A Game of Paradigms

A Usability Study of Functional Idioms in Gameplay Programming

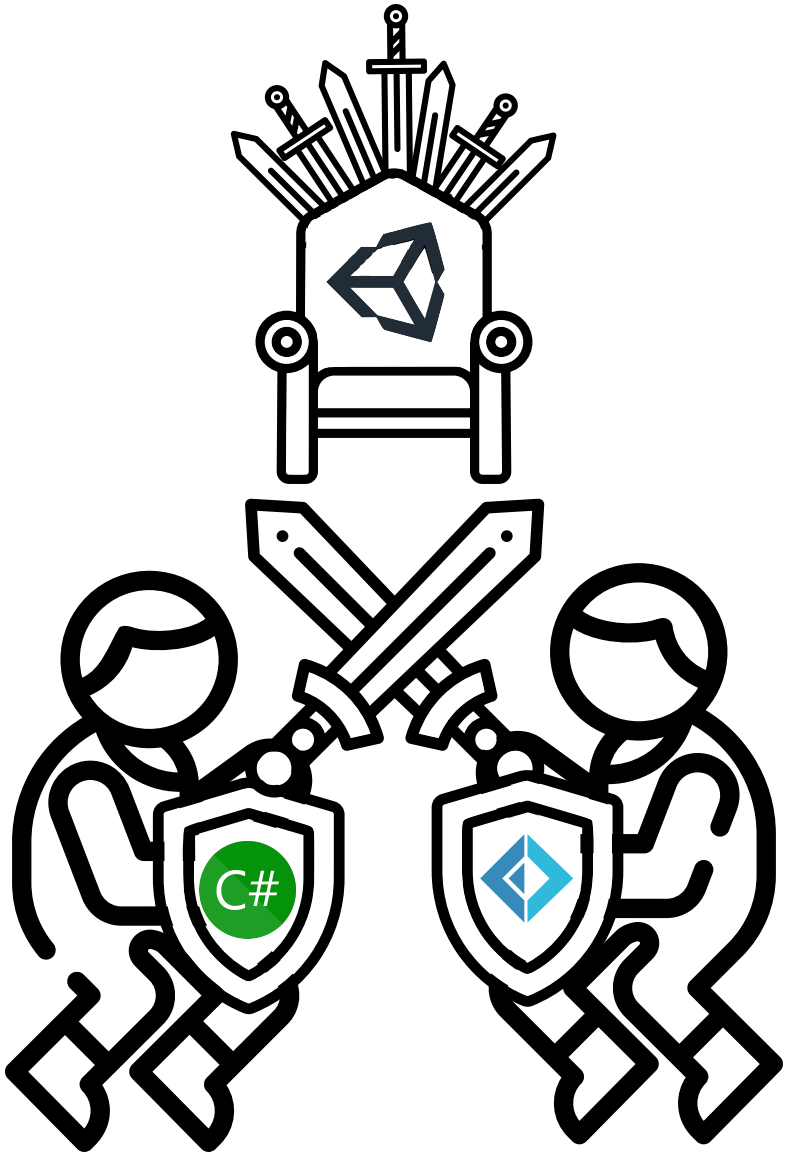


Figure 1: Made using icons by Freepik and Nikita Golubev from www.flaticon.com

Project Report - 10th Semester Computer Science

PT103F19

Aalborg University
Department of Computer Science, SICT

AALBORG UNIVERSITY

STUDENT REPORT

Software Engineering
Aalborg University
http://www.cs.aau.dk/

Title:
A Game of Paradigms: A Usability Study of Functional Idioms in Gameplay Programming

Theme:
Scientific Theme: Language evaluation, game development, functional programming, interpreted languages, evaluation strategey

Project Period:
Spring Semester 2019

Project Group:
PT103F19

Participant(s):
Thomas Gwynfryn McCollin
Tobias Morell

Supervisor(s):
Bent Thomsen

Copies: Digital distribution only

Page Numbers: 131

Date of Completion:
June 6, 2019

Abstract:

This project examines the use of functional programming in gameplay programming. Two notable game development gurus, John Carmack and Tim Sweeney, claim that increased use of functional programming in game development would be beneficial. This project puts those claims to the test by comparing the use of C# and F# in the Unity game engine.
We first examine experienced game developers attitude towards the claims via a usability evaluation. We found that the participants were able to write more concise and modular code in F#, but still were reluctant to use it in practise. In need of a stronger incentive we turned to a performance study, intended to measure if concurrent code in F# is more performant than concurrent code in C#. We found that F# is slightly slower than C# in most cases. Finally we put the observed benefits of F# into the context of modern game development practices to examine why those benefits are not appealing to experienced gameplay programmers.

Contents

Preface

The field of game development is subject to many opinions from the industry and is often backed by little scientific evidence. We will therefore need to cite pages that are not "traditionally scientific" in this report, examples of which are Quora and StackOverflow. These sources are of questionable quality and need not reflect the broad opinion on the game development industry. However, we include them here to indicate that "some" game developers share the expressed belief.

Acknowledgements

Bent Thomsen

Adam Kjær Søgaard

Hans-Christian Greve

PT101F19 and PT102F19 for the good mood in the group room

The participants of the usability evaluation

Resume

This project examines the claims of John Carmack and Tim Sweeney. These two notable game development figures claims than increased used of functional programming in game development would be beneficial. The two gurus suggest using functional programming in different ways; Carmack argues that developers should adhere to a functional style in any language, whereas Sweeney suggests the introduction of a new pure functional language with explicit effects typing and lenient evaluation.

We decided to examine why functional programming is not yet used by AAA game engines, such as Unity and Unreal Engine. This was first examined by conducting a usability evaluation. The participants in the evaluation were profession game developers from Aalborg, who had experience in Unity and C#. During the test sessions the participants would implement one or more task, which was designed

to resemble an aspect of gameplay programming. Half of the tasks were to be implemented in F# and the rest in C#. A total of eight tasks were designed for the experiment, which were sorted into four categories. The purpose of the categories was to allow a side-by-side comparison of F# and C# under similar conditions. After the session a short interview was conducted to allow the participants to express whether or not they found the use of functional programming beneficial in game development. The participants generally agreed that the use of functional programming would prove many benefits, such as more modular code and immutability, but still were reluctant to use it in practice. Their primary arguments against F# was that the cost of learning the language would be too high, compared to the provided benefits.

After learning that game developers were not keen on switching, we decided to examine the performance impact of using F# instead of C#. We did so with a particular focus on concurrency, to learn whether concurrent code F# was more performant than in C#. In most cases, F# runs slightly slower than C#. Furthermore, we found that F#'s Async Workflows parallelisation strategy was fragile, as using `Async.Parallel` instead of `Async.StartChild` can result in a massive performance degradation. In the best case circumstances, the `Task` parallelisation of C# and Async Workflows of F# seems to fare equally well. We also conducted a test in Unity, which measures the framerate given an increasing number of `MonoBehaviours`. We found that `MonoBehaviours` written in F# was a little less performant compared to C#. Finally, the performance of the Functional Reactive Programming (FRP) system, which was written as part of this project, was much slower still. The reason for this was that the current implementation is simple and suboptimal, meaning that each `FRPBehaviour` is a full-blown FRP system with conditions-checking and event-dispatching.

Finally, we put out findings into the perspective of modern game development and discuss why the benefits provided by F# might be less important to the developers than productivity.

1 | Introduction

Game development has been dominated by the object oriented programming paradigm since the advent of game engines[1], arguably even before that. Game engines have separated hardware and rendering specific problems from the game development and thus enabled the emergence of smaller independent game developers[2]. These developers have varying levels of training in modern software disciplines[3], [4] which exacerbates the complexities of game development[5].

While C/C++ still seems to dominate the game development industry[6], some of the widely used commercial game engines have migrated from C/C++ to other alternatives. One example is Unreal Engine, which has released the visual programming language Blueprint[7] to support developers and reduce game development complexity[8]. While these languages are promising, this niche may also be filled by functional languages. Proponents of functional languages have, in the past, claimed that functional languages excel under similar conditions[9]–[11]. Therefore such languages are examined in a game development context.

In a previous semester project, we have investigated the performance impact of managed languages in a game development context[12]. Some game development techniques utilising functional programming were also measured, but these systems were not competitive. Instead we examine the functional programming approaches proposed by influential game developers: John Carmack (founder of id Software) and Tim Sweeney (founder of Epic Games). Their proposed approaches to functional programming, while striving for the same goal, tackle the issue differently.

> *"No matter what language you work in, programming in a functional style provides benefits. You should do it whenever it is convenient, and you should think hard about the decision when it isn't convenient."*
>
> -John Carmack[13]

John Carmack suggests placing the responsibility on the programmer and underlines that the functional programming style should be adhered to whenever possible[13]. This approach is well supported in the multiparadigm programming language C#, which is predominantly object oriented, but supports various functional constructs, such as lambda expressions[14] and soon pattern matching[15]. Another important aspect of this approach is purity. Game developers argue that games are inherently stateful[5], [16], but smaller functions may be side-effect free

and effectively become pure. Carmack, however, believes that enforcing this is left to the programmer.

> *"Purely Functional is the right default [...]*
> *Imperative constructs are vital features that must be exposed through explicit effects-typing constructs [...]*
> *Lenient evaluation is the right default."*
>
> -Tim Sweeney[17]

The approach proposed by Tim Sweeney suggests a new game-development oriented and functional-programming language[17]. Thus responsibility is moved away from the programmer. The suggested language should be pure, but support imperative constructs. Another quirk of the language is the proposed evaluation strategy. Sweeney believes that the language should use the lenient evaluation strategy and use strict/eager evaluation as a compiler optimisation, furthermore lazy evaluation may be made available via explicit constructs to the programmer.

1.1 Problem Statement

The gurus John Carmack and Tim Sweeney present two different approaches to the use of functional programming in game development. On one hand Carmack recommends writing code in a functional style whenever possible, but he does not believe fully functional programming languages are practical tools for game development. The main features mentioned are higher-order and pure functions along with action items.

In contrast Sweeney argues for a pure functional language supporting multiple evaluation strategies explicitly controlled by the programmer. In addition, he argues for constructs that allow for explicit handling of the imperative nature of games, however the exact nature of such constructs is left to the language designers.

Using the .NET platform we can conduct a side-by-side evaluation of functional and object-oriented programming. C# is object-oriented and has many functional constructs, which presents a viable candidate of Carmack's suggested approach. F# is a functional programming language and presents a possible candidate for Sweeney's approach. With two programming languages of these paradigms, we can test their applicability to game development. We propose the following research questions to examine the use of functional programming in game development:

1. **How well do experienced game developers express gameplay code in the functional paradigm, in comparison to object-oriented programming?**

2. How can functional programming be incorporated in game development?
3. What are the performance impacts of using functional idioms in a game engine?
4. What is required of functional programming to be adopted by game developers?

1.2 Project Scope

As this project examines the use of functional programming in game development, we have to select a game framework or game engine that will act as a host for the experiment. We have suggested using C# and F# in the experiment because they run in the same platform and are representative of the gurus' suggestions. There are a couple of game engines that support the .NET platform, including Unity, Godot, MonoGame and CryEngine, among others[12]. We chose Unity in this project, because it is the most popular engine that supports the .NET platform[18]. None of the platforms support F#, but the communities have added support in all cases[19]–[22].

2 | Related Work

In this chapter we discuss related work. We first examine how performance of parallel programs may be analysed. Tim Sweeney mentions effects typing constructs in his approach, which we discuss next. Finally we examine other projects that have tried to incorporate functional programming in game engines and classify performance benchmarking strategies.

2.1 Implicit Parallelisation

Functional programming has long claimed to be inherently parallelisable[23], [24]. The claimed advantage of functional programs is the high degree of modularity, which simplifies the parallelisation process. A number of languages have attempted to implement programming systems that utilise this advantage, however few have reached mainstream use. In this section we will explore the theoretical background for implementing an implicitly parallelisable system in F#.

First different evaluation strategies are explored, since F# supports both strict and non-strict evaluation. In addition to these two dominant evaluation strategies, another promising strategy, called lenient evaluation, is examined. Once the strategies are outlined, the question of when to parallelise is addressed with the work/span methodology.

2.1.1 Evaluation Strategies

Programming language behaviour is heavily dependent on the evaluation strategy employed. Conventionally two such strategies are prominent in the literature[25]; *strict* (also called eager) and *non-strict* (commonly known as lazy, though this is not entirely accurate). [25] suggests a *lenient* evaluated language. This strategy is non-strict, but not lazy, which places it in-between eager and lazy[25]. This intermediary position means that the lenient strategy benefits from both eager and lazy advantages. According to literature lenient evaluation lends itself to implicit parallelism[26]. This section will explore the three strategies and clarify the distinctions between them.

In Sweeney's discussion of the game-programming language of tomorrow, he un-

derlines that it should make use of the lenient evaluation strategy[17]. The reason for this is that eager evaluation strategy is too limiting and lazy evaluation is too costly. Sweeney also underlines that the compiler of this new language should be capable of optimising pieces of code to use eager evaluation, whenever it is more optimal.

Strict Evaluation

Strict evaluation, often called eager evaluation, is a prominent evaluation strategy in traditional programming languages. The defining feature of the strategy lies in the handling of function parameters. Eager evaluation requires fully evaluated parameters before a function may be evaluated[27, p. 103], however this is not the only requirement. Depending on the choice of parameter passing approach, the evaluation strategy may also be affected. Chief among eager parameter passing approaches are call-by-value and call-by-reference. In the case of call-by-value a parameter must be evaluated and its result passed to the function. On the other hand call-by-reference has the same requirement, but instead the memory address of the result is passed to the function.

Non-strict Evaluation

Contrary to strict evaluation, non-strict evaluation does not require parameter evaluation until they are needed[28]. This strategy is often implemented using call-by-name or call-by-need parameter-passing approaches. This strategy affords the programmers more expressive power[29] and allows for infinite data structures and non-terminating functions[27, p. 103]. The parameter passing approaches state that parameter evaluation is delayed until they are called, therefore unused parameters need not be calculated at all.

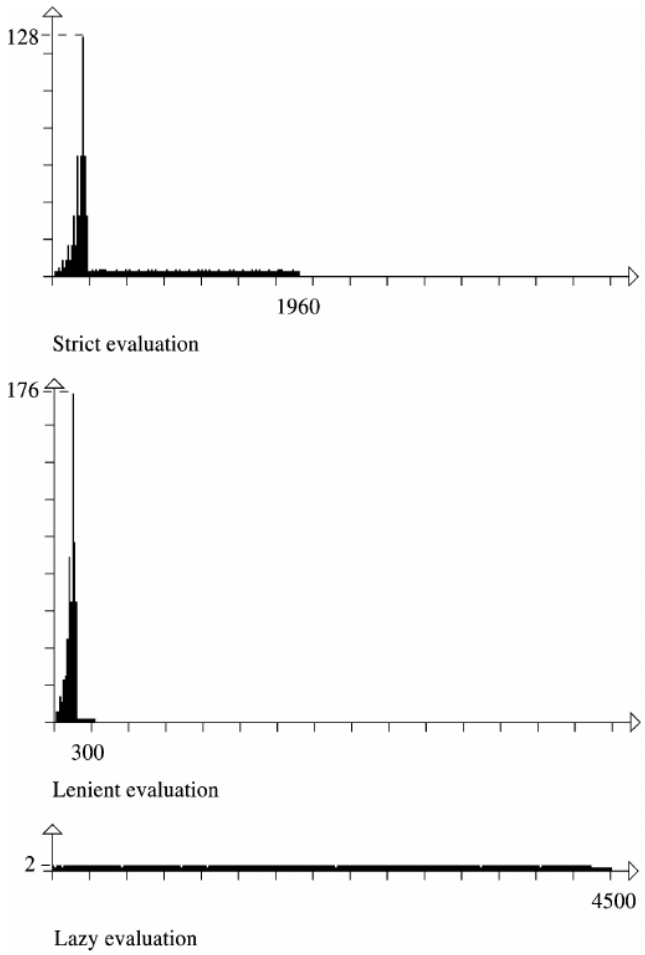
Lenient Evaluation

The lenient evaluation strategy does not restrict the order of parameter evaluation. The only requirement is that the variables are available when they are needed, or in other words that the data dependencies are satisfied. Parameter passing approaches in this strategy often make extensive use of parallelism and concurrency techniques, due to the inherent parallelisability of lenient evaluation[26].

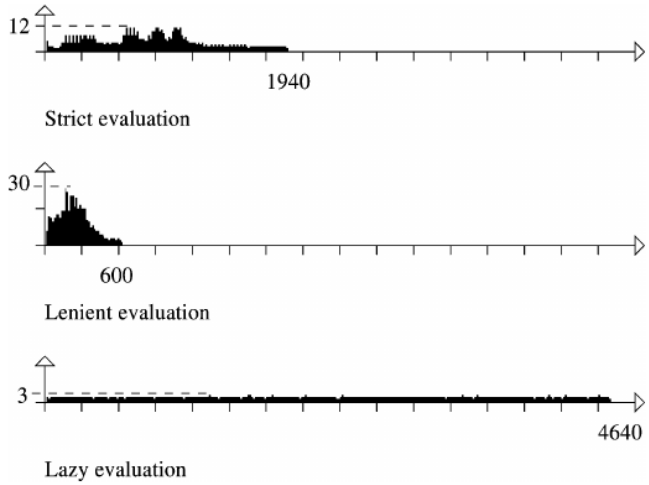
The most notable instantiation of this evaluation strategy is call-by-future[30]. In this implementation the main thread of the program executes the program and every time it encounters a function invocation, it spawns a future for each argument to the function. The main thread then continues to execute the function-body, synchronising with the futures as arguments are needed in the function-body.

This has the result that functions and arguments are computed concurrently. The parallelisability of programs may vary greatly, depending on how the program is written and what it is intended to do. Therefore, we examine static analysis methods to determine when to parallelise.

A visualisation of parallelism in the different evaluation strategies can be seen in Figure 2. Here two cases are compared: a summation of a binary tree (Figure 2a) and sorting and summation of a list (Figure 2b). The X-axis represents time and the Y-axis represents the number items that can be computed in parallel at that given point in time.



(a) Parallelism profiles for computing the sum of the leaves of a binary tree.



(b) Parallelism profiles for sorting a list and computing its sum.

Figure 2: Parallelism profiles for evaluation strategies, graphs taken from [26].

2.1.2 Formal Performance of Parallel Programs

This section will detail a formal method of estimating performance of parallel programs called work and span. This method counts the number of primitive

operations required to execute the entire program. This is called the sequential running time of the program and is denoted: $T(n)$ where n is the problem size[31]. The sped up running time, using additional processors, is denoted: $T_p(n)$. Here p denotes the number of processors, Thus:

$T(n)$ denotes the total sequential running time.

$T_p(n)$ denotes the total running time of the program when parallelised as much as possible.

In order to estimate $T_p(n)$ the *work* and *span* of the program must be identified. The work is the total running time of all processors, ignoring synchronisation overheads. This is equivalent to running the program on a single processor or sequentially. Therefore

$$\text{work} = T(n)$$

The span is the longest data dependent path in the program i.e. the longest path of strictly sequential computation. This is sometimes called the critical path or the computational depth[32]. The shorter the span, the more parallelisable the program. Finally the cost of the program can be calculated. This is the total running time across all processors including the time spent idling. The cost is denoted pT_p.

Given this information about a parallel program, the speed-up gain from parallelisation can be calculated. This calculation assumes an infinite number of processors, T_∞. A number of different metrics for this gain exist, they are as follows.

Speed-up is the raw gain from running the program on multiple processors.

$$S_p = T_1/T_p$$

Efficiency is the speed-up per processor.

$$S_p/p$$

Parallelism is the maximum possible speedup given a number of processors.

$$T_1/T_\infty$$

Slackness is a measure of the program's parallelisability. A slackness of less then one implies that perfect linear speedup is possible.

$$T_1/(pT_\infty)$$

Since no actual machine has an infinite number of processors, the above equations require slight modifications to simulate real machines. Any computation that can run on N processors can be executed on smaller number of processors, $p < N$[33]. This is achieved by dividing the work load onto the processors, instead of assigning one processor per task. Furthermore, running on fewer then N processors the execution is bounded by:

$$T_p \leq T_N + \frac{T_1 - T_n}{p}$$

The bound T_p can be expressed with upper and lower bound[34]:

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

These metrics are calculable at compile time and therefore present a potential answer to the implicit parallelisation question.

2.1.3 Lenient Parallelisation

Using a highly granular parallelisation system, such as .NET `Tasks`, each argument to a function could be evaluated as a task, unless the expression is too small. Determining the computational cost of an argument is calculated by estimating the work of the expression. This information, along with the span of the program, should provide enough information to the system to effectively parallelise a lenient program.

2.2 Effects Typing

In the problem statement we presented a quote from Tim Sweeney suggesting that "*imperative constructs should be made available via explicit effects typing*" (see Chapter 1). Effects typing i.e. type and effect systems, are type systems that track types and the changes made to them[35]. These changes are referred to as effects. Such changes could be modifying a mutable variable, opening a file handle, reading or writing to a shared resource. These systems are mainly seen in academia, where annotated type and effect systems have been implemented. The annotated approach has proven to be unwieldy to program[36], which may explain the small number of languages using such systems. Recently type and effect systems have been subject to renewed interest and the Rust programming language incorporates a similar system[37].

The effects tracked by type and effect systems can be extended to parallel changes. Thus the system can be used to manage concurrency and guarantee equivalent

behaviour between sequential and concurrent implementations of the same problem[38]. While type and effects systems have been around for a a while, it has recently been proven to be able to calculate when it is sound to parallelise a task[39]. This information is discernible via static analysis of the source code, using an annotated type and effect system, at compile time.

2.3 Functional Programming in Games

In this section we examine other projects that research functional programming in game development. These projects range from scientific articles and reports to open-source projects.

2.3.1 Functional Programming in Unity

In parallel to this project, another group on the Programming Technology specialisation course is researching the use of F# in Unity as well. This project researches how well children, who are members of the Coding Pirates group, and students on the Medialogy education are able to express gameplay code in F#. In their experiment, the participants are to complete eight tasks, which results in a space invaders game. After solving each task the participant must self-evaluate how well the exercise was completed along with how much effort it took to solve it[40].

There are several projects that aim to integrate functional programming in Unity. Examples of those are Unity F# Integration[19], F# Kit[41] and Arcadia[42]. The two former allow the programmer to write gameplay code in F#, whereas the latter implements Clojure in Unity. We found that said game frameworks needed a larger community and better documentation in order to be truly useful. Furthermore Arcadia introduced a significant performance impact, compared to C# in Unity.

2.3.2 Reactive Programming in Unity

Just as there are projects that seek to integrate functional programming in Unity, there are also projects that seek to integrate reactive programming. An example of such is UniRx[43]. This project implements a series of extensions that allow the programmer to use reactive programming in Unity. The advantages of this are that keyboard, mouse and other types of input can be treated as event streams and filtered in. Furthermore, they underline the simplicity of implementing parallel and asynchronous web requests[44].

2.3.3 Functional Reactive Programming in Games

The scientific community has also shown interest in functional programming in games. Particularly the paradigm FRP has seen a lot of research. This idea of FRP originates from FRAN[45] and was later used to implement the game framework Yampa Arcade[46]. [47] implements a First Person Shooter (FPS) game called FRAG in Yampa Arcade and Haskell and concludes that it requires fewer lines of code to implement concurrent updates of game objects in Yampa compared to multithreading. In previous research we have also examined the use of functional programming in game development[12], in particular in the game engine Nu, the game framework Helm and in the Arcadia extension for Unity. These projects aim to enable game development in respectively F#, Haskell and Clojure. We found that said projects were hard to use, mainly because of lacking documentation and a small community around them. Furthermore, the Arcadia extension for Unity had a huge impact on the performance of the gameplay code[12].

2.4 Benchmarks

Estimating the general performance of a programming language is a difficult task. A programming language, even a Domain Specific Language (DSL), has broad problem domains compared to conventional programs. Therefore benchmarks suites are employed to test different aspects of the programming language. Such suites have been used to measure the Scheme programming language and its derivatives[48], [49] and popular game engines[12]. When measuring the performance of managed languages, microbenchmarks are often employed to account for warm-up time and garbage collection[50].

2.4.1 Benchmark Categories

There exists a number of different benchmarking techniques. This section will outline a few of these and clarify their differences. The different kinds of benchmarking are outlined in [12]:

- Microbenchmarks
- Macrobenchmarks
- Application Benchmarks

Microbenchmarks are small tests that can be run repeatedly. They test a small part of the code and are sometimes referred to as component tests. These tests are

comparable to unit tests in size. In the microbenchmarking methodology outlined in [50], these small tests are run several times to measure the arithmetic mean and standard deviation. The number of runs varies, the idea is that the code has run at least twice and at least for 0.25 wall-clock seconds.

Macrobenchmarks are considered tests of multiple microbenchmarks sequenced together. These are comparable to an automatic integration test or a test of a partial system. These tests measure the overhead of component composition. This methodology is still based on the same principles as microbenchmarking and therefore computes an arithmetic mean and standard deviation for the composed system.

Application benchmarks test a full application, with start-up times included. This is sometimes referred to as real program testing. This testing category measures a real use case and therefore doesn't always employ arithmetic mean and deviation.

3 | Research

In this chapter we present the research that was conducted as foundation for the project. We first discuss FRP, which is presented as a suitable candidate for integrating functional programming in games[46], [47], [51]. We then discuss how the usability of programming languages can be ascertained. Finally we examine Unity's concurrency system and discuss how that differs from .NET's `Task` model.

3.1 Functional Reactive Programming

FRP stems from the Functional Reactive Animation (FRAN) framework presented in [45]. In the scientific community the most notable FRP framework is the Yampa Arcade project[46], which was initially used to implement a clone of Space Invaders. It was later shown that Yampa Arcade could be used for commercial-grade games of that time by developing a game called FRAG[47].

FRP is a mixture of functional programming and reactive programming, treating programs as data- or event streams (events are sometimes called signals). [52] describes events as time-stamped values, i.e. discrete variables with respect to time. Examples of events are button clicks, GPS location updates and gestural inputs[53]. As an example, a mouse button may be either clicked or not clicked at some time t. The event stream from the mouse button is a list of tuples: $(time, clicked)$. Programs are expressed in a declarative manner as sets of event handlers (sometimes called signal functions or behaviours) that react to the event streams. The strength of FRP lies in the ability to combine multiple event handlers. Such combinations may come in the form of chaining together several event handlers or creating one event handler that responds to multiple events[52].

[51] presents an overview of different game-related FRP systems and conclude:

> *"Perhaps a commercial language like F#, with better support and integration into the presentation pipeline (.NET, C#, XNA, and the Kinect SDK), would be a better choice for making FP (functional programming) into a real game changer."*
>
> -Christopher Maraffi and David Seagal[51]

Apart from Yampa Arcade, FRP has also been implemented in other game frame-

works, such as Helm [54] and more recently Nu[55]. In previous work we examined both and concluded that to be truly useful they need a bigger community and more documentation[12].

3.1.1 FRP Performance

In scientific literature, the consensus seems to be that FRP is too slow to be adopted in game development[47], [51], which is backed by findings from Nu, where 7,000 objects can be simulated using pure FRP, whereas the number is 25,000 for imperative objects[56]. On the other hand, [57] claims that Netflix, a world-wide video streaming service, uses FRP on both frontend and backend. If that is true, it is a strong indication that the performance overhead is not as large as the scientific community fears.

3.1.2 Other Applications

FRP is also used in other areas than game development. As mentioned earlier, FRP dates back to FRAN, which could be used to model 3D geometry[45]. FRP has also seen its use in music with the Euterpea Haskell DSL[58].

3.2 Usability Evaluation of Programming Languages

In this section we present usability evaluation methods for programming languages. As usability evaluation of programming languages does not have a standardised method or framework, we discuss several different usability evaluation models. These include user interface evaluation methods, such as Instant Data Analysis (IDA), and methods that are tailored towards programming language analysis such as discount method for language evaluation. We wish to research a broad palette of methods, as this allows us to combine the strategies in an attempt to obtain valuable data.

3.2.1 Instant Data Analysis

IDA is an analysis strategy, which is meant to be used with think-aloud based usability evaluation techniques[59]. [59] suggests conducting between four and six usability evaluation sessions, which are followed by a one hour IDA brainstorming session. During the brainstorming session the data-logger and test monitor discuss

the usability problems found during the sessions. Meanwhile, the facilitator takes notes, categorises the problems, asks questions for clarification and directs the discussion. After the brainstorming session, the facilitator writes down a prioritised list of usability problems, which ranks their severity, placement in the software system and gives a short description of the problem. The IDA method has the advantage that a full-scale usability evaluation can be conducted in a single day, while still discovering a majority of the problems[59].

3.2.2 Discount Method for Language Evaluation

Discount Method for Language Evaluation is a work-in-progress method based on IDA[60] and the Discount Usability Evaluation Method[61]. It is a technique intended for low-cost evaluation of a programming language during its development. The technique requires very limited setup and can be used even before the language's compiler has been written. The method requires the test participant to implement a set of programming problems in the language. The test participants are equipped with a sample sheet, that gives code examples and brief explanations of how the language is structured. All the programs are written either in a text-editor or using pen and paper. These tools present no error-checking and code-completion. This leaves all errors present in the code for subsequent evaluation. The errors are then classified according to how much work it would require to fix them.

3.2.3 Cognitive Dimensions

In order to talk about notation systems and their usability features in general, these features must be generalised to their fundamental dimensions. These dimensions are the cognitive dimensions, which dictate the usefulness of design strategies for various design problems. One such vocabulary is the cognitive dimensions framework[62]. The framework itself consists of a set of thirteen dimensions, each of which represent a generic usability problem area. We will not go in detail with the dimensions as more detailed descriptions can be found in [62] and [12]. In previous work we used the cognitive dimensions framework to compare gameplay programming in C++ and C#[12].

3.2.4 Attention Investment Models

In order to understand how users interact with a programming system, their problem solving approach needs to be examined. This can be done by analysing the generic nature of a user's first programming steps. From this analysis the cognitive demands can be explored and mapped[63].

The cognitive demands consists of *cost*, *investment*, *payoff* and *risk*:

Cost is the number of attention "units" required by a programming activity to be completed.

Investment is the actual attention spent by the user and may be greater than the cost.

Payoff is the reduced cost of undertaking the task in future, due to the assistance of the program.

Risk is the chance that the program is not completed and the investment wasted.

The model uses an agent architecture. This means that each course of action is represented as an agent competing for the user's attention. Human focus is simulated by only allowing one agent to be processed at a time. In fact all tasks in the system are represented as agents, including subtasks and the division of tasks into subtasks. Thus inquiry into the problem and the problem solving activities can both be modelled using the same system.

The model can be used to give a broad stroke estimation of a user's attention expenditure when using a system. This can help designers improve the system by identifying problematic features. The model can also be applied with more rigour, to achieve a finer grain understanding of the attention economy of a system. This approach entails simulating the behaviour of the user in the agent architecture yielding even more information, but is significantly more costly.

The Attention Investment Model serves as a cognitive model of programming efforts that offers a consistent account of all programming behaviour, from professionals to end-users.

3.2.5 Champagne Prototyping

The Champagne Prototyping method was developed for the testing of Microsoft Excel[64]. The reason for its development was a lack of cheap prototyping techniques that could be deployed early in the project. The methodology is designed to answer a question about a feature or product. A cheap prototype is created specifically to answer this question. It is important to keep in mind that the prototype must be complete enough that it can be used to answer the question. The authors found the following points to be the minimum necessities:

- The prototype must be fully operational. Useful usability data can only be gathered from a working system.
- Therefore the prototype must be based on an existing product.

- Dysfunctional prototypes can be used for demonstration purposes and can be used to gauge the users reaction to the visuals, but cannot be used if the user is to interact with it.

A small number of credible participants are selected. It is important that these participants are experts in the relevant field and that they have no programming experience (unless it is relevant for the field). The participants are asked to perform some task in the prototype. The nature of the task should be dependent on the question being answered. Following task completion, the participants are subjected to a scenario-based interview. Finally the results are analysed using attention investment models and cognitive dimensions.

3.2.6 Expert Review Method

Another approach to usability evaluation of programming languages is the expert review method[65]. In this method there are two different roles; the test participant and the expert. The test participant must be an experienced programmer, who has no experience in the language under test. The expert must be highly skilled in the language under test. In [65] the expert is a prominent member of the language's compiler team.

In the expert review method a test participant solves a series of problems in each of the languages under test. In [65] the authors use the Cowichan problems[66]. The method consists of four phases for each language:

1. The test participant implements solutions to each of the problems in the language under test.
2. The expert reviews the programs and writes a list of comments and suggestions for improvements.
3. The test participant incorporates the feedback from the expert.
4. The expert reviews the solutions again to check that the feedback was not misunderstood.

The feedback from phases two and four presents valuable information on possible pitfalls in the language and places for improvement.

In [65] the authors evaluate the programs using four metrics: code size, execution time, speed up and correction time. In other research by the same authors, they evaluate in greater depth the performance of the programs before and after the expert review phases[67].

3.3 Concurrency in Unity

Initially Unity was single-threaded, but over the last two years, Unity Technologies has made an effort to implement concurrency in the form of a C# Job System[68] and more recently an Entity Component System (ECS)[69]. Unity does not support the `async/await`-model, that is usually seen on the .NET platform, when dealing with `MonoBehaviours`[70], [71].

3.3.1 C# Job System

Unity's C# Job System provides a *"simple and safe"* way of writing multithreaded code in Unity. It can be used on top of the "traditional" way of writing Unity code, i.e. `MonoBehaviours`. In the job system, the developer expresses concurrent code using jobs rather than threads. Unity is in charge of running the jobs on a group of worker threads, which is shared with the engine code[68].

Jobs in Unity must be implemented as `structs` that implement one of the `IJob`-interfaces. This interface defines an `Execute`-method, in which the concurrent code must be written. Jobs can be scheduled by calling one of the `Schedule`-methods, which returns a `JobHandle`, that can be used to manage dependencies between jobs. The developer is in charge of figuring out the dependencies between the jobs, as Unity does not provide any means of dependency management[68].

Listing 29 in Appendix A shows an example that moves forward all bullets in a game using an `IParallelForTransform`-job.

3.3.2 Entity Component Systems

ECS is a design-pattern, which presents an alternative way of representing objects in a game world. It is an alternative to the scene graph pattern[72], which is used in both Unity and Unreal today. ECSs consist of three different components[69], [73]:

Entities represent single objects in the game. It is a very simple data structure, e.g. an ID, which is used to look up components associated with the object.

Components are data containers, which defines data that is needed in order to carry out a certain behaviour. It is very important that each component is kept small and defines as few fields as possible.

Systems define the behaviour of the entities. They consist of two parts; a filter, which defines the components that must be present for the system to take

effect and an update method, which applies the system's behaviour to the entities at regular intervals.

The advantage of ECSs, compared to scene graphs, are that they tend to increase code reusability, as systems may be used to control multiple different types of objects. Furthermore, ECSs will attempt to group components together in so-called chunks, which increases spatial locality. In Unity, the ECS may be used in conjunction with the C# Job System to update chunks in parallel[69]. Listing 31 in Appendix A lists an example, which moves all bullets in the scene forward.

4 | Usability Evaluation

In this chapter we research usability of functional-style programming in gameplay programming. We first use the cognitive dimensions framework to discuss our experience with the transition from C# to F# and compare the two languages.

Afterwards we conduct a usability test of F# in Unity. The tests are formulated based on the Champagne Prototyping methodology (see Section 3.2.5). For this test the Unity game engine was extended, via a plugin to support programming in F# [19]. In addition, a FRP module was implemented in F# for Unity. These two extensions served as the prototype under test.

4.1 Cognitive Dimensions of F# and C#

In order to ascertain the usability of F# in comparison to C#, we have conducted an analysis based on the cognitive dimensions framework[62]. C# is the gameplay programming language available in the Unity game engine, therefore, if F# is to usurp this position the advantages and disadvantages should be made clear. We conduct this analysis on the basis of our own experience with F#. Both authors of this report have prior experience with C# programming in Unity and no prior experience with F#.

Abstract Gradient

The abstract gradient is measured from abstraction hating, through abstraction tolerant, to abstraction loving. The abstractions measured are the notations' ability to group elements and refer to them as a single entity. Most modern textual-programming languages make extensive use of abstraction and functional languages even more so[28]. F# is a functional-programming language with object-oriented features allowing for extensive abstractions.

On the other hand C# is an object-oriented language which supports functional features. This means that C# also supports extensive abstraction. The main difference lies in the fact that C# is object-oriented programming first and F# is functional programming first. Considering the high level of abstraction in both languages they will both be considered abstraction loving in this report. [74] presents

a discussion, underlining that functional languages generally rely on function abstraction, treating types as thin data containers, whereas object-oriented languages rely on data abstraction, where functionality is associated with the types. The author notes that this may surface as making it harder to add new types in functional languages and adding functionality in object-oriented languages. A similar orthogonality is also expressed in two out of three of Tennent's language design principles[75], namely the principle of abstraction and the principle of data type completeness.

Closeness of Mapping

The measure of how close to the problem domain a language can get is called the closeness of mapping. In order to solve a real problem, the problem must be expressible in the language and the closer the language is to the real world, the easier it is to express[62]. Textual programming languages are abstractions over the real problem domain and therefore often do not map directly to the domain.

Both languages have mechanisms to model the problem domain. In object-oriented programming, the world is represented as objects and the objects are abstracted over via classes[76]. Functional programming models the problem as behaviour (functions) which are applied to data[10]. The advantage of the object-oriented approach is that the object abstraction comes quite close the problem domain.

> *"The object expresses the user's view of reality [...]"*
>
> -Object Oriented Analysis & Design[77]

This approach is in contrast to the functional paradigm which models reality mathematically. This approach is not as close as the object model, however mathematical modelling of the world is widespread in many different fields of study.

> *"Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives. These functions are much like ordinary mathematical functions [...]"*
>
> -John Hughes[10]

The abstraction models of the languages are the tools used by the programmers to model the world. C# uses a model, which lends itself more to closeness of mapping, but both languages make use of custom types and naming which allow programmers to mold their programs in accordance with the problem domain. Additionally both languages support each other's modelling approach. An example

of this can be seen in Listing 1, where a recursive class in C# can be implemented via custom types in F#.

```
type Talent(strength, intellect, agility) =
  member val Strength = strength with get, set
  member val Agility = agility with get, set
  member val Intellect = intellect with get, set

type Tree =
| Node of TalentValue:Talent * Children:Tree list
| Leaf of TalentValue:Talent
```

```
public class MyTalent
{
  public int Strength;
  public int Agility;
  public int Intelligence;

  public List<MyTalent> SubTalents = new List<MyTalent>();

  [...]
}
```

Listing 1: Talent tree data structure implementations (F# on top, C# below).

The example in Listing 1 implements a class and a discriminated union in F#, which together implement the behaviour of the C# class. The F# approach has separated the tree from the talent, where in the C# solution the tree emerges from the recursive nature of the class (see Listing 2). The C# solution is closer to the problem domain, but the F# solution is closer to the mathematical concept of trees. In C#, with the use of more classes, a generalisable tree walker could also be implemented. We argue that this would yield higher reusability of the code.

```fsharp
let rec foldTree folding init tree =
  match tree with
  | Node (t, c) ->
    let f = folding t init
    let cf =
      c
      |> List.map (foldTree folding init)
      |> List.fold folding init
    folding f cf
  | Leaf (t) ->
    folding t init

let sumNodes root =
  foldTree (fun t1 t2 -> Talent(t1.Strength + t2.Strength,
  ↪  t1.Intellect + t2.Intellect, t1.Agility + t2.Agility))
  ↪  (Talent(0,0,0)) root
```

```csharp
[...]
public MyTalent SumTalents() {
  var t = new MyTalent(Strength, Agility, Intelligence);
  if(SubTalents?.Count == 0)
    return t;

  foreach (var child in SubTalents) {
    var childTalentValues = child.SumTalents();
    t.Strength += childTalentValues.Strength;
    t.Agility += childTalentValues.Agility;
    t.Intelligence += childTalentValues.Intelligence;
  }
  return t;
}
}
```

Listing 2: Talent walker implementations (F# on top, C# below).

Consistency

In the Cognitive Dimensions framework consistency is the coherence between the language designer's understanding and the language user's intuition of the language[62]. This does not mean that consistency is the difference in language knowledge, but rather the difficulty of extrapolating behaviour and syntax of language features based on knowledge of a subset of the language or other language features.

F# uses a strict type system which infers types. This feature allows the programmer to omit explicit typing while still gaining the benefits of it. In some cases the type inference can cause confusion or act in an unexpected way, as when a `int16` value is used in the declaration of a `int` value. In Listing 3 an example of this can

be seen. Line 2 gives an error because an `int` literal and an `int16` name binding are multiplied. This behaviour is consistent with F#'s rules, but is surprising for programmers who are versed in C-style languages.

```
1 let x = 10s
2 let y = 2 * x
```

Listing 3: An example of type incompatibility in F#. 10s is an `int16` and 2 is an `int`.

Naming conventions can present consistency difficulties for some languages. An example of this are the type modules in F#, such as `List`. These modules supply helper functions for working with a particular type, an example of which is `List.append`. In C#, equivalent functionality would have been placed as instance methods on said types. This may cause some confusion for C-family programmers, as they may find functionality they seek in unexpected places.

In addition to naming conventions causing confusion, `lists` have another problem. F# and C# share the .NET runtime and can therefore use each other's language features. While this is an advantage, it also presents some disadvantages, most notably that F# lists and C# lists are not the same type. This clashes with programmer expectations and converting to the correct list type can be surprisingly difficult, which we have demonstrated in Listing 4.

```
1 private static List<T> GetParams<T>
  ↪  (Microsoft.FSharp.Collections.List<T> parameters)
2 {
3     return new List<T>(parameters);
4 }
```

Listing 4: Conversion from F# List to C# List.

In functional programming languages function signatures can often be specified by the programmer, to help the compiler catch unexpected behaviour. This is also possible in F#, however in an unexpected manner. Function signatures in F# are reported using the Hindly-Milner type system's syntax[78]. However, when the programmer attempts to declare the function signature manually, they cannot use the same syntax. Instead a Python-like syntax is used. An example can be seen in Listing 5.

```
// reported function signature
val add: int -> int -> int

// function definition without explicit signature
let add x y = x + y
// function definition with explicit signature
let add (x:int) (y:int) : int = x + y
```

Listing 5: Difference between reported and user-defined function signatures in F#.

In F# lambda expressions are denoted using the `fun` keyword and `->` operator. The use of `fun` vs. the use of `func` may initially be confusing for programmers, but is quickly learned. After the initial confusion, the feature is consistent with the rest of F#, as lambda expressions are defined like functions. This is not the case in C#, where lambda expressions are defined using the => operator. This is not consistent with the rest of C#, because C# does not use a function signature similar to the Hindly-Milner type system.

F# has two primary collection types: lists and arrays. The array-collection type can provide some benefits when looking up elements by index. However, when looking up an element by index, dot notation is used to call the `[]` function, thus a lookup becomes `array.[0]`. This clashes with expectations of a C-family programmer where `array[0]` is the norm.

While F# presents some consistency problems, they're are consistent within the language. This indicates that the problems may be experienced more by novice programmers and that they dissipate with experience.

Diffuseness/Terseness

This dimension measures the conciseness of a notation system on a scale from terse, meaning too brief, to diffuse, meaning not brief enough. The golden middle ground is referred to as concise. This measure is affected by the symbols used for operators as well as the notation system's naming conventions. If the notation is too brief, understanding its meaning may be quite difficult and small changes can have large consequences. On the other hand, notations that are too verbose, cannot be viewed on a single screen and are therefore more difficult to overview.

To compare the conciseness of both languages, two solutions to a problem will be examined. The problem consists of calculating three sums based on properties of a list of objects. The objects are given and the sums must be printed to the console. In Listing 6 the C# solution can be seen. The method takes a collection of `Items` and iterates over them, keeping a running tally of three sums. Once all objects have been summed, the results are printed to the console.

```
1  public void Solution1(IEnumerable<Item> Armour)
2  {
3    var totalAgi = 0;
4    var totalStr = 0;
5    var totalInt = 0;
6
7    foreach (var item in Armour)
8    {
9      totalAgi += item.Agility;
10     totalStr += item.Strength;
11     totalInt += item.Intellect;
12   }
13   Debug.Log($"Exercise 1\n\tAgility: {totalAgi}\n\tStrength:
       ↪  {totalStr}\n\tIntellect: {totalInt}");
14 }
```

Listing 6: Summing the attribute bonuses of a character's armour in C#.

The approach taken in F# is somewhat different, the solution can be seen in Listing 7. Instead of iterating over the given array, a map-reduce approach is used. On line 8 the array is piped into a map which deconstructs each `Item` into a triple. The triples are then piped into a reduce function which calls the sum function defined on line 1. In the start function on line 16, the sums are computed and printed to the console.

```
1    let sum (triplet1:int*int*int) (triplet2:int*int*int) =
2      let (a1, b1, c1) = triplet1.Deconstruct()
3      let (a2, b2, c2) = triplet2.Deconstruct()
4      (a1+a2,b1+b2,c1+c2)
5
6    [...]
7
8    let totalStats (armour:Item[]) =
9      armour
10     |> Array.map (fun a -> (a.Agility, a.Intellect, a.Strength))
11     |> Array.reduce sum
12
13   [...]
14
15   member this.Start() =
16     let i = ItemStore.AllItems()
17     let (agi, int, str) = totalStats(i)
18     Debug.Log("Agility: " + agi.ToString())
19     Debug.Log("Intellect: " + int.ToString())
20     Debug.Log("Strength: " + str.ToString())
21
22    [...]
```

Listing 7: Summing the attribute bonuses of a character's armour in F#.

The F# solution is slightly longer than the C# solution, which is due to dividing the functionality into several smaller functions. This lengthened the implementation of the first solution, but reduced the overall length of the code. The C# code ended up being 35 lines longer than the F# code. The full examples can be seen in Appendix E.

The most prominent syntactic differences between C# and F# are the operators, scope delimiters and line end delimiters. In C# blocks are denoted using the `{` and `}` symbols, whereas F# uses indentation. In addition statements are terminated using a `;` in C#, where a newline character is used in F#. These differences mean that F# uses fewer symbols in general than C#, however some programmers find the F#'s syntax more difficult to read[79].

Another difference that greatly affects terseness/diffuseness is the approach to code reuse the languages use. In C# inheritance and the Composite Pattern[80] are often used[81]. In functional languages function composition, chaining and currying are often used to implement design patterns[82]. Both approaches reduce the codebase, but do so in different ways.

Error-proneness

Errors produced by the programmer fall into one of two categories, either the error is a slip or a mistake. Slips are instances where the programmer knows what to do, but did something else by accident and mistakes are instances when a programmer makes a logical error. These errors can be exacerbated by language features.

According to [62], textual programming languages are inherently more error-prone than visual languages. The given examples are implicit declaration, line-ends and delimiters. Implicit declarations are not applicable in either C# or F#, however line-ends are used in both languages. C# uses `;` to denote line-ends whereas F# uses newlines. A C-family programmer may find it difficult to overview F# code for this reason. In C# a `;` need not be followed by a line-end and depending on the type of statement, practices may differ (it is not typical to break lines in `for`-loop declarations, but it is after variable declarations).

The strong type system used in F# can also cause unexpected errors. While the system prevents some errors down the line, C-family programmers would expect type coercion to assist with operations on integer values of different sizes. We gave an example of this in Listing 3, which yields an error because an `int` and `int16` are multiplied. This causes initial errors, but may prevent type errors later in development[83].

Programmers may inadvertently change the parameters of a function by separating parameters using commas (see Listing 8). In most cases this is valid F# and compiles, however the function now takes a single tuple parameter. This mistake

can occur without the programmer noticing any difference, until they have to invoke the function. At this point the invocation has changed from `add 2 4`(two parameters) to `addTupled (2, 4)`(tuple parameter). Both declarations are valid, but the latter may cause confusion when the programmer attempts to use the functions as higher-order, as he would have to pack all arguments in tuples. We have listed examples in Listing 8.

```
// Multiple parameters
let add x y = x + y

// Single parameter
let addTupled (x, y) = x + y

//Sum list without tupled parameters
let sum = [1..10] |> List.reduce add

//Sum list with tupled parameters
let sum = [1..10]
|> List.reduce (fun acc elm -> addTupled (acc, elm))
```

Listing 8: Examples of functions with and without tupled parameters and it's influence on their applications as higher-order.

Hard Mental Operations

The hard mental operations dimension defines how often incomprehensible expressions occur in the code. Hard mental operations often occur in conjunction with boolean expressions[62].

Boolean expressions are expressed similarly in C# and F#, with the only exception that C# uses `!` to negate expressions, whereas F# uses the `not` function. Boolean expressions in both languages are equally hard to read. We have illustrated examples in Listing 9. In this example the reader may incorrectly assume that the `&&` operator is evaluated before `!`, and that the expression is evaluated as `!(expr1 && expr2)`. The exact same case is present in F#. Programmers may reduce perceived ambiguity by inserting parentheses, but may risk changing the order of evaluation by doing so.

```
if(!expr1 && expr2) {
    //[...]
}
```

```
if not expr1 && expr2 then
    //[...]
```

Listing 9: Hard mental operations illustrated using boolean expressions in C# and F#.

In F# it is optional for the programmer to indicate types when writing functions.

Sometimes it may be necessary to verify that the compiler's inference is correct by looking at the deduced function signature. We argue that this may present hard mental operations in both languages, as the function signatures will quickly get incomprehensible as the number of arguments grow. Take for example the function signature of the `ReactTo` function that we wrote as part of the FRP plugin for Unity:

```
member FRPBehaviour.ReactTo : event:FRPEvent * condition:('T0 -> bool)
* handler:('T0 -> unit) -> unit.
```

We should underline that this function uses tupled arguments, because we wanted to overload it with a function to unconditionally react to events. But even without tupled arguments the definition would have been:

```
member FRPBehaviour.ReactTo : event:FRPEvent -> condition:('T0 -> bool)
-> handler:('T0 -> unit) -> unit.
```

In C# the equivalent would have been:

```
public void ReactTo(FRPEvent event, Func<T, bool> condition, Action<T>
handler)
```

Whether one or the other is easier to comprehend than the other is a matter of opinion. The problem is more prominent in F#, however, due to the type inference in the compiler.

Hidden Dependencies

Hidden dependencies discuss how many relationships there are between components, that are not visible from at least one of the components. We discuss several problems in this section, but we must underline that some of these problems can be mitigated by using an Integrated Development Environment (IDE), as they often allow programmers to trace dependencies.

The problem of hidden dependencies is prominent in both C# and F#, though in two different flavours (see Listing 10). In F# the problem is present on the function level. This is because F# programmers are allowed to write functions that are directly nested in a module. These functions may be imported into another module or namespace with the `open` keyword. Given that a name of a function does not collide, the function may be referred to without the use of its fully qualified name. In C# the problem occurs because programmers are allowed to reference code in base classes without using their fully qualified name. One example of this in Unity is that any `MonoBehaviour` may directly call `Destroy`. This method is actually a static method on the `GameObject`-class, from which `MonoBehaviour` inherits. This may make it more difficult to distinct between static and non-static methods and "hide" the base class from the reader. In F# programmers are required to give fully qualified names when dealing with classes.

```
class Base {
    protected static string
    ↪ Method() {
        return "Method";
    }
}

class Inherited : Base {
    private string Method2() {
        return Method() + "2";
    }
}
```

```
module X =
    let function () =
        "function"

open X
module Y =
    let function2 () =
        function() + "2"
```

Listing 10: Hidden dependencies in function/method calls in C# and F#.

In both languages the problem of function/method hiding may occur. An example of this is if a base class defines a `virtual` method in C# or an `abstract` method with default implementation in F#. This method may be hidden further down the inheritance tree by implementing a function with the same name without using the `override` keyword. This will give a warning at compile-time, which can be removed by supplying the `new` keyword in front of the method that hides the existing implementation.

C# has a modified version of the `goto`-statement, which traditionally allow programmers to jump to labels anywhere in the source code. In C#, however, the jumps are restricted to either a label in the same scope or in an enclosing scope[84]. Nevertheless, it may be hard to comprehend the exact target of a `goto` if it is deeply nested in multiple loops. According to a StackOverflow discussion[85], it seems that the `goto` statement sees limited use in practice.

The problem of hidden dependencies may also occur if a component is dependent on a global variable. In games it is common to see such types of dependencies[5], [86], [87]. This problem is easier to mitigate in F# because the global variable is per default immutable and the developer has to explicitly indicate if he wants a mutable variable. In C# it's easier to make slips, as everything is mutable per default. In Unity this problem is also present in-between components, as it's a common pattern to declare a field on a class that references some component and thereby assign that field from Unity's Inspector[88]. This has the consequence that there is no way of knowing which components depends on each other by inspecting the source code. Furthermore, it is not possible to trace dependencies backwards in Unity without writing custom Editor scripts[89].

Premature Commitment

Premature commitment describes how much guessing ahead the programmers must do when programming in a given language.

In F# there is a fixed ordering when defining types, that enforces all name bindings (`let`) to be declared before members. This is similar to the problem of *Commitment to layout* presented in [62]. Luckily, these declarations may quickly be moved around in F# source code by cutting and pasting. This problem is not present in C#, where the programmer is free to choose any ordering when declaring methods, fields and properties on classes.

In C# the problem of premature commitment may surface when dealing with class hierarchies. The problem is also present in F#, as it is also object-oriented, but we argue that C# is more prone to the class-related problems as it is object-oriented first. The problem of premature commitment arises when a programmer has to implement a base class, without being certain which other classes might inherit therefrom. This introduces guess-work and might result in missing functionality, unneeded class members and potentially the requirement of reimplementing the base class. This problem is even more prominent when inheriting from third party code that contains multiple classes[90], as the programmer may choose to inherit from one class and later discover that it was a wrong decision and therefore the the entire class must be reimplemented. The problem is more prominent in Unreal than Unity, as Unreal has multiple different base classes for components[91], where Unity has one.

Another small-scale issue of premature commitment arises when using F#'s collection functions (such as `List.map` or `Array.reduce`). These functions take as first argument a function and as second the collection to operate on. If they are used without the pipe operator (`|>`), the IDE will be unable to aid the programmer when he is implementing the mapping function until the second argument has been given. The correct order is thus to write the name of the function, add empty brackets, add the name of the collection and finaloly implement the function.

Another problem of premature commitment arises in F#, because circular dependencies between classes are not allowed. The dependencies between classes are visible in IDEs, where classes that are further down the source file list may depend on classes that are further up. If circular dependencies are needed in a program architecture, the programmer must define and implement an interface on one of the classes and reorder the source files. This constraint might seem annoying at first, but has the benefit of yielding class architectures with looser coupling[92].

Progressive Evaluation

Progressive evaluation defines how well a partially finished program can be executed and evaluated. Higher progressive evaluation means that more incomplete programs can be executed. C# and F# programs can only be executed if the source code is syntactically correct. We therefore deem that their progressive evaluation is equivalent. Both languages are supported by Read Eval Print Loop (REPL)-tools called called C# Interactive[93] and F# Interactive[94] respectively.

These tools enable programmers to run and evaluate anything from single lines of code to whole files. We used this tool frequently in the beginning of the project, when we were learning F# to experiment with different functions and constructs, before implementing them in the source code.

Role-expressiveness

Role-expressiveness defines how easily a program can be read and comprehended. The dimension is easily confused with hard mental operations or secondary notation, from which it should be kept apart[62]. Role-expressiveness thus defines how self-explanatory a program is.

Being languages that run in the same platform, C# and F# share many constructs and all libraries. This means that a discussion of the standard library is of little interest. There are, however, some differences in the syntax that we will highlight.

First and foremost C# uses either `var` or type names in variable declarations. In F# the equivalent is `let`, possibly followed by `mutable` to indicate mutability. Depending on the programmer's background, these keywords may be more or less expressive. From a mathematical background it makes sense to create name bindings by using `let`, as that's common in proofs and similar mathematical lingo. The type of a name binding may be inferred by how it's used. Furthermore, `let` indicates a name binding and not a variable, which further underlines that F# is pure until otherwise is expressed. This contrasts with the classic C-style way of defining variables; by using their type name. Some programmers that are less versed in mathematical notation may prefer this way, as it is more expressive of the variable's type. Finally, the `var` keyword is simply an abbreviation for 'variable', which goes well hand-in-hand with its purpose; a variable which the compiler should determine the type of. Common for the last two types of declarations are that they do not indicate anything about mutability and thus require knowledge about the language at hand.

Lambda functions are available in both languages. In C# they're expressed as `(a,b) => a + b`, where as in F# they're expressed as `fun a b -> a + b`. In this case F# uses the keyword `fun` to indicate a lambda, which is quite literally an abbreviation of 'function'. One thing worth noting is that the `fun` keyword may easily be confused with the English word fun. Another abbreviation such as `fn` or `func`, would probably have been better. C# uses the =>, which is of limited expressiveness, especially because C# does not use arrow-style function signatures anywhere else.

In order to declare custom data structures in C#, one can use `class`, `struct` or `enum`, depending on the purpose of the structure. In F# all data structures are constructed using the `type` keyword. Depending on the symbols used in and around the definition, the outcome will change (see Listing 11). Consequently

this means that the `type` keyword in F# has very limited role-expressiveness, compared to those of C#.

```
type Enum =
| B = 0
| C = 1

type Union =
| B
| C

type DiscriminatingUnion =
| B of bool
| C of char
```

```
type Record = { b: bool, c:
↪  char }

type Class() =
    let b = true
    let c = 'x'

[<Struct>]
type Struct(b:bool, c:char) =
    member this.B = b
    member this.C = c
```

Listing 11: Different kinds of data structures defined using the `type`-keyword in F#.

Secondary Notation and Escape from Formalism

Secondary notation and escape from formalism defines how well a programming environment supports conveying information that is not part of the source code. Typical examples of such are comments, indentation and grouping code into paragraphs in textual languages[62].

C# and F# are very alike in this dimension. They both support the `//`-operator, which indicates that the rest of the line should be commented out and matching pairs of `/*` and `*/`, which comments everything out between them. Furthermore functions, methods, classes and more or less any program construct may be annotated with `///`-comments, which allow the programmer to add eXtended Markup Language (XML)-documentation[95]. The programmer may use this to describe the intention of the construct along with its arguments and, if needed, link to other constructs in the source code. Other developers may open this documentation in a pop-up box in their IDE, whenever such construct is encountered.

Viscosity

Viscosity defines how much effort a developer has to put in to make a small change. [62] notes that textual languages are less viscous than visual programming languages in their comparison of Basic, ProGraph and LabVIEW.

We argue that the primary difference between C# and F# is scope delimitation. C# scopes are delimited by pairs of curly brackets, whereas in F# they are delimited by indentation. If a programmer is to move code from one scope to another in C#, he would have to either insert or delete pairs of curly brackets, whereas in F# he would select the code that needs to be moved and press TAB or Shift+TAB.

Visibility and Juxtaposability

Visibility and Juxtaposability determines whether required material is accessible without cognitive work[62]. In textual languages this dimension is not necessarily determined by the language, but more so by the environment (IDE).

There are two prominent IDEs for the .NET platform: Visual Studio and Rider. Both IDEs support numerous ways of making source code available. Examples of such are:

- Splitting the text editor both horizontally and vertically, such that two files can be open side-by-side
- Jumping to implementations by control-clicking.
- Hovering over function or type names to read a description of what they're meant for.
- Opening documentation pop-ups that explain how to use of classes and methods.

4.2 Usability Evaluation

In this section we present the usability evaluation that was conducted as part of the project. We first go over the setup of the test, presenting participant selection criteria and describe the tasks that were given to the participants. We then turn to the results of the test, where we adopt the analysis method described in Champagne Prototyping [64]. The Champagne Prototyping method yields rather shallow analysis results and hence we present a more in-depth analysis of the problems using the Cognitive Dimensions framework. Finally we discuss potential sources of errors in the experiment.

4.2.1 Setup

The test setup draws inspiration from the Champagne Prototyping method and Discount Method for Language Evaluation. The tasks, prototype and participants were selected according to the former, whereas the use of a cheat-sheet for the participant was inspired by the latter. Contrary to the suggestion of using a text editor in Discount Method for Language Evaluation, we chose to use IDEs, as we're testing well established languages. We allowed the users to choose JetBrain's Rider or Microsoft's Visual Studio, depending on what they were used to.

For each participant a one and a half hour session was planned. We expected that actual coding time would be roughly one hour, as we conducted a questionnaire before the test to learn about the test participants' experience with Unity and a debriefing interview after the test to allow the participant to share their opinion on F# in Unity, C# and functional programming. Out of the one hour coding time, we intended to use 20 minutes on C# and 40 minutes on F#, as the participants were required to have experience with C# in Unity and therefore likely would complete the C# tasks faster.

The cheat-sheet served a two-fold purpose and was made available online[1] using Github pages. The first purpose of the document was to give the test participants an introduction to F# in Unity prior to the test and second to act as a cheat-sheet during the test. Similarly, the tasks were also made available online during the test[2]. We also created a Github repository, in which the test-setups were stored[3]. The master branch of said repository holds a Unity project with eight scenes, one for each of the test cases. The purpose of this setup was to remove Unity as a factor in the experiment and avoid having the participants spend time setting up scenes. For each of the test participants we created a new branch in the repository, which would allow us to view each of the participants' code in isolation.

During the tests we recorded the screen and audio on the test computer. The files were not transcribed, but whenever we quote one of the participants we refer to transcriptions of larger pieces of dialogue that are listed in Appendix D. We do so in order to give the reader more context on the quotes and avoid "plucking" sentences out of their context.

Test Cases

We created a total of eight test cases, which fall into four categories; player controller, generalised walkers, map-reduce and "concurrent" update. We list concurrent with double ticks here, as the tasks were designed to be implemented with `async/await` in C# and asynchronous workflows in F#. The reason we chose this was that it would be possible for the participants to implement a concurrent version with minor rework, given the right setup. Unity's concurrency model requires more boilerplate code, which we deemed would not provide any useful information.

FPS Controller The participant is to implement a FPS controller, i.e. a component which can be added to a player character to move it around the world with the WASD-keys and rotate the camera with the mouse.

[1] https://sppt-2019.github.io/unity-fsharp-introduction/. Please note that the document is in Danish, as all participants were Danes.

[2] https://sppt-2019.github.io/unity-fsharp-introduction/tasks/. Please note that the document is in Danish, as all participants were Danes.

[3] https://github.com/sppt-2019/Unity-FSharp

3rd Person Controller This test case is similar to the previous test case, except that the camera must rotate around the player character.

Talent Tree-Walker The participant is first asked to implement a data structure that can hold a talent tree. Afterwards the player is given a pre-made talent tree from which the participant must calculate two things. At first a character's bonuses in three attributes should be calculated by summing all talents that are `Picked` and afterwards the maximum achievable bonus of all attributes by summing all talents.

Armour Graph In this test case, the participant is given a list of armour equipped on a character. The participant is to implement code that sums the character's bonus in three attributes from the armour. In the second half of the test, we assume that certain pieces of armour can scale the attributes from all other pieces of armour and ask the participant to calculate the scaled attribute bonuses.

Dialogue Tree The participant is to first implement a data structure that can hold a dialogue tree. Afterwards the participant must parse a list of dialogue options into a tree using his own data structure. Finally the participant is to find all unique dialogue paths that has a certain outcome.

Currency The participant is presented with three different types of coin. He must first implement code that exchanges a given number of said coins into the minimum number of total coins. Afterwards he should add a function to calculate whether a player can buy a certain item from a vendor and finally implement code that buys the item and updates the player's wallet.

Unit Management (RTS) The participant is to implement an inverse state machine. By inverse we mean that the state machine should hold collections of entities for each state in the state machine. At each `Update` the state machine should map the corresponding state's update-function over each collection to create new collections of entities. Finally the participant is asked to implement a "concurrent" mapping of the update-function.

Magnetic objects The participant is to simulate magnetism. He is presented with a list of objects, some of them which are magnetic. All magnetic objects should be attracted to a common center-point at a given speed. In the second half of this task the participant is asked to implement a "concurrent" version of the simulation.

Category	F# Task	C# Task
Player Controller	FPS Controller	3rd Person Controller
Generalised Walkers	Talent Tree	Armour Graph
Map-Reduce	Dialogue	Currency
"Concurrent" Update	Magnetism	Unit Management

Table 1: Categories and their associated tasks.

In Table 1 we list the categories and their associated tasks. As we wanted to explore how suitable Carmack's and Sweeney's approaches to game development are, we decided to use a fixed ordering of the test categories for each participant. We had initially planned that the participants would be given one of the tasks from the controller category as the first task and distribute the remaining semi-randomly, but discovered after the first actual test that the participants would only have time to solve a single task. We therefore decided that the participants would be given starting tasks from different categories. This ordering allow us to compare each of the participants' the solutions to tasks of the same category.

FRP System

We chose to add support for F# via the Unity F# Integration plugin[19]. The test cases are to be implemented in a FRP system that we developed in F#. The FRP system introduces a new class called `FRPBehaviour`. Classes that inherit from `FRPBehaviour` inherit a method called `ReactTo`, which exists in two variations. The first variations accepts an event type (such as Update, Keyboard or MouseMove) and a handler. This method will unconditionally invoke the handler whenever an event of the given type occurs. The second variation accepts an event type, a filtering function and a handler. This variation will only invoke the handler when the filtering function returns true. Listing 12 shows an example of the magnetism task implemented in F#. This code rotates all balls to look at the center and thereby moves them forward.

```
1  member this.Start() =
2      let balls = GameObject.FindGameObjectsWithTag("Magnetic")
3
4      this.ReactTo (FRPEvent.Update,
5          (fun () ->
6              let center:Vector3 = getCenter(balls)
7              let updatedBalls =
8                  balls
9                  |> Array.map (fun b -> lookAt(b, center))
10                 |> Array.map step
11         )
12     )
```

Listing 12: Implementation of the magnetism task in F#. The `getCenter` and `lookAt` functions are excluded for brevity.

The F# plugin also allows programmers to implement gameplay code using the standard Unity strategy (i.e. a chain of if-else statements to check for input in the `Update`-method). We decided that we wanted to remove this option entirely and therefore implemented an `Update` method in the `FRPBehaviour` that was reserved for condition checking. This method cannot be overridden, which entirely

prevents the user from using `Update`-based programming. We took this decision partly because scientific research underlines that FRP is well suited for game development[46], [47], [51] and partly because we argue that the temptation of writing C# code in F# syntax would be too large if we did not take counter measures.

Participants

The participants for this experiment were recruited by sending emails to game studios and other game-related companies in Denmark, asking if their employees were interested in participating in the experiment. The participants were required to have experience with C# and Unity in the game development industry. We gathered a total of six participants; two participants with Indie game-development experience, three from a company that creates augmented reality/virtual reality applications and one who teaches children game development.

For selection criteria we looked for participants that had developed games or had significant experience with C# in Unity. In addition the participants need not have experience with F#. The criteria were ranked as follows:

1. Game development experience.
2. Unity & C# experience.
3. No F# experience.

Pilot Test

We conducted a pilot test with a participant from the Programming Technology specialisation course at Aalborg University in order to get some feedback on the setup. The test participant has experience with Unity from Indie game development and teaching Unity programming to children in the Danish secondary school/advanced level (not to be confused with our test participant, who has a similar job). Contrary to the requirements for the actual test, the pilot test participant had experience with F# and functional programming in general.

Prior to the pilot test we had assumed that a 50/50 distribution between C# and F# would give the most valuable results, as it would not skew the results timewise. During the pilot test the participant, who had prior experience with F#, was able to complete three tasks in C# and two in F#. We therefore decided to change the distribution to 20 minutes of C# and 40 minutes of F#.

Apart from that, the participant noted that some of the tasks were hard to understand, mainly because they required the participant to utilise code that was

pre-implemented. We decided to rewrite the tasks and insert code snippets with existing classes where-ever the participant would have to use them. Finally, the participant noted that the function-name `ReactTo`, which is used to bind an event handler to a given event, was odd. He suggested renaming it to `Bind`. We chose to ignore this suggestion, with the argument that the sound of `ReactTo` has a strong connection with FRP, whereas `Bind` has a stronger connection with conventional functional programming.

4.2.2 Results Analysis

The result of the test was six git branches with source code written by the participants along with five video-files. Sadly the video-recording program broke down during one of the tests and we were unable to recover the file. We took notes during the tests and will refer to those instead. Whenever we quote the notes rather than the participant, we will clearly indicate that.

In general the participants were able to complete roughly one test case, some didn't and some started the second as well. This was the case for both F# and C#. Table 2 lists the participants and which exercises they were assigned. Please note that participant four was assigned the Dialogue Tree task, but asked for another task as he found it too hard.

Participant	**F#**	**C#**
1	FPS Controller	3rd Person Controller
2	Magnetism	Unit Management
3	Armour Graph	Talent Tree
4	Dialogue (skipped as it was too hard), FPS	3rd Person Controller
5	Magnetism	Unit Management, Currency
6	Dialogue	Currency

Table 2: Participants and their assigned tasks in F# and C#.

Questionnaire

All participants were asked to estimate their own skill levels in these categories and give a ballpark estimate of how many Unity applications they had developed. The results can be seen in Table 3.

Participant	C#	Unity	Game Dev	Functional	Unity Apps
1	9	9	5	3	25
2	8	8	7	2	10
3	8	8	2	1	12
4	10	10	8	1	10
5	9	9	9	2	10
6	8	8	9	6	5

Table 3: Participants self-evaluation scores.

Data Processing

In the following section we will process the data from the tests, using methodology presented in Section 3.2. The tests were constructed using the Champagne Prototyping methodology and the analysis of the data will also follow this approach. Champagne Prototyping uses two methodologies for data processing; Attention Investment Model and Cognitive Dimensions.

4.2.3 Champagne Prototyping

In this section we analyse the video files from the usability experiment using the champagne prototyping methodology. This method uses two different analysis strategies; attention investment and cognitive dimensions. In both of these analyses we count mentions of each feature or dimension. We present these using bullets in tables, which is recommended by [64]. A closed bullet (●) means that a participant mentioned an aspect of the feature, when the participant mentioned it multiple times an open bullet is used (○). The categories, *Correct & Unprompted* and *Correct & Prompted* are instances where the participants mention or explain a feature correctly and/or positively. The last category are instances where features are mentioned negatively, incorrectly or is a cause for confusion.

4.2.4 Attention Investment Model

The attention investment model, presented in Section 3.2.4, can be used to map out the programming steps undertaken by a user. This section will endeavor to do so, using the user test video material. Firstly, user comprehension of the feature being evaluated is measured. This feature is the use of F# and FRP in gameplay programming. In order to measure comprehension, the instances where participants mention aspects of the feature were recorded and categorised, as can be seen in Table 4.

Recognised Aspects	Correct & Unprompted	Correct & Prompted	Incorrect
Modularity	●●		●
ReactTo	○	●●○	●●●
Types	●●		●○○○○○
List Operations	●	●●○	●○

Table 4: User comprehension of the FRP features.

In addition to a measure of the participants' comprehension, the attention investment model also provides a quantification of their efforts in the programming activity. This consists of four metrics, mentioned in Section 3.2.4. This can be seen in Table 5. The risk metric measures the amount of times participants mentioned or discussed things that could go wrong, which includes increased difficulty of some tasks using F#. The cost metric is the attention and time required to switch to F#. Any musing over the difficulties of switching over is included. The payoff is the reduced cost of gameplay programming after switching to F#. The imperative alternative metric is the number of times participants mentioned the problems in C#, or other imperative languages, that form the basis for switching to F#.

Attention Investment	
Investment Risk	●●●●○
Investment Cost	●○○○○
Investment Payoff	●○○○○
Imperative Alternative	●●●

Table 5: Attention investment findings.

The participants were able to correctly use and describe F# and FRP behaviour in some instances and struggled in other instances. As can be seen in Table 4, not all participants were cognisant of all feature aspects, e.g. all participants misunderstood or struggled with the type system. Functional programming claims a greater degree of modularity than imperative languages[10], however, less than half of the participants expressed cognisance of this. Some participants wrote much more modular code in F# than in C#, but did not mention it.

As can be seen in Table 5 the participants noted a high cost with a high payoff. Participants could see the usefulness of FRP, but several participants expressed uncertainty of any benefit provided by F#. Another point is that very few participants noted the problems with existing solutions, even when compared to F#. The high cost is attributed to loss of productivity while a developer learns to use F#.

4.2.5 Cognitive Dimensions

In this section we use the cognitive dimensions framework to aid the analysis of the video from the usability test. In this analysis we count the number of times the participants experience or mention problems with each dimension. We use the same notation that was presented in the previous section. The frequency of mentions can be seen in Table 6. The mentions counted are any instances where the participants said or did some thing that fit under a cognitive dimension. Many of the counted instances are therefore not explicit statements made by the participants, but rather instances where their actions fell under one of the categories.

Dimension	**F#**	**C#**
Abstract Gradient	●	●
Closeness of Mapping	●●	
Consistency	●●○○○	●●
Diffuseness/Terseness	●●○	●
Error-proneness	●○○○○	●●
Hard Mental Operations	●●○	○
Hidden Dependencies	●●	●
Premature Commitment	●●●	
Progressive Evaluation	●●○	●●
Role-expressiveness	●●○●	●●
Secondary Notation and Escape from Formalism		
Viscosity	●	●
Visibility and Juxtaposability	●○	●

Table 6: Cognitive dimensions findings (results from participant 5 has been omitted).

It is not surprising that participants encountered more problems with F# than they did with C#. Participants were selected based on the criteria stated in Section 4.2.1, these included requirements for C# experience and limited to no F# experience. This allowed us to study the potential adoption difficulties of F#. The higher frequency of mentions when using F# is inline with this.

While an overview can be gained from Table 6, it does not provide adequate information of *why* a participant would undertake a certain strategy or action. Therefore, selected instances among the mentions, are analysed further in the following sections.

4.2.6 Instance Analysis

Some instances during the test made the difficulties of using F# and the functional paradigm more clear than others. In this section, these instances are explored and

examined in order to discover the problems faced by developers adopting F# in a gameplay programming setting. We list examples under seven different dimensions here. Those seven dimensions were chosen because we could find concrete code examples or situations during the sessions where the dimension was visible.

Consistency

All participants had prior C# experience which affected their expectations. This was apparent when participants applied C# methodology in the F# code. An example is type-confusion. Several participants noted that they preferred strict typing or specifying types manually. An example of this can be seen in Listing 13.

```
1 [<SerializeField>]
2 let mutable _velocity = 5.0f
3 [...]
4 member this.HandleMoveForward() =
5   this.transform.position+=new Vector3(0,0,this._velocity)
```

Listing 13: Problem experienced with types in F#. The `Vector3` constructor accepts `floats` and are invoked with `int`-parameters.

In Listing 13 the participant correctly types the `_velocity` variable on line 2. However, when attempting to set the object's position on line 5, the participant uses `0` instead of `0.0f`. This valid in C#, but not in F#. This is an instance of confusion surrounding the automatic type inference of variables and the typing of literals.

```
1 let moveMagneticBalls (objs:GameObject[]) (center:GameObject) =
2   objs center |> Array.map (fun i ->
3     i.transform.LookAt(center.transform)
4     i.transform.Translate(i.transform.forward * Time.deltaTime *
    ↪  speed))
```

Listing 14: Closure misunderstanding. The user attempts to catch `center` in the closure by piping it into the map-function.

Some participants also experienced issues with closures. In Listing 14 a participant has defined a function to move a number of objects towards a center point. The center point, `center`, is passed as a parameter, but the participant became confused as to how to pass it to the lambda expression. Therefore center was added on line 2 after `objs` and piped into the `map` function. This causes an error because `objs center` is now a function invocation to a non-existent function,

`objs`, with the argument `center`. Passing `center` to the lambda function is unnecessary, because it is captured in the lambda's closure.

The behaviour described is consistent within F#, but not with the expectations of the participant. Arguably, this instance is a product of the participant's inexperience with functional programming, however it is an example of the disharmony between the largely consistent rules of F# and the expectations of programmers. These consistency issues were primarily present in the F# code, which is not surprising considering the participants' experience.

Error-proneness

Several of the participants did not connect indentation with scope. This meant that the scoping of variables at times presented a challenge. In most test cases the solutions were implemented in a class. As a consequence, the improperly indented function bodies would work initially because the function body would be part of the class instead of the function. Such incorrectly indented functions may still be called within the scope of the class (see Listing 15).

```
1  type FRP_FPSController() =
2      inherit FRPBehaviour()
3
4      member this.Start() =
5        this.HandleMoveForward()
6
7      member this.HandleMoveForward() =
8      let newPosition = this.transform.position + new Vector3(0.0f,
       ↪  0.0f, _velocity)
9      this.transform.position <- newPosition
```

Listing 15: Incorrect indentation of `HandleMoveForward`. A problem is reported when code is added after the function declaration.

In Listing 15, lines 7 and 8 are indented incorrectly, but the code compiles and behaves as expected. The IDE gives a warning on both lines, but no errors are thrown until new `member` functions are defined below `HandleMoveForward`.

Premature Commitment

In F# all `let` functions must be declared before the first `member` function. An example of this can be seen in Listing 16. This caused some initial confusion for participants, but it was quickly overcome. The reason for the strict order was not intuitive for the participants. The difference between the use of `let` and `member`

is the accessibility of the method or function in question. The `let` keyword denotes a function in the class instance's scope, effectively a private function. The `member` keyword denotes an instance function, which is similar to methods in C#.

```
1 type Candy(price:int) =
2   member val Price = price with get
3   let discountTuesday = this.Price / 2
```

Listing 16: Incorrect order of function declarations. `let` declarations must come before `members`.

Role-expressiveness

Unit and Void In F# the last expression in a function body is implicitly returned. In some cases that should not be the case and the function should return `Unit` (equivalent to C#'s `void`). If the last line in the function body is not of type `Unit`, programmer must add an additional line containing only `()`. This confused some of the participants, as they felt that it was unnecessary to explicitly indicate a non-existing return type. One of the participants even asked directly what `Unit` was and the monitor answered with an explanation. Approximately fifteen minutes later the participant encountered another `Unit`-type problem and was unable to recover without an additional explanation.

Let Declarations Another role-expressiveness problem we encountered was that some participants attempted to `let` declare multiple bindings without initialising them. Some participants assumed that `let` declarations without assignment would result in default values (such as `null` for classes).

Pipe Operators A single participant also noted that he found F#'s pipe operations *"not nice to read"*. He stated that he would prefer the SQL-like variation of Language Integrated Query (LINQ) in C# because it is closer to plain English. The monitor asked if it was related to the names of the functions (`map` and `reduce` or `Select` and `Aggregate`), to which the participant stated that it was solely related to the way pipe operations were structured.

Bindings and Operators In F# the = symbol is used to denote two different operators; the value-binding operator, and the equality operator. Notably the symbol is not used for rebinding (F#'s equivalent of assignments), which it is in C#. A comparison of the two different rebinding/assignment styles can be seen in Listing 17. Rebindings are only allowed on `mutable` variables, which should be limited to a scope.

```
let mutable x = 0
x <- 1
```

```
int x = 0;
x = 1;
```

Listing 17: Assignment Comparison in F# (left) and C# (right).

Furthermore, since `x = 1` is valid in F#, no errors were encountered immediately. Instead the program behaved unexpectedly and the participants received a warning, that the value of a boolean expression was being ignored. None of the participants were able to deduce the source of the error without assistance from the monitor. The degree of assistance required ranged from a hint to the monitor explaining the problem so that the test could continue.

Some participants encountered minor errors when declaring variables, because variables are implicitly immutable in F#. This is the opposite of C#, where the `const` keyword is used to explicitly declare a variable immutable. However, once the participants realised this, they had no issue using it.

Types and Type Inference Many participants had problems with the type system and type inference. The functions provided in the sample sheet had explicitly typed parameters, which the participants partially mimicked when declaring their own functions. However, participants did not specify function return types nor did they utilise typing of their namebindings. An example of such a function can be seen in Listing 18.

```
[...]
let getTotalWithMod (items:Item list) (attribute:Item->int)
↪  (attributeMod:Item->float32) =
[...]
```

Listing 18: Participant function with type annotations on parameters, but not on return type.

In addition several participants remarked that they preferred strictly typed languages, even though they had encountered several type errors. Some of the participants had Python experience, which may explain this assumption, because Python 3.x annotates function parameters similarly to F# and Python is dynamically typed.

Another problem faced by the participants were related to class type declarations, where the participants did not realise that the brackets after a type's name could be used to pass constructor arguments.

Function Declarations Some participants also expressed frustration with how functions are defined. The issue may stem from a problem with our sample sheet, which did not contain an example of a simple function definition. Plenty of functions were defined, but only ever as part of examples of other language features. Furthermore, functions share the `let` keyword with name bindings, which contributed to the confusion during the test.

Several participants struggled with lambda expressions in F#. Some of these participants expressed that they were not familiar with lambda expressions in C# either, but even the participants with lambda expression experience from C# had issues with them in F#. The difference in syntax can be see in Listing 19.

```
1 str => Console.WriteLine(str);
```

```
1 fun str -> printfn str
```

Listing 19: Lambda Expression Syntax, C# on the left and F# on the right.

Secondary Notation

In the test we saw very limited use of secondary notation. This may be caused by the relatively small tasks, along with the fact that the code was not meant to be used in production nor expanded upon. The participants, however, would often open documentation pop-up boxes to read about functions and methods before putting them to use. Generally, our intuition was that the participants had an easier time understanding the C# documentation than the F# documentation. We suspect that there two reasons:

1. The programmers were more experienced in C#.
2. F# uses the arrow-style function signatures (e.g. `val map : mapping : ('T -> 'U) -> list:'T list -> 'U list`, which is the signature of `List.map`) in its documentation.

This notation indicates that a function of multiple parameters can be curried. Currying is not supported in C# and was a concept that most of the participants were unfamiliar with.

Viscosity

In our test cases, viscosity is particularly visible in the "concurrent"-update category. The reason for this is that the participants were asked to develop a sequential solution first, followed by a concurrent implementation. Generally viscosity is low in both languages. In F# we saw the magnetism task implemented using the

pipe operator. Such an implementation can be extended to a concurrent solution by piping into `Async.Parallel` and then `Async.RunSynchronously` (see Listing 20). A similar solution can be achieved in C# using LINQ.

```
let speed = 3f;

let moveBallForward (ball:GameObject) =
    ball.transform.Translate(ball.transform.forward *
    ↪  Time.deltaTime * speed)

let Update () =
    let balls = GameObject.FindGameObjectsWithTag("Magnetic")
    balls
    |> Array.map moveBallForward
    ()

let UpdateAsync () =
    let balls = GameObject.FindGameObjectsWithTag("Magnetic")
    balls
    |> Array.map (fun b -> async {moveBallForward})
    |> Async.Parallel
    |> Async.RunSynchronously
    ()
```

Listing 20: Transforming from sequential to concurrent list operations in F#.

As with many other dimensions, viscosity can also be affected by the programmer's style of programming. This is exemplified in Listing 21 through Listing 23, both of which are taken from one of the solutions in C#. In order to implement "concurrent" update, the participant had to construct a new list in the `Update`-method and wrap the calls to the state methods in `Task.Run` (e.g. `updateTasks.Add(Task.Run(() => Flee(fleeingShooter)))`). This change is manageable, but imagine how much effort it would take if we wanted to add an additional state to the units. If such change was to be implemented, the programmer would have to:

- Add an additional list and `foreach`-statement in Listing 21.
- Add an additional case to the switch-statement in `JoinState` in Listing 21.
- Change the signature of the `RemoveFromList` in Listing 23 to accept another list and add an additional `else if` for the third list.
- Add an additional case to the switch in Listing 22, and update the call to `RemoveFromList` in all other cases.

An alternative and less viscous solution was implemented by another participant in the test. We have listed that in Appendix E.1.

```
class StateMachine : MonoBehaviour
{
    [...] //Pre-implemented code

    private List<Shooter> fleeingShooters;
    private List<Shooter> movingShooters;
    private List<Shooter> attackingShooters;

    public void JoinState(Shooter shooter, State state)
    {
        switch(state)
        {
            case State.Fleeing:
                fleeingShooters.Add(shooter);
                break;
            case State.Moving:
                movingShooters.Add(shooter);
                break;
            case State.Attacking:
                attackingShooters.Add(shooter);
                break;
            default:
                break;
        }
    }

    private void Update()
    {
        foreach(var fleeingShooter in fleeingShooters)
        {
            Flee(fleeingShooter);
        }
        foreach (var movingShooter in movingShooters)
        {
            Move(movingShooter);
        }
        foreach (var attackingShooter in attackingShooters)
        {
            Attack(attackingShooter);
        }
    }

    [...] //TransferState
    [...] //RemoveFromList

    [...] //methods for each unit state
}
```

Listing 21: Example of viscous C# implementation of the Unit Management Test.

```
public void TransferState(Shooter shooter, State state)
{
    switch (state)
    {
        case State.Fleeing:
            fleeingShooters.Add(shooter);
            RemoveFromList(shooter, ref movingShooters, ref
             ↪ attackingShooters);
            break;
        case State.Moving:
            movingShooters.Add(shooter);
            RemoveFromList(shooter, ref fleeingShooters, ref
             ↪ attackingShooters);
            break;
        case State.Attacking:
            attackingShooters.Add(shooter);
            RemoveFromList(shooter, ref movingShooters, ref
             ↪ fleeingShooters);
            break;
        default:
            break;
    }
}
```

Listing 22: TransferState-method, which is part of the viscous Unit Management implementation from Listing 21.

```
private void RemoveFromList(Shooter shooter, ref List<Shooter>
 ↪ list1, ref List<Shooter> list2)
{
    if (list1.IndexOf(shooter) != -1) {
        list1.Remove(shooter);
        return;
    }
    if(list2.IndexOf(shooter) != -1) {
        list2.Remove(shooter);
        return;
    }
}
```

Listing 23: RemoveFromList-method, which is part of the viscous Unit Management implementation from Listing 21.

Visibility and Juxtaposability

All the participants in the tests chose to use Visual Studio, which was used for both C# and F#. The differences in this dimension are thus minimal. It is interesting, however, that none of the visibility and juxtaposability enhancing features of the IDE were used during the tests. This may have two possible explanations:

1. The tests were relatively small-scale and required minimal interaction with existing code.
2. When existing code was needed, the participants would refer to the cheat-sheet or task descriptions, where example code was given. Some participants actually chose to open those documents side-by-side with Visual Studio.

4.2.7 Threats to Validity

In this section we will examine potential sources of errors and other threats to validity. Such threats are categorised as either internal or external. An internal threat occurs when data is mishandled, misinterpreted or in some other way skewed to such a degree that the results are untrustworthy. The second category consists of errors caused by the data being inapplicable to other cases. This is inline with terminology outlined in [96].

Internal Validity

In this section the internal validity is explored. The goal of this section is to map out the possible shortcomings originating from data handling and interpretation. Each potential threat is explained in turn and actions undertaken to mediate the threat are outlined. In some cases a threat cannot be sufficiently mediated, in which case it is simply listed.

Evaluation Parameters The user tests were evaluated in accordance with the Champagne Prototyping method (see Section 3.2.5). This methodology is intended to be used to measure the usability of a single feature, not an entire programming language. Therefore it was modified to fit our case better. The methodology consists of two other usability techniques: Cognitive Dimensions and Attention Investment Models (see Section 3.2.3 and Section 3.2.4). We modified the Cognitive Dimensions aspect to focus on the languages as a whole and kept the feature focus of Attention Investment Model. Thus the evaluation parameters were centered on the users' experience with F# and their understanding of the FRP system.

The Cognitive Dimensions framework is originally intended to estimate the usability, or provide a vocabulary to such an estimation, of a notational system such as a programming language. Therefore our modification of Champagne Prototyping is to use Cognitive Dimensions for its original purpose. Furthermore, a single aspect of Discount Method for Language Evaluation was used, namely the sample sheet (see Section 3.2.2). This was employed to assist test participants with F#.

Task Difficulty The user test consisted of a number of tasks that participants were asked to complete. These tasks were inspired by game development scenarios and applicability of functional idioms. Before the test we were worried that some tests were more difficult than others. This could skew the results, as participants completing the more difficult tasks would struggle more than participants completing the easier tasks. The dialogue tree task (see Section 4.2.1) presented a much more difficult problem than expected. The task relied on recursion and tree structures which were unfamiliar to the participants.

After the user test it became apparent that some tasks were indeed more difficult than others, significantly so in some cases. The difficulty of the tasks themselves did not affect the results to the degree we had expected, instead the difficult tasks brought conflicting idioms and faulty problem solving approaches to light that may not have been as apparent in the easier tasks.

Task Presentation Some tests were not formulated clearly enough. This meant that participants misunderstood the tasks and therefore did not sufficiently complete the task. These misunderstandings came from unclear task descriptions and lacking context. We attempted to mediate this by conducting a pilot test. This test clarified several unclear task descriptions before the user test. Unfortunately this test did not catch all the obscurities in the tasks.

Furthermore, some tasks were problematic, not because of formulation, but because of the test definition. An example of this is the dialogue tree task. We observed that the participants misunderstood the purpose of the task and attempted to solve it in a suboptimal manner. The problem was partially caused by a preimplemented class, which the participants had to use. We had attempted to mediate the problem by listing the class' source code, but it seemed the participants could not visualise the data structure.

Sample Size & Test Duration The user test was conducted with six participants outlined in Section 4.2.1. This was inline with the Champagne Prototyping methodology. In order to analyse the test results qualitative methods were employed. According to [97] six participants should allow us to discover 80% of the usability issues. However, there is no consensus on the optimum number of participants for such a usability study[98]. However, the participants only had an hour

to complete exercises in both F# and C#. This meant that most participants only managed to complete a single test case in each language, which provides only limited insight into the usability of the languages. Some test cases focused on certain aspects of the programming language's idiom, such as recursion, which some participants were unfamiliar or inexperienced with.

These problems could be mediated by allocating more time for the tests, however taking up an hour of the participants' day, presents scheduling issues already. Therefore, extending the tests was not an option. In order to mediate this issue the test could be restructured so that the participants worked solely with F#. This would provide more information about the use and challenges of F#, but not allow us to compare the F# and C# code.

External Validity

This section explores the potential threats originating from outside the test itself. This means any threat that can affect or skew the results that is not directly related to the test conduction. An example of this is whether the test tests the right things and whether the findings are broadly applicable or not.

Applicability to Game Development The test cases were designed to mimic gameplay programming scenarios. These cases were constructed to showcase situations where conventional and functional solution strategies could compete. However, the realism of the these cases is questionable. Our knowledge of the game development industry stems from a literature study and personal hobby, which need not align with industrial reality.

Some participants commented on the realism of the test cases. One participant noted that the character controller tasks were not representative of real development as they are freely available, and even if they weren't the developer would buy one from the asset store. Other participants indicated that they had spent a lot of time writing character controllers.

Development Environment & Test Setup In order to use F# with Unity a custom plugin was used. This plugin was developed by another student on the Programming Technology specialisation at Aalborg University[19]. Unity does not officially support F# and in order to run the F# code a work around was employed. The plugin automated this process and allowed the test participants to focus on the task at hand. However, the use of the plugin presented some problems. Compilation of F# code was done in the Unity editor instead of the IDE. These problems were mediated along the way, by reporting them to the developer. The plugin was patched repeatedly during test construction, and no plugin errors were encountered during the test.

Some errors were encountered during the test, however these errors originated from the setup code. Some errors had been identified and fixed as a result of running the pilot test. Unfortunately full test coverage was not achieved during the pilot test and some errors remained undiscovered until the test. This could have been mediated with additional pilot testing or unit testing to verify that changes did not break the scenes.

5 | Concurrency in C#, F# and Unity

In the previous chapter we uncovered that even though experienced gameplay programmers wrote more concise code in F#, they were reluctant to switch. In order to introduce functional gameplay programming in the industry, we therefore need a stronger incentive. One such incentive could be implicit (or at least simpler) concurrency. In [26] the author argues that the lenient evaluation strategy, which was also proposed by Sweeney, may provide high implicit parallelisability. This evaluation strategy may be implemented in the FRP system that we prototyped during the usability test.

In this section we research how the lenient evaluation strategy compares to classic concurrency strategies. During this research we found that there is a certain threshold of task-sizes, which must be exceeded before concurrency has a positive effect on execution time. We attempt to estimate this threshold in Section 5.2 using two tests; a busy-wait delay estimation and matrix summation.

In the final section of this chapter we measure the performance of the FRP system to ascertain whether or not the use of functional programming in Unity results in a performance penalty.

5.1 Benchmarks

In this section we explore the performance of the lenient evaluation strategy (see Section 2.1) and how well it parallelises. The reason for our study is that interest in the lenient evaluation strategy has been lacking to say the least. We could not find a reason as to why and decided to explore whether it was because [26] gave false promises of the high implicit parallelisability of leniently evaluated programs.

5.1.1 Test cases

[26] presents two test cases that we reuse in this experiment:

Binary Tree Sum where a tree-walker sums the values of all leaves of a binary

tree.

Binary Tree Accumulation and Sort where a tree-walker flattens all values of a tree's leaves into a list and sorts it.

The difference between the two test cases is that there are no data-dependencies in the summation test, i.e. we can calculate the results of the left subtree and the right subtree in parallel. In the Binary Tree Accumulation and Sort benchmark the tree must be traversed in a left-to-right order and thus there is a data-dependency between the results from the left and right subtree.

5.1.2 Implementations

We implemented the benchmark suite in both F# and C#. In this test F# will be the primary test subject, as we wish to examine how well it parallelises. The C# implementation will be used as control, in the sense that we will compare, for the different strategies, which of the languages is fastest. We are well aware that F# is not lenient, but we wanted to compare the strategies within the same language runtime and therefore attempted to map the lenient evaluation strategy onto F# using its async workflows. Listing 24 gives a code example how to do so (a similar mapping could also be made to .NET's `Tasks`).

```
1  //Lenient function, body evaluated asynchronously from caller
2  //parameters evaluated asynchronously from function body
3  let CalculateC a b =
4      let c = a * 50 //Implicit synchronisation with a-thread
5      c + b;   //Implicit synchronisation with b-thread
6
7  //F# lenient mapping of the CalculateC-function
8  //async function that runs asynchronously from the caller
9  let CalculateCLenient a b = async {
10     let! aResult = a
11     let c = aResult * 50
12     let! bResult = b
13     return c + b
14 }
```

Listing 24: Lenient evaluation mapping using F# Async Workflows.

The mapping shown in Listing 24 involves wrapping all arguments to methods in Async Workflows and explicitly synchronise using the `let!` keyword whenever there is a data-dependency between them. This means that all values of arguments are calculated asynchronously from the method body. The method is marked as `async`, meaning that an Async Workflow is spawned for every invocation of the

method, i.e. it runs asynchronously from the caller. This leaves the majority of the footwork to F#'s scheduler, which must figure out in which order to execute them.

We implemented the test cases in four different synchronization models:

Sequential which uses divide and conquer on one unit of execution to do all the calculations. This provides the baseline speed for the computation on a single core.

Async Workflows which also uses divide and conquer, but in each recursion step, two Async Workflows are spawned to compute the results of the left and right subtree. The Workflows are then synchronised at the end of each recursive call.

Task which uses C#'s `Task`-library and divide and conquer. It is similar to the previous, except that it spawns `Tasks`.

Lenient using the mapping displayed in Listing 24, i.e. wrapping everything in Async Workflows and have the .NET Core task scheduler figure out the computation order.

5.1.3 Test Setup

According to [50] the most reliable results are obtained when the tests are repeated multiple times and the average and standard deviation calculated for the results. We therefore decided to take the average execution speed over 100 repetitions for varying problem sizes, starting with 1 leaf node and gradually doubling the number up to a total of 65536 nodes.

The tests were run on a laptop, of which the specifications are listed in Table 7.

Processor		
Model	Intel Core i7 4702HQ	
Clock Frequency	2.2	GHz
Max Turbo	3.2	GHz
Physical	4	Cores
Logical[1]	8	Cores
Memory		
Memory Size	16	GiB
Memory Speed	1600	MHz
Memory Type	DDR3L 1600	
Software		
Operating System	Ubuntu 18.04 64bit	
C# runtime	dotnet 2.2.104	

Table 7: System specifications of the test machine.

We wish to analyse the following research questions:

1. Which of the four presented strategies handle increasing sizes of trees best?
2. Does F# have worse performance than C# when running equivalent (concurrent) code?

5.1.4 Results

In this section we analyse and discuss the results based on the two research questions presented in the previous section.

Parallel Strategy and Performance

The results are plotted in Figure 3 and Figure 4 (and listed in Table 11 and Table 12 in Appendix B).

[1]Logical cores are sometimes called threads. However logical cores is used here to avoid confusion with the software concept; threads, which is distinct from hardware threads.

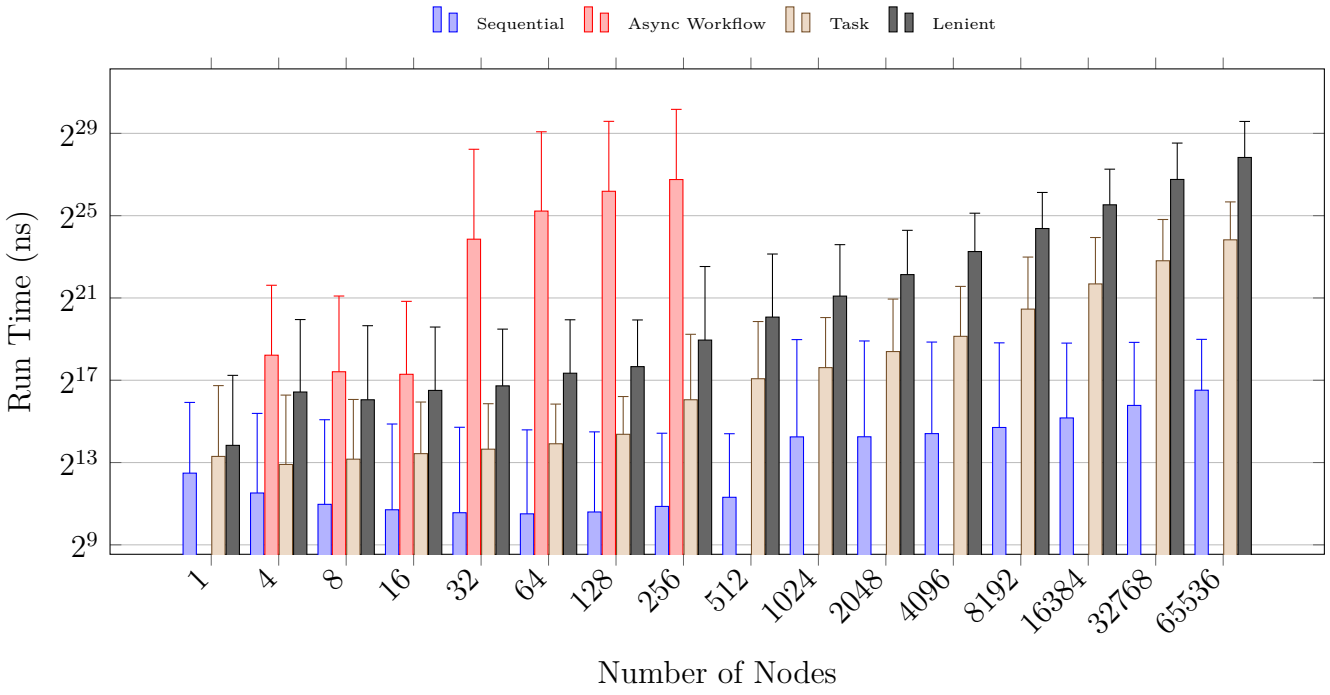


Figure 3: Binary accumulation benchmark results in F# (lower is better).

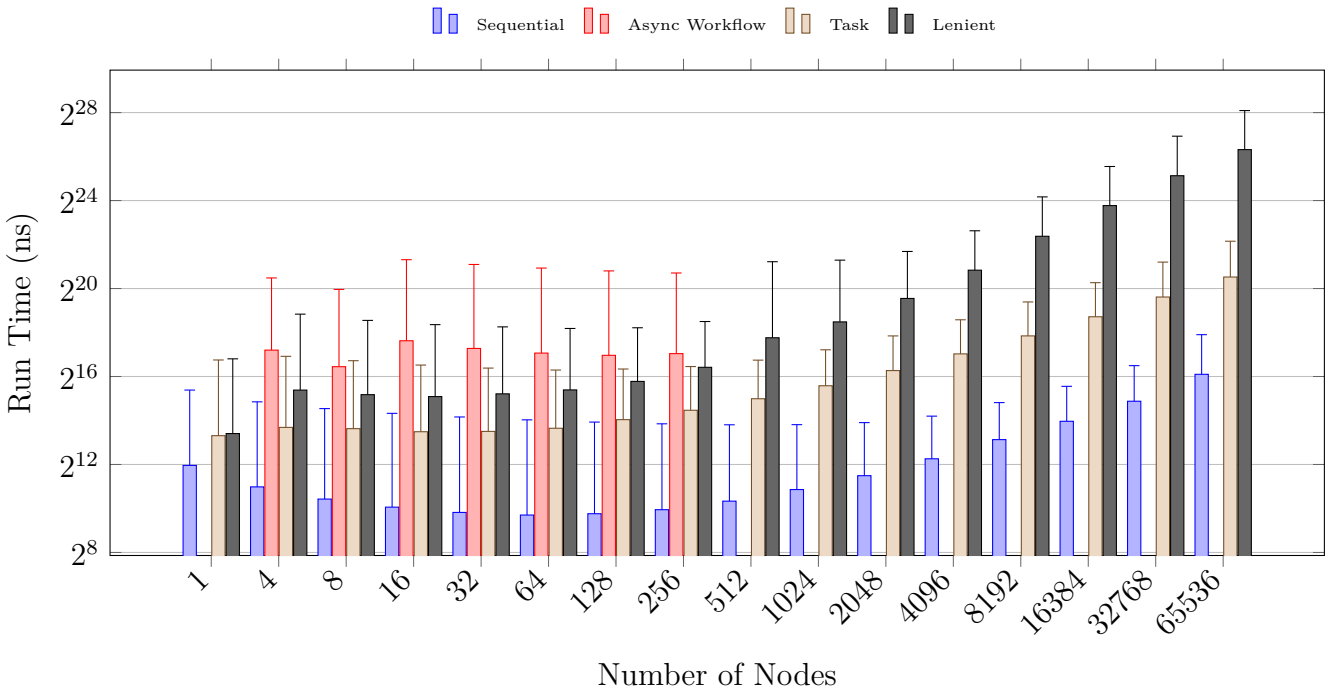


Figure 4: Binary summation benchmark results in F# (lower is better).

Much to our surprise, the sequential implementation was actually the fastest in all cases. It seems that the overhead of spawning and synchronising `Tasks` outweighs the performance gain of concurrency when the problem sizes are in the magnitude

of additions and list appending. Furthermore, the Accumulation test case presented in [26] is a poor choice when it comes to parallelism, as it must traverse the tree in a left-to-right manner, meaning that the only things that can be executed in parallel are the recursive calls down the tree. We suspect that the advantages of parallel programming will be more prominent, as the amount of work in each Async Workflow or `Task` increases. We will research this hypothesis in greater depth in the following section.

The execution times of the lenient strategy grows faster with problem size than those of the task strategy. This means that the lenient approach handle increasing sizes of trees worse. It is, however, not as bad as Async Workflows. We have only obtained data for Async Workflows up to trees containing 256 nodes, as the running times grew so rapidly that it was not feasible to continue. Strangely enough, the lenient strategy scales much better. The only difference between the two implementations are that workflows are started with `Async.StartChild` in the lenient implementation and `Async.Parallel` in the other. These types of results showcase the fragility of parallel programs, where small and seemingly irrelevant details may have a huge impact on the performance.

Language Performance with C# Tasks

Figure 5 and Figure 6 shows the running time of the sequential implementation in C# and F#. These results show that F# is faster in the accumulation benchmark up to tree sizes of roughly 256 nodes. In the summation benchmark C# is faster all the way. Both data sets seem to decrease in running time up to tree sizes of 128 nodes. This is strange, as it means that the implementation handles more computations in less time. We are unsure what causes this.

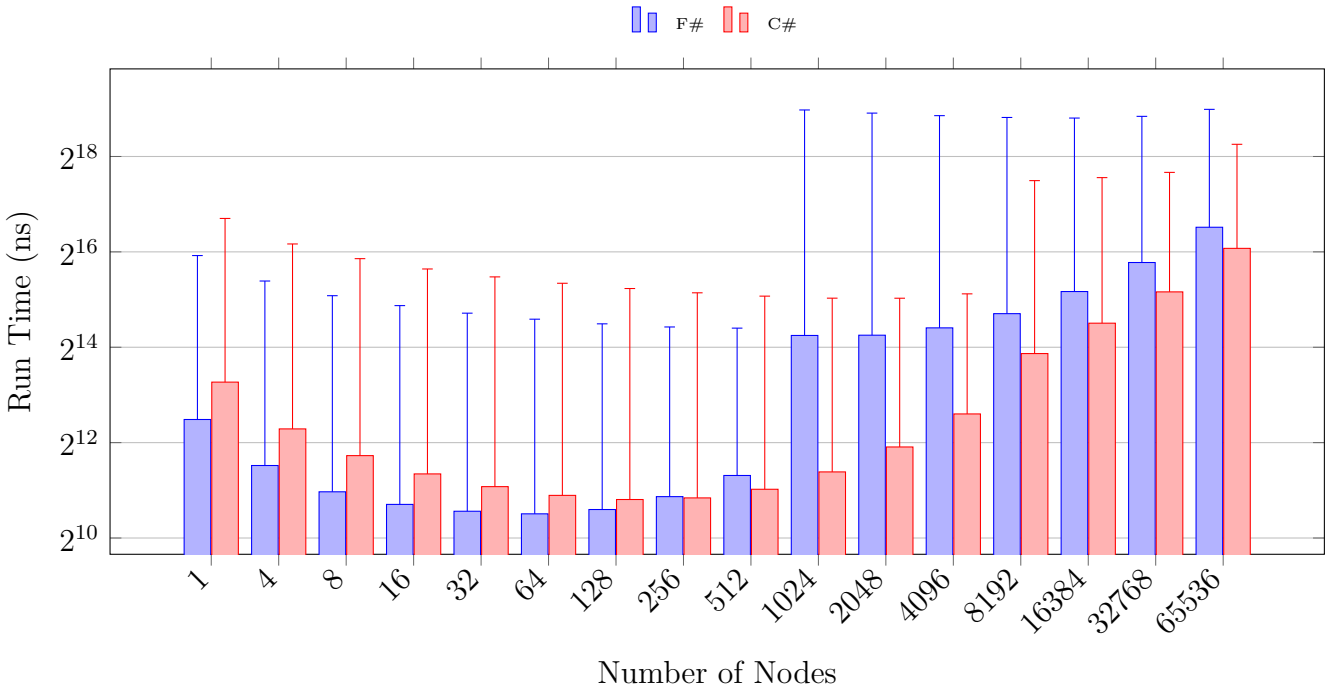


Figure 5: Binary Accumulation in F# and C# using the sequential solutions (lower is better).

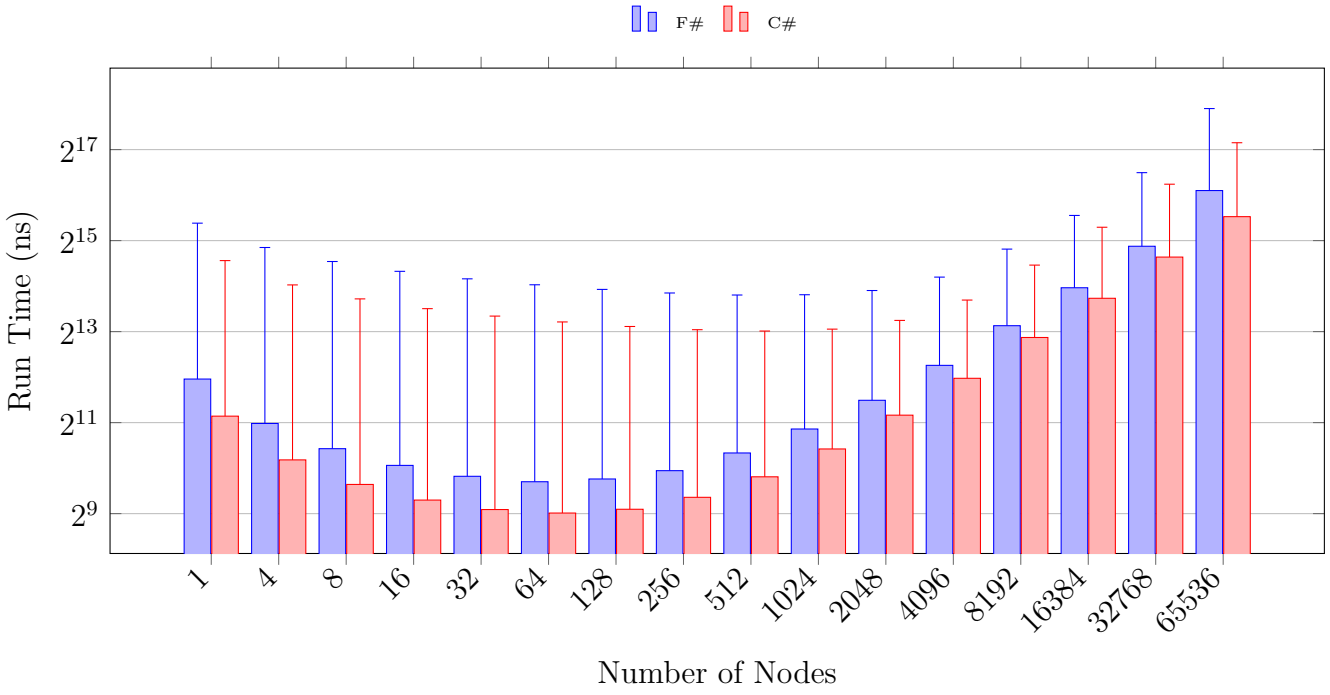


Figure 6: Binary Summation in F# and C# using the sequential solutions (lower is better).

In Figure 7 and Figure 8 we have plotted the running times for F# and C# in the two test cases. Both the C# and F# implementation uses the Task strategy (i.e. divide-and-conquer that spawns two tasks in each recursion step). These results

align with the previous in that F# is fastest when there is a small number of nodes in the tree (up to 128 nodes in accumulation and 8 nodes in summation). After that point C# is faster, and it seems that C# handle increasing number of nodes better. The strange curve we observed in the sequential data sets are also present in C# in the binary summation benchmark. Again, we're unsure what causes this behaviour.

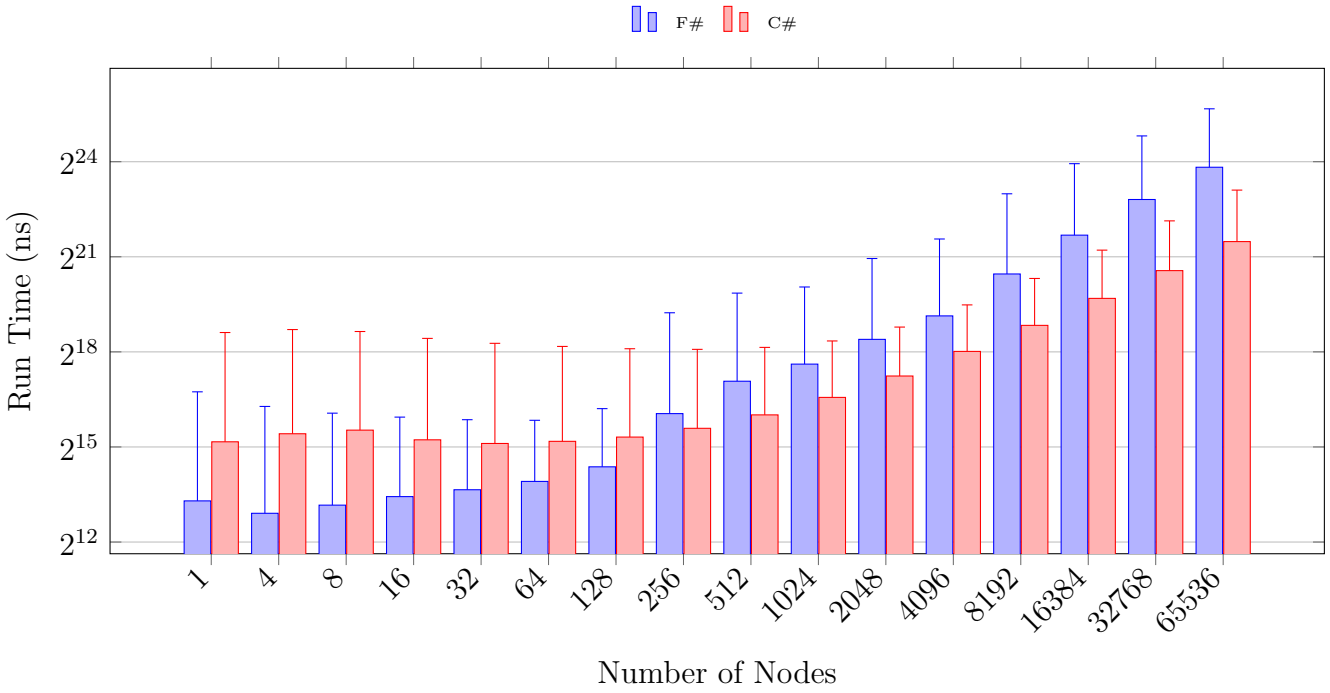


Figure 7: Binary Accumulation in F# and C# using `Task` parallelisation (lower is better).

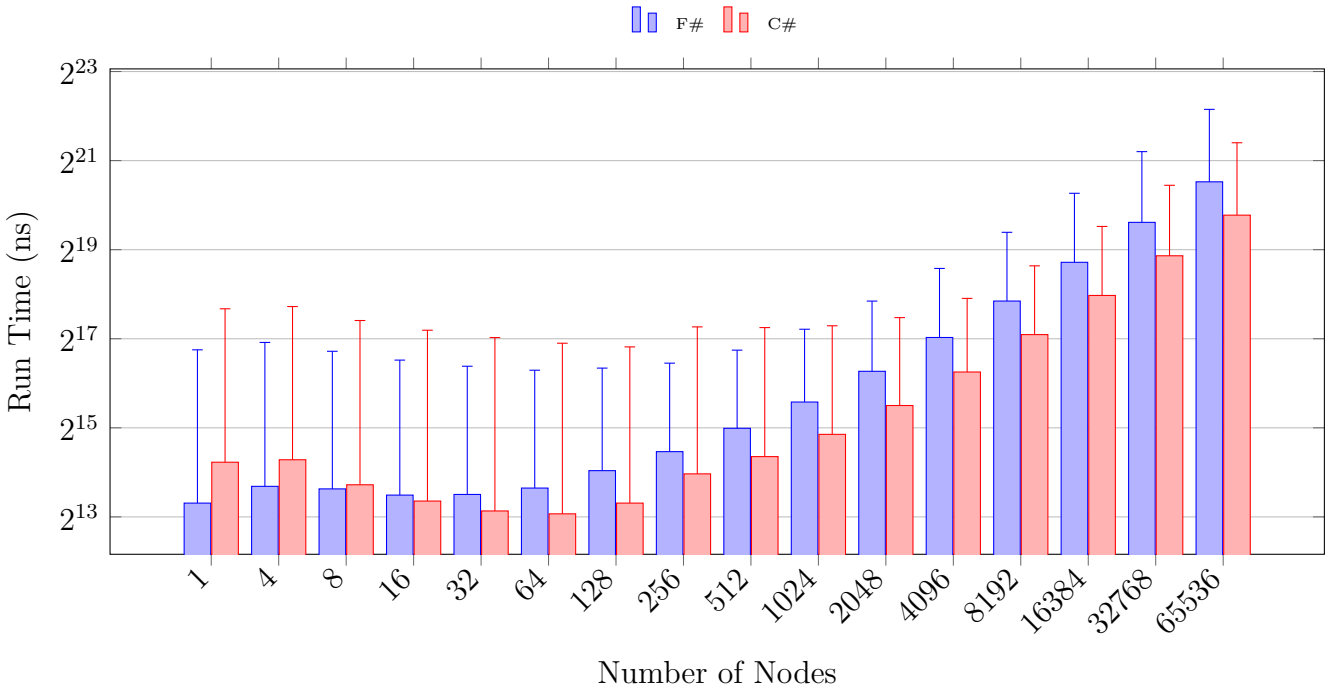


Figure 8: Binary Summation in F# and C# using `Task` parallelisation (lower is better).

5.2 Parallel Overhead & Performance

In this section we present results from an experiment that estimates how much work needs to be done in an Async Workflow or `Task` to outweigh the performance penalty. This is the real-world equivalent of the program's span (see Section 2.1.2 and Section 2.1). We also implement a matrix summation benchmark to determine how different parallelisation strategies handle matrices of increasing sizes.

5.2.1 Estimating Minimum Concurrent Workload

In this experiment we estimate the minimum concurrent workload of .NET's `Task` system. By minimum concurrent workload we mean how much time each task must execute before it is worthwhile to spawn it, compared to a sequential solution. The reason for this exploration is that we found the sequential solution to be faster in the binary tree benchmarks presented in the previous section. In this case we use the C# implementation, as it was the fastest.

Test Setup

We use the binary tree summation benchmark presented in the previous section with a minor modification: Every time the algorithm finds a `Node`, it busy-waits for a given amount of time to simulate work. The busy-wait was implemented with a loop, whose number of iterations is gradually halved until the sequential solution executes faster than the concurrent. The hypothesis is that the concurrent solutions will be faster with higher iteration counts, because it is capable of busy-waiting multiple tasks at the same time.

We implemented the test in two variations, which are listed in Listing 25 (helper methods listed in Listing 26). The first variation emulates a data dependency between the wait and the results from the subtrees, i.e. the wait is intended to emulate a computation that must be carried out after the results of both subtrees have been computed. The other variation emulates a situation where the left and right subtree can be computed in parallel with the wait, i.e. no data dependency between the delay and the subtrees. The tree has a total of 60 leaf nodes. In addition to the binary tree summation, we also implemented a N-ary tree summation in the same variations as that of binary.

```
1  public static async Task<int> SumLeaves(Tree<int> tree, int
   ↪  workBias)
2  {
3      if (tree is Leaf<int> leaf)
4          return leaf.Value;
5
6      var sums = tree.Children.Select(c => SumLeaves(c,
       ↪  workBias)).ToList();
7
8  #if !DELAY_DEPENDS_ON_LR
9      var workBias = Task.Run(() => DoFakeWork(workBias));
10     await Task.WhenAll(sums);
11     var fakeSum = await workBias;
12 #else
13     await Task.WhenAll(sums);
14     var fakeSum = DoFakeWork(workBias);
15 #endif
16
17     return sums.Sum(t => t.Result) + fakeSum;
18 }
```

Listing 25: Implementation of the two different data dependency strategies with an N-ary tree. The strategy may be selected by either defining or undefining the `DELAY_DEPENDS_ON_LR` preprocessor flag.

```
public static int DoFakeWork(int workBias) {
    //Start the 'work bias' before blocking wait on child
    ↪  computation, i.e. we're waiting while the children
    //are computing
    var fakeSum = 0;
    for(var i = 0; i < workBias / 2; i++) {
        fakeSum += i;
    }
    for (var i = 0; i < workBias / 2; i++) {
        fakeSum -= i;
    }
    return fakeSum;
}
```

Listing 26: Implementation of the `DoFakeWork` method, which is used in Listing 25.

Results

The results are plotted in Figure 9 and Figure 10 (and listed in Table 15 and Table 16 in Appendix B.3).

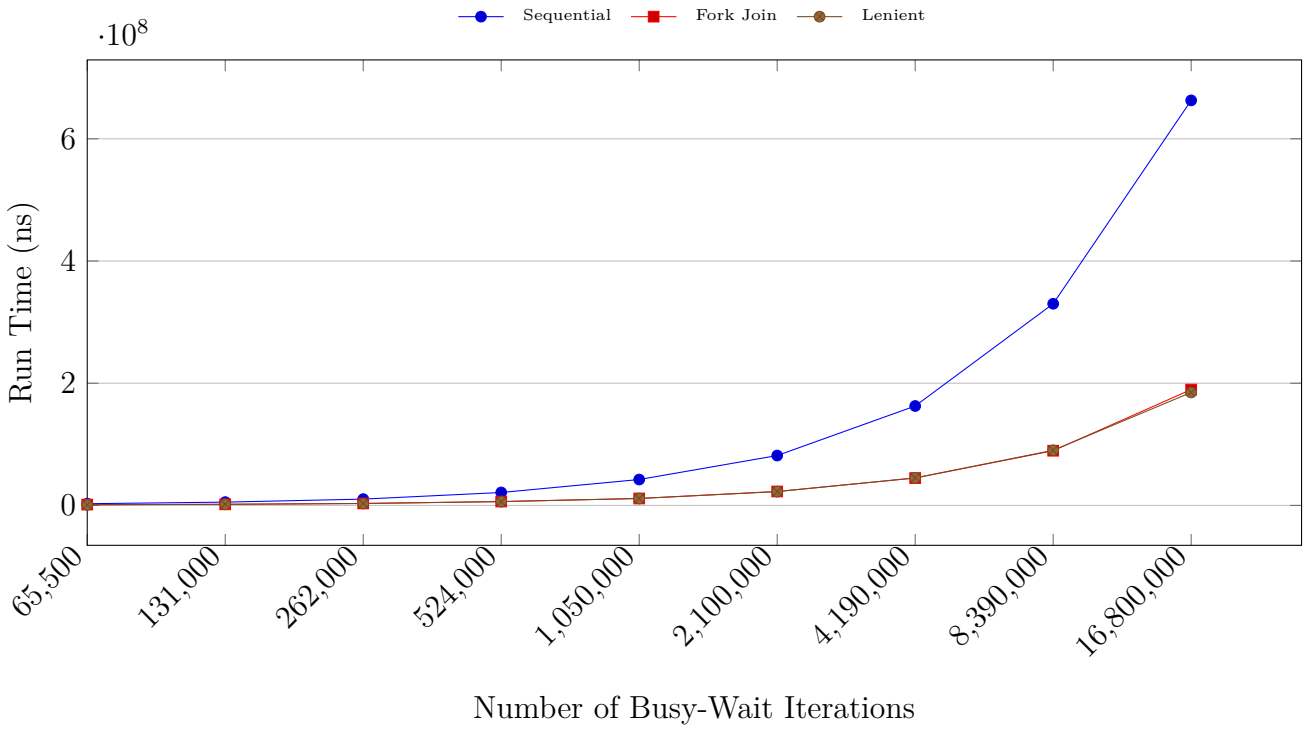


Figure 9: Minimum concurrent workload with data dependency benchmark results (lower is better).

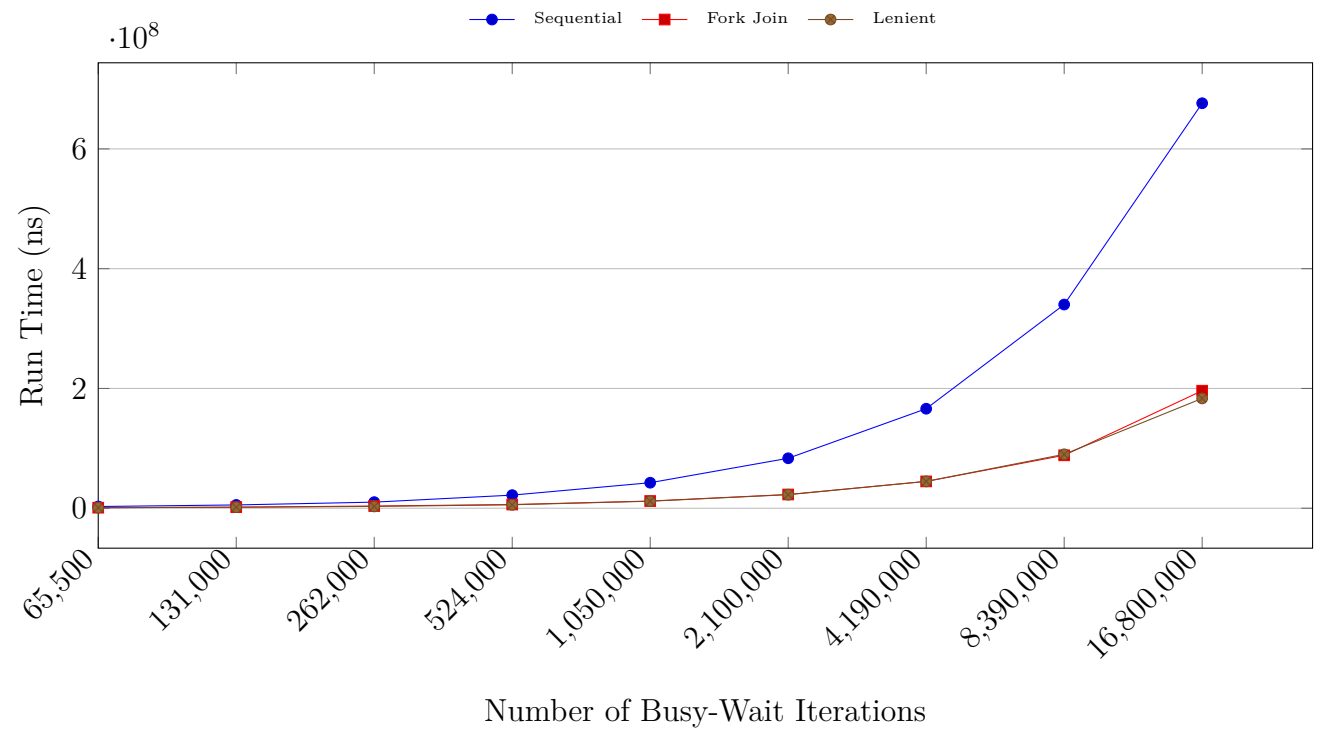


Figure 10: Minimum concurrent workload without data dependency benchmark results (lower is better).

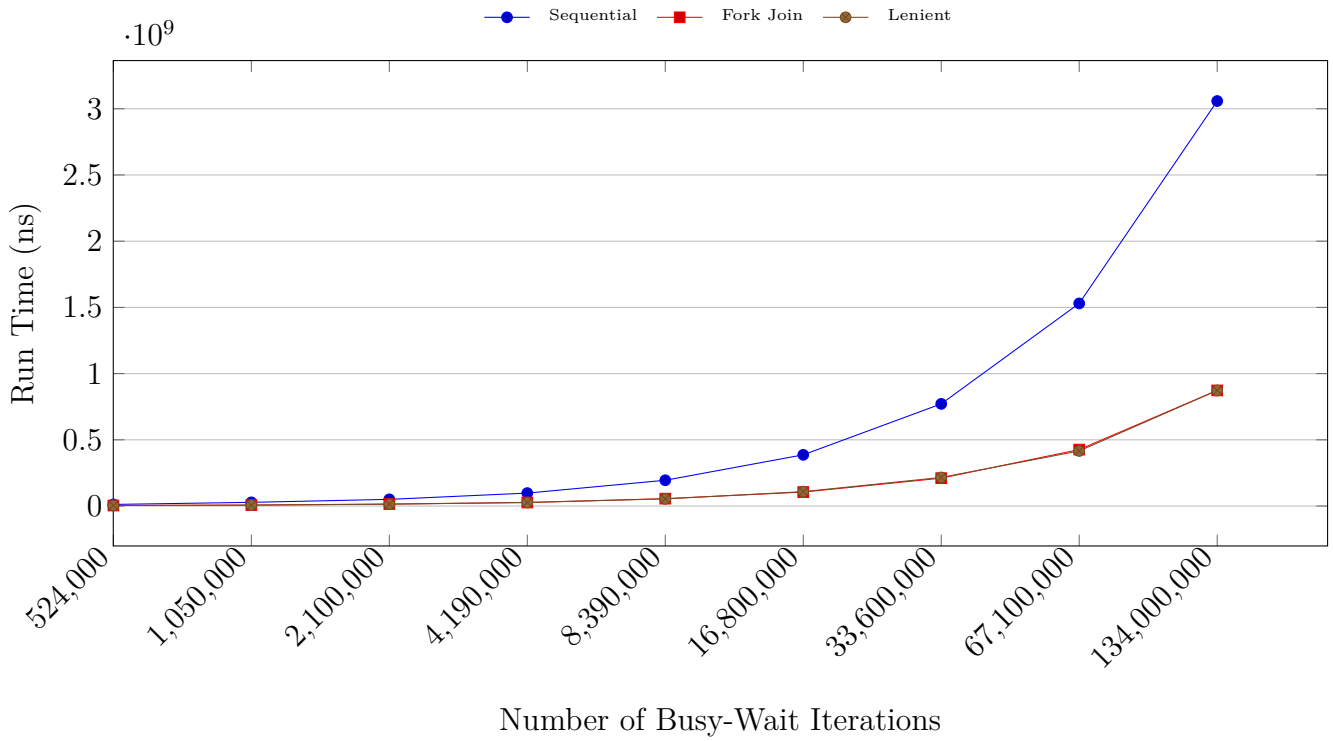


Figure 11: Minimum concurrent workload with data dependency on N-ary tree benchmark results (lower is better).

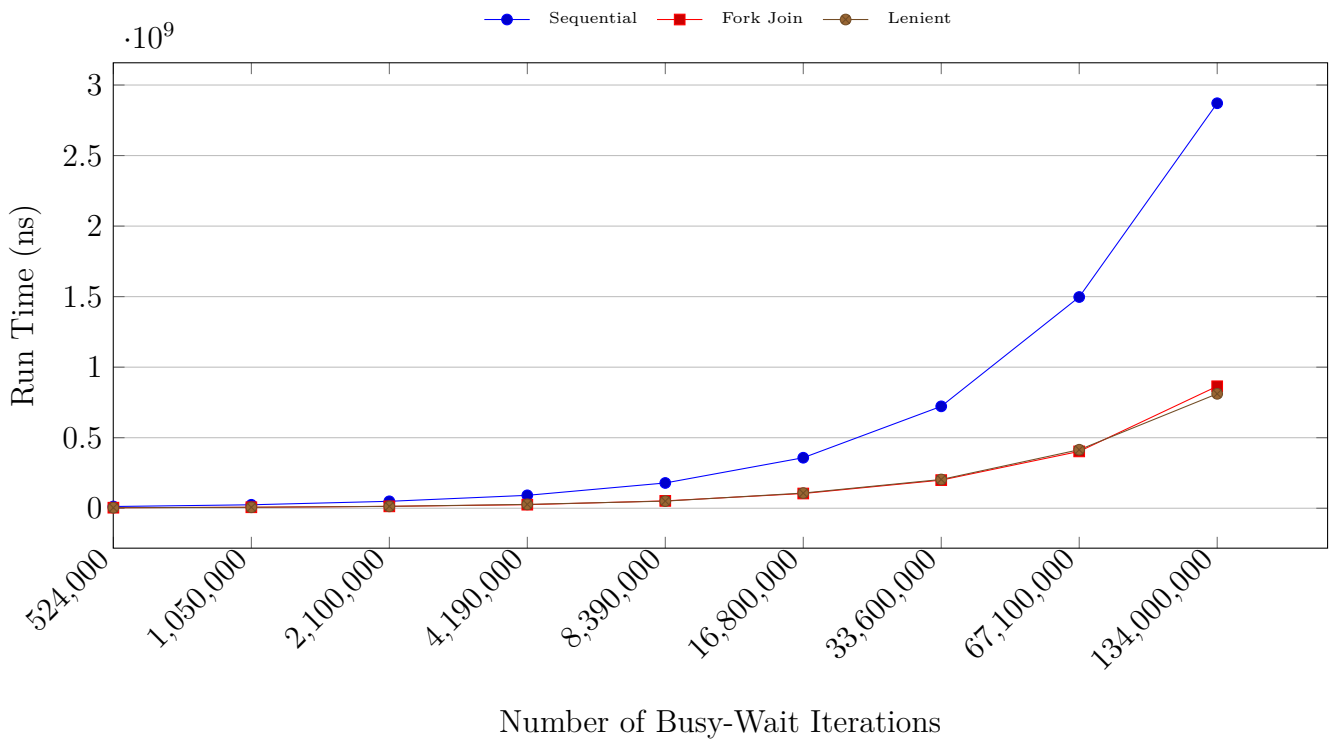


Figure 12: Minimum concurrent workload without data dependency on N-ary tree benchmark results (lower is better).

	No data dependency	Data dependency
Binary	1024	4096
N-ary	2048	2048

Table 8: Iterations of the busy-wait loop before the sequantial solution becomes the fastest.

The number of iterations in the busy-wait loop before the sequential solution is faster than the concurrent is listed in Table 8. These results are in alignment with our hypothesis. The plots show that the parallel solutions grow slower than the sequential solutions because they are capable of executing multiple busy-waits concurrently. Finally, we notice that the lenient and fork join strategy lie very close in execution speed. This is a promising result for the lenient evaluation strategy, as it shows that it may be as fast as a traditional concurrency strategy.

Some concurrency models batch smaller jobs together to form larger jobs. Such batching may reduce the time spent context switching and thus increase the execution speed of the concurrent solutions. Such strategy is employed by Unity's C# job system[68].

5.2.2 Matrix Summation

In this section we execute a matrix summation benchmark. This benchmark measures the time it takes to sum all indices of a random $N \times N$ matrix. This benchmark was implemented in both C# and F# with four different parallelisation strategies to explore how well they scale to increasing sizes of N:

Sequential utilises a double-nested for-loop to iterate over the matrix and sum the values. This benchmark provides a baseline value for running the computation on one thread.

Map-Reduce maps a function that sums each column over the matrix. The resulting list of column sums is then reduced to the overall sum of the matrix. In C# we utilise the LINQ-methods `Select`, `Sum` and `Aggregate` and in F# we use `List.map` and `List.reduce`.

Parallel Foreach uses a parallel loop to iterate over the columns of the matrix that may execute the summation of each column in parallel.

Tasks is similar to parallel foreach, with the only exception that we manually spawn a `Task` that calculates the sum of each column.

We have not included a lenient variation in this experiment, as a lenient implementation would be largely equivalent to a `Task`-mapping of the lenient evaluation strategy. The most notable difference being that a lenient-evaluation strategy would most likely also construct the matrix in parallel with the summation. As the time it takes to construct a matrix is not included in the results here, this should have no effect on the validity of the results.

As the matrices are of size $N \times N$, they contain a total of N^2 elements, which are initialised with random values between `Int64.Minvalue` and `Int64.MaxValue`. When running the test with large matrices we found that the result would overflow, which throws an exception because C# and F# are managed languages. In order to avoid this, we used the `unchecked`-keyword in C#, which disables bounds-checking on an integral arithmetic operation[99]. [99] states that using `unchecked` *"might improve performance"*, compared to checked integral arithmetic operations. The `unchecked` keyword does not exist in F#, instead we found a StackOverflow post, which stated that implementing a custom operator would be our best choice (`let (+!) (x:int64) (y:int64) = Operators.(+) x y`)[100].

Results

The results are plotted in Figure 13 and Figure 14. The first thing to notice is that Map-Reduce seems to be roughly equal to the sequential in running time.

This could indicate that the `Select`-method of C#'s LINQ, which was used to implement Map-Reduce, does not parallelise its iterations. We will thus treat Map-Reduce as a sequential solution for the rest of this result discussion.

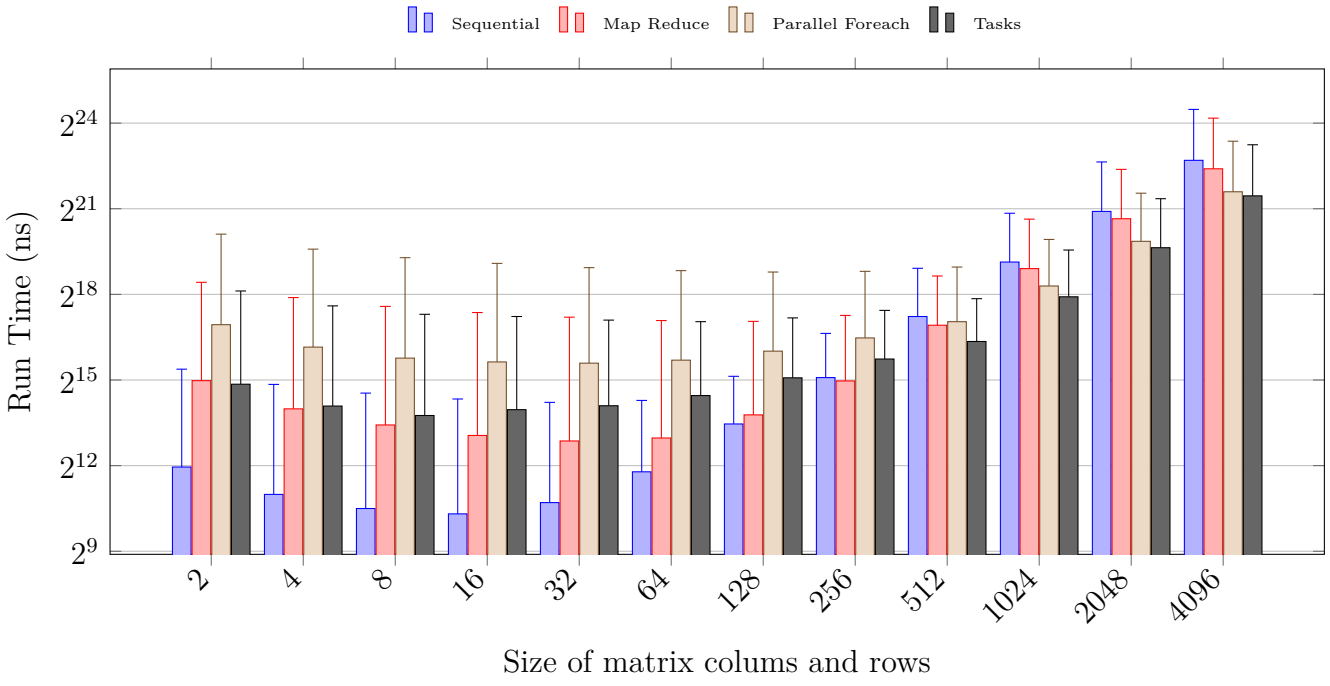


Figure 13: Matrix summation benchmark results in F#.

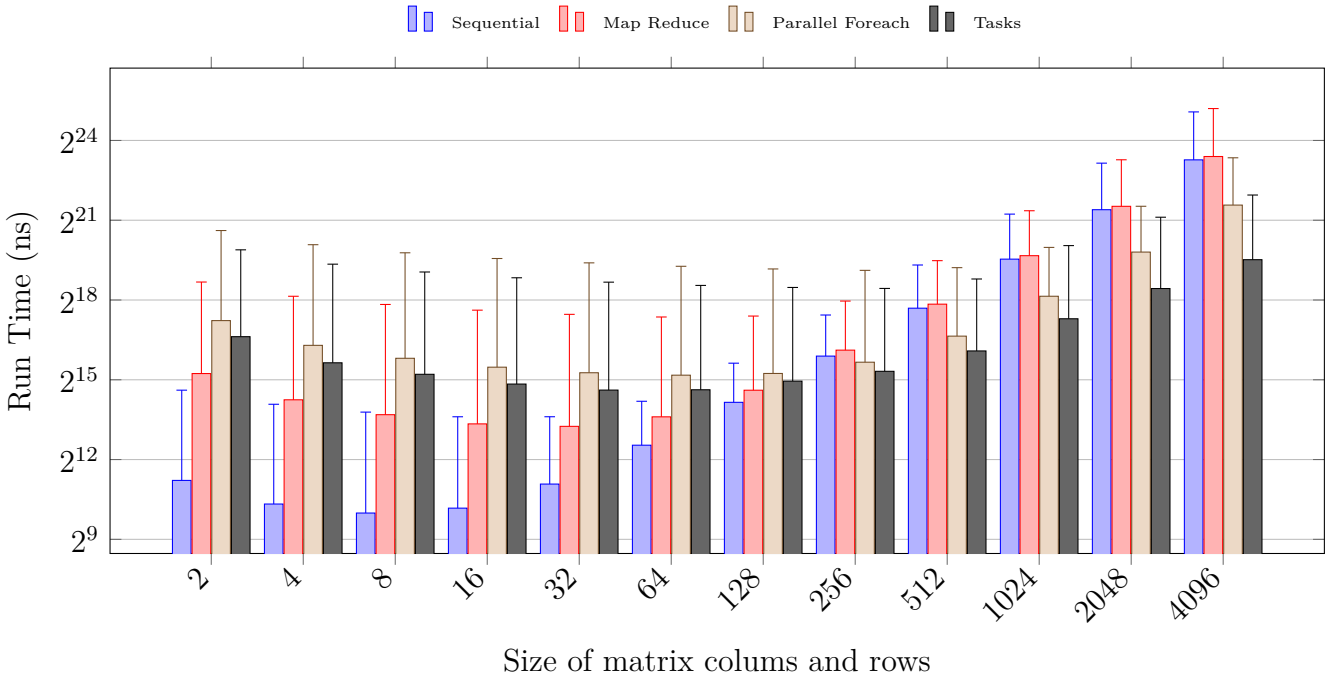


Figure 14: Matrix summation benchmark results in C#.

In general, the results from this experiment is in alignment with those of the previous, in that there is an initial overhead associated with concurrency. In this case, it seems the sequential and concurrent solutions evens out at job sizes of around 512 summations, after which point the concurrent solutions are faster.

After overcoming the initial overhead, the concurrent solutions handle increasing matrix sizes much better than their sequential counterparts. This is even more notable in Figure 15, which plots the same data as a line and without logarithmic y-axis. As the matrix sizes continue to grow, it may be possible to split the columns in multiple separate tasks, possibly making the concurrent implementations faster yet.

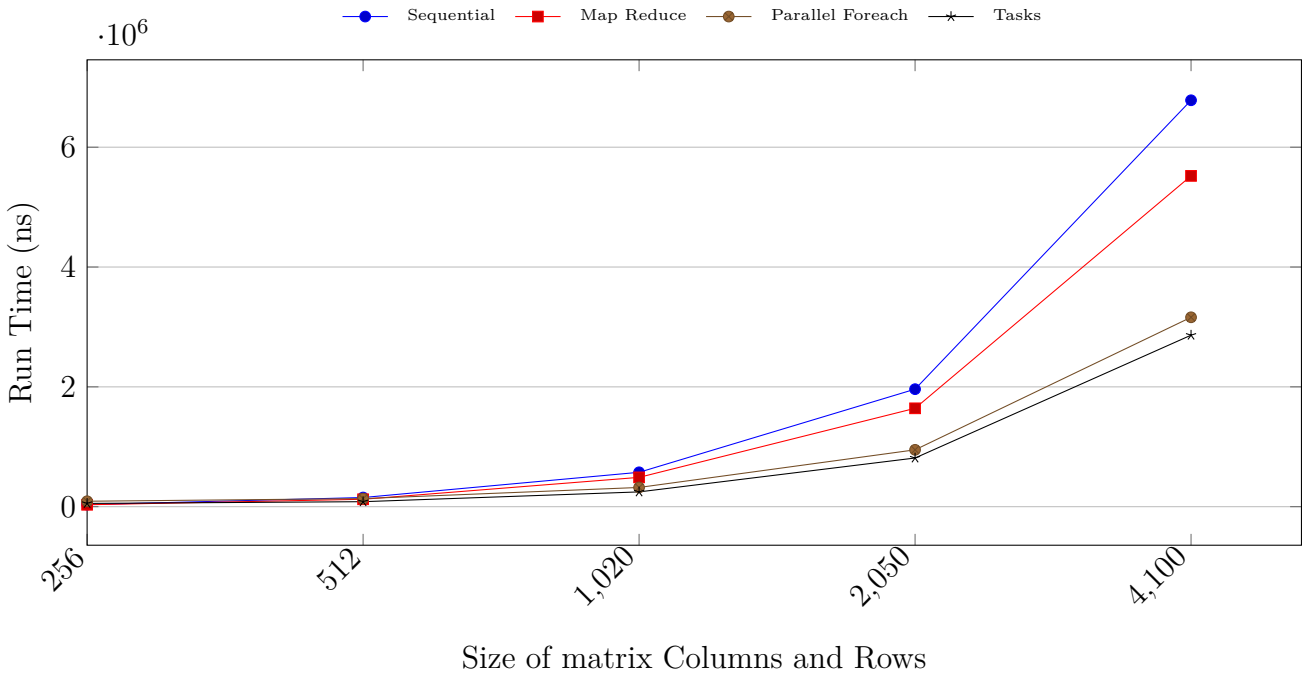


Figure 15: Matrix summation benchmark results in F#, here plotted as a line and without logarithmic y-axis.

In Figure 16 we have plotted the concurrent implementations in C# and F#. This plot shows that the Task implementation in F# is faster with smaller problem sizes. It remains the fastest up until N is around 128. After that point the C# Task implementation becomes the fastest and continues to remain so. It seems to scale better with problem sizes than the other implementations. The parallel foreach implementations lie around the same running time, meaning that the languages are equally performant with that concurrency strategy.

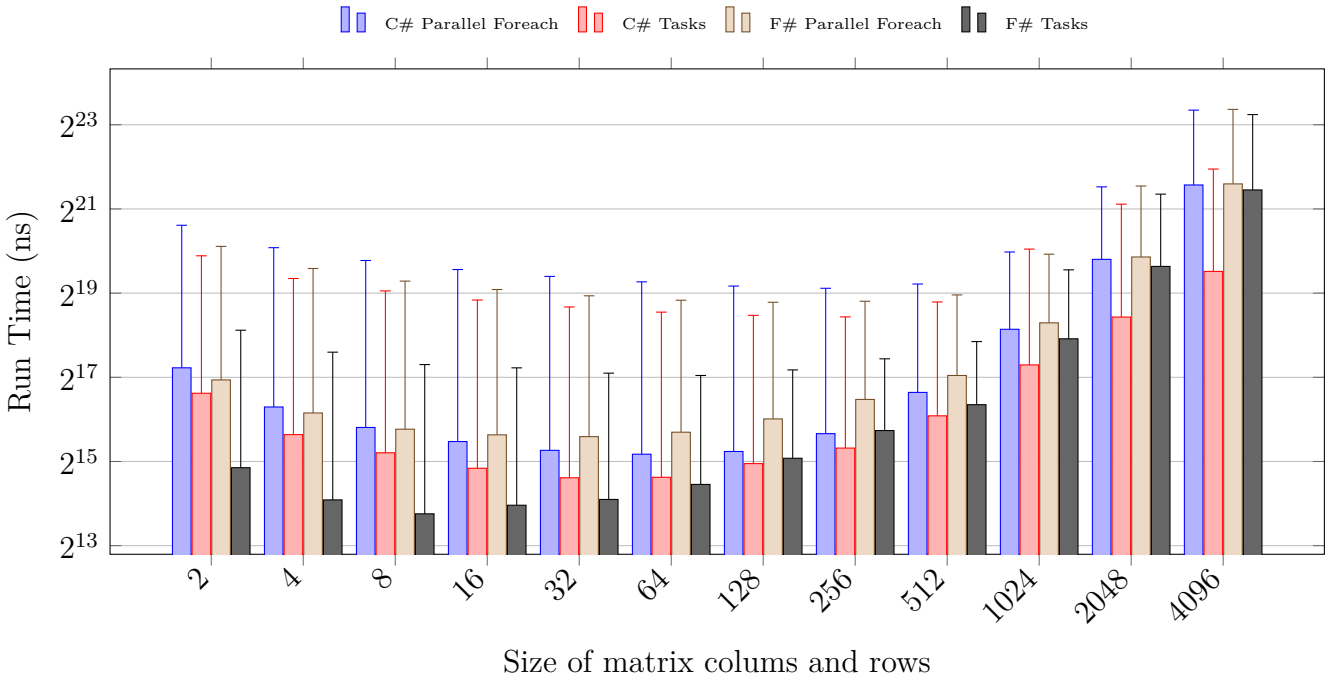


Figure 16: Matrix summation in C# compared to F#.

5.2.3 Discussion

Our results show, first and foremost, that there is an initial overhead in concurrent programming. In order to speed up a computation the task sizes must exceed a certain threshold. Our results show that this threshold lies around the size of 1024-4096 iterations[2] in a for-loop that increments a variable, but also that it varies with the type of problem. Furthermore, the results show that the concurrency of lenient evaluation strategy in these cases is similar to that of classic concurrency strategies, but that the choice of problems in [26] is not well suited to parallelisation. Finally, C#'s `Task`-model seems to handle the matrix summation problem well, after exceeding the threshold of roughly 512 summations in each column.

As part of the project we implemented a simple FRP system in Unity, because this particular programming paradigm is well suited to gameplay programming in functional languages[46], [47], [51]. We had initially decided that this experiment would use either lenient evaluation or Async Workflows under the hood to update `MonoBehaviours` concurrently, but discovered too late that Unity uses a custom concurrency strategy called Unity C# Job System[68] (see Section 3.3). Unity therefore does not allow `MonoBehaviour` updates from `Tasks` and Async Workflows[70]. The resulting FRP system therefore runs sequentially.

[2] According to `sharplab.io` each iteration results in 11 Intermediate Language (IL) instructions, so between 11264 and 45056 IL instructions. The exact number of instructions is unknown, as the likely optimises the code during run time.

5.3 Performance Benchmarking the FRP System

In this chapter we examine the performance of F# and our FRP system in Unity. We first examine the aspect of Garbage Collection/Garbage Collector (GC) by looking at Unity best-practice guidelines, which suggests that garbage is to be avoided to the extent that it's possible. We were not aware of this when we implemented reference solutions to the eight usability test cases that we presented in Section 4.2.1. We therefore adapt one of the solutions to conform to Unity best-practice guidelines and benchmark that against a more "careless" implementation.

5.3.1 Unity Garbage Collection

In this section we first examine best practices for developing applications in Unity with a particular focus on garbage. We then list different GC algorithms, briefly characterise them and investigate which algorithms are used in Mono, dotnet and Unity. Finally we measure the running times of F# against C# in Unity and a functional map-based approach against an imperative one.

Best Practices

Unity recommends careful memory management when writing in C# and avoiding unnecessary heap allocations[101]. The performance optimisation guideline in [101] lists many common performance bottlenecks for Unity developers. The most notable of those are lack of caching and extensive use of boxing. Unity provides many methods and properties that allow developers to access collections of components, such as the `GameObject.FindObjectsWithTag` method and `Mesh.vertices` property[101], [102]. The implementation of those methods will allocate a new array for the objects behind the scenes every time they're invocated. We list an example of this from [102] as `Wrong` in Listing 27. In the example `mesh.veritices` might seem like an innocent property access, but each time the property is accessed, a new array is allocated. This means that the code allocates four new arrays in every iteration of the loop. This puts a huge burden on the GC and will, according to [102], result in noticeable performance degradation. Instead, developers should use the code listed as `Correct` in Listing 27, which does the exact same, but only allocates one array for all iterations, due to better use of caching.

The problems highlighted in Listing 27 are an instance of common subexpression elimination, and one could speculate whether or not Unity's C# compiler should be capable of performing such optimisations. Nevertheless, Unity's best practice

```
//sample implementation of mesh.vertices
class Mesh {
    public Vector3[] vertices {
        get {
            var verts = new Vector3[/*number of vertices*/]
            //find the vertices and put them into verts
            return verts;
        }
    }
}

//Wrong
for(var i = 0; i < mesh.vertices.Length; i++)
{
    float x, y, z;

    x = mesh.vertices[i].x;
    y = mesh.vertices[i].y;
    z = mesh.vertices[i].z;

    DoSomething(x, y, z);
}

//Correct
var verts = mesh.vertices;
for (var i = 0; i < verts.Length; i++) {
    DoSomething(verts[i].x, verts[i].y, verts[i].z);
}
```

Listing 27: Common performance bottleneck in Unity [102]. `mesh.vertices` should be cached. Example is taken from [102].

guidelines list them as an example and explain how developers should transform their code manually[102].

The problem of boxing occurs when a value-type should be used by reference, for instance when constructing a list of integers or appending a float to a string. This generates a small amount of garbage, which can quickly accumulate, e.g. during list iterations. Furthermore, [101] underlines the importance of avoiding LINQ-statements all together, due to the garbage generated under the hood. [102] recommends avoiding coding styles that requires passing functions as arguments and to completely avoid closures, due to the amount of garbage generated by said language constructs. This is in conflict with many functional idioms, which we will explore later.

Garbage Collection Algorithm

Unity uses the Boehm–Demers–Weiser GC, which is a conservative mark-sweep GC[102], originally created for automatic memory management in C and C++[103]. Mark-sweep algorithms are the simplest type of GCs and have the primary disadvantages that they halt computation while running, increase in execution time as more objects are allocated and may fragment memory[104].

The dotnet runtime uses a generational GC with three generations for smaller objects and a single generation for large objects[105]. The younger generations are collected more often than the older and all surviving objects are moved to the older generations. Each time an older generation is collected, all younger generations are also collected. Generational GCs have the advantage that short-lived object allocations have a smaller performance penalty, but the disadvantage that they introduce additional overhead if old objects contain references to young objects[104]. Depending on the system the dotnet runtime may use different GC strategies, including concurrent versions[105]. Concurrent GCs can collect garbage concurrently with the computation, meaning that GC pauses are minimised or entirely removed[105].

Mono has previously used the Boehm–Demers–Weiser GC, but has since moved to a concurrent, generational GC called sgen[106]. We have previously mentioned that Unity uses the Mono runtime, which may cause some confusion, so a clarification is in order. Unity supports two different runtimes: Mono and IL2CPP. Unity's Mono runtime is a fork of the official Mono runtime[107], meaning that updates to the official Mono are not necessarily applied to Unity's Mono runtime. The IL2CPP runtime Ahead-of-Time (AoT) compiles code in IL to C++, which also uses the Boehm–Demers–Weiser GC[108]. However, as part of Unity's 2019.1.0 release an experimental *"incremental garbage collector, which should reduce stutters and time spikes"* was added[109].

Functional Programming and Garbage Collection

All these recommendations stand in direct contrast to the common practices employed in the functional programming paradigm. In functional programming it's typical to map over collections, which has two problems compared to this Unity performance guideline:

1. map allocates a new collection instead of mutating the existing collection.
2. map requires a function as one of the arguments, which defines what should happen to each of the elements in the collection.

This practice also extends to other generalised constructs, such as the tree-walker[110].

These guidelines explain why Unity Technologies does not want to add F# support, despite over 3500 votes on their feedback forums in April 2018[111]. The vote was later closed by Unity, without any explanation[3].

Investigating Performance

GC is not the only thing that may affect performance in a managed language. There is also the problem of calling from the native (or unmanaged) code to the managed code. An investigation of Unity's integration with the managed runtime shows that a there is a considerable overhead in calling the pre-defined `MonoBehaviour`-methods (such as `Update`) in Unity 5.2.2[112]. In [112] the author sets up two different scenes:

1. A scene containing 10,000 separate `MonoBehaviours` with an `Update`-method that increments a variable.
2. A scene containing one `MonoBehaviour`, which contains an array of 10,000 objects. Each time the `Update`-method is called, the `MonoBehaviour` iterates through the 10,000 objects and calls a custom `MyUpdate`-method.

On an iPhone 6 the first approach took an average of 5.4ms to update the 10,000 objects, whereas the second took 0.22ms[112]. In the first approach only 0.4% of the time is spent actually executing the `Update`-code, the remaining 99.6% is spent doing sanity checks, iterating `MonoBehaviours` and instrumenting calls from the native code into the runtime[112].

Test Setup

The question then arises if the (potentially) increased overhead from GC can be outweighed by having a single `MonoBehaviour` manage several other behaviours in the same scene. In order to investigate, we reused the implementation of the Unit Management test case from the usability test (see Section 4.2.1). This solution is listed in Listing 28. This test case may be solved by creating a collection of tuples: `(Unit, State)`. The state machine contains a series of unit management methods; one for each state. These methods take a unit as argument and returns a state. At each iteration the corresponding state's methods are mapped over the collection to create a new collection of game objects and their updated state, which is stored for the subsequent update. This approach avoids dealing with the problems of updating the list while iterating and potentially applying two updates to one `GameObject`. The advantage is that a single `MonoBehaviour`

[3]In previous work we have cited the Unity forums to support this claim[12], but as of February 2019 Unity has closed their feedback forums, meaning that this citation is no longer valid.

is in charge of updating all units in the "Realtime Strategy Game" and the disadvantage is that it generates substantially more garbage, as a new collection is allocated at each `Update`. We refer to this method as "Inverse" in the remainder of this section.

The other approach, here referred to as "Normal", creates a `MonoBehaviour` for each unit, which contains it's own state machine. This has the advantage that we can exploit caching and generate less garbage, as suggested by Unity Technologies[101]. It comes at the disadvantage that each unit must have its own `Update`-method, potentially introducing a large overhead[112].

Methodology

We decided to implement the two approaches in both C# and F#. We run the tests in Unity 2019.1.0f2 and unless otherwise stated, the IL2CPP runtime is used. In all test cases we used a `MonoBehaviour` written in C# to measure the time between each `Update`-call, i.e. the time it takes to generate a frame. We decided to run the test in five setups with 500, 1000, 1500, 2000 and 2500 units. For each setup we generated 900 frames, as that corresponds to 15 seconds of gameplay at 60 Frame per Second (FPS). Each measurement was added to a `HashSet`, which was written to a CSV file after the test. This means that the measurements include all game-related code, both including rendering, physics and so alike. However, as this system is ultimately going to be used to develop games, we conclude that delta time (or equivalently FPS) is a sufficient metric, as that is of utmost importance to the player.

The following research questions outlines the intent of the experiment:

- How does the performance penalty from extensive garbage generation compare to the performance penalty from an increased number of calls between unmanaged and managed runtimes?
- Does AoT-compilation in the IL2CPP runtime actually provide a speed up?
- Does the use of F# (and FRP) introduce an additional overhead?
- Unity introduced a new incremental GC in Unity 2019.1. Does this new GC provide a speed-up when using either C# or FRP?

5.3.2 Results

In this section we discuss the results from the benchmarks. We use the questions presented in the previous section as baseline for the discussion.

```
private List<(State state, Unit unit)> _stateList;
public void Update()
{
    //Apply updates and store the updated states in a list
    var newStates = _stateList.Select(s =>
    {
        switch (s.state)
        {
            case State.Fleeing:
                return Flee(s.unit);
            case State.Moving:
                return Move(s.unit);
            case State.Attacking:
                return Attack(s.unit);
            default: return (State.Moving, s.unit);
        }
    }).ToList();

    //zip the list with the old states to create tuples: (new
    ↪ state, old state)
    foreach (var statePair in newStates.Zip(_stateList, (sNew,
    ↪ sOld) => (sNew,sOld)))
    {
        //Compare old state and new state, initialise the unit for
        ↪ the new state if changed
        if (statePair.sNew.state != statePair.sOld.state)
        {
            var unit = statePair.sNew.unit;
            _initialiseState(statePair.sNew.state,
            ↪ statePair.sNew.unit);
            //Create a new list containing the updated unit
            _stateList = _stateList.Select(s => s.unit == unit ?
            ↪ (state, unit) ? s);
        }
    }
}
```

Listing 28: Possible solution for the Unit Management test cases.

Performance Penalty from Extensive Garbage Generation

The results are listed in Table 9 and plotted in Figure 17. The results indicate that F# adds a small overhead, which increases as the number of units grow. This can be seen by comparing C# Normal and F# Normal. Furthermore, the C# Inverse also adds a small overhead compared to C# Normal. This could indicate that Unity has optimised the calls between native and managed code since v5.2.2. We also observe that the inverse FRP state machine performs notably worse than the other approaches as the number of units grow. The reason is that each unit is in fact an entire FRP-system with condition-checking and event-dispatching. We outline in Section 8.2.1 how a (potentially) more performant system could be structured.

Number of Units	C#	C# Inverse	F#	FRP Inverse
500	53.58	54.04	54.35	53.63
1000	50.24	51.33	51.53	41.24
1500	19.15	21.17	18.90	12.34
2000	19.09	16.23	11.70	7.73
2500	13.63	10.89	8.43	5.50

Table 9: Average framerate when simulating the given number of units in Unity's Mono runtime.

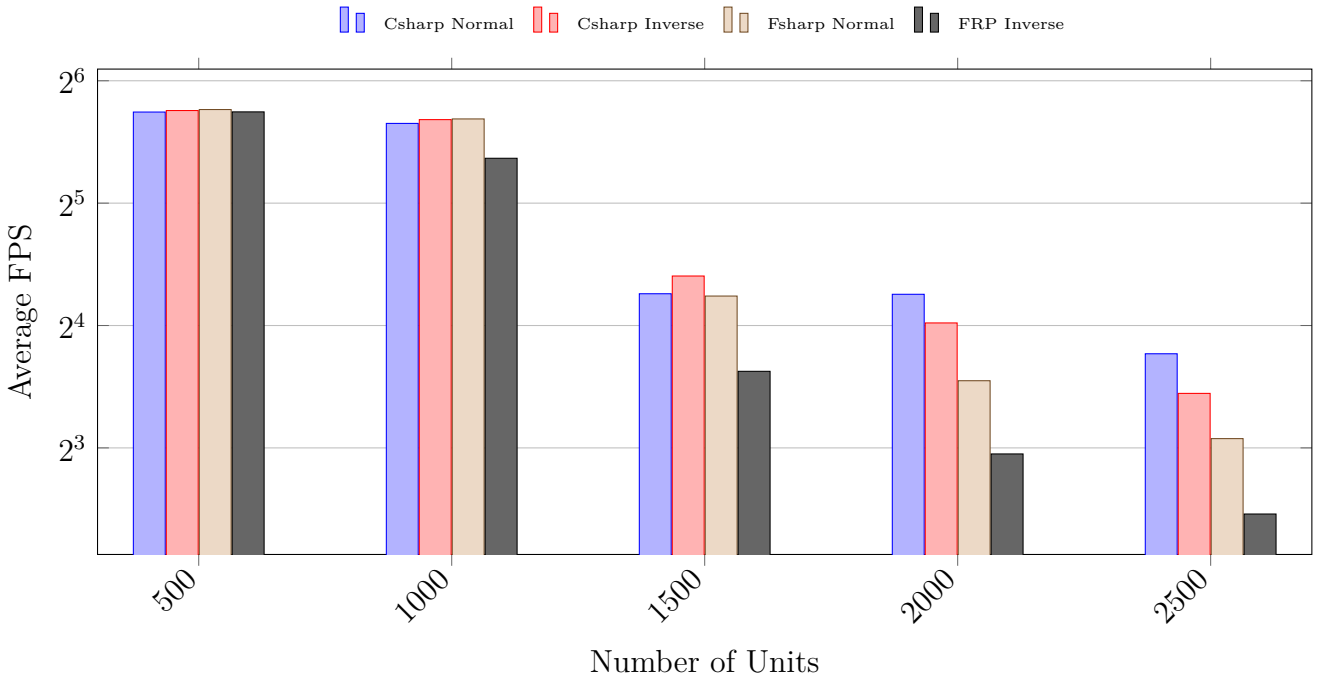


Figure 17: Average FPS in Unit Management benchmark using the Mono runtime (higher is better).

Performance of Runtimes

The results, listed in Table 10 and plotted in Figure 18, show that IL2CPP does not necessarily provide a speed up. We deem this as C# Normal is faster in Mono, whereas IL2CPP provides roughly two more FPS in C# Inverse and F# Normal.

Another interesting observation we made during the test is that there is a very large spike in the time it takes to generate the third frame. This spike is around 20 times the time it takes to generate the other frames. One could explain this spike in Mono as runtime-optimisation, but as it is also present in IL2CPP, which is AoT, that cannot be the case. We do therefore not know what causes the spike.

Runtime	IL2CPP	Mono
Csharp Normal	19.15	19.82
Csharp Inverse	21.17	18.54
Fsharp Normal	18.90	17.42
FRP Inverse	12.34	12.23

Table 10: Average framerate in Unity's two runtimes measured with 1500 units in the scene.

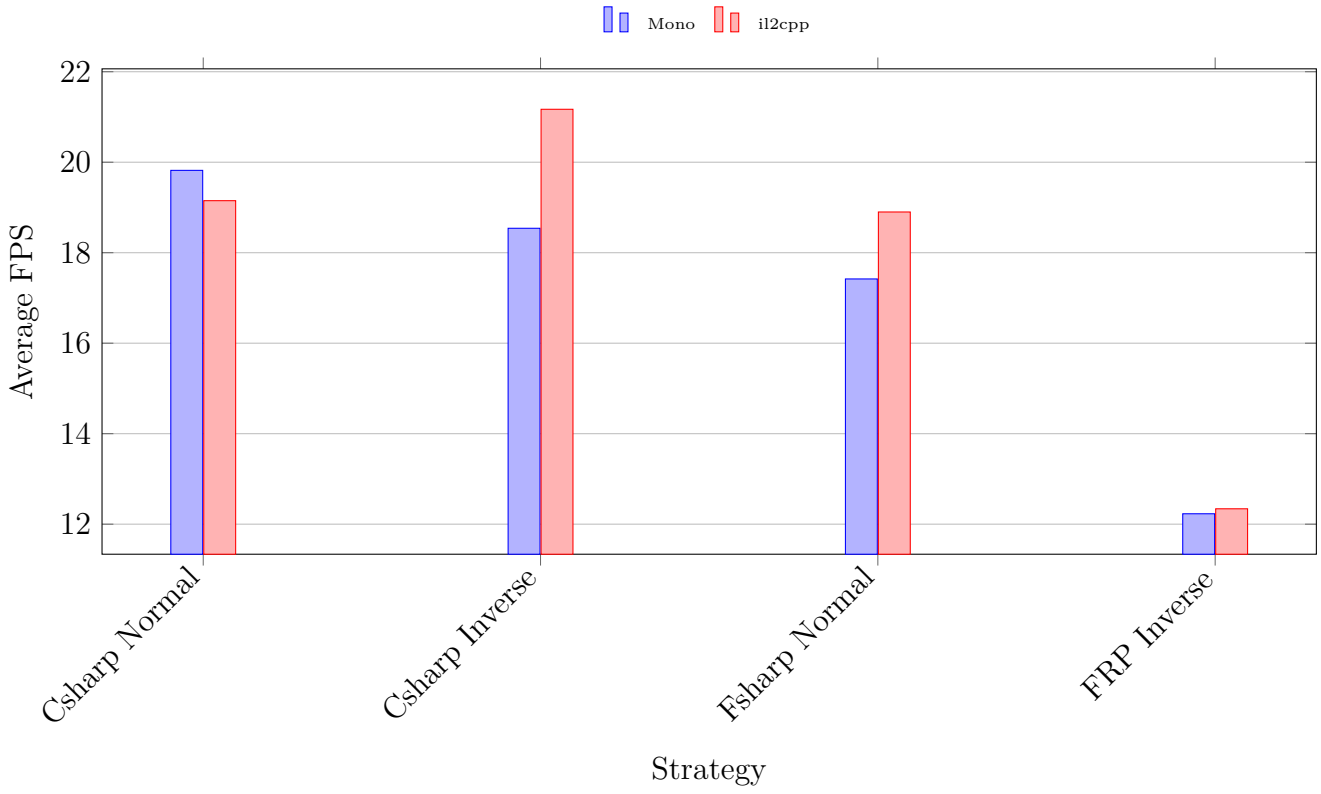


Figure 18: Average FPS of the two different runtimes in the Unit Management benchmark (higher is better).

Performance of the FRP-system

The results are plotted in Figure 19. The results show that FRP introduces additional overhead. This overhead results in a decrease of ten FPS on average over the 900 frames. On the other hand, the FRP-system yields a smoother curve, which means that the game will be subject to less stuttering and fewer lag spikes.

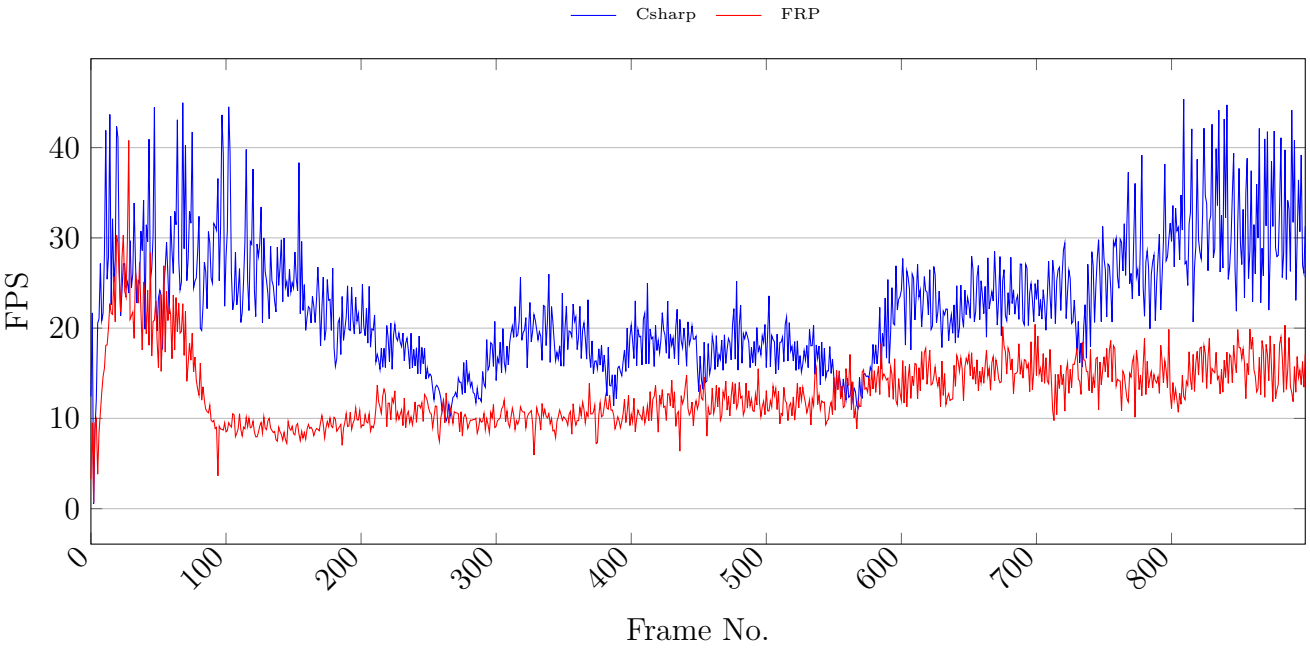


Figure 19: FPS for each frame in FRP and C# Inverse using the Mono runtime (higher is better).

Unity's Incremental Garbage Collector

The results from running the C# Normal implementation with the two different runtimes and GCs are plotted in Figure 20. These results show that the two GCs perform more or less equivalently. It might even seem that the incremental GC performs worse than the original after the framerate stabilises after the 350th frame.

In general, the incremental GC has lower variation, except from Mono after frame 650 where the FPS varies wildly (which can be seen by the high standard deviations in Figure 20).

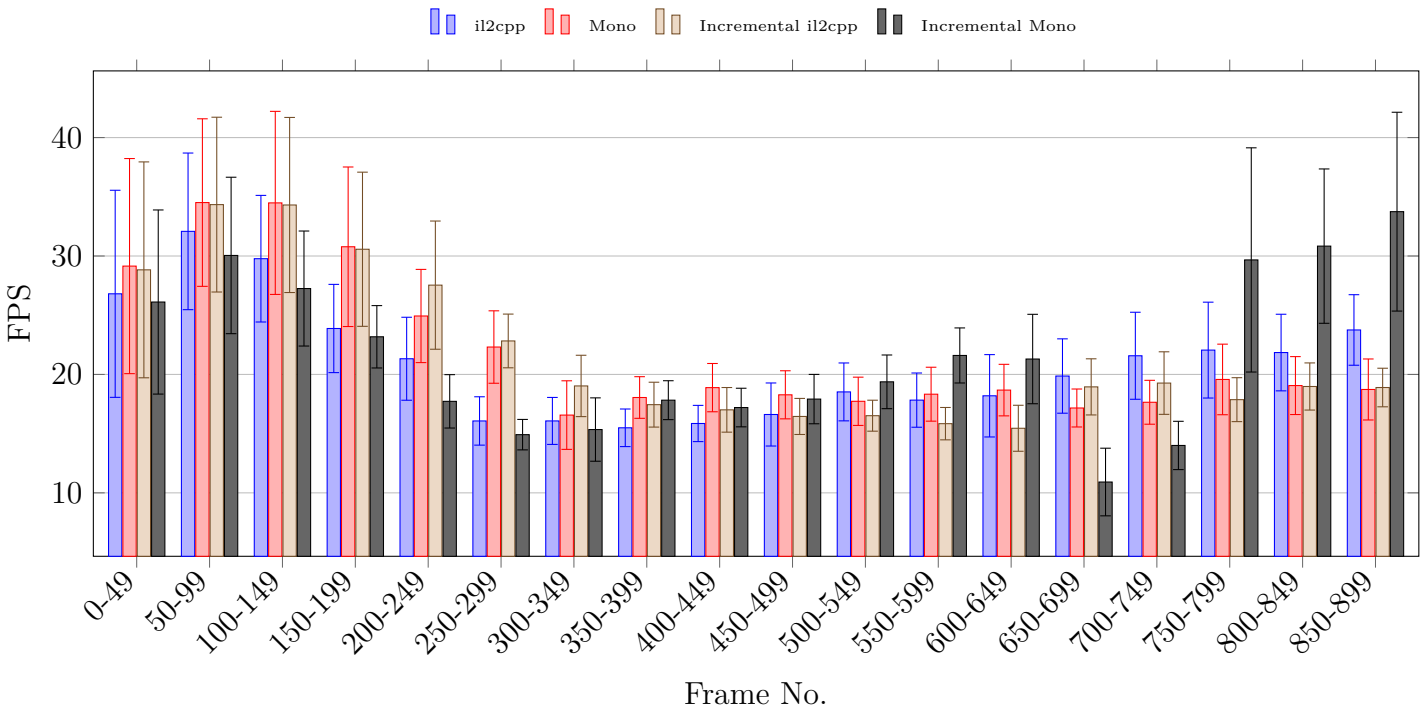


Figure 20: Average FPS over 50 frames in C# using the two different Unity GCs (higher is better).

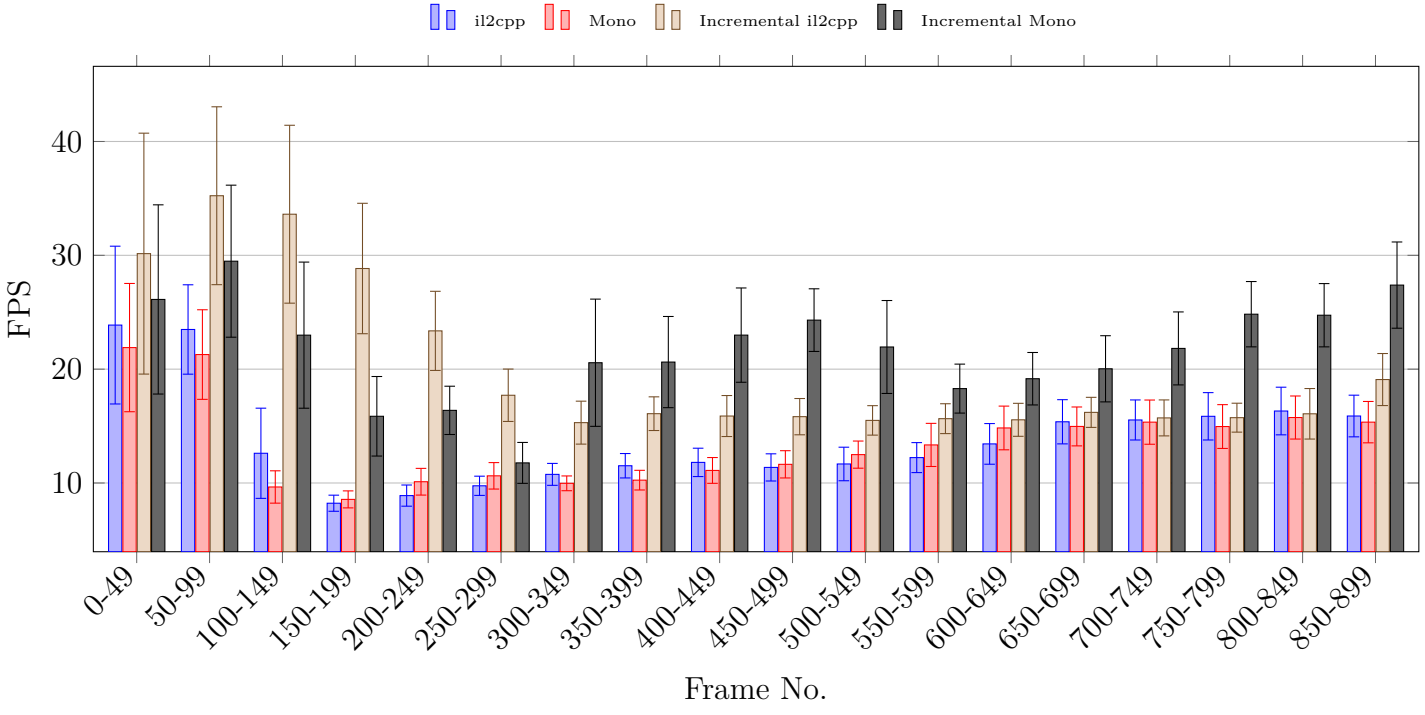


Figure 21: Average FPS over 50 frames in F# using the two different Unity GCs (higher is better).

We also tested the two GCs with the FRP-system we developed in F#. The results

are plotted in Figure 21. These results show that the incremental GC performs slightly better than the original one. However, it comes at the cost of much higher variation in the framerate, especially in the Mono runtime. The IL2CPP incremental curve has large spikes up until around the 200th frame, after which point it stabilises. The Mono curve continues to variate between an average of 10 FPS to over 30 FPS.

5.4 Threats to Validity

In this section we examine threats to validity in the performance benchmarks that were presented in this chapter. Once again we use the methodology proposed by [96]. The points discussed here are in general applicable to both the concurrency benchmarks, the critical work benchmarks and the Unity FRP benchmarks.

5.4.1 Internal Threats

In this section we discuss internal threats. Internal threats may originate from the test cases, execution and data analysis. We both discuss the origin of the threats as well as what we did to counteract it.

Experience

One of the threats to validity is our experience with F#. None of us had written F# before starting this project and thus our experience is in the vicinity of months. We have spent time examining best practices and how to implement concurrent code in F#, but some programming tasks come more easily with doing rather than reading. Furthermore, it is well known that writing concurrent code is notoriously difficult and requires greater experience[65], [67].

Test Cases

In the F# concurrency benchmark we reused the problems presented in [26], as the author claimed that they were well suited for the lenient evaluation strategy. We discovered that it was not the case. In order to research how the lenient evaluation strategy fares when the amount of work grows, we introduced a modified version of the binary summation benchmark, which emulates work in each node. We also introduced a matrix summation test, where each column was summed concurrently. These tests allow only a rather limited peek into how well F# parallelises in comparison to C#, as they are very specialised problems. Furthermore, they do

not provide any information about how the two languages parallelise in the context of a game, as we argue that it is not common to sum large matrices and accumulate leaves of trees in games.

Implementation of Binary Accumulation and Sort

We have made a systematic error in the implementation of the Binary Accumulation and Sort benchmark. [26] first accumulates the leaves of the tree and thereby sorts them. In our implementation we have only accumulated them and not sorted the leaves. This error is present in all implementations, so they are still mutually comparable. This may explain why our results differ from those of the paper.

5.4.2 External Threats

In this section we examine external threats. External threats are those that arise outside the testing environment and may affect how broadly applicable the test results are. We first present the origin on the threat and then outline how the tests could be altered in the future to provide more broadly applicable results.

Limited Generalisability

All the benchmarks were run on a single test machine, as listed in Table 7. This means that the results may have limited generalisability to other system setups. This threat can rather easily be counteracted by running the benchmarks on other computers. The Central Processing Unit (CPU) of the test machine was announced on the 6th of April 2013[113], meaning that it is a little over six years old at the time of writing. It would therefore be beneficial to rerun the benchmarks on a newer processor to obtain more contemporary results.

6 | Discussion

In this chapter we discuss the results of the project. We start with a discussion of the adoption potential of F# and examine the test participants' arguments for and against F#. We then move on to discussing the methodology adopted in this project, namely the use of Champagne Prototyping, Attention Investment Model and Cognitive Dimensions. Finally we discuss the technology choices we made in the project, which consists of using Unity as the host game engine and C# and F# as languages.

6.1 F# Adoption Potential

While the F# code produced during the user test in general was shorter than the C# code (see Appendix E), the participants expressed that they would not personally choose F# in a real project. Even participants who clearly identified the benefits of F#, held this opinion. In this section we will explore the participants' reasons for this standpoint. The participants stated a number of rationales for this during the debriefing interview, which we categorised in four categories; prevalence and popularity of C#, learning overhead and cost, comfort zone and difference of priority. Each category is, in turn, analysed and explained.

6.1.1 Prevalence and Popularity of C#

Several participants stated that C# was simply more prevalent, which means that documentation, forum posts, tools and libraries would be more available. These sources of information are very important because they aid developer productivity, which is of higher importance than performance in game development. This was pointed out by participant 5 and was consistent with what other participants expressed. In the quotes below the relevant sentences have been highlighted with bold text.

> *"If my boss told me to do it, then I would use it, I wouldn't refuse. But I wouldn't use it in my spare time,* ***C# is too well documented and familiar****. It would take a while to get into F# and the functions I am familiar with from C# [...]"*

-Participant 1, Appendix D.1

Participant 1 states that he would not use F#, unless specifically asked to do so. Learning a new language or tool takes too long time, even when the benefits of the tool or language are evident. This is echoed by participant 4:

> *"And I have worked with C# and Unity almost everyday and* ***I am as familiar with C# as is needed in Unity*** *[...]*
> *Alright, so it isn't anything I would ever use, but that is because I have used this [C# in Unity] since 2011.* ***It is so integrated in me that I just do that, that, and that*** *and I am happy that I am at the point where I don't need to think about syntax errors, because it is usually logic errors I get."*

-Participant 4, Appendix D.2

This is inline with the literature, which states (among other things) that developers tend to select languages they feel they know[114]. However, participant 4 also argued that new languages are mainly useful for larger development studios. This is in contradiction with the literature, where evidence points to the opposite. Finally, the participants' agreed that the use of events is highly positive. We argue that the use of events is a corner stone in the domain specificity of the FRP system, echoing scientific research that says that domain specific languages are more likely to be adopted[114].

6.1.2 Learning Overhead & Cost

All participants struggled with F#'s syntax, which was expected. The syntax struggle was also a factor in participants' decision making process. While F# was a new language, it also presented a new paradigm. This paradigm shift was difficult for the participants, but few mentioned the shift directly.

> *"I really like functional, but now that I tried programming for games I am not sure."*

-Participant 6, Appendix D.3

However, most participants did not take note of the paradigmatic changes directly. Instead their focus was on the syntactical differences as well as the new keywords and operators. This may be caused by the their limited experience with F# and that the paradigm shift may become more apparent to the participants.

"It was very difficult to look at a new language again after such a long time."

-Participant 4, Appendix D.2

"However syntax-wise I was quite lost."

-Participant 1, Appendix D.1

All participants, except 6, had very limited experience with the functional paradigm. The five first participants had Medialogy educations, which does not teach functional programming. According to [114] only 15% of computer science students learn functional programming languages if they are not taught during their education. This further increases the learning cost associated with F#, as the participants must acquaint themselves with both a new paradigm and a new language.

Participants were made aware of a potential advantage of F#; the higher parallelisability, which may afford developers greater utilisation of the system resources. However, participant 5 pointed out that "*performance is provided by the engine, not the game*"[1], which explains why none of the participants were particularly interested in F#'s potential for simpler or even implicit parallelisation.

6.1.3 Comfort Zone

As earlier stated, developers tend to select programming languages that they are familiar with[114]. This is reflected by the participants as well, where both participant 1 and 4 noted that they were too used to C# to switch to F#. Furthermore, participant 1 stated that the languages he used tended to be the more popular languages.

"I think in the context of what I would be assigned to do, if someone assigned this to me I would do it, but typically it is the more popular languages you are assigned."

-Participant 1, Appendix D.1

In addition most participants had developed habits from C# that they would have to unlearn or manage to productively write F#. Syntactically the habits consisted of writing semicolons, curly braces and explicitly writing types in front of variables. On the semantic side, the inversion of access modifier defaults, immutability and implicit returns posed the largest challenge.

[1]The footage of the fifth test was lost, therefore any and all quotes from participant 5 are those noted by the test observer during the test.

"Yea, I mean it is only a question of time before you get used to not making semicolons and curly brackets."

-Participant 1, Appendix D.1

"It would help a lot if I had longer time to work with a project."

-Participant 6, Appendix D.3

Participants largely agreed that the habitual issues would disappear with more practice. Given more experience with F# and experience with actual projects and problems, the paradigmatic differences may become more apparent to the participants.

6.1.4 Difference of Priority

Some of the potential benefits offered by F# are not as appealing to the participants as initially believed. Surprisingly, multiple participants expressed that performance was not a concern, which is in direct conflict with literature on software developers in general[114]. In addition productivity was deemed more important than program correctness.

"As is, I can't see the advantage of it because I already use Unity which manages everything. So there isn't, as such, any thing [requirement] there."

-Participant 4, Appendix D.2

Participant 5 expressed that F# was more readable because it forced correct program indentation, but still maintained that productivity was more important. Due to internet distribution platforms, bugs can be fixed at a later date and meeting the initial deadline is the main challenge. Furthermore, we speculate that games have a shorter lifetime (usually only one larger release and a series of smaller patches) than other software products and therefore maintenance is of lower priority.

6.2 Methodology

In this section we will discuss the chosen methodology for this project. A number of different methodologies were presented in Section 3.2, of these we opted for Champagne Prototyping. None of the methodologies were exact matches for our case, causing us to modify Champagne Prototyping to fit our case better. This modification consisted of including additional Attention Investment Models and Cognitive Dimensions usability methodologies.

6.2.1 Champagne Prototyping

The Champagne Prototyping method was selected because of its cheap deployment cost and the availability of a F# plugin for Unity [19]. This meant that early in the project the necessities of the method were met (see Section 3.2.5), i.e. a fully operational prototype based on an existing product. However, Discount Method for Language Evaluation was also an option. This methodology only requires a set of problems and some participants to solve them. The use of an IDE is optional, as the methodology is intended to test languages before a compiler has been implemented. This means that it is an ideal fit for evaluating programming languages. Champagne Prototyping uses a scenario-based task formulation strategy, which we argue is better suited for testing user interfaces. We therefore decided to draw inspiration from the tasks formulations in Discount Method for Language Evaluation and apply them with the participant selection and analysis of Champagne Prototyping. We chose the analysis of Champagne Prototyping for two reasons:

1. Discount Method for Language Evaluation uses IDA, which was simply not possible for us due to planning issues.
2. Cognitive Dimensions and Attention Investment Model, which are used by Champagne Prototyping, are better suited for evaluating programming languages than IDA.

A third alternative is Expert Review Method. However, as the name implies, this method requires an expert in each language under test. In addition, participants would be required to participate in multiple sessions. This was a concern, as finding qualified participants willing to spend an hour on the test already proposed a challenge. Given this challenge, Champagne Prototyping was the method which provided the most benefit at the smallest cost. This freed up time to improve the test setup and explore features such as the FRP system.

Attention Investment Model

The primary contribution of the Attention Investment Model in this project is the overview of participant comprehension. This can be seen in Table 4, where an overview of what the participants understood of the FRP system. The method was used as a broad-brush measure to participant comprehension, however it can be used as a finer grain tool. When used in this way, the method could be used to simulate participant behaviour and thus explain their decision making process[63]. This approach was not used due to the significant effort required. In addition, it was not obvious from the paper how to apply the method in practice.

Furthermore, the Attention Investment Model method was designed to map the decisions made by the programmer during a programming activity. In particu-

lar, the method may be used to explore when and why a programmer decides to generalize an implementation, instead of hard coding it. However, in Champagne Prototyping it is targeted at a certain feature. We used Attention Investment Model in accordance with Champagne Prototyping and in a similar manner to Cognitive Dimensions; as a vocabulary to discuss the decision-making process of the participants (see Section 4.2.4).

Multipass Cognitive Dimensions

The Cognitive Dimensions framework is used in Champagne Prototyping, but we also used the framework on its own. The reason for this was to discuss our experience with F# and C# separately from the participants' experience. This allowed us to compare the experiences and make educated guesses on whether some issues are resolved with more practice. This means that Cognitive Dimensions was applied twice, first as vocabulary to compare C# and F#, and second as part of Champagne Prototyping to gain an overview of usability issues. The first application was in the same style as our previous study of gameplay programming languages[12].

The main difference we observed between the two passes was that the participants only sparsely talked about paradigms. The participants had a tendency of associating all their problems with the new syntax, rather than the new way of thinking. Furthermore, we saw very little use of the function abstraction we discussed under the abstraction gradient dimension (see Section 4.1). Instead, the participants would attempt to use the object-orientation of F# to implement gameplay code like they were used to. We also observed that a participant did not want to add types in his F# code, because he argued that it would clash with the functional paradigm. Regarding consistency, we observed that the participants experienced many of the same issues as we did, particularly related to lists, type compatibility and function signatures. Many participants struggled with F#'s type inference, which we chose to classify as role-expressiveness. The reason for this was that some participants thought they were dealing with a dynamic type system, and we thus argue that F#'s type system does not sufficiently express that it is, in fact, strongly typed. This was an unexpected discovery, as we knew in advance that F# has a strong type system with inference.

6.3 Technology Choices

In this section we discuss the choices of technology for the project. We first discuss the choice of Unity as the host game engine for the experiment. Afterwards we discuss the choice of C# and F# as languages in this experiment.

6.3.1 Game Engine

In this project we chose to use the game engine Unity. We had three reasons for this choice; Unity is popular, it supports the .NET runtime and we have prior experience with it. We argued that any engine that supports the .NET runtime, such as Godot, CryEngine and MonoGame, would have been viable choices. Furthermore, we prefabricated code, scenes and assets that the programmers were to use during the tests. The goal of said prefabricated code was to rule out the game engine as a factor in the experiments and merely turn it into a "play button". We therefore argue that any game engine would have been viable choices, albeit some would require more setup than others. Take for example Unreal Engine, which has a virtual machine for running Blueprints[115]. This virtual machine could probably also host a functional language. However, it would require a new language to be built as none are readily available.

Unity is source-available for customers with the "Pro" subscription plan. This means that the engine and editor may only be adapted with the tools Unity Technologies provide (i.e. writing custom inspectors and editor plugins with C#). This puts a natural limit on how "far" custom implementations may diverge from Unity's implementation. A better, but also more time consuming approach would be to use the open source game engine Godot. Godot has support for GDScript and Mono, among others. We speculate that this could mean that new languages may be introduced with relative ease.

6.3.2 Programming Languages

We chose to use C# and F# in this experiment. This choice was mainly out of convenience because those two languages share the .NET runtime. The choice is associated with a potential source of error, namely that both languages are multi-paradigm. This means that C# programmers may in fact write fully functional programs and vice versa. In practice we saw fairly limited use of functional code in C# and more use of imperative code in F#. We suspect that this is a matter of habit, and is not an indicator that functional programming is unsuited for game development. The reason is that the participants were most likely used to a certain problem-solving strategy when they program, which is centered in the imperative paradigm.

The shared runtime is also worth discussing. We experienced that some of the participants were unsure of why one would use F# instead of C# because all the classes are called the same. This is a consequence of the shared runtime. We thought that the well-known classes would result in a smoother transition and a sense of familiarity, but it seems that the known classes and the way of expressing functionality in some cases were indistinguishable for our participants. With that in mind, it may have provided more useful information if we had used an entirely

other game engine for the functional part. On the other hand, this may introduce yet another learning factor in form of a completely new environment.

6.4 Performance Difference of F# and C#

In the benchmarks undertaken in this project, F# has consistently been out performed by C#. This section discuss this performance difference and it's significance. In addition, the parallelisation strategies, in both languages, are compared.

The benchmarks were structured to test the potential performance speed-up of using lenient evaluation strategy for implicit parallelisation (see Section 5.1). Therefore the first benchmarks compare a sequential implementation (as a control), an F# strategy (Async Workflows), a C# strategy (Tasks) and the lenient evaluation strategy. We found that the sequential control outperformed the concurrent implementations in all cases (see Section 5.1.4). This prompted the next benchmark, where we examined the growth of the strategies (see Section 5.2). However, the Task based and the lenient strategies performed comparably, while the execution times of Async Workflows grew very fast as the number of nodes increase. In these benchmarks F# was marginally slower than C#.

Given the surprising results in the first round of benchmarks, we hypothesised that the cause was the problem size. In order to tests this hypothesis another round of benchmarks were constructed (see Section 5.2). These tested the strategies with an arbitrary workload that was gradually decreased until the sequential strategies outperformed the concurrent ones. The workload was implemented as a pair of `for` loops iterating a given number of times (see Section 5.2.1 and Listing 26). The results indicated that our hypothesis was correct and that the parallel strategies scaled better than the sequential strategy(see Section 5.2.1 and Section 5.2.2).

Finally, a round of benchmarking was conducted inside the Unity engine. The focus of these benchmarks was to ascertain if F# caused a significant performance penalty in this environment. According to Unity's performance guidelines garbage generation will severely impact performance and F# generates more garbage than C# (see Section 5.3.1)[110], [111]. The findings indicates that F# scales worse than C#, depending on the employed strategy (see Section 5.3.2, Table 9 and Figure 17). However, this issue is largely dependent on Unity's use of an outdated GC strategy (see Section 5.3.1) and could therefore be solved by switching to modern GC algorithms.

Outside of Unity, F# has a small impact on performance, however the additional garbage has a larger impact on performance in Unity. This performance problem in Unity can be solved by modernising the GC. Therefore we can conclude that the performance is not significant for most cases. This is supported by another study, which found that F# was 7% slower than C# [40] and another study which

found a 5% difference[116].

Furthermore, the consensus of the participants in the usability evaluation seemed to be that productivity was more important than performance. This further underlines that it is no longer technical limitations that hinder the use of functional programming in game development. These arguments break a long tradition in game development, where performance have been the most important factor. We speculate that the participants of this study care little about performance, because they are not developing AAA games that push the limit of computational power. This could be one of the reasons why they're less concerned with performance than tradition says.

7 | Conclusion

This chapter serves as the conclusion of the report. Here the work undertaken during this project is summarised in detail, whereafter the research questions posed in the problem statement (see Section 1.1) are answered.

7.1 Project Summary

In this project we examined the claims of two game development gurus: John Carmack and Tim Sweeney. These claims suggested that increased use of functional programming in game development would be beneficial. These potential benefits of functional languages were examined. Tim Sweeney directly suggested two features; explicit effects typing and lenient evaluation strategy. The performance impact of these language features were examined as well as other related boons, such as FRP, implicit parallelism and other concurrency benefits.

We examined if their point-of-view was shared by game development professionals in Aalborg, by conducting a usability evaluation, where the participants were tasked with implementing gameplay code in F#. The test consisted of gameplay programming tasks inspired by game development. The developers were asked to implement solutions to these tasks in F# and C#. We found that the programmers were reluctant to adopt F# because they believed that the cost of learning a new language would out-weigh the benefits it could provide.

In need of a stronger incentive to promote F# we decided to examine if F# could provide more performant code, than C#, via concurrency. We found that F# introduces a performance penalty compared to C#, which is especially noticeable as problem sizes grow. Furthermore, the Async Workflows concurrency strategy employed by F# seems to be fragile, in the sense that small differences can severely impact performance (using `Async.StartChild` instead of `Async.Parallel`). In comparison, the Task model employed in C# seems more robust and also results in more performant concurrent code. The same results were obtained when benchmarking F# in Unity, where `MonoBehaviours` implemented in F# were a little less performant than those in C#. The FRP system was implemented sub-optimally, where each `FRPBehaviour` is in fact a full-blown FRP system, which resulted in poor performance. However, the FRP and Async Workflows were well understood by the participants during the test.

7.2 Research Questions

In this section we will answer the exploratory questions, which have guided the research in this project. The research questions are listed here for convenience:

1. **How well do experienced game developers express gameplay code in the functional paradigm, in comparison to object-oriented programming?**
2. How can functional programming be incorporated in game development?
3. What are the performance impacts of using functional idioms in a game engine?
4. What is required of functional programming to be adopted by game developers?

7.2.1 Expressing Gameplay Code in F#

The game developers struggled with various aspects of F# and the FRP system, however they managed to produce concise gameplay code (see Table 4 and Section 6.1). In some cases the F# code was shorter than the C# code written during the test (see Appendix E). In general, the F# code produced during the test often had qualities[1] that were lacking in the C# code. Even still, developers were unsure of whether F# was worth investing time in, which is illustrated in in Table 5. Several developers recognised the benefits of F#, but remained unsure if it was worth the investment risk, to switch to the language.

Of the findings from the usability test, the most surprising result was the difference of priorities (see Section 6.1.4). It was expected that developers would be mindful of risks associated with learning a new tool and that they may be reluctant to try something new, but developers disregarded the advantages of F# because they were unnecessary. The participants did not explicitly state the concrete reason why these features were unnecessary, but we believe the rapid and short life cycle of modern games is partly the cause.

7.2.2 Incorporating F# in Game Development

In this project we chose to add support for F# in Unity by installing a 3rd party plugin. Using this plugin we implemented a simple FRP system that allows gameplay programmers to employ the functional reactive programming paradigm. The

[1] Qualities such as conciseness, readability and modularity.

participants of the usability evaluation agreed that event-driven programming is well-suited for game development and would improve code quality.

The FRP system makes use of a paradigm within the functional paradigm, which forces developers to think about their code in a different way (see Section 3.1 and Section 4.2.1). This served as a segue into the functional paradigm. In contrast many of the traditional introductions to functional programming are not necessarily useful in real-world game development.

7.2.3 Performance Impacts of F#

Using microbenchmarking techniques and a set of benchmarks, we examined the performance of F# both standalone and in conjunction with Unity (see Section 5.1). We found that F# introduces a small overhead compared to C#. We implemented three different benchmarks in this project: binary tree summation/accumulation, matrix summation and unit management (see Section 5.1.2). All the benchmarks showed that F# was slightly slower than equivalent code in C#. Furthermore, the binary tree summation/accumulation benchmark showed that the Async Workflow concurrency model in F# was much slower than the other concurrent implementations (see Section 5.1.4). As C# and F# run on the same platform, programmers may use C#'s Task model in F#, which yields only slightly slower code in F#.

7.2.4 F# Adoption Requirements

The plausibility of the participants adopting F# has been discussed in Section 6.1. The participants list a number of problems that need to addressed before they would consider using F#. The primary concerns raised by the participants are the cost of learning the language, and the mismatch between desired benefits and gained benefits.

Easing the learning process may also alleviate another issue with F#: the availability of Unity/F# documentation[2]. Additional documentation and assistance would allow developers to learn much more rapidly, which was a major concern for most participants.

The benefits provided by F# were modularity and maintainability. Some participants noted that F# is ideal if you know the game you want to implement well, however they concluded that this is not the case in most game development scenarios (see Section 6.1 and Appendix D.2). The modularity afforded by F#

[2]While F# has sample documentation, very limited documentation is available for using F# with Unity.

was not considered an indicator for faster development, but rather an indicator of more maintainable code.

7.3 Closing Remarks

In conclusion, F# was met with mixed enthusiasm from the participants, while the use of events in gameplay programming was met with almost unanimous approval. The primary reasons identified, for the participants stance on F#, were the high perceived cost of learning the language, and the cultural shock of the functional paradigm. The benefits of F# were clear to several participants, but many did not consider them relevant to game development, where productivity is of utmost importance. This schism, may be caused by participants not fully grasping the advantages, but this requires additional research. The .NET platform has reached a point where the performance impact of using functional programming in game development is minimal. This means that the remaining barriers are habitual and preferential. We envision that such barriers can be breached by teaching the functional paradigm early in game developers' education.

8 | Future Work

In this chapter we outline options for future work. The options fall into three categories; further researching the use of lenient evaluation and if/how it can be adopted in F#, optimisation of the FRP system we developed and alternative usability evaluation methods that can evaluate the use of F# after longer exposure.

8.1 Lenient Evaluation in F#

In this section we present an avenue for future research into a lenient parallelisation system in F#. The theoretical background for this system is rooted in the lenient evaluation strategy and the work/span work estimation system (see Section 2.1 and Section 2.1.2). Two different pilot implementations were tested using Async Workflows from F# and .NET `Tasks` (see Section 5.1 and Section 5.1.2). The two parallelisation systems promise fine grain parallelisation, which is suited for the lenient system.

The performance of Async Workflows was not up to par with other approaches (at least not when workflows are started without the explicit `Async.StartChild`), however `Tasks` presented a promising system for lenient parallelisation in F#. This can be seen in Figure 3 and Figure 4. The pilot implementation is a naïve implementation of lenient evaluation in F#. The system could benefit from using work/span and hardware information to make informed decisions about what and when to parallelise. The benefit of using work/span, is that the analysis can be made at compile time.

However, the above results are not concrete proof that such a system could provide a significant speed up in real world scenarios. Therefore a new implementation, using work/span or a different analysis tool, should be implemented and benchmarked. This benchmark should be compared to state of the art sequential and concurrent implementations to ascertain relative speed up. However, if the speed up is not significant, the system may still hold merit as a potentially easier to use parallelisation system.

8.2 Improving the FRP System

In this section we discuss how the FRP system can be improved in terms of performance. We first outline the performance problems with the current implementation and suggest a refactoring that should improve performance. Afterwards we discuss the possibility of introducing implicit concurrent updates using Unity's C# Job System.

8.2.1 FRP Optimisation

The problem with the FRP system is that each `FRPBehaviour` is actually a full-blown FRP-system with condition-checking and event-dispatching. The FRP system could well be a singleton, as this means that conditions are only checked once.

Solving this problem would require a larger refactoring, as this relates to the Unity lifecycle of `GameObjects`. First and foremost some Unity methods are required to be tied to the `GameObject` they belong to. Examples of such methods are `OnCollisionEnter` and `OnTriggerExit`. Other methods, such as `Update` and reacting to keyboard strokes could, on the other hand, be tied to a `FRPEngine`. In this approach the programmer will register each `FRPBehaviour` to the set of events he would like to react to along with a condition and a handler.

The problem arises when `GameObjects` are destroyed and the event handlers must be unsubscribed. We have not added support to remove `FRPBehaviours` and their event handlers from the FRP system, as the current version "cleans" up after itself when `FRPBehaviours` are destroyed. This is to be understood in the sense that the whole system is deallocated and thus never risks invoking event handlers on objects that have been destroyed. A possible pitfall of the suggested optimisation is the risk of invoking event handlers on `FRPBehaviours` that have already been destroyed. This raises an exception in Unity and yields an unresponsive game.

In order to truly determine whether or not FRP comes with a performance penalty, these changes would have to be incorporated.

8.2.2 Unity Concurrency

The possibility of promising implicit concurrency from using the FRP system is indeed appealing. Yampa Arcade has managed to implement a concurrent FRP system[46]. In order for this to be possible we would need to utilise the C# Job System (which, despite its name, should also work in F#) or ECS in Unity. The

former allows programmers to implement concurrent code on `MonoBehaviours`, whereas the latter uses `Entitys` rather than `MonoBehaviours`.

The current implementation implements a `FRPBehaviour`, which inherits from `MonoBehaviour`. The C# Job System therefore presents the smallest required change. If we were to implement concurrency using C# Job System, we would create a collection for each event type that holds references to the objects that has subscribed to the event along with the handler they subscribed. In each `Update` those collections will be iterated, in parallel, to check the event conditions and call the associated handlers. There are some engineering challenges still, as Unity requires programmers to manually schedule the jobs and only allows value-types (i.e. `structs`) to be passed as arguments to a job.

With the coming release of Unity's ECS, it would also be interesting to examine how well that pairs with FRP. We speculate that they might go well hand-in-hand, as events in FRP can be replaced with Systems in ECS that are invoked only when certain conditions are met. The events of FRP may then be represented as a series of components that are attached to the entities. There are likely many engineering challenges to face when developing such system, and our experience with Unity's ECS tells us that it is far from simple to use.

8.3 Longer Term Usability Evaluation

The usability evaluation presented in this project gives only a small glimpse into the challenges faced by experienced C# programmers when they start using F#. We suspect that part of the reason why the participants were not keen on adopting F# was that they only experienced the "initial frustration" of switching to a new language. Another interesting topic to explore is whether experienced gameplay programmers are more positive towards F# after gaining more experience. This would require a longer-term usability evaluation to be conducted.

A longer-term usability evaluation presents the challenge that the programmers may not be monitored all the time, thus decreasing the amount of insight we can obtain on F#'s learning curve. If we assume that it is not necessary to monitor the participants, we can draw inspiration from the expert-review method presented in [65] and the idea of structuring exercises as game prototypes presented in [40]. This combined method would formulate a series of game prototype exercises, and send them to participants at regular interval along with a questionnaire that examines how happy the participant was with using F#. Alternatively several small usability test sessions could be arranged, where the participants solve similar exercises. Both these approaches require a huge time investment from the participants and the latter requires a lot of planning. Nonetheless, these approaches may provide another interesting point-of-view as to why functional programming has not been broadly accepted in the game development industry.

Alternatively a comparative experiment could draw inspiration from [117], that studies the effect of dynamic and static types on programming languages. Such experiment would gather a larger group of participants and have half of them implement a game prototype in C# and half in F#. The total development time should be roughly 27 hours[117]. Alternatively, online resources could be used to evaluate the usability, by having the participants complete an online course before the actual session.

8.4 Reactive Programming in C#

The participants of the usability all agreed that the use of events in gameplay programming would increase code quality compared to Unity's current `Update`-strategy. It would therefore be interesting to research if reactive programming would receive a "warmer welcome" into the game development industry. One possible strategy would be to repeat a similar experiment, but instead of using F#, a reactive programming system for Unity (such as UniRx[43]) would be used. As all tasks in this experiment would be implemented in C#, it would also have the benefit that the "initial frustration", which we discussed earlier in this section, would be entirely removed.

Bibliography

[1] E. F. Anderson, "A classification of scripting systems for entertainment and serious computer games", in *2011 Third International Conference on Games and Virtual Worlds for Serious Applications*, May 2011, pp. 47–54.

[2] D. Michael, "Indie game development survival guide (game development series)", 2003.

[3] M. M. McGill, "Defining the expectation gap: A comparison of industry needs and existing game development curriculum", in *Proceedings of the 4th International Conference on Foundations of Digital Games*, ACM, 2009, pp. 129–136.

[4] M. Hewner and M. Guzdial, "What game developers look for in a new graduate: Interviews and surveys at one game company", in *Proceedings of the 41st ACM technical symposium on Computer science education*, ACM, 2010, pp. 275–279.

[5] J. Blow, "Game development: Harder than you think", *Queue*, vol. 1, no. 10, p. 28, 2004. [Online]. Available: http://faculty.salisbury.edu/~xswang/Research/papers/game/queuefeb04/blow.pdf.

[6] Wikipedia. (May 2019). List of game engines. English, [Online]. Available: https://en.wikipedia.org/wiki/List_of_game_engines (visited on May 22, 2019).

[7] Unreal. (2018). Introduction to blueprints. English, [Online]. Available: https://docs.unrealengine.com/en-US/Engine/Blueprints/GettingStarted (visited on Dec. 12, 2018).

[8] Unreal Engine. (Aug. 2015). Blueprint overview. English, [Online]. Available: https://docs.unrealengine.com/en-us/Engine/Blueprints/Overview (visited on May 22, 2019).

[9] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data", *IEEE transactions on software engineering*, vol. 35, no. 4, pp. 534–550, 2009.

[10] J. Hughes, "Why functional programming matters", *The computer journal*, vol. 32, no. 2, pp. 98–107, 1989. [Online]. Available: https://academic.oup.com/comjnl/article-pdf/32/2/98/1445644/320098.pdf.

[11] Z. Hu, J. Hughes, and M. Wang, "How functional programming mattered", *National Science Review*, vol. 2, no. 3, pp. 349–370, 2015.

[12] T. Morell, M. E. R. Andersen, T. S. Jensen, T. G. McCollin, and D. D. A. van Bolhuis, "An analysis of gameplay programming languages in free-to-use game engines", Aalborg University, Tech. Rep., 2019. [Online]. Available: https://projekter.aau.dk/projekter/da/studentthesis/en-analyse-af-spilopfoerselsprogrammeringssprog-i-gratisatbruge-spilmotorer(99442369-1b23-4deb-b99d-060ae5cd5db2).html.

[13] J. Carmack. (Apr. 2012). In-depth: Functional programming in C++. English, [Online]. Available: https://www.gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php (visited on Sep. 21, 2018).

[14] Microsoft Docs. (Mar. 2019). Lambda expressions (c# programming guide). English, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions (visited on Apr. 30, 2019).

[15] M. Torgersen. (Jan. 2019). Do more with patterns in c# 8.0. English, [Online]. Available: https://devblogs.microsoft.com/dotnet/do-more-with-patterns-in-c-8-0/ (visited on Apr. 30, 2019).

[16] Daniel Super. (Sep. 2016). Why do we don't use functional programming to implement a game engine? English, [Online]. Available: https://www.quora.com/Why-do-we-dont-use-functional-programming-to-implement-a-game-engine (visited on May 22, 2019).

[17] T. Sweeney. (2006). The next mainstream programming language: A game developer's perspective. English, Epic Games, [Online]. Available: https://www.st.cs.uni-saarland.de/edu/seminare/2005/advanced-fp/docs/sweeny.pdf (visited on Nov. 20, 2018).

[18] gamedesigning.org. (Jan. 2019). The top 10 video game engines. English, [Online]. Available: https://www.gamedesigning.org/career/video-game-engines/ (visited on May 22, 2019).

[19] M. Rosenbjerg. (Mar. 2019). Unity f# integration. English, [Online]. Available: https://github.com/sppt-2k19/unity-fsharp-integration (visited on Apr. 2, 2019).

[20] L. Kokemohr. (Mar. 2018). Using f# in godot 3. English, [Online]. Available: http://www.lkokemohr.de/fsharp_godot.html (visited on May 7, 2019).

[21] A. Brown. (Oct. 2013). Making a platformer in f# with monogame. English, [Online]. Available: https://bruinbrown.wordpress.com/2013/10/06/making-a-platformer-in-f-with-monogame/ (visited on May 7, 2019).

[22] rookboom. (Nov. 2012). Using f# in crymono. English, [Online]. Available: https://www.cryengine.com/community_archive/viewtopic.php?f=375&t=102099 (visited on May 7, 2019).

[23] P. Hudak, "Para-functional programming", *Computer;(United States)*, vol. 19, no. 8, 1986.

[24] H. W. Loidl, "Granularity in large-scale parallel functional programming", PhD thesis, University of Glasgow, 1998.

[25] G. Tremblay, "Lenient evaluation is neither strict nor lazy", *Comput. Lang.*, vol. 26, no. 1, pp. 43–66, 2000. DOI: 10.1016/S0096-0551(01)00006-6. [Online]. Available: https://doi.org/10.1016/S0096-0551(01)00006-6.

[26] G. Tremblay and B. Malenfant, "Lenient evaluation and parallelism", *Comput. Lang.*, vol. 26, no. 1, pp. 27–41, 2000. DOI: 10.1016/S0096-0551(01)00007-8. [Online]. Available: https://doi.org/10.1016/S0096-0551(01)00007-8.

[27] H. Hüttel, *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010.

[28] P. Hudak, "Conception, evolution, and application of functional programming languages", *ACM Computing Surveys (CSUR)*, vol. 21, no. 3, pp. 359–411, 1989. [Online]. Available: https://cse.sc.edu/~mgv/csce330f15/haskell/p359-hudak.pdf.

[29] R. Bird, G. Jones, and O. De Moor, "More haste, less speed: Lazy versus eager evaluation", *Journal of Functional Programming*, vol. 7, no. 5, pp. 541–547, 1997.

[30] H. C. Baker Jr and C. Hewitt, "The incremental garbage collection of processes", *ACM Sigplan Notices*, vol. 12, no. 8, pp. 55–59, 1977.

[31] H. Casanova, A. Legrand, and Y. Robert, *Parallel Algorithms*. CRC Press, 2008, ISBN: 978-1-58488-945-8. [Online]. Available: http://www.crcpress.com/product/isbn/9781584889458.

[32] G. E. Blelloch, "Programming parallel algorithms", *Commun. ACM*, vol. 39, no. 3, pp. 85–97, Mar. 1996, ISSN: 0001-0782. DOI: 10.1145/227234.227246. [Online]. Available: http://doi.acm.org/10.1145/227234.227246.

[33] J. L. Gustafson, "Brent's theorem", in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA: Springer US, 2011, pp. 182–185, ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_80. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_80.

[34] R. P. Brent, "The parallel evaluation of general arithmetic expressions", *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 201–206, 1974.

[35] F. Nielson, H. R. Nielson, and C. Hankin, "Type and effect systems", in *Principles of Program Analysis*, Springer, 1999, pp. 283–363.

[36] M. Toro and É. Tanter, "Customizable gradual polymorphic effects for scala", in *ACM SIGPLAN Notices*, ACM, vol. 50, 2015, pp. 935–953.

[37] R. D. Team. (Apr. 2013). Rust documentation. English, Rust Docs Team, [Online]. Available: https://doc.rust-lang.org/ (visited on Mar. 11, 2019).

[38] M. Krogh-Jespersen, K. Svendsen, and L. Birkedal, "A relational model of types-and-effects in higher-order concurrent separation logic", in *ACM SIGPLAN Notices*, ACM, vol. 52, 2017, pp. 218–231.

[39] L. Birkedal, F. Sieczkowski, and J. Thamsborg, "A concurrent logical relation", in *Computer Science Logic (CSL'12)-26th International Workshop/21st Annual Conference of the EACSL*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[40] M. E. R. Andersen, T. S. Jensen, and D. D. A. van Bolhuis, "Functional got game? - evaluating functional programming in game development", Aalborg University, Tech. Rep., 2019. [Online]. Available: https://projekter.aau.dk/projekter/da/studentthesis/en-analyse-af-spilopfoerselsprogrammeringssprog-i-gratisatbruge-spilmotorer(9944
1b23-4deb-b99d-060ae5cd5db2).html.

[41] vis2k. (Feb. 2018). F# kit - asset store. English, [Online]. Available: https://assetstore.unity.com/packages/tools/utilities/f-kit-63652 (visited on May 20, 2019).

[42] R. N. T̃. Gardner. (Feb. 2017). Arcadia: Clojure in unity. English, [Online]. Available: https://github.com/arcadia-unity/Arcadia (visited on Sep. 25, 2018).

[43] Y. Kawai. (Sep. 2018). Unirx - reactive extensions for unity. English, [Online]. Available: https://assetstore.unity.com/packages/tools/integration/unirx-reactive-extensions-for-unity-17276 (visited on May 20, 2019).

[44] ——, (Sep. 2018). Unirx - reactive extensions for unity. English, [Online]. Available: https://github.com/neuecc/UniRx (visited on May 20, 2019).

[45] C. Elliott and P. Hudak, "Functional reactive animation", in *International Conference on Functional Programming*, 1997. [Online]. Available: http://conal.net/papers/icfp97/.

[46] A. Courtney, H. Nilsson, and J. Peterson, "The yampa arcade", in *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, ACM, 2003, pp. 7–18.

[47] M. H. Cheong, "Functional programming and 3d games", *BEng thesis, University of New South Wales, Sydney, Australia*, 2005.

[48] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt, "Pycket: A tracing jit for a functional language", in *ACM SIGPLAN Notices*, ACM, vol. 50, 2015, pp. 22–34.

[49] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt, "Chaperones and impersonators: Run-time support for reasonable interposition", in *ACM SIGPLAN Notices*, ACM, vol. 47, 2012, pp. 943–962.

[50] P. Sestoft, "Microbenchmarks in java and c#", *Lecture Notes, Sept*, 2013. [Online]. Available: https://itu.dk/~sestoft/papers/benchmarking.pdf.

[51] C. Maraffi and D. Seagal, "Leveling up: Could functional programming be a game changer?",

[52] Lettier. (Oct. 2018). Your easy guide to fucntional reactive programming (frp). English, Medium, [Online]. Available: https://medium.com/@lettier/functional-reactive-programming-a0c7b08f6b67 (visited on Mar. 5, 2019).

[53] N. Singh. (Mar. 2018). A quick introduction to functional reactive programming (frp). English, Medium, [Online]. Available: https://medium.freecodecamp.org/functional-reactive-programming-frp-imperative-vs-declarative-vs-reactive-style-84878272c77f (visited on Mar. 5, 2019).

[54] Z. Corr. (Oct. 2016). Helm: A functionally reactive game engine. English, [Online]. Available: http://hackage.haskell.org/package/helm (visited on Sep. 25, 2018).

[55] B. Edds. (Mar. 2017). Nu. English, [Online]. Available: https://github.com/bryanedds/Nu (visited on Sep. 21, 2018).

[56] ——, (Jun. 2016). Why functional programming works for games. English, [Online]. Available: https://medium.com/@bryanedds/functional-game-programming-can-work-95ed0df14f77 (visited on Dec. 6, 2018).

[57] E. Rey. (Jun. 2018). Functional reactive programming explained in a simple way, in javascript... yes, in a simple way. English, Medium, [Online]. Available: https://itnext.io/functional-reactive-programming-explained-in-a-simple-way-in-javascript-yes-in-a-simple-way-925b14cddf75 (visited on Mar. 5, 2019).

[58] D. Quick. (Nov. 2018). Euterpea - a haskell library for music creation. English, [Online]. Available: http://www.euterpea.com/ (visited on Apr. 30, 2019).

[59] J. Kjeldskov, M. B. Skov, and J. Stage, "Instant data analysis: Conducting usability evaluations in a day", in *Proceedings of the third Nordic conference on Human-computer interaction*, ACM, 2004, pp. 233–240.

[60] S. Kurtev, T. A. Christensen, and B. Thomsen, "Discount method for programming language evaluation", in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, ACM, 2016, pp. 1–8. [Online]. Available: https://projekter.aau.dk/projekter/files/239518386/report.pdf.

[61] D. Benyon, "Designing interactive systems: A comprehensive guide to hci, ux and interaction design", 2014.

[62] T. R. G. Green, M. Petre, *et al.*, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework", *Journal of visual languages and computing*, vol. 7, no. 2, pp. 131–174, 1996.

[63] A. F. Blackwell, "First steps in programming: A rationale for attention investment models", in *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, IEEE, 2002, pp. 2–10.

[64] A. F. Blackwell, M. M. Burnett, and S. P. Jones, "Champagne prototyping: A research technique for early evaluation of complex end-user programming systems", in *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, IEEE, 2004, pp. 47–54.

[65] S. Nanz, S. West, and K. S. Da Silveira, "Examining the expert gap in parallel programming", in *European Conference on Parallel Processing*, Springer, 2013, pp. 434–445.

[66] G. V. Wilson and R. B. Irvin, *Assessing and comparing the usability of parallel programming systems*. Citeseer, 1995.

[67] S. Nanz, S. West, K. S. Da Silveira, and B. Meyer, "Benchmarking usability and performance of multicore languages", in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, IEEE, 2013, pp. 183–192.

[68] Unity Technologies. (Apr. 2019). Unity - manual: C# job system. English, [Online]. Available: https://docs.unity3d.com/Manual/JobSystem.html (visited on Apr. 29, 2019).

[69] ——, (Jun. 2018). Introduction to the entity component system and c# job system. English, [Online]. Available: https://unity3d.com/learn/tutorials/topics/scripting/introduction-entity-component-system-and-c-job-system (visited on Apr. 29, 2019).

[70] S. Vermeulen. (Sep. 2017). Async-await instead of coroutines in unity 2017. English, [Online]. Available: http://www.stevevermeulen.com/index.php/2017/09/using-async-await-in-unity3d-2017/ (visited on Apr. 30, 2019).

[71] Microsoft Docs. (Mar. 2019). The task asynchronous programming model in c#. English, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/ (visited on Apr. 30, 2019).

[72] G. Foster. (Dec. 2013). Understanding and implementing scene graphs. English, [Online]. Available: http://archive.gamedev.net/archive/reference/programming/features/scenegraph/index.html (visited on Apr. 30, 2019).

[73] M. Jordan. (Nov. 2018). Entities, components and systems. English, [Online]. Available: https://medium.com/ingeniouslysimple/entities-components-and-systems-89c31464240d (visited on Apr. 30, 2019).

[74] E. Bendersky. (May 2016). The expression problem and its solutions. English, [Online]. Available: https://eli.thegreenplace.net/2016/the-expression-problem-and-its-solutions/ (visited on May 21, 2019).

[75] R. D. Tennent, "Language design methods based on semantic principles", *Acta Informatica*, vol. 8, no. 2, pp. 97–112, 1977.

[76] E. Kindler and I. Krivy, "Object-oriented simulation of systems with sophisticated control", *International Journal of General Systems*, vol. 40, no. 3, pp. 313–343, 2011.

[77] L. Mathiassen, A. Munk-Madsen, P. A. Nielsen, and J. Stage, *Object-oriented analysis & design*. Citeseer, 2000, vol. 25.

[78] S. Wlaschin. (Jul. 2012). Understanding type inference | f# for fun and profit. English, [Online]. Available: https://fsharpforfunandprofit.com/posts/type-inference/ (visited on May 7, 2019).

[79] C. Smith. (Aug. 2009). If #light syntax is so much better in f#, why isn't it the default? English, [Online]. Available: https://stackoverflow.com/questions/1354722/if-light-syntax-is-so-much-better-in-f-why-isnt-it-the-default (visited on May 22, 2019).

[80] S. Kumar. (Jul. 2017). Composite design pattern. English, [Online]. Available: https://www.geeksforgeeks.org/composite-design-pattern/ (visited on May 22, 2019).

[81] K. Naik. (Mar. 2019). Design patterns in .net. English, [Online]. Available: https://www.c-sharpcorner.com/UploadFile/bd5be5/design-patterns-in-net/ (visited on May 22, 2019).

[82] N. Ford. (Mar. 2012). Functional design patterns. English, [Online]. Available: https://www.ibm.com/developerworks/library/j-ft10/index.html (visited on May 22, 2019).

[83] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github", in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014, pp. 155–165.

[84] A. D. Green. (Jan. 2017). Statements - c# language specification. English, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/statements#the-goto-statement (visited on May 7, 2019).

[85] B. Scott. (Jul. 2011). Does anyone still use [goto] in c# and if so why? English, [Online]. Available: https://stackoverflow.com/questions/6545720/does-anyone-still-use-goto-in-c-sharp-and-if-so-why (visited on May 7, 2019).

[86] V. Guana, E. Stroulia, and V. Nguyen, "Building a game engine: A tale of modern model-driven engineering", in *Games and Software Engineering (GAS), 2015 IEEE/ACM 4th International Workshop on*, IEEE, 2015, pp. 15–21.

[87] R. Nystrom, *Game programming patterns*. Genever Benning, 2014.

[88] Unity Technologies. (Apr. 2019). Variables and the inspector. English, [Online]. Available: https://docs.unity3d.com/Manual/VariablesAndTheInspecto
html (visited on May 7, 2019).

[89] tnetennba. (Aug. 2011). How do i find which objects are referencing another? English, [Online]. Available: https://answers.unity.com/questions/155746/how-do-i-find-which-objects-are-referencing-anothe.html (visited on May 7, 2019).

[90] L. Mikhajlov and E. Sekerinski, "A study of the fragile base class problem", in *European Conference on Object-Oriented Programming*, Springer, 1998, pp. 355–382.

[91] Unreal Engine. (Aug. 2015). Components. English, [Online]. Available: https://docs.unrealengine.com/en-us/Programming/UnrealArchitecture/Actors/Components (visited on May 21, 2019).

[92] A. Singla. (Jul. 2016). Understanding interfaces via loose coupling and tight coupling. English, [Online]. Available: https://www.c-sharpcorner.com/blogs/understanding-interfaces-via-loose-coupling-and-tight-coupling (visited on May 6, 2019).

[93] C. Borley. (Sep. 2018). C# interactive walkthrough. English, [Online]. Available: https://github.com/dotnet/roslyn/wiki/C%23-Interactive-Walkthrough (visited on May 6, 2019).

[94] Microsoft Docs. (May 2016). Interactive programming with f#. English, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/fsharp-interactive/ (visited on May 6, 2019).

[95] M. Docs. (May 2016). Xml documentation. English, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/xml-documentation (visited on May 3, 2019).

[96] S. McLeod. (2013). What is validity? English, [Online]. Available: https://www.simplypsychology.org/validity.html (visited on Jan. 4, 2019).

[97] R. A. Virzi, "Refining the test phase of usability evaluation: How many subjects is enough?", *Human factors*, vol. 34, no. 4, pp. 457–468, 1992.

[98] W. Hwang and G. Salvendy, "Number of people required for usability evaluation: The 10±2 rule", *Communications of the ACM*, vol. 53, no. 5, pp. 130–133, 2010.

[99] Microsoft Docs. (Dec. 2018). Unchecked keyword - c# reference. English, Inquisitir, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/unchecked (visited on Feb. 26, 2019).

[100] Mankarse. (Aug. 2018). How do i explicitly use unchecked arithmetic operators in f#. English, [Online]. Available: https://stackoverflow.com/questions/51672820/how-do-i-explicitly-use-unchecked-arithmetic-operators-in-f (visited on May 15, 2019).

[101] Unity Technologies. (May 2018). Optimizing garbage collection in unity games. English, [Online]. Available: https://unity3d.com/learn/tutorials/topics/performance-optimization/optimizing-garbage-collection-unity-games (visited on Apr. 2, 2019).

[102] ——, (Dec. 2018). Optimizing garbage collection in unity games. English, [Online]. Available: https://docs.unity3d.com/Manual/BestPracticeUnderstanding1.html (visited on Apr. 2, 2019).

[103] H.-J. Boehm and M. Spertus, "Transparent programmer-directed garbage collection for c+", *URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers*, no. 2310, 2007.

[104] P. Sestoft, *Programming language concepts*. Springer, 2017.

[105] M. Docs. (Aug. 2018). Fundamentals of garbage collection. English, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals (visited on Apr. 2, 2019).

[106] Mono Project. (Feb. 2019). Generational gc. English, [Online]. Available: https://www.mono-project.com/docs/advanced/garbage-collector/sgen/ (visited on Apr. 8, 2019).

[107] Unity Technologies. (Mar. 2019). Mono: Mono open source ecma cli, c# and .net implementation. English, [Online]. Available: https://github.com/Unity-Technologies/mono (visited on Apr. 12, 2019).

[108] J. Peterson. (Jul. 2015). Il2cpp internals - garbage collector integration. English, [Online]. Available: https://blogs.unity3d.com/2015/07/09/il2cpp-internals-garbage-collector-integration/ (visited on Apr. 8, 2019).

[109] Unity Technologies. (Mar. 2019). Roadmap - unity. English, [Online]. Available: https://unity3d.com/unity/roadmap (visited on Apr. 8, 2019).

[110] K. Nørmark, B. Thomsen, and L. L. Thomsen, "Mapping and visiting in functional and object-oriented programming.", *Journal of Object Technology*, vol. 7, no. 7, pp. 75–107, 2008.

[111] Unity Technologies. (Jul. 2013). Unity feedback - f# support. English, [Online]. Available: https://web.archive.org/web/20180408181140/https://feedback.unity3d.com/suggestions/f-support (visited on Apr. 2, 2019).

[112] V. Simonov. (Dec. 2015). 10000 update() calls. English, [Online]. Available: https://blogs.unity3d.com/2015/12/23/1k-update-calls/ (visited on Apr. 8, 2019).

[113] K. Hinnum. (May 2013). Intel core i7 4702hq notebook processor. English, [Online]. Available: https://www.notebookcheck.net/Intel-Core-i7-4702HQ-Notebook-Processor.93265.0.html (visited on May 20, 2019).

[114] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption", in *ACM SIGPLAN Notices*, ACM, vol. 48, 2013, pp. 1–18.

[115] Epic Games. (2019). Nativizing blueprints. English, [Online]. Available: https://docs.unrealengine.com/en-US/Engine/Blueprints/TechnicalGuide/NativizingBlueprints (visited on Jan. 8, 2019).

[116] G. Maggiore, A. Spanò, R. Orsini, M. Bugliesi, M. Abbadi, and E. Steffinlongo, "A formal specification for casanova, a language for computer games.", in *EICS*, 2012, pp. 287–292.

[117] S. Hanenberg, "An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time", in *ACM Sigplan Notices*, ACM, vol. 45, 2010, pp. 22–35.

List of Figures

List of Tables

List of Listings

A | Concurrency in Unity

```
1   public class BulletJobSystem : MonoBehaviour {
2       //This array holds all bullets that need to be moved
3       private TransformAccessArray _transforms;
4       //This job handle may be used to synchronise with
        ↪ previously scheduled move jobs
5       private JobHandle _bulletMoveHandle;
6       public GameObject BulletPrefab;
7
8       //This struct implements the actual job.
9       public struct MoveBulletsJob : IJobParallelForTransform {
10          public float _moveSpeed;
11          public float _deltaTime;
12
13          //In the Execute-method we define how we want to apply
            ↪ an update to a single bullet.
14          public void Execute(int index, TransformAccess
            ↪ transform) {
15              transform.position += _moveSpeed * _deltaTime *
                ↪ (transform.rotation * new Vector3(0,0,1));
16          }
17      }
18
19      void Start() {
20          //Create an array with 0 elements and unlimited
            ↪ capacity
21          _transforms = new TransformAccessArray(0, -1);
22      }
23  }
```

Listing 29: Implementation of a job that moves bullets forward in Unity. `SpawnBullet` and `Update` listed in Listing 30.

```
public void SpawnBullet(Transform shooter) {
    //Instantiate the bullet as usual in Unity
    var bullet = Instantiate(BulletPrefab);
    bullet.transform.position = shooter.position;
    bullet.transform.forward = shooter.forward;

    //Synchronise with the job and add the new bullet to the
    ↪  _transforms array.
    _bulletMoveHandle.Complete();
    _transforms.capacity++;
    _transforms.Add(bullet.transform);
}

void Update() {
    //Complete the job, i.e. only apply one update at a time
    _bulletMoveHandle.Complete();

    //And schedule a new bullet update
    var bulletMoveJob = new MoveBulletsJob (5f, Time.deltaTime);
    _bulletMoveHandle = bulletMoveJob.Schedule(_transform);
    JobHandle.ScheduleBatchedJobs();
}
```

Listing 30: Spawning bullets and updating when using Unity's C# Job System, part of Listing 29.

```
//MoveForward component. Defines data that is needed to move forward
struct MoveForward {
    public float Speed;
}

//C# Job system that moves forward in parallel
public class MoveForwardSystem : JobComponentSystem {
    //The generic types of the interface implements the filter
    struct MoveForwardJob : IJobForEach<Translation, Rotation, MoveForward> {
        private readonly float _deltaTime;

        //Move the entity forward. We do not have access to "standard" Unity methods, so we use the dedicated math-library
        public void Execute(ref Translation trans, [ReadOnly]ref Rotation rotation, [ReadOnly]ref MoveForward moveForward) {
            var forward = math.forward(rotation);
            var step = math.mul(moveForward.Speed, _deltaTime);
            var moveVec = math.mul(forward, new float3(step));
            trans.Value += moveVec;
        }
    }
}

//On each update schedule the movement
protected void JobHandle OnUpdate(JobHandle inputDeps) {
    var job = new MoveForwardJob{_deltaTime = Time.deltaTime};
    return job.Schedule(this, inputDeps);
}
```

Listing 31: Implementation of an ECS that moves bullets forward in Unity.

B | Benchmark Data

B.1 Binary Tree Benchmarks - F#

Problem Size (nodes)	Sequential	Async Workflows	Lenient
1	5726.39	0.00	10 060.09
4	2934.12	304 391.32	7672.24
8	2002.27	174 295.21	9167.75
16	1667.49	160 006.82	11 047.56
32	1509.51	15 158 107.00	12 837.72
64	1453.71	39 083 439.31	15 403.18
128	1547.82	76 112 965.72	21 190.94
256	1866.72	113 039 254.44	67 846.72
512	2538.33	0.00	137 755.58
1024	19 424.02	0.00	200 246.98
2048	19 496.68	0.00	344 141.83
4096	21 684.81	0.00	574 822.19
8192	26 659.38	0.00	1 437 555.04
16384	36 769.07	0.00	3 359 999.51
32768	56 058.70	0.00	7 327 466.76
65536	93 581.38	0.00	14 832 354.85

Table 11: Running time in ns in Binary Tree Accumulation benchmark for F#.

Problem Size (nodes)	Sequential	Async Workflows	Lenient
1	3973.07	0.00	10 140.72
4	2019.53	150 249.33	13 163.97
8	1375.65	89 144.24	12 650.41
16	1066.52	201 812.51	11 486.46
32	903.28	158 448.38	11 603.26
64	831.55	136 880.42	12 814.82
128	867.02	127 539.91	16 802.54
256	984.22	134 530.20	22 587.04
512	1288.59	0.00	32 486.31
1024	1856.41	0.00	48 883.27
2048	2874.54	0.00	78 875.67
4096	4893.77	0.00	133 471.83
8192	8954.68	0.00	235 329.22
16384	15 959.52	0.00	430 012.89
32768	30 024.27	0.00	800 875.09
65536	70 139.41	0.00	1 504 625.89

Table 12: Running time in ns in Binary Tree Summation benchmark for F#.

B.2 Binary Tree Benchmarks - C#

Problem Size (nodes)	Sequential	Fork Join	Lenient
1	9857.23	36 629.28	66 965.76
4	4994.40	43 676.94	67 174.41
8	3387.22	47 271.08	48 758.94
16	2596.80	38 250.07	43 577.53
32	2158.45	35 319.98	45 700.71
64	1901.25	37 010.60	55 606.06
128	1790.22	40 612.43	73 102.69
256	1832.47	49 199.53	115 121.84
512	2077.46	65 980.95	197 065.25
1024	2673.38	96 712.47	372 377.09
2048	3841.37	154 412.53	829 143.96
4096	6208.19	264 385.89	2 227 778.63
8192	14 926.54	467 222.64	3 921 581.87
16384	23 213.30	843 050.00	6 591 468.74
32768	36 584.34	1 547 022.41	14 741 916.07
65536	68 863.90	2 918 545.61	32 985 892.27

Table 13: Running time in ns in Binary Tree Accumulation benchmark for C#.

Problem Size (nodes)	Sequential	Fork Join	Lenient
1	2257.68	19 163.44	23 054.61
4	1159.63	19 908.28	12 203.96
8	798.08	13 487.85	8374.54
16	628.82	10 467.70	6693.74
32	544.37	8973.08	6038.03
64	516.16	8581.24	6234.30
128	546.74	10 140.04	7547.88
256	655.92	15 984.02	10 434.79
512	896.26	20 904.28	16 115.95
1024	1369.66	29 559.09	26 874.44
2048	2291.90	46 303.59	47 002.19
4096	4018.59	77 999.85	84 505.84
8192	7482.40	139 589.84	156 087.74
16384	13 593.56	256 724.40	291 697.92
32768	25 462.02	475 981.33	548 702.11
65536	47 126.04	895 730.04	1 036 433.09

Table 14: Running time in ns in Binary Tree Summation benchmark for C#.

B.3 Critical Work Data

Work Bias (iterations)	Sequential	Fork Join	Lenient
16777216	662923924	189587662	184895505
8388608	330032131	89513135	90196290
4194304	162695008	44769190	45062797
2097152	81608808	22795847	22560273
1048576	42350887	11337276	11355387
524288	21133781	6319716	6158697
262144	10286965	3165200	3134721
131072	5363634	1725501	1876498
65536	2762979	1334927	877759
32768	1565743	666707	518750
16384	702927	278365	234715
8192	466160	201201	156440
4096	203296	146571	106618
2048	112218	99716	105708
1024	60694	94271	558531

Table 15: Results from binary tree summation benchmark with work bias, all measures in ns. Models data dependency between subtrees and delay.

Work Bias (iterations)	Sequential	Fork Join	Lenient
16777216	676290609	196231564	183374685
8388608	340133573	88402011	90044598
4194304	166155127	44803618	44810012
2097152	83439476	22945558	22630346
1048576	42600364	12011160	11702572
524288	21881030	6159118	5769415
262144	10171517	3691751	2985699
131072	5494585	1792840	1708981
65536	2869688	926506	842725
32768	1399074	478496	428797
16384	703882	307624	211632
8192	358160	172348	141992
4096	242239	138516	544065

Table 16: Results from binary tree summation benchmark with work bias, all measures in ns. Models no data dependency between subtrees and delay.

C | Interview Guide

The interview structure is based on the *scenario-based interview* technique presented in [64]. The scenario in this case is the task being solved by the test participant. The interview is conducted during the test itself. The questions asked are open and may be used to lead the participant towards the relevant problem, but not to lead them to answers. The questions should be designed to clarify that the prototype is being tested and not the participant themselves. With this in mind the following questions are posed.

1. Is it clear how FRP-reactions work?
2. Do you consider FRP advantageous? Why?
3. Would you use this tool in a production environment?
   (a) What if FRP had a performance impact?
   (b) What if FRP had a performance gain?

All the posed questions are open and participants were encouraged to engage in dialogue as a response. Their comments were recorded and at a later date, codified and analysed. The theoretic justification for each question is summarised in the remainder of this chapter.

Question 1 attempts to query the participant's understanding of FRP, without placing the focus on the participant. This can mitigate the pressure experienced by the test participants and can therefore lead to clearer results. Determining how well the FRP paradigm is understood is directly related to answering research question 1 and 3. Furthermore the participants may provide insight into how the FRP system can be made easier to understand.

Question 2 ascertains whether the participants consider the paradigm shift worthwhile. The participants will be exploring both new programming and game development paradigms. Therefore their perception of benefits affords insight into the usability of the paradigm. This question attempts to answer research question 2.

The third is an inquiry into research question 3. The participant is directly asked about, in their opinion, the viability of such an approach in a production environment. Their response is followed up by either question a or b. If the participant believes that the system can be employed in a production environment, then question a is asked, otherwise they are asked question b.

D | Usability Quote Transcriptions

D.1 Participant 1: F# Debrief

`Monitor:` What did you think, especially about the first part, of the test?

`Participant:` Yea, I think it was too bad that I wasn't better at understanding it.

`Monitor:` Don't think about it like that.

`Participant:` Because I can see the idea behind it. The idea behind instead of running Update, which was my first approach anyway. But instead of running Update you just setup event handlers and handle them. If I had a bit better understanding, I could have done it smarter than handle W, handle S, handle A and such. Instead use GetAxis if I had figured that out. In that way it is pretty smart and yields more readable code compared to doing it all in Update, which is what I did here.

`Monitor:` So you felt like it forced you to split code into smaller bites? Is that correct?

`Participant:` Yes at least in the case with player controllers, you need to input parse all the time. There it makes much more sense, because generally when we write software we basically never use Update. Because everything we make is usually event-based. And it gives much more readable code. We are many that write the same, instead of picking up the phone and asking if anyone is there all the time, it is better to have an event that says: the phone is ringing. I like that a lot, I am a big fan and I think you'll find many... However syntax-wise I was quite lost.

`Monitor:` Ok, is that something you could get used to or do think it is terrible to read?

`Participant:` It would take a long time. I prefer, which C# doesn't need to be, very strictly typed, on everything. I can tell immediately what [type] values are but there are a lot of languages that don't adhere to that. That I don't need to specify it is a float or whatever.

`Monitor:` So types are what would be the hardest to get used to?

`Participant:` Yea, I mean it is only a question of time before you get used to not making curly brackets and semicolons. If that makes more or less readable is difficult to say, because if you are used to the other thing it is probably more readable.

`Monitor:` Would you use this kind of framework for a real project? If you didn't have these issues with syntax and event setup.

`Participant:` That is a good question. No I don't think so.

`Monitor:` How come?

`Participant:` I think I am too used to C#, especially. I think in the context of what I would be assigned to do. If someone assigned this to me I would do it, but

typically it is the more popular languages you are assigned. Some kind of Java derivative, Python, C languages, C++ or C#- we never use C today.

`Monitor:` What if it [FRP] offered implicit parallelism, would you use it then?

`Participant:` No I don't think so, not unless I was assigned to use it. If my boss told me to do it, then I would use it - I wouldn't refuse. But I wouldn't use it in my spare time and in my own projects. C# is too well documented and familiar. It would take a while to get into F# and the functions I am familiar with from C#, which also probably exist in F#. I like this event approach, that you just setup, in Start or Awake, just setup what you want. And then just get and deal with the code when you need it instead of checking every time. This could also be setup in C#.

D.2 Participant 4: F# Debrief

`Monitor:` What do you think of the experiment, especially the F# part?

`Participant:` It was very difficult to look at a new language again after such a long time. And I have worked with C# and Unity almost everyday and I am as familiar with C# as is needed in Unity, even though there are still things I want to learn like lambda expressions and such, but haven't needed. Therefore I got totally confused by F# because it is completely different from what I have done before. Therefore I can't tell what it would be like to learn F# first, whether I would find it easier, because I remember when learning C++ and C# with that bracket there and such. It was actually the same thing I was struggling with here, with this F, is called F or F#?

`Monitor:` F#

`Participant:` Alright, so it isn't anything I would ever use, but that is because I have used this since 2011. It is so integrated in me that I just do that, that, and that and I am happy that I am at the point where I don't need to think about syntax errors, because it is usually logic errors I get. In that regard I am too lazy to learn a new language, but that is also because I use this outside of here, in my spare time, where I don't want to spend a lot of time learning a new language...

`Monitor:` What about this way of thinking about the game as events?

`Participant:` I like the way of thinking, because I can see where it is going, where you can avoid a lot of unit testing or not. I don't know what's it called... I mean like older games where errors were unacceptable, but today games just have these kinds of errors because you can just patch it. Whoops, deadlines! Patch, patch, patch, you couldn't do that before. This a language where you really need to know what needs to be in and that variable mustn't change otherwise things fuck up, possibly. It is just there, the package, it works and you can of course make mutable, but the idea is to try to avoid it, right? The idea of events, I like when things are event based, so you avoid having 3000 update loops spinning around, checking whether or not it is supposed to run, which it isn't 69 times out of a 100

times. In that way I can see that it makes sense. It requires, maybe, that you design properly and think this is what I need, of course you can correct the code, but yes that is what I think.

`Monitor:` Do we have any other questions? Yes, what if F# offered a performance gain, such as implicit parallelism?

`Participant:` I need that explained.

`Monitor:` It means that the code you have written now could be run on multiple cores without additional changes.

`Participant:` As is, I can't see the advantage of it because I already use Unity which manages everything. So there isn't as such any thing [requirement] there. I look at as it works there, there and there, but with to my knowledge I can't see an advantage

D.3 Participant 6: F# Debrief

`Monitor:` Well, what do you think of, especially the F# way, of programming games?

`Participant:` I really like functional, but now that I tried programming for games I am not sure. I have talked about wanting to try using functional programming for games. It might provide something and I will proberbly need more time with it, to actually do it. Because this is the first time I'm doing it [F#] with Unity. It would help a lot if I had longer time work with a project.

..

`Monitor:` What advantages and disadvantages can you see with using F#?

`Participant:` Some things are just easier in functional. The task was a tree problem, but I actually think trees are more intuitive due to recursion in functional. Where as you need to make too many hacks in imperative than in functional, where you can make a nice tree traversal. But things like instantiation of objects, when we need 4 players at this position, that is just easier in C#. List handling is also good in functional.

E | Code Examples

```
using System;
using System.Collections.Generic;
using UnityEngine;

public class ArmourBehaviour : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        var Armour = Item.Exercise1();
        var GroupedArmour = Item.Exercise2();

        Solution1(Armour);
        Solution2(GroupedArmour);
    }

    public void Solution1(IEnumerable<Item> Armour)
    {
        int totalAgi = 0;
        int totalStr = 0;
        int totalInt = 0;

        foreach (var item in Armour)
        {
            totalAgi += item.Agility;
            totalStr += item.Strength;
            totalInt += item.Intellect;
        }
        Debug.Log($"Exercise 1\n\t" +
                  $"Agility: {totalAgi}\n\t" +
                  $"Strength: {totalStr}\n\t" +
                  $"Intellect: {totalInt}");
    }
}
```

Listing 32: Armour Graph test case implemented in C#, Part 1.

```
35  public void Solution2(List<Group> GroupedArmour)
36  {
37      GroupedArmour = Item.Exercise2();
38
39      var groupTotals = new List<Tuple<ItemGroup, int, int, int>>();
40
41      foreach (var group in GroupedArmour)
42      {
43          int totalAgi = 0;
44          int totalStr = 0;
45          int totalInt = 0;
46
47          float agiMod = 0.0f;
48          float strMod = 0.0f;
49          float intMod = 0.0f;
50
51          foreach (var item in group.Items)
52          {
53              totalAgi += item.Agility;
54              totalStr += item.Strength;
55              totalInt += item.Intellect;
56
57              agiMod += item.AgilityMod;
58              strMod += item.StrengthMod;
59              intMod += item.IntellectMod;
60          }
61
62          totalAgi = (int) (totalAgi * agiMod);
63          totalStr = (int) (totalStr * strMod);
64          totalInt = (int) (totalInt * intMod);
65
66          groupTotals.Add(new Tuple<ItemGroup, int, int,
            ↪  int>(group.GroupName, totalAgi, totalStr,
            ↪  totalInt));[firstline=35]
67      }
68
69      string res = "Exercise 2\n";
70      foreach (var tuple in groupTotals)
71      {
72          res += $"{tuple.Item1}\n\t" +
73                      $"Agi: {tuple.Item2}\n\t" +
74                      $"Str: {tuple.Item3}\n\t" +
75                      $"Int: {tuple.Item4}\n";
76      }
77      Debug.Log(res);
78  }
79
80  // Update is called once per frame
81  void Update()
82  {
83
84  }
```

Listing 33: Armour Graph test case implemented in C#, Part 2.

```
namespace UnityFunctional
open UnityEngine
open System

type FRP_ArmourBehaviour() =
    inherit FRPBehaviour()

    let sum (triplet1:int*int*int) (triplet2:int*int*int) =
        let (a1, b1, c1) = triplet1.Deconstruct()
        let (a2, b2, c2) = triplet2.Deconstruct()
        (a1+a2,b1+b2,c1+c2)

    let fSum (triplet1:float32*float32*float32)
    ↪  (triplet2:float32*float32*float32) =
        let (a1, b1, c1) = triplet1.Deconstruct()
        let (a2, b2, c2) = triplet2.Deconstruct()
        (a1+a2,b1+b2,c1+c2)

    let totalStats (armour:Item[]) =
        armour
        |> Array.map (fun a -> (a.Agility, a.Intellect,
        ↪  a.Strength))
        |> Array.reduce (fun acc elm ->
            sum acc elm)

    let scaledStats (groups:Group list) =
        groups
        |> List.map (fun g ->
                      List.ofSeq(g.Items)
                      |> List.map (fun i -> (i.Group,(i.AgilityMod,
                      ↪  i.IntellectMod,
                      ↪  i.StrengthMod),(i.Agility, i.Intellect,
                      ↪  i.Strength)))
                      |> List.reduce (fun acc elm ->
                          let (gr,aMod,aStats) = acc.Deconstruct()
                          let (gr,md,stats) = elm.Deconstruct()
                          (gr,(fSum aMod md), (sum aStats stats))))
        |> List.map (fun i ->
            let (grp,(agiMod,intMod,strMod),(agi,int,str)) =
            ↪  i.Deconstruct()
            (grp, float32(agi) * agiMod, float32(int)*intMod,
            ↪  float32(str) * strMod))
```

Listing 34: Armour Graph test case implemented F#, Part 1.

```
36  member this.Start() =
37      let i = ItemStore.AllItems()
38      let (agi, int, str) = totalStats(i)
39      Debug.Log("Agility: " + agi.ToString())
40      Debug.Log("Intellect: " + int.ToString())
41      Debug.Log("Strength: " + str.ToString())
42
43      let g = ItemStore.GroupedItems()
44      let ss = scaledStats(g)
45
46      ss
47      |> List.iter (fun gs ->
48          let (gr, sAgi,sInt,sStr) = gs.Deconstruct()
49          Debug.Log(gr.ToString() + "\n" + String.Join("\n",
            ↪  ["Agility: " + sAgi.ToString(); "Intellect: " +
            ↪  sInt.ToString(); "Strength: "+ sStr.ToString()])))
```

Listing 35: Armour Graph test case implemented in F#, Part 2.

E.1 A Less Viscous Implementation of Unit Management

```
class StateMachine : MonoBehaviour
{
    [...] //Pre-implemented code, such as JoinState

    public List<Shooter> shooterList = new List<Shooter>();
    public List<State> stateList = new List<State>();

    public void JoinState(Shooter shooter, State state)
    {
        shooterList.Add(shooter);
        stateList.Add(state);
    }

    public void TransferState(Shooter shooter, State state)
    {
        if (shooterList.Contains(shooter))
        {
            stateList[shooterList.IndexOf(shooter)] = state;
        }
    }

    private void Update()
    {
        for (int i = 0; i < shooterList.Count; i++)
        {
            switch (stateList[i])
            {
                case State.Attacking:
                    Attack(shooterList[i]);
                    break;
                case State.Fleeing:
                    Flee(shooterList[i]);
                    break;
                case State.Moving:
                default:
                    Move(shooterList[i]);
                    break;
            }
        }
    }

    [...] //methods for each unit state
}
```

Listing 36: A less viscous implementation of the unit management test case.