CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Monadic Programming in Imperative Languages

A framework and reference implementations for implementing monadic programming abstractions within imperative languages

Master's thesis in Computer science and engineering

Joakim Anderlind
Mårten Åsberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Monadic Programming in Imperative Languages

A framework and reference implementations for implementing monadic programming abstractions within imperative languages

Joakim Anderlind
Mårten Åsberg

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Monadic Programming in Imperative Languages
A framework and reference implementations for implementing monadic programming abstractions within imperative languages
Joakim Anderlind
Mårten Åsberg

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Examiner: Robin Adams, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Monadic Programming in Imperative Languages
A framework and reference implementations for implementing monadic programming abstractions within imperative languages
Joakim Anderlind
Mårten Åsberg
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Using monads as a generalised interface for abstracting computations has seen great success in functional languages such as Haskell. By generalising computations as monadic actions, one can use a specialised syntax, such as the `do`-notation of Haskell, to compose these actions into clearer programs. Although certain monadic features have been introduced into imperative languages, such as the various `flatMap` methods in Java or the try-operator in Rust, there is a lack of a generalised interface for composing monadic actions.

The goal of this thesis is to investigate how monadic abstractions from functional languages can be transferred to imperative languages. To accomplish this, we present an interface for composing monadic actions that can be integrated within existing imperative languages.

We begin with defining how our interface should be integrated within a host language while preserving the original semantics of said languages to as large a degree as possible. Then we develop an implementation of our interface in Rust via its proc-macro system. Additionally, we developed a C# implementation of our interface for comparison. Finally, we benchmark the Rust implementation to analyse the potential performance cost of our interface.

Using our Rust and C# implementations, we were able to integrate monads such as the Option, Writer, State, and List monads with minimal changes to existing syntax. Benchmarks shows that the interface itself introduces a noticeable performance cost compared to reference implementations without our interface. For future work, we suggest working on improvements of the usability and developer experience, that our interface may be implemented in other imperative languages, and that more research is needed into improving the performance of our interface.

Keywords: monads, programming languages, language design, Rust, C#

Acknowledgements

We would like to thank our supervisor Magnus Myreen and examiner Robin Adams for their guidance and support during this project. We would also like to extend our thanks to Lukas Andersson for his peer review.

Joakim Anderlind and Mårten Åsberg
Gothenburg, June, 2023

Contents

List of Figures

List of Source Codes

List of Tables

1

Introduction

Modern software development is affected by two opposing forces: our ever-growing expectation of what software should be capable of and the need to reduce the cost, complexity, and time to market, of software projects.

There is no shortage of ideas that attempt to address this dilemma. Paradigms such as microservices and object-orientated programming seek to modularise large projects into smaller, less coupled components to ease maintainability and extendibility. Test-driven development, property-based testing, and formal verification help to ensure that systems are implemented with fewer errors, thus saving resources. Domain Specific Languages (DSLs) are yet another tool for optimising programmer productivity by tailoring a programming language for a specific purpose to minimise the dissonance between business logic and implementation. DSLs are less expressive than general-purpose languages, but decoupling business logic from implementation complexity reduces the cognitive burden on the developer. However, there are several drawbacks that come with using DSLs. DSLs require a significant resource cost to develop and maintain, in addition to the difficulty in acquiring and training developers to become proficient in the language.

A popular approach in programming language design is to incorporate DSLs into general-purpose languages to gain the advantages of a more focused syntax within a familiar environment. Examples include C#'s "LINQ" (Source Code 1.1), Scala's Futures (Source Code 1.2), and Rust's Optional/Result error handling (Source Code 1.3), which all simplify the implementation of paradigms within their respective languages. These language features are often selling points of programming languages, promising that any complexity associated with implementing the chosen paradigm is hidden by the compiler/interpreter.

Features such as these (database query building, asynchronous computations, and value-based error handling) can be implemented via monadic operations [1–3]. First adopted from category theory by Moggi in 1991 [4] as a generalised way of reasoning

```
IEnumerable<int> gradesQuery =
  from grade in grades
  where grade > 80
  select grade;
```

Source Code 1.1: An example of LINQ syntax from C#

```scala
val f: Future[List[String]] = Future {
  session.getRecentPosts()
}

for
  posts <- f
  post <- posts
do println(post)
```

Source Code 1.2: An example of the Future and list comprehension syntax from Scala, taken from the Scala documentation

```rust
fn parse(input: str) -> Optional<Int> {
  let input_value = parse_int(input)?;
  Some(input_value + 3)
}
```

Source Code 1.3: An example of the try operator from Rust

about the semantics of computations, researchers found that monads are capable of acting as an interface for many common concepts in programming. The use of monads as an interface for computations has been thoroughly field-tested in Haskell, which uses a syntax similar to list comprehensions to structure monadic actions into programs. Canonically, monads in Haskell were introduced to enable effectful computations in the otherwise pure language; however, the interface has seen significant use by library creators for structuring computations. Notable examples are the Maybe, State, and Continuation monads, which all extend Haskell with language features accessible through a common monadic interface.

Although most popular general-purpose programming languages already allow impure computations, features such as `async`, database querying, and error management are still desired. Languages such as Scala and F# have derived interfaces similar to Haskell's `do`-notation, both choosing to move away from category theory to become more approachable for developers. Scala's implementation is based on list comprehension, which Wadler [5] proved to be suitable for denoting Monadic computations. F# has support for "Computation expressions", or CE's, which differ from the functional style of its origin. The CE syntax allows for nearly identical syntax to an imperative language with loops and exception management, but the library author defines how the statements are combined in code. The creator of F#, Syme [6], says that he was inspired by Haskell's approach to syntax when researching a way of describing `async` operations, and that this syntax was partly chosen to minimise the cost of rewriting non-monadic code into monadic. It should be noted, though, that while all three languages mentioned so far choose an imperative notation to describe monadic programs, all languages have a functional syntax and their monad interface is implemented as a sub-language within the host.

Note that this project is not the first attempt to introduce support for monadic abstractions into an imperative language. Du Bois and Echevarria [7] incorporated

`do`-notation into Java as an interface to their *State Transactional Memory* implementation, and later Silva Feitosa *et al.* [8] extended *Featherweight Java* [9] with the `>>=` operator of Haskell to program quantum simulation in an imperative language. Libraries such as *do-notation* [10] and *FC++* [11] targeting Rust and C++, respectively, inject `do`-notation syntax into their imperative languages. While these interfaces are sufficiently capable of working with monadic abstractions to varying degrees, each interface departs significantly from its host language. We believe that this separation limits the adoption and integration of existing developers and software projects.

1.1 Problem Statement

We believe the current methods for incorporating monadic abstractions within imperative languages are not integrated well enough into the host language for the feature to be accessible. As such, this thesis gives the following contributions:

- The specification of a language feature for imperative languages that embeds monadic computations into existing code.
- A reference implementation of the aforementioned language feature implemented in Rust.

In addition, we present a smaller C# implementation of our interface produced as a comparison to the Rust interface.

1.2 Outline

The thesis is structured into six chapters and an appendix. The Background chapter will provide the necessary context for the concepts referenced in the thesis. Then the Concept and Implementation chapters will provide our framework's theoretical and practical implementation. In the Results chapter, we present example use cases of the syntax from our framework, showcasing the applications of this thesis's contributions. Finally, in the Conclusion chapter, we discuss the limitations of our framework and implementation, related work, and future work.

2

Background

This chapter provides the necessary background for understanding the following chapters. First, we provide a shallow introduction of the concepts from category theory that monadic programming is based on. Followed by a discussion of programming paradigms focusing on the differences between functional and imperative programming. Lastly, the programming language Rust will be described, looking at its type system and procedural macros.

2.1 Monads and Category Theory

The nature of this thesis focuses more on programming language design rather than formal reasoning around computational semantics. As such, the following section offers a simplified summary of the concepts borrowed from category theory, focusing on its application to computer science. For a more rigorous introduction, see Mac Lane [12].

Category theory revolves around generalising mathematical structures and their operations into *categories* of objects and morphisms between them. In computer science, objects and morphisms translate cleanly to types and functions, and as such, provide an excellent tool for reasoning about computations.

For a given category C with a terminal object and binary products, a monoid is an object A in C with the following morphisims:

$$identity : 1 \to A$$

$$join : A \times A \to A$$

where $join$ is associative and $identity$ is both a left- and right-identity for join.

Functors are morphisims between categories which preserves compositions and identity morphisms. More formally given categories C, G with morphisms f, r between objects in C, a functor F between C and G has the following properties:

$$F(f \circ r) = F(f) \circ F(r)$$

$$x \in Objects(C) \Rightarrow F(id_x) = id_{F(x)}$$

A functor mapping a category to itself is called an endofunctor. Given this we can define a monad M in the category C as an endofunctor with the following natural transformations:

$$unit : I_C \to M$$

$$join : M^2 \to M$$

Where I_C is the identity functor for the category C, and M^2 is the composition $M \circ M$. This definition is similar to the definition of monoids we defined earlier. Indeed, a monad is just a monoid in the category of endofunctors [12].

In computer science monads and functors are usually defined in the category of types, as such are simplified to be more convenient. In Source Code 2.1 we see the definitions for the functor and monad type-classes in Haskell. Here we have the *bind* operation which can be derived from the $fmap$ and *return* operations, conversely the *join* operation can be derived from the *bind* and *identity* functions.

Monads have seen extensive usage as a tool for reasoning about certain computations. Moggi [4] showed that many nontrivial computations, such as nondeterministic, probabilistic, and side-effectful computations, can be generalised as operations on monads. While Moggi was to find a new method of reasoning about the semantics of programs, researchers found that many concepts within computer science translated well to operations on monads.

2.1.1 Monads in Computer Science

Instead of the mathematical definition, monads can be described as a generic type class, trait, or interface that takes one type argument and supports the two operations described in Source Code 2.1, namely `return` and `bind`. The `return` operation takes a value and builds a monadic operation that returns that value without doing anything else. The `bind` operation takes a monadic computation and a function that produces a monadic computation of another value type; the bind applies the function to the existing computation as a continuation, inserting its own logic in between before doing so.

For example, the `Maybe` type as defined in Source Code 2.2, has simple enough monadic functions. The `return` function will build a `Maybe` with the `Just` constructor function. The `bind` function will work differently depending on the input computation; if it receives a `Nothing` it will output a `Nothing`, and only if it receives a `Just` will it use the continuation function to produce the output.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class Monad m where
    return :: a -> m a
    bind :: m a -> (a -> m b) -> m b
```

Source Code 2.1: The operations of Functors and Monads as defined in Haskell

```haskell
data Maybe a = Nothing | Just a

instance Monad Maybe where
    bind (Just x) cont = cont x
    bind Nothing  _    = Nothing

    return x = Just x
```

Source Code 2.2: A definition of the `Maybe` type and its `Monad` implementation

2.2 Programming paradigms

To understand the reasons and difficulties with introducing monads, a concept often seen in declarative languages, to imperative languages, we must understand the differences between declarative and imperative languages. Declarative and imperative languages are two of many different programming paradigms. Programming paradigms are a system used to classify programming languages based on features, such as how the program is executed or how the code is organised.

One language can belong to or contain features from several different paradigms. Such a language is called a multi-paradigm programming language. Many modern programming languages support multiple programming paradigms. For example, Rust has features from functional and object-orientated programming [13] but can not be described as either alone.

Lloyd [14] describes **declarative programming** informally by stating that "[it] involves stating what to compute, but not necessarily how it should be computed". In this paradigm, it is up to the compiler to describe how to do the computations; this, however, can happen at different levels. Functional languages, such as Haskell, are a subset of declarative languages that go relatively close to the processor and force the programmer to, for example, describe that an addition is supposed to be computed and then take over and describe how the addition is supposed to be computed. While other declarative languages, such as Regular Expressions, are satisfied with a description of what to do, such as "find uppercase letters of the English alphabet" (`[A-Z]`), and will translate that into a description of how it is done. Because the programmer cannot describe how the calculation should be done, no side effects can occur, the programmer cannot describe them, and the compiler should not unexpectedly introduce them.

Imperative programming achieves its results through a series of statements that alter the state of the system [15]. By necessity, statements in imperative languages have side effects, such as altering the state of the system. Notable examples include C, Java, C#, and Python.

2.2.1 Pure vs impure

The word "pure" in the field of computer science refers to a deterministic operation without side effects. Pure computations are easier to reason about as the output

depends only on the input. However, creating a program that consists of only pure operations is often not practical, as those do not have any observable effects.

This was a problem in Haskell as the language is entirely pure, which made interacting with Haskell programs difficult. Although there were many candidates for interfaces capable of accessing and manipulating impure actions, the IO monad was chosen as the canonical solution [16].

2.3 The Rust Programming Language

Rust is a general-purpose, multi-paradigm programming language designed for performance and safety. It has a robust type system and a powerful macro system. Rust is a relatively young language, with the first stable version released in 2015 and, at the time of writing, is at version 1.69 [17]. According to the *Stack Overflow Developer Survey 2022* [18] Rust is the most wanted and loved language among developers.

2.3.1 The Type System

Most type systems are seen as a form of a *lightweight* formal verification used to prove the absence of certain classes of bugs [19].

The type system of Rust has many features that can be expected of functional languages.

- Strong type inference, which means that type specifications are rarely needed outside of item definitions, such as types and functions.
- Algebraic data types (ADTs), which are data types formed by other types through algebraic operations. In Rust, this appears as enumeration types, which are tagged unions constructed by the sum of other types.
- Instead of interfaces, as many OOP languages have, Rust has traits similar to Haskell. Rust traits can have associated types and also *generic* associated types (GATs) [20].

Associated types are a way for trait implementations to define related types, with GATs being the generic progression of associated types. The trait declares that an associated type is used, and an implementation of that trait defines which concrete type it is. A GAT can be used for cases where the associated type needs to depend on another type defined at the point of use. GATs allow associated types to be, as the name implies, generic. This can make the type system feel more flexible while still enforcing type-checking at every point.

2.3.2 Procedural macros

Rust features a powerful macro system in two parts, declarative macros and procedural macros. Declarative macros are built as a separate built-in declarative language that transforms the textual tokens of the language. It is used to build simple yet

powerful macros to reduce boilerplate code or create nicer-looking syntax. On the other hand, procedural macros are fully featured Rust functions and are perfectly suited for implementing macros that require more complex logic.

There are three ways in which a procedural macro can be used.

- **Function**-like macros. These look like function invocations, take the text provided to them as "arguments", and are replaced by their output. Any text can be given as "arguments", with the condition that it is surrounded by a pair of matching brackets. An example can be seen in Source Code 2.3.

  ```
  println!("Hello, world!");
  println!("Hello: {}!", user_name);
  ```

  Source Code 2.3: The `println!` function-like macro uses macro features to take a varying number of arguments, something not possible for regular Rust functions

- Custom **derive** macros. These macros take in the text of a type declaration, such as structs or enums, and output additional items, most often a standard trait implementation. An example can be seen in Source Code 2.4.

  ```
  #[derive(Debug)]
  struct Example;
  ```

  Source Code 2.4: The derive macro for `Debug` automatically implements the `Debug` trait for the `Example` struct

- **Attribute**-like macros. These can be applied to any item, such as functions, type declarations, and others, and transform that item. The item and, optionally, the arguments are provided to the macro and are entirely replaced by the output. The item must be syntactically valid Rust code even before the macro transforms it. An example can be seen in Source Code 2.5.

  ```
  #[example_attribute(arguments)]
  fn transform_me() { }
  ```

  Source Code 2.5: An attribute-like macro

Notice the drawbacks stemming from macros only working on the syntactical level of a Rust program. This means that macros do not have access to additional information that the compiler might have, such as an AST[1] or typing information. An AST can be constructed from the text tokens, usually with the help of a library, but nothing can be done to learn the types without access to the complete source of a program and its dependencies.

2.3.3 Try-op

Value-based error handling is the recommended paradigm for managing recoverable errors in Rust. This can be accomplished with the `Option` or `Result` ADTs, which

[1] abstract syntax tree

encode the possibility of failures at the type level. Compared to exceptions (as seen in C++ and Java), which can complicate reasoning about computer programs, this method avoids hidden control flow manipulations. The trade-off is that extracting a value from a possibly incomplete operation requires the developer to manage both possibilities. This can be cumbersome when combining multiple operations, which may fail.

Rust provides the `?`, or the Try-Operator, to simplify this use-case. By adding a `?` as a suffix on a value of type `Option` or `Result` in a function that returns the corresponding type, the inner value can be accessed without an explicit check that the computation that produced the value was successful. If any `Option` or `Result` value evaluates to a failure, the function returns early with the faulty value.

Viewing `Option` or `Result` as Monads, the `?` operation behaves very similarly to the binds defined for the Maybe and Either monads in Haskell. The interface of the `?` operator is accessible to developers as the `Try` trait. However, the interface is designed for control flow semantics and cannot be used to implement a fully-featured monadic bind operation.

3

Concept

This chapter introduces the proposed framework from a language-agnostic perspective. We discuss how the framework builds upon existing systems and how certain features common to imperative languages require special care when implementing the framework. How such cases were resolved in the reference implementation is described in the Implementation chapter.

3.1 Core Idea

We propose a language extension for imperative languages that merges monadic computations and the underlying language into one system. In essence, the extension incorporates the capabilities of Haskell's `do`-notation and F#'s CE into the host language while preserving the underlying syntax's semantics. As an example, look at the verbose code in Source Code 3.1 that deals with Javas `Optional` type and then see how much cleaner it looks with a syntax sugar for monadic binds.

The core concept revolves around the fact that the notation for monadic computations in both Haskell and F# are imperative by nature. In both cases, the notation is de-sugared from the imperative notation to a chain of bind operations and closures. The proposed extension expands upon this process by de-sugaring a complete imperative language into a chain of binds and closures. This is done by translating the common features of imperative languages (loops, branching, returns) into operations on monads (see Appendix A).

Normal syntax

```
var input = getInput();
if (!input.isPresent())
    return Optional.empty();
var result = compute(input.get());
if (!result.isPresent())
    return Optional.empty();
return request(result.get());
```

With bind syntax

```
var input = getInput()?;
var result = compute(input)?;
return request(result);
```

Source Code 3.1: Javas `Optional` type requires tedious checking at each step, which can be overcome with a bind syntax

Haskell's `do`-notation

```
func = do
    a <- actionOne
    b <- actionTwo
    actionThree a b
```

Pseudo Imperative Language with the extension

```
function func() -> M a {
    let a = actionOne?
    let b = actionTwo?
    return actionThree a b
}
```

Source Code 3.2: The resulting imperative code is similar to Haskell's `do`-notation.

Monadic code

```
10 + a_monadic_value?
```

Transformed code

```
a_monadic_value.bind(|temp_name| {
    10 + temp_name
})
```

Source Code 3.3: The transformation of a bound value in a simple expression

Semantically, the extension functions similarly to Haskell's `do`-notation, as seen in Source Code 3.2. Values can be "pulled out" of monadic computations via bind expressions, here shown as the `?` operator, then passed along to other operations of the same type, and finally joined into a single monadic action.

The extension is more similar to F#'s CEs regarding expressivity. Like CEs, the extension allows for imperative language features such as loops and early returns when creating monadic programs. However, unlike CE's, the functionality is all derived from the monads' bind and return operations instead of each feature being implemented individually by the developer. This lessens the burden when implementing the interface for the extension and makes using the extension consistent between monads.

3.2 Binds as expressions

As we have taken great inspiration from the try operator of Rust, we decided that binds should be treated as expressions. This means that in contrast to Haskell's `do`-notation and F#'s CE's statement-like binds, our binds can be used in a more flexible way. However, it also means greater care must be taken when implementing certain expressions and language constructs.

Most often, when binding an expression, the actual binding can occur right before the result is used. The example in Source Code 3.3 shows how a binding in a simple expression can be transformed.

Monadic code

```
a_bool || a_monadic_value?
```

Transformed code

```
a_monadic_value.bind(|temp_name| {
    a_bool || temp_name
})
```

Source Code 3.4: A faulty transformation of a bound value in an or-expression

Monadic code

```
a_bool || a_monadic_value?
```

Transformed code

```
if a_bool {
  Monad::ret(true)
} else {
  a_monadic_value
}
.bind(|temp_name| temp_name)
```

Source Code 3.5: A correct transformation of a bound value in an or-expression

Now, consider the example in Source Code 3.4, if `a_bool` is `true` you would not expect the effects of `a_monadic_value` to have been run. However, the naive implementation of extracting all binds before the expression would cause the effects to be run either way. A smarter implementation could look like the transformed code in Source Code 3.5, where care is taken to run the effect only if its value is needed.

There are more operations that need similar special care, such as the closely related *short-circuiting and operator* (`&&`).

3.3 Handling Imperative Language Features

Similarly to expressions, some statements cannot simply be transformed by extracting bound expressions and binding them before the statement itself. The most notable of such statements are blocks, conditionals, and loops.

Blocks require the least special handling of the three and can be treated much like the body of a function. It must be remembered only to bind the following statements to the result of the block. If blocks can evaluate to values, such as in Rust, this is only a matter of providing the block as an argument to the bind function. If however, blocks can not evaluate to values, special care must be taken to inject the following statements into the binds in the block.

`if` statements can appear in a few different varieties, and depending on which form the statement takes, it will require different special handling. In all forms of `if` statements, the "then" and "else" branches can be transformed into and then treated as blocks, regardless of their initial form. For `if` statements with a single unconditional "else" branch, no additional special handling is required. For `if` statements without any other branches, an "else" branch can be created that, depending on the implementation of binding in the "then" branch returns a monadic action with the unit type as the result value. Finally, for `if` statements followed by one or more `else if`, the conditions must be given special treatment, or they may be monadically evaluated out of order, as shown in Source Code 3.6.

The transformation in Source Code 3.6 shows that without special handling the condition of the `else if` will be executed even if one of the preceding conditions were `true`. To handle this, the `else if` must be broken up into an else branch with a block containing the `if` statement; how this transformation is applied can be seen in Source Code 3.7. When this split is done, the else branch will be a monadic action,

Monadic code

```
if a_bool {
    // ...
} else if a_monadic_value {
    // ...
} else {
    // ...
}
```

Transformed code

```
a_monadic_value.bind(|temp_name| {
    if a_bool {
        // ...
    } else if temp_name {
        // ...
    } else {
        // ...
    }
})
```

Source Code 3.6: A faulty transformation of a bound value in the condition of an `else if` statement

Monadic code

```
if a_bool {
  // ...
} else if a_monadic_value {
  // ...
} else {
  // ...
}
```

Transformed code

```
if a_bool {
  // ...
} else {
  a_monadic_value
    .bind(|temp_name| {
      if temp_name {
        // ...
      } else {
        // ...
      }
    })
}
```

Source Code 3.7: A correct transformation of a bound value in the condition of an `else if` statement

and as a consequence, the then branch must be updated to match; this responsibility can be put on the end user as the type of the monadic action in the then branch might not be known. In many languages, `else if` is simply an else branch with a single `if` statement, and thus the "splitting" will be taken care of by ensuring that all branches are blocks.

Loops with monadic actions in their bodies can be handled in multiple ways. One idea is to keep a variable that, for each iteration of the loop, is assigned the result of binding the body's action to the variable; after all iterations, the statements following the loop can be bound to the variable. In addition to other issues, such as closing over loop variables [21], this means that the entire loop has to be run before any bindings can be applied to the inside of the loop. This means that bindings can not extract any intermediate results; e.g., the logging monad could not print anything from the loop before the loop has completed. This would be especially bad when dealing with infinite loops. A better idea is to perform the loop with recursion.

Monadic code

```
for i in 0..10 {
  if i % 2 != 0 {
    continue;
  }
  if i > 5 {
    break;
  }
  println!("{i}");
}
```

Transformed code

```
loop_function(0..10, |i| {
  if i % 2 != 0 {
    return Continue;
  }
  if i > 5 {
    return Break;
  }
  println!("{i}");
  return Continue;
})
```

Source Code 3.8: The transformation of a loop

Performing loops with recursion requires more than what syntax transformation can do. It requires some function that takes the iterator or condition and a closure of the body that should accept any iterator variable as a parameter and return a monadic action of type unit. This function should take the following steps:

1. Begin by taking one step of the iterator or checking the condition, and if it fails, return a moandic unit.
2. Execute the body, bind a closure to it, and return the result.
3. The closure should call the function recursively. That is, if the iterator of condition holds again, the body will be executed again.

Other notable imperative statements that require special handling are control-flow statements such as `return`, `break`/`continue`, and `throw`. Of these, we decided that `throw` is beyond the scope of this project, as monads can provide similar error management, and we failed to design a satisfactory implementation for `return`.

As `break` and `continue` statements only appear within the body of loops, this can be implemented by extending the implementation of the loop function. Instead of having the bodies of loops return a monadic action of type unit, they should return a monadic action with a type that can denote either breaking out of or continuing the loop. Within the body, all `break` and `continue` statements can be replaced by `return` statements that return a value denoting breakage or continuation respectively. These returns will return from the closure of the body and thus skip the following statements within the body. Once the closure has returned, the bound closure in the loop function must check whether or not the loop should continue. One thing to note is that the bodies of loops now must return monadic actions of the new type. This can be done by inserting a `return Continue` statement at the end of the body, which will not affect the normal flow of the program. The transformation of the body of loops can be seen in Source Code 3.8.

4

Implementation

This chapter covers the work done on writing the Rust reference implementation of the concept discussed in the previous chapter. We begin by describing how monads have been structured in Rust, before continuing with a deep dive into how the syntax of Rust was amended to include a syntactical sugar for monadic binds. Along the way of writing the Rust implementation, we encountered some difficulties, and so this chapter will finish by discussing a reimplementation of some features in C# for comparison.

4.1 Monads

The original plan was to write a one-to-one direct translation of the three type-classes (`Functor`, `Applicative`, and `Monad`) that make up monads in Haskell. However, that idea was soon scraped after we realised that it would contribute anything important to the project other than the satisfaction of being close to the mathematical origins of monads. For this, we only require the `Monad` trait in our Rust implementation.

The `Monad` trait is defined with a "static" `ret` function used to return pure values and an associated `bind` function used to compose an action with a continuation. A simplified version of the `Monad` trait can be seen in Source Code 4.1. Notice also the GAT `Bind<B>`, which allows those that implement monads to define the result of the bind operation, usually defined as a variant of the same type, without locking users to returning values of a specific type.

The "continuation function" `f` that the `bind` function takes implements the trait `FnMut`, which is a type of closure that can be called multiple times and might mutate its captured variables each time. This is in contrast to the `Fn` trait, which are pure functions that can be called multiple times, and the `FnOnce` trait, which are functions that can consume their captured variables and thus only be called once. `FnMut` is chosen because it allows us to keep "imperative" features, such as mutating variables, while gaining the most from the monadic interface, such as calling the continuation multiple times in a list comprehension monad.

With this definition of the `Monad` trait, Source Code 4.2 shows how Rust's `Option` can implement it. Notice that the binds "continuation function" `f` is required to implement `FnOnce` instead of `FnMut` as in the trait definition. This is because

```
trait Monad<A> {
  type Bind<B>;

  fn bind<B, F>(self, f: F) -> Self::Bind<B>
  where
    F: FnMut(A) -> Self::Bind<B>;

  fn ret(a: A) -> Self;
}
```

Source Code 4.1: A simplified version of our `Monad` trait

```
impl<A> Monad<A> for Option<A> {
  type Bind<B> = Option<B>;

  fn bind<B, F>(self, f: F) -> Self::Bind<B>
  where
    F: FnOnce(A) -> Self::Bind<B>,
  {
    match self {
      Some(a) => f(a),
      None => None,
    }
  }

  fn ret(a: A) -> Self {
    Some(a)
  }
}
```

Source Code 4.2: A `Monad` implementation for `Option`

`Option`s definition of `bind` runs only `f` at most once, and it is possible because `FnMut` implements `FnOnce`.

4.2 Bind sugar

Before the start of this project, we had looked at the macro-systems and related technologies, such as transpilers, for several languages. We concluded that the proc-macro-system of Rust was the best fit for our purposes. Some of the other systems and languages considered were ESBuild [22] or babel [23] for JavaScript/TypeScript, but both were deemed too cumbersome to work with. As well as SourceGenerators [24] for C#, which provides more features than proc-macros, such as access to type information, but were ruled out because they lack one important feature, the ability to alter existing source code.

Original code

```
#[monadic]
fn monadic_fn() -> Option<i32> {
    let a = 20 + Some(12)?;
    Option::ret(a + 10)
}
```

Macro expanded

```
fn monadic_fn() -> Option<i32> {
  Some(12).bind(|__temp_ident_0| {
    let a = 20 + __temp_ident_0;
    Option::ret(a + 10)
  })
}
```

Source Code 4.3: Illustration of how binds are expanded

As mentioned in subsection 2.3.2 (Procedural macros), Rust proc-macros only receive the textual tokens of an item, in our case a function, and only through libraries such as *syn* [25] can we retrieve the AST of items. With the text tokens parsed to an AST, we could begin to consider how it should be transformed. We began by looking through the type tree of the AST found in *syn* to find nodes where we thought it would be easier and, respectively, more challenging to introduce the monadic bind around.

We quickly learnt that almost everything in Rust is an expression, and soon after, we learnt that this makes things quite easy. Each block, starting with the "body block" of the function, is processed in a series of steps.

1. We begin by moving all statements that define new items, such as types or functions, to the top of their block. This allows the items to be in the same scope as before, even after later binds introduce new blocks with new scopes.

2. Next, all other statements are gone through, and their expressions are visited recursively. If any monadic bind operator is encountered, it is removed and replaced by a temporary identifier, and the monadic expression is placed in a queue. If any block is encountered while visiting an expression, such as the "then"-block of a `if`-expression, the statements in that block are visited just like the "body block" of a function.

3. When a statement that contained a bind has been visited, the queue is iterated through, creating a bind for each monadic expression. The "continuation function" is constructed to contain the rest of the statements. If there were multiple binds in a single expression, the next "continuation function" will be the bind-expression of the previous one.

In Source Code 4.3, a simplified illustration of what the binding transformation can look like is shown.

4.2.1 Implementing imperative language features

As discussed in section 3.3 (Handling Imperative Language Features), some features, and here AST nodes, require special handling. The handling of those features has been thoroughly explained in that section, and for the most part, the Rust implementation is easily derived. This section will cover some of the more interesting details of the Rust implementation.

```
error: Explicit returns are not supported, please use implicit returns
  --> macro_test_cases/early_return_fail.rs:34:3
   |
34 |         return Identity(false);
   |         ^^^^^^^^^^^^^^^^^^^^^^
```

Source Code 4.4: A custom compiler error for early returns

```
1 fn loop<F: FnMut() -> M, M>(mut body: F) -> M {
2     body()
3         .bind(move |()| {
4             loop(body)
5         })
6 }
```

Source Code 4.5: Pseudo-Rust presenting the ownership problem of loop-functions

Banned nodes, such as the early return statement, can be handled especially elegantly in Rust. In the proc-macro, a compiler error with a helpful error message can be inserted at the location of banned nodes. The Rust compiler will then point to the node and print the message, an example of which can be seen in Source Code 4.4.

The Rust implementation of the recursive loop functions contains `unsafe`-blocks, but that is not to say that it is necessarily unsafe code. The `unsafe`-blocks allow copying the body closure to bend the rules of Rusts borrow-checker.

Source Code 4.5 shows a simplified representation of how a loop function could look in Rust so that we can discuss the ownership problem overcome with the `unsafe`-blocks. The `loop` function begins, on line 1, by taking ownership of the `body` closure. The `body` is then called on line 2, borrowing it mutably. On line 3, an unnamed closure is constructed and bound to the result of calling the `body`, the `body` closure is moved into this closure which takes ownership of it. To take the next step in the loop, the `loop` function is called recursively on line 4. At the first invocation of the unnamed closure, it gives up ownership of the `body` to the call to the `loop` function, after which the closure can not be called again, this is the problem.

The solution to this is to insert an `unsafe`-block between lines 3 and 4 that copies the `body` closure and then only give up ownership of the copy while retaining ownership of the original. Other solutions that do not require `unsafe`-blocks, such as having the body closure implement `Clone`. However, closures that implement `Clone` are `Fn` and cannot mutate their captured state. The solution used also comes with limitations; all implementations of `bind` functions must act appropriately. They cannot run several copies of the continuation function simultaneously, asynchronously or in parallel, lest it gives rise to undefined behaviour. When `bind` implementations follow this rule, the loops work well.

4.2.2 A C# implementation

To see how things could work if not for the sometimes restrictive borrow-checker of Rust, we decided to reimplement some features in C#. As mentioned, C#'s SourceGenerators cannot alter source code; however, they can generate new source code based on the contents of auxiliary files. This is why we write the monadic C# code in auxiliary files and have SourceGenerators modify the code before presenting it as new source code. This will drastically worsen the developer experience, and no editor will be able to give full support for C# code in non C# files.

When implementing the C# version, much of the work could be "translated" from the Rust version with minor alterations. Blocks in C# can evaluate to results and thus must work differently than the Rust implementation. The most exciting deviation occurred when loop functions were implemented.

In Rust, statements following a block are bound to the result of the block; this cannot work in C#. Instead, the statements following the blocks are bound to any point inside the block that would have returned a result from the block. This is usually the last statement within the block but can be different depending on the statements within the block.

Without the borrow-checker, loop functions could be implemented in a much more straightforward way. The C# way is almost identical to the Rust way, except the `unsafe` block is unnecessary since C# allows multiple mutable references to any object by default. Nevertheless, even though C# has no problem with our implementation of loop functions, monad creators should still follow the same rules when implementing the `Bind` method.

5

Results

This chapter presents example use cases of the concept and implementation discussed in the previous two chapters. More specifically, this chapter presents various implementations of monads and functions using those monads written in our interface, along with comparisons of functionally equivalent non-monadic code. Finally, benchmarks of the two versions are presented to compare execution time.

5.1 Monad Examples

The monads used to demonstrate the functionality of our interface were chosen via two heuristics: prevalence in existing research in monadic interfaces and their ability to abstract away complexities existing in the reference implementation. First, the Option monad is presented as a method of encoding the possibility of missing values into the type signature of functions, as introduced by Spivey [3] as an alternative to throwing exceptions. Then we demonstrate the Writer and State monads to demonstrate how side-effectful computations can be managed using our interface. Finally, we present the List monad to demonstrate how our interface can simplify complex operations.

5.1.1 The Option Monad

The Option monad consists of the sum type `Option<T>` which in turn consists of the constructors `Some(T)` and `None`. This allows developers to encode the possibility of an absence of a result into the type signature of computations, forcing users to handle possibly missing values before being able to use any existing value. The bind operation maps the given "continuation function" over the inner value if it exists, otherwise the `None` is propagated as the result [3]. The `Option<T>` type has already been introduced into modern imperative languages such as C++, Java, and Rust. While all three of the languages have a bind equivalent operation for their Option type, such as `and_then` in C++ and `flatMap` in Java, only Rust offers support for composing Option computations via a syntactical construct. The complexities of interacting with the Option type can be hidden using our monadic interface, as shown in Source Code 5.1.

Reference implementation

```
var input = getInput();
if (!input.isPresent())
  return Optional.empty();
var result = compute(input.get());
if (!result.isPresent())
  return Optional.empty();
return request(result.get());
```

With monadic abstraction

```
var input = getInput()?;
var result = compute(input)?;
return request(result)?;
```

Source Code 5.1: A comparison of a program using the Option monad with and without our interface

5.1.2 The Writer Monad

The Writer monad [26] describes computations, which produce a stream of data accumulated into a record in addition to the resulting value. Accumulating logs from computations or generating assembly from an AST are commonly abstracted behind the Writer monad in functional languages such as Haskell.

Source Code 5.2 shows how the Writer monad can be used with our interface. It is implemented as a wrapper type around a `Vec<String>`[1].

The use of the Writer monad does not decrease the amount of code needed in this case; however, it moves the responsibility of moving around the reference to the record from the function arguments to the monadic bind. This clarifies that the operation of `startup_nuclear_reactor` does not depend on the current state of the log but rather that the computation will produce a log in addition to the result.

5.1.3 The State monad

The State monad [26] describes computations that read and modify a context. It is often used in functional programming as an alternative to non-local variables in contexts where a more imperative style of state management is more fitting. However, in imperative languages, this use case is often not as necessary since non-local variables are already a common feature, such as global or class variables. Instead, the State monad is helpful as a building block of monads with more complex binding properties.

In Source Code 5.3, we see that the State monad does not substantially improve code readability. Such state management is well supported in modern imperative languages such as Rust and as such the State monad is unnecessary on its own.

5.1.4 The List monad

List comprehensions provide a special syntax for aggregating, filtering, and mapping sequences of data into a single list. Although more common in functional languages, it has seen adoption as “list comprehension” in Python and the LINQ syntax in

[1] A dynamic list of strings

Reference implementation

```rust
fn main() {
  let mut logs = Vec::new();
  let success =
    startup_nuclear_reactor(&mut logs);
  println!(
    "Startup success: {success}"
  );
  for log in logs {
    println!("{log}");
  }
}

fn startup_nuclear_reactor(
  log: &mut Vec<String>,
) -> bool {
  run_safety_checks(log);

  let core_temp = get_core_temp(log);
  log.push(format!(
    "Temp: {core_temp}"
  ));

  core_temp < 200.0
}
```

With monadic abstraction

```rust
fn main() {
  let startup =
    startup_nuclear_reactor();
  let (success, logs) = startup.run();
  println!(
    "Startup success: {success}"
  );
  for log in logs {
    println!("{log}");
  }
}

#[monadic]
fn startup_nuclear_reactor(
) -> Writer<bool> {
  run_safety_checks()?;

  let core_temp = get_core_temp()?;
  Writer::log(format!(
    "Temp: {core_temp}"
  ))?;

  Writer::ret(core_temp < 200.0)
}
```

Source Code 5.2: A comparison of a program using the Writer monad with and without our interface

Reference implementation

```rust
fn program(&mut self) {
  self.play_turn(0, 0);
  self.play_turn(0, 2);
}

fn play_turn(
  &mut self,
  x: usize,
  y: usize,
) {
  self.set_piece(x, y);
  self.switch_current_player();
  self.inc_round_counter();
}
```

With monadic abstraction

```rust
#[monadic]
fn program() -> State<GameState, ()>
{
  play_turn(0, 0)?;
  play_turn(0, 2)
}

#[monadic]
fn play_turn(
  x: usize,
  y: usize,
) -> State<GameState, ()> {
  set_piece(x, y)?;
  switch_current_player()?;
  inc_round_counter()
}
```

Source Code 5.3: A comparison of a program using the State monad with and without our interface

C#. List comprehensions are a specialisation of monad comprehensions [5] and, as such, can be implemented via our interface. In Source Code 5.4, we demonstrate an example of list comprehension in C#.

Note that each statement in the monadic example (Source Code 5.4) and every statement following it will be executed multiple times, once for every element in the bound sequence. This has a few implications. First, each bind will increase the number of iterations the function body will run by a factor of the total elements in the bound sequence. As the complexity of the iterations is hidden from the developer, this presents a non-obvious performance trap. Second, as each bind will be executed along with all the following statements for the number of elements of the

Reference implementation

```rust
IEnumerable<(X, Y)>
  ListComprehension<X, Y>(
    IEnumerable<X> xs,
    IEnumerable<Y> ys)
{
  return xs.SelectMany(x =>
    ys.SelectMany(y =>
      new[] {(x, y)})
}
```

With monadic abstraction

```rust
[Monadic]
IEnumerable<(X, Y)>
  ListComprehension<X, Y>(
    IEnumerable<X> xs,
    IEnumerable<Y> ys)
{
  var x = xs!;
  var y = ys!;
  return Return((x, y));
}
```

Source Code 5.4: A comparison of a program using the List monad with and without our interface

sequence being bound, the order of the binds affects the result of the computation. Changing the order of the two binds in Source Code 5.4 will change the result of the computation, which may be unexpected, as the declarations seem independent if one is unaware of how the comprehension is implemented, but by looking at the reference implementation this becomes clear.

5.2 Benchmarks

To measure the performance cost of our interface with monadic abstractions, we created two Rust programs performing equivalent computations: one using our interface with monadic abstractions and one that did not. We then used our benchmarking suite to find the average execution time of these computations and analysed the differences. We selected the Option monad (subsection 5.1.1), Writer monad (subsection 5.1.2), and the State monad (subsection 5.1.3) as test cases for the benchmarks.

Our benchmarking suite was built around the Rust-based benchmarking library *Criterion*. *Criterion* will run a given function repeatedly over a given time span, periodically sampling the execution time to derive a mean execution time for the function. We set the sample size to 10,000 and the measurement time to 20 seconds. Both versions were compiled in release mode to enable optimisations. Then both versions were executed, and the results were exported to `csv` files. The benchmark results were aggregated and analysed with Python and the libraries *pandas* and *seaborne*. The source code for the Option, Writer, and State benchmarks can be found in Appendix A, B, and C respectively, the benchmark configuration and Python scripts for code analysis can be found in the software repository.

All benchmarks were run on a Linux-based laptop with a *AMD Ryzen 7 5800U* CPU with a maximum clock speed of 4.5 GHz.

5.2.1 Option Benchmark

As our interface and Rust syntax are equivalent when working with the Option type, the reference and monadic implementations are identical apart from the "monadic" decorator added for the monadic version. Each benchmark's computation is to validate that a number, its half, and its quarter are all even numbers and then add these numbers together. The program should return `None` if any of these validations fail. The results of these benchmarks are shown in Figure 5.1 and in Table 5.1.

	Mean	Standard deviation
Reference Benchmark	0.6953 ns	0.0316 ns
Monadic Benchmark	1.4571 ns	0.0970 ns

Table 5.1: Mean and standard deviation of the execution duration measured from benchmarking the Option monad and reference implementation

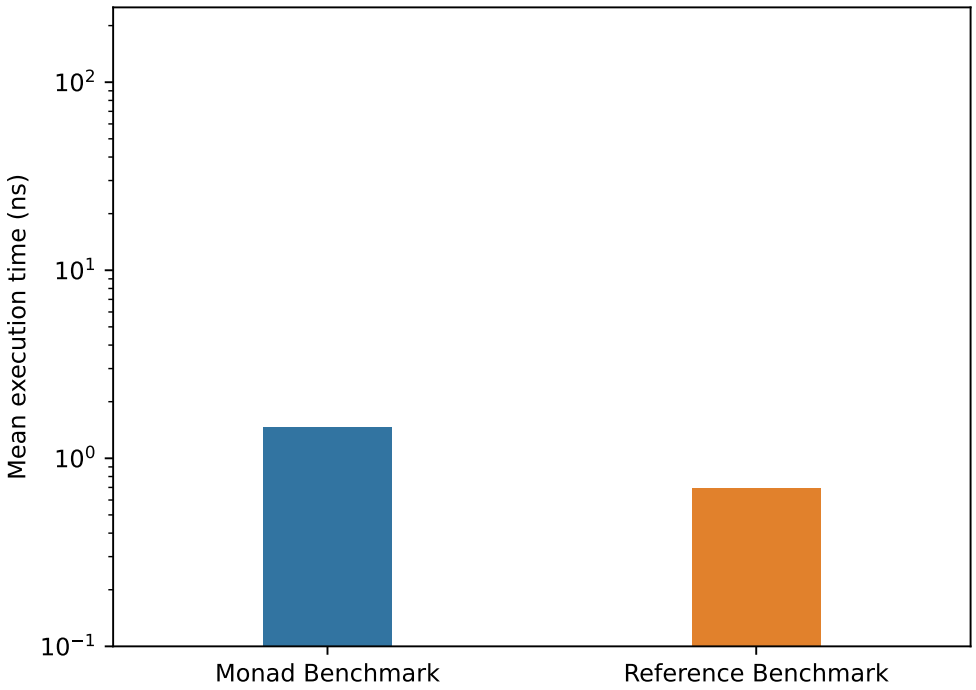

Figure 5.1: A log bar chart showing the measured mean execution time between the Monad benchmark using our interface with the Option monad and the Reference benchmark

5.2.2 State Benchmark

This benchmark consisted of manipulating a struct containing two integers and a floating point. Structurally both benchmarks were similar; however, the state monad needed an extra step of running the composed state monad with an arbitrarily chosen input parameter, as simply calling the function with our interface produces a lazy state action. The results of these benchmarks are shown in Figure 5.2 and in Table 5.2.

	Mean	Standard deviation
Reference Benchmark	1.8805 ns	0.0316 ns
Monadic Benchmark	176.6908 ns	151.6677 ns

Table 5.2: Mean and standard deviation of the execution duration measured from benchmarking the State monad and reference implementation

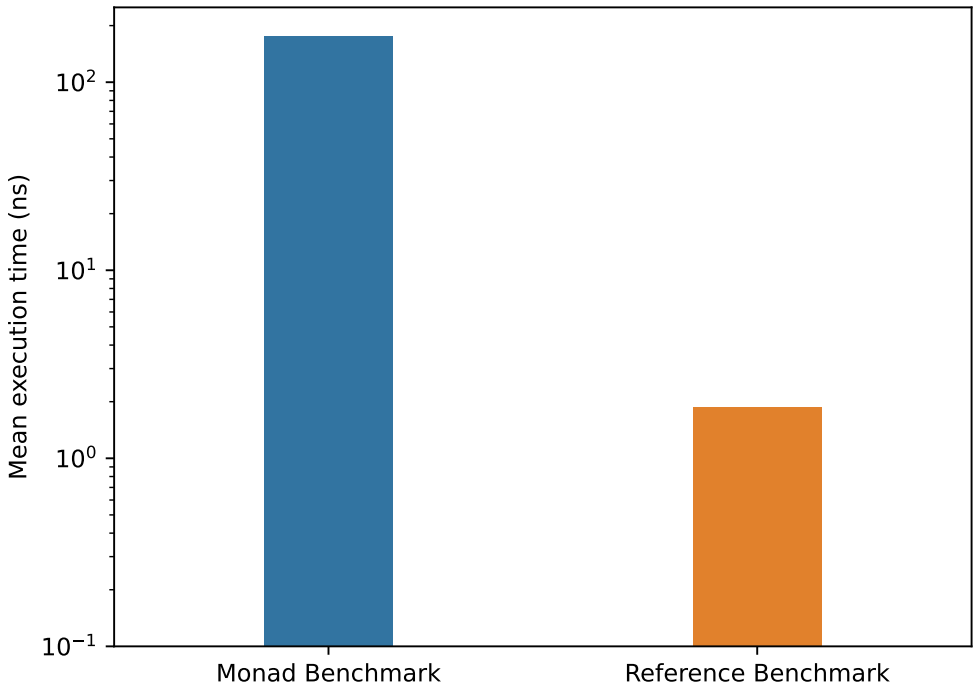

Figure 5.2: A log bar chart showing the measured mean execution time between the Monad benchmark using our interface with the State monad and the Reference benchmark

5.2.3 Writer Benchmark

The writer benchmark consisted of writing static strings and an input parameter to a collection of logs. The reference implementation requires the user to pass a reference to a collection for accumulating the log, while the monadic implementation implicitly creates the collection. The results of these benchmarks are shown in Figure 5.3 and in Table 5.3.

	Mean	Standard deviation
Reference Benchmark	84.61119 ns	74.3226 ns
Monadic Benchmark	218.5688 ns	28.8661 ns

Table 5.3: Mean and standard deviation of the execution duration measured from benchmarking the Writer monad and reference implementation

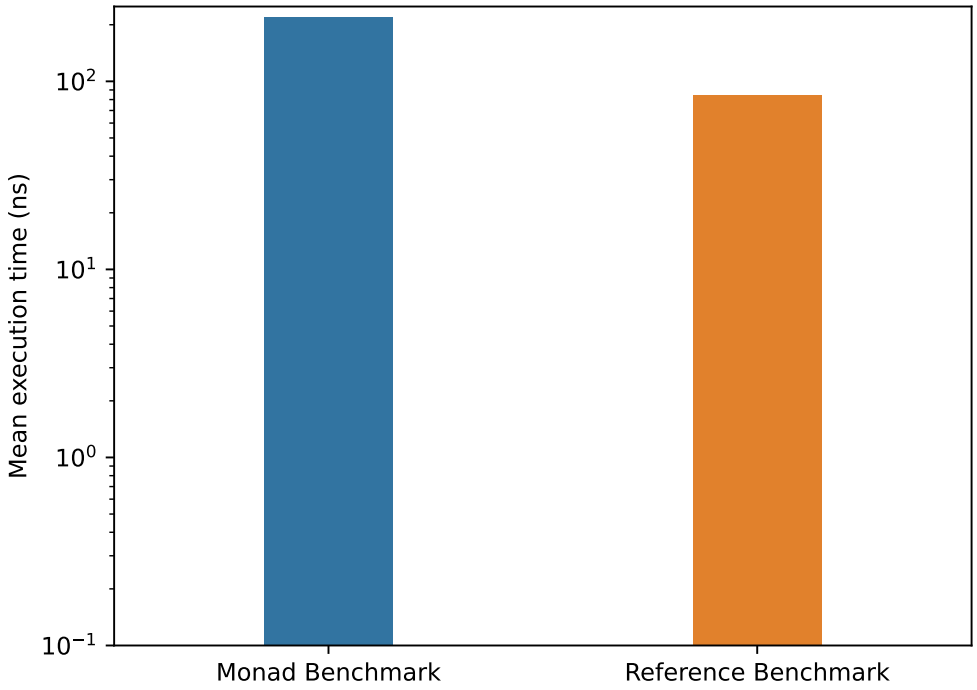

Figure 5.3: A log bar chart showing the measured mean execution time between the Monad benchmark using our interface with the Writer monad and the Reference benchmark

6

Conclusion

In this chapter, we analyse the results of implementing the Rust and C# projects, as well as the project as a whole. First, we discuss important issues of the framework on a conceptual level, then focus on the differences and similarities between the Rust and C# code bases. Then we discuss the results of our benchmarks and possible bottlenecks. Finally, we discuss related work and conclude with suggestions for future work.

6.1 Discussion

The implementation of a monadic bind syntax in both Rust and C# has shown that it is feasible to have such syntax in imperative languages. However, the implementations in this project are not ready to be used in real-world applications. What follows is a discussion of the weaknesses and failures of this project that can provide insights for future works.

6.1.1 Early returns

We faced many challenges when trying to implement early returns within our interface. Early returns are a way of exiting a function before reaching the end, something which is usually done in monadic code by utilising a monad, such as the `Option` or `Result` monads. Due to the difficulties we faced and the fact that they can be implemented with monads, we eventually decided to abandon early returns and refer users to a monadic implementation.

However, we had several ideas for implementing early returns along the way. From the more radical idea of using `panic`s and `catch_unwind`, which are Rust features that allow us to throw and catch "exceptions". This method is feasible; however, we recognise that it is a very problematic idea, its use is discouraged in the documentation [27], and we chose to abandon it. To the more sensible idea of implementing a monad transformer with a monad specific to early returns. Monad transformers are a way of combining different types of monads in a single monadic computation, which could be used to combine any other monad with an "early return" monad. This idea was abandoned when we realised that this would force those who write monads to implement all monads as monad transformers in addition to the "regular" monad implementation, which we considered unreasonable. We also considered solutions

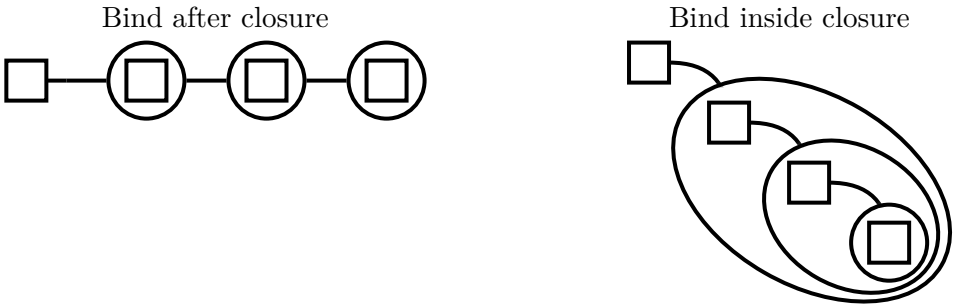

Figure 6.1: The two ways of designing binds, with boxes being monadic values, lines being binds, and circles being closures

from languages other than Rust and found that Kotlin's lambdas, which can return from the enclosing function [28], could have been used, although further consideration is required.

6.1.2 Laziness of binds

When designing binds, there are two ways to chain multiple binds together, either the "next" bind is placed inside or after the closure of the "current" bind. Figure 6.1 shows a visual representation of the two different ways of designing binds. We make this distinction because of Haskell's laziness and how it can or cannot translate to other languages.

Imagine using a monad with a lazy bind function; that is, the "continuation function" is only executed when its result is needed. If the binds are placed after each other, the whole loop must be iterated before even one bind can be executed. However, if the binds were placed inside each other instead, then only one step of the loop would need to be taken before the first bind could be executed.

Where the binds are placed could have huge impacts if iterations of the loop allocated large structures, as they all would need to be allocated at once, or if the loop would be infinite, no steps could ever be taken in such a loop. We suspect the effects are less drastic on monads with non-lazy bind functions, but we have not investigated this further.

6.1.3 Rust vs. C#

When implementing the ideas from this work, the primary focus has been on a Rust version, but efforts have also gone into writing a C# version. Rust was chosen as the primary focus because of its expressive type-system and powerful macro-system, and because it was the primary focus, it is the best developed version. However, that does not mean the C# version lacks advantages. Both versions have their advantages and disadvantages.

The developer experience is much better with the Rust version. This is mainly because the Rust macro-system supports these types of use cases, while C# does

Rust

```rust
let agr = Rc::new(RefCell::new(0.0));
let agr2 = agr.clone();
for i in 0..4 {
  *agr2.borrow_mut() += i as f32;
  Logging::log("loop")?;
}?;
Logging::ret(*agr.borrow() / 4.0)
```

C#

```csharp
var agr = 0.0f;
monadic for (var i = 0; i < 4; i++)
{
  agr += (float)i;
  Logging.Log("loop")!;
}
return Logging.Return(agr / 4.0f);
```

Source Code 6.1: Rust and C# code with mutability across "bind boundaries"

not. The Rust version can be used with almost any tool that works with regular Rust code, while tools for C# treat the monadic code like a plain text file.

Similarities to the original languages are greater in the C# version. When allowing multiple executions of the "continuation function", Rusts borrow-checker will ensure that variables that cross "bind boundaries" are immutable. An example of a way to circumvent this can be seen in Source Code 6.1, where the Rust version requires some advanced features for a seemingly trivial program. While that is similar to how binds work in Haskell, with all variables being immutable at all times, that does present difficulties when expecting Rust semantics. C#, which lacks the borrow checker, does not have this problem, and mutable variables can be used anywhere, if they should be, is another question.

C# also lacks the expressive type system that both Haskell and Rust feature. As stated in subsection 2.3.1 (The Type System), one of the great features of Rusts type system is GATs, a feature that C# lacks. We overcame the lack of GATs in C# by defining both the "current" result type and the result type after a bind at the point of declaration of all monadic actions. This means that one monadic action can only ever be bound to once, which might seem like a problem, but in fact, only is so if the binds are placed after each other instead of inside each other. Any language whose type system lacks GATs will have to overcome it in some way, be it in this or a similar way or with other features of the specific type system.

6.1.4 Performance cost

The results gathered in section 5.2 (Benchmarks) show that using monadic abstractions with our interface in Rust has a noticeable performance cost. The Optional and Writer monad accrued approximately two times the mean execution time compared to their reference implementations, while the State monad execution time increased by a factor of 94. For the Optional and Writer monads, we hypothesise that the main factor behind the increased cost is the difficulty of normalising the multiple nested closures which our interface produces. For the State monad, we hypothesise that the lazy nature and multiple function calls are the leading factors.

6.2 Related work

Inspired by previous research in representing computations as monads [3, 4], Wadler [5] presented a new language feature for structuring monadic computations: *Monad Comprehensions*. Monad Comprehensions are a variation of List Comprehensions popular in many declarative and imperative programming languages, which is in turn inspired by the notation used in Zermelo–Fraenkel set theory [29].

Haskell's `do`-notation was introduced in version 1.3 as a new and more convenient syntax for structuring monadic computations. Originally created by Peyton Jones for the *gopher Haskell* compiler [30], the syntax was introduced to accommodate the introduction of the *IO Monad* [16]. `do`-notation worked similarly to Wadlers *Monad Comprehensions* but with a layout more similar to the imperative syntax of C.

Syne added *Computation Expressions* as a notation for `async` computations in the 1.0 release of the functional programming language F#. Similar to `do`-notation from Haskell, *Computational Expressions* structure monadic computations in an imperative manner. However, computational expressions take an additional step of allowing for more traditionally imperative language features such as `while` loops and `try` exception handling. This development aimed to simplify the transition from "non-monadic code to monadic code" to make the language feature more accessible to developers. This is in contrast to Haskell, where `do`-notation has a very distinct separation from the standard syntax of the language. However, computational expressions require that the implementer must implement the logic behind each type of statement, which could result in unexpected behaviour for developers who expect semantics similar to other imperative languages such as C# and Java.

Although monads have seen the most use in functional/declarative languages, they have found a use for structuring DSLs within imperative languages. Du Bois and Echevarria [7] extended the Java language with a DSL similar to Haskell's `do`-notation to add *Software Transnational Memory (STM)* as a feature in Java. This was implemented using a transpiler that mapped the `do`-notation statements into a chain of closures. Another example is the work of Silva Feitosa *et al.* [8], who extended Featherweight Java with monadic binds to facilitate a notation for structuring quantum computations. While the language itself is based on a scaled-down variation of Java, the interpreter and parser were developed from the ground up in Haskell. Both these works required a significant engineering effort to extend a language with the notation needed to accomplish their research goals.

6.3 Future work

We can see two main areas of future work: better implementations in more languages and a deeper dive into the performance of the bind syntax abstraction. Both of these areas aim to improve the framework's usability, improve the developer experience with better tools, and improve the user experience with more preferment programs.

This project has produced two working examples of monadic bind syntax in imperative

languages, but they are far from finished products. The Rust implementation is only a *Minimum Viable Product*, it works and can be used under the tested conditions, but it should not be used in production environments. The C# implementation is not on the same level; it could, at best, be considered a *Proof Of Concept*. In addition to polishing up these existing examples or entirely replacing them with better implementations, implementations in other languages are also a future possibility.

Additionally, we have shown that implementing our interface has a considerable performance cost compared to an equivalent reference program. Research is needed to properly determine the core factors behind the performance costs and how monadic abstractions can be more performantly implemented in imperative languages.

Bibliography

[1] B. Bringert *et al.*, "Student Paper: HaskellDB Improved," in *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '04, Snowbird, Utah, USA: Association for Computing Machinery, 2004, pp. 108–115, ISBN: 1581138504. DOI: 10.1145/1017472.1017473.

[2] P. Li and S. Zdancewic, "Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-Level Concurrency Primitives," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, San Diego, California, USA: Association for Computing Machinery, 2007, pp. 189–199, ISBN: 9781595936332. DOI: 10.1145/1250734.1250756.

[3] M. Spivey, "A functional theory of exceptions," *Science of Computer Programming*, vol. 14, no. 1, pp. 25–42, 1990, ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(90)90056-J. [Online]. Available: https://www.sciencedirect.com/science/article/pii/016764239090056J.

[4] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, pp. 55–92, 1991, Selections from 1989 IEEE Symposium on Logic in Computer Science, ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(91)90052-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540191900524.

[5] P. Wadler, "Comprehending Monads," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90, Nice, France: Association for Computing Machinery, 1990, pp. 61–78, ISBN: 089791368X. DOI: 10.1145/91556.91592.

[6] D. Syme, "The Early History of F#," *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, 2020. DOI: 10.1145/3386325.

[7] A. R. Du Bois and M. Echevarria, "A Domain Specific Language for Composable Memory Transactions in Java," in *Domain-Specific Languages: IFIP TC 2 Working Conference, DSL 2009 Oxford, UK, July 15-17, 2009 Proceedings*, Springer, 2009, pp. 170–186.

[8] S. da Silva Feitosa, J. K. Vizzotto, E. K. Piveta, and A. R. Du Bois, "A Monadic Semantics for Quantum Computing in Featherweight Java," in *Programming Languages*, F. Castor and Y. D. Liu, Eds., Cham: Springer International Publishing, 2016, pp. 31–45, ISBN: 978-3-319-45279-1.

[9] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: A Minimal Core Calculus for Java and GJ," *ACM Trans. Program. Lang. Syst.*, vol. 23,

no. 3, pp. 396–450, May 2001, ISSN: 0164-0925. DOI: 10.1145/503502.503505. [Online]. Available: https://doi.org/10.1145/503502.503505.

[10] D. Sabardie. "do-notation." (2021), [Online]. Available: https://github.com/phaazon/do-notation (visited on 2023-04-27).

[11] B. McNamara and Y. Smaragdakis, "Syntax sugar for FC++: Lambda, infix, monads, and more," in *Draft Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, 2003, p. 15.

[12] S. Mac Lane, *Categories for the Working Mathematician*. Springer-Verlag New York, Inc., 1998, ISBN: 978-0-387-98403-2. DOI: 10.1007/978-1-4757-4721-8.

[13] S. Klabnik, C. Nichols, and with contributions from the Rust Community, *The Rust Programming Language*. 2022. [Online]. Available: https://doc.rust-lang.org/stable/book/ (visited on 2023-02-28).

[14] J. Lloyd, "Practical advantages of declarative programming," English, Conference Proceedings/Title of Journal: Joint Conference on Declarative Programming, 1994, pp. 3–17.

[15] L. B. Wilson and R. G. Clark, *Comparative programming languages* (International computer science series). Addison-Wesley, 1993, ISBN: 0201568853.

[16] S. L. Peyton Jones and P. Wadler, "Imperative Functional Programming," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93, Charleston, South Carolina, USA: Association for Computing Machinery, 1993, pp. 71–84, ISBN: 0897915607. DOI: 10.1145/158511.158524.

[17] The Rust Release Team. "Announcing Rust 1.69.0 | Rust Blog." (Apr. 20, 2023), [Online]. Available: https://blog.rust-lang.org/2023/04/20/Rust-1.69.0.html (visited on 2023-05-18).

[18] Stack Overflow. "Stack Overflow Developer Survey 2022." (Jun. 22, 2022), [Online]. Available: https://survey.stackoverflow.co/2022/ (visited on 2023-03-02).

[19] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002, ISBN: 9780262256810.

[20] J. Huey. "Generic associated types to be stable in Rust 1.65 | Rust Blog." (2022), [Online]. Available: https://blog.rust-lang.org/2022/10/28/gats-stabilization.html (visited on 2023-03-01).

[21] E. Lippert. "Closing over the loop variable considered harmful, part one." (2009), [Online]. Available: https://ericlippert.com/2009/11/12/closing-over-the-loop-variable-considered-harmful-part-one/ (visited on 2023-05-02).

[22] E. Wallace. "esbuild." (2020), [Online]. Available: https://esbuild.github.io/ (visited on 2023-03-14).

[23] Babel Team. "Babel." (2015), [Online]. Available: https://babeljs.io/ (visited on 2023-03-14).

[24] P. Carter. "Introducing C# Source Generators." (Apr. 29, 2020), [Online]. Available: https://devblogs.microsoft.com/dotnet/introducing-c-source-generators/ (visited on 2023-03-14).

[25] D. Tolnay. "syn." (2023), [Online]. Available: https://docs.rs/syn/2.0.12/syn/index.html (visited on 2023-03-31).

[26] M. P. Jones, "Functional programming with overloading and higher-order polymorphism," in *Advanced Functional Programming*, J. Jeuring and E. Meijer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 97–136, ISBN: 978-3-540-49270-2.

[27] "catch_unwind in std::panic - Rust," The Rust Foundation. (2015), [Online]. Available: `https://doc.rust-lang.org/std/panic/fn.catch_unwind.html` (visited on 2023-05-18).

[28] "Returns and Jumps," Kotlin. (May 11, 2023), [Online]. Available: `https://kotlinlang.org/docs/returns.html` (visited on 2023-05-12).

[29] S. L. Peyton Jones, *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. USA: Prentice-Hall, Inc., 1987, ISBN: 013453333X.

[30] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A History of Haskell: Being Lazy with Class," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III, San Diego, California: Association for Computing Machinery, 2007, 12–1–12–55, ISBN: 9781595937667. DOI: `10.1145/1238844.1238856`.

A

Option Benchmark Source Code

Reference implementation

```
fn non_monadic(v: u8) -> Option<u8> {
  let a = is_even(v)?;
  let b = is_even(v / 2)?;
  let c = is_even(v / 4)?;
  let result = is_even(a + b + c)?;
  Some(result)
}
```

With monadic abstraction

```
#[monadic]
fn monadic(v: u8) -> Option<u8> {
  let a = is_even(v)?;
  let b = is_even(v / 2)?;
  let c = is_even(v / 4)?;
  let result = is_even(a + b + c)?;
  Option::ret(result)
}
```

B

Writer Benchmark Source Code

Reference implementation

```rust
fn non_monadic(
  v: usize,
  l: &mut Logger,
) -> usize {
  l.log("STARTING REACTION");
  let reaction_level = (v + 43) * 2;
  l.log(&format!(
    "REACTION LEVEL: {reaction_level}",
  ));
  l.log("STOPPING REACTION");
  reaction_level
}
```

With monadic abstraction

```rust
#[monadic]
fn monadic(v: usize) -> Logging<usize> {
  Logging::log("STARTING REACTION")?;
  let reaction_level = (v + 43) * 2;
  Logging::log(&format!(
    "REACTION LEVEL: {reaction_level}",
  ))?;
  Logging::log("STOPPING REACTION")?;
  Logging::ret(reaction_level)
}
```

C

State Benchmark Source Code

Reference implementation

```
fn non_monadic(
  s: GameState,
) -> GameState {
  let s = s.inc_x(10).inc_y(10);
  let x = s.get_x();
  let y = s.get_y();
  s.set_score((x + y) as f32)
}
```

With monadic abstraction

```
fn monad_runner(
  s: GameState,
) -> GameState {
  monadic().run(s).0
}

#[monadic]
fn monadic(
) -> State<'static, GameState, ()> {
  inc_x(10)?;
  inc_y(10)?;
  let x = get_x()?;
  let y = get_y()?;
  set_score((x + y) as f32)
}
```