**Daniel Vlasits**

# Low-Level Programming with Dependent Types

Part II project in Computer Science

**Robinson College**

**2023**

*to*

**Department of Computer Science and Technology**
**University of Cambridge**
**Cambridge**

# DECLARATION

I, Daniel Vlasits of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT.

I am content for my dissertation to be made available to the students and staff of the University.

Signed: Daniel Vlasits

Date: 09/05/2023

# Proforma

Candidate No: **2406F**
Project Title: **Low-Level Programming with Dependent Types**
Examination: **Computer Science Tripos - Part II 2023**
Word-Count: **11,989**
Code Line Count: **2,103**
Project Originator: **Dr Jeremy Yallop**
Project Supervisor: **Dr Jeremy Yallop, Yulong Huang**

### Original Aims of Project

Implement a library (QUIPS library) using Quantitative Type Theory in the programming language Idris 2 which guarantees at compile time that code is safe from pointer bugs. The library allows memory allocation, deallocation and read/write operations to such allocated memory. The QUIPS library should provide these strong safety guarantees without compromising efficiency.

### Work Completed

The QUIPS library was successfully implemented and guarantees that when used result in no pointer bugs. Extensions were also implemented to allow a clean syntax for the libraries use and highly polymorphic arrays and structs were also added. Example algorithms are implemented using the library and evaluated for performance. The performance is shown to be significantly faster than the equivalently safe data structure provided in Idris. This is the first library providing such strong safety guarantees without resorting to array bounds checking or garbage collection.

### Special Difficulties

None.

# Contents

# Chapter 1

# Introduction

Bugs due to pointer misuse are pervasive in programs. At best they cause program crashes, and at worst cause security vulnerabilities. Tooling has been developed to handle such bugs for stack based memory exploits such as buffer overflow, however, much less tooling exists to handle heap-based memory exploits such as use-after-free vulnerabilities[1].

Memory-safe programming languages are usually criticized for being unable to do low-level programming[2], and are therefore slower than other languages for certain tasks. This arises as it proves difficult to verify the safety of programs written using low-level operations.

This project creates a library for fast and safe low-level programming (the library uses Quantitative type theory in Idris for Pointer Safety and is called the *QUIPS library*) that provides important guarantees at compile-time:

1. Eliminating memory leaks by guaranteeing that all pointers allocated are freed
2. Eliminating use-after-free errors
3. Compile-time array-bounds checking to forbid index-out-of-bound errors
4. Forbidding the de-referencing of a pointer with no underlying allocated memory

I show this is possible by utilising two powerful type-systems: *linear and dependent types.*

Linear type systems ensure objects are used exactly once allowing for reasoning about resource usage. With dependent types, data types can be defined more precisely than algebraic data types. Dependent types allow definitions to depend on values. For example, instead of "List of Integers" you could more accurately describe a list as "List of five integers", in this manner the type includes the integer value five. This allows data layouts to be described more flexibly and have the type checker validate more properties of code such as to guarantee that when a list indexing occurs it will always be within the bounds of the array.

[1] `https://encyclopedia.kaspersky.com/glossary/use-after-free/`

[2] `https://www.howtogeek.com/devops/what-is-garbage-collection-and-how-does-it-affect-your-programs-performance`

Quantitative Type Theory (QTT) combines these two systems, allowing for dependent types whilst also optionally restricting the usage of certain values. Idris 2 is a programming language which has been built with Quantitative Type Theory at its core, therefore supporting both linear and dependent types. I have used Idris 2 to develop the purpose-built QUIPS library.

I claim the QUIPS library is the first of its kind to make such strong safety guarantees to the programmer whilst retaining low-level operations. The QUIPS library allows the the user to program with pointers and arrays, whilst guaranteeing the absence of the above run time errors. Importantly, safety is guaranteed without the need for garbage collection or array bounds checking.

# Chapter 2

# Preparation

This chapter first covers the problems the QUIPS library aims to solve, and their importance, with the goal of consolidating the requirements. I then move on to looking at the theory behind the techniques used. Which includes:

1. Linear, dependent and Ouantitative Type Theory
2. The importance of monads in functional programming languages
3. The need to provide proofs to a compiler

The chapter finishes (2.9) looking at why Idris 2 is the language of choice and other engineering decisions made.

## 2.1 The Problem

Here I look at the pointer-based exploits still commonplace that rely on the heap. This section describes the various pointer based bugs and why they are worth addressing.

Half of all known exploitable bugs in Chrome are reportedly **use-after-free** bugs[1]. A **use-after-free** bug occurs when a pointer referencing non-allocated memory is accessed. This can lead to undefined behaviour and security vulnerabilities.

Another common bug occurs when freeing the same pointer twice (**double-free**). The double-free bug can lead to *memory leaks* and heap exploitation, potentially giving an adversary access to forbidden memory locations. An exploit relying on this vulnerability was found for Whatsapp[2] in 2019 and more recently for OpenSSH[3] in 2023. Both exploits could allow a remote attacker connection to devices, and the stealing of personal files.

[1] `https://security.googleblog.com/2022/09/use-after-freedom-miracleptr.html`

[2] `https://gbhackers.com/whatsapp-double-free-vulnerability/`

[3] `https://blog.qualys.com/vulnerabilities-threat-research/2023/02/03/cve-2023-25136-pre-auth-double-free-vulnerability-in-openssh-server-9-1`

In addition to the above free-related bugs **indexing-out-of-bounds** in an array is a common cause of segmentation faults and undefined behaviour in low-level systems languages. To protect against this, higher-level languages perform runtime validation checks on every array-index (array bounds checking), halting with an error if necessary. This incurs a performance overhead compared to systems languages such as C. A trade off therefore takes place resulting in compromising either speed or safety.

**Memory leaks** occur when memory is not deallocated after use. Memory leaks in long running programs result in memory fragmentation and pages filling up with not needed garbage. This can lead to the system slowing down due to thrashing[4] or outright failing.

I address these bugs using the type system available in Idris 2 by developing a library which maps closely onto C-style array functionality. Previously, a compile-time solution which mitigates all the bugs mentioned without relying on array bounds checking or garbage collection does not exist.

## 2.2 Requirements Analysis

The QUIPS library was designed to satisfy the following requirements:

The QUIPS library allows a programmer to write code, requiring pointers and arrays. The QUIPS library is not garbage collected and requires manual free commands when memory is no longer needed.

The QUIPS programming library should provide functions for allocating, reading, writing and freeing of pointers and arrays, which the programmer can then use as if writing in C.

The type signatures in place should guarantee that if the program passes the type-checker, the bugs mentioned in the previous section cannot occur at run time.

The solution should also allow for better performance than current data-structures available in functional languages.

## 2.3 Linear Types

Compile-time verification is necessary to guarantee pointers are used safely without incurring a performance overhead. This is precisely what a type system aims to do: to reduce the chances of error in program execution by performing a syntactic check on the code at compile time. Ideally, type-checking would only reject incorrect programs. However, as this is in general undecidable, type checkers necessarily overestimate the number of invalid programs, a trade-off worth taking when the goal is to write safe code.

In the following section, I exhibit a type system that can control the usage of pointers in a restricted manner, for example, to guarantee a pointer must be freed. However, type systems such as for OCaml (or those taught in IB semantics) are unconcerned with the number of times

[4] `https://en.wikipedia.org/wiki/Thrashing_(computer_science)`

a value is used, and focus instead on the legitimate usage of values within the context. When using pointers, safety requires that a guarantee is given to ensure pointers are free'd exactly once and have zero uses afterwards. Therefore, a more precise formalisation is required to allow for the controlling of such a resource at the type-level.

This idea of controlling the number of times certain variables can be used is the intuition for linear types[7]. I initially give a high-level overview of linear functions, the key addition made by linear types.

Linear types introduce a new type-constructor ($\multimap$):

$$f : A \multimap B \tag{2.1}$$

From now on, I call *ML-style types* the type systems employed by languages such as Haskell, ML and OCaml. The definition above is almost equivalent to the ML-style function type-constructor ($\rightarrow$), however, for the body of $f$ to pass the linear type-checker it must be linear in its argument. Informally, a function satisfies this condition if its argument is used exactly once in computing the return value.

$$\begin{aligned}
&\text{nonLinear1 } x = x + x \\
&\text{nonLinear2 } x = \text{let } y = x + 2 \text{ in } 5 \\
&\text{linear1 } x = (x, 5) \\
&\text{linear2 } x = \text{let } y = x + 3 \text{ in } y
\end{aligned}$$

*nonLinear1* uses $x$ twice, whereas *nonLinear2* ends up dropping $y$ which was computed using $x$ and therefore argument $x$ is not used exactly once in the computation of the return value.

Consider the type-signature below:

$$\text{applyOnce} : (a \multimap a) \multimap a \multimap a \tag{2.2}$$

The types enforce *applyOnce* must use its first and second argument exactly once. Due to this constraint, all implementations of *applyOnce* must be semantically equivalent under a linear type system. *applyOnce* must apply its first argument to the second and return the output. Applying the function more than once to the second argument would result in a type error. Similarly, discarding the function and solely returning the second argument would also result in a type error.

Linearity is vital in the QUIPS library for constraining function implementations to only those which do not abuse pointers.

Next I describe the implementation details of linear types by considering basic rules afforded to most type systems which allow for the reuse of values. The removal of some of these rules creates linear types. There are three rules permitted in most commonplace type systems: *weakening*, *contraction* and *exchange*. They are named *structural rules* and have the following definitions.

- A type system allows for *weakening* when elements from the context may be discarded:

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau} \tag{2.3}$$

  To show $e : \tau$, *weakening* allows the removal of the assumption $x : \tau'$ from the context.

- A type system allows for *contraction* when elements from the context may be reused:

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \tau}{\Gamma, x : \tau \vdash [x/x_1][x/x_2]e : \tau} \tag{2.4}$$

  If the variable $x$ occurs multiple times in $e$, contraction allows the renaming of each occurrence in the term, and giving each occurrence its own assumption in the context. Effectively, this allows assumptions from the context to be used any number of times to type terms.

- Finally, a type system allows for *exchange* when elements of the context may be re-ordered:

$$\frac{\Gamma_1, x : \tau_1, y : \tau_2, \Gamma_2 \vdash e : \tau}{\Gamma_1, y : \tau_2, x : \tau_1, \Gamma_2 \vdash e : \tau} \tag{2.5}$$

  When the context contains the assumptions $y : \tau_2, x : \tau_1$, *contraction* allows re-ordering the assumptions to give $x : \tau_1, y : \tau_2$.

A type-system which allows all three rules is called *intuitionistic*[6]. ML-Style type systems are *intuitionistic.*

To create a *substructural* type-system one or more of the above rules are disallowed. The removal of the *weakening* and *contraction* rules whilst retaining *exchange* results in a linear type system.

To typecheck a term within a linear type system the context must be fully consumed at the end of each branch in the well-typedness proof. If type-checking a term requires recursively checking more than one branch the context must be equally split amongst all branches. This is written:

$$\frac{\triangle_1 \vdash M : S \to T \quad \triangle_2 \vdash N : S}{\triangle_1, \triangle_2 \vdash MN : T} \tag{2.6}$$

This rule represents the requirements needed to linearly type check function application. The context must be split into two, one half is used to validate that $M$ is a function of the correct type and the other that $N$ is the correct argument.

The removal of the *weakening* and *contraction* structural rules results in a type system which requires every value to be used exactly once. Importantly, the program may not discard values or use them multiple times. On its own this would be a very restrictive type system, however combined with an intuitionistic type system it allows for optionally controlling the resource usage of some values whilst leaving others free.

It is therefore the combination of linear and intuitionistic types which leads to the most flexibility in the QUIPS library. This is implemented by allowing two type-constructors for functions

$$\begin{aligned}\text{linear\_function} &: a \multimap b\\ \text{intuitionistic\_function} &: a \to b\end{aligned}$$

An *intuitionistic_function* should be type-checked exactly as would be normally expected. *linear_function* should be type-checked to guarantee the function consumes its input exactly once.

Combining intuitionistic types with linear types can be done by adding a judgement which splits the typing context into two parts:

$$\Gamma|\Delta \vdash M : U \tag{2.7}$$

Where the first context ($\Gamma$) allows all three structural rules, denoting the intuitionistic assumptions. The second context ($\Delta$) disallows weakening and contraction.

## 2.4 Dependent Types

Using only safe functions as building blocks, an ML-style type system can eliminate all run time errors. However, this often requires using datatypes such as the *Option* or *Either* data type. The canonical example of this is the *head* operation, which returns the first element of a list and usually has the type signature below. This results in the function throwing a run time exception when passed an empty list:

```
head : List a -> a
```

The run time exception can lead to a crash and occurs as the type-signature is not able to constrain the allowed function arguments to non-empty lists. This section describes an extension to the standard type system to allow for putting additional constraints on the possible underlying representations of types.

A common solution to the problem of *head* being a partial function is to return an *Option* type as below, e.g. in Idris:

```
headSafe : List a -> Maybe a
headSafe [] = Nothing
headSafe (x::_) = Just x
```

*headSafe* will not cause a crash to be thrown at runtime, however, this is just a type checked exception to the program. Usually, when the programmer calls the *head* function they expect the list to be non-empty and therefore ideally, the programmer may want to guarantee that *headSafe* is never called on an empty list as this likely implies a bug in the algorithm.

Currently, the type of a list (*List a*) reveals the type of the elements contained in the list, however the type says nothing about the number of elements in the list. The idea is to enrich the type system to be able to write *List a size* where size is a natural number representing the size of the array. The type now states the type of the elements within the list alongside its size. Here I start with the *List a* data type and work towards representing the *List a size* datatype.

The ML-Style list type is defined as follows:

```
data List a = NIL | Cons a (List a)
```

This defines one type constructor *List* and two data-constructors *NIL* and *Cons*. Consider the types of the three new functions this definition gives:

```
List : Type -> Type
NIL : List a
Cons : a -> List a -> List a
```

I have now started treating *Type* as a Type in its own right, therefore, the type constructor *List* is in fact just a function which takes in a type such as *Char* or *Int* and produces a new type *List Char*. *NIL* represents the empty list, and the remaining constructor *Cons* takes in values of type *a* and produces a value of type *List a*.

To allow for more complex data types a richer syntax is used for defining data types; writing the *List* type using this syntax gives:

```
data List : Type -> Type where
    NIL : List a
    Cons : a -> List a -> List a
```

In Idris 2, this is an equivalent formalisation of the ML-Style definition.

Before proceeding it is necessary to introduce a simple important data type, the natural numbers:

```
data Nat = Z | S Nat
```

A natural number is either *Z* (zero) or a successor to a natural number *S Nat*. This data type allows for the usage of non-negative integers. In this manner it constrains the possible underlying values of the type to non-negative numbers.

The definition for *List* can now be extended in the following manner to be able to represent *List a size*; to avoid name clashes I call this type *Vect*:

```
data Vect : Type -> Nat -> Type where
    NIL : Vect a Z
    Cons : a -> Vect a size -> Vect a (S size)
```

The function *Vect* now requires a type *and* an integer to be a new valid type. *NIL* now has the new type *Vect a Z* to represent it is an empty vector. The *Cons* function prepends a new element onto a vector returning another vector one larger in size. *Vect* is said to be "indexed" by the natural numbers, representing the size of the underlying list.

Consider the append function for basic lists, the type signature promises the type of the output list matches the types of the incoming lists:

```
appendList : List a -> List a -> List a
```

However, dependent types[4] are rich enough such that the type signature can also guarantee the size of the output is the sum of the two input sizes:

```
appendVect : Vect a n -> Vect a m -> Vect a (n + m)
```

Dependent types also allow the output type of functions to depend on the values of the input. For example, consider the function which takes in an integer and creates a vector of the corresponding size (with every element being an empty string):

```
create : (x : Nat) -> Vect String x
```

The output type *Vect String* $x$ depends on the *value* of the input $x$. The equivalence between the $x$ in *Vect String* $x$ and the input argument $x$ is expressed by labelling the arguments in the type signature if they are to be used later in other types.

It is now possible to perform computation in the type-signature to determine the type, as has already been shown with the addition computation in the append function: $n + m$. To illustrate further it is also possible to write functions such as func, which can return a *String* or *Int* type depending on the value of the input argument:

```
determineType : Bool -> Type
determineType False = String
determineType True = Int

func : (b : Bool) -> determineType b
func False = "False"
func True = 1
```

As the size of a *Vect* is now represented at compile time it is now possible to define an indexing function into a *Vect* which accepts only natural numbers smaller than the size of the array. The *Fin* datatype allows for describing sets of natural numbers smaller than a given value.

```
data Fin : Nat -> Type where
   FZ : Fin (S k)
   FS : Fin k -> Fin (S k)
```

This signature introduces a family of types, a certain type *Fin x*, represents the set of all natural numbers smaller than $x$. The definition describes the two ways in which a value of type *Fin (S k)* can be created. *FZ* representing zero is smaller than all positive integers, and is therefore a value of type *Fin (S k)*. Secondly if we have a value of type *Fin k* (i.e. a number less than $k$), then this number is also less than *S k*.

The construction of the value 3 of type *Fin 6* is the following:

```
threeFinSix : Fin 6
threeFinSix = FS (FS (FS (FZ : Fin 3)))
```

On the other hand, it is not possible to construct the value 3 of type *Fin 2*. The possible underlying values of *Fin x* are all natural numbers less than $x$.

This type could now be used when we want a function to take in natural numbers of a maximal certain size. For example, a function only accepting numbers smaller than 100 would be written.

```
noLargerThan100 : (Fin 100) -> ...
```

In a similar manner I can now write a safe indexing function as follows:

```
safeIndex : (Fin size) -> Vect size a -> a
```

At its core a dependent type is one where a type can depend on a value. They allow programmers to easily create types which further restrain the set of possible values of types.

## 2.5 Quantitative Type Theory

Quantitative Type Theory[2] was developed to combine linear and dependent types into a single type system. Quantitative Type Theory allows for very expressive and precise type signatures.

Within Quantitative Type Theory, arguments can be given three usage constraints: 0, 1 and $\omega$. The usage constraint 0 is a guarantee from the compiler that the value is not used at runtime. The usage 1 is identical to linear usage as described in the section on linear types (section 2.3) Finally, $\omega$ allows unrestricted usage as in a standard type system.

Here I will highlight the relevant syntax[1] of Quantitative Type Theory within Idris as it will be used throughout the implementation chapter.

Equation 2.1 ($f : A \multimap B$) is encoded by:

```
f : (1 _ : A) -> B
```

Here the function $f$ is a linear function as described in the Linear Type section. It is natural to use (_) if the value is not referenced elsewhere; using the value elsewhere would constitute a dependently typed function such as *dependentLinearFunc*:

```
dependentLinearFunc : (1 mem : A) -> B mem
```

Here the type of the output (*B mem*) depends on the value of the input *and* the input is used linearly in the function. Effectively, the type signature operates as a let binding of the form:

```
let mem = usage 1 of type A in
                mem -> B mem
```

If linearity is not required, the optional usage annotation is dropped:

```
intuitionisticFunc : A -> B
dependentFunc : (mem : A) -> B mem
```

## 2.6 Monads

Functional programming languages involve pure computations. This makes code easier to parallelise and optimise. However, it has the downside of making it difficult to sequence computations for Input/Output operations. The developers of Haskell discovered monads[8] as the abstraction to deal with this, and developed a clean syntax called *do-notation* for their use. Monads allow side-effecting code to be structured in a predictable and manageable fashion.

Here is a simple example of a program written to handle I/O using do-notation syntax:

```
main : IO ()
main = do
        x <- getLine
        print x
```

The de-sugaring of this code without do-notation looks as follows:

```
main = getLine >>= print
```

*getLine* has type *IO String*; this type signals that *getLine* performs Input/Output and returns a *String*. This returned value can be accessed using the bind operator (>>=) which takes this string and feeds it to another operation, in this case print which takes in a *String* and returns a value of type *IO ()*.

For the *IO* monad, do-notation can be directly mapped onto an imperative program, a line is fetched from stdin, assigned to the variable $x$ and then printed back. A common misconception is that this function is no longer pure; however, *main* is in fact still a pure function that returns a list of instructions to be run. These instructions are then run by the Idris interpreter when it is told to execute the instructions. In this case the instructions would be to read from stdin, then take the value and print it.

The list of instructions to be executed can be viewed when evaluating the main function above in the Idris 2 REPL, when doing so an abstract syntax tree is presented of the instructions to be run; this tree is of type *IO ()*. It is only when the Idris interpreter is told to execute the function that it will actually fetch a line from the user and provide output.

This *do-notation* syntax allows for sequencing instructions in pure functional languages.

## 2.7 Curry-Howard Correspondence

The Curry-Howard correspondence shows a relationship between proofs and programs.[9]

Consider proving the tautology:

$$a \implies (b \implies a) \tag{2.8}$$

The proof takes the following structure:

1. Required to Prove (RTP): $a \implies (b \implies a)$
2. Assume $a$, RTP: $b \implies a$
3. Assume $b$, RTP: a
4. Done by assumption 2

Next consider constructing a typing derivation for the following term:

$$\lambda x.\lambda y.x : \mathbf{A} \to (\mathbf{B} \to \mathbf{A}) \tag{2.9}$$

$$\cfrac{\cfrac{\cfrac{}{x\text{: }\mathbf{A}, y\text{: }\mathbf{B} \vdash x : \mathbf{A}}}{x\text{: }\mathbf{A} \vdash \lambda y.x : \mathbf{B} \to \mathbf{A}}}{\vdash \lambda x.\lambda y.x : \mathbf{A} \to (\mathbf{B} \to \mathbf{A})}$$

The verification of the type (2.9) provides a direct mapping onto the proof (2.8). The assumptions from the proof $(a, b)$ are placed in the context $(x : A, y : B)$; similarly, at each stage, for example step 2, the type of the term being type-checked $(\mathbf{B} \to \mathbf{A})$ is equivalent to what is still left to be proven $(b \implies a)$. In this manner the program provides a blueprint for the compiler to create the proof.

The Curry-Howard correspondence allows programmers to prove formulae by providing function implementations satisfying certain type signatures. In the above example, implication ( $\implies$ ) corresponds to the function type ($\to$). Similar correspondences can be made for other mathematical constructs:

| **Programming** | **Logic** |
|---|---|
| Functions | Implication |
| Tuples | Conjunction |
| Tagged Union | Disjunction |

Type systems therefore, have two major roles. Not only do they serve an important role in programming languages, but are also a fundamental construct from logic.

## 2.8 Proofs using dependent types

The previous section described the correspondence between propositional logic and an ML-Style type system. Here I present more powerful correspondences between dependent types and logic, allowing for proofs involving universal quantification.

I showed in the dependent types section how the *Fin* datastructure could be used to guarantee safe indexing. Here I present another way of guaranteeing safe indexing, by passing around proofs.

```
data LTE : Nat -> Nat -> Type where
    LTZero : LTE 0 x
    LTESucc : LTE x y -> LTE (S x) (S y)
```

The *LTE* data structure captures all pairs of valid integers $x, y$, such that $x \leq y$. For example, to construct a value of type *LTE 2 4* we write:

```
twoLessThanFour : LTE 2 4
twoLessThanFour = LTESucc (LTESucc (LTZero : LTE 0 2))
```

We have the correspondence that there exists a value of type *LTE x y* if and only if $x \leq y$. As *twoLessThanFour* provides a value of type *LTE 2 4* the program therefore represents a proof that $2 \leq 4$, which allows for re-writing the indexing function in a new manner:

```
index : (x : Nat) -> LTE (S x) size -> Vect a size -> a
```

The *index* function now takes in any natural number but also requires a second argument of type *LTE (S x) size*, which encodes the proof that the index is of a size valid for the vector.

In Idris this signature can be written as follows:

```
index : (x : Nat) -> {auto prf : LTE (S x) size} -> Vect a size -> a

main =
    ....  index loc (array : Vect large String) .....
```

The curly braces and the auto keyword ask the Idris compiler to find a value of the corresponding type by itself. During type-checking, the compiler now tries to automatically construct a value of type *LTE (S x) size* i.e. to deduce a proof of its own when the function is called. However, as in general generating mathematical proofs is a non-decidable problem, the compiler may tell the programmer it has failed to come up with a valid value as required, and the programmer must then manually provide this value (proof).

Trying to prove $\forall x : x \leq (x+2)$ would now be equivalent to writing a program with the type signature below.

```
required_to_prove : LTE x (S (S x))
```

In Idris 2 the above syntax implicitly quantifies over all $x$ and shows dependent types are strong enough to represent universal quantification. Writing the quantification explicitly looks as below:

```
    required_to_prove_explicit : (x : Nat) -> LTE x (S (S x))
```

## 2.9 Tool Selection

Here I discuss the requirements of the programming language necessary to implement the QUIPS library. I then discuss the available programming languages meeting the requirements, finally explaining the choice of Idris 2 and the relevant rationale.

To develop the QUIPS library a programming language is needed which implements both linear and dependent types. To use dependent types effectively it is important the language has good facilities for proving necessary propositions and programming with high-level constructs such as monads.

A foreign function interface (FFI) allows for calling functions in other languages. The language of choice must have a foreign function interface to a low-level language which can assign and free pointers. This allows for pointer operation and manipulation which would otherwise not be possible in a general high-level functional programming language.

There are various languages which implement dependent types such as Agda and Coq. However, they are generally geared towards theorem proving rather than software engineering, lacking interoperability with systems libraries and high level functional programming constructs such as do-notation and type-classes. Furthermore, the only FFI Agda has access to is for Haskell. As Haskell has an FFI this could be combined but would lead to cumbersome code.

Haskell has added the linear type constructor[5] ($\multimap$) as a language extension, however, the dependent types available in Haskell are limited mostly to generalised algebraic datatypes[5] (a limited form of dependent types), and expressing more complex dependently typed functions is unnatural and significantly more difficult than in a language built from the ground up with dependent types in mind.

As described earlier, Quantitative Type Theory was recently invented to combine linear type systems with dependent types, and was implemented as the core of the programming language Idris 2 and claims to be the first full-scale programming language to do so[2]. Idris 2 has a very clean FFI interface with C which allows us to allocate and free memory at a low-level. These two features of Idris 2 make it the language of choice.

The key software engineering practice employed for this project is Type-Driven-Development[3]. The approach focuses on making types a fundamental building block of your code, which is used to then write the implementation. Using types in this manner, with a language as expressive as Idris allows the compiler to provide guidance on your code as you program.

For the model of development I used the spiral model[6]. I had to repeatedly plan potential solutions and then engineer them. These solutions would sometimes fall short when evaluated and would illuminate a new and different approach to solving the problem. I went through this process many times, reflected in the total lines of code I wrote being 5000 lines, but only 2000 made it into the final repository.

[5] `https://en.wikipedia.org/wiki/Generalized_algebraic_data_type`
[6] `https://www.geeksforgeeks.org/software-engineering-spiral-model/`

# Chapter 3

# Implementation

This section first develops functions to guarantee the runtime safety of pointers to a single mutable variable at compile time, simplifying the syntax with monadic-style programming (3.2, 3.3). The pointer functions are then extended to allow for programmers to operate on arrays (3.5). Finally, we move on to some extensions to handle polymorphic arrays and C-Style structs (3.7, 3.8).

## 3.1 Repository Overview

The layout of the code repository is as follows:

```
Low-Level Programming with Dependent Types
├── QUIPS
│   ├── QuipsCore.idr
│   ├── PairTypes.idr
│   ├── StructExtensions.idr
│   ├── array_write.c
│   ├── pointer_functions.c
│   └── struct_extensions.c
├── Algorithms
│   ├── BinarySearch.idr
│   └── Heap.idr
└── Evaluation
    ├── BinarySearchProfile.idr
    ├── HeapProfile.idr
    ├── IndexingProfile.idr
    ├── profileBinarySearch.py
    ├── profileHeapSort.py
    ├── profileIndexing.py
    ├── simpleErrors.idr
    └── HaskellArray.hs
```

The *QUIPS* folder contains the code for the QUIPS library implementation, this is a complete module by itself which can be packaged and published as an Idris library. The *QuipsCore.idr* contains all functions from the implementation chapter excluding section 3.8, which is included in the *StructExtension.idr* file.

The *Algorithms* folder contains implementations for binary search and a heap data structure, built using the QUIPS library, this folder could similarly be released with a dependency on the QUIPS library. Finally, the *Evaluation* folder contains all the code necessary to demonstrate the QUIPS library catching pointer bugs and to evaluate the performance of the QUIPS library.

## 3.2 Basic Pointer Operations

We first focus on exposing a set of functions that allows programmers to manipulate pointers to a mutable store within Idris. The functions must guarantee that if a program successfully type-checks there can be no pointer bugs at runtime. The four functions required to enable this are allocating, writing, reading and freeing pointers.

The foreign function interface provides a type *AnyPtr* which allows for passing a pointer within Idris code. Using this interface in a naive manner results in the following type signatures for the necessary functions:

```
allocUnsafe: IO AnyPtr
writeUnsafe: Int -> AnyPtr -> IO ()
readUnsafe : AnyPtr -> IO Int
freeUnsafe : AnyPtr -> IO ()
```

As these functions mutate state destructively, they take place within the *IO monad*. The functions have simple implementations in C, for example *read* has the form:

```
int read_int_pointer(void* ptr) { return *(int *)ptr}
```

The above functions allow for allocating and freeing pointers in Idris, however provide no guarantees of safety. For example, there is no enforcing of a pointer being freed only once after use.

The following code demonstrates this issue, it type checks, however frees the same pointer twice at run time:

```
main : IO ()
main = do
        myPointer <- allocUnsafe
        freeUnsafe myPointer
        freeUnsafe myPointer
```

For safety, there needs to be more control placed over pointer usage. Re-framing the problem using continuation passing style[1], *alloc* should take in a safe computation on a pointer and proceed to execute said computation on a fresh pointer. To guarantee pointer safety at compile time a type signature is necessary to capture the idea of a safe computation.

The *QUIPS library* contains a set of functions to guarantee that any function constructed with linear usage in the pointer argument is safe. The intuition here is to force the pointer to be used by the programmer and forbid discarding pointers in their code. This guarantees that the pointed to memory is freed.

Therefore we give *alloc* and *free* the type signatures:

[1]https://en.wikipedia.org/wiki/Continuation-passing_style

```
alloc : ((1 _ : AnyPtr) -> IO Int) -> IO Int
free : (1 _ : AnyPtr) -> IO Int
```

To allow the returning of the value inside a pointer, the *free* function will free the pointer and return a copy of the last integer stored at the pointers location. As only Integer pointers are being considered the computations return an *IO Int*.

We proceed using continuation passing style to define the remaining two functions in a similar manner:

```
write : ((1 _ : AnyPtr) -> IO Int) -> Int -> (1 _ : AnyPtr) -> IO Int
read :  (Int -> (1 _ : AnyPtr) -> IO Int) -> (1 _ : AnyPtr) -> IO Int
```

The implementation of *write* is the following:

1. Take in a computation on a pointer (**k**) along with an integer (**i**) and a pointer (**p**)
2. Write **i** into **p**
3. Run **k** on **p**, returning the result in the *IO monad*

For *read* the computation taken in represents one which requires a copy of the dereferenced value of the pointer alongside the actual pointer. The function can then use both values to compute a result. *read* has the following implementation:

1. Take in a pointer computation (**k**) requiring an integer
2. Take in a pointer (**p**)
3. Dereference **p** to obtain a value (**v**)
4. Execute **k** passing in **v**, alongside **p**

Once again, the idea here is to create functions which either consume pointers, or require another safe computation on a pointer as a continuation. In this manner a proof by induction shows that all computations created by combining these functions cannot cause a pointer bug at run time. This proof will be presented in the evaluation chapter, section 4.1.

Valid programs can be written by composing these functions, for example the below two functions write the value of 10, read the value and double it.

```
alloc . flip write 10 . read . (\cont, x => cont (x*2)) . write $ free

alloc (flip write 10 (read ((\cont, x => cont (x*2)) (write free))))
```

Here (.) represents the function composition operator, and $ will apply the function value on the left to the value on the right, the second line implements the same value as the first line just explicitly with parentheses.

The final stage necessary for safety is wrapping the *AnyPtr* type into a new private data type such that the constructor is not exposed outside of the QUIPS library. This prevents the programmer from writing code to modify the pointer outside of the given functions, limiting its scope:

```
private
data SafePointer = ConstructSafePointer AnyPtr
```

The functions all require continuations, which produce an IO Int. Therefore, as long as the *AnyPtr* is wrapped in a type that makes it inaccessible to the programmer, it cannot escape the continuations and therefore must be freed exactly once.

There is a subtlety here: three of the four functions, do not implement the type they claim. Consider *read*:

```
read :  (Int -> (1 _ : AnyPtr) -> IO Int) -> (1 _ : AnyPtr) -> IO Int
```

The pointer which *read* takes in is used more than once, it is dereferenced and both the value and pointer are passed to the continuation, breaking linearity. However, *read* must have this type signature otherwise the continuation produced by *read* would not be linear in the pointer and therefore the compiler would not let the programmer pass *read* when partially applied to its first argument to *alloc*.

The function *assert_linear* is used to overrule the compiler and allow *read* to have the above type signature. This shows an underlying assumption made that the implementations of the functions are safe. However, the implementations are short enough that this is easy to see upon careful inspection of the code.

```
assert_linear : (a -> b) -> ((1 _ a) -> b)
```

I verify the safety of the implementations of the four functions (*alloc, read, write and free*) manually by checking the limited amount of code each use. The Idris type checker then uses the types of these functions to guarantee a program written using the QUIPS library remains safe. In this manner the QUIPS library isolates a limited amount of reusable code (the QUIPS library) which must be verified for safety once, and can then be used safely by any user of the QUIPS library, which is then automatically verified for safety by the compiler.

## 3.3 Linear Monad

Programming using continuation-passing-style as shown above is challenging and unnatural for pointer-based programming. To make the QUIPS library more usable we introduce a cleaner syntax for programmers. Here the linear monad is introduced to enable a more imperative style of coding with pointers. At a high level the monad allows programmers to return pointers with varying resource usage constraints.

One usage of monads is to eliminate the need for continuation passing style. Recall the type-signature of bind for IO:

```
bind : IO SafePointer -> (SafePointer -> IO SafePointer) -> IO SafePointer
```

*bind* has a close resemblance to the previous functions (*alloc, read, write*): the first argument represents the pointer to be used and the second the continuation to be performed.

The above type signature has two issues, the continuation does not respect the linearity of the pointer, and the output pointer could be used by the programmer arbitrarily many times.

A new monad is needed which allows the tracking of the usage constraints of the value it contains. The type of *bind* then enforces these usage constraints in the continuations.

```
data Usage = Linear | Unrestricted
data L : (usage : Usage) -> Type -> Type where ...
```

Here a value of **L** contains information at the type level stating the usage required of its contents. Below are two examples of valid types:

```
L Linear SafePointer
L Unrestricted Integer
```

The first contains a pointer to be consumed exactly once, the second contains an integer with unrestricted usage.

The type of the bind operator must now vary depending on whether a value of **Usage** is `Linear` or `Unrestricted`, this leads to the following alternative type signatures for bind.

```
L Linear a -> ((1 _ : a) -> L usage b) -> L usage b
L Unrestricted a -> (a -> L usage b) -> L usage b
```

The above states that if the usage is linear the continuation must be linear in its argument, otherwise it may be unrestricted. This can be considered a form of overloading for the type of the bind operator.

The implementation relies on defining a function which calculates the type of the continuation:

```
contType : (u_inital : Usage) -> (u_output : Usage) -> Type -> Type -> Type
contType Linear u_output a b = (1 _ : a) -> L u_output b
contType Unrestricted u_output a b = a -> L u_output b
```

This helper function is used to define the type of bind:

```
bind : L usage_in a -> contType usage_in usage_out
    -> L usage_out b
```

To eventually execute a computation represented by a linear monad, a function is needed which transforms it into **IO a**. However, extraction of this value is only allowed if the final value of the monadic operations has unrestricted usage, otherwise the usage restriction on the element would be broken:

```
runLin : L Unrestricted a -> IO a
```

Using **L** allows for the following types of *alloc*, *read* and *write*:

```
allocLin : L Linear SafePointer
writeLin : Int -> (1 _ : SafePointer) -> L Linear SafePointer
freeLin : (1 _ : SafePointer) -> L Unrestricted ()
```

*Bind* hides the complexity of the linear continuations using polymorphism which allows for cleaner type signatures. *allocLin* now returns a pointer which must be used exactly once. *Free* consumes a pointer returning *unit* with unrestricted usage. *Write* updates the value of the pointer and returns another pointer requiring linear usage. The type signatures allow the programmer to write to a pointer as many times as they like but at the end the pointer must be consumed using *free*.

This allows a programmer to write the following program:

```
main = runLin $ do
    myPointer <- allocLin
    myPointer' <- writeLin 10 myPointer
    myPointer'' <- writeLin 20 myPointer'
    freeLin myPointer''
```

Under the hood every *myPointer* is the same pointer, however we distinguish them here to show how

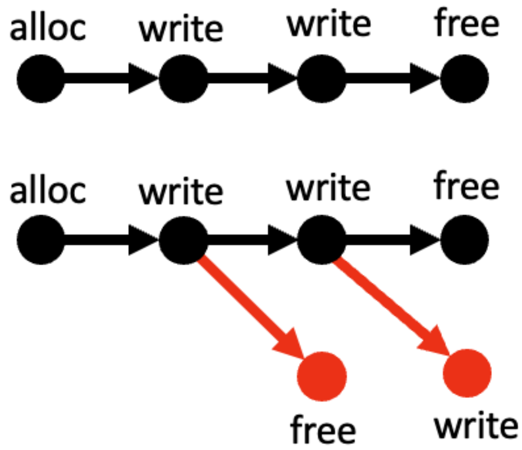


Figure 3.1: Diagram of pointer usage

the type-checker would validate this code. The compiler verifies each *myPointer* is used exactly once.

Writing to a pointer returns a new pointer with linear usage, in this manner the pointer is threaded through a series of operations and eventually freed in a perfect line, to guarantee that pointer bugs cannot occur. This idea of a perfect line is represented in Figure 3.1

Variable shadowing allows the previous program to be re-written to represent the equality of the underlying pointers:

```
main = do
    myPointer <- allocLin
    myPointer <- writeLin myPointer 10
    myPointer <- writeLin myPointer 20
    freeLin myPointer
```

*Read* must now return two values, the pointer (**p**) and the integer (**i**) at address **p**. Returning a tuple ***(i, p)*** would require the same usage of **i** and **p** which is not the intended outcome. The pointer **p** should have linear usage, however, **i** should be unrestricted in its usage.

To handle this we introduce a new data-structure, a tuple which is unrestricted in its first argument and linear in the second:

```
data SemiLinPair : Type -> Type -> Type where
  (*?) : a -> (1 _ : b) -> LinPair a b
```

Here the usage annotations are used within the type declaration, the declaration guarantees that when a *SemiLinPair* must be used linearly and is deconstructed into its elements, the first will have unrestricted usage, whereas the second must require linear usage.

If no usage annotations are provided the usage defaults to unrestricted usage in all arguments.

We can now define the type signature of *readLin*:

```
readLin : (1 _ : SafePointer) -> L Linear (SemiLinPair Int SafePointer)
```

The functions written in the Basic Pointer Operations chapter (3.2) can now be re-written:

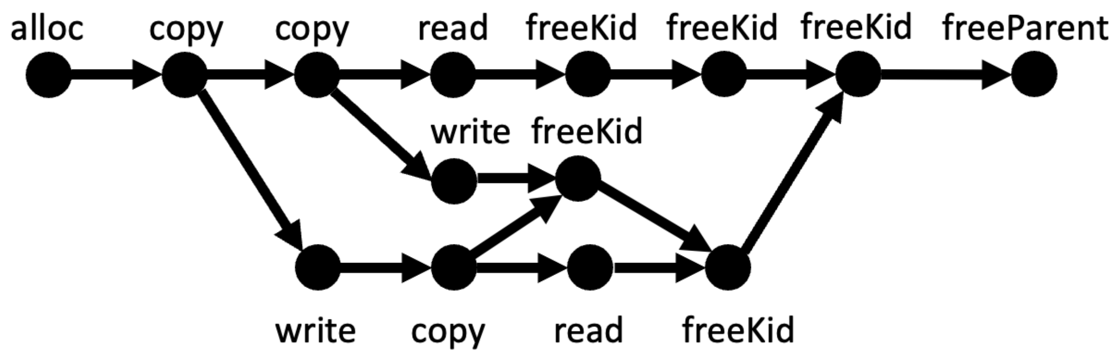


Figure 3.2: Diagram of pointer duplication

```
main = do
            myPointer <- allocLin
            myPointer <- writeLin 10 myPointer
            doubled *? myPointer <- readLin myPointer
            myPointer <- writeLin (doubled * 2) myPointer
            freeLin myPointer
```

## 3.4 Pointer Duplication

When using pointers in multi-threaded applications, algorithms may need to distribute the same pointer to multiple functions, allowing them both simultaneous access. For example, in the producer-consumer model one function produces values whilst another removes them.

As shown in figure 3.1, the type-checker would reject such a program as duplicating a pointer is not a linear usage. If multiple pointers were allowed, one section of the code could free the memory behind a pointer, whilst another thread continues to read from the same location, resulting in a **use-after-free** bug. A solution is necessary which guarantees that once the underlying memory is freed, none of the pointers will be used. This section describes an approach to allow for the safe duplication of pointers.

The solution I present relies on considering one pointer the *parent* and any further pointers created the *children*. When the programmer duplicates a pointer, they create a child pointer (**cp**) which references the same location in memory. When **cp** is deleted, another pointer to the same location in memory must be provided as a means of verification that the underlying memory has not yet been freed.

There will therefore be a set of pointers, and to delete a pointer (**p**) from this set another pointer must be passed alongside **p**. The only pointer which can then be deleted by themselves is the parent pointer, in this manner it is forced to be the last pointer freed. Deleting a child pointer does not actually free any underlying memory, it is only the freeing of the parent pointer which results in the underlying memory being freed.

To perform the necessary checks at compile time, every pointer needs to be marked at the type level with the set to which it belongs. We assign every pointer to a set by indexing the pointers by another boxed *AnyPtr* datatype called *SafePointer'*. This identifier represents the set of all pointers referencing the same location in memory. The pointer is also indexed by a new data type *PointerType* representing whether the pointer is a parent pointer or one of the children.

```
data PointerType = Parent | Child

data TrackedPointer : PointerType -> SafePointer' -> Type where ...
```

There are two free functions necessary to implement for the *TrackedPointer*. The *freeParentTP* function will consume a parent pointer *and* free the underlying memory. The second function *freeKidTP* takes in two pointers of the same set, consumes the child pointer and returns another linear reference to the first pointer.

```
freeParentTP : (1 _ : TrackedPointer Parent self) -> L Unrestricted ()
freeKidTP : (1 _ : TrackedPointer pointerType self)
            -> (1 _ : TrackedPointer _ self)
            -> L Linear (TrackedPointer pointerType self)
```

The important elements to note are *freeParentTP* requires the pointer to be indexed by the value *Parent* and *freeKidTP* requires the *self* identifiers to match up.

To highlight the simplicity of the implementation of *freeKidTP* it is below:

```
freeKidTP parent (CreateTrackedPointer _) = pure1 parent
```

*pure* is a generic function which places values within a monad. However, as Idris does not contain multiplicity polymorphism, the usage of all values must be explicitly specified. Therefore *pure1* is the function which wraps a value into the monad of type *L Linear*.

The *copyTP* function must produce another child pointer in the same set, and therefore has the following type:

```
copyTP : (1 _ : TrackedPointer pointerType self) ->
         L IO {use=1} (LinPair
                       (TrackedPointer Child self)
                       (TrackedPointer pointerType self))
```

*LinPair* is similar to *SemiLinPair*, but requires the linear usage of both its arguments when deconstructed in a linear context.

The type signature of *TrackedPointer* complicates the implementation of *alloc*. Simply copying the relevant definition of *alloc* from the previous section results in:

```
allocTP : L Linear (TrackedPointer Parent newptr)
```

In the above case, the compiler considers *newptr* a hole in the type signature and cannot deduce its value. The solution requires the introduction of the dependent pair type *Res*. *Res* allows the type of the second argument in the pair to depend on the value of the first, allowing the *allocTP* function to return both the pointer and the tracked pointer indexed by said pointer. This allows for the following type signature:

```
allocTP : L Linear (Res SafePointer' (\self => TrackedPointer Parent self))
```

*Res* is linear in its second argument and the QUIPS library contains no functions for interacting with *SafePointer'*. Therefore, the type signature provided is pointer safe and there are no longer holes in the type signature.

The programming language Rust also provides strong guarantees about pointer safety. However, it achieves this by limiting the programmer to have access to only one mutable reference to a value at a time. In Rust if the programmer needs more than one pointer, they must revert back to reference counting at run time using smart pointers.

My contribution goes further than Rust and produces a more versatile system than the current implementation for pointers allowed in Rust. My contribution allows for a method of using multiple mutable references but validating their safe use at compile time, a form of compile time reference counting. Removing the program slow down of reference counting at run time.

## 3.5 Arrays

Next we build on the pointer based functions implemented so far to extend the QUIPS programming library with tools to interact with arrays of integers. The low-level data structure remains solely a pointer representing the first element in the array. Therefore, the previous sections guarantee that an array is safely allocated and freed, and this section will focus on eliminating array-index errors.

To implement array-index safety we wrap the pointer in a new datatype which tracks the size of the array at the type-level. To keep the type-signatures manageable, I will build the functions up using *SafePointer* for the purposes of this section, ignoring the necessary types required for array-pointer duplication, however, the implementation for duplication extends to the array type in exactly the same manner.

The *IntArray* data structure is indexed by the size of the array, the size is also placed inside the data structure alongside the pointer. The size is needed inside the data structure to have it available at run time, not just compile time:

```
private
data IntArray : Nat -> Type where
    ConstructIntArray : (len : Nat) -> SafePointer -> Array (S len)
```

*ConstructIntArray* allows for the assignment of sizes of vectors with arbitrary pointers. Making the constructor private is therefore vital to restrict the creation of arrays to the QUIPS library only. Otherwise *IntArray* types could be created where the length does not match the actual length of the underlying structure.

To guarantee *IntArray* cannot have size zero, the size of an *IntArray size* is $size = len + 1$ which is reflected in the return type of *ConstructIntArray* containing *S len*. This is most useful when the size of the array is required in proofs, to not have to worry about the empty list case. The QUIPS library contains the only function for creating array types and has the following signature:

```
createIntArray : (size : Nat) -> L Linear (IntArray (S size))
```

**createArray** is a simple dependently typed function, the value of the input *size* affects the type of the output, (the size of the output array). This function is implemented by calling a C function which allocates the required amount of contiguous memory and returns a pointer to the head of the array.

Using this datatype and the Linear Monad from earlier *writeIntArray* has a straightforward type signature:

```
writeIntArray : (Fin (S size)) -> Int -> (1 _ : IntArray (S size))
```

```
              -> L Linear (IntArray (S size)
```

As discussed in section 2.4 *Fin a* must contain a natural number smaller than *a*, therefore to write to an array of a given size the programmer must provide an index smaller than the size of the array.

Here *(1 _ : IntArray (S size))* is being used in the same manner as the pointer before. We then return another linear reference to the same array to guarantee it is eventually freed.

*readIntArray* works in a similar manner, and returns the *SemiLinPair* type, with the value contained at the index in its first argument, and the array in its second:

```
readIntArray : (Fin (S size)) -> (1 _ : IntArray (S size))
            -> L Linear (SemiLinPair Int (IntArary (S size)))
```

For completeness the signature of *freeIntArray* is below, but it is almost identical to the *freeLin* function from section 3.3.

```
freeIntArray : (1 _ : IntArray size) -> L Unrestricted ()
```

## 3.6 Basic Polymorphism

Currently the four functions set out in section 3.5 operate only on integer arrays. Here we extend this to allow for a restricted form of polymorphism, creating arrays of either *integers* or *chars*.

This can be implemented using *typeclasses* in Idris. *Typeclasses* allow functions with the same name to have different implementations dependent on the types in use.

We extend the *IntArray* data structure to allow the type contained in the array to be tracked.

```
private
data SimplePolyArray : Nat -> (elem : Type) -> Type where ...
```

A typeclass is described by an interface representing the set of functions that can be called polymorphically for all types within a certain typeclass. The interface for the *CType* typeclass can now be implemented for every type one would like to be able to use with the *SimplePolyArray*. The functions are almost identical to the ones developed in the previous section however, they allow the functions output/input to vary according to the type *a*.

```
interface CType a where
        createArray : (size : Nat) -> L Linear (SimplePolyArray (S size) a)
        readArray :   (index : Fin size) -> (1 _ : (SimplePolyArray size a))
                       -> L Linear (SemiLinPair a (SimplePolyArray size a))
        writeArray : a -> (index : Fin size) -> (1 _ : SimplePolyArray size a)
                       -> L Linear (SimplePolyArray size a)
        freeArray : (1 _ : SimplePolyArray size a) -> L Unrestricted ()
```

The QUIPS library contains implementations for these functions for the *Int* and *Char* types in Idris.

Below is an example use of the *SimplePolyArray*, in this case the *Char* ('a') is written to the array and therefore it is a *Char* array. The compiler is aware of this at compile time and selects the necessary code to guarantee the correct amount of space is allocated. If an *Integer* was written instead to the array it would allocate enough space as needed for an Integer array. This is done automatically by the compiler at compile time and does not require the user specifying the size needed as would be required in C when using the *malloc*[2] command.

```
main = runLin $ do
        _ # arr <- createArraySimplePoly 10
        arr <- writeArraySimplePoly 5 'a' arr
        val *? arr <- readArraySimplePoly 5 arr
        print val
        freeArraySimplePoly arr
```

## 3.7 Array Polymorphism

This section extends polymorphism to handle arbitrarily nested arrays, and in the next section C-Style structs. Typeclass polymorphism works well for basic types such as *chars* and *integers*, but becomes limiting for more complex forms of polymorphism. Here I introduce a more versatile method to handle the added complexity.

The types *Char* and *Int* are used to explain the technique, adding arrays to the model at the end.

To represent the valid types to store in an array we introduce the *ValidType* data structure to represent the accepted types.

```
data ValidType = ValidInt | ValidChar

data PolymorphicArray : Nat -> ValidType -> Type where
        createPolyArray : SafePointer -> PolymorphicArray size type
```

The array is now indexed by a value of type *ValidType* to represent the elements contained within.

To create an array a value of type *ValidType* is passed, the function will then allocate the correct amount of space in memory depending on the size of the element types.

```
sizeOf : ValidType -> Int
sizeOf ValidInt = 32
sizeOf ValidChar = 8

createPolyArr : (size : Nat) -> (type : ValidType)
                    -> PolymorphicArray (S size) type
```

Considering the *read* function for example, the output type of the element which was read must depend on the value of *ValidType* the array is indexed by.

Dependent types allow for computation to take place in the type signature. A function can convert between *ValidType* and the actual type, and the result is used in the return type of relevant functions.

```
validTypeToType : ValidType -> Type
validTypeTotype ValidInt = Int
validTypeToType ValidChar = Char
```

[2] https://www.tutorialspoint.com/c_standard_library/c_function_malloc.htm

This function can now be used in the type signature for *read*.

```
readArrPoly : (Fin m) -> PolymorphicArray m type -> validTypeTotype type
```

*readArrPoly* allows the return type to vary depending on the type of the elements in the array, and therefore allows for array polymorphism. However, *readArrPoly* has a downside compared with typeclasses: the code must case-split at runtime, depending on the value of *type*, which leads to unnecessary runtime overhead that is ideally eliminated at compile time.

To address the run time case-split overhead the compiler must be forced to decide the value of *ValidType* for all the types at compile time. The data-type *ConvertValidType* has only two values, *ConvertValidInt* and *ConvertValidChar*, each representing the valid conversions between *ValidType* and *Type*.

```
data ConvertValidType : (t : ValidType) -> Type -> Type where
    [search t]
    ConvertValidInt : ConvertValidType ValidInt Int
    ConvertValidChar : ConvertValidType ValidChar Char
```

The read function is now written as follows, where the compiler implicitly constructs a value of the above type, depending on what is passed as an argument. It is the [**search t**] command which tells the compiler that based on the value of *t*, it should look for a valid value of type *ConvertValidType*:

```
readArrPoly : (Fin m) -> PolymorphicArray m type
            -> {auto 0 conv : ConvertValidType type outType} -> outType
```

The power of the above type is the assignment of usage *0* to *conv*. The program can pattern match on conv, but, the compiler guarantees that conv will not be used at runtime and therefore there is no cost due to pattern matching at runtime.

The *ValidType* constructor can now be easily extended to handle arrays by adding an additional constructor *ValidArr*:

```
data ValidType : = ... | ValidArr Nat ValidType

sizeOf ValidType -> Int
sizeOf ValidInt = 32
sizeOf ValidChar = 8
sizeOf ValidArr size elementType = size * (sizeOf elementType)
```

In this project I will only deal with completely flat data structures. An array that contains 10 arrays contains the arrays within itself in a flat manner, rather than containing pointers to other arrays.

Until now, arrays have only contained primitive types, but now they can contain arrays. This leads to complications when reading an array from an array.

Illustrating this with an example, suppose we have a pointer (**p**) to an array containing arrays of integers (**outer array**). The programmer then reads **p** and obtains an array of integers (**inner array**) referenced by a pointer (**cp**). The compiler must guarantee that whilst either **p** or **cp** are in use, the **outer array** is not freed. In practice, this means the compiler must guarantee that **cp** is consumed before the free command is called on **p**, which frees the **outer array**.

Next, consider multiple reads of the **outer array** resulting in a list of **child pointers** ($\mathbf{cp_1}, \mathbf{cp_2}, \dots$) referencing arrays within the **outer array**. All **child pointers** must be consumed before the **outer array** is actually freed. The solution to this is equivalent to the one presented in section 3.4. These pointers all lie in a set, where **p** is indexed stating it to be the parent, the rest of the pointers are the children. To free a child pointer (**cp**), another pointer from the set must be passed alongside to verify the **outer array** has not yet been freed. The only pointer which can be freed by itself is the parent pointer referencing the **outer array**. In this manner, it is guaranteed that **child pointers** are consumed before the **outer array** is freed.

The added polymorphism also complicates the *writeArrPoly* function. Consider a naive implementation of *writeArrPoly*:

```
writeArrPoly : Fin size -> (1 _ : PolymorphicArray (S size) type)
            -> validTypeToType type -> L Linear (PolymorphicArray (S size) type)
```

Assume we have an array of integers (**small array**), and an array of arrays of integers (**large array**). These two arrays are in separate locations in memory, i.e. one is not a slice of the other. The programmer then issues a command to copy **small array** into **large array**, returning a fresh pointer to **large array**. In this case, **small array** has not yet been freed, but there is no longer a pointer to it requiring freeing, which constitutes a memory leak.

This is addressed by returning the **small array** alongside **large array** after the copying has taken place. This is only necessary when the element to be written is an array, defined below are two concreted types for *writeArrPoly* depending on whether the *ValidType* is *ValidInt* or *ValidArr*

```
writeArrPoly : Fin size -> (1 _ : PolymorphicArray size ValidInt)
                    (1 _ : Int) -> L Linear (PolymorphicArray size ValidInt)

writeArrPoly : Fin siz -> (1 _ : PolymorphicArray size (ValidArr 10 ValidInt))
    -> (1 _ : PolymorphicArray 10 VaildInt)
    -> L Linear (LinPair PolymorphicArray 10 VaildInt
                    PolymorphicArray size (ValidArr 10 ValidInt))
```

The key difference is in the output. When writing an array, the output contains a *LinPair* requiring both arrays provided to be consumed again linearly, to guarantee both will be freed.

## 3.8 Structs

Structs are a common data structure used in C, which allow the programmer to layout what they would like to store in memory. The closest equivalent to this in a non-dependently typed language are tuples, however, tuples require a separate data type per length. In this section we set out how to add structs to the existing polymorphism framework.

To fully describe the layout of an array, it was enough to index by the size and the singular type of elements contained within. A struct can have any size but more importantly can have a different type at each position. Therefore, to fully describe a struct requires an Idris *Vect* (**V**). The values in **V** represent the types of the data present in the struct.

To consider structs as a valid type (section 3.7) in the polymorphism scheme, *ValidType* is extended to include a constructor representing a struct: *ValidStruct*. *ValidStruct* contains **V** representing the size and types of the elements of the struct.

```
data ValidType = ... | ValidStruct (Vect size ValidType)
```

A *Struct* data structure looks similar to the *PolymorphicArray* as it contains a pointer to the start of the struct but differs in being indexed by a *Vect* describing the contents.

```
data Struct : Vect size ValidType -> Type where
        StructCreate : (layout : Vect (S k) ValidType) -> SafePointer
                            -> Struct layout
```

For example, a struct of the type below, represents a struct with two elements, the first element is an array of size 10 containing integers. The second element is a struct containing an *Int* and a *Char*:

```
exampleStruct : Struct (Vect 2 [ValidArr 10 ValidInt,
                                ValidStruct (Vect 2 [ValidInt, ValidChar])])
```

A struct can be created with the *createStruct* function which takes in a layout representing the struct required:

```
createStruct : (layout : Vect (S k) ValidType) -> L Linear (Struct layout)
```

Consider writing into *exampleStruct*, when writing to the first element, an array of integers must be provided, however, when writing to the second, a struct containing an *Int* and *Char* must be provided instead. To maintain type safety, the type checker must check the value written is the correct type depending on the index, requiring the following type signature for *writeStruct*:

```
writeStuct : (loc : Fin size) -> Struct layout
    -> (1 _ : validTypetoType (index loc layout)) -> L Linear (Struct layout)
```

*readStruct* is typed similarly, where the output type depends on both the value of *loc* and the *layout*:

```
readStruct : (loc : Fin size) -> Struct layout
        -> (L Linear (SemiLinPair (validTypetoType (index loc layout)) (Struct layout)))
```

Once again structs face the same complexities as the *PolymorphicArray*, discussed at the end of section 3.7. To correctly implement structs, all the same safety features must be implemented for structs too.

# Chapter 4

# Evaluation

The project was a success. The QUIPS library successfully allows for low-level pointer allocation and freeing, whilst guaranteeing pointer bugs cannot occur at run time and therefore achieved the success criteria. Additionally, we developed a clean syntax with the help of monads, and extended the QUIPS library to handle polymorphic arrays and structs.

This chapter evaluates the QUIPS library on the following four fronts. A proof of correctness is sketched to verify pointer bugs cannot occur. I present simple programs to the Idris compiler for safety verification, to demonstrate the catching and rejecting of incorrect programs. I then compare the efficiency of the library to using equivalently safe data-structures in Idris and another less safe array data structure available in Haskell. Finally, I consider the usability of the library, showing a more involved use of the library looking at binary search, heapsort and the necessary dependently typed proofs needed.

## 4.1 Proof

Here I develop a proof to guarantee that when using *allocLin*, *freeLin*, *readLin* and *writeLin*, it is guaranteed that all pointers are allocated and in case of terminating programs freed, and that none of the pointer bugs discussed in section 2.1 can occur. The four functions to examine are:

```
allocLin : L Linear SafePointer
writeLin : Int -> (1 _ : SafePointer) -> L Linear SafePointer
readLin : (1 _ : SafePointer) -> L Linear (SemiLinPair Int SafePointer)
freeLin : (1 _ : SafePointer) -> L Unrestricted ()
```

Consider a pointer (**p**) created by a call to *allocLin*. The linear usage of the *SafePointer* guarantees that in every possible control flow of the program, **p** is consumed exactly once. We consider an arbitrary execution flow of **p** and show an invariant on **p** whilst it is yet-to-be-consumed: that the underlying memory of **p** is valid and **p** is the only pointer to a certain location in memory.

Firstly, the pointer could be passed to an arbitrary function which wraps the pointer inside a data structure that has linear usage in the respective argument, for example:

```
data RandomStructure : Type where
    MakeStructure : (1 _ : SafePointer) -> Int -> RandomStructure
```

```
wrapPointer : (1 _ : SafePointer) -> L Linear RandomStructure
wrapPointer ptr = pure1 (MakeStructure ptr 10)
```

However, as the pointer can only be placed in data structures which are linear in the argument of the pointer, this structure must eventually be decomposed back down to the pointer which then needs to be consumed exactly once. The invariant is therefore maintained as the linearity guarantees there is always only one pointer and that it still needs to be consumed.

We are left with considering applications of the three functions which consume a pointer and access memory *writeLin*, *readLin*, *freeLin*. I will show that none of these functions can cause a pointer bug and they respect the required invariant.

Consider a call to *writeLin*, the function takes in an element and writes it to the pointers location in memory. The invariant guarantees that the pointer's memory is valid and therefore the write cannot cause an error. The function finishes by returning the pointer and requiring it to be used linearly once again. Our invariant is maintained as the pointer still references valid memory, the previous pointer was consumed and therefore the returned pointer is the only one pointing to its respective location in memory.

The proof looks almost identical for *readLin*, which cannot cause an error as the invariant guarantees the pointer has valid underlying memory which is returned alongside another copy of the pointer. Once again the original pointer was consumed and a new given which requires linear consumption, maintaining the invariant.

Finally we have *freeLin*, the function consumes a pointer and frees the underlying memory. As the pointer is valid the function won't cause a double-free bug, from here on the pointer is permanently consumed. As this was the only pointer for the respective memory, no further bugs can occur as the memory has been cleared and cannot be freed again as there is no pointer to it.

## 4.2 Simple Errors

In this section, I provide simple code snippets presenting incorrect programs, and the output of the compiler when doing so. This provides evidence for the safety of the *QUIPS library* in rejecting incorrect programs, whilst allowing safe programs to compile.

**Guaranteeing Pointers are Freed**

```
main = runLin $ do
        _ # ptr <- allocTP
        print "End of Program"

Error: While processing right hand side of main.
    There are 0 uses of linear name ptr.
```

In the above program, the pointer *ptr* is allocated but never freed, a common bug which leads to memory leaks. The compiler detects the bug and outputs an error message detailing that *ptr* is not used after being allocated.

This bug can be resolved by adding code to free the pointer:

```
main = runLin $ do
            _ # ptr <- allocTP
```

```
            freeTP ptr
            print "End of Program"
```

```
Successful Compilation
```

The program now successfully compiles and can be executed.

**Use-After-Free bugs**

Use-After-Free bugs (2.1) occur when a pointer is used after it has been freed. In the below example a pointer (*ptr*) is allocated, freed and then de-referenced to obtain the underlying value. This class of bugs leads to many security vulnerabilities as discussed in the preparation chapter.

```
main = runLin $ do
        _ # ptr <- allocTP
        freeTP ptr
        val *? ptr' <- readTP ptr
        print val
```

```
Error: While processing right hand side of main.
        There are 0 uses of linear name ptr'.
```

The compiler does not compile this code either and produces an error highlighting that the new pointer (*ptr'*) is never freed. The edited program is below which includes a second call to *free* to remove *ptr'*:

```
main = runLin $ do
        _ # ptr <- allocTP
        freeTP ptr
        val *? ptr' <- readTP ptr
        freeTP ptr'
        print val
```

```
Error: While processing right hand side of main. There are 2 uses of linear name ptr.
```

The compiler now produces an error signalling that (*ptr*) is used twice instead of once, *ptr* is passed to the free function, and then to the read function. This shows a case of the *Use-After-Free* bug being caught by the compiler.

**Double Free Bugs**

*Double Free* bugs (section 2.1) occur when the same pointer is freed twice. The program below contains a double free bug as the pointer (*ptr*) is freed twice.

```
main = runLin $ do
            _ # ptr <- allocTP
            freeTP ptr
            freeTP ptr
            print "End of Program"
```

```
Error: While processing right hand side of main.
        There are 2 uses of linear name ptr.
```

During the type-checking stage the compiler rejects the program, highlighting that the pointer (*ptr*) is used more than once, breaking the linearity constraints placed on it. Once the programmer removes the second call to *freeTP* the program successfully compiles.

**Calling Unsafe Functions**

Consider a hypothetical unsafe function below programmed by a user that frees the pointer passed as an argument twice:

```
process : TrackedPointer True self -> L IO ()
process ptr = do
        freeTP ptr
        freeTP ptr
        print "This process has concluded"
```

This function successfully compiles as the pointer is not required to have linear usage in its type-signature. However, when the programmer tries to invoke this function with a pointer:

```
main = runLin $ do
        _ # ptr <- allocTP
        process ptr

Error: While processing right hand side of main.
    Trying to use linear name ptr in non-linear context.
```

The QUIPS library considers only functions which have a linear usage in the pointer to be safe. Therefore, when `process` is called with a pointer the compiler produces an error as `process` is not guaranteed to have safe operation.

**Copying Pointers**

Here is an example of creating a duplicate pointer and not freeing the duplicate:

```
    main = runLin $ do
        _ # ptr <- allocTP
        ptr' ?? ptr <- copyTP ptr
        freeTP ptr

Error: While processing right hand side of main.
    There are 0 uses of linear name ptr'.
```

This can be fixed by calling the *freeKidTP* method on the child pointer:

```
        _ # ptr <- allocTP
        ptr' ?? ptr <- copyTP ptr
        ptr <- freeKidTP ptr ptr'
        freeTP ptr

Successful Compilation
```

Here the child pointer *ptr'* is now correctly freed before the main parent pointer is.

**Array Index out of Bounds**

Indexing-out-of-bounds occurs when an array is indexed by too large a value. In the code below an array is created and indexed by a value out of bounds:

```
main = runLin $ do
            _ # arr <- createIntArray 19
            let to_write = 1000
            arr <- writeIntArray 100 to_write  arr
            freeIntArray arr

Error: While processing right hand side of main.
    Can't find an implementation for So
        (with block in integerLessThanNat 100 False 20).
```

The error message produced by the Idris compiler complains that the indexing value is too large and it cannot create a proof that the value is within range of the size of the array.

Adjusting the index to be within bounds gives a successful compilation:

```
main = runLin $ do
            _ # arr <- createIntArray 19
            let to_write = 1000
            arr <- writeIntArray 5 to_write arr
            freeIntArray arr

Successful Compilation
```

**Polymorphic Structs**

Consider the following correct code:

```
main = runLin $ do
    _ # arr <- createPolyArr (TStruct [TInt, TChar]) 100
    _ # struct <- createStruct  [TInt, TChar]
    struct ?? arr <- writeArrPoly 0 arr struct
    freeStruct  struct
    freePolyArr arr

Successful Compilation
```

The code creates an array containing structs, then creates an instance of the required struct and copies it into the first index in the array.

Two potential errors here are: wrongly specifying the layout of the inner struct, or not freeing the original struct after it has been copied into the array. Both of these are correctly identified and cause type errors:

```
    _ # struct <- createStructL [TChar, TInt]

Error: Mismatch between: TInt and TChar.
```

Finally, not freeing the struct leads to the same error as shown before when pointers are not freed:

```
Error: While processing right hand side of main.
            There are 0 uses of linear name struct.
```

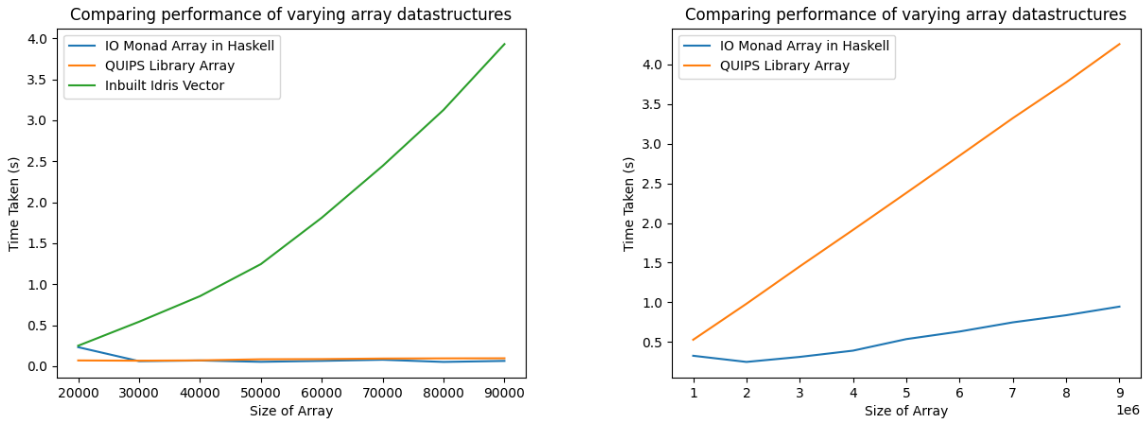


Figure 4.1: Comparing the performance of array like data structures. Both graphs measure performance of repeated indexing into an array to sum the elements starting with the last element, on the left the data structures being compared are: Haskell array data structure, QUIPS library array and inbuilt Idris vector. The right compares Haskell and QUIPS.

## 4.3 Performance

One goal of the library was to provide more performant data structures than the existing options in functional languages. The existing equivalently safe data-structure in Idris is the *Vect* data structure.

Figure 4.1 shows the clear improvement the QUIPS library array data structure has over the Idris vector in terms of performance.

The second diagram compares an array in Haskell with the QUIPS library array data structure (*IntArray* 3.5). The *IntArray* is a safer data structure as it guarantees no indexing-out-of-bounds and both the *IntArray* and Haskell array have the same time complexity, which represents array indexing in constant time. However, the Haskell array is faster.

In principle the QUIPS library should be faster as it removes the need for array bounds checking and garbage collection, which the Haskell library relies on. Therefore, there is nothing fundamental preventing the QUIPS library outperforming the Haskell array.

The speed difference comes down to a difference in the time taken for foregin function Interface calls. I measured the time taken to execute a series of foreign function Calls in both Haskell and Idris 2. Running a pointer update 50,000,000 times in Idris takes around 12.6 seconds giving 250 nanoseconds per foreign function call. On the other hand 500,000,000 calls in Haskell takes only 1.96 seconds giving a time per call of 4 nanoseconds.

## 4.4 QUIPS library Usage

To test the usability of the QUIPS library, I implemented the binary search algorithm and a priority queue data structure using the QUIPS library.

### Binary Search

To write binary search and require the minimum overhead for proofs requires careful consideration for the way in which the program is constructed, as dependent types enable many different ways of writing the same program. Here I breakdown the steps required and show examples from the binary search implementation.

### Invariants

The first step is deciding on the invariants for the arguments of the functions. For binary search the invariants chosen are: the left pointer (**lp**) is smaller than the right pointer (**rp**), and **rp** is less than or equal to the size of the array (**size**):

$$lp < rp \textbf{ and } rp \leq size \tag{4.1}$$

### Expressing as Types

These invariants can then be directly translated onto the necessary proofs required as arguments to the *binarySearch* function:

```
binarySearch : (lp : Nat) -> (rp : Nat) ->
                  (1 arr : IntArray size ...) ->
                       (invariant1 : LTE (S lp) rp) ->
                       (invariant2 : LTE rp size) -> ...
```

The *LTE* data type (2.8) is used to represent the invariants *invariant1* and *invariant2* which corespond to the invariants 4.1. A successful function call guarantees that at the start of execution the invariants are respected.

### Providing Proofs

The element which then needs to be indexed for binary search is $(lp + rp)//2$. To index into this position requires proving to the compiler that under the given invariants this index is within range of the size of the array. Therefore, the compiler must be provided a proof showing $(lp + rp)//2 \leq size$

This required showing:

$$lp + rp < 2 * size \tag{4.2}$$

$$x < 2 * a \implies (x//2) < a \tag{4.3}$$

To prove the above lemmas to the compiler required implementing functions with the type signatures:

```
lemma1 : (prf1 : LTE (S lp) rp) -> (prf2 : LTE rp size) -> LTE (lp + rp) (size + size)
```

```
lemma2 : LTE x (a + a) -> LTE (div2 x) a
```

Writing all the necessary lemmas and figuring out the correct way to write binary search did require a significant amount of code and thought. This is the main downside of dependent types, and will therefore most likely reserve dependent types for very critical systems.

The final step is to show the compiler that a subsequent recursive call to *binarySearch* continues to respect the necessary invariants, and requires the same approach as taken to prove indexing is valid. In this case, the compiler must be shown that the new values for **lp** and **rp** respect the invariants, allowing a recursive call to take place.

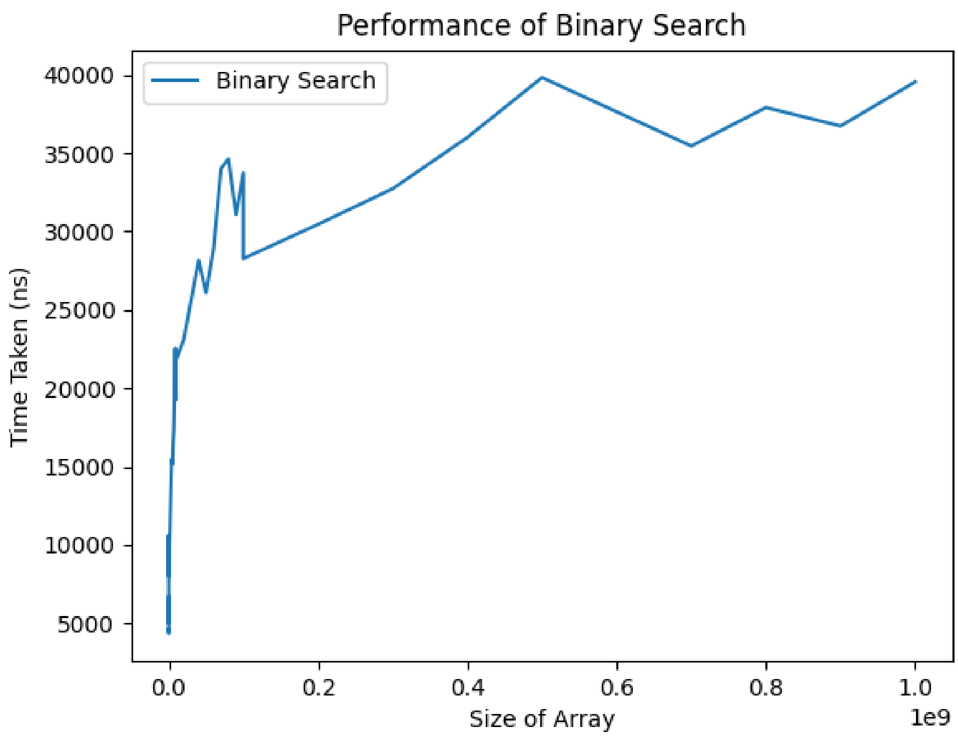


Figure 4.2: Binary Search Time Complexity

It is important to note that once I had finished the algorithm, it did not crash, however the algorithm was incorrect on first try. Therefore, no matter how many possible bugs eliminated with dependent types it is still important to test the algorithm.

## Binary Search Performance

Binary search is a particularly good algorithm to evaluate the QUIPS library as it has logarithmic complexity. Due to the use of the natural numbers data structure and the representation it relies on, naive implementations of algorithms using them can end up with linear time complexity.

A potential linear slow down can come from two places. The first is a manually implemented algorithm using natural numbers, which in this case was the *div2* function:

```
div2 : Nat -> Nat
div2 0 = 0
div2 (S 0) = 0
div2 (S (S k)) = S (div2 k)

%transform "div2" div2 j = integerToNat (div (natToInteger j)  2)
```

*div2* is very useful for proving properties about division by two. However, it is linear in its time complexity. Idris 2 provides a general *%transform* directive to allow a different function to be used at runtime. The *%transform* directive is used to replace *div2* with a function that converts the argument to an integer and uses an efficient inbuilt floor division operator. Idris natural numbers are represented as integers at compile time and therefore this is an efficient operation.

The other major potential slow down comes from the necessity of combining the run time code with the necessary proofs. It is vital to tell the compiler that the proofs are not required at run time, to make sure they do not slow down the performance of the code. To guarantee the proofs are not used at run time I came up with two helpful data structures.

```
data PPair : (a : Type) -> (P : a -> Type) -> Type where
    MakePPair : {P : a -> Type} -> (x : a) -> (0 _ : P x) -> PPair a P

data ErasedProof : Type -> Type where
    MkProof : (0 _ : a) -> ErasedProof a
```

These data structures were vital in allowing the performant binary search presented in the figure above. *PPair* is a data structure which contains a value and a proof that the value satisfies some condition. However, that condition has usage zero to guarantee it is erased at run time. The second data structure *ErasedProof* allows a function to return an erased proof, as the data structure guarantees it will not be used at run time.

## Priority Queue

The second data structure I implemented using the QUIPS library is the min-heap data structure. This was significantly more straightforward than Binary Search. The data-structure was tested running heap sort on an array.

Figure 4.3 shows the quasi-linear time complexity of my implementation of heap sort. Once again it was vital to guarantee the proofs were erased as with Binary Search.

The QUIPS library allowed the complexity of the array data-structure to be hidden behind a *MinHeap* data structure, which contained an *IntArray* with linear usage:

```
data MinHeap : (size : Nat) -> Type where
    CreateMinHeap : (1 _ : IntArray (S s) True self)
                        -> (containing : Nat) -> MinHeap (S s)
```

Methods were then exposed which allowed for pushing and popping from the *MinHeap*. There was also a function exposed which freed the heap when it was no longer needed. In this way a programmer can build on the existing data structures available in the QUIPS library to create higher-level data structures that are still pointer safe.

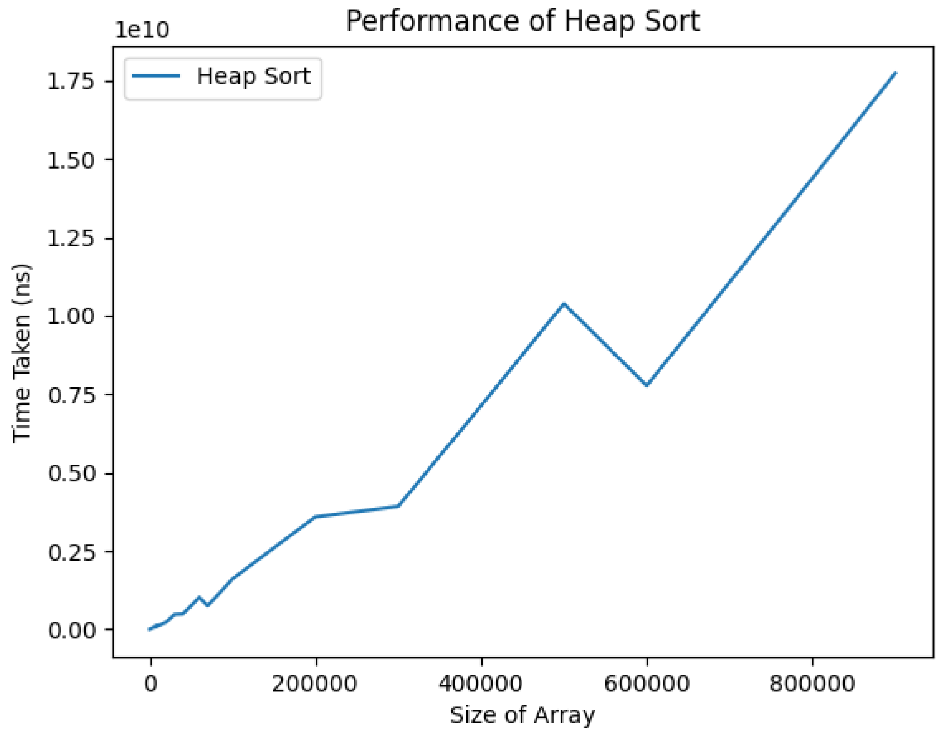


Figure 4.3: Heap Sort time complexity

## 4.5 Totality Requirements

If using the module for pointer copying discussed in section 3.4 there is one limitation of the language. When a pointer (**p**) is duplicated to obtain a child pointer (**cp**), both **p** and **cp** are guaranteed to reference valid memory via a proof by contradiction, which relies on the program being non-terminating.

Suppose we have a child pointer (**cp**), and a parent pointer (**p**) to a location in memory and suppose the parent pointer has now been freed. Supposing a terminating control flow, **cp** must eventually be freed, however, **cp** cannot be freed as the programmer cannot provide **p** as verification that the memory is still valid. The program was therefore rejected at compile time as **cp** cannot be consumed and would report an error stating 0 uses of linear name **cp**.

However, if non termination is allowed the programmer can break the safety of the language as follows:

```
writeForever : (1 _ : TrackedPointer _ _) -> L Unrestricted IO ()
writeForever tp = do
            tp <- writeTP 10 tp
            writeForever tp

main = runLin $ do
            _ # x <- allocTP
            copy ?? x <- copyTP x
            freeTP x
            writeForever copy
```

In the above code, a pointer is allocated, copied and then freed. The copied pointer is then sent to a non-terminating function which continuously writes to the pointer. As this function is non-terminating the linear usage never forces the pointer to be freed which would cause the contradiction as discussed in the previous paragraph.

This limitation can be fully resolved by including the *%default total* directive in Idris code. This forces the compiler to verify that all functions presented are total, if the compiler fails to verify by itself that all functions are total an error is presented to the user, who can then either try to prove the function total if they believe it to be, or adjust the code if the function is in fact not total.

In the case of the code above, the compiler outputs the following error:

```
Error: writeForever is not total,
possibly not terminating due to recursive path
Main.main -> Main.writeForever -> Main.writeForever
```

# Chapter 5

# Conclusion

The introduction and preparation chapters laid out the problems the QUIPS library aimed to solve. To create a library for safe pointer based operations that guarantee the absence of pointer bugs. Here I discuss the successes and failures I experienced, my personal reflections on the project and suggest how the project could be taken further.

## 5.1 Successes and Failures

The project was a success, the QUIPS library as implemented provides a proof of concept that Quantitative Type Theory allows for the implementation of a safe library for low-level operations. The solution for copying pointers is a new contribution, which naturally extends to a safe method for providing pointers to slices of arrays arbitrarily many times. I believe this is a particularly nice application of Quantitative Type Theory as well, as the solution requires both the linear type system and dependent types to allow pointer copying.

One issue I encountered when writing very polymorphic code was an Idris compiler bug which occurs when pattern matching against *Type*. I reported the bug to the Idris community who believe the bug is similar to two known bugs[1,2], however it is yet to be fixed and I didn't have the expertise to fix it myself.

As there is no multiplicity polymorphism, I could not write code as abstract as I usually would. Consider the following simple code:

```
idLinear : (1 _ : Int) -> Int
idLinear a = a

funcCorrect : (1 _ : Int) -> Int
funcCorrect a = idLin (idLin a)

funcIncorrect : (1 _ : Int) -> Int
funcIncorrect = idLin . idLin

Error: When unifying: (1 _ : Int) -> Int and b -> c
```

[1]https://github.com/idris-lang/Idris2/issues/2790
[2]https://github.com/idris-lang/Idris2/issues/1865

Both *funcCorrect* and *funcIncorrect* should successfully compile and have identical semantics, but Idris reports a type error for *funcIncorrect*. The error presents a limitation, due to the non-existence of multiplicity polymorphism, Idris fails to concrete the type `b -> c` to `1 _ : Int -> Int`, as it is not possible to express in the type of the composition operator that if passed two linear functions then the resulting function is linear.

The foreign function interface is not very performant and therefore arrays in Haskell are faster than the corresponding Idris code. However, there is nothing precluding a similarly performant array interface in Idris as there is in Haskell, that can be used with the type system developed for the QUIPS library. In this manner the QUIPS library should then be faster, as the QUIPS library does not need to be garbage collected or array bound checked.

## 5.2 Personal Reflections

In this project I worked with dependent types and proving the safety of code. Dependent types are a powerful tool for specifying data types, however, the more complex the data structure the harder it is in general to verify the safety of the program. It is important to consider the complexity introduced by dependently typed data structures as some forms lead to more added complexity than others. For example, indexing by natural numbers leads to requiring tough mathematical proofs, however indexing by simpler types such as booleans usually still provides powerful features without increasing the overhead to the programmer significantly.

The programmer must find the right trade off for their project when deciding what must be proven at compile time, and what can be handled at run time. Otherwise, a project will take much longer than expected.

If you believe a language / library has a certain property, I learnt it's important to try to sketch a proof. As it is easy to convince oneself a certain property holds, but there are many edge cases and possibilities that can be ignored or overlooked.

## 5.3 Further Work

There are many possible extensions of the QUIPS library. Firstly, work needs to be done on the Idris foreign function interface, as in its current form it is very slow, and provides a bottleneck for implementing performant solutions.

Functional languages have started to develop a unified abstraction for interacting with data-structures called lenses[3], another extension would be providing an interface for the data structures in the QUIPS library with lenses.

Lenses allow for a highly compositional form of indexing into data structures, instead of reading an array to obtain a struct, and then reading that struct to obtain a value. Lenses allow the read functions to be composed and focus on arbitrary elements within a data structure, even highly nested ones, simplifying otherwise overly cumbersome code. The unified abstraction would also allow programmers to quickly augment their code with the QUIPS library array data structure and obtain the benefits, as all the read and write functions would remain identical.

[3] `https://hackage.haskell.org/package/lens`

# Bibliography


[1] Robert Atkey. *Syntax and Semantics of Quantitative Type Theory*. http://www.t-news.cn/Floc2018/FLoC2018-pages/proceedings_paper_665.pdf.

[2] Edwin Brady. *Idris 2: Quantitative Type Theory in Practice*. https://arxiv.org/abs/2104.00480.

[3] Edwin Brady. *Type-Driven Development*. https://www.manning.com/books/type-driven-development-with-idris.

[4] Martin Hoffman. *Syntax and Semantics of Dependent Types*. https://www.cambridge.org/core/books/semantics-and-logics-of-computation/syntax-and-semantics-of-dependent-types/119C8085C6A1A0CD7F24928EF866748F.

[5] Ryan R. Newton Simon Peyton Jones Arnaud Spiwack Jean-Philippe Bernardy, Mathieu Boespflug. *Linear Haskell: practical linearity in a higher-order polymorphic language*. https://arxiv.org/abs/1710.09756.

[6] Erik Palmgren Peter Dybjer. *Intuitionistic Type Theory*. https://plato.stanford.edu/archives/spr2023/entries/type-theory-intuitionistic/.

[7] Philip Wadler. *Linear types can change the world*. https://homepages.inf.ed.ac.uk/wadler/papers/linear/linear.ps.

[8] Philip Wadler. *Monads for functional programming*. https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf.

[9] Philip Wadler. *Propositions as Types*. https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf.