School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Enhancements to the Cogent Property-Based Testing Framework

by

Oscar Lewis Downing

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Computer Engineering

Submitted: April 2021

Supervisor: Christine Rizkallah

Student ID: z5114817

Abstract

Cogent is a restricted linearly-typed functional programming language with a certifying compiler designed for writing trustworthy and efficient systems code. Cogent comes equipped with a foreign function interface (FFI) to C, which allows interoperability with existing C programs and a means of escaping the restrictions of Cogent. Property-based testing can be used to find bugs in both specifications and implementations. In the context of Cogent, properties used in the high level formal specification can be translated into properties used in Property-Based testing, and vice versa. This allows it to serve as a precursory check to the process of formal verification, enabling an incremental approach to a fully verified system.

However, to conduct tests, one must set up a refinement framework for comparing the concrete function under test against an abstract Haskell function encoding functional correctness. Setting this up involves a lot of repetitive work and degrades the usefulness of the testing framework. Furthermore, testing becomes more onerous when testing C functions via the C FFI.

The aim of this thesis is to improve the usability of the framework and reduce the work required by increasing the level of automation. This is achieved with the introduction of code generators that build up the required functions, and a Domain-Specific Language that provides fine grain control over these generators. Initial results have shown a reduction in user effort and especially so when testing via the C FFI.

Acknowledgements

Christine Rizkallah, thank you for your guidance and supervision throughout this thesis.

Zilin Chen, thank you for your patient explanations and valuable expertise.

Finally, my deepest thanks to my lovely and understanding wife, whose love and support I would surely be lost without.

Contents

Chapter 1

Introduction

The COGENT language (O'Connor 2019) is functional programming language with features that facilitate verification of systems code. The COGENT compiler compiles the COGENT program down to a C language version; it also generates a proof in ISABELLE/HOL (Nipkow and Klein 2014) which states the C version of the COGENT program a refinement of a high level program that models correctness: the correctness specification. COGENT language has restrictions in place to facilitate this proof, however, such restrictions mean some programs can not be defined within the domain of COGENT and must be written externally with C and called via the COGENT C foreign function interface (FFI); verification of this must be done completely manually.

Verification by refinement is the main method for gaining assurance about COGENT programs. COGENT also provides a testing framework which mirrors the refinement conducted in verification. The testing framework has proven capable and features the ability to test the C code called via the C FFI, however, writing the code to drive tests and reasoning about it has shown to be troublesome and error prone.

This thesis aims to remedy this by increasing the levels of automation in the framework and by introducing a custom Domain-Specific Language (DSL): a front end interface for user specification of test requirements. Initial benchmarks of the project point towards a reduction in the lines of code required from the user. Moreover, the DSL allows for a single file for testing, and the syntax is simple enough such that one can look at the test file and clearly see what is being tested, how data is transformed along the way and what well-formed data looks like for the tests.

chapter 2 explains the background and motivation for this thesis, and chapter 3 details the solution approach and implementation. chapter 4 discusses the results and significance of the work, and chapter 5 draws up conclusions and suggests directions for future enhancements.

Chapter 2

Background

2.1 Cogent

2.1.1 Language Overview

The COGENT programming language (O'Connor 2019) has been built from the ground up for writing verifiable systems code and reducing the human cost of verification. It is a restricted, pure and total, functional language with a uniqueness type system: a linear type only has exactly one usable reference. The COGENT programs is compiled down to C, which allows the restrictions imposed on COGENT to be maintained in the generated C code; these restrictions assist in verification of the program while still maintaining efficiency.

2.1.2 The Cogent C FFI

The restrictions of COGENT help ease the process of verification, however, there are many things COGENT cannot express, such as side effects (e.g. memory allocation); to overcome the restrictions, the language comes with a built-in FFI, to the C language. Thus, programmers can define abstract definitions in COGENT and complete the implementation in C itself. Additionally, this permits COGENT programs to be interoperable with existing C programs and aids the viability of COGENT as a language for systems development.

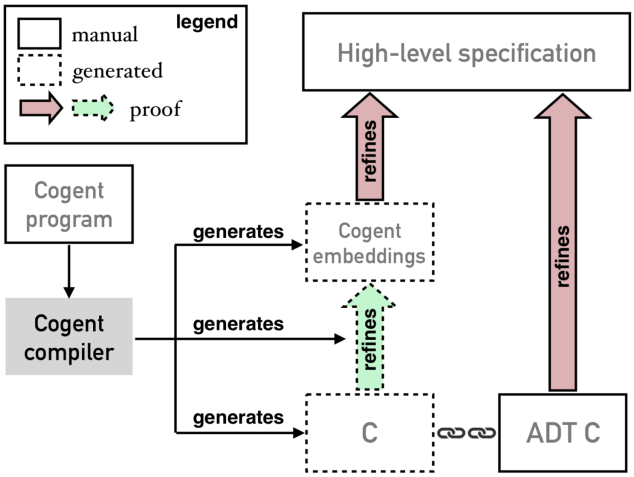

Figure 2.1: Overview of the COGENT Refinement Framework (Chen et al. 2017)

2.1.3 The Cogent Refinement Framework

Gaining trust about a COGENT program is achieved with a rigorous mathematical step-wise refinement proof, where a low level implementation of the program is proven to be a refinement of a high level model of functional correctness: the correctness specification (O'Connor, Chen, Rizkallah, et al. 2016). COGENT features a certifying compiler which means it aids in establishing the refinement proof in ISABELLE/HOL (Nipkow and Klein 2014). That is, the concrete (low level) C code is proven to be a verifiable transformation of the abstract (high level) formal specification. The compiler the proof, and then it is up to the proof engineer to validate the proof scripts with execution.

Figure 2.1 is an overview of the COGENT Refinement Framework. The compiler generates C code which is the concrete implementation (and is the code executed at runtime), and also generates an ISABELLE/HOL embedding of this C code using the *AutoCorres* tool (Greenaway et al. 2014); an ISABELLE/HOL embedding of the original COGENT program is also generated, as well as a series of proofs for each COGENT compiler transformation that occurs upon the original program. The generated artefacts of the compiler establish a step-wise refinement proof between the ISABELLE/HOL embedding of the C semantics and the ISABELLE/HOL embedding of semantics of the COGENT program. The final refinement step between the ISABELLE/HOL embedding of the COGENT program and a correctness specification must be manually written. Then, end-to-end verification is achieved, i.e. top level specification models the behaviour of generated C code.

For COGENT functions with implementations called via the C FFI, the compiler does not generate any proofs. The refinement proof must be constructed manually, which can be time-consuming. However, the human cost of this verification can be amortized over time as the functions tend to be library functions that can be reused.

The refinement framework in COGENT is essential for proving properties about COGENT programs. For the programs that can be written within the restricted (but expanding) domain of COGENT, the human cost of the verification is drastically reduced. The C FFI provides an escape from these restrictions, however it must be used with caution as verification is much more involved.

COGENT language is both a viable and verifiable systems language. However, there will always be programs that have impure components and their behaviour also needs to be validated; proofs for these programs are costly and require a great deal of expertise. Testing can play a role here, as it strikes a good balance between easy of learning and rigour of coverage. COGENT has a testing framework for this purpose, and it mirrors the refinement proof discussed in this section.

2.2 Testing

At a high level testing involves a set of requirements that specify correctness and evaluating a system on is subcomponents with the intent to find whether it satisfies these requirements or not. At it most basic form testing is essentially feeding in input and "checking" behaviour either by inspecting the outputs or program internals. Black box testing has become popular because it is only concerned with checking outputs against expected outputs, and requires minimal internal knowledge of the program (Khan 2011).

Testing serves the purpose of helping to gain assurance quickly with minimal programming overhead. A degree of testing capability is available in most programming languages and is especially handy for programmers. While it not the same as the gold standard of a proof, it does have a lesser learning curve and plays an important role in the development cycle, as tests can be used to gain assurance about code incrementally and quickly; an estimated 70% of development time is spent testing (Srivastava et al. 2011)!

COGENT has a testing framework for this purpose. Under the hood the testing library is property-based (a form of black-box testing) and this enables the tests to mirror the refinement

described in subsection 2.1.3. But, rather than with proofs, inputs are randomly generated to validate refinement between the concrete function under test (FUT) and the model of correctness.

2.3 Property-Based Testing

2.3.1 Overview

Testing is another form of creating assurance about a program's functional correctness. There are two major types of testing: White Box testing where program internals are considered when evaluating correctness, and Black Box testing where internals are not checked only program input/output is considered. Property-Based Testing (PBT) is a form of black box testing where inputs are randomly generated and outputs are validated against a user-defined predicate.

In PBT, a function's correctness is encoded by a predicate, asserting something about that function's inputs and outputs; this is the property of the function. Several properties may be defined to capture different aspects of the function's behaviour. When running the tests, random inputs are generated and fed into the property; either true or false comes out and the test driver determines if a counterexample has been found. If no counterexamples are found then this gives us pretty good assurance the function behaves as expected.

2.3.2 *QuickCheck*

The Haskell *QuickCheck* (Claessen and Hughes 2000) library pioneered PBT, and it is used in the Cogent PBT framework. *QuickCheck* provides combinator functions to build up Property definitions. A test unit about a function usually contains a predicate and a test data generator.

Listing 2.1: Involution Property of the reverse list function

```
1 prop_revRev :: Property
2 prop_revRev = forAll arbitrary revPred
3     where revPred :: [Int] -> Bool
4           revPred xs = reverse (reverse xs) == xs
```

The example shown in Listing 2.1 showcases testing the involution property of a list reversing function, it reads: for all arbitrary integers reversing a list, and then reversing it again will be equal to the original list. In this case, the test data generator is `arbitrary` which generates random test data based on the input type, and the property predicate is function `revPred`.

PBT tends to have good test execution performance and this can be improved drastically when the test data generator is custom. This boils down to defining a nice test data generator using the combinators provided, meaning specific ranges of test data can be defined. Alternatively preconditions for test data can be defined to achieve this, however they can be inefficient since they will discard generated values that fail the precondition.

QuickCheck is used because it allows properties, in a mathematical sense, to be defined and tested in Haskell, therefore higher order logic Isabelle/HOL can be translated into *QuickCheck* logic. Thus, properties in Isabelle/HOL can be translated into Haskell *QuickCheck* properties and vice versa (with the caveat that there may be some work involved with the translation). Which is advantageous because PBT has been shown to be capable of finding bugs in both implementations and specifications (Hughes 2016).

2.4 Current Cogent PBT Framework

2.4.1 Overview

The Cogent PBT framework (Chen et al. 2017) uses Haskell *QuickCheck* library. To conduct tests, first the Cogent program must be translated into Haskell: a *shallow embedding* is generated, which is semantically equivalent to the original Cogent program but with Haskell syntax. The compiler will automatically generate this Haskell embedding for Cogent programs. Any function written via the C FFI is not translated but rather the interfacing functions required to use the Haskell C FFI (Chakravarty 2000) are generated.

QuickCheck allows you to define functional correctness similarly to how you would in Isabelle/HOL: verification by refinement expressed as a *QuickCheck* property function in Haskell and validated by random data. The property is a predicate detailing how the concrete implementation (the Haskell embedding) refines the abstract correctness specification which must be written in Haskell. Thus, refinement between functions can be tested with *QuickCheck*, mirroring the verification approach: a single step refinement between a Haskell

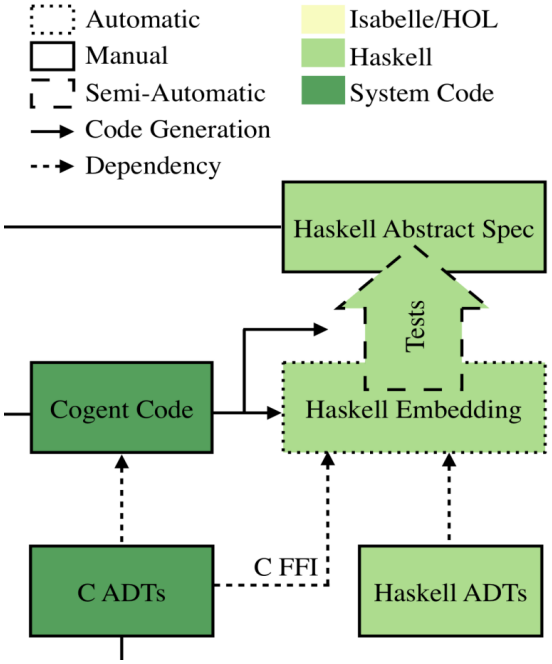

Figure 2.2: Overview of the COGENT PBT Framework Chen et al. 2017

correctness specification and a concrete implementation, where during execution of tests random data is generated in search of a counterexample. The implementation can be either the pure COGENT program shallowly embedded it HASKELL, or impure C function, that is called via the C FFI, or this can be manually written in HASKELL itself. Instead of proving refinement between a specification and an implementation, refinement is tested with random data (generated by *QuickCheck*).

Figure 2.2 summaries the components of the framework, and showcases the automated and manual components. The compiler generates a HASKELL shallow embedding of the COGENT function, this is semantically equivalent to the COGENT function; for pure Cogent functions this is completely automatic, however, for C FFI functions only the interface functions are generated and the user must handle the monadic effects due to the IO, alternatively, a semantically equivalent function can be manually defined in HASKELL. The "tests" arrow here represents the *QuickCheck* property function that encodes the refinement: the expression connects the concrete HASKELL embedding of the COGENT function to the HASKELL abstract correctness specification. This is semi-automatic as the functions that compose the property expression must be defined and then the property is automatically tested with *QuickCheck*.

2.4.2 Testing Refinement in the Cogent PBT framework

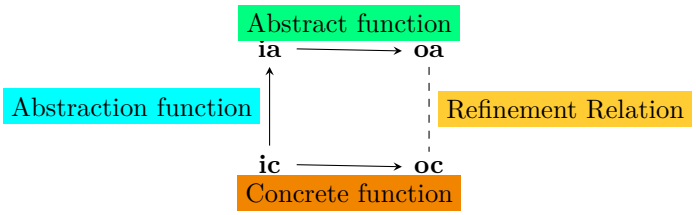

Figure 2.3: Refinement Statement Diagram.

abs : $ia \to oa$

Figure 2.4: Abstract function (the correctness specification)

conc : $ic \to oc$

Figure 2.5: Concrete function (the function under test)

absf : $ic \to ia$

Figure 2.6: Abstraction function

rrel : $oa \sim oc$

Figure 2.7: Refinement Relation

Figure 2.3 displays is a simplified refinement statement, similar to proving refinement, testing refinement follows a similar pattern, comparing an abstract specification (**abs**) against a concrete implementation (**conc**). However, testing requires input, in the PBT framework inputs are randomly generated; in the diagram, **ic** and **ia** represent input for concrete function and abstract function, respectively, and they must be related: they can be generated independently however it would be extremely inefficient. Instead, the abstraction function **absf** must be introduced to convert from concrete type (**ic**) to abstract type (**ia**), allowing concrete inputs to be randomly generated, and thus ensuring the inputs are related. Abstraction is essentially converting data representations from one with more detail to one with less detail, keeping only those data fields deemed necessary. Thus, the abstraction function pulls apart the concrete

input and populates the abstract output type with only the fields vital testing. The abstract function **abs** models the concrete implementation with minimal implementation detail. Finally, the refinement relation **rrel** compares the outputs of the concrete function (**oc**) and the abstract function (**oa**), once again, depending on the implementation details of the **oc** may be ignored in the comparison To summarise, **ic** is randomly generated, **absf** converts it to **ia** and both **conc** and **abs** are executed and **rrel** compares the outputs; and both **absf** and **rrel** can ignore details from the concrete implementation, as **abs** is the source of truth in terms of correctness.

Refinement Statement as a *QuickCheck* Property

Listing 2.2: Refinement Statement QuickCheck Property

```
prop_fn :: Property
prop_fn = forAll gen_fn
                 (\ic -> let oc = conc ic
                             ia = absf ic
                             oa = abs ia
                         in corres' rrel oa oc)
```

As displayed in Figure 2.3 the code snippet Listing 2.2 showcases how the refinement statement is encoded in Haskell as a *QuickCheck* property for testing. The anonymous function takes in **ic** and encodes that refinement statement. Test data generator supplies **ic**. Thus, the property forms a predicate with Haskell expressions, and during test execution *QuickCheck* will run the test data generator (**gen_fn**) and the refinement predicate will be tested.

Example using *averageBag* function

The *Bag* Cogent program determines the average of a list containing 32-bit unsigned integers with an aggregation function, originally demonstrated by O'Connor 2019.

Listing 2.3: *Bag* type

```
type Bag = { count : U32, sum : U32 }
```

The *Bag* type in Listing 2.3 is used for tracking the sum and size of the list, it is a Cogent record type consisting of the fields *count* and *sum*.

Listing 2.4: *averageBag* function

```
averageBag : Bag -> < EmptyBag | Success U32 >
averageBag b
           = if b.count == 0 then EmptyBag
             else Success (b.sum / b.count)
```

The *averageBag* function Listing 2.4, is a function within the *Bag* program that determines the average of the *Bag* type: it performs a safe division returning a variant type enclosing the answer. It is the concrete implementation under test.

Listing 2.5: *averageBag* Abstract function (correctness specification)

```
hs_averageBag :: (Int, Int) -> Maybe Int
hs_averageBag ia = if fst ia == 0 then Nothing
                   else Just $ snd ia / fst ia
```

Listing 2.5 is the abstract specification in HASKELL that models the implementation in Listing 2.4. Note that the concrete function is translated into HASKELL semantics for testing however in Listing 2.4 the COGENT version is shown for simplicity.

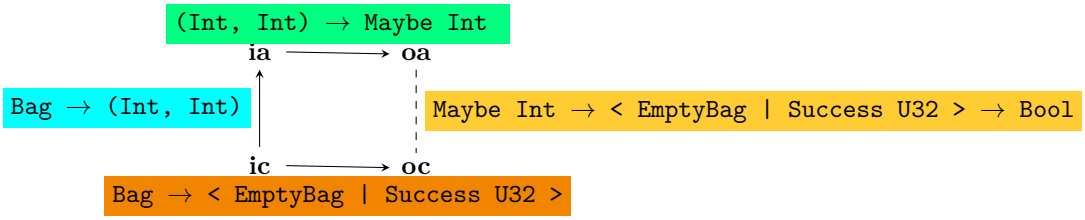

Figure 2.8: Refinement Statement Diagram with *averageBag* types.

The Figure 2.8 operates the same as Figure 2.3, however, in Figure 2.8 the function type signatures are inserted, to assist in demonstrating the flow of data when tests are executed. The test data generator must be defined to create a *Bag* type, essentially each field is populated with random data using *QuickCheck arbitrary* test data generator. Random concrete inputs are generated and fed into the refinement *QuickCheck* property. This entails each of these functions being executed and following the refinement in Figure 2.8: evaluating to either true or false, either finding a counterexample that breaks the refinement property or the property is validated as no counterexamples were found.

2.5 Analysis of Existing Framework

The current PBT framework is a good option for when testing COGENT functions, however, the clean abstracted view shown so far has hidden the drawbacks of using this framework. In practice, testing involves messing around with a compiler generated version (a shallow embedding) of the original code, since HASKELL is used to conduct PBT, a HASKELL shallow embedding is generated which is quite un-intuitive because all types are given new names.

PBT is a good fit for the Cogent ecosystem, but requires a lot of work to set up, as the HASKELL shallow embedding of COGENT types are unintuitive: the embedding converts types from COGENT structural types (types differentiated by structure) to HASKELL Nominal types (types differentiated by names). For each COGENT type in the original program, the compiler will generate a parametric type with a new mechanical name, which is instantiated by the function itself. The shallow embedding is necessary for converting between type systems, however, it gets confusing quickly when testing, and especially so for larger programs.

The following analysis focuses on two key drawbacks of the PBT framework, the pain of dealing with the HASKELL shallow embedding section 2.5, and the toil associated with the defining the functions that compose the *QuickCheck* refinement property section 2.5.

Haskell Embedding

Previously section 2.4.2, the *averageBag* example has hidden the shallow embedding, in this section the shallow embedding is exposed for analysis.

Listing 2.6: *averageBag* input COGENT type

```
type Bag = { count : U32
           , sum   : U32 }
```

Listing 2.7: *averageBag* output COGENT type

```
type R = < EmptyBag | Success U32 >
```

Listing 2.8: *averageBag* input HASKELL shallow embedding type

```
data R4 t1 t2 = R4{ count :: t1
                  , sum   :: t2 }
-- example instantiation:
R4 Word32 Word32
```

Listing 2.9: *averageBag* output HASKELL shallow embedding type

```
data V0 t1 t2 = V0_EmptyBag t1
              | V0_Success t2
-- example instantiation:
V0 () U32
```

When testing the *averageBag* function the input type shown in Listing 2.6 becomes Listing 2.7 in the shallow embedding stage of the compiler. The *Bag* type is given a mechanical name and converted to a polymorphic type which is instantiated when defining the function. The output type of *averageBag* is shown in Listing 2.8 becomes the type in Listing 2.9, during the shallow embedding stage of the compiler. Similarly, for the variant type, the COGENT structural type becomes HASKELL nominal type.

For each function return type a new R or V is enumerated upon. This becomes more confusing when testing C functions via the FFI, as the concrete function is not shallowly embedding but the original C program is connected to HASKELL with the FFI. When testing, C types are represented in HASKELL by a record or a primitive, and wrapped in the IO monad, resulting in unintuitive function input and output types. To combat this a DSL was proposed by Chen et al. 2017 to allow for an abstracted view of testing where the mess of the embedding is kept behind the scenes. This would make the framework more usable.

Functions for setting up Refinement Test

Listing 2.10: Functions required to be defined for testing (Highlighted)

```
prop_fn :: Property
prop_fn = forAll gen_fn
                 (\ic -> let oc = conc ic
                             ia = absf ic
                             oa = abs ia
                         in corres' rrel oa oc)
```

The HASKELL function in Listing 2.10 shows the *QuickCheck* property which encodes the refinement Figure 2.3. The highlighted functions that compose the property are the ones that must be defined by the user to set up the tests. In other words, additional work on top of writing the function under test. **absf** must transform **ic** into **ia**, and **rrel** must compare **oa** and **oc**. **gen_fn** is the test data generator and must also be defined.

The core of work is fairly straightforward but repetitive and mostly involving decomposition and composition of types. This is because the abstraction function **absf** must convert the COGENT type (its HASKELL embedding version) into a more high level HASKELL type. Moreover, the refinement relation involves a comparison between types from different languages, the output from the concrete function (a COGENT type but embedded in HASKELL) and the output from the abstract function (HASKELL type). Furthermore, the test data generator must generate the components of the type and populate the constructor. All these functions heavily require the deconstruction/construction of types. A task not in itself tedious but for more complex types becomes quite laborious and error prone, especially so when testing functions via the C FFI, as everything is wrapped in additional constructors (to handle interoperability safely).

The aim of this thesis is to introduce enhancements initially proposed in Chen et al. 2017 which would help to overcome these shortcomings of the COGENT PBT framework, leading to it becoming a more viable option for gaining assurance for COGENT functions and especially those functions implemented via the COGENT C FFI.

2.6 *Lens* Library

The core issue involved when abstracting one type to another, or when comparing two different types during refinement, is how to succinctly pull apart a type into its subcomponents. This is because during abstraction from concrete to abstract, concrete implementation details may be discarded, and therefore, the subcomponents of the overall type need to be extracted and placed into the abstract type constructor. Furthermore, when comparing the concrete and abstract outputs, the inhabitants of each must be extracted and then compared with a relation. Thus, there is motivation for a simple syntax for extracting type inhabitants. The *Lens* library is a widely used HASKELL library that specialises in this and therefore must be surveyed.

This section provides an overview of the original HASKELL *Lens* library (Kmett 2021b) which developed upon ideas introduced by Foster et al. 2007 and Pickering, Gibbons, and Wu 2017.

What is the *Lens* library?

In a nutshell, the *Lens* library simplifies viewing and updating HASKELL data structures by introducing functions that offer a generic way of focusing on a particular parts of the structure. The "focus" function is called the *view* and the structure it is focusing on is called the *source*. Using *view*, a particular field from the *source* can be focused on: either reading or updating. To drill down through a nested data structure, to get to a particular field with *view*, field name/tuple position/variant alternative name are composed.

Listing 2.11: *Lens* view expression

```
let ic :: (U32, Bag{count,sum})
    ic = (10, (Bag 10 2))
    sum' = ic^._2.sum
in  ...
```

The example Listing 2.11 shows how *Lens* infix *view* function (`^.`) can be used to extract the *sum* field from the nested data structure **ic**. (`^.`) is the *view* infix syntax, the first argument is on the left-hand side of the *view* and the second argument is on the right-hand side. The first argument is the object being viewed, the *source*, and second argument is a composition of field names that drill down to extract the *sum* field. Aesthetically similar to first class accessors from Object-Oriented languages, except tuples elements are extracted with underscore prefixing their index.

As outline above, the COGENT PBT framework shortcomings steam from unintuitive types in the shallow embedding, and repetitive work required to establish tests. The *Lens* library can be leveraged in the solution to these problems, as its syntax is extremely uniform, minimal, and easy to generate, and therefore can be used to improve usability.

2.7 Possible Solutions

Domain-Specific Language for the Cogent PBT framework

As pointed out by Chen et al. 2017, a custom domain-specific language (DSL) for defining test requirements would be a useful additional into the framework; provided there is: fine grain control over the code generation, and the syntax is simple and terse. The DSL must be

custom due to the specific requirements of the COGENT PBT framework. A key indicator of usefulness in this regard is the Lines Of Code (LOC) metric, which gives a rough estimation of work. When using the DSL to define a test, a reduction in the LOC versus the manual implementation would indicate a reduction in required work.

Code Generation

Enhancing the code generated from the compiler would assist in reducing the toil of testing. However, limited information can be striped from the COGENT program itself, and therefore back-end modules must work in tandem with the proposed front-end DSL. Thus, this would allow the functions that compose the refinement *QuickCheck* property to be generated, as well as the property itself.

Lens integration

Lens is a powerful HASKELL library which exposes functions for viewing and updating structures. Integrating the library into the DSL would expose a concise and expressive way of decomposing and composing types to the user defining test requirements. Which would be extremely useful and can assist in improving usability, as discussed in the previous section section 2.6.

Beginners Luck

Beginner's Luck, Lampropoulos et al. 2016, is a HASKELL embedded DSL used to streamline *QuickCheck* PBT by allowing definition of test data generator and also correctness specification to be written in one fell swoop. Therefore, it could assist in improving usability of the framework by simplifying the definition of specification and test data generator. However, *Luck* by itself would not be enough to solve the drawbacks of the framework, it's integration into the DSL may prove useful. In the end, *Luck* was deemed out of scope for this thesis.

Choice of Shallow Embeddings

ISABELLE/HOL has *QuickCheck* library as well (Bulwahn 2012), and a ISABELLE/HOL shallow embedding is generated by the compiler. Meaning, this thesis can focus on the HASKELL

embedding and build up a solution for that or focus on Isabelle/HOL, because they both have *QuickCheck* test libraries. In the end, Haskell was chosen because of preference and because it suited the use case better.

To summarise, the approach selected for tackling the drawbacks of the Cogent PBT framework is a combination enhancing code generation and implementing a DSL (with *Lens* integration) for the specification of test requirements.

Chapter 3

Solution & Implementation

3.1 Solution: DSL and Generators

The COGENT PBT framework is in need of usability enhancements and the focus of this thesis was to implement enhancements initially proposed by Chen et al. 2017. In terms of the main modules of the proposed enhancements, the primary focus can be broken down into three key components.

Custom Domain-Specific Language

Users of the testing framework would benefit from a custom DSL as it would provide a tailored way of describing testing requirements. Moreover, it would abstract away the messy details discussed previously in section 2.5, and result single test file for the function under test (FUT) — rather than having to dig around the HASKELL shallow embedding and supporting HASKELL test file.

Code Generators

The back-end compiler modules that would underpin the DSL must have enhanced generation in order to properly set up the refinement test in *QuickCheck*. With the DSL providing an interface for users to specify requirements and code generators in the compiler building the

refinement test, the users will be able to get into a test ready state much faster than with the current framework.

Lens **Integration**

The *Lens* library provides simple and powerful syntax for decomposing type. Integrating the library into the DSL would provide uniform syntax for referring to the type fields which is beneficially to usability because the *Lens* expressions are easy to construct.

3.2 Scope and Requirements

3.2.1 Overview

Naturally, there are many ways to enhance the framework, however, given the time and resource constraints for this thesis, the scope of the enhancement has been restricted to the key listed above in section 3.1. This was to account for the effort expected to spent designing, implementing, integrating and testing.

The focus of this thesis is to improve the usability of the framework, consequently, the requirements can be broken down into two core usability requirements:

1. **Less work**. The enhancements must reduce the work required of the user. The metric chosen to discern the success of this is the lines of code (LOC) of the test file. Thus, the LOC required of the user when writing the test file must be fewer when using the new framework versus when using the current framework.
2. **Decent testability scope**. The enhancements must be able to test a good majority of possible scenarios. The metric chosen to determine the success in this regard is if the enhancements can handle a set scope of types. If these types are covered then by extension the functions are testable. The types targeted are:
    - Core COGENT types, that is just the original type system proposed by O'Connor 2019. This would allow pure COGENT functions to be tested.
    - C types, This would allow C FFI functions to be tested.

The following section subsection 3.2.2 details the specifics of the proposed solution and the scope and requirements of the implementation.

3.2.2 Solution

The approach to enhancements, in terms of implementation, are twofold: a custom DSL to encode test requirements and an upgrade of code generation.

The following functions have been targeted for code generation:

- prop_fn - The refinement statement itself.
- absf - The abstraction function.
- rrel - The refinement relation.
- gen_fn - The test data generator.

Building these functions during the HASKELL embedding stage (during code generation) would result in the refinement test being set up, allowing the user to simply run the tests.

The domain-specific language (DSL) for the COGENT PBT framework has also been implemented, fulfilling the following requirements:

- Custom for describing abstraction and refinement relation.
- Able to describe well-formed test data, where by "well-formed" test data is data with efficient and practical range of values which varies depending on context.
- No compromise in expressiveness versus a manual test implementation. A key objective of the DSL is to ensure there is no compromise on what the user can express with the DSL, since the DSL will eventually completely replace manual testing approach.

Moreover, the end goal of DSL is to have a language, which allows for a blend of COGENT and HASKELL, in which users can describe all the test requirements. It should exist as an abstraction layer away from the shallow embedding. A single test file, which one can inspect and quickly determine: what it means for that function to be considered correct and what tests back up the validation. By making the way of testing more uniform, the ease of use can be improved and the chance messing things up in the set-up is greatly reduced.

In terms of the generators, a key requirement was to ensure both COGENT types and COGENT Abstract types can be handled. Which boils down to being able to test both: Pure Cogent functions and Cogent functions implemented via C FFI. Focusing on the core COGENT language, no extensions. It is the most useful covering C FFI functions because testing or formally verifying them requires the most effort.

3.3 Implementation

3.3.1 Overview

This following outlines the additions to COGENT PBT framework, in a high level architecture view.

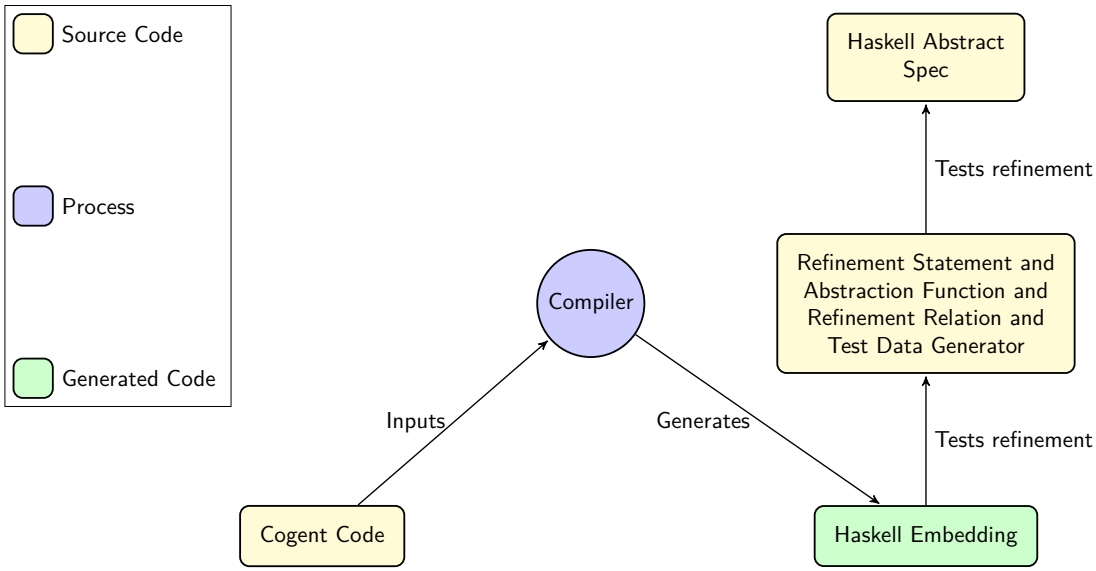

Figure 3.1: Original COGENT PBT framework

The Figure 3.1 is a slightly simplified overview of the original PBT framework. The user must write, not only the original file, but also the functions in the yellow boxes on the right-hand side. The user must do this manual work, which breaks the abstraction layer as user must write not only the source code but also must modify the generated shallow embedding. The shallow embedding of the concrete function is automatically generated by the compiler, as well as the C FFI code, so that functionality is translated. The additions to framework build of this existing

architecture.

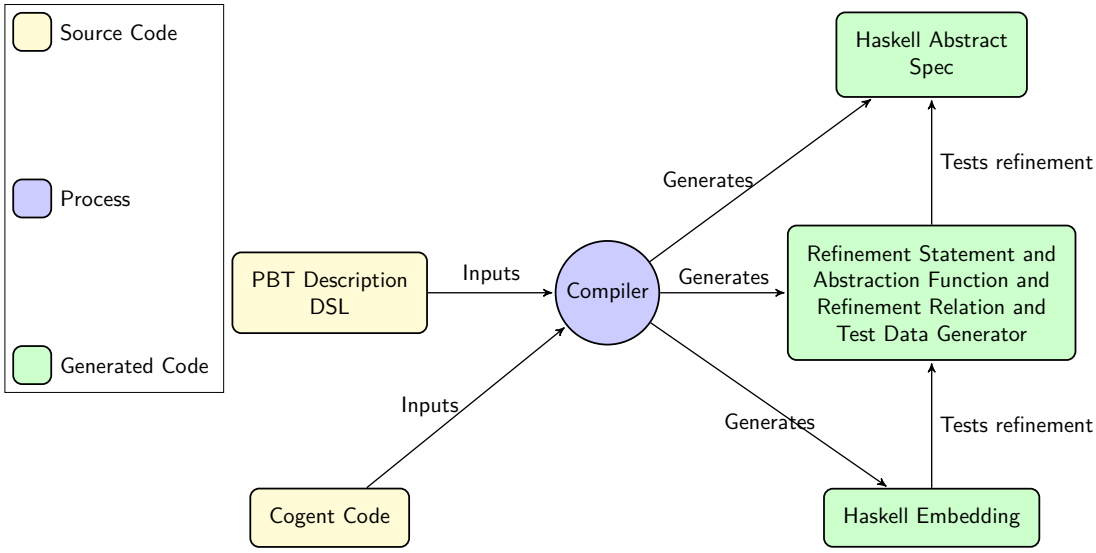

Figure 3.2: COGENT PBT framework with enhancements

The Figure 3.2 is an overview PBT framework with the additions, now able to parse the DSL and generated much more of the refinement statement.

The user must write a test file with the DSL for the function under test The DSL is a blend of COGENT and HASKELL syntax that is parsed, de-sugared and then compiled into the output HASKELL test file. The user's role has been completely lifted into the source layer, abstracted away from generated code. Now, the user can just worry about this DSL abstraction, and generators fill in the rest, so testing can be established right away.

3.3.2 The PBT Description DSL

The DSL implementation has simple syntax and semantics with no compromise to expressiveness. Moreover, it achieves the goal of being an abstraction away from shallow embedding.

The implementation of the DSL is simple front-end for taking user inputs and providing them to the back-end generators via a HASKELL record type, it is built using the HASKELL *Parsec* library (Leijen, Martini, and Latter 2021). HASKELL syntax is allowed to be embedded within the DSL, which aids in the expressiveness of the DSL. The DSL covers core cogent types and C FFI types. Moreover, its syntax allows for a uniform way to refer to type and type

inhabitants. But doesn't require the user to have to go digging around the shallow embedding, and its types. Furthermore, it has been designed to be as a wrapper around HASKELL syntax, that is the DSL expressions provide context for the HASKELL expressions. This allows the DSL to replace manual testing completely because edge cases that don't suit the automation well can be handled with more verbose definitions: that is the DSL core syntax is uniform for all test cases and HASKELL provides that expressiveness for this situation.

Code generation is guided by user inputs via DSL: either fully generating each function or partially generating The generator module in the compiler takes in the information the user supplies and builds HASKELL Abstract Syntax Tree (AST) for each required function, this is detailed in subsection 3.3.4.

Grammar for the PBT description DSL

This section contains the grammar for the DSL.

⟨*PBT Desc*⟩	::=	`` `*func name*` { ⟨*Decl*⟩$^+$ } ``	(Top level test description)
⟨*Decl*⟩	::=	⟨*Context Keyword*⟩ { ⟨*Expr*⟩$^+$ }	(Declaration)
⟨*Context Keyword*⟩	::=	**pure** | **nond** | **absf** | **rrel** | **welf** | **spec**	(Context)

⟨*Expr*⟩	::=	⟨*Meta Expr*⟩ ;	(Cogent/HS Syntax)
	|	⟨*Expr*⟩ : ⟨*Expr*⟩ ;	(Type operator)
	|	⟨*Expr*⟩ := ⟨*Expr*⟩ ;	(Definition operator)
	|	⟨*Expr*⟩$^{0\mid 1}$:| ⟨*Expr*⟩ ;	(Predicate operator)
⟨*Meta Expr*⟩	::=	⟨*Key Ident*⟩	
	|	*Cogent Type* | *Haskell Type* | *Haskell Expr*	
⟨*Key Ident*⟩	::=	**ia** | **oa**	(Abstract Input/Output Idents)
	|	**ic** | **oc**	(Concrete Input/Output Idents)
	|	⟨*Key Ident*⟩ ⟨*Haskell Expr*⟩ ;	

Where *func name* is name of function under test, *field name* is name of the field in the record, and *position* $\in \mathbb{N}_1$ is inhabitant index in tuple.

Usage

The DSL encodes the refinement displayed in Figure 2.3. The user must specify:

- The Haskell functional correctness specification (abstract function)
- Purity of conc , the non-determinism of abstract function , and the types of abstract function : **ia** and **oa**

Defining the specification in the DSL is a must and is convenient to keep it here. Just defining this will result in generators building a "direct" refinement test.

The "direct" refinement test involves: Abstraction that entails converting from COGENT type to HASKELL type; pulling out each concrete type inhabitant from the structure and then populating the abstract type constructor with them in the same order. Relation is a comparison where each concrete type inhabitant is compared with its corresponding abstract inhabitant with equality. For the random test data generator, not much can be generated other than populating each type inhabitant with an arbitrary test data generator. "direct" refinement test only works for abstracting to built-in HASKELL types. However, provides a default way to establish tests.

To gain more fine control over the generated code, the user can specify:

- How **ic** is abstracted into **ia** — otherwise defaults to "direct"
- How **oc** is related to **oa** — otherwise defaults to point-wise equality
- gen_fn function for **ic** — otherwise defaults to arbitrary

Grammar Overview

This section provides an explanation of the DSL grammar displayed in section 3.3.2. The DSL Keywords:

pure | **nond** | **absf** | **rrel** | **welf** | **spec**

Are used for defining things about the generators for a specific function. The keywords are like exposed interfaces for the generators in which arguments can be given, only in the specific form

it accepts. Each of the keyword blocks take arguments in the same format which composed of the DSL operators.

The DSL Operators:

1. **:** (for types)
2. **:=** (for definitions)
3. **:|** (for predicates)

Form the base of the DSL expression used to populate the keyword blocks. These expression are uniform for each keyword block and they can cover everything that needs to be defined: type, definition, or predicate.

The expressions on either side of the DSL operators must contain on of the key identifiers. The DSL key identifiers:

1. **ia** | **oa** (for abstract inputs and outputs)
2. **ic** | **oc** (for concrete inputs and outputs)

Allow for inputs and outputs to be referred to uniformly and generically, no matter the function.

The key identifier must go on the Left-hand Side (LHS) of the operator and that binds the Right-hand Side (RHS) expression to the abstract/concrete input and output and/or its type inhabitants.

The use of keywords and identifiers allows for the user to effectively communicate to the back-end generators.

Listing 3.1: Minimal DSL Template

```
1 `fname` {
2    pure { expr_{HS-bool} }
3    nond { expr_{HS-bool} }
4    spec { oa := expr_{HS} ;
5    }
6    absf { ia : τ_{ia} ;
7    }
8    rrel { oa : τ_{oa} ;
9    }
```

```
10    welf {
11    }
12 }
```

The code snippet in Listing 3.1 is a example template that demonstrates how to define a boilerplate test file using the DSL.

pure and **nond** must be *Boolean* and act as toggles, this is required. **spec** is where to define the function correctness specification, and it is also required, however it is just placed directly into the generated file, defined in the DSL for convenience. **absf** is where to define the **ic** and **ia** types. Only **ia** is required. A definition for abstraction function can also be given with the definition operator. **rrel** is where to would define **oc** and **oa** types, only **oa** is required. A predicate may be defined for the relation, using the predicate operator, however this is left out for simplicity. **welf** is where to define how to create well-formed test data, using the definition and predicate operators. This is hidden for simplicity.

Each keyword block in the template can be provided a custom definition, similar to **spec**, however, this is not displayed for simplicity. But, essentially all extra customisability is done with the same syntax, with different semantic interpretation for each context (which is specified by the keyword with all link up to their respective generators).

PBT description DSL Example: *averageBag*

The DSL is used to specify a test for the *averageBag* function, from the *Bag* program. The test file using DSL is displayed in Listing 3.2 and it is a little more than the boilerplate definition, but the extras are just to improve readability. **spec** : **ia** $\rightarrow$ **oa** is the definition of the correctness specification for this function. **absf** : **ic** $\rightarrow$ **ia**, only the types are defined, the abstraction being "direct". **rrel** : **oc** $\rightarrow$ **oa** $\rightarrow$ **Bool**, only the types are defined, the relation being "direct" comparison. **welf** : **ic**, nothing is defined will default to arbitrary.

DSL captures all this information depicted in refinement in Figure 2.3. That is the types and the specification, which is the bare minimum that required for testing. But more customisability is available, as discussed in section 3.3.2.

Listing 3.2: Example Test file with DSL

```
`averageBag` {
   pure { True } nond { False }
   spec { ia : (Int, Int);
          oa : Maybe Int;
          oa := if fst ia == 0 then Nothing
                else Just $ snd ia / fst ia;
   }
   absf { ic : Bag{count,sum};
          ia : (Int, Int);
   }
   rrel { oc : < Failure | Success U32 > ;
          oa : Maybe Int;
   }
   welf {
   }
}
```

PBT Description DSL Customisability

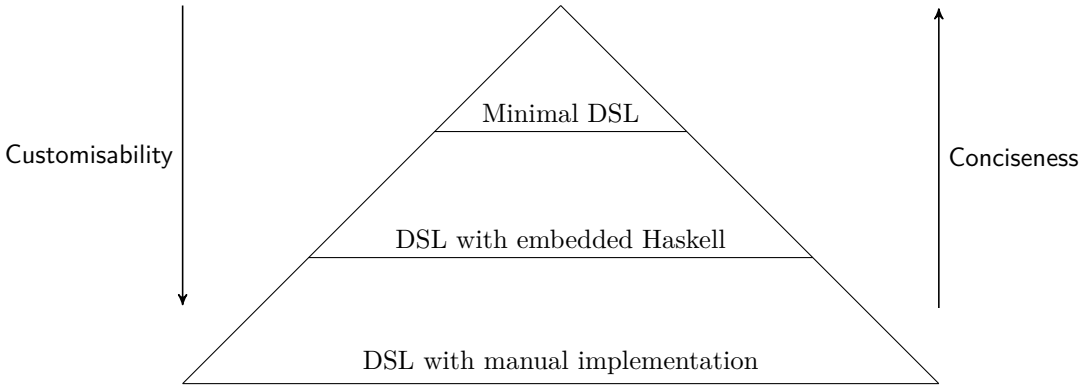

Figure 3.3: PBT Description DSL Customisability vs Conciseness

As discussed custom definitions are possible within DSL.

Rather than tacking on these features to the DSL, it was decided to have allow Haskell syntax to be used in the DSL. Figure 3.3 demonstrates the relationship between conciseness and customisability of the DSL, with the blocks representing the different levels that are available. The user can either just write a minimum definition for testing, just as in example shown in Listing 3.2, or Haskell syntax can be embedded within the DSL. This provides more

customisability. Moreover, it is even possible to completely define the manual test with the DSL. This allows the DSL to replace manual testing completely because edge cases that don't suit the automation can be handled with more verbose definitions in the DSL.

In Figure 3.3 the top level is the minimum definition and the bottom is for when falling back on manual implementation of the test file is the only option. In the middle block, the granularity of the definitions available becomes apparent. This fine grain control is achieved with the HASKELL *Lens* library. The deep integration of the library allows for increased conciseness of definitions for the front-end DSL and also provides perks to the back-end which helps streamline generation.

3.3.3 *Lens* Integration

The Role of *Lens*

Lens syntax is used in two components of the implementation: *Lens* is exposed to the user for usage from within the DSL, that is: user can refer to type inhabitants from within the DSL using *Lens view* expression. *Lens* is also used in back-end generators, for extracting type inhabitants.

The essential problem of the generators is how can type inhabitants of the shallow embedding types be extracted uniformly. This "unpacking" needs to be performed in order to work with the types, and it needs to be done arbitrarily and systematically, no matter the nesting of the type. *Lens* are being used to do this as *Lens* expression are unique and can be built up only requiring field names/tuple positions (which is exposed to the generator module by the compiler).

In terms of the *Lens* syntax exposed to the user within the DSL, this consists of two parts: *Lens* syntax on LHS of DSL operator and *Lens* syntax on RHS of DSL operator.

On the RHS of the DSL operator the expression is HASKELL expression and therefore *Lens* can be used as normal. It is encouraged to be used here as it is concise and powerful.

On the LHS of the DSL operator the *Lens* expression can be used to transform the key identifier (**ia**, **ic**, **oa**, **oc**) and thus allows for the key identifiers type inhabitants can be bound to DSL operator RHS expressions. This extends the key identifiers allowing the user to write a definition or predicate for any of the type inhabitants by using the syntax to transform the key identifier

to extract the type inhabitant. The snippet Listing 3.3 showcases the usage of *Lens* view syntax on the LHS.

Listing 3.3: Example using Lens view expression in DSL

```
`averageBag` {
   ...
   welf { ic : Bag{count,sum} ;
          ic^.count := choose (0, 1001) ;
   }
}
```

In both cases of usage in the DSL the *Lens* allows for COGENT types to be manipulated, however under the hood the shallow embedding is the actual type being manipulated. Thus, the *Lens* syntax assists in ensuring these operations can occur at the DSL abstraction level, hiding the shallow embedding details. Resulting in nicer experience for the user.

Putting all that together: *averageBag*

Listing 3.4: Example DSL: Putting it all together

```
`averageBag` {
   pure { True } nond { False }
   spec { oa := if ia^._1 == 0 then Nothing
                else Just $ ia^._2 / ia^._1;
   }
   absf { ic : Bag{count,sum} ;
          ia : (Int, Int);
   }
   rrel { oc : < Failure | Success U32 > ;
          oa : Maybe Int;
   }
   welf { ic^.count := choose (0, 1001) ;
   }
}
```

The snippet in Listing 3.4 is a test file using the DSL with *Lens* syntax integrated. In the **spec** block, an RHS example is shown being used in the expression for the specification. In the **welf** block, an LHS example is shown being used for defining a well-formed test data generator

(*choose*) for the type inhabitant *count*. The *sum* type inhabitant is not specified and this results in generators completing the definition (just arbitrary).

3.3.4 Code Generators

This section provides an overview of the generator implementation.

The generator for the refinement property is straightforward and did not require an algorithm but rather uses the **pure** and **nond** toggles and essentially builds the function expression from a AST template.

The generators for the abstraction function (**absf**), refinement relation function (**rrel**) and the test data generator function (**welf**), all required a systematic way of building up ASTs for function expressions.

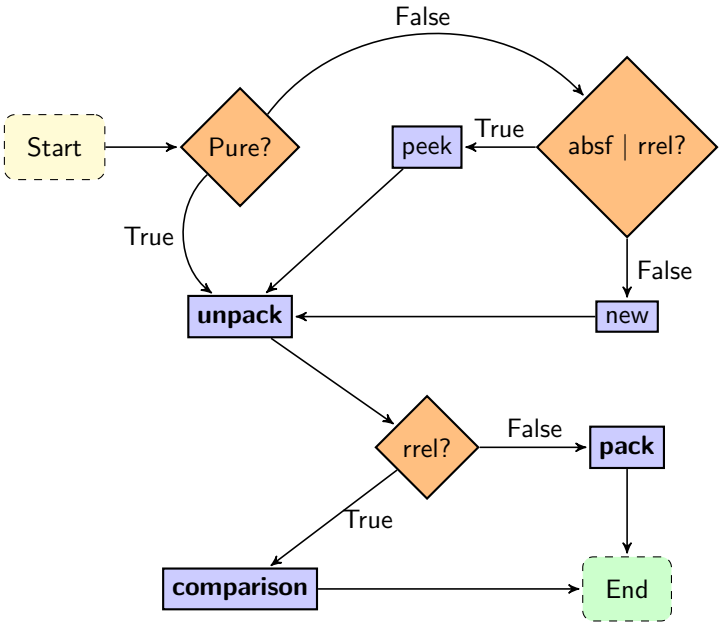

Figure 3.4: Generator Algorithm Flowchart

All functions are generated in the same way in the back-end, building up Haskell abstract syntax trees for the expressions of the functions, using the Haskell AST builder library Burton 2021. The general algorithm consists of main two stages, and the Figure 3.4 displays a flowchart representation of the algorithm.

Diamond blocks represent decisions, rectangle blocks represent processes, and rectangles with curved edges indicating the beginning and ending of the algorithm.

The first thing the generator algorithm must consider is if the function under test is a pure COGENT function or if it is an impure function, and thus is implemented via the C FFI. If the function is pure then no additional logic needs to wrap the expression and therefore moves onto the next stage. If the function is impure then the function must be wrapped in HASKELL C FFI functions that handle calling C functions from within HASKELL. The wrapper will essentially either refer to an object in C land (i.e. using **peek** process) or do some memory allocation through this interface (i.e. using **new** process).

The next stage, is the **unpack** stage, which involves building *Lens* expressions to extract all type inhabitants and then binding them to variables to be used in the function. After this the algorithm checks if it's building a relation **rrel**. If so, then the **comparison** process is engaged where the comparison expression is point wise equality between type inhabitants. If not, the **pack** process is invoked and this involves populating the output constructor with variables.

To summarise, C FFI logic is automatically built if needed; then expressions are built with two parts: **unpack** expression and **pack** expression (or **comparison** for **rrel**). This is done systematically as shown in the flowchart displayed in Figure 3.4.

Generating for C FFI functions

Generating for pure COGENT function involves building up expressions with lens, and this can be useful, however C FFI functions benefit much more from the DSL and generation.

Listing 3.5: Example Test file using DSL for `wordarray_create_u8` function

```
1  `wordarray_create_u8` {
2     pure { False } nond { False }
3     spec { ia : Word8 ;
4            oa : Maybe (Array Word32 Word8) ;
5            oa := if ia == 0 then Just (array (1,0) [])
6                  else Just (listArray (0, fromIntegral (ia-1)) (repeat 0));
7     }
8     absf { ic : (SysState, U32) ;
9            ia : Word8;
10           ia := ic^._2 & fromIntegral ;
11    }
12    rrel { oc : < Error SysState | Success (SysState, WordArray U8) > ;
13           oa : Maybe (Array Word32 Word8) ;
```

```
14          :| (isJust (oa^? _Just) && isJust (oc^?_Success)) ||
15                   (isNothing (oa^?_Just) && isNothing (oc^?_Success)
16                        && isJust (oc^?_Error)) ;
17   }
18   welf { ic^._2 := choose (0, 4095) ; }
19 }
```

This section details the generation for FFI functions, focusing on generating the well-formed test data generator for the example shown in Listing 3.5 This example tests the `wordarray_create_u8` function which allocates memory using C *malloc* function, returning a variant for either successful allocation or failed allocation. In the example, **welf** block, the count field is being defined. The user is defining a test data generator, `choose :: (a, a) -> Gen a`, which randomly picks a value from the given range. The *Lens* expression on **ic** LHS binds the test data generator to the field the *Lens* is extracting i.e. the second tuple index of **ic**.

The generated code for the C FFI from the compiler is a direct translation of functionality from the COGENT FFI to the HASKELL FFI. It results in function input and output types being converted into either a record or a primitive (even variants are converted to records which are tagged enums in C). This means during generation of the testing functions, the algorithm must first analyse the AST of the HASKELL user expression. Then, *Lens* expressions are transformed into *Lens* expressions that will work for these FFI types. This means that the user may write *Lens* expression generically for the COGENT type by composing field names, or in the case of the example tuple index. Then, the code generators will do the back-end wiring to ensure the C FFI interface parameters match up exactly.

3.4 Implementation Trade-offs

Key to implementation was finding a balance between expressiveness and conciseness. A key design decision was to integrate the *Lens* library which is extremely useful, however, its inclusion was spurred due to time constraints as the scoped work was initially too much. Utilising this tool saved timed as accessing operations were taken care of. Moreover, its integration assisted in allowing information to be specified in a scalable manner, as discussed in section 3.3.2, and paved the way to being able to cover a good majority of possible test case scenarios as accessors are generic.

Lens allows for generic accessors for COGENT types defined in DSL. However, there are restrictions: of the functionality discussed in section 3.3.3, LHS functionality is restricted to **welf** block, as it is most useful here.

Another trade off in terms of the types, is that for abstract input and output are restricted to only HASKELL built in types. The provides good cover of test scenarios because the abstract function, and its type should be a more high level type, with less concrete details i.e. removes unnecessary fields, and therefore this should be able to covered by the HASKELL base types (where a rich set of sum and product types are available).

Overall, the trade-offs were necessary for achieving the goals of this thesis and but were advantageous. *Lens* integration powerful and extensible, and limiting scope in terms of the types assisted in delivery of implementation.

Chapter 4

Evaluation

4.1 Results

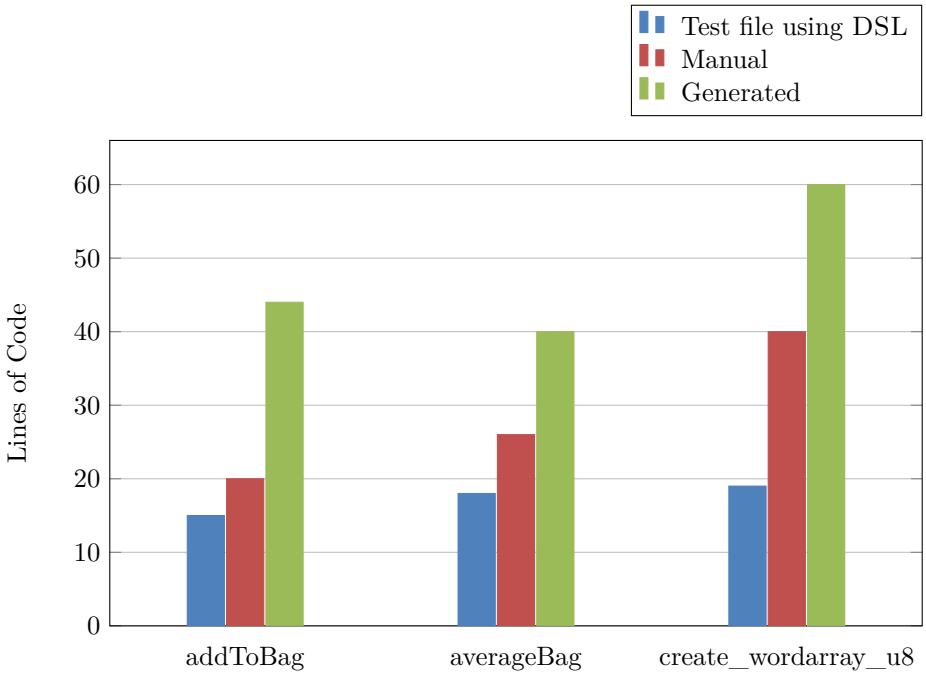

Figure 4.1: Comparison of LOC between PBT test file, generated Haskell test file and manual Haskell test file

The bar graph in Figure 4.1 compares the lines of code of the PBT test file, generated HASKELL file and manual HASKELL file, for a series of functions tested. Comparing the DSL test file against the manual test file, provides a good benchmark for success. It can be inferred from Figure 4.1 that the DSL does in fact result in a reduction in LOC. This bodes well for the framework, and with more tests the trend is expected to continue. The LOC saved by using the DSL to abstract away verbose manipulation of the embedded types, adds up over the course of each of the set up functions: the abstraction function, refinement relation, test data generator and the refinement statement itself.

This bar graph doesn't include imports LOC either, they have been removed to ensure accurate comparison, as it would blow out the manual and generated results by about 50 LOC.

Furthermore, it can be inferred that there the DSL is more beneficial for the more complex types and C FFI functions.

DSL allows for just important details to be defined, and the generators can fill in the rest, making well-founded assumptions that can be made considering the context of the test framework. The DSL syntax has been designed to be lightweight, utilising *Lens* and increasing automating, results in tests files using the DSL to have much fewer LOC than the manual implementation.

4.2 Usability

The benefits of the enhancements are:

- Fewer lines of codes
- Single test file defining correctness and providing a layer of abstraction away from generated code

Moreover, the user doesn't have to worry about:

- The shallow embedding of the Cogent types
- Pointers when testing C functions

The goal of this thesis was to reduce the toil in testing, and this has been achieved without compromising on expressiveness. This implies there has been no degradation in usability from

the enhancements but rather the enhancements improve usability significantly. With the use of the DSL, the user can get up to a test ready state with fewer LOC.

Furthermore, the DSL is flexible enough to cover a good majority of test scenarios, and falling back on manual implementation remains an option available in the DSL. Moreover, the DSL is useful because the user only needs to know how to access their own types not the shallow embedding types.

The shallow embedding types are abstracted away and generic accessing expressions can be used to declare abstraction function, refinement relation, and test data generator. These generic accessors expressions are deeply integrated into the DSL and it allows definitions to be defined on a type inhabitant level.

Additionally, when testing C functions implemented via the C FFI, pointers are automatically unwrapped and exposed for access when testing, and accessing remains generic i.e. extracting fields can be done as if it were a pure COGENT function. This is because, the expressions used for accessing are just chains of field names/tuples indexes that are indifferent to the overall type structure.

Overall, the enhancements make testing more stream-lined, structured and easier to reason about. Initial benchmarks have shown a reduction in LOC required to get up and running. The new PBT framework has improved usability, resulting in it becoming a more viable option for gaining assurance about COGENT and C functions.

4.3 Areas of Improvement

The focus of this thesis was to build the functionality to allow testing of both COGENT functions and C FFI functions with the new DSL. In its current state the project has meet the core requirements, however, there is some need for improvement. Specifically, the following areas be addressed before continuing development:

- The DSL was originally scoped to handle non-determinism, however, due to time constraints, this feature has not been implemented. The details have been summarised in section 5.2.
- Scope of abstract input and output HASKELL types able to be defined in the DSL is limited to the HASKELL built-in types, e.g. Tuples and Maybe/Either optional

4.4 Summary

The enhancements made to the PBT framework meet a good majority of the requirements. The DSL is intuitive and usable, while still maintaining expressiveness, and in tandem with the upgraded code generators make testing a more viable option for gaining assurance about COGENT and C functions. In summary, the overarching goal of enhancing the framework by improving usability has been successfully achieved.

Chapter 5

Conclusion

5.1 Conclusion

This report has explained the motivation and requirements for enhancing the COGENT PBT framework. The existing framework has been analysed and solutions have been evaluated. Ultimately, the solution introduces a DSL for specifying test requirements, and upgrades code generation. The high-level implementation details have be discussed, including the design decisions and areas for improvement. Finally, the enhancements were evaluated in terms of usability and found to have satisfied the expected outcomes.

5.2 Future Work

Following on from this project, the COGENT PBT framework can be further improved and extended upon.

- **Extending AST analysis of user inputs**. Currently, the framework does analysis of *Lens* expressions in DSL, however, this can be further upgraded to enhance code generation.
- **Type classes for organising generated code**. Currently, generated output is messy. If there was a testing type class that contained the functions required for testing, it could

be instantiated with a type that is essentially just the name of the test case, thus the code would be more organised and easy to reason about.

- **Handle non-determinism**. Currently, the DSL is not unable to model non Determinism in the specification.
- **Allow follow-up function**. Currently, testing only allows for one function to be specified to be executed, however it would be useful to allow a follow-up function to be specified. For example when testing a function the allocates memory, the user would need to specify a function to be called that de-allocated this memory.
- **Enhance Refinement Relation with abstraction generator**. Currently, the refinement relation pulls apart each type and does point wise comparison. This can be enhanced by first abstracting the concrete output into abstract output and then equality can be derived for that abstract type. This would require the current abstraction function generator to be integrated into the refinement relation generator.

Bibliography

[1] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. "Lava: Hardware Design in Haskell". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, USA: Association for Computing Machinery, 1998, pp. 174–184. ISBN: 1581130244. DOI: 10.1145/289423.289440. URL: https://doi.org/10.1145/289423.289440.

[2] Lukas Bulwahn. "The New Quickcheck for Isabelle". In: *Certified Programs and Proofs*. Ed. by Chris Hawblitzel and Dale Miller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 92–108. ISBN: 978-3-642-35308-6.

[3] Dan Burton. *Hackage repository of the Haskell Source Extensions library*. URL: https://hackage.haskell.org/package/haskell-src-exts (visited on 04/01/2021).

[4] Manuel M. T. Chakravarty. "C ⟶ HASKELL, or Yet Another Interfacing Tool". In: *Implementation of Functional Languages*. Ed. by Pieter Koopman and Chris Clack. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 131–148. ISBN: 978-3-540-44658-3. DOI: 10.1007/10722298_8.

[5] Zilin Chen, Liam O'Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. "The Cogent Case for Property-Based Testing". In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. PLOS '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 1–7. ISBN: 9781450351539. DOI: 10.1145/3144555.3144556. URL: https://doi.org/10.1145/3144555.3144556.

[6] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279. ISBN: 1581132026. DOI: 10.1145/351240.351266. URL: https://doi.org/10.1145/351240.351266.

[7] Nate Foster, Michael Greenwald, Jon Moore, Benjamin Pierce, and Alan Schmitt. "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem". In: *ACM Transactions on Programming Languages and Systems* 29 (May 2007). DOI: 10.1145/1232420.1232424.

[8] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. "Don't Sweat the Small Stuff: Formal Verification of C Code without the Pain". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 429–439. ISBN: 9781450327848. DOI: 10.1145/2594291.2594296. URL: https://doi.org/10.1145/2594291.2594296.

[9] Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. "Testing Noninterference, Quickly". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 455–468. ISBN: 9781450323260. DOI: 10.1145/2500365.2500574. URL: https://doi.org/10.1145/2500365.2500574.

[10] John Hughes. "Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane". In: *A List of Successes That Can Change the World, Lecture Notes in ComputerScience* 9600 (Mar. 2016), pp. 169–186. DOI: 10.1007/978-3-319-30936-1_9. URL: https://doi.org/10.1007/978-3-319-30936-1_9.

[11] Mohd Khan. "Different Approaches To Black box Testing Technique For Finding Errors". In: *International Journal of Software Engineering and Applications* 2 (Oct. 2011). DOI: 10.5121/ijsea.2011.2404.

[12] Edward Kmett. *GitHub repository of the lens library*. URL: https://github.com/ekmett/lens (visited on 04/01/2021).

[13] Edward Kmett. *Hackage repository of the lens library*. URL: https://hackage.haskell.org/package/lens (visited on 04/01/2021).

[14] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. "Beginner's Luck: A Language for Property-Based Generators". In: *CoRR* abs/1607.05443 (2016). arXiv: 1607.05443. URL: http://arxiv.org/abs/1607.05443.

[15] Daan Leijen, Paolo Martini, and Antoine Latter. *Hackage repository of the Parsec library*. URL: https://hackage.haskell.org/package/parsec (visited on 03/01/2021).

[16] Wojciech Mostowski, Thomas Arts, and John Hughes. "Modelling of Autosar Libraries for Large Scale Testing". In: *MARS@ETAPS*. 2017.

[17] Tobias Nipkow and Gerwin Klein. *Concrete Semantics withIsabelle/HOL*. 2014.

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

[19] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. "Refinement through Restraint: Bringing down the Cost of Verification". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 89–102. ISBN: 9781450342193. DOI: 10.1145/2951913.2951940. URL: https://doi.org/10.1145/2951913.2951940.

[20] Liam O'Connor, Zilin Chen, Partha Susarla, Christine Rizkallah, Gerwin Klein, and Gabriele Keller. "Bringing Effortless Refinement of Data Layouts to Cogent". In: *ISoLA*. 2018.

[21] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. "Profunctor Optics: Modular Data Accessors". In: *The Art, Science, and Engineering of Programming* 1.2 (Apr. 2017). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2017/1/7. URL: http://dx.doi.org/10.22152/programming-journal.org/2017/1/7.

[22] Dr. Praveen Srivastava. "Estimation of Software Testing Effort: An intelligent Approach". In: *Birla Institute of Technology and Science, Pilani, Rajasthan, India* (Jan. 2009).

[23] Dr. Praveen Srivastava, S Kumar, Ajit Pratap Singh, and G Raghurama. "Software Testing Effort: An Assessment Through Fuzzy Criteria Approach". In: *Journal of Uncertain Systems* 5 (Aug. 2011), pp. 183–201.