termination ordering. This termination checker permits a subset of the structurally recursive functions and is extensible to other classes of recursive functions.

The main work of this thesis is the *local descent* algorithm, which attempts to prove that measures decrease at the recursive call. This idea can be adapted to other functional languages that do not have theorem provers (Isabelle's implementation) and perhaps provides a less complex alternative to existing implementations.

Chapter 2

Literature Review

2.1 Cogent

The motivation behind COGENT (O'Connor, 2019) is to provide engineers with a more convenient and cost effective method for verifying program correctness. COGENT features a certifying compiler that generates C code, an Isabelle/HOL embedding, and a proof that the C code refines the Isabelle/HOL embedding (O'Connor et al., 2016). Isabelle is a higher order logic (HOL) theorem prover (Nipkow et al., 2002). Refinement is a correctness preserving transformation, usually from the abstract (a specification) to the concrete (an implementation). In this case, the refinement proof provides an assurance that the results proven for the Isabelle/HOL embedding also hold for the C code.

COGENT features a linear and uniqueness type system that guarantees memory safety. A language is memory safe if invalid memory accesses are provably impossible for a well-typed program. The COGENT language is total due to exhaustive pattern matching and termination checking.

MINIGENT is a stripped down version of COGENT used for experimental purposes. Previously, a basic termination checker was implemented in MINIGENT (Murray, 2019). This thesis extends the aforementioned termination checker, enabling it to recognise a larger subset of terminating recursive functions.

2.1.1 Linear and Uniqueness Types

COGENT uses a uniqueness type system, a kind of linear type system (Wadler, 1990a). In a linear type system, variables of linear type must be used exactly once. Uniqueness types apply this restriction globally so that throughout the lifetime of a linear-typed object, there is exactly one unique writeable reference to it. This ensures that the programmer frees all allocated memory without relying on run-time support such as garbage collection.

In COGENT, objects are either boxed (on the heap) or unboxed (on the stack). Linearity and uniqueness type constraints are only applied to writeable, boxed values as these require memory management and may be destructively mutated.

In the COGENT syntax, a hash (`#`) behind a record indicates that it is unboxed. A bang (!) behind a record indicates that a it is read-only.

2.1.2 Type System

COGENT's type system consists of primitive types, type constructors for variants, records and recursive types.

Primitive Types

COGENT's primitive types consist of Boolean types, the four unsigned integer types `U8, U16, U32, U64` and the unit type `()`, a single trivial inhabitant. Each of these primitive types are of fixed size and are unboxed.

Variants

Variants are n-ary sum types. In COGENT, variants consist of a set of constructor names paired with types. The equivalent of Haskell's Maybe type is written in COGENT as:

```
1 type Maybe a = < Nothing | Just a >
```

Records

Records types are analogous to C structures; they consist of a set of named fields. A record may be boxed (stored on the heap) or unboxed (stored on the stack). The record `Position` stores x and y coordinates and a Boolean indicating if the position contains a mine or not:

```
1 type Position = {x: U8, y: U8, mine: Bool}
```

Recall that COGENT has a uniqueness type system, a kind of linear type system. Variables of linear type have exactly one unique writable reference and must be used exactly once. If a record is writeable, boxed, or contains linear fields, it is linear and must obey the corresponding restrictions. Otherwise, if a record is readonly or unboxed with no linear fields, it is not linear and can be freely shared or discarded.

Linearly typed variables are used exactly once. Hence, when a linearly typed record is mutated, it is used and a new record must be introduced with the updated value. The two mutating operations we can perform on a record are `take`, which removes (uses) a field, and `put`, which assigns a field.

Recursive Types

A recursive type is a type that may reference itself. In COGENT, recursive types are implemented using records. Murray (2019) extends the record syntax introduced by O'Connor (2019) with the recursive parameter `rec` to describe recursive record types in COGENT:

$\tau ::= \text{rec } t \ \{\overline{f_i^u \tau_i}\}$

where f_i^u describes the i^{th} field of the record with usage tag u, which is of type τ_i. A usage tag is a boolean that indicates whether the corresponding field is taken or present. The $\overline{\text{overline}}$ indicates a set; the order of the fields is unimportant. Recursive types are always boxed and must obey linearity restrictions.

For a concrete example, consider the recursive list type in COGENT:

```
type List = rec t { l : < Nil Unit | Cons (t, U32) > }
```

The recursive parameter `t` is denoted by the keyword `rec`. The `List` data type has one field `l`, which contains a variant constructed by either `Nil` or `Cons`. The usage tag changes depending on the operations performed on the list. For example, a list with one element would have usage tag 'present'. If that element is removed via the `take` operator, the usage tag becomes 'taken'. If an element is inserted via the `put` operator, the usage tag once again becomes 'present'.

2.1.3 Type Inference

COGENT features constraint-based type inference involving a constraint generator and a constraint solver (O'Connor, 2019). The constraint generator takes a context, term and type and outputs constraints describing the relationship between types. A type may contain unknowns or unification variables that are to be determined by the constraint solver. The constraint solver takes a set of constraints and a set of axioms about type variables. Given that the original constraint set was satisfiable, there should remain a satisfying assignment to each unification variable.

2.2 Recursion

Before elaborating on termination checking strategies, it is useful to have a brief discussion on the definitions and properties of recursive types and recursive functions.

2.2.1 Operators on recursive types

The operators **roll** and **unroll**, also classically referred to as *fold* and *unfold*, are used to manipulate recursive types.

roll constructs a recursive type from a recursive expression by placing it into the recursive form specified by the definition.

unroll performs one unfolding of a recursive type by substituting the recursive parameter into the body of the recursive type. The unrolling of a recursive type $rec\, t.\, \tau$ is the type derived by replacing all instances of τ by itself, $rec\, t.\, \tau$.

The standard static semantics for the roll and unroll operators are:

$$\frac{\Gamma \vdash e : \tau\ [t \coloneqq \text{rec } t.\ \tau]}{\Gamma \vdash \textbf{roll}\ e : \text{rec } t.\ \tau} \qquad \frac{\Gamma \vdash e : \text{ rec } t.\ \tau}{\Gamma \vdash \textbf{unroll}\ e : \ \tau\ [t \coloneqq \text{rec } t.\ \tau]}$$

A concrete example of unroll performed on a list of integers:

$$\text{datatype } list = \text{rec } t.\ 1 + (Int\ \times\ t)$$

$$\text{unroll } list = 1 + (Int \times \text{rec } t.\ 1 + (Int\ \times\ t))$$

$$\text{unroll } (\ \text{unroll } list) = 1 + (Int\ \times (1 + (Int \times \text{rec } t.\ 1 + (Int\ \times\ t))))$$

Each unroll operator performs one unfolding of the recursive data type via substitution.

In COGENT specifically, the roll and unroll operators are defined as:

$$\textbf{roll } \text{rec } t.\ \{\overline{f_i^u \tau_i}\} = t_r$$

$$\textbf{unroll } t_r = \text{rec } t.\ \{\overline{f_i^u \tau_i}\}$$

Where t_r is a recursive type, and rec t $\{\overline{f_i^u \tau_i}\}$ is the recursive type expanded once.

Recursive types are implemented by tagging records with a recursive parameter, meaning that recursion can only happen on records.

2.2.2 Types of Recursive Functions

There are various types of recursive functions, some of which are easier to prove terminating than others.

Primitive Recursive Functions

Colson (1991) describes a primitive recursive function as one that be constructed from the primitive recursive combinators; the constant function 0, the successor function *succ*, the projection function $\pi_i^n(x_0, ..., x_i, ..., x_n) = x_i$, the composition function $S_m^n(f; c_1, ..., c_n) = f(c1, ..., c_n)$ and the primitive recursive combinator $Rec(b, s)$.

If a function can be expressed using these primitive recursive combinators, then it is guaranteed to be terminating. Many of the recursive functions that we encounter in ordinary mathematics and programming are primitive recursive; identity, min, max, bounded sums and products, primality.

Structurally Recursive Functions

Structural recursion is induction over recursive data types, such as linked lists and trees. In COGENT, structural recursion is performed over recursive types implemented using records. In a structurally recursive function, at least one 'structural size' is removed with each recursive call. Successive recursive calls eventually reach the bottom of the data structure and terminate. This means that structurally recursive functions are terminating, by definition.

The `sumList` function takes in a list of integers and outputs the sum of the list. The recursive call has argument `xs`, which is one structural size smaller than the input argument `x:xs`.

```
1  sumList :: [Int] -> Int
2  sumList [] = 0
3  sumList (x:xs) = x + sumList xs
```

With structural recursion, we can define functions outside of the set of primitive recursive func-

tions, such as the Ackermann function. The Ackermann function operates over natural numbers defined by `zero` and `suc`. Here, either the first argument decreases, or it stays the same and the second argument decreases.

```
1  ack :: Nat -> Nat -> Nat
2  ack zero m = suc m
3  ack (suc n) zero = ack n (suc zero)
4  ack (suc n) (suc m) = ack n (ack (suc n) m)
```

Partial Recursive Functions

The class of partial recursive functions coincides with the class of Turing-computable and λ-definable functions. They cannot be obtained via the primitive recursive combinators. Attempting to check functions for termination in this class is impossible as a result of the halting problem.

This thesis will focus on the class of structurally recursive functions. In Chapter 4 on future work, we provide suggestions describing how the termination checker can be extended beyond structurally recursive functions.

2.3 Termination Checking Techniques

The two main approaches utilised in implementing termination checkers are structural recursion and type annotations. Methods that analyse the program for structural recursion are implemented in ESFP (Telford and Turner, 2000), Twelf, and the *foetus* (Abel, 2010) termination checker (later implemented in an older version of Agda). Methods using type annotations are implemented in Applied Type System (Xi, 2004), MiniAgda (Abel, 2010) and, recently, Agda.

2.3.1 Structural Recursion

In most functional programming languages, recursive functions are defined using pattern matching. Abel and Altenkirch (2000) describe the requirements necessary for these recursive functions

to be terminating. First, patterns must be exhaustive and mutually exclusive. In general, and in COGENT, this is guaranteed by the type system of a language. Second, the function must be *wellfounded*. To define wellfounded recursion, we start with a wellfounded relation. A relation on a set A is wellfounded if every non-empty subset of A has a *least element* with respect to this relation. The idea of a wellfounded relation in mathematics can be mapped to wellfounded recursion, where infinite nested recursive calls are impossible due to the existence of a base case, or minimum element. Wellfoundedness can be ensured if a *termination ordering* can be given for the recursive function. A termination ordering is an ordering where arguments to child recursive calls are smaller than arguments to parent recursive calls. If a termination ordering exists, the function is wellfounded and terminates.

Structural ordering is one method of finding a termination ordering. If we can measure structural sizes between expressions and prove that successive recursive calls decrease in size, we can prove termination.

The *foetus* termination checker

Abel and Altenkirch (2000) introduce a language based upon lambda calculus, *foetus*, with sum types, product types and strictly positive recursive types. The *foetus* language contains a termination checker that searches for a termination ordering in the function.

The termination checker consists of three phases:

1. **Function call extraction**: collecting function calls and structural size information.
2. **Call graph**: generate a call graph from the size relations between expressions.
3. **Termination**: search the call graph for a termination ordering.

The success of a termination checker based on structural recursion is reliant on the size information collected from the function expression. The *foetus* termination checker uses destructors to determine size relations; a recursive argument t can be shown to be structurally smaller than its parent argument x if it is derived from x using only destructors. The order of a value is decreased

by the unfold destructor, which unfolds a recursive type. Case, projection and application keep it at the same level.

The *foetus* termination checker, at its most basic level, is unable to recognise terminating recursive functions when:

1. Arguments contain constructors. This is because only destructors are considered to alter the size of an element.
2. Recursive arguments decrease in lexicographic order, as in the case of the Ackermann function. To resolve this, we need to search for a lexicographic termination ordering amongst the recursive calls.
3. The function is not structurally recursive. This is a limitation of the structural recursion method.

Measures and Lexicographic Ordering

The lexicographic termination tactic of Isabelle/HOL (Bulwahn et al., 2007) utilises the concept of *measures* to collect more size information. In Isabelle, measures are functions that map (in this case) function arguments to natural numbers. They are generated based on the *type* of an argument. Each type has different characteristics that we can measure on, so each type will generate a different set of measures.

Isabelle's termination checker works as follows:

1. Generate a set of measures to use for size analysis.
2. Apply each measure to each function call, input and recursive.
3. Compare the measures on each (input, recursive) argument pair. Attempt to prove that the measures decrease from the input argument to the recursive argument; this is termed *local descent*. Collect the results of local descent into a matrix. The rows refer to each recursive call, and the columns refer to each measure.

4. Search for a lexicographic termination ordering from the matrix to determine termination; this is termed *global descent*.

Isabelle's method solves the first two limitations of the *foetus* termination checker. By measuring on the type of each argument, we can collect information on constructors as well as destructors. By creating a matrix containing the proof results of local descent, we can search for a lexicographic termination ordering.

This method can also be adjusted for non-structurally recursive functions by including measures (perhaps manually) and solving for them.

2.3.2 Type Annotations

Type annotations (Abel, 2010) are an alternative, type-based method for termination checking where sizes are integrated directly into the type system. The programmer annotates functions with size information; this is passed into the type checker, which resolves size constraints to determine termination. If a satisfying assignment can be found for all *size unification variables*, the program terminates. As more size information is available, this method can improve upon many of the limitations present in structurally recursive methods. A type-based termination checker requires us to:

1. Integrate size arguments into the type system
2. Annotate types with explicit sizes
3. Type check the size annotations to prove termination

Type annotations may be implemented through dependent types (a type whose definition depends on the value of another type). In systems without dependent types, type annotations can be implemented via type constructors (create a new type containing a pair of type and size).

Type annotations present a more powerful termination checking method than structural recursion as more information is available and more dependencies can be recognised. This technique also scales well to higher-order constructions, including mutually recursive data types.

However, these gains are balanced out by a range of drawbacks. First, implementing sized types requires significant changes to the type system and type checker. Second, programmers are required to explicitly annotate recursive functions with size arguments. Lastly, memory is required to carry around type annotations until type checking is complete. Because of these disadvantages, we decided not to pursue the type annotation method.

2.4 COGENT's Previous Termination Checker

A method based off of structural recursion, as described Abel and Altenkirch (2000) was previously implemented in Cogent (Murray, 2019). This implementation has three parts: assertion generation, graph construction and graph traversal.

Assertion Generation

The expression body of a function is analysed to produce assertions between program variables. These assertions describe whether one program variable is less than or equal to another in terms of structural size. We determine these relationships by examining how data structures are mutated. In Cogent, recursive functions are implemented using records tagged with the recursive parameter. Mutating operations on records such as `take` (take a field from a record) and `put` (put a field into a record), indicate structural decrease and increase, respectively. Other expressions such as `let` and `case` do not change the structural size. Importantly, the implementation only gathers information on *program variables*. It does not collect assertions on any other Cogent expressions. This means that there are many terminating recursive functions that this implementation cannot recognise due to a lack of information.

Graph Construction

From the set of assertions, we construct a graph. An assertion $x < y$ generates a directed edge from y to x. An assertion $x = y$ generates a bidirectional edge between x and y.

Graph Traversal

Finally, we search the graph for a termination ordering. A one-way path from the input argument to each recursive argument indicates that the function is terminating, as we have a chain of assertions that demonstrate a strict structural decrease.

2.4.1 Limitations

This implementation cannot deduce termination when:

1. Expressions are not explicitly placed into variables.
   Functions must contain explicit `take` and `put` expressions when the record is mutated. This means that we cannot use shorthand notation for removing elements (eg. `record.field`) or constructing elements (eg. `record = {field}`).

2. Arguments contain constructors.
   The termination checker can only determine termination when there is a clear path of destructors from the input argument to the recursive argument. It cannot perform the arithmetic that would be required for arguments that contain constructors.

3. Recursive arguments decrease in lexicographic order.
   Functions such as the Ackermann function have arguments that decrease in a lexicographic order. This means that some recursive calls may not nececessarily contain a size decrease, but they lead to another recursive call that will terminate.

4. The function is integer recursive.
   This is a limitation of the structurally recursive technique; integers do not have structural size in the conventional sense. If the language models integers as Peano numbers using *zero* and *succ* (eg. Agda), then structurally recursive techniques would be adequate. However, COGENT integers are implemented like C integers and thus can overflow and underflow. This means that any function performing recursion on integers would not be well-founded.

Chapter 3

Completed Work

3.1 Termination Checker Algorithm

The current termination checker in MINIGENT takes inspiration from Isabelle's termination checker (Bulwahn et al., 2007) and Agda's *foetus* termination checker (Abel and Altenkirch, 2000). The implementation follows a similar approach to the algorithm used in Isabelle, with large changes to the first three components: generating measures, applying measures, and local descent. Isabelle uses a theorem prover to solve the proofs required by the termination checker; as COGENT and MINIGENT do not have theorem provers, we must look at alternative methods. The *foetus* termination checker uses a manual approach by extracting dependencies from the program code. We follow a similar method for local descent, however, as the *foetus* language is adapted from λ-calculus and does not contain the same expressions supported by MINIGENT, our traversal of the program code is completely different.

A brief overview of the algorithm is detailed below:

1. Measures: create a set of measures to use for size analysis, based on the type of the function argument. Each measure provides a different way to analyse the size of the function argument.
2. Local descent: apply each measure to each function argument (input and recursive) and

attempt to prove that the measure decreases at the recursive call. Collect the results into a matrix.

3. Global descent: search for a lexicographic termination ordering in the matrix to determine termination.

3.1.1 Merge Example

We will use the following merge function (the merge part of mergesort) to describe the termination algorithm. In MINIGENT and COGENT, functions that take multiple arguments instead take a single record containing these arguments as fields of the record.

The following is an implementation of the merge function written in Haskell.

```
1  merge :: ([Int], [Int]) -> [Int]
2  merge ([], ys) = ys
3  merge (xs, []) = xs
4  merge ((x:xs), (y:ys)) =
5    if (x < y) then x:merge (xs, (y:ys))
6    else y:merge ((x:xs), ys)
```

Below is an implementation of the merge function written in MINIGENT. MINIGENT is a barebone subset of COGENT that only considers the essential language features of COGENT. As such, it does not offer syntactic sugar over case expressions. Each case expression has two options and the second option must be expanded into another case expression if we are casing over $n > 2$ options, or an esac expression for irrefutable matching. Hence the length of the following code.

The highlighted lines contain recursive calls. Line 1 contains the function `alloc`, which allocates space for an empty list structure. Lines 3 - 5 contain the function signature for merge. It takes in a record containing two recursive list types labelled with fields x and y, and returns a recursive list type.

```
1  alloc : Unit -> rec t { l: < Nil Unit | Cons { data: U32, rest: t! }# > take }!;
2
```

```
3  merge : {x: rec t { l: < Nil Unit | Cons { data: U32, rest: t! }# >}!,
4           y: rec t { l: < Nil Unit | Cons { data: U32, rest: t! }# >}!}#
5       -> rec t { l: < Nil Unit | Cons { data: U32, rest: t! }# >}!;
6  merge m =
7    case m.x.l of
8      Nil u -> m.y
9      | x ->
10     case x of
11       Cons xs ->
12         case m.y.l of
13           Nil u -> m.x
14           | y ->
15           case y of
16             Cons ys ->
17               if (xs.data < ys.data) then
18 ,               let result = merge {x = xs.rest, y = m.y} in
19                 let item = alloc Unit in
20                   let item1 =
21                     put item.l :=
22                       Cons {
23                         data = ys.data,
24                         rest = result
25                       }
26                   in item1
27               else
28                 let result = merge {x = m.x, y = ys.rest} in
29                 let item = alloc Unit in
30                   let item1 =
31                     put item.l :=
32                       Cons {
33                         data = xs.data,
34                         rest = result
35                       }
36                   in item1
```

3.1.2 Measures

First, a set of measures based on the *type* of the function argument is created to use for size analysis. In Isabelle (Bulwahn et al., 2007), measures are functions that map (in this case) function arguments to natural numbers. The merge function takes in a pair of lists, so a suitable set of measures would be $\{m_0 = fst, m_1 = snd\}$. As we cannot pattern match on types in MINIGENT and COGENT like we can in Isabelle, our implementation uses a measure data structure where each node represents a function. The measure data structure is an n-ary tree with different nodes representing each type.

- **Records**: we generate a set of measure nodes that *project* each field, called *ProjM*. For each record encountered, the measure data structure is copied n times, where n is the number of fields in the record. A different field is attached to each copy.
- **Variants**: we generate the cartesian product of the different variant possibilities. For each variant encountered, we create a *CaseM* node containing n children. Each possibility becomes a child node.
- **Primitive Types**: leaf node, called *PrimM*.
- **Recursive Parameter**: leaf node, called *RecParM*.

Looking at the argument type of merge again:

```
1 {x: rec t { l: < Nil Unit | Cons { data: U32, rest: t! }# >}!,
2  y: rec t { l: < Nil Unit | Cons { data: U32, rest: t! }# >}!}#
```

This is a record that contains two recursive list types. The recursive list type is a record that holds a single variant. The variant is either the primitive `Nil` constructor, or it constructs a record with a field containing a primitive and a field containing another list, indicated by the recursive parameter.

The corresponding set of measures would be:

$m_0 = ProjM\ \mathbf{x}\ (ProjM\ \mathbf{l}\ (CaseM\ [(\mathbf{Nil}, PrimM), (\mathbf{Cons}, ProjM\ \mathbf{data}\ PrimM)]))$

$m_1 = ProjM\ \mathbf{x}\ (ProjM\ \mathbf{l}\ (CaseM\ [(\mathbf{Nil}, PrimM), (\mathbf{Cons}, ProjM\ \mathbf{rest}\ RecParM)]))$

$m_2 = ProjM\ \mathbf{y}\ (ProjM\ \mathbf{l}\ (CaseM\ [(\mathbf{Nil}, PrimM), (\mathbf{Cons}, ProjM\ \mathbf{data}\ PrimM)]))$

$m_3 = ProjM\ \mathbf{y}\ (ProjM\ \mathbf{l}\ (CaseM\ [(\mathbf{Nil}, PrimM), (\mathbf{Cons}, ProjM\ \mathbf{rest}\ RecParM)]))$

The blue elements indicate where the measures differ from each other. In more detail, consider the measure m_1:

$m_1 = ProjM\ \mathbf{x}\ (ProjM\ \mathbf{l}\ (CaseM\ [(\mathbf{Nil}, PrimM), (\mathbf{Cons}, ProjM\ \mathbf{rest}\ RecParM)]))$

The argument type of merge is a record containing two fields, x and y. The measure data structure m_1 first projects the field x. x is a recursive record containing a single type l. The measure node then projects the field l. l is a variant that is constructed using either the Nil or $Cons$ constructor. The measure node provides both choices. Variants, indicated by $CaseM$ nodes, are the only measure nodes that branch into multiple children. The Nil constructor holds a primitive unit type. The $Cons$ constructor holds a record. From this record, the measure node projects the field $rest$, which is a recursive parameter.

These measures represent the four ways we can analyse the argument; on either the x or y fields, and then on either the *data* or *rest* fields. The number of measures we have is essentially the number of base fields in the data type.

3.1.3 Local Descent

The greatest challenge we faced during this thesis was implementing local descent. Local descent involves taking each (input, recursive) argument pair and applying each measure to them. For each pair, we should receive a list of n pairs, where n is the number of measures. Then we compare the measures on the input and recursive arguments to see if there is a size decrease at the recursive call for that measure.

To solve local descent, Isabelle uses `auto`, a classical logic solver that combines term rewriting,

classical reasoning, and some arithmetic. As COGENT does not have such a theorem prover, we implement an alternative method.

The termination checker previously used in Agda was first trialled in *foetus*, an experimental language based off of lambda calculus. The *foetus* termination checker performs local descent by extracting dependencies from the program and analysing how many times the input argument is *unfolded* before being recursed over. MINIGENT and *foetus* have different language features, so whilst our method is inspired from *foetus*, the implementation is different.

The implementation for local descent contains three major components:

- Create **templates**. Templates are data structures that describe the contents of the input and recursive arguments.
- Fill out two templates for each (input, recursive) argument pair.
- Apply measures to each template and compare the result to see if there is a decrease.

Templates

Templates are data structures that map out the contents of the function argument. They follow the same format as the actual type and are annotated with fresh variable names. These names map to expressions from the program code and are used for comparison later on.

A template for the argument of merge would look like this:

{(x, _, {(l, _, [(Nil, _, Unit), ($Cons$ _ {($data$, _, Int), ($rest$, _, RecPar)})])}),
(y, _, {(l, _,[(Nil, _, Unit), ($Cons$ _ {($data$, _, Int), ($rest$, _, RecPar)})])})}

The underscores are placeholders for fresh variable names that are filled in after function traversal.

Function Traversal

To complete the template, we traverse the program code and collect information on the structure of each function argument. This traversal must track three key pieces of information:

- **Path**: The input argument may take a different path of mutations through the program code to arrive at each recursive argument due to branching expressions. The specific path for each (input, recursive) argument pair relevant to that recursive call must be tracked.
- **Structure**: Collect enough information to flesh out the structure of each function argument (eg. at a variant type, select a specific branch and remove the others).
- **Expressions**: Remember key variables and expressions in the program code. When comparing sizes, having the structure is not enough; two list data types may have same structure, but not the same size. For example, consider two list elements with structures $Cons\{n, t\}$ and $Nil\ Unit$, where n is an integer and t is a recursive parameter. We know for certain that the former is structurally larger than the latter. However, given two elements with the structure $Cons\{n, t\}$, there is no way of knowing if t refers to the same list or different lists across the two structures. To resolve this, fresh variable names are added; $Cons\{n, t : \alpha\}$ and $Cons\{n, t : \beta\}$. The fresh variable names α and β map to expressions in the program code. Then we can check to see the relationship between those expressions, if any.

With these requirements in mind, we perform a top-down traversal to collect the path for each (input, recursive) argument pair. At the terminus of each path we attach a fresh variable name mapped to the relevant 'leftover' expression when necessary. Then, the path is reversed and used to fill out the template.

In the merge function, we have our input call and our two recursive calls:

$$(Input, Rec1) \quad (m, \{x : xs.rest, y : m.y\})$$
$$(Input, Rec2) \quad (m, \{x : m.x, y : ys.rest\})$$

After traversing the function and filling in the templates, we get:

Argument Pair	*Template Pair*
$(Input, Rec1)$	$(\{(x : \delta \ ... \ xs.rest : \alpha), y : \gamma\}, \{x : \alpha, y : \gamma\})$
$(Input, Rec2)$	$(\{x : \delta, (y : \gamma \ ... \ ys.rest : \beta)\}, \{x : \delta, y : \beta\})$

Here, the Greek letters refer to fresh variable names. α maps to `xs.rest` and β maps to `ys.rest`. δ maps to `x` and γ maps to `y`. The ellipses refer to other parts of the template, omitted for clarity. In this case, it implies $\alpha < \delta$ and $\beta < \gamma$, where $<$ means 'structurally smaller than'.

Templates are used to map out the *structure* of the arguments; fresh variable names are used to *link* the structures together so that they are comparable.

Applying Measures

Given a measure and a template, we match and remove corresponding nodes. A recursive parameter indicates that we have entered another structure and the structural size count is incremented. After applying the measures, we should end up with a structural size count and a fresh variable name referencing the last element of the template.

Consider m_1 applied to the first element of the pair ($Input$, $Rec1$).

$m_1 = ProjM$ **x** ($ProjM$ **l** ($CaseM$ [(**Nil**, $PrimM$), (**Cons**, $ProjM$ **rest** $RecParM$)]))

$template_{Input} = \{(\mathbf{x}, \mu, \{(\mathbf{l}, \_, [(\mathbf{Cons}\ \_\ \{(\mathbf{data}, \_, \text{Int}), (\mathbf{rest}, \alpha, \text{RecPar})\})]), (\mathbf{y}, \beta)\}$

Bold text indicates a field name or a constructor name that is part of the argument type. Greek letters refer to fresh variable names. Normal text indicates a Cogent type (eg. RecPar).

The template and the measures should correspond, given that they are both built around the type of the same argument. In this case, the first node of the measure, $ProjM$ **x**, projects the **x** field of a record. The first element in the template is a record referenced by the fresh variable μ. It contains two fields, **x** and **y**. The measure node and the template node match on the field **x**. We take the field **x** and continue reducing, similar to cancelling out terms in algebra. When we arrive at the $CaseM$ measure node, there are two options, **Nil** and **Cons**. The corresponding

variant in the template has only one option **Cons**; this is because the function traversal collected the path taken by that recursive call. The template nodes branch on records; the measure nodes do not. The measure nodes branch on variants; the template nodes do not. This means that we can always decide which path to continue down. Finally, we arrive at the fresh variable α. This is a recursive parameter, indicating that we have performed one unfolding, so we increment the structural size count.

Thus, applying m_1 to the template of the input argument results in the pair (α, 1).

Consider the same measure m_1 applied to the template for the recursive argument, *Rec1*.

$m_1 = \mathit{ProjM}\ \mathbf{x}\ (\mathit{ProjM}\ \mathbf{l}\ (\mathit{CaseM}\ [(\mathbf{Nil}, \mathit{PrimM}), (\mathbf{Cons}, \mathit{ProjM}\ \mathbf{rest}\ \mathit{RecParM})]))$

$\mathit{template}_{Rec1} = \{x : \alpha, y : \gamma\}$

The template is fairly sparse. After projecting the **x** field, we are left with the pair (α, 0).

Comparison

We have three structural size relations: $\{<, =, ?\}$. These indicate whether one element is structurally smaller than, structurally equal to or unknown when compared to another element. A recursive argument that is structurally greater than the input argument is also labelled as unknown.

Pairs are compared lexicographically. If the fresh variable names refer to the same expression, then we compare the structural size count for $<$ or $=$. If the fresh variable names refer to unrelated expressions we have an unknown.

Comparing the recursive result (α, 0) with the input result (α, 1) in the previous example, it's clear that there is a decrease in the recursive call along the measure m_1 and we get the relation $(\alpha, 0) < (\alpha, 1)$

Local Descent

The entire process looks like this:

First, traverse the function expression and fill in a template for each function argument. Create pairs of (input, recursive) templates.

Argument Pair	*Template Pair*
$(Input, Rec1)$	$(\{(x : \delta \ ... \ x.rest : \alpha), y : \gamma\}, \{x : \alpha, y : \gamma\})$
$(Input, Rec2)$	$(\{x : \delta, (y : \gamma \ ... \ y.rest : \beta)\}, \{x : \delta, y : \beta\})$

Apply each measure to each argument:

Argument Pair	*Template Pair*	m_0	m_1	m_2	m_3
$(Input, Rec1)$	$(\{(x : \delta \ ... \ x.rest : \alpha), y : \gamma\}, \{x : \alpha, y : \gamma\})$	_	$((\alpha, 1), (\alpha, 0))$	_	$((\gamma, 0), (\gamma, 0))$
$(Input, Rec2)$	$(\{x : \delta, (y : \gamma \ ... \ y.rest : \beta)\}, \{x : \delta, y : \beta\})$	_	$((\delta, 0), (\delta, 0))$	_	$((\beta, 1), (\beta, 0))$

m_0 and m_2 measure on the integer field of the list structure. At the moment, all measures on primitive types default to the zero function. That is, we consider all integers to have 0 structural size and have omitted them for clarity.

Comparing the pairs from the last four columns of the previous matrix, we receive by lexicographic ordering of the pairs:

m_0	m_1	m_2	m_3
_	$(\alpha, 0) < (\alpha, 1)$	_	$(\gamma, 0) = (\gamma, 0)$
_	$(\delta, 0) = (\delta, 0)$	_	$(\beta, 0) < (\beta, 1)$

Again, the columns m_0 and m_2 are comparing on integers, and result in equality as all integers are considered to have the same structural size.

The resultant matrix from local descent, where each row refers to a recursive call, and each column refers to a measure:

$$\begin{matrix} = & < & = & = \\ = & = & = & < \end{matrix}$$

3.1.4 Global Descent

Global descent involves searching for a lexicographic termination ordering in the matrix. The algorithm (Bulwahn et al., 2007) is as follows:

- Find a column with at least one $<$ and no ? (unknowns)
- Remove each row that has a $<$ in the chosen column
- Repeat until the matrix is empty (terminates) or cannot be reduced further (does not terminate)

The matrix we received from local descent is reducible by taking the second column (which has one $<$ and no unknowns), removing the first row, and then taking the last column (which now has one $<$ and nothing else) and removing the single remaining row.

Each row in the matrix represents a different recursive call and each column represents a different measure. Finding a column with at least one $<$ and no ? relations indicates that at least one recursive call will decrease and none will increase in size, for that measure. The terminating recursive calls are removed (rows containing $<$ in the selected column). Having an unknown in the matrix does not necessarily make it unsolvable, as some recursive functions may have calls that increase certain fields before decreasing. The key is to ensure that the recursive calls that do increase eventually enter recursive calls that decrease (eg. the Ackermann function). This algorithm essentially finds a path that will result in a lexicographic termination ordering.

3.2 Minigent Test Suite

To trial this implementation of the termination checker, we have created a suite of Minigent recursive functions. These include:

- Simple recursive functions, eg. sum list, count nodes, filter, concat. There are versions that are verbose, concise, and ones that contain variable shadowing.

- Lexicographic recursive functions, eg. merge, slow deallocator
- Recursive functions that call other recursive functions, eg. quicksort
- Integer recursive functions eg. addition
- Non-terminating examples (to ensure that termination checker fails when it should)

Chapter 4

Limitations and Future Work

4.1 Branching Arguments

Currently, the termination checker cannot handle functions that take in branching arguments. For example:

```
1  recFun (case x of A -> ... | B b -> ... | ...)
```

However, this is only a problem in Minigent, as Cogent will normalise functions before the termination checking phase.

4.2 Nested Functions

At the moment, the termination checker cannot deduce termination when external functions are called inside a recursive function. For example, consider this implementation of quicksort in Haskell:

```
1  quicksort :: [Int] -> [Int]
2  quicksort [] = []
3  quicksort (x:xs) = (quicksort lte) ++ [x] ++ (quicksort gt)
```

```
4     where lte = filter (<= x) xs
5           gt  = filter (> x) xs
```

The filter function is an unknown; we don't have any information how it modifies the structural size of its argument, if at all.

A potential fix for this is to add size annotations for functions to indicate if they increase, decrease, or do nothing to the structural size of their argument.

4.3 Integer Recursion

At the moment, the termination checker can only identify structurally recursive functions. In Agda, integers are represented as Peano numbers using *zero* and *succ*. The same representation is used for negative numbers. In such a representation, integers are structures and a structurally recursive termination checker naturally recognises integer recursive functions. In Cogent however, integers are C-like and can overflow and underflow. They do not have a structural size in the conventional sense, as they are not made up of constructors. This means that a structurally recursive termination checker is unable to recognise common functions that perform integer recursion. For example, recursive add:

```
1 add :: Nat -> Nat -> Nat
2 add 0 y = y
3 add x y = add (x-1) (y+1)
```

The general method discussed above can be extended to use non-structural measures for integers. One measure for integers is the absolute value; the integer 5 would be considered structurally larger than the integer 4, which would be considered structurally larger than 0, the *least* element. Recall that structurally recursive functions must be well-founded; they must perform recursion over well-founded sets. In this case, our absolute-value integer set would have 0 as its least element. After including a measure for integers, we would also have to alter the local descent algorithm to take into account integers, arithmetic expressions and integer comparison. Finally,

a constraint solver is required to resolve arithmetic expressions before size comparisons can be completed.

4.4 Custom Measures

A potential extension to the current termination checker is to allow user-defined measures. This approach is moving closer to the idea of type annotations Abel (2010) discussed earlier. Type annotations place all responsibility on the programmer to annotate sizes for each function and requires an alteration of the type system. Custom measures are different in that they allow the user to *optionally* provide a measure to help the termination checker for more difficult recursive functions. This would not interfere with the type system and in most cases the programmer would not be burdened. More investigation is required to determine how powerful and effective custom measures can be in comparison to the methods of structural recursion and type annotation.

Adding custom measures would be a fairly large task. A measure language is required and measure annotations must be passed through the compiler phases until the termination checker. These custom measures must also be integrated with the current termination checker, which is a significant amount of work.

Custom measures is the stepping stone towards manual termination proofs. Ideally, for particularly gnarly cases, the user should have the option of providing their own termination proof. At the moment, we've implemented a flag that toggles termination checking for each function, which can perhaps be extended to these projects.

Bibliography

Abel, A. (2004). Termination checking with types. *ITA*, 38:277–319.

Abel, A. (2010). Miniagda: Integrating sized and dependent types. In Bove, A., Komendantskaya, E., and Niqui, M., editors, *PAR*, volume 43 of *EPTCS*, pages 14–28.

Abel, A. and Altenkirch, T. (2000). A predicative analysis of structural recursion. *Journal of Functional Programming*, 12.

Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T. C., Klein, G., and Heiser, G. (2016). Cogent: Verifying high-assurance file system implementations. In Conte, T. and Zhou, Y., editors, *ASPLOS*, pages 175–188. ACM.

Bulwahn, L., Krauss, A., and Nipkow, T. (2007). Finding lexicographic orders for termination proofs in isabelle/hol. volume 4732, pages 38–53.

Colson, L. (1991). About primitive recursive algorithms. *Theor. Comput. Sci.*, 83(1):57–69.

Coquand, T. and Paulin, C. (1988). Inductively defined types. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, pages 50–66.

Murray, E. (2019). Recursive types for cogent.

Nipkow, T., Wenzel, M., and Paulson, L. C. (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg.

O'Connor, L. (2019). *Type Systems for Systems Types*. PhD thesis, University of New South Wales. Computer Science & Engineering.

O'Connor, L., Chen, Z., Rizkallah, C., Amani, S., Lim, J., Murray, T. C., Nagashima, Y., Sewell, T., and Klein, G. (2016). 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*, pages 89–102.

Telford, A. and Turner, D. (2000). A hierarchy of languages with strong termination properties. Technical Report TR 2-00, Computing Lab, University of Kent at Canterbury, The Computing Laboratory, The University, Canterbury, Kent, CT2 7NF. Paper currently under revision.

Wadler, P. (1990a). 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*, page 561.

Wadler, P. (1990b). Recursive types for free!

Xi, H. (2004). Applied type system. In Berardi, S., Coppo, M., and Damiani, F., editors, *Types for Proofs and Programs*, pages 394–408, Berlin, Heidelberg. Springer Berlin Heidelberg.