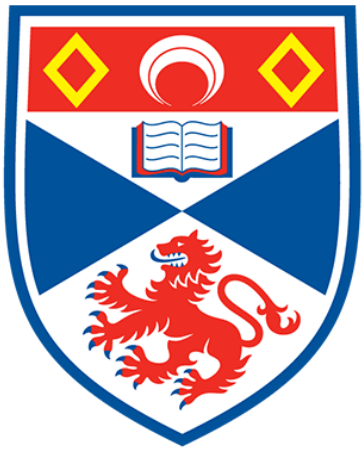
University of St Andrews

January 6, 2023

MSci Project

A Dependently Typed Parallel Runtime System

Findlay Sloan



University of
St Andrews

supervised by

Dr Christopher Brown

Contents

List of Figures

Abstract

As the multi-core CPU architectures are long the standard, writing parallel programs are essential to extract the maximum performance from the CPU. This dissertation explores the use of dependant types[1] in creating a model for parallelism, as well as investigates their use in defining algorithmic skeletons[2]. It describes extending a small, dependently typed programming language `pi-forall` [3] with a new C++ compilation backend, as well as developing concurrency primitives in the language. Using the primitives, the farm and pipeline skeletons are defined using dependent types, and furthermore, proofs are created to ensure the correctness of the skeletons. Finally, the skeletons are evaluated against other algorithmic skeleton libraries such as Skel[4] and GrPPi[5].

Declaration

"I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. "The main text of this project report is 32,500 words long, including project specification and plan. "In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work."

Chapter 1

Introduction

The limits of single core CPU hardware are being reached. Moore's Law[6], which is a prediction that the number of transistors on a chip will double every 2 years, has held true since 1975. Due to the scale of transistors getting smaller and smaller, the limit of this prediction is being reached and soon will no longer hold. The problem is not just fitting the number of transistors on the CPU, but also the energy required to power the CPU. The power for the CPU is given by the following equation[7]:

$$P = CV^2 f \tag{1.1}$$

where P is the power consumption of the CPU, C is the switched load capacitance, V is the voltage and f is the frequency of the processor. Equation 1.1 shows that the power is directly proportional to the voltage squared and the chip's frequency. Thus higher clock speeds, which are higher frequency, require higher voltage, causing problems with the power required growing rapidly. Another consequence of the growth in power required is that the heat generated from the chip will scale with the power, so heat dissipation will also become a problem. The solution to all these problems was to increase the number of cores on the CPU instead of focusing on increasing the single core performance of the CPU. This allows the processor's overall performance to scale with the number of cores instead of relying on the diminishing returns gained from improving the clock speed. The consequence of this decision is that parallel programming is essential to utilise the modern CPU's performance potential fully.

Parallel programming in its current state has many problems. There is no easy way for a programmer to convert existing non-parallel programs into a parallel program[8]. Some problems are specific to programming in a parallel setting. Running code in parallel introduces overhead, which, if it exceeds the tasks' computation time that's being run in parallel, there is performance degradation due to the parallelisation introduced. Parallel programs also need to consider synchronisation techniques to avoid issues like race conditions, which can cause programs to exhibit non-deterministic behaviour. There is also a bottleneck in the speedup that a parallel program can achieve when compared with the non-parallel version. Amdahl's Law[9], given in Equation 1.2, where p is the proportion of parallelisable code in the program and n is the number of cores used, is used to calculate the theoretical speedup possible by parallelising the program on n cores. It is also possible to calculate the thoertical maximum speedup by taking the limit as $n \to \infty$, which produces the equation $(1-p)^{-1}$. This means that if a program is only 75% parallelisable, i.e. 25% of the program must run sequentially, then the parallel program's speedup will tend towards 4, as the number of cores

grow when compared to the non-parallel version. This is a hard limit, so no matter how many cores the program is run on, the speedup cannot exceed 4. These factors combine to create a nearly impossible problem for current compilers to perform static analysis on.

$$speedup = \frac{1}{(1-p) + \frac{p}{n}} \tag{1.2}$$

One of the most important methods of static analysis done by a compiler is the type checker, which is used to check that a program is well-typed according to a set of typing rules and reject any programs that are not well-typed. A simple example would be a function expecting an integer as an argument but was instead given a string. The typing rules would ensure this is rejected due to not being well-typed. As the type checker rejects ill-typed programs, this can help programmers catch bugs, as programs with type errors will be badly formed and possibly erroneous. More advanced type systems allow more complex rules, making static analysis more advanced. Dependant types[10], which is a more advanced type system, allows types to depend on values. This can mean a type can encode a specification, and the type system will enforce that specification. If a specification is sufficiently precise, it can limit the elements that inhabit the type. An example of this would be the `List` type, shown in Figure 1.1, and a dependently-typed specification of a list that is known as `Vec`, shown in Figure 1.2. The reason that `Vec` is more advanced than `List` is that a `Vec` encodes its length. This is done by depending on a natural number, `n`, that represents the length of the `Vec`, which the highlighted section on Line 1 of Figure 1.2 shows. Both definitions are very similar, and this can be seen by comparing `List` definition with the non-highlighted sections of the `Vec` definition. Both are inductive definitions where the base case constructors, `Nil`/`NilV`, are for an empty `List`/`Vec` and contain no extra information. The recursive case constructors, `Cons`/`ConsV`, are for a non-empty `List`/`Vec` and the arguments represent the head and the tail of the `List`/`Vec`. The highlighted sections of Figure 1.2 are used to encode the length of the `Vec`. On Line 2 of Figure 1.2, the highlighted argument `[n = Zero]` is a constraint that the `NilV` constructor can only be used when the value of `n`, which the type depends on and represents the length of the list, is `Zero`. On Line 3, the `ConsV` constructor has two new additions. The first is the `Nat` argument called `m` which is used in the third argument to ensure that the tail of the list has length `m`. The second is the final argument and is the constraint that `n = Succ m` and this ensures that the value of `n` is `m + 1`. This guarantee will ensure that the element in the tail of the list has length `m` and that when combined with the head element, will produce a `Vec` of length `m + 1`.

```
1 data List (A : Type) : Type where
2     Nil
3     Cons of (A) (List A)
```

Figure 1.1: `pi-forall` List datatype definition

```
1 data Vec (A : Type) (n : Nat) : Type where
2     NilV of [n = Zero]
3     ConsV of [m : Nat] (A) (Vec A m) [n = Succ m]
```

Figure 1.2: `pi-forall` Vec datatype definition

With the `Vec` type, static analysis about the length of the list can be done at compile time through the type checker. For example, the head function, shown in Figure 1.3, will fail and raise an error when the `List` applied to it is non-empty, but with a `Vec` type, this failing case is no longer considered, as if the value of `n` is `Succ m` for some `Nat m` then it is impossible that the list is empty.

This can be useful in reducing the number of bugs a programmer could introduce as now a `List` structure could be guaranteed to be non-empty by using a `Vec` or that an element is never accessed at an index out of the bounds of the `Vec`.

```
1 head : [A :Type] -> [m:Nat] -> Vec A (Succ m) -> A
2 head = \ [A][m] x.  case x of
3      ConsV [m'] y ys -> y
4      -- NilV case is impossible
```

Figure 1.3: `pi-forall` Head Function

The Curry-Howard Isomorphism[11] observes a mathematical equivalence between propositions and types as well as programs and proofs. This means that the types are propositions and the inhabitants of the types are proofs of the proposition. Writing a program in a dependently typed language allows a programmer to create proofs about the correctness of the program.

If parallelism can be modelled in types, it would be possible to ensure it is used correctly This is because any inhabitants of the types are proofs that guarantee the correct use of the parallelism. An example of this would be a type that models the patterns such that only correctly constructed types can be elements of the type. This will allow the type checker to reject ill-formed patterns that could introduce bugs, meaning there are now stronger guarantees that the parallelism is used correctly. One example is modelling parallelism through algorithmic skeletons[2], which are high-level abstractions of common patterns that appear in parallel programs. They take care of the lower-level synchronisation details for the programmer, allowing them to think about the problems in a more high-level fashion and try to fit the problems onto a pattern or a composition of different patterns.

Using dependant types, the parallel patterns can be modelled in a type that includes proof obligations on the correctness of their construction. Only a correctly constructed pattern, which means that parallelism is utilised correctly, will be accepted by the type checker.

This dissertation has three main objectives.

1. Add a compilation backend from the small dependently typed language, `pi-forall`[3], that will create a C++ executable. This will then allow `pi-forall` programs to execute. This objective is successfully met in Chapter
2. Extend the `pi-forall` language and backend with concurrency primitives, allowing parallel programs to be written in the language. This objective is successfully met in Chapter 6
3. Model algorithmic skeletons using the dependant types in the language and explore the use of proofs within the patterns. This objective is successfully met in Chapter 7

1.1 Overview

Chapter 2 explores the background knowledge used in the dissertation. It starts by discussing Logic and Type Theory and their relationship. It then explains the `pi-forall` programming language and type checker. Finally, it defines algorithmic skeletons and their uses.

Chapter 3 discusses some of the related work to the dissertation, focusing on other dependently typed languages and also other algorithmic skeleton implementations.

Chapter 4 describes the software engineering approaches taken throughout the project's development.

Chapter 5 details how the C++ compilation background was created and shows many examples of how different `pi-forall` expressions get translated into C++. A section also discusses the limitations and areas for further work in the translator.

Chapter 6 gives the particulars on the numerous iterations of the design and implementation of the concurrency primitives and the C++ parallel run time system. It also starts discussing some of the `pi-forall` proofs that are used to ensure the correctness of the system. It also finishes by discussing limitations and areas for further work.

Chapter 7 gives an overview of the farm and pipeline parallel patterns, as well as details the proofs that are used throughout to ensure the correctness of the structures. It ends by discussing the limitations of the current design and outlines better designs when discussing future work.

Chapter 8 evaluates the parallel pattern implementation by comparing them against other pattern implementations written in other languages. There is also a section that critically appraises the completed project against the objectives of the project.

Chapter 9 concludes the dissertation by giving a summary of the project and going through the success and drawbacks of the project. Then it finishes by discussing what future work could be based on the results from this dissertation.

Chapter 2

Background

This chapter serves as an introduction to the required background knowledge for this dissertation. Section 2.1 introduces Logic, starting with Propositional Logic and describing the more advanced Predicate Logic. Section 2.2 introduces Lambda Calculus, both the original untyped Lambda Calculus and the simply typed Lambda Calculus. Section 2.3 details the Curry-Howard Isomorphism and its implications. Section 2.4 gives an introduction to type theory, which builds on the typed Lambda Calculus from Section 2.2. Section 2.5 describes the `pi-forall` programming language, giving its typing rules, syntax and a discussion on its limitations. Finally, Section 2.6 gives an overview of Algorithmic Skeletons.

2.1 Logic

2.1.1 Propositional Logic

Propositional logic[1] is a logic where logical formulas, or propositions, can be combined to form more complex formulas through connectives. Each logical formula is either True or False. "The sky is blue" is an example of a true statement, while "grass is blue" is an example of a false statement.

If A and B are logical formulas, then they can be combined with the following connectives to make a new formula:

1. $A \wedge B$ The **and** connective
2. $A \vee B$ The **or** connective
3. $A \rightarrow B$ The **implies** connective
4. $\neg A$ The **not** connective
5. $\bot$ The **false** proposition

The $\wedge$ connective represents logical conjunction, meaning the overall formula is true when A and B are both true. The $\vee$ connective represents logical disjunction, meaning that the overall formula is true when either one of, or both, A and B are true. The $\rightarrow$ connective represents logical consequence, meaning the overall formula is true only if when A is true, then B is true. The $\neg$ connective represents logical negation, which means the formula is true when A is false. Finally, the connective $\bot$ represents absurdity or falsehood and can never hold.

Natural Deduction

Natural Deduction [1] is a proof system that can be used with propositional logic to create a proofs that a conclusion follows from some premises, so when the premises are true, it is possible to conclude that the conclusion is true as well. The system contains a set of inference rules from which new formulas can be inferred based on true formulas. This means that new logical formulas can be proven true by applying the rules in succession.

The inference rules can be split into two types: introduction rules, which introduce a logical connective, and elimination rules, which eliminate the connective.

Each inference rule contains two things, the top half of the rule is the premises, and the bottom half is the conclusion that can be made if all the premises hold. If there are no premises in the rule, it is an axiom, meaning the rule can always be used as it always holds.

$$\frac{A \qquad B}{A \wedge B} \wedge I \qquad \frac{A \wedge B}{A} \wedge E_1 \qquad \frac{A \wedge B}{B} \wedge E_2$$

Figure 2.1: $\wedge$ Natural Deduction Rules

The $\wedge$ inference rules are shown in Figure 2.1 and contain the introduction rule, $\wedge I$, and the two converse elimination rules, $\wedge E_1$ and $\wedge E_2$. The $\wedge$ introduction rule is used to introduce the $\wedge$ connective and has two premises required for the rule, A and B. This means that the conclusion, $A \wedge B$, can only be formed when the formula A is true and the formula B is true. There are two elimination rules because, from the formula $A \wedge B$, it is possible to infer that both the formulas A and B are true. Rule $\wedge E_1$ is used to infer A on the left-hand side, while Rule $\wedge E_2$ is used to infer B on the right-hand side. An example of using these 3 rules is shown in Figure 2.2 which shows the commutativity property of $\wedge$. It uses the assumption, $(A \wedge B) \wedge C$, to prove the conclusion, $A \wedge (B \wedge C)$.

$$\dfrac{\dfrac{\dfrac{(A \wedge B) \wedge C}{A \wedge B} \wedge E_1}{A} \wedge E_1 \qquad \dfrac{\dfrac{\dfrac{(A \wedge B) \wedge C}{A \wedge B} \wedge E_1}{B} \wedge E_2 \qquad \dfrac{(A \wedge B) \wedge C}{C} \wedge E_2}{B \wedge C} \wedge I}{A \wedge (B \wedge C)} \wedge I$$

Figure 2.2: $\wedge$ Commutativity Proof

The inference rules for $\rightarrow$ are shown in Figure 2.3 and give the introduction rule, $\rightarrow I$, and the elimination rule, $\rightarrow E$. In the $\rightarrow I$ rule, to obtain the conclusion that $A \rightarrow B$, the formula B needs to be shown to be true and that it can be inferred from the temporary assumption A. When the conclusion is derived, the temporary assumption A is discharged, which is that the square brackets indicate, that A is temporary. It also has a superscript number which is also in the rule name to indicate that the assumption with number 1 was discharged. For the elimination rule, if there is a formula, $A \rightarrow B$, and a formula A that are both true, then it is possible to infer that the formula is B true.

$$\frac{\begin{array}{c}[A]^1\\ \vdots\\ B\end{array}}{A \to B} \to I^1 \qquad\qquad \frac{A \to B \qquad A}{B} \to E$$

Figure 2.3: $\to$ Natural Deduction Rules

The $\vee$ inference rules are shown in Figure 2.4 and give two introduction rules, $\vee I_1$ and $\vee I_2$, and the elimination rule, $\vee E$. To infer a proof for $A \vee B$, either a proof of A or B needs to be given. The $\vee E$ rule states that if a proposition C can be inferred from an assumption of A and an assumption of B, then it follows that C can be inferred from $A \vee B$ with the assumptions A and B discharged. The square brackets again show that the assumptions A and B are discharged much like in the previous $\to I$ rule.

$$\frac{A}{A \vee B} \vee I_1 \qquad\qquad \frac{B}{A \vee B} \vee I_2 \qquad\qquad \frac{A \vee B \qquad \begin{array}{c}[A]^1\\ \vdots\\ C\end{array} \qquad \begin{array}{c}[B]^2\\ \vdots\\ C\end{array}}{C} \vee E^{1,2}$$

Figure 2.4: $\vee$ Natural Deduction Rules

There is only a single inference rule for the $\bot$ connective, shown in Figure 2.5, which is $\bot E$. As it represents absurdity, it should be impossible to derive it, hence there is only an elimination rule. The rule states that from $\bot$ anything can be inferred, as from absurdity any formula can be inferred.

$$\frac{\bot}{A} \bot E$$

Figure 2.5: $\neg$ Natural Deduction Rules

The $\neg$ inference rules are shown in Figure 2.6 and give the introduction rule, $\neg I$, and the elimination rule, $\neg E$. In the $\neg I$ rule, the conclusion of $\neg A$ can only be inferred when a proof of B and $\neg B$ can be inferred from an assumption of A. The rule will also discharge the assumption A. For the $\neg E$ rule, if a proof of A and $\neg A$ can be obtained, which is absurd, then any formula B can be obtained.

$$\frac{\begin{array}{cc}[A]^1 & [A]^1\\ \vdots & \vdots\\ B & \neg B\end{array}}{\neg A} \neg I^1 \qquad\qquad \frac{A \qquad \neg A}{B} \neg E$$

Figure 2.6: $\neg$ Natural Deduction Rules

Negation can be defined in terms of implication and $\bot$, as shown below:

$$\neg A \equiv_{df} A \to \bot$$

where $\equiv_{df}$ is "defined to be equal". An intuition for how this is true can be found by looking at the $\neg I$ rule. From an assumption of A, if both B and $\neg B$ can be derived then, it would be possible to derive anything, such as $\bot$. If the assumption A was discharged using the $\rightarrow I$ rule, then $A \rightarrow \bot$ would follow. This means that from an assumption A, if it is possible to infer $\bot$ then the assumption A cannot hold, therefore $\neg A$ holds.

2.1.2 Predicate Logic

Predicate Logic is an extension of Propositional Logic, as it builds on the foundation of Propositional Logic by introducing new syntax, where the space of formulas is extended to include predicates over variables. For example, in Propositional Logic, the statements such as "Jane is alive" and "Bob is alive" are unrelated statements, while in Predicate Logic, it is possible to create a predicate "is alive" and the two statements can be defined using the same syntax by using the "is alive" predicate. With Predicate Logic, it is possible to make statements about all objects within some set, or that an object exists for a property or formulas about the relationship between objects. This is extremely powerful as with just these extensions, it's possible for propositional equality to be defined.

As Propositional Logic is used as a foundation for Predicate Logic, all the previously discussed syntax is used in Predicate Logic. It introduces syntax for variables, constants and predicates, which are non-logical objects. From the previous example of "Bob is alive", "Bob" would be a constant and "is alive" would be the predicate. Instead of using a constant, it is possible to use a variable, x to be used in the predicate. A predicate, P takes n arguments that it predicates over and has arity n. For example, the "is alive" predicate, P, would form a formula when an object is provided, such as the constant b, representing "Bob", to form $P(b)$, or the variable x, representing an object within the domain, to form $P(x)$. Predicate logic also introduces two quantifiers that apply to the variable in a formula, the universal and existential quantifiers. The universal quantifier, $\forall$, represents the assertion that the formula holds for every single object in the domain. The syntax for using it is shown below:

$$\forall x.A$$

which asserts that the formula A is true for every single x within its domain. The formula A can contain the variable x within it.

The existential quantifier, $\exists$, represents the assertion that there exists at least one object within the domain that the formula is true for. The syntax for the quantifier is shown below:

$$\exists x.A$$

which asserts that the formula A is true for at least one object x within the domain. Like with the universal quantifier, the variable x can appear within the formula A.

Equality is defined as a primitive relation with its own syntax, the infix operator $=$. An example of its use is shown below:

$$x = x$$

Natural Deduction Rules

Predicate Logic also introduces new Natural Deduction inference rules for the new syntax it introduced. Figure 2.7 gives the introduction rule, $\forall I$, and the elimination rule, $\forall E$, for the $\forall$ quantifier. In the $\forall I$ rule, from a proof of A, where x is free within A, we can infer that $\forall x.A$ holds. For the $\forall E$ rule, if a proof of $\forall A$ is provided, then $A[t/x]$ can be derived, as it is substituting x for some constant t, which is possible as the formula A holds for all x, hence would hold for t.

$$\frac{A[t/x]}{\forall x.A}\,\forall I \qquad\qquad \frac{\forall x.A}{A[t/x]}\,\forall E$$

Figure 2.7: $\neg$ Natural Deduction Rules

Figure 2.9 gives the inference rules for the $\exists$ quantifier. The $\exists I$ introduces the quantifier, which when given a proof of $A[t/x]$, which is a proof of A holding when substituted with some object t, it is possible to derive a proof for the formula $\exists x.A$. The $\exists E$ rule is the elimination rule for $\exists E$ quantifier. If an assumption that for some object x, $A[t/x]$ holds derives a proof for a formula B where x does not appear, then it is possible to infer B, with This rule will also discharge the assumption.

$$\frac{A[t/x]}{\exists x.A}\,\exists I \qquad\qquad \frac{\exists x.A \qquad \begin{array}{c}[A]^1\\ \vdots \\ B\end{array}}{B}\,\exists E^1$$

Figure 2.8: $\neg$ Natural Deduction Rules

The equality relation is a predicate which holds when the two objects are defined to be equal, meaning that the objects are defined to be the same object within the set. A property of this relation is called reflectivity, which is that any object x, $x = x$ holds. This property is shown as the $= I$ rule in Figure 2.9, which has no premises as the conclusion, $x = x$, always holds. The $= E$ rule can derive a substitution, using a proof of an equality, $t_1 = t_2$ and a formula where t_1 appears, $A[t_1/x]$, to substitute the occurrence of t_1 with t_2 to infer the formula $A[t_2/x]$.

$$\frac{}{x = x}\,= I \qquad\qquad \frac{A[t_1/x] \qquad t_1 = t_2}{A[t_2/x]}\,= E$$

Figure 2.9: $\neg$ Natural Deduction Rules

2.2 Lambda Calculus

2.2.1 Untyped Lambda Calculus

The original Lambda Calculus[12] was introduced by Alonzo Church and is now known as the untyped Lambda Calculus. It became widely popular when it was discovered to be a formalisation of computation, in which every object is a function and became the basis of the functional programming paradigm. It contains three types of objects, shown below:

If M, N are valid lambda terms, then a lambda term can take one of the following forms:

1. x is a **variable**
2. $\lambda x.M$ is an **abstraction**
3. MN is an **application**

The abstraction is a function, that takes as a formal parameter, x, and returns the lambda term M. In an abstraction, the variable x is said to be bound by the abstraction. All other occurrences of the variables are called free, unless they are bound by some other abstraction. The application represents the application of M to the term N. It can be thought of as a function application where N is the argument to the function M.

It is possible to perform a substitution on lambda terms, replacing all occurrences of a term with another. To substitute f for x in a lambda term M is denoted by $M[x \mapsto f]$, note that this is the same syntax as the substitution syntax for predicate logic. The substitution's semantics depend on the lambda term type M. If M is the variable x for which we are substituting for then $x[x \mapsto f] \equiv_{df} f$ as x is replaced by f. If M is another variable y where $y \not\equiv_{df} x$ then $y[x \mapsto f] \equiv_{df} y$. If M is an abstraction $M \equiv_{df} \lambda x.e$, then $(\lambda x.e)[x \mapsto f] \equiv_{df} \lambda x.e$. This is because the variable x being substituted does not change the definition of M, so there is no need to replace x with another variable. If M is an abstraction where the variable bound is not x such as $\lambda y.e$ then there would be a substitution, such that $(\lambda y.e)[x \mapsto f] \equiv_{df} \lambda y.(e[x \mapsto f])$. This only holds when the variable y does not appear free in f. If M is an application such that $M \equiv ab$ then $(ab)[x \mapsto f] \equiv_{df} (a[x \mapsto f]b[x \mapsto f])$.

There are two methods of computation that can be done on a lambda term:

1. α-renaming which allows the formal parameters of a lambda abstraction to be renamed, and will result in an equivalent lambda term. For example the term $\lambda x.x$ and $\lambda y.y$ are equivalent, as the only difference is the name of the bound variable, which can be done through an α-renaming.
2. β-reduce is the method to reduce a lambda term and states that $(\lambda x.e)f \to e[x \mapsto f]$ where $\to$ is "reduces to".

With these formal methods of computation, it is now possible to evaluate lambda functions. For example the identity function, $\lambda x.x$, is a function that returns it's argument. This can be seen by evaluating the term $(\lambda x.x)e$ to obtain e, shown below:

$$(\lambda x.x)e \to x[x \mapsto e] \equiv e$$

Although it is only possible use a β-reduce on an application of an abstraction and another term, it is possible to evaluate other lambda terms by apply β-reduction on the inner terms. For example a lambda term g can reduce to g' when any terms within g are β-reduced. This is shown below more formally, where it is possible to reduce $a \to_\beta a'$, where $\to_\beta$ is a β-reduction:

$$(ab) \to_\beta (a'b)$$

$$(ba) \to_\beta (ba')$$

$$(\lambda x.a) \rightarrow_\beta (\lambda x.a')$$

A lambda term is said to be in **normal form** if there is no more possible reductions to be made in the term. It is in **head normal form** if the form of the lambda term is $\lambda x_1 \ldots \lambda x_n.e_1 \ldots e_m$ and $e_1 \ldots e_m$ are arbitrary lambda terms. It is defined to be in **weak head normal form** when all terms are abstractions or of the form $ye_1 \ldots e_n$ where y is a variable and $e_1 \ldots e_n$ are arbitrary lambda terms.

2.2.2 Typed Lambda Calculus

Typed lambda calculus was an extension of the untyped lambda calculus where each lambda term, e, has a type, τ, and it expressed as $e : \tau$ where the expression is before the colon and the type is after.

1. The variable, x, can be said to have type τ and is written as $x : \tau$
2. The abstraction, $\lambda x.e$ has the type $(\sigma \rightarrow \tau)$ where the variable x has type σ and e has type τ. In other words the type $\sigma \rightarrow \tau$ is the type of the function when given an object of type σ returns an object of type τ.
3. The application, ab has the type τ only when a has the type $\sigma \rightarrow \tau$ and b has the type σ. This means that a has to be a function and b has to be the correctly typed argument to the function.

As the typed lambda calculus is built on the untyped lambda calculus, all of the untyped syntax and reduction methods are used, with extensions for the types. Two new lambda terms can be added as primitives, which also introduce two new types, which are named the product type and the sum type. The product type is a pair of objects and the syntax for the product typed lambda expression is $(a, b) : A \times B$ where a has type A and b has type B. The sum type, with syntax, $inl(a) : A + B$ or $inr(b) : A + B$, where a has type A and b has type B. The type represents that the inner object could be either type A or B. The two constructors inl and inr denote if the inner object has the left type, inl or the right type, inr, from the sum type $A + B$.

Typing Rules

The simply typed lambda calculus contains, **typing rules** which are used to ensure that a lambda expression is well-typed. Like Natural Deduction for logic, there are two types of typing rules, the first is for introducing an element of the type, called introduction rules, and the second is for using elements of the type, called elimination rules. It also takes a form similar to natural deduction inference rules in that the top half of the rule shows the premises required for the conclusion element to be well-typed.

The simplest typing rules are the product type rules, which are shown in Figure 2.10. The $\times I$ rule states that in order to form an element of type $A \times B$, then an element of type A and type B need to be given. There is two elimination rules, $\times E_1$ and $\times E_2$, as from an element of type $A \times B$, x, then it is possible to obtain the element $\pi_1\, x$, with type A, or the element $\pi_2\, x$, of type B. The π_1 and π_2 are ways to deconstruct the element x where $\pi_1\, x$ gives the first element of x and $\pi_2\, x$ gives the second element of x.

$$\frac{a : A \qquad b : B}{(a, b) : A \times B} \times I \qquad \frac{x : A \times B}{\pi_1\ x : A} \times E_1$$

$$\frac{x : A \times B}{\pi_2\ x : B} \times E_2$$

Figure 2.10: $\times$ Typing Rules

The typing rules for functions are given in Figure 2.11. The introduction rule, $\to I$, is how a function can be introduced such that it is well typed and states that if a variable x with type A can form a term e of type B, then it is possible to construct the lambda abstraction $\lambda x.e$ with type $A \to B$. As the variable x becomes bound in the lambda abstraction, it is similar to discharging the assumption in the Natural Deduction. The $\to E$ rule states that when given a function of type $A \to B$, f, and an element of type A named x then it is possible to obtain an element of type B through the application of f to x.

$$\frac{\begin{array}{c}[x : A]^1 \\ \vdots \\ e : B\end{array}}{\lambda x.e : A \to B} \to I^1 \qquad \frac{f : A \to B \qquad x : A}{f\ x : B} \to E$$

Figure 2.11: $\to$ Typing Rules

The typing rules for the sum types has two introduction rules, $+I_1$ and $+I_2$, and a single elimination rules, $+E$. The two introduction rules state that if there is an element from either of the types comprising the sum type, then it is possible to introduce an element of the sum type. If the element has the left type, then it is introduced with the *inl* constructor, while if it is the right type, the *inr* constructor is used instead. The constructor not only indicates what the inner type is, but also ensures the element a is treated differently to $inl(a)$ as they do have different types. The elimination rule introduces a new type of lambda term, which is the case statement. It works by scrutinising a term, x, and then two functions f and g which both must return the same type, C. It will then apply either one of the functions to the inner value of x depending on its type. If the element x uses the *inl* constructor, then the inner value will be applied to the function f. Conversely, if the *inr* constructor is used, then the inner value is applied to the function g. Regardless of which function the inner value is applied to, the resulting object will have type C, so the overall case statement will have type C.

$$\frac{a : A}{inl(a) : A + B} +I_1 \qquad \frac{b : B}{inr(b) : A + B} +I_2$$

$$\frac{x : A + B \qquad f : A \to C \qquad g : B \to C}{case\ x\ f\ g : C} +E$$

Figure 2.12: $+$ Typing Rules

2.3 Curry-Howard Isomorphism

There is a correspondence between the typing rules and the Natural Deduction inference rules. They not only look similar, but they actually are isomorphic to each other. This can be made more

clear by looking at a subset of the rules. Figure 2.13 shows the Natural deduction $\wedge$ inference rules and the $\times$ typing rules. If in the typing rules, only the types were kept, and the $\times$ symbol became the $\wedge$, then the rules would be identical. This means that it is possible to re-think what $x : A$ to mean x is a proof for the proposition A, rather than the object x has the type A. This shows that there is a relationship between the product type and logical conjunction.

$$\frac{A \qquad B}{A \wedge B} \wedge I \qquad \frac{a : A \qquad b : B}{(a, b) : A \times B} \times I$$

$$\frac{A \wedge B}{A} \wedge E_1 \qquad \frac{x : A \times B}{\pi_1\, x : A} \times E_1$$

$$\frac{A \wedge B}{B} \wedge E_2 \qquad \frac{x : A \times B}{\pi_2\, x : B} \times E_2$$

Figure 2.13: $\times$ Typing Rules

There is a similar relationship between functions and implication and also sum types and logical disjunction. This shows that there is a correspondence between propositional logic and the simply typed lambda calculus.

There is a similar correspondence that exists for predicate logic, but it required a new theory called type theory to be developed.

2.4 Type Theory

Type theory builds on the typing rules introduced in the simply typed lambda calculus, but introduces new rules that correspond to the $\forall$ and $\exists$ Natural Deduction rules. The new types introduced are called Dependant Types, with the $\exists$ quantifier corresponding to the dependant pair and the $\forall$ quantifier corresponding to the dependant function type. Brady[13] describes a dependant type as "a type that's calculated from some other values. More formally, it can be written as $\Pi_{x:A}B(x)$, and it can be seen that the type B depends on the value x. This non-dependant function type can be thought of as a special case of the dependant function type, which is the output type does not depend on x. The dependent pair can be expressed as $\Sigma_{x:A}B(x)$, and is a pair of objects, with the first being x of type A, and the second object has type $B(x)$, so B is a function that takes the input and returns a type, so, therefore, the type depends on the value of the first element of the pair. Similarly to how the non-dependant function type is a special case of the dependant function type, the product type is a special case of the dependant pair, when the second element's type does not depend on the first value x.

As equality is defined as a primitive relation in Predicate Logic, equality can be defined as a primitive type in type theory.

Typing Rules

The typing rules for the dependant function type are shown in Figure 2.14. The ΠI rule states that if it is possible to derive a proof of the Proposition B, which is indexed by x, then it is possible to introduce a lambda abstraction that has the dependant type $\Pi_{x:A}B(x)$. For the ΠE rule, if there is a function with a dependant type, and an element of type A, then it is possible, using an application of f and a, to obtain an element of the type $B[a/x]$ where the value a is substituted for the variable

x in B. The typing rules for dependant functions are very similar to the typing rules for functions shown previously in the simply typed lambda calculus section.

$$\frac{\begin{array}{c}[x:A]^1 \\ \vdots \\ e:B(x)\end{array}}{\lambda x.e : \Pi_{x:A}.B(x)}\,\Pi I^1 \qquad\qquad \frac{f:\Pi_{x:A}.B(x) \qquad a:A}{f\ a : B[a/x]}\,\Pi E$$

Figure 2.14: Π Typing Rules

The typing rules for the dependant pair are shown in Figure 2.15. The introduction rule, ΣI, states that it is possible to create a dependant pair when given an element of type A, a, and an element of type $P[a/x]$, where a is substituted for x in the type P. As this is a pair, like the product type, there is two elimination rules to obtain the first and second elements of the pair.

$$\frac{a:A \qquad p:P[a/x]}{(x,p):\Sigma_{a:A}P(a)}\,\Sigma I \qquad\qquad \frac{p:\Sigma_{a:A}P(a)}{\pi_1\ p : A}\,\Sigma E_1$$

$$\frac{p:\Sigma_{a:A}P(a)}{\pi_2\ p : P[a/x]}\,\Sigma E_2$$

Figure 2.15: Σ Typing Rules

As equality is a primitive type, it has its own typing rules, which are shown in Figure 2.16. The type is the proposition that the two values are equal. The introduction rule, $= I$, states that it is possible to construct the $Refl$ object with type $a = a$ and needs no premises as it always holds. Like in Predicate Logic, this rule shows is the reflexivity property of equality. The object, $Refl$, is an element of the equality type, and can only be introduced using the $= I$ rule. The $= E$ rule defines substitution can eliminate an equality, as it states if a element, p of type $a = b$ can be given and an element e, with type $A[a/x]$, it is possible to substitute a for b in the type of p.

$$\frac{a:A}{Refl : a = a} = I \qquad\qquad \frac{p:a=b \qquad e:A[a/x]}{p:A[b/x]} = E$$

Figure 2.16: $=$ Typing Rules

2.4.1 Example of Correspondence

To illustrate an example of the correspondence between Type Theory and Predicate Logic, a proof of the symmetric property is shown using Predicate Logic's Natural Deduction rules as well as Type Theory's typing rules are shown in Figure 2.17.

$$\frac{\dfrac{\overline{x=x}\,=I \qquad [x=y]^1}{y=x}\,=E}{(x=y)\to(y=x)}\,\to I^1$$

(a) Using Natural Deduction

$$\frac{\dfrac{\overline{Refl : x=x}\,=I \qquad [p:x=y]^1}{Refl : y=x}\,=E}{\lambda a.Refl : (x=y)\to(y=x)}\,\to I^1$$

(b) Using Typing Rules

Figure 2.17: Correspondence Example

Both look very similar and at each stage, they are using the equivalent rules from the natural deduction and typing rules.

2.5 pi-forall

`pi-forall` is a small, dependently typed programming language developed by Stephanie Weirich. It uses a syntax similar to Haskell[14] and is a strict, pure functional language. This means that functions in `pi-forall` have no side effects. This means that changes cannot be made by a function to the real world outside of the function. Examples of this would be reading user input or writing data to a file, as these are both affecting the world outside of the function. In a language like Haskell, which also has no side effects, they utilise monads[15] to handle side-effecting code.

`pi-forall` also has functions as first-class citizens, which means that functions can be passed around freely into and out of other functions and also be contained in user-defined data types. As the language is dependently typed, types are also first class, meaning that a function can take in a type or return a type. There is also a basic modularity system built in, as each pi file is itself a module and can be imported into other pi files.

2.5.1 Limitations

The language does not have as many features as other mainstream dependently typed languages due to aiming to be more of a tractable subset for researching about dependently typed language design. This means that there are many trade offs to keep the implementation small. The first is that the implementation only contains a type checker. The programs written in the language cannot actually be executed, they can only be checked for typing errors. The `pi-forall` type checker also contains limitations. There is no enforced termination[3], which can lead to logical inconsistencies in the proofs written in the language. This means that in `pi-forall` undefined statements can be proved.

Another limitation is that there is no type inference for implicit arguments[3]. In many more popular dependently typed programming languages, missing arguments are inferred using techniques such as unification, which `pi-forall` does not support. An example of this is shown in Figure 2.18, which shows that the `ConsV` constructor, on Line 3, the `m` variable is explicitly given with it's type. It is not able to infer the type of `m` or allow it to be a missing argument. The type of `m` could be inferred from the 3rd argument, `Vec A m`, which from the definition given in Line 1, `m` must be of type `Nat`. This limitation can cause issues with program expressiveness, as a larger program will be more complex, hence the number of explicit arguments that have to be given increases, which can make the `pi-forall` program very lengthy and verbose.

```
1 data Vec (A : Type) (n : Nat) : Type where
2     NilV of [n = Zero]
3     ConsV of [m : Nat] (A) (Vec A m) [n = Succ m]
```

Figure 2.18: `pi-forall` `Vec` definition

Another limitation is that the `pi-forall` type system is inconsistent. This is due to the typing rule for T-Type and was proven by Girard[16] in 1972. As the paradox arises due to the type of `Type` being `Type`, it can be avoided by adjusting the typing rule and adding universes, where a universe's type is not itself, rather its type is that of a higher universe. This means that the type `Type` would no longer contain `Type`.

Another limitation is that `pi-forall` uses a technique based on weak-head-normalisation which is the most simple method of checking equality. The current standard is to use a technique based on normalisation-by-evaluation which is a much stronger method when compared with weak-head-normalisation.

2.5.2 Type Checking

`pi-forall` has a type checker that is implemented according to a set of typing rules that are defined in the documentation of `pi-forall`[3]

There is a single function called `tcTerm` which will take in the tree and type check each part of the tree. It will either return the same tree passed in, meaning the check was successful, or an error will be produced, as there is a type error.

2.5.3 Syntax

BNF Grammar

```
Concrete syntax for the language:
Optional components in this BNF are marked with < >

  terms:
    a,b,A,B ::=
      Type                    Universes
    | x                       Variables   (start with lowercase)
    | \ x . a                 Function definition
    | a b                     Application
    | (x : A) -> B            Pi type

    | (a : A)                 Annotations
    | (a)                     Parens
    | TRUSTME                 An axiom 'TRUSTME', inhabits all types
    | PRINTME                 Show the current goal and context

    | let x = a in b          Let expression

    | Unit                    Unit type
    | ()                      Unit value

    | Bool                    Boolean type
    | True | False            Boolean values
    | if a then b else c      If

    | { x : A | B }           Dependent pair type
    | A * B                   Nondependent pair syntactic sugar
    | (a, b)                  Prod introduction
    | let (x,y) = a in b      Prod elimination

    | a = b                   Equality type
    | Refl                    Equality proof
```

```
  | subst a by b             Type conversion
  | contra a                 Contra

  | C a ...                  Type / Term constructors
  | case a [y] of            Pattern matching
      C1 [x] y z -> b1
      C2 x [y]   -> b2

  | \ [x <:A> ] . a          Irr lambda
  | a [b]                    Irr application
  | [x : A] -> B             Irr pi

declarations:

    foo : A                  Function Type Decleration
    foo = a                  Function Implementation

    data T D : Type where    Datatype Decleration
       C1 of D1              Constructor Declerations
       ...
       Cn of Dn

telescopes:
  D ::=
                             Empty
   | (x : A) D               runtime cons
   | (A) D                   runtime cons
   | [x : A] D               irrelevant cons
   | [A = B] D               equality constraint

Syntax sugar:

 - You can collapse lambdas, like:

       \ x [y] z . a

   This gets parsed as \ x . \ [y] . \ z . a
```

Variables

Figure 2.19 shows how a variable, called x, can be defined in pi-forall. The variable `x` has type `A` and is equal to the expression `a`.

```
1 | x : A
2 | x = a
```

Figure 2.19: `pi-forall` variable syntax

Functions

The syntax for writing a function in `pi-forall` is shown in Figure 2.20. It has two parts, the first is the type signature and the second is the function's implementation. The type signature describes the type of the function in terms of types of inputs and outputs. It is worth noting that in Haskell the type signature of a function follows the `::` operator; in `pi-forall` only a single colon is used. As `pi-forall` allows currying, the output of the function depends on the number of inputs given. This means that `f`, the function example given in Figure 2.20, will only return an element of type `A_n` when all inputs are given. If the function is not fully applied, i.e. it is partially applied, then a function will be returned. For example, if all inputs were given except the last, which has type `A_n-1`, then `f` will return a function of type `A_n-1 -> A_n`. For the function implementation, the arguments given to the function are bound to parameter variables. In Figure 2.20, `a_1,...,a_n-1` are the formal parameters what the arguments applied to the function `f` are bound to. Once the inputs have been bound, the '.' is used to denote that the function body has started, and some expression will follow.

```
1 f : A_1 -> A_2 ->... -> A_n-1 -> A_n              -- Type Signature
2 f = \ a_1 a_2 ... a_n-1 . <implementation>        -- Function Implemenation
```

Figure 2.20: `pi-forall` function definition syntax[3]

Application of a function to an argument is done by placing the argument immediately after the function, so `f a_1 ... a_n` would apply the arguments `a_1, ..., a_n` to the function `f`.

Datatypes

The syntax for writing a simple datatype in `pi-forall` is shown in Figure 2.21. This would be used to write data types such as `Bool` or `Nat`.

```
1 data T : Type where
2   K1                      -- No args
3   K2 of (A)               -- single arg of type A
4   K3 of (x : A)           -- similar to K2 but A is named x
5   K4 of (x : A) (y : B) -- two arguments, the type B can mention value x
6   K5 of (x : A) [x = a]
7     -- K5 contains a single argument of type A and also an equality
8     -- constraint that constrains the value of x to be equal to value a
```

Figure 2.21: `pi-forall` simple datatype syntax [3]

To add indexes to the datatypes, the syntax is extended by depending on values `A_1` to `A_n`, which are of types `B_1` to `B_n`.

```
1 data T (A_1 : B_1) ... (A_n : B_n) : Type where
2   -- Constructors are similar to as before and values A_1 ... A_n are
    allowed to appear in constructor definitions
```

Figure 2.22: `pi-forall` parameterised datatype syntax [3]

An example of the generic type `Maybe` is shown in Figure 2.23. The `Maybe` type is generic of any type `A` as from the type definition on Line 1, `Maybe` depends on the type `A` and refers to `A` in the `Just` constructor.

```
data Maybe (A : Type) : Type where
    Nothing          -- Empty constructor
    Just of (A)      -- Single argument constructor of type A
```

Figure 2.23: `pi-forall` parameterised Maybe datatype definition

A generic datatype definition is already a dependent type. It just has the restriction that the values the type depends on are also types. If that rule is relaxed, so that the values the type depends on are no longer types, then true dependent types can be defined. An example of this was shown in the Introduction Chapter and the previous Limitations Section, Section 2.5.1, in Figure 2.18. It has been shown again in Figure 2.24. The `Vec` type is a more advanced specification of the `List` type and it is indexed by the type `A` and also `n` of type `Nat`, which represents the length of the `Vec`. This differs from the generic `Maybe` type as one of the indices of `Vec` is of type `Nat` as opposed to the type `Type`.

```
data Vec (A : Type) (n : Nat) : Type where
    Nil  of                              [n = Zero]
    Cons of [m : Nat] (A) (Vec A m) [n = Succ m]
```

Figure 2.24: `pi-forall` Vec datatype definition

`pi-forall` uses the datatype definition to add some extra types to the language, which are: `Bool` ,`Sigma`, `Unit` and `Nat`. `pi-forall` bakes them into the syntax tree of any `pi-forall` program so they are not actually defined as code, however the code for the types has been translated from the abstract syntax tree form into `pi-forall` code as shown in Figure 2.26. The reason for adding these types is so that syntactic sugar can be added to the language. For example, any number literal such as 1, will be parsed and transformed into the equivalent `Nat` type, such as `Succ (Zero)`, which tries to limit the verbosity of writing numbers. Another example is the syntactic sugar for `Sigma`, where it is easier to write dependant pairs and product types.

To use a constructor to create an element of the type, its arguments need to be fully satisfied. This just means it appears like a function call where the constructor name is the function and the argument gets applied to it. As it is fully satisfied, there is no partial application of constructors. An example would be an element of type `Nat`, `Succ Zero`.

Pattern matching is how a datatype can be deconstructed to obtain the data within the constructor's arguments. An example is shown in Figure 2.25. For each matched constructor, the arguments will get bound to the variable name in the pattern.

```
case n of
    Zero    -> a
    Succ n  -> a
```

Figure 2.25: `pi-forall` Pattern Matching a `Nat`

Irrelevance

`pi-forall` has a feature where some arguments in a function or a datatype can be made 'irrelevant'. This just means that the arguments are not needed when executing the program and are only used during type checking. Irrelevant arguments are marked by surrounding the argument in square brackets, as shown in Figure 2.27, where the first argument is marked as irrelevant. The input will

```
data Bool : Type where
    False
    True

data Sigma (A: Type) (B : A -> Type) : Type where
    Prod of (x:A) (B x)

data Unit : Type where
    ()

data Nat : Type where
    Zero
    Succ of (Nat)
```

Figure 2.26: `pi-forall` pre-defined datatypes

still need to be bound in the function body, but the variable will also need to be wrapped in the square brackets.

```
id : [A : Type] -> A -> A
id = \ [A] a . a
```

Figure 2.27: `pi-forall` irrelevance example

Irrelevant arguments cannot be used in the function body in a relevant way as the program would not type check. This is because if an irrelevant argument was used in a relevant way, it could be used in a computation that is done during execution. This would lead to issues, such as what the value of the argument is, as it is erased everywhere else when doing the computation.

A simple motivation for utilising erasure is that a programmer may write code that relies heavily on proofs during type checking to ensure the correctness of the program, but when actually running the program does not want the executable to be bloated with the proofs that are not longer needed. Allowing the flexibility of some arguments to be erased will help to create a more efficient executable.

The `pi-forall` method of erasure is one of the most simple methods, by making the programmer manually tag what should be erased. Other dependently typed programming languages have different methods of erasure which is discussed more in the Related Work chapter (Link).

Equality

Unlike with the other pre-defined types like `Bool` and `Nat`, `pi-forall` does not define the equality type using the datatype syntax, like how Idris[13] does. Instead, it made the type its own primitive type, with the only constructor `Refl` made into a keyword. This means that when type checking, it is not checked like a user-defined datatype, instead it has its own typing rules.

To write the type in `pi-forall` involves two terms and requires the `=` operator to be placed in between the terms, an example being `True = True`. As the type is the propositional equality of the two propositions, an element of the type can only be constructed when the two values are equal and this is done using the `Refl` keyword. This is because a property of equality is that it is reflexive, which means $\forall x.x = x$ where x is any term. `Refl` is a proof that the two terms in the type are equal. It is a much stronger statement that a boolean expression checking if they are equal, as it

is a proof that they are defined to be always equal. If they are not equal then the type checker is unable to construct the equality type using the `Refl` keyword and the equality type is uninhabited. This will cause a type error and the program will be rejected by the type checker. An example of this is shown in Figure 2.28.

```
1 t : True = True
2 t = Refl    -- Type checks, so therefore the equality holds
3
4 f : False = True
5 f = Refl    -- Can never type check, so equality does not hold
```

Figure 2.28: `pi-forall` equality example

If a new type is created called `Void`, shown in Figure 2.29, that contains no constructors then we can show equalities can never hold. As there are no constructors of the type, it is an uninhabited type and so a value with type `Void` can never be constructed.

```
1 data Void : Type where
2     -- No constructors so type is uninhabited
```

Figure 2.29: `pi-forall` Void datatype definition

If a function had the type `a = b -> Void`, then this is a proof that `a = b` does not hold. This is because when the function is given a proof that `a = b`, no value can possibly be constructed, hence the `Void` type. To write the implementation of this function, then a contradiction must be found in the proof of `a = b`. To find a contradiction in `pi-forall`, the `contra` keyword is used. The `contra` keyword takes a single argument, which must be an equality type. It will evaluate the two terms to weak-head-normal-form (whnf) and check if there is there is a contradiction by checking if the constructors are different. For example the `Bool` type has two constructors `True` and `False`, which means that when evaluated to whnf, the contradiction can only be found when either of the equality types were `True = False` or `False = True`, which is shown in Figure 2.30. The Figure also shows that if the types are equal, then no contradiction can be found. The method of evaluating only to whnf does introduce problems. For example, with the `Nat` type, multiple values can be constructed using the `Succ` constructor. The type `1 = 2`, which using `Nat` constructors would be equivalent to `Succ (Zero) = Succ (Succ (Zero))`. Both terms start with the same constructor when evaluated to whnf, so no contradiction would be found. Instead, lemmas would be needed to prove that `1 = 2 -> Void`.

```
1 trueStatment : True = False -> Void
2 trueStatment = \pf . contra pf -- Type Checks
3
4 falseStatment : True = True -> Void
5 falseStatment = \pf . -- Can never type check as the void type is never
    produced
```

Figure 2.30: `pi-forall` contradiction examples

`pi-forall` uses substitution to convert the type of a term to another, when they are proven to be equal. The syntax for this is `subst a by b`, where b has type `a_1 = a_2` and the substitution will

substitute all occurrences of `a_1` with the equivalent `a_2` in `a`. With just these rules, powerful things about equality can be proven. In Figure 2.31, the function `sym` proves that propositional equality is symmetric. The only relevant argument the function is given is called `pf` and has the type `x = y`. By using `Refl`, which will have type `x=x` due to reflexivity, a substitution can happen where the the `x` can be substituted for `y` to convert the type of `Refl` to become `y = x`.

```
1 sym : [A : Type] -> [x : A] -> [y -> A] -> (x = y) -> (y = x)
2 sym = \ [A] [x] [y] pf . subst Refl by pf
```

Figure 2.31: `pi-forall` symmetric property example

2.6 Parallelism

2.6.1 Algorithmic Skeletons

Algorithmic Skeletons[2] are high-level parallelism models with many advantages compared to using lower libraries to introduce parallelism to a program. Although using a lower-level library would afford a programmer more control and potentially more computation speed in the final program, more control can mean more bugs that arise as the programmer has to consider synchronisation and communication between all the threads and processes. By introducing skeletons, which take care of the communication and synchronisation for the user, the parallelism behaviour in a program becomes much more predictable.

Another advantage the skeletons provide is that they can make the code more reusable. The level at which the parallelism is introduced will affect the code's modularity. If a low-level library such as pthreads[17] was used, there is a high chance that the parallelism code will become tightly coupled with the program's business logic. By using the skeletons, to fully take advantage of the parallelism, they can force the programmers to think about using them more modularly, hence making the parallelism sections more decoupled from a program's business logic. Another implication of using skeletons is that some parallel-specific bugs, such as race conditions, can be avoided in a program as the synchronisation is taken care of by the skeleton and not the programmer.

The most simple algorithmic skeleton to understand is the Farm skeleton, with a diagram shown in Figure 2.32, which, when given an input, will split that input among several workers, where each will apply the same function to the input and then pass it out to be collected. This skeleton is used when some task is completely independent of every other task, hence why running the tasks on different threads does not affect the results. One thing to note about the Farm skeleton is that it will not preserve the order of the list, as the ordering depends on the order the collector receives from the workers.

The other simple parallel pattern is the Pipeline pattern, with a diagram shown in Figure 2.33, which, when given an input list, will apply a series of functions to the input to obtain a new list. Each stage would have a function associated with it, and it would take the previous stage's output as input. Each element of the list will become $f_n(...(f_2(f_1(x))))$ where $f_1, f_2, ...f_n$ are the stage functions and x is an input. The parallelism arises from each stage being executed in parallel.

2.6.2 Composing Skeletons

Although only the Farm and Pipeline have been defined, there are countless variations of them, as well as other basic skeletons, but one of the more powerful aspects of algorithmic skeletons is the

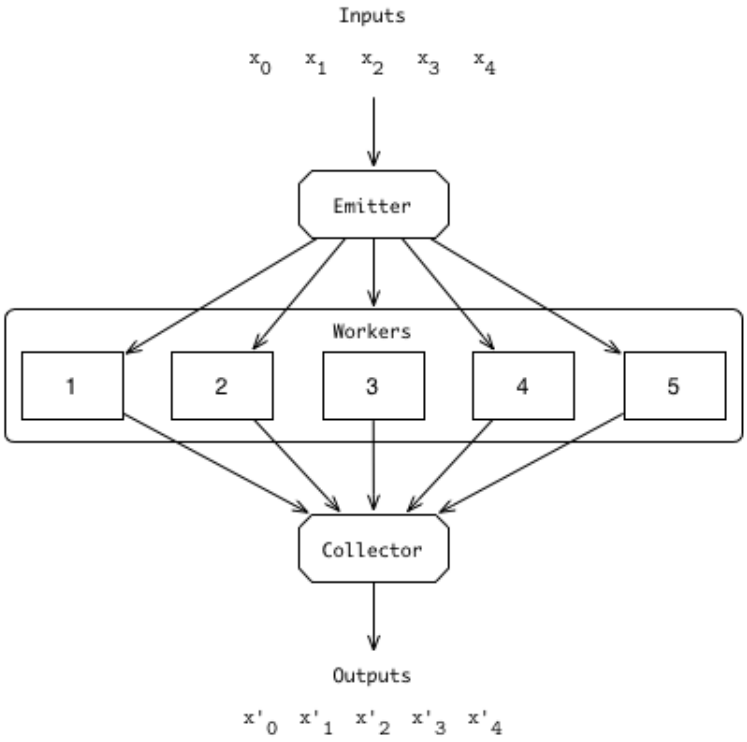


Figure 2.32: Farm Pattern[18]

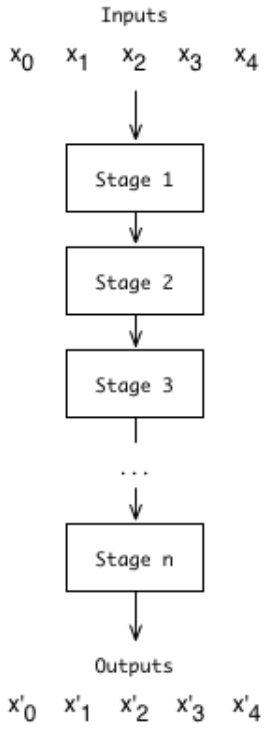


Figure 2.33: Pipeline Pattern[18]

ability to compose them together. Each of the basic skeletons can be combined to produce even more complex skeletons, which could also be composed again so that a skeleton could be created to best suit a problem. For example, each worker in a Farm skeleton could be a Pipeline skeleton, or a stage in a Pipeline could be a Farm. Although these may seem like basic compositions, with a larger array of skeletons, many possible skeletons can be created.

2.6.3 Issues with Skeletons

One of the main issues with skeletons is when the problem does not map well to any skeletons. This would mean that it would be difficult to gain any speedup from using the skeletons, and a more tailored approach would be needed, although this is rare.

A more common issue that arises with skeletons is their misuse. For example, the granularity of problems can cause issues. For example, if a Farm was used to add 1 to a million numbers, then the overhead of the skeleton is much larger than the computation. This is when the granularity is too small, so introducing parallelism might not introduce speedup but slowdown instead. Another example is the regularity of tasks or input. For example, suppose a Pipeline has one stage that takes much larger due to the function or the input. In that case, the irregularity issue will slow down the entire pipeline as the stages in front will have no input to consume and will be left idling.

These issues can be overcome through proper use and understanding. In the case of small granularity of input, chunking inputs together to make the task larger would help improve the issue. If there is irregularity, finding where it is and using other patterns can help solve the issue. For example, if one stage takes twice as long as the others, using a Farm with 2 workers embedded in the stage, potentially taking half the time, would solve the issue.

2.7 Summary

In this chapter, the relevant background knowledge for the dissertation has been described, starting with Logic and Lambda Calculus, as well as talking about the relationship between them, as well as the extension to typed Lambda Calculus, Type Theory. It also describes `pi-forall`, the dependently typed programming language this dissertation builds upon. Finally, it gives a definition and examples of basic Algorithmic Skeletons.

Chapter 3

Related Work

In this chapter there is a discussion on some of the related work to this dissertation. As there is not really any related work that combines algorithmic skeletons and dependant types, they have been split, so Section 3.1 covers related work in dependently typed languages and 3.2 discusses related work with algorithmic skeletons.

3.1 Dependently Typed Languages

There are many dependently typed languages that exist, with the most popular ones being Idris[19], Agda[20] and Coq[21].

Idris, like `pi-forall` is a pure functional language with strict evaluation. It also has Haskell inspired syntax. Unlike `pi-forall`, it enforces termination checking, so any proofs written in the language are correct. It also has type unification, so that missing arguments can be inferred. It is one of the most popular dependently typed programming languages that is used to produce software, rather than being used mainly as a proof assistant.

Agda is designed to be a proof assistant, but can also be used to write software with. It has many of the advantages Idris has over `pi-forall`.

While Coq is a programming language, it is mainly used as an interactive theorem prover, and has a large library of modules which help support this.

3.2 Parallel Patterns

3.2.1 Pattern Libraries

There are many implementations of parallel patterns or a range of different programming languages, with some notable implementations described below.

Eden[22] is parallel programming language that extends from Haskell. It defines a process type which programmers can use to create parallel functions, and all of the communication is handled by the run time system. It is very high level parallel language as programmers do not care at all how the communication is used to achieve the parallelism, and they also do not have to worry about synchronisation. There is also a library that contains skeleton implementations which can be used

to introduce parallelism as well. A programmer is also able to implement their own skeletons that might better suit the program.

Skel[4] is a streaming parallel skeleton library to be used in Erlang programs. The library contains a set of many of the most common algorithmic skeletons. It was originally written so that sequential Erlang programs could be refactored into a a parallel program using alrogithmic skeletons.

GrPPi[5] is a generic parallel programming interface which can be used to write algorithmic skeletons in C++. The patterns are generic over different C++ parallelism libraries, such as ISO C++ thread, OpenMP[23], Intell TBB[24] and FastFlow[25]. This means that with the same C++ code, different runtime libraries can be used by changing the backend, and the skeleton code is fully decoupled from the execution code.

Chapter 4

Software Engineering Process

This chapter gives an overview of the software engineering processes used throughout development.

4.1 Agile Software Development

Throughout the development of this project, agile software development practices were followed. Features were implemented and then continuously iterated on until they met the requirements.

There were also meetings twice a week with the supervisor to ensure that the project's progress was on track and to talk about the next steps in the development cycles.

As modelling concurrency and creating parallel patterns has not been explored in a dependently typed language, the design for those sections had to be constantly iterated on, which was aided by the agile workflow. This also made it straightforward to pivot into new designs.

4.2 Testing Approach

Testing was done throughout the project's development; however, due to the ever-changing nature of the design, unit tests were never written, and testing was much more informal. For example, when testing the latest feature in the translator, some sample programs would be translated to check that it works correctly.

There was one area that had some systematic testing, which was in the `pi-forall` type checker and was provided in the initial source code. This had a suite of tests used as regressions tests when changing the type checker to ensure no change broke the type checker. It is now impossible to run the regression tests due to the changes made by the translator.

Chapter 5

Translation

In this chapter, a discussion on how the translation from `pi-forall` to C++ was designed and implemented will be given. Section 5.1 discusses the overall design of the translator and Section 5.2 details how the decision on C++ being the backend language was reached. Section 5.3 gives the `pi-forall` Abstract Syntax Tree's definition and Section 5.4 discusses how the AST was annotated with extra information required during the translation. Section 5.5 defines an Intermediate Representation used during the translation, while Section 5.6 discusses the key design decisions made in the translator design. Section 5.7 talks about the state required during the translation and Section 5.8 describes the functions used to actually translate the program. Section 5.9 describes how after the translation, the code is generated into a file. Section 5.10 shows the way in which pi-forall is extended to make the next objectives easier. Finally Section 5.11 discusses the many limitations in the translation and Section 5.12 outlines areas for future development, based the translation system created.

5.1 Overall Design

The overall design for the translation can be split into two sections. The first part will walk the abstract syntax tree and convert it into an intermediate representation, and the second part will generate the C++ code from the intermediate representation.

5.2 Backend Language Choice

When choosing what language to compile the `pi-forall` code to, there were a couple of choices. The first was Erlang. Erlang is a general-purpose programming language that is optimised for concurrent programming. It runs programs on the Erlang runtime system, and processes spawned in Erlang are lightweight and completely isolated from other processes. This means that no data is automatically shared between processes, and the only method of inter-process communication is done through message passing. As algorithmic skeletons would be utilising channels for data communication, message passing would be a good feature to have. Erlang is also a functional programming language, like `pi-forall`, which could make the translation easier.

The other option for language choice was C++, which is another general-purpose programming language and was originally designed to be an extension to C, but with classes, to make it an Object Oriented Programming language. Modern C++ has evolved with many new features, with new functional features such as closures being implemented. An advantage to C++ being used

would be that there would be much more control in how the parallelism is eventually implemented. C++ is also a language that I have much more experience in, which also factored into the decision.

Overall the main advantage of Erlang was that the parallel runtime system would be implemented already, as the Erlang runtime system, and with C++, it would need to be implemented. With the dissertation, I wanted to explore designing a runtime system, and the primitives from the ground up, hence C++ was chosen.

5.3 Abstract Syntax Tree

`pi-forall` parses a program into a `Module` object that is shown in Figure 5.1. For the translation, the only important piece of information is on Line 4, `moduleEntries`, and stores a list of `Decl` objects which is the abstract syntax tree of the program. The other information is irrelevant to the translation so will not be explained. The `Decl` type is defined in Figure 5.2 and represents a `pi-forall` declaration. Line 2 shows the `TypeSig` constructor which represents the declaration for the type of a term and has an argument of type `Sig` which contains the type signature information. The `Def` declaration represents the definition of a function with the name being represented by the `TName` and the function body is represented in the `Term`. The `RecDef` is similar to the `Def` constructor, but instead represents a recursive function, where the name of the function will appear in the `Term`. The `Demote` constructor is only used during type checking to change the relevance of a `Term`. The `Data` constructor represents a `pi-forall` data type and also includes the relevant data for each constructor, with the `TCName` being the name of the datatype, `Telecope` being the context for the type, which is where the indexes are contained, and `[ConstructorDef]` is a list of constructor definitions. The `DataSig` constructor represents just the type signature of the datatype, so it contains the same information as the `Data` constructor but without the constructor definitions.

```
1 data Module = Module
2   { moduleName :: MName,
3     moduleImports :: [ModuleImport],
4     moduleEntries :: [Decl],
5     moduleConstructors :: ConstructorNames
6   }
```

Figure 5.1: `pi-forall`'s Module Definition

```
1 data Decl
2   = TypeSig Sig
3   | Def TName Term
4   | RecDef TName Term
5   | Demote Epsilon
6   | Data TCName Telescope [ConstructorDef]
7   | DataSig TCName Telescope
```

Figure 5.2: `pi-forall`'s Decl Definition

The `Term` is an object that every `pi-forall` expression will be translated into. It is also important to note that there is no distinction between the `Term` and `Type`, and are defined to be synonymous. This is because in `pi-forall` any term can be a type, or a type can be a term.

Figure 5.3 shows the syntax for the `Term`, and it is described below:

```
data Term
  = Type
  | Var TName
  | Lam Epsilon (Unbound.Bind TName Term)
  | App Term Arg
  | Pi Epsilon Type (Unbound.Bind TName Type)
  | Ann Term Type
  | Pos SourcePos Term
  | TrustMe
  | PrintMe
  | Let Term (Unbound.Bind TName Term)
  | TyEq Term Term
  | Refl
  | Subst Term Term
  | Contra Term
  | TCon TCName [Arg]
  | DCon DCName [Arg]
  | Case Term [Match]
```

Figure 5.3: `pi-forall`'s Term Definition

- `Type` represents the type of all types.
- `Var` represents a variable, where `TName` is its name.
- `Lam` represents a lambda abstraction, e.g `\x . a` where the variable name `x` is bound in the `Term a`. The `Epsilon` contains the relevance information of the variable `x`.
- `App` represents a function application of a `Term` to an `Arg`, where `Arg` is defined in Figure 5.5.
- `Pi` represents a `Pi` type.
- `Ann` represents an term that is annotated with a type, such as `(x : Nat)`.
- `Pos` represents the source code position information for a `Term`.
- `TrustMe` represents the `pi-forall` `TRUSTME` expression.
- `PrintMe` is the `pi-forall` representation of the `PRINTME` expression.
- `Let` represents let expression that can introduce a new expression that is bound to a name
- `TyEq` represents the equality type, `(a = b)`.
- `Refl` represents the `pi-forall` `refl` keyword.
- `Subst` represents the substitution of an equality in an expression.
- `Contra` represents the `pi-forall` `contra` keyword.
- `TCon` represents the a user defined type, such as `Maybe Nat`.
- `DCon` represents a data type constructor and its arguments, such as `Just 1`.
- `Case` represents the pattern match expression on a data type, and there is a list of `Match` which

contains the match information.

Figure 5.4 shows the types used to represent names in the tree. The only interesting one is the `TName` which ensures that the term is named by using the Unbound library.

```
1  type TName = Unbound.Name Term
2
3  -- | module names
4  type MName = String
5
6  {- SOLN DATA -}
7  -- | type constructor names
8  type TCName = String
9
10 -- | data constructor names
11 type DCName = String
```

Figure 5.4: `pi-forall`'s Name Definition

Throughout the `Term` definitions, the `Arg` and `Epsilon` type appear, and they have been defined in Figure 5.5. The `Epsilon` type only has two constructors and is used to track relevance. The `Arg` type is a record containing the information on a single argument, such as its relevance information and the term for the argument.

```
1 data Arg = Arg {argEp :: Epsilon, unArg :: Term}
2 data Epsilon
3   = Rel
4   | Irr
```

Figure 5.5: `pi-forall`'s Arg and Epsilon Definition

The `Match` type represents a single pattern match within a case statement, and the type is shown in Figure 5.6, along with the `Pattern` type that represents the pattern being matched. The match contains two pieces of information, the pattern being matched and the term. Note that the pattern is bound inside the term, as the pattern can contain variable bindings that appear in the term. The pattern has two constructors; in a pattern match, only a constructor or variable can appear. For example, the match `Succ n` would use the `PatCon` constructor to represent the `Succ` and the `PatVar` constructor to represent the variable `n`.

```
1 newtype Match = Match (Unbound.Bind Pattern Term)
2 data Pattern
3   = PatCon DCName [(Pattern, Epsilon)]
4   | PatVar TName
```

Figure 5.6: `pi-forall`'s Match and Pattern Definition

Figure 5.7 shows the definitions of the `Sig` type, which represents a type signature. It includes the name, the relevance and the type that it represents. Note that `Type` is synonymous with `Term`, so it will be the term representing the type.

Figure 5.8 shows the definition of the `Telecope` and `ConstructorDef` type, both of which appear in the `Data` and `DataSig` constructor for the `Decl` type. The telescope represents a context with a list

```
1 data Sig = Sig {sigName :: TName, sigEp :: Epsilon, sigType :: Type}
```

Figure 5.7: `pi-forall`'s Sig Definition

of declarations, where each declaration can appear later on in the list. In the case of a data type, it is used to store each of the types that index a dependent type. For example, the `Vec` type is indexed by a `Type` and a `Nat`. Both of those would be in the `Decl` list as a `TypeSig`. The `ConstructorDef` only has two important arguments, the `DCName` representing the constructor name, and also a `Telescope`, which will contain the data types telescope extended with the arguments for the constructor. The reason it is extended is so that the indexes of the data type can appear in the constructor's terms.

```
1 newtype Telescope = Telescope [Decl]
2
3 data ConstructorDef = ConstructorDef SourcePos DCName Telescope
```

Figure 5.8: `pi-forall`'s Telescope and ConstructorDef Definition

5.3.1 Examples

To help explain the AST, a simple example of translating `pi-forall` code into the AST has been given in this section. For brevity, only the `Decl` list will be shown from the `Module` type defined in Figure 5.1.

Figure 5.9 shows a function names `doSomething` which has three arguments, the first being the type `A`, the second and thirds are of type `A` and the function will return an element of type `A`. From the function's implementation, the function will take in the arguments and return the second argument, `a`.

Below the `pi-forall` code is the equivalent AST. There is a `TypeSig Decl`, encoding Line 1 of the `pi-forall` code, and the `Def Decl`, encoding Line 2. For the `TypeSig` the important information is contained in Line 5 of the AST, which is the `sigType`. From there, the `Pi Term` is used to represent the function type. Note that the variable `A` appears in the later `Pi` types. As the second and third argument is unnamed, the variable names in the `Pi` term are just synthetic, `<_>` and `<_1>`. The relevance information can also be seen in the first `Pi` term `Irr` appears, which corresponds to the irrelevant argument `A`, while the rest are `Rel`. The `Def` starting on Line 6, contains the AST for the functions `doSomething`. The `Lam` term appears three times so that the three variables can be bound; this is because a `Lam` only has a single variable being bound. The term in the final lambda is a `Var` which represents the `a` from the `pi-forall` code. The reason `a` does not appear in the `Var` is because the object is not a string but a `TName`, which is a bound name.

```
doSomething : [A : Type] -> A -> A -> A
doSomething = \ [A] a b . a
```

```
[
    TypeSig (Sig {
        sigName = doSomething,
        sigEp = Rel,
        sigType = (Pi Irr Type (<A> Pi Rel A (<_> Pi Rel A (<_1> A))))},
    Def doSomething
        (Lam Irr (<A>
            (Lam Rel (<a>
                Lam Rel (<b>
                    (Var 1@0)))))
]
```

Figure 5.9: `pi-forall`'s doSomething AST Example

5.4 Annotating AST

During the translation, it was found that just having the AST was not enough. There were instances where the type of a term were needed, such as the arguments to a function, but it was never given in the tree. This meant that the tree had to be annotated with the type of each term at each point. This involved changing the `Def` and `RecDef` constructors for the `Decl` type as shown in Figure 5.10. Instead of just a `Term`, both have now become a tuple of the same `Term` with the second element being a stack of `Term` objects that correspond to the type of each term. with the tree `Term`.

```
data Decl
  = TypeSig Sig
  | Def TName (Term, [Term])
  | RecDef TName (Term, [Term])
  | Demote Epsilon
  | Data TCName Telescope [ConstructorDef]
  | DataSig TCName Telescope
```

Figure 5.10: `pi-forall`'s Annotated Decl Definition

This was done by adjusting every type-checking function so that the type checker would no longer return only the AST but the annotated AST, where at every node there is a stack of types that could be traversed to get the annotated types.

5.5 Intermediate Representation

In the original design, an Intermediate Representation was designed and implemented. The reason for this was when translating to C++, it was not possible to immediately translate every line of `pi-forall` to C++. Some information from other sections of the code would be needed. For example, the name of the datatype for a constructor was needed when translating `pi-forall` case statements. The intermediate representation was, therefore, a structure that not only contained parts of the translated C++ code, but it also contained extra information needed, for example, it served as an easy method to search the user-defined data types and their constructor. Figure 5.11 shows the intermediate representation structure. In the syntax tree, there is 6 possible Decl that need to be translated. The first is the TypeSig, which contains the type signature information for a `pi-`

```
data InterDef =
  FunctionDef {
    functionName :: String,
    templateArgs :: [String],
    args :: [String],
    returnType :: String,
    definition :: String }
  |
  FunctionImpl {
    functionName :: String,
    argNames :: [String],
    cLines :: [String] }
  |
  DataType {
    typeName :: String,
    constructorNames :: [String],
    cLines :: [String] }

```

Figure 5.11: Haskell Intermediate Representation design

`forall` function. This will be translated to the FunctionDef record and contains the function name, the template arguments, and the arguments to the function's return type. Finally, it contains the definition field, which is the C++ code that is the function declaration that can be used in a header. The next Decl type is Def, which contains the function's definition sub-tree. This will be translated into a FunctionImpl record, which contains the function name, the argument names and finally, the translated C++ lines. Another Decl type is the RecDef, the constructor used when the `pi-forall` function is recursive. The translator makes no distinction between recursive and non-recursive functions and will translate them the same. This means that a new intermediate representation is unnecessary, and the FunctionImpl can be used for RecDef. Like functions, datatypes are split into two Decls, with the first DataSig containing the datatype's type signature, and the Data containing the definition of the datatype. Of the two, the DataSig is actually not needed as all the information it contains is contained in the Data constructor, so each Data Decl is translated into a DataType record.

This was planned to be removed, as towards the end of the project an easier approach was found, but due to appearing in every function, there was too much technical debt to remove it without a complete rewrite.

5.6 Key Decisions

There needed to be some key design decisions that would be used throughout the translation.

5.6.1 Functions

The initial design for translating a `pi-forall` function was to just translate it to a C++ function, however, this quickly leads to problems. For example, `pi-forall` functions are higher-order objects, meaning that they can be passed into and out of functions, which is possible with standard C++ functions but can quickly become messy as they are passed around as function pointers. There is also an issue with the partial application of functions. For example, in `pi-forall` it is possible

to define a function plus, with type Nat→Nat→Nat, that adds two numbers. If applied to the number 1, it will result in a function with type Nat→Nat that adds one to the number given as its argument. With a normal C++ function, the function arguments must be fully given; hence partial application is impossible.

The final solution was to use the `std::function` type, which is, allows a function pointer to be passed around easily, solving the function pointer problem, and also to make every function, regardless of arguments, have only a single parameter, and return a function, which solves the partial application problem. This is because if only a single argument is provided, for example, a number in the plus example, then a function that is returned expects the second number. This is exactly what the `pi-forall` behaviour is expected to be, hence this was the final solution. An example of the translated plus function is shown in Figure

```
1 std::function<std::function<int(int)>(int)> plus =
2     [](auto a) {
3         return [a](auto b) {
4             return a + b;
5         };
6     };
7
8 plus(1) // Partial application, returns a function to add 1
9 plus(1)(2) // Fully applied, returns 3
```

Figure 5.12: C++ Function Example

There is also no concept of a main function in `pi-forall`, instead a function that has no input such as `main : A`, also known as a variable, would be evaluated when the C++ code is run. By evaluating the variable through another function call, the variable is acting similar to a main function, in that it is evaluated when the program is run.

5.6.2 Expressions

When translating an expression which contains sub-expressions, a decision was made that the result from each sub-expression will always be stored in a variable, and the variables will then be used in the translated main expression. For example, if translating a function application, `f a b`, then it would look something like the Figure 5.13. This means that if `a` or `b` are more complex expressions that also require multiple lines, then there is room above Line 1 for that code.

```
1 auto _1 = a;
2 auto _2 = b;
3 auto _3 = f(a)(b);
```

Figure 5.13: C++ Expression Example

5.6.3 Data Types

The second major decision was how to translate a data type, which also included translating dependent types. This was a major design decision because it affected not only data types translation but also how related areas like pattern matching would be translated. Hence the design needed to encode not only the structure but also allow intuitive methods of accessing the data held within the objects. Similarly to the functions, there were a couple of iterations of the data type translations, with each iteration refining the design.

First Iteration

The first design only considered simple data types, which were non-dependant types. Each data type has two important pieces of information, the constructor and the arguments to the constructor, which is the inner data. Thus each data type would be translated into a C++ class with two attributes, the constructor type and the argument data. For the constructor type attribute to store the relevant constructor, a new C++ enum would be created during the translation that contained the possible values for the data type constructors, and the enum would be used as the type for the constructor type attribute. For the argument data, the actual type of this is not known as it depends on which constructor the data has, and like the enum, there is a class created for each possible constructor that contains attributes for the data. For this reason, the actual type is void*, but the pointer will point to one of the possible constructor classes, and the data can be retrieved by first casting the pointer to the correct constructor class and then dereferencing the pointer. An example of translating the `Nat` type, shown in Figure 5.14, can be seen in Figure 5.15 that uses the described iteration.

```
1 data Nat : Type where
2     Zero
3     Succ of (Nat)
```

Figure 5.14: `pi-forall` Nat datatype definition

On Line One of Figure 5.15, the generated enum can be seen, with the two values being the two constructors `Zero` and `Succ`. Line Four shows the main class with the two attributes, Line Six and Seven, for the constructor type and the data pointer. Line Eight and Nine are static methods used to construct a `_Nat` object depending on which constructor should be used. If a `Zero` object is to be constructed then the `_Zero` function can be used. In the case of a `Succ` object, the function `_Succ` takes a parameter, which is the data argument for that constructor. The implementation of these functions is further down on Line 24 and Line 27 and is placed after the constructor class definitions for the code to compile. Line Ten shows the constructor for the `_Nat` object and is used by the previous two functions to construct the object. Line 15 shows the first constructor class, for `Zero` and as there are no arguments to this constructor from the `pi-forall` definition, there are no C++ attributes and only a constructor function for the `_Nat_Zero` object. For the `Succ` constructor, the class on Line 19 is created, and the only argument of type `Nat` can be shown on Line 21 as the attribute `_1` with type `_Nat` as well as the constructor function for the object shown on Line 22.

A few other decisions were also decided in the first iteration. Following the usual method, the datatype naming scheme enforces that all generated names start with an `"_"` to avoid naming conflicts. The enum all follow the structure of `"_enum_"` + Datatype name + `"_type"`. The class name for the data type is just `"_"` + Datatype name, and for the constructor class names, it is `"_"` + datatype name + `"_"` + constructor name.

Second Iteration

The second iteration refined the design by considering how the C++ objects would be used in a program. In `pi-forall`, objects are immutable, meaning that the data inside an object cannot change. The previous design has an issue because pointers are used, which means that if there are multiple references to an object, and it gets changed in one place, all references will be updated. Suppose there is a function to increment a `Nat`, then in `pi-forall` the original value would remain unchanged, and a newly incremented number would be created. It is possible, with the previous design, that when creating the newly incremented object, it will also update the passed parameter,

```
enum _enum_Nat_type { Zero, Succ };
class _Nat_Zero;
class _Nat_Succ;
class _Nat {
 public:
  enum _enum_Nat_type type;
  void* data;
  static _Nat _Zero();
  static _Nat _Succ(_Nat _1);
  _Nat(_enum_Nat_type t,void* d) {
    type = t;
    data = d;
  }
};
class _Nat_Zero {
 public:
  _Nat_Zero(){};
};
class _Nat_Succ {
 public:
  _Nat _1;
  _Nat_Succ(_Nat _1) { this->_1 = _1; };
};
inline _Nat _Nat::_Zero() {
  return _Nat(Zero, new _Nat_Zero());
};
inline _Nat _Nat::_Succ(_Nat _1) {
  return _Nat(Succ, new _Nat_Succ(_1));
};
```

Figure 5.15: C++ Nat Translation (based on the first iteration)

which means that it is no longer immutable. To preserve immutability, deep copying can be done. The passed parameter is a clone with completely different pointers but is the same value for the data, so when it is incremented, the original parameter object is not changed. This can be done in C++ by overriding the copy constructors. Currently, the copy semantics in C++ means that the parameters to functions are copied; however, it is a shallow copy, meaning that all pointers will be copied by value so that they will point to the same memory location, causing the mutability problem. If the copy constructors were overridden with new deep copy mechanics, then it would ensure it is impossible to change the data in the object, as it will be a completely separate object.

As pointers are used, the memory for the objects is allocated on the heap, and the memory should be reclaimed when the object is no longer needed. With the first and second iterations, there were memory leaks as no memory was freed due to double-free errors, or memory was freed too early, causing segmentation faults. The solution was to implement a simple garbage collector using reference counting. The C++ shared_ptr class is a smart pointer that allows for the shared ownership of an object through the pointer. Only when all the shared pointer objects are destroyed is the actual inner object destroyed, as the pointer will count the number of references to the object. Although this introduces overhead, it will completely solve the double-free segmentation fault and the dereferencing null pointer segmentation faults and stop the memory leaks that occurred when there was no object freeing.

These features' implementation is shown in Figure 5.16, and the main changes are the copy constructors being generated and using `std::shared_ptr<void>`.

```
enum _enum_Nat_type { Zero, Succ };
class _Nat_Zero;
class _Nat_Succ;
class _Nat {
 public:
  enum _enum_Nat_type type;
  std::shared_ptr<void> data;
  static _Nat _Zero();
  static _Nat _Succ(_Nat _1);
  _Nat(_enum_Nat_type t, std::shared_ptr<void> d) {
    type = t;
    data = d;
  }
  _Nat(const _Nat& other);
  _Nat() = default;
};
class _Nat_Zero {
 public:
  _Nat_Zero(){};
  _Nat_Zero(const _Nat_Zero* other) {}
};
class _Nat_Succ {
 public:
  _Nat _1;
  _Nat_Succ(_Nat _1) { this->_1 = _1; };
  _Nat_Succ(const _Nat_Succ* other) { this->_1 = other->_1; }
};
inline _Nat _Nat::_Zero() {
  return _Nat(Zero, std::static_pointer_cast<void>(std::make_shared<
  _Nat_Zero>()));
};
inline _Nat _Nat::_Succ(_Nat _1) {
  return _Nat(Succ, std::static_pointer_cast<void>(std::make_shared<
  _Nat_Succ>(_1)));
};
_Nat::_Nat(const _Nat& other) {
  type = other.type;
  data = other.data;
}
std::function<uint64_t(_Nat)> intFromNat = [](_Nat n) {
    if (n.type == Zero) {
        return 0;
    } else {
        return (uint64_t)1 + intFromNat((*(std::static_pointer_cast<
  _Nat_Succ>(n.data)))._1);
    }
};
```

Figure 5.16: C++ Nat Translation (based on the second iteration)

Third Iteration

The most apparent issue of previous iterations is how more complex data types are handled, such as indexed data types. It is possible to erase the dependent types during the translation, as once type checking has passed, they are no longer needed. In `pi-forall`, if the value were needed, then it would need to have been explicitly given elsewhere; hence the actual indexes into the type can be erased. One exception to this rule is when the value the type is indexed by is also a type. An example of this case is the `Maybe` type, defined in Figure 5.17, which is a type that can represent the absence of a value, with the `Nothing` constructor, or the existence of a value, with the `Just` constructor. On Line Three, the constructor has an argument with the type `A`, so when translating this constructor to a class, the type of the argument would become an issue, as it now depends on `A`.

```
1 data Maybe (A : Type) : Type where
2     Nothing
3     Just of (A)
```

Figure 5.17: `pi-forall` Maybe datatype definition

The solution is to use C++ templates, which allows a class definition to become generic over a type, `A` and then refer to `A` as a type in the class definition. This is shown in Figure 5.18, which shows a small section of the Maybe translation highlighting the use of templates. For example, on Line 2, it shows that the `_Maybe` class is templated over a type `A`, and on Line 20, in the `_Maybe_Just` class, the argument `_1`, which representing the first argument of the `Just` constructor, has type `A`.

```
1  ...
2  template <class A>
3  class _Maybe {
4   public:
5    enum _enum_Maybe_type type;
6    std::shared_ptr<void> data;
7    static _Maybe<A> _Nothing();
8    static _Maybe<A> _Just(A _1);
9    _Maybe<A>(_enum_Maybe_type t, std::shared_ptr<void> d) {
10     type = t;
11     data = d;
12   }
13   _Maybe<A>(const _Maybe<A>& other);
14   _Maybe<A>() = default;
15 };
16 ...
17 template <class A>
18 class _Maybe_Just {
19  public:
20   A _1;
21   _Maybe_Just(A _1) { this->_1 = _1; };
22   _Maybe_Just(const _Maybe_Just* other) { this->_1 = other->_1; }
23 };
24 ...
```

Figure 5.18: C++ Maybe Translation Excerpt

A seemingly more complex example would also be the `Vec` type, which is indexed by not only the

type, `A`, but also the natural number, `n`. As stated previously, it would be possible to erase the indexed natural number and then template the class over the type `A`. Although this seems like it would be the solution, the `ConsV` constructor adds a new complication, as highlighted in Figure 5.19, which is the constraint `[n = Succ m]`. Again this is very easy to translate as any constraint can be erased, as they are only used during the type checking to constrain the values of the constructor and can never be used at execution, as the constraint is made irrelevant using the square brackets. This supports the reasoning as to why the values can be erased in the type because they will either appear in a constraint, which can always be erased, or in another type for an argument in a constructor, which means they are erased.

```
1 | ConsV of [m : Nat] (A) (Vec A m) [n = Succ m]
```

Figure 5.19: `pi-forall` ConsV constructor for the Vec datatype definition

5.6.4 Pattern Matching

Figure 5.20 shows an example of what the generated code for a pattern match would look like, for a `pi-forall` case statement. Using the type attribute of the main datatype class makes it possible to use a switch statement so that the correct cast can be done on that data attribute. Inside each case, it can then create the variables that should be bound, like `n`. The reason for keeping the name, is that `n` can appear in the expression `b`. Finally, to explain Lines 1 and 16, the switch statement is in a lambda expression that immediately gets evaluated; this is just so that no matter which case statement is entered, returning a value will set it to the variable `ret` on Line 1.

```
1 | case x of
2 |     Zero -> a
3 |     Succ n -> b
```

```
1  auto ret = []() {
2      auto _1 = x;
3      switch(_1.type) {
4          case Zero: {
5              auto _2 = *(std::static_pointer_cast<_Nat_Zero>(_1.data);
6              // code for a
7              return ...;
8          }
9          case Succ: {
10             auto _2 = *(std::static_pointer_cast<_Nat_Succ>(_1.data);
11             auto n = _2._1;
12             // return code for b
13             return ...;
14         }
15     }
16 }();
```

Figure 5.20: Pattern Match Example

5.6.5 Equality

As equality is defined in the Tree using TyEq rather than being defined as a data type, it meant that there needed to be a custom C++ class written for it, shown in Figure 5.21. It translates it

into the class `_TyEq` and there is only a single constructor, `_Refl()`, which will construct the object. This will allow the TyEq and Refl Terms to be translated.

```
1 class _TyEq {
2  public:
3   static _TyEq _Refl();
4 };
5
6 inline _TyEq _TyEq::_Refl() { return _TyEq{}; }
```

Figure 5.21: C++ TyEq Translation

This also makes it easy to translate the Subst and Contra. For example, with the above translation for TyEq, nothing will change in a substitution statement that is required for run time, so to translate the expression `subst a by b`, only the code for `a` will need to be generated. The contra, is slightly more complex, as it is used in fuctions where the return type is `Void` which is uninhabited. To make things easier, the type `_Void` has been given a single constructor so that when a contra expression is found, the constructor for the `_Void` type is returned. This does defeat the point of `Void` being uninhabited, but this was the only way to make a translated contra expression compile.

5.6.6 Unit

The `Unit` function needed to have a single change because it's constructor is `()`, so during the translation, this constructor name is mapped to a new name `unit`.

5.6.7 Nat

When using the translated `Nat` datatype, the `_Nat` class, a large number can quickly take up a substantial memory footprint, as it needs to have references to all previous numbers. There is also efficiency issues, as doing a simple operation like addition, subtraction, and multiplication are very slow when using the Peano[26] arithmetic representation. For this reason, a simple change was made not to translate `Nat` as a data type but instead translate it to the `uint64_t` type, which represents an unsigned 64 bit number. This will also mean that simple functions like plus and multiply need to be replaced with the faster C++ primitive operators, $+$ and $\times$.

5.6.8 Types

Another critical decision was the mapping `pi-forall` types to C++ types. This was done by creating a function, `generateCType`, which will take a Type and return a String which is the equivalent C++ type. Only a limited number of the Types from the AST can be translated directly using this function, which includes: Pi, TCon, App, Var, Lam, TyEq.

For Pi, it was previously discussed that functions would be translated using `std::function`, which means that any Pi Type will be translated to the `std::function` type, where the template is filled in with the return type and argument type.

For the TCon, this will represent a data type, which means that it will translate it to the C++ class for the type. It first checks if the TCon represents the `Nat` type as this will get translated to the C++ type `uint64_t`. Otherwise, during this generation, it will get the name of the type and prepend the `_` and then it will look up if there are any template arguments from the state. If there are template arguments, then it will also generate the relevant types to add as a template. For

example, the `Bool` type has the tree `TCon Bool []` which will be translated to the C++ class `_Bool`. A more complex example would be using the `Maybe` Type, which is represented by `TCon Maybe [Arg {argEp = Rel, unArg = Bool}]`, which would get translated to the C++ type `_Maybe<_Bool>` by extracting the `Bool` from the argument and then placing it in the `_Maybe` template.

For the App, the type is given by the return type of the function term, The var will look up the type of some function elsewhere, the type of the lambda will be the type of the bound term and the type of TyEq is always `_TyEq`.

5.7 Translation State

Throughout the translation, state is required to be passed around. An example would be when a fresh variable name is needed; this can be achieved by getting a unique number and using that as part of the name. Instead of passing the state around through every function, the State monad is used. The state passed around is the tuple TranslatorState, defined in Figure 5.22. The [InterDef] stores the current list of translated intermediate representations. The Int is a counter used to get a unique number to ensure that variable names are fresh and unique. The first [String] is a stack of variable names to be passed between function calls. For example, when translating a sub-expression, it might need to assign the result to a variable, and that variable is passed through this stack. The second [String] is another stack that contains variable names, but this time it is used to store the variable capture stack needed for C++ lambda closures. The [Term] is a stack that contains the type, so with each part of the tree, there is an entry in the type stack that gives the type of the Term. Finally, the [Decl] contains a list of all the Decls being translated so that if any information from them is needed, it can be looked up.

```
type TranslatorState = (
    [InterDef], -- Translated IRs
    Int,        -- Fresh variable counter
    [String],   -- Variable name stack
    [String],   -- Capture Stack
    [Term],     -- Type Stack
    [Decl]      -- Decl's Translated
)
```

Figure 5.22: Haskell Translator State Definition

5.8 Translation Functions

Throughout this section, small excerpts of Haskell code are given, but for brevity, unimportant information is missed, indicated by .... The code is meant to provide an illustration for the general flow of the translation within each function.

5.8.1 Entry Function

The entry point for the translation is the translate function, shown in Figure 5.23, where the Module is the AST that will be translated, and the list of intermediate representations is a list of initial pre-defined functions and data types. These intermediate representations are passed in as types like `Nat` and `Sigma` are defined elsewhere in their own tree and need to be passed in to be translated. The function will return the list of intermediate representations for the `pi-forall` program and

also contain the initial set of intermediate representations. This function will also set up the initial translation state that's used during the translation.

```
1 translate :: Module -> [InterDef] -> [InterDef]
2
```

Figure 5.23: Translation Entry Function Type Signature

From this entry function, the `translateModuleEntry` function will be called and a skeleton of the function is shown in Figure 5.24, with parts such as function arguments or entire lines missed out for brevity. The function translates a single `Decl` into an `InterDef`, as can be seen from the type signature on Line One. Lines Two and Three show the function `generateFunctionDef` being called when the `Decl` matches to the `TypeSig`. When the `Decl` matches to either the `Def` or `RecDef` constructor, the function `initialGenerateFunctoinImpl` will be called, seen on Lines Five and Seven. The `Data` constructor will be matched on Line 10, resulting in the `generateData` function being called. If a `Demote` or `DataSig` constructor is matched then the program will exit and crash, but this will not happen as they cannot be matched from the AST of a program.

```
1  translateModuleEntry :: Decl -> State TranslatorState InterDef
2  translateModuleEntry (TypeSig sig) =
3    generateFunctionDef ... -- Translating Type Signature
4  translateModuleEntry (Def name (term, typeStack)) = do
5    initialGenerateFunctionImpl ... -- Translating Function Implementation
6  translateModuleEntry (RecDef name (term, ts)) = do
7    initialGenerateFunctionImpl ... -- Translating Function Implementation
8  translateModuleEntry (Demote epsilon) =
9    undefined -- Never need to translate
10 translateModuleEntry d@(Data tcName telescope constructorDefs) =
11   generateData ... -- Translating Data Type
12 translateModuleEntry (DataSig tcName telescope) =
13   undefined -- Never need to translate
```

Figure 5.24: translateModuleEntry Function

5.8.2 TypeSig Translation

The generateFunctionDef function translates a `pi-forall` type signature into C++ function declaration, and its type signature is shown in Figure 5.25. The function will walk the `TypeSig` given and recursively construct the `FunctionDef` object, where initially an empty `FunctionDef InterDef` is passed.

Within a `TypeSig` Tree, only a limited subset of the Terms can appear: Pos, Var, Pi, TCon and TyEq. All the rest will not appear, meaning only those Terms need to be translated.

Pos

If the Type given is of the form (Pos position term), then there is a recursive call on the inner term, as the position is ignored during the translation, meaning that this node is effectively ignored.

```
1 generateFunctionDef :: InterDef -> Type -> State TranslatorState InterDef
```

Figure 5.25: Haskell generateFunctionDef Type Signature

Pi

The Pi Term represents the function type, `A -> B`, where `A` and `B` are Terms themselves, and also stores information on whether `A` is relevant or irrelevant. If `A` is irrelevant, then `A` can be erased and only `B` needs to be translated. The only exception is when `A` is a Type; the function will have to be templated over the type `A`. This will involve updating the InterDef with the template information so it can be templated over `A` and then returning the generated InterDef for B.

When `A` is relevant, matched in the code shown in Figure 5.27, then the C++ type of `A` will need to be obtained by calling the `generateCType` function shown on Line 2. Line 3 shows the recursive call to generate C++ for the Term `B` which will generate the `InterDef` for `B`. Finally, to produce the `InterDef` for the overall Pi Term, the C++ lines need to be concatenated to produce a C++ `std::function` where the return type of the function can be obtained from the variable `b`, and the parameter type is stored in `tyACType`.

```
1 generateFunctionDef (FunctionDef ...) (Pi Rel tyA bnd) = do
2   tyACType <- generateCType tyA -- Generating the C++ Type for A
3   b <- generateFunctionDef ... -- Generating C++ Type for B
4   return $ FunctionDef ...
5     ("std::function<" ++ (definition b) ++ "(" ++ tyACType ++ ")>")
```

Figure 5.26: Haskell Relevant Pi Translation Code

Var, TCon and TyEq

If the Term didn't match either the Pos or the Pi Term then it must match the Var, TCon or TyEq. This will also mean that it is the last Term in the type signature as only a Pi Type can contain is a function type. All three use the same code implementation to translate, Figure REF shows the translation code. On Line Two, the Term, `t`, has its C++ Type generated and on Line 2 the InterDef is constructed. When constructing the InterDef, it not only sets the C++ header definition to the generated C++ Type but also just before it also sets the function's return type to be the C++ type.

```
1 generateFunctionDef (FunctionDef name tempArgs args _ definition) t = do
2   cTy <- generateCType t
3   return $ FunctionDef name tempArgs args cTy cTy
```

Figure 5.27: Haskell Relevant Pi Translation Code

5.8.3 Def / RecDef Translation

The entry function for the Def and RecDef is `initialGenerateFunctionImpl :: InterDef -> Term -> State TranslatorState InterDef`, which will check only for `Lam` terms so that it can check if the def is a function, or if it is a variable. If it is a function, then it will continue traversing through the `Lam` terms until there is no more, so that all the arguments are retrieved, from which it can start to build the lambda expressions. Once it reaches a term that is not a `Lam`, it will call the `generateFunctionImpl` function that has the same type signature, but is designed to translate the body of the function, and not the arguments from the lambda. This is also important for setting up the capture stack, as any inner lambdas will need to have the capture stack set up properly so that the lambda expressions have all the required variables captured.

Inside the `generateFunctionImpl` it has a pattern match for every single type of `Term`. For brevity, this will not be shown here, only the most key decisions. The function is fairly straightforward where at each term it will recursively call to build the code for the sub-terms and then just slot it into the code template for each type of term. For example, a pattern match would just set up the switch statement and then recursively translate each case expression and just place the generated code for each expression in the correct case.

Only the key design decisions will be explained here, as most of what will be translated has been discussed earlier in Section 5.6

varStack

When translating any term in `generateFunctionImpl`, the translator needs to know if the resulting C++ lines are going to be just returned up, or if it will be store in a variable. For example, if a variable, x, is to be translated, it can either produce `x` or `auto _1 = x`. This will depend on the varStack. When a recursive call is made to `generateFunctionImpl` on a subterm, if it is needed to be stored in a variable, then a variable is pushed onto the stack before the recursive call, which will then be popped off during the recursive call so the C++ string can store the result in the variable. Otherwise, if there is nothing in the stack, it will just return the variable name. This is used in every term to decide how sub-terms should be stored when translated.

App

The other interesting translation is the translation of an application, which has its own function. If a function has more than one argument, then it will create multiple `App` terms because only a single arg can be applied at a time, and the outermost `App` contains the argument applied last. So there is a function that will traverse through all the `App` terms to obtain the inner function and the list of args and reverse the order of the list. Then the`generateFunctionImplFromApp` will generate the code for each argument and store each result in a variable, which will be applied to the function in the correct order.

Irrelevance

The final interesting problem solved in the translation is with irrelevance and the type stack. If something is irrelevant, it will be skipped over and not translated, but this posed a problem with the typeStack, as the type for all the terms that appear irrelevant are still there, so they need to be removed. The led to the creation of a function `numberToDrop` which would fold over the irrelevant term and return how many elements from the type stack needed to be dropped for the term.

5.8.4 Data Translation

```
1 generateData :: Decl -> State TranslatorState InterDef
2 generateData (Data tcName telescope constructorDefs) =
3     ...
```

Figure 5.28: generateData Code Excerpt

The `generateData` function, shown in Figure 5.28, will translate a Data Decl into the Datatype InterDef. From the definition of the Datatype InterDef (Figure 5.11), three pieces of data are needed: the type name, the constructor names, and the C++ definitions. The type name can be

```
1 let conNames = map (getConName) constructorDefs
2   where getConName (ConstructorDef _ conName _) = conName
```

Figure 5.29: Constructor Names Code

obtained easily as it is just the variable `tcName`. It is similarly easy to obtain the constructor names, with the code excerpt shown in Figure 5.29.

For the C++ definition, many elements need to be translated, but to translate them, only two more pieces of information is needed: the template information and the constructor attribute types. The code for getting the template information is shown in Figure 5.30 and it produces a list of the template names required for a class. If no templates are required for a datatype, an example being `Nat`, then the list is empty. Otherwise, like in the `Maybe` the example shown previously on Page 47, the template list would include `A`. By inspecting the telescope of the datatype, it is possible to generate the names of the templates. When there is a relevant argument in the telescope, there may be a type that needs to be included in the template list. The `case` statement on Line Four will inspect if the type signature is relevant or irrelevant. On Line Five, nothing is added to the accumulator list when it is irrelevant. On Line Six, the `relInner` function is called, which will either add a new type to the template or skip over. The `relInner` function will append the type's name to the list only when there is a type, shown on Line 11, and Line 9 and 10 show it recursing through the Pos and Ann Terms. The other interesting case is when the Type matches to the Pi Type. In this case, if the final return type of the Pi type is `Type` then it should be included as a template; otherwise, it can be ignored. This is done by recursing on the bound term in the Pi Type, which if it results in `Type` will eventually match with the function on Line 11. Otherwise, should any other Type be found, it will match with the "catch-all" case on Line 15, which will ignore the Type as it shouldn't be added to the Template.

```
1  let templates = genTemplateTypes telescope
2    where
3    genTemplateTypes (Telescope decls) =
4      foldr (\(TypeSig (Sig name e t)) acc -> case e of
5          Irr -> acc
6          Rel -> relInner (Unbound.name2String name) acc t
7      ) [] decls
8    relInner :: String -> [String] -> Type -> [String]
9    relInner name acc (Pos _ t) = relInner name acc t
10   relInner name acc (Ann t _) = relInner name acc t
11   relInner name acc (Type) = name : acc
12   relInner name acc t@(Pi _ _ bnd) =
13     let (bndName, bndT) = unsafeUnbind bnd in
14     relInner name acc bndT
15   relInner name acc _ = acc
```

Figure 5.30: C++ Template Information Generation Code Excerpt

The other piece of information is the constructor class attribute types. For example, the `Just` constructor has an element of type `A` as its argument. This will need to be extracted from the `Telescope` of each `ConstructorDef`. The function `conClassAttributes` will generate the list of attributes required for each constructor class by traversing the telescope. The code for this function is to large to be shown in a figure, so is skipped.

The code string generation can begin after all the required information has been obtained. First, the code string for the Enum is generated using the type and constructor names resulting in: `enum _enum_Maybe_type { Nothing, Just };`. Next, the forward declarations and main class code can be created using the names and template information and the constructor attributes for the static functions to create the class object based on the constructor. The generated code for this is shown in Figure 5.31.

```
1  template <class A>
2  class _Maybe_Nothing;
3  template <class A>
4  class _Maybe_Just;
5  template <class A>
6  class _Maybe {
7   public:
8    enum _enum_Maybe_type type;
9    std::shared_ptr<void> data;
10   static _Maybe<A> _Nothing();
11   static _Maybe<A> _Just(A _1);
12   _Maybe<A>(_enum_Maybe_type t, std::shared_ptr<void> d) {
13     type = t;
14     data = d;
15   }
16   _Maybe<A>(const _Maybe<A>& other);
17   _Maybe<A>() = default;
18 };
```

Figure 5.31: Main Class Generation Example for `Maybe`

Next the constructor classes can be generated using the same information, with an example for the `Just` constructor's generated class `_Maybe_Just` being given in Figure 5.32

```
1  template <class A>
2  class _Maybe_Just {
3   public:
4    A _1;
5    _Maybe_Just(A _1) { this->_1 = _1; };
6    _Maybe_Just(const _Maybe_Just* other) { this->_1 = other->_1; }
7  };
```

Figure 5.32: `_Maybe_Just` constructor class

Finally, the functions that were declared previously, the static functions to construct the main class based on one of the type's constructors, can be generated. Figure 5.33 shows an example of the function. Again the names, template information and constructor attributes are required for this part.

```
1 template <class A>
2 inline _Maybe<A> _Maybe<A>::_Just(A _1) {
3   return _Maybe<A>(Just, std::make_shared<_Maybe_Just<A>>(_1));
4 };
```

Figure 5.33: `_Maybe_Just` constructor class

5.9 Code Generation

As the generated code is stored within an InterDef, it needs to be extracted, which is done during the final code generation stage, where the C++ file is generated. It will go through each InterDef and write the function or data type to the file. As no modules are imported, the type signature header definitions are not needed in the file, so they are discarded. If, at some point, modules were used, then the header definitions would already be translated.

An example of translating the function `append` is shown in Figure 5.34.

```
append : [A :Type] ->[m:Nat] -> [n:Nat] -> Vec A m
     -> Vec A n -> Vec A (plus m n)
append = \[A] [m] [n] v1 ys . case v1 of
     ConsV [m0] x xs -> ConsV [plus m0 n] x (append [A] [m0][n] xs ys)
     NilV -> ys
```

```
template <class A>
std::function<std::function<_Vec<A>(_Vec<A>)>(_Vec<A>)> append = [](auto
   v1) {
  auto _171 = [v1](auto ys) {
    auto _172 = [v1, ys]() {
      auto _173 = v1;
      switch (_173.type) {
        case ConsV: {
          auto _175 = *(std::static_pointer_cast<_Vec_ConsV<A>>(_173.data
   ));
          auto x = _175._1;
          auto xs = _175._2;
          auto _176 = x;
          auto _179 = xs;
          auto _180 = ys;
          auto _177 = append<A>(_179)(_180);
          auto _174 = _Vec<A>::_ConsV(_176, _177);
          return _174;
        }
        case NilV: {
          auto _185 = *(std::static_pointer_cast<_Vec_NilV<A>>(_173.data)
   );
          auto _184 = ys;
          return _184;
        }
      }
    }();
    return _172;
  };
  return _171;
};
```

Figure 5.34: append function translation

5.10 Extending pi-forall

When writing the translator, there was also a couple of extensions to `pi-forall` that were created to make the next sections, creating the primitives and pattern easier.

The first was adding a new primitive type that would represent a C++ char, named `Char`. This involved extending the parser, AST, also the type checker, with the equality checker that is within it. After this, syntactic sugar was added for strings and lists, again by extending the parser. For example `<1, 2, 3>` would get converted into the term `Cons 1 (Cons 2 (Cons 3 Nil))`, or the syntactic sugar for strings, "aa" would be converted into `Cons 'a' (Cons 'a' Nil)`.

During the translation, there was also a method through which `pi-forall` functions could be replaced with a manually written function. This was extremely useful in creating functions like `printChar` which would print a character to the terminal and was a manually written C++ function which would print the character. This foreign function interface, which would allow foreign functions to be injected into the code, was used extensively in the final concurrency primitives design in Chapter 6.

5.11 Limitations and Issues

There are numerous limitations to the translation system that has been designed and implemented. Some were intentional, while some were planned to be implemented but were impossible due to time constraints or technical debt.

The first is that it cannot fully translate all `pi-forall` programs. This is mainly intentional, as it would have been too large a task to implement a translation for a dependently typed language fully. The first limitation is that a type can never appear in a relevant way. For example, the code shown in Figure 5.35 would be impossible to translate. This is because no direct translation is available for a Type in C++. To keep things simple by keeping the type irrelevant, it is possible to use templates to translate everything. It also allows many essential uses of dependant types to be still possible.

```
1 id : (A : Type) -> A -> A
2 id = A a . a
```

Figure 5.35: `pi-forall` function where a Type is relevant

The module system was advantageous in `pi-forall` to help split a program up into separate files, but as it is a non-essential feature, it was removed when translating, mainly due to time constraints. Due to this limitation, a `pi-forall` program cannot import functions or types from other files, which makes a `pi-forall` file very long, and also the standard library would need to be imported for every new program.

An unintentional limitation was to force a user not to reuse a variable name within a function, even if a new scope is created. This limitation was created by technical debt, as the generated C++ code would not compile due to missing variables from capture stacks. The easiest solution to this problem was to prevent variable names from being reused within a function's scope.

There is also a limitation with the efficiency of the generated code. An excerpt of generated code is shown in Figure 5.36, which calls the map function and stores the result. Instead of directly calling the function with the two arguments, the arguments are placed in an intermediary variable, Lines

One and Two, which are used instead when the function is called. This is just a simple example, but it can scale when there is a large number of arguments and function calls. As this was a consequence of a design decision, there was a solution to try and optimise this, which was to compile the C++ code with the O3 optimisation flag turned on, which would optimise out the redundant assignment operations and inline the arguments. This however does not work in every case.

```
1 ...
2 auto _840 = natToString;
3 auto _844 = x;
4 auto _839 = map<uint64_t, _List<char>>(_840)(_841);
5 ...
```

Figure 5.36: Generated Code Excerpt

Another limitation on the efficiency of the translated code is a large amount of copying. As copying is used everywhere in the translated code, with large objects, it can cause overhead as every copy is a deep copy that creates an entirely fresh object.

The code generation is quite quick, but only after the program has been parsed. The `pi-forall` parser is extremely slow when a program has lots of functions or a few large functions. So the translation is limited by the parser of the program.

There is also an issue that the implementation is riddled with technical debt. Some are a consequence of the original source code, and the rest is due to the translator's design. For example, in hindsight, there are better methods than translating into an intermediary representation, but as that is what the design started out like, by the time a better method was thought of, it was too late.

5.12 Future Work

There is a lot of work that can be done in the future with this translator. For example, an entire rewrite could take place that would remove the technical debt and streamline the design. There is also ideas on how to translate the type of types, `Type`, which could be done with a new class to represent them.

The module system could be implemented as a new feature so that programs could import other programs, as well as new syntactic sugars implemented on top of the parser, such as `do` notation from Haskell.

Another idea that builds off this work, is to use the Idris Backend Generator, which allows custom backends to be written for Idris. If it is designed the same way as it is currently, so the same style of code is produced, then it would be possible to use any of the concurrency primitives or parallel patterns in Idris, which would be a much nicer experience for them to be used.

5.13 Summary

In this chapter, the overall design behind the translator is detailed, with the key design decisions being discussed and explained. The `pi-forall` abstract syntax tree is defined and explained, as well as the main functions used to translate the AST into C++ code. Finally, the chapter finishes by discussing the limitations of the translator as well as the areas that can be worked on in the future.

Chapter 6

Concurrency Primitives

In this chapter, there is a discussion on how the concurrency primitives were implemented in `pi -forall`. Section 6.1 outlines the initial design used for the primitives and Section 6.2 shows the iterated design. Section 6.3 describes what the final design is and finally Section 6.4 outlines areas that can be worked on in the future.

Please note that for brevity, some of the `pi-forall` figures will omit irrelevant arguments as they can get to hundreds of characters long and do no aid in understanding.

6.1 Initial Design

To create the high-level parallel patterns, there would first need to be concurrency primitives created that could be utilised to build the more abstract patterns. The model would be built mainly in `pi-forall` with some C++ needed for the low-level thread interactions. Another goal of the primitives is that parallel programs could be written without the use of the parallel patterns created in the next chapter, and using only `pi-forall` code, could implement new parallel patterns or other models of parallelism.

In the initial design, the first choice was to implement an interface in `pi-forall` to a C++ implementation of message-passing concurrency. This meant that types would be created to represent a C++ concurrent queue or thread, and functions that would interact with these objects at a high level to take advantage of the dependant types.

To take advantage of the dependent types, the first focus was on correctness. For example, when sending a message along a channel, is there a way to type-check a message passed along a channel, or is there a method of type-checking that the correct channel is being utilised? The answer to both of these is yes.

6.1.1 PID

The first structure created was the `PID` type, and it represented a process identifier and is shown in Figure 6.1. The `PID` type is indexed by a number that is its process identifier, and as seen on Line Two, the `MkPID` constructor can only be used when the `id` argument is equal to the indexed `pid` of the type, which means that there is only a single inhabitant of the `PID n` type, where `n` is any number. As there is only one inhabitant per number, if two ids are the same, then the processes are the same, and they will have the same type.

```
1 data PID (pid : Nat) : Type where
2     MkPID of [id : Nat] [id = pid]
```

Figure 6.1: `pi-forall` `PID` datatype definition

6.1.2 Channel

The `Channel` type represented the queue that would be utilised to send messages between processes, and its definition is shown in Figure 6.2. Similarly to the `PID` type, the `Channel` type is indexed by an id, in this case, the channel id, which from the constraint in the constructor ensures that there is only one element of a channel with the same id, so if two channels had the same id, then they would have to be identical. It is also indexed by a type `A` which represents the type of messages that can be sent over the channel. This would mean that in functions, it would be possible to type-check the elements being sent or received.

```
1 data Channel (A : Type) (chid : Nat) : Type where
2     MkChannel of (id : Nat) [id = chid]
```

Figure 6.2: `pi-forall` `Channel` datatype definition

6.1.3 IO Monad

When using the concurrency primitives, it is crucial to chain operations; for example, a typical use case would be running a function and sending the result over the channel. In normal `pi-forall` there is no mechanism to chain expressions; hence a method for this chaining had to be implemented. The easiest method for this was implementing the `IO` monad, which would also have the advantage of being the basis for user interaction, such as printing to the screen. The code for the `IO` monad is given in Figure 6.3 and is a strict implementation of the Haskell IO Monad.

```
1  data IO (A : Type) : Type where
2      MkIO of (A)
3
4  returnIO : [A : Type] -> A -> IO A
5  returnIO = \[A] a . MkIO a
6
7  bindEq : [A : Type] -> [B : Type] -> IO A -> (A -> IO B) -> IO B
8  bindEq = \ [A] [B] a f . case a of
9                               MkIO inner -> f inner
10
11 bind : [A : Type] -> [B : Type] -> IO A -> IO B -> IO B
12 bind = \ [A] [B] a b . bindEq [A] [B] a (\ c . b : A -> IO B)
```

Figure 6.3: `pi-forall` IO Monad definition

Lines One and Two show the data type definition, with the main functionality of the monad being implemented by the three functions: `returnIO`, `bindEq`, and `bind`. The `returnIO` takes an element and lifts it into the monad and is how the type is introduced. There are then two methods of chaining `IO` actions, where `bindEq` will take an `IO` action and a function that will consume the result of the given `IO` action to produce a new `IO` action. The other method is `bind` which chains the two given `IO` actions together and is implemented in terms of the `bindEq` but the function will be the constant function which returns the second action. The main difference in the use cases is with

`bindEq` the operations that are being chained can pass values between the actions, while `bind` chains two separate actions.

6.1.4 Maybe Type

Another type that was utilised in the design is the `Maybe` type, shown in Figure 6.4, so that absent values can be represented. For example, the creation of the channel or process may be unsuccessful for various reasons, and that shouldn't break the `pi-forall` code, so using the `Maybe` type will allow users to have error handling at the `pi-forall` program level.

```
1 data Maybe (A : Type) : Type where
2   Nothing
3   Just of (A)
```

Figure 6.4: `pi-forall` Maybe Monad definition

6.1.5 Primitive Functions

The initial set of functions were given a type signature but no implementation; instead, the implementation would be done in C++, and the function would serve as an interface to C++ code. By using the `TRUSTME` expression in `pi-forall`, a function would be trusted that its type checked, and no code would have to be given. If the function were used, the arguments would be able to be type-checked, which is how the correct use of the primitives was enforced, even though the implementation is not written in `pi-forall` but C++ instead.

Spawn

The `spawn` function, shown in Figure 6.5, would create a new process, hence the `PID` being returned. As the `PID` type is indexed by the number `id`, a dependant pair is returned so that any `PID id` can be returned. The returned pair is within the `IO` Monad as the `spawn` function is an `IO` action. It is assumed that a user will not create a `PID` object manually and instead only use the `spawn` function to create it.

```
1 spawn : IO {id : Nat | PID id}
2 spawn = TRUSTME
```

Figure 6.5: `pi-forall spawn` function type signature

Run

The `run` function, shown in Figure 6.6, would run an `IO` action, `f`, on a process. The action of running a function on a process does not return anything, hence the function returns `IO Unit`.

The reason for a separate function for creating and using a process was a plan for future-proofing. The plan was to eventually expand the `Channel` type with references to the `PID` that it connected. This meant that if the function running on a process referred to a channel, the channel could only be constructed once the `PID` elements had been created, so the `Channel` had to be constructed after the spawning of the process, but the function couldn't be created until after the `Channel` had been created.

```
1 run : {id : Nat | PID id} -> (f : IO Unit) -> IO Unit
```

Figure 6.6: `pi-forall run` function type signature

Link

The `link` function, shown in Figure 6.7, is the method of creating a channel by linking some processes. It takes as argument the type of messages to be passed along the channel, `A` and much like `spawn`, it returns a dependant pair of the channel id, `id`, and the channel itself within the `IO` monad. It is assumed that a `Channel` object will only be created through the link function

```
1 link : [A : Type] -> IO {id : Nat | Channel A id}
```

Figure 6.7: `pi-forall link` function type signature

Send

The `send` function, shown in Figure 6.8, would send a message, `x`, along a channel given to the function. There is also a simple correctness check for the type of the message, as both the message type and the type of messages in the channel are `A`. This means that if the type of the channel was some other type, `B`, the send function would not type-check if `x` was sent along that channel. This allows for some correctness of messages passed along the channel.

```
1 send : [A : Type] -> {id : Nat | Channel A id} -> (x : A) -> IO Unit
```

Figure 6.8: `pi-forall send` function type signature

Receive

The `recieve` function, shown in Figure 6.9, would receive a message of type `A` from a channel. Note that much like the `send` function, this function can also ensure the correctness of the message type during compile time. For example, if after receiving a message `x` of type `A`, then it cannot be used as an argument to a function expecting an element of type `B`.

End

The `end` function, shown in Figure 6.10, would be a function that would return an `IO Unit` and would do nothing, but would serve as a method of terminating the chain of `IO` actions that might occur on a process, hence terminating the function being run on a process. It is worth noting that this is not a synchronisation technique for processes.

6.1.6 C++ Parallel Runtime System

After designing the `pi-forall` primitives, the next step was to relate them to C++ to create a parallel runtime system so that by using the primitives, parallel programs could be written.

The `PID` and `Channel` types were straightforward to correlate to C++ as the `PID` relates to a C++ `std::thread` object and the `Channel` would relate to concurrent queue object, which was implemented as the class `LockingCQueue` shown in Figure **??**. As both types are indexed by and number representing an `id` the actual related C++ objects could be held in a map in the runtime system where the `id` would map to the C++ object. Figure 6.11 shows the map structures and functions that manipulate

```
recieve : [A : Type] -> {id : Nat | Channel A id} -> IO A
```

Figure 6.9: `pi-forall recieve` function type signature

```
end : IO Unit
```

Figure 6.10: `pi-forall end` function type signature

them. Lines One and Two show the map for the threads as well as a mutex for the structure so that when functions access or update the map, the structure can be made thread-safe. Similarly Lines Six and Seven show the map for the channels and a mutex for the structure. As `LockingCQueue` is templated over a type `A`, the class is not fully formed without the template information so this means that with different template arguments the types are completely separate. e.g. `LockingCQueue<int>` and `LockingCQueue<bool>` are different types. This has an implication that no single type store all possible `LockingCQueue<A>`, hence a `void*` type is used to allow for any channel to be stored in the map. There are two functions for each map, one to get the object from the map and one to add a new object to the map, which are used in the C++ implementations of the primitive functions shown earlier.

```
1  // map of threads
2  std::map<uint64_t, std::thread*> pidsMap;
3  std::mutex pidMutex;
4
5  // map of channels
6  std::map<uint64_t, void*> channelsMap;
7  std::mutex channelMutex;
8
9  // Functions
10 std::thread* getThread(uint64_t pid);
11
12 void addThread(uint64_t pid, std::thread* t);
13
14 void* getChannel(uint64_t chid);
15
16 template <typename A>
17 void addChannel(uint64_t chid, LockingCQueue<A>* queue);
```

Figure 6.11: C++ Maps for threads and channels

LockingCQueue

The `LockingCQueue` was taken from a previous practical, CS4204 P1, and no changes were made to it. It contains two mutexes so that an enqueue and dequeue can be done at the same time, and also a semaphore so that there is blocking behaviour.

Flaws

At this point, after the functions and underlying C++ structure were designed, there were some large flaws. For example, what happens if a user accidentally creates a new process with the same id or channel with the same id, then in the map they would be overwritten and lost. There is no safety assured in `pi-forall`. This meant that in the C++ implementation of the primitives, there

```
template <typename T>
class LockingCQueue{
    private:
        QNode<T> *front;
        QNode<T> *rear;
        std::mutex enqMutex;
        std::mutex deqMutex;
        Semaphore *sem;
    public:
        LockingCQueue();
        ~LockingCQueue();
        void enqueue(T payload);
        T dequeue();
};
```

Figure 6.12: C++ `LockingCQueue` class

would need to be correctness checks, such as in the spawn function no process with the same id already exists.

This almost defeats the purpose of using a dependently typed language as its type system is so powerful it is able to prove that something is correct by construction, rather than checking after construction.

6.2 Second Design

In the second design, there were a couple of goals to be achieved by the design, firstly ensure the correctness of the id's for constructing `PID` and `Channel` elements and secondly expand with the new `Channel` type that would depend on the `PID` objects that it connected.

6.2.1 New PID

In this design, the `PID` was removed because there was not any benefit to it when compared to just storing the pid as a number. This was because throughout the design when it used the `PID` type, it would have to deconstruct the object to get the id, so to make things easier and clearer, only the pid number is used, rather than the type.

6.2.2 New Channel

The new `Channel` type, shown in Figure 6.13, is similar to the original but it has two new indexes that represent the `PID` sending into the channel and the `PID` receiving from the channel. The constructor is also extended to contain the two `PID` as well as constraints to ensure that the two `PID` are the same as the ones that appear in the type.

The benefit of this design is that the concurrency primitives could be extended to ensure more correctness when sending or creating channels. For example the `send` function could be changed so that it would only type check when the sending process is the correct process, and vise versa for `recieve`.

There was some large drawbacks to this approach. The first is that it forced a channel to only pass messages between two processes instead of N processes. Secondly it would result in the primitives

```
data Channel
    (A : Type)
    (chid : Nat)
    (sendPid : {id : Nat | PID id})
    (recievePid : {id : Nat | PID id}) : Type where
  MkChannel of (id : Nat) (sPid : {id : Nat | PID id})
               (rPid : {id : Nat | PID id})[id = chid]
               [sPid = sendPid] [rPid = recievePid]
```

Figure 6.13: `pi-forall` new `Channel` class

being much more complex and would take a lot more time to design as well as use later when designing the parallel patterns.

For those reasons this goal was scrapped, but was planned to be added again as a new `Channel` type that was restricted to linking between two processes, but there was no time for implementing it.

There was one small change, but it was with how the channel was represented in C++. Instead of the `LockingCQueue` being templated the same type `A` that appears in the `pi-forall` definition, it was instead templated over `_Maybe<A>`, the C++ translated `Maybe` class. This was to allow for a signal, the `Nothing` element, to be sent over the channel signifying a processes has ended it's input.

6.2.3 Correctness by Construction

As the previous goal of the new `Channel` type was discarded, the design focus could instead be spend on ensuring the correctness of creating, as well as using, the `PID` and `Channel` type. For a `PID` or `Channel` type to be correctly constructed, the id must not already be used. By keeping track of all the ids of the currently created `PID` elements, a proof that an new id is not in the set of all ids will mean that a `PID` indexed by the new id will not clash with any other `PID` elements. The same will hold for the `Channel` type and it's id.

6.2.4 Elem Type

```
data Elem (a : Type) (x : a) (B : List a) : Type where
  Here of (x : a) (xs : List a) [B=Cons x xs]
  There of (y : a) (x : a) (xs : List a)
           (later : Elem a x xs) [B=Cons y xs]
```

Figure 6.14: `Elem` datatype `pi-forall` implementation

The datatype Elem is defined in Figure 6.14 and is indexed by the type `a`, an element `x` of type `a` and a `List` of type `a` named `B` and is the proposition that $x \in B$, i.e. `x` is a member of the list `B`. An element of this type would be a proof for $x \in B$.

There are two constructors for the type, as a proof of membership can be formed in two ways, either the element is the head of the list, or the element is in a later part of the list. The `Here` constructor contains two parameters, `x`, which is the head of the list, and `xs`, which is the tail of the list. There is also a constraint `B = Cons x xs` that guarantees the `Here` constructor is only constructible when the list `B` has `x` as its head and `xs` as its tail. Due to `x` being the head of the list, the `Here` constructor is a proof that $x \in B$.

The `There` constructor takes four parameters: `y` is the head of the list, `x` is the element to check for membership, `xs` is the tail of the list and `later` is an element of type `Elem a x xs` which is a proof that $x \in xs$. There is also a constraint that guarantees `y` and `xs` are the head and tail of the list `B` respectively. Due to `xs` guaranteed to be the tail of the list `B`, and a proof that $x \in xs$, the `There` constructor is a proof that $x \in B$.

An example of elements of the `Elem` type are shown in Figure 6.15. The element `a` is a proof of the proposition that $2 \in [3, 2, 1]$, while the proposition is $0 \in [3, 2, 1]$ is false so the there is not possible element that `b` could be as the type is uninhabited.

```
1 a : Elem Nat 2 (Cons 3 (Cons 2 (Cons 1 Nil)))
2 a = There 3 2 (Cons 2 (Cons 1 Nil)) (Here 2 (Cons 1 Nil))
3
4 b : Elem Nat 0 (Cons 3 (Cons 2 (Cons 1 Nil)))
5 b = -- Unihabited
```

Figure 6.15: `Elem` elements

6.2.5 Dec Type

It can be quite tedious to manually find an element of the `Elem` type for any list, or find it uninhabited, which is why a function to create the proof from a list and an element is desired, such as `f : (xs : List A) -> (x :A) -> Elem A x xs`. However, this is impossible to create as it is not always possible to produce an element of the `Elem` type, i.e. when the type is uninhabited as the proposition does not hold. The solution to this problem is the `Dec` datatype, shown in Figure 6.16, and is the decidability property of the indexed prop. As the proposition is either true or false, two constructors are given for the type. The `Yes` constructor can only be constructed when the proposition does hold, due to containing the `prf` value, which is a proof that the proposition holds. The `No` constructor can only be constructed when the proposition does not hold, with `cont` containing a proof that the given proposition does not hold. This uses the `Void` type defined on Line 5 so that `cont` is a proof of contradiction for the proposition `prop`.

```
1 data Dec (prop : Type) : Type where
2   Yes of (prf : prop)
3   No  of (cont : prop -> Void)
4
5 data Void : Type where {}
```

Figure 6.16: `Dec` datatype `pi-forall` implementation

As the `Dec` type can always decide over any proposition, the function could instead decide over the `Elem` proposition and by inspecting the result, a proof could be obtained from the `Yes` constructor, or a counter proof from the `No`.

6.2.6 deqEqNat Function

Before deciding if a number is in a list of numbers, first, proving equality between two numbers must be done. This is done by the `decEqNat` function, which decides if two natural numbers, `a` and `b`, are equal and is shown in Figure 6.18. It will produce an element of type `Dec (a = b)`, which

can only be either contain a `Yes` object that contains a proof that the proposition, `a = b`, holds or a `No` object with a proof that the proposition does not hold.

In Figure 6.18, lemmas are used that need to be explained before going through the `decEqNat` function and they are shown in more detail in Figure 6.17. The first lemma in Figure 6.17 is the `sym` lemma, which provides a proof that `y = x` when given a proof that `x = y`. This lemma proves the symmetric property of propositional equality. The proof is obtained by substituting into `Refl`, which is a proof that `x = x`, and by using the proof that `x = y` to substitute into `Refl`, a proof that `y = x` can be obtained. This is the `pi-forall` equivalent of the Natural Deduction proof in Figure 2.17 on Page 22.

The lemma on Line 5, `f_equal` proves that function application is pure; if the input is equal, then so it the output after an application of a function `f`. It is implemented very similarly to the `sym` lemma and substitutes the proof given to the lemma, that `x = y` into a proof that `f x = f x` to obtain a proof for the proposition `f x = f y`.

Line 6 shows the `ZnotS` lemma, which is a proof of the Peano axiom $\forall x(0 \neq S(x))$, which states that `Zero` is not the successor of any number. This is a simple proof by contradiction, as the variable `r` has type `Zero = Succ n` which, by using the `contra` expression, finds a contradiction when the constructors on each side of the equality are different, the contradiction of `r` can be found, which provides a proof for `(Zero = Succ n) -> Void`.

The `negEqSym` lemma on Line 13 proves that inequality is symmetric, just like propositional logic. It does this using the `sym` lemma on `h` which is a proof that `b = a`, to obtain a proof for `a = b`, which from `p` is uninhabited, hence proving `(b = a) -> Void`.

The final lemma in the figure is `succInjective` on Line 17. This proves the injective property for the `Succ` constructor. It does this by using the `f_equal` lemma on two numbers, `Succ left` and `Succ right`, with the function being the `pred` function which gets the previous number.

The `decEqNat` function can be read as a proof by induction on the two numbers `a` and `b`. There are three base cases and an inductive step. Case analysis is used based on the four possible cases of `a` and `b`. As the Natural numbers are defined according to Peano[26] arithmetic there are two possible cases per number, `Zero` or `Succ n`, for some Natural number `n`.

The first base case, one Line 4, is when both numbers are `Zero` and they can be proven to be equal using reflexivity and therefore a `Yes` object can be constructed with a proof that `Zero = Zero`.

The second base case, on Line 5, is when `a` is `Zero` and `b` is the `Succ n` of some `n`. The proof that `(Zero = Succ n) -> Void` is given by the `ZnotS` lemma, and a `No` object can be constructed.

The third base case, on line 7, when `a` is the `Succ n` of some `n` and `b` is `Zero`, which again uses `ZnotS` to produce a proof that `(Zero = Succ n) -> Void` and then passes this proof to the lemma `negEqSym` to produce the proof that `(Succ n = Zero) -> Void` so that a `No` object can be constructed. Unlike in the second case lemma `negEqSym` is needed as the object constructed must be of type `Dec (Succ n = Zero)` and it is impossible to the `No` object for that type using a proof that `(Zero = Succ n) -> Void`), hence the use of the lemma.

Finally there is the inductive step, on lines 8-10, and is when `a` is `Succ n` for some Natural number `n` and `b` is `Succ m` of some Natural number `m`. It recursively calls the function on the numbers `n` and `m` and splits based on the two possible cases, which are the `Yes` and `No` cases. In the `Yes` case, the variable `p`, seen on line 9, is a proof that `(n = m)`, and using the lemma `f_equal` we can produce a proof that `Succ n = Succ m` and hence another `Yes` object. In the `No` case, the variable p, on

```
1  sym : [A:Type] -> [x:A] -> [y:A] -> (x = y) -> y = x
2  sym = \ [A] [x] [y] pf .
3    subst Refl by pf
4
5  f_equal :  [A:Type] -> [B : Type] -> [f : A -> B] -> [x:A]
6          -> [y:A] -> x = y -> f x = f y
7  f_equal = \[A][B][f][x][y] pf .
8    subst Refl by pf
9
10 ZnotS : (n : Nat) -> ((Zero = Succ n) -> Void)
11 ZnotS = \n r . contra r
12
13 negEqSym : [t : Type]-> [a : t]-> [b : t] -> (a = b -> Void)
14         -> (b = a -> Void)
15 negEqSym = \[t] [a] [b] p h . p (sym [t] [b] [a] h)
16
17 succInjective : [left : Nat] -> [right : Nat]
18              -> (p : Succ left = Succ right) -> (left = right)
19 succInjective = \[left] [right] p .
20   f_equal [Nat] [Nat] [pred] [Succ left] [Succ right] p
21
22 pred : Nat -> Nat
23 pred = \n . case n of
24     Zero -> Zero
25     Succ m -> m
```

Figure 6.17: Lemmas used in `deqEqNat` `pi-forall` implementation

line 10, is a proof that `(n = m) -> Void`. We can create a lambda function `\h .e` which has type `(Succ n = Succ m) -> Void`, meaning that the variable h has type `(Succ n = Succ m)`. For this lambda expression to be a well formed expression, `e` needs to have type `Void`. If `e` was equal to `p e'` where `e'` was of type `(n = m)` then the expression `e` would have type `Void`. Using the lemma `succInjective`, `e'` can be produced, and a proof that `(Succ n = Succ m) -> Void` can be created and used to produce a `No` object.

6.2.7 pi-forall Bug

When developing the `decEqNat` a major bug was found in pi-forall type checker which meant that lots of functions that should type check did not.

It occurred when using an old version of the `Dec` type, where a type appeared in a relevant way and is shown in Figure 6.19. If an element of the type was already constructed and a substitution was done on the prop variable so that instead of `prop`, it is `a`, where there is proof that `a = prop`. This means that from the constraint would become `[p = a]`. However this did not happen, instead it stay the same, but now `prop` refers to nothing. It occurred because the propagation of the substitution was never carried out through constraints.

This bug took two weeks to find, which cost a lot of development time, but once it was found, it was an extremely simple one-line fix, which ensure the substitution was propagated.

```
1  decEqNat : (a : Nat) -> (b : Nat) -> Dec (a = b)
2  decEqNat = \a b . case a of
3       Zero -> case b of
4                 Zero -> Yes Refl
5                 Succ n -> No (ZnotS n)
6       Succ n -> case b of
7                   Zero -> No (negEqSym [Nat] [Zero] [Succ n] (ZnotS n))
8                   Succ m -> case (decEqNat n m) of
9                                 Yes p -> Yes (f_equal [Nat] [Nat] [\x . Succ
     x] [n] [m] p)
10                                No p -> No (\h . p (succInjective [n] [m] h))
```

Figure 6.18: `deqEqNat` `pi-forall` implementation

```
1  data Dec (prop : Type) : Type where
2      Yes of (p : Type) (pf : p) [p = prop]
3      No of (p : Type) (cPf : p -> Void) [p = prop]
```

Figure 6.19: `pi-forall` old `Dec` definition

6.2.8 isElem Function

Figure 6.21 gives the definition for the function `isElem`, which is a decidable proof for the proposition that a number `x` is a member of the list `xs`, or $x \in xs$. The return type of the function is `Dec ( Elem Nat x xs)` which will either be a `Yes`, which will contain a proof that $x \in xs$, or a `No`, which contains a proof that $x \notin xs$, i.e. `x` is a member of the list `xs`. The function is not generic over all list types because it must use a function to decide if two elements of the list are equal, and there is no way to have that be generic. The list type was limited to `Nat` as this was all that was needed. If other types of lists needed to be checked for membership then a new function would need to be created.

Lemmas

On Lines 3 and 10, there are to functions `xNotElemOfNil` and `neitherHereNorThere` which are lemmas to help in the proof. `neitherHereNorThere` is a lemma which when given a proof that $x \neq y$, `xneqy`, and $x \notin xs$, `xninxs`, then it can construct a proof that $x \notin$ `Cons y ys`. The variable `p` is actually of type `Elem Nat x (Cons y xs)` and is shown to be uninhabited by considering both the possible cases from its constructors, `Here` and `There`. In the `Here` constructor, a contradiction is formed as the `Here` constructor has a constraint that ensures that it can only be formed when `x` is the head of the list, however, `y` is the head of the list and there is a proof that $x \neq y$, called `xneqy` It is similar in the `There` constructor but a contradiction is formed as the `There` constructor ensures that the element appears later in the list, however `xninxs` is a proof that `x` doesn't appear later, hence the contradiction.

`xNotElemOfNil` is a lemma which produces a proof that `x` is not in the empty list `Nil`, or $x \notin$ `Nil`. This lemma shows that the variable `p`, which has type `Elem Nat x Nil` is uninhabited by pattern matching, which produces no possible constructors for `p`.

```
1  neitherHereNorThere : [x : Nat] -> [y : Nat] -> [xs : List Nat]
2                      -> ((x = y) -> Void) -> ((Elem Nat x xs) -> Void)
3                      -> ((Elem Nat x (Cons y xs)) -> Void)
4  neitherHereNorThere = \[x] [y] [xs] xneqy xninxs p . case p of
5      Here a as -> xneqy Refl
6      There b a as prf -> xninxs prf
7
8
9  xNotElemOfNil : [x : Nat] -> ((p : Elem Nat x Nil) -> Void)
10 xNotElemOfNil = \[x] p . case p of {}
```

Figure 6.20: `isElem` lemmas `pi-forall` implementation

```
1  isElem : (x : Nat) -> (xs : List Nat) -> Dec (Elem Nat x xs)
2  isElem = \x xs . case xs of
3      Nil -> No (xNotElemOfNil [x])
4      Cons y ys ->
5        case (decEqNat x y) of
6           Yes p -> Yes (Here y ys)
7           No xneqy ->
8            case (isElem x ys) of
9               Yes xinys -> Yes (There y x ys xinys)
10              No  xninys -> No (neitherHereNorThere [x] [y] [ys] xneqy
    xninys)
```

Figure 6.21: `isElem` `pi-forall` implementation

The function `isElem` is a proof by induction on the `List xs`. The proof involves two cases: the base case and the inductive case.

The base case, shown on Line 3, is when `xs` is `Nil`, i.e. the list is empty. In an empty list, it is impossible for any element to be a member of the list, and a proof of this can be constructed using the lemma `xNotElemOfNil` which will produce a proof with the type `(Elem Nat x Nil) -> Void` which is a proof that the `x` is not an element of a list with value `Nil`. This proof can then be used as an argument to construct a `No` which has type `Dec (Elem Nat x xs)`.

The inductive case(Lines 4-10) is when the the list `xs` is a non empty list, i.e. `Cons y ys`, and where `y` and `ys` is the head and tail of the list `xs`. The first step is to call `decEqNat`, on Line 5, to decide if `x` and `y` are equal. There are two possibilities from this: either a `Yes` object is found, or a `No`. In the case of a `Yes` object, on Line 6, `p` has the type `x = y`, and is a proof that $x = y$, meaning that the head of the list, `y`, is equal to the element we are checking membership for, `x`; hence a proof that $x \in xs$ can be directly constructed using the `Here` constructor using, as arguments, `y` and `ys`. In the second case, when there is a `No`, on Line 7, `xneqy` has the type `(x = y) -> Void` and is a proof that $x \neq y$, meaning that the head of the list, `y`, is not the element we want to check membership for. The next step is to check the remainder of the list, which is done through a recursive call to `isElem`, Line 8, on `ys`, the tail of `xs`. In the case a `Yes` object is produced, on Line 9, `xinys` has the type `Elem Nat x ys`, which is a proof that $x \in ys$, and can be used with the `There` constructor to produce an element of the type `(Elem Nat x xs)` which can be applied to the `Yes` constructor. If a `No` object was produced, on Line 10, then `xninys` has the type `(Elem Nat x ys) -> Void)` and is a proof that $x \notin ys$. The lemma `neitherHereNorThere` takes, as parameters, a proof with the type

`(x = y) -> Void` and another proof with type `(Elem Nat x ys) -> Void` and will return a proof that has type `(Elem Nat x (Cons y ys)) -> Void`, which is equivalent to the type `(Elem Nat x xs ) -> Void` as in this case `xs = Cons y ys`. This will be a proof that $x \notin xs$ and can be applied to the `No` constructor to produce an element of the type `Dec (Elem Nat x xs)`.

Note on proofs

Note that the functions `decEqNat` and `isElem` were based on code provided by my supervisor.

6.2.9 Updated Primitives

Using the above proofs, the original primitives were re-designed to have proofs for the correctness of the ids. Another design decision was made during this rewrite: to try and write as much of the implementation in `pi-forall` as possible, minimising the C++ code.

pidSet and chidSet

Being passed around all the primitives is the `pidSet` and the `chidSet` which is a set of the process ids and a set of channel ids with the type being `List Nat`. Only the ids are stored in the sets rather than the `PID` and `Channel` elements to make it easier. They are named set because at all times, even thought their type is `List`, they will behave like a set as all their elements are unique.

link

In the new updated link function, shown in Figure 6.22, the id for the channel is now passed in as an argument and the idSet for the channel ids. Line 2 shows the argument for the proof that the id is not already in the idSet. The return type is again in the `IO` monad as creating the channel is an `IO` action, but also returns a type, which is a `Maybe` type that is indexed by a dependant tuple with 5 elements, which is like a dependant pair but with 5 elements. The reason for the `Maybe` type is to indicate if the channel was created, so only when the channel is created is the tuple returned. The structure of the tuple is as follows:

- The first element is the channel id, named `chid`.
- The second element is the channel itself, with the channel id being `chid`.
- The third element is the new id set named `newIdSet`
- The fourth element is a proof that the `chid` used for the channel is the same number as the id passed as an argument `id`.
- The final element is a proof that the new id set, `newIdSet` is equal to the list where the `id` is the head of the list and `idSet`, the set passed in, is the tail of the list.

Only the first three elements of the tuple contain the data needed after creating a channel, but the last two ensure that a proof has been provided in the function's implementation that the id and idSet produced are correct. This idea of extra proofs in a tuple is used throughout the primitives and the patterns in the next chapter to ensure the correctness of the concurrency.

The actual implementation of `link` the uses the function `prim_create_channel`, which is a function that is written in C++ to create the channel. It then returns success or failure, which allows the `pi-forall` implementation to exist and create the relevant channel and proofs.

```
link : [A : Type] -> (id : Nat) -> (idSet : List Nat)
    -> ((Elem Nat id idSet) -> Void)
    -> IO (Maybe ({chid : Nat | (Channel A chid) * {newIdSet : List Nat |
    (id = chid) * (Cons id idSet : List Nat) = newIdSet}}))
link = \ [A] id idSet pf . case (prim_create_channel [A] id) of
    True    ->
      returnIO
      (Just ((id, (MkChannel id, (Cons id idSet, (Refl, Refl))))))
    False   ->
      returnIO Nothing
```

Figure 6.22: `pi-forall` `link` function

send

In the `send` function, the only change is that the element being sent over the channel, `x`, now has the type `Maybe A` rather than `A`. This is to help propagate the termination over a channel, so actual messages will be `Just a` and the terminator is sending `Nothing`. The implementation calls the `channelEnqueue` function, a function written in C++ to add the element x to the correct `LockingCQueue`.

```
send : [A : Type] -> (id : Nat) -> (x : Maybe A) -> Channel A id -> IO
   Unit
send = \ [A] id x ch . channelEnqueue [A] id x ch
```

Figure 6.23: `pi-forall` `send` function

6.2.10 receive

The `receive` function, like the `send` has only changed slightly, with a `Maybe` being returned from the `IO` action. It also has an implementation that calls the `channelDequeue` function, which is another function written in C++ that dequeues a value from the correct `LockingCQueue`.

```
receive : [A : Type] -> (id : Nat) -> Channel A id -> IO (Maybe A)
receive = \ [A] id ch . channelDequeue [A] id ch
```

Figure 6.24: `pi-forall` `receive` function

end

The function `end` remained the same as before, but instead of the implementation being written in C++, it was written in `pi-forall` and the code is shown in Figure 6.25.

```
end : IO Unit
end = returnIO [Unit] ()
```

Figure 6.25: `pi-forall` `end` function

6.2.11 spawnAndRun

The functions spawn and run were updated much like the link function to use a pidSet, however as stated in Section 6.2.2, there is no restriction now on the ordering of spawn, link and run, so spawn and run were combined into spawnAndRun. For brevity, the updated versions of spawn and run will not be given, but it is extremely similar to spawnAndRun, which is now the main primitive for creating a process.

The function is shown in Figure 6.26. The first two arguments are the process id, pid and the set of currently used process ids, pidSet. The third argument is a proof that pid is not already in the list pidSet, and the fourth argument is the function to run on the process. The function will return a Maybe object from within the IO monad and will return a dependant tuple with 4 elements that is described below:

- The first element is the pid for the created process, named newPid.
- The second element is the new pid set that will contain newPid, named newPidSet.
- The third element is a proof that the pid passed as a parameter, pid is equal to the pid returned, newPid.
- The fourth element is a proof that newPidSet is the list with the two variables passed as parameters pid and pidSet as its head and tail.

```
1 spawnAndRun : (pid : Nat) -> (pidSet : List Nat )
2             -> ((Elem Nat pid pidSet) -> Void) -> (process : IO Unit)
3             -> IO (Maybe ({newPid : Nat | {newPidSet : List Nat | (pid =
    newPid) * ((Cons pid pidSet : List Nat) = newPidSet) }}))
```

Figure 6.26: pi-forall spawnAndRun function

The function has no pi-forall implementation, instead has a C++ implementation, which the important excerpt is given in Figure 6.27. It also slightly changes the type signature of the function, so that instead of process having type IO Unit, it instead has C++ type std::function<void(void)> The reason for this is that when passing the function around if it is the generated _IO class then it will fully evaluate the function; this defeats the purpose of running in a separate thread. By storing it as a std::function the execution can be delayed until Line 2 in the spawnAndRun function, ensuring it is evaluated in a separate thread. This is also why the function being run in the thread constructor is a lambda function that evaluates the process, as it is the only way to call the function when it starts the new thread. As the type of process parameter for the spawnAndRun function was changed, this meant that in the translation, an update was made when translating a function, so that when translating an application of the spawnAndRun function, the process expression is wrapped in a lambda to turn the C++ type from _IO<_Unit> to std::function<void(void)>, much like the wrapping of process on Line 2 of Figure 6.27.

The spawnAndRun function will also update the thread map so that a new thread has been added.

If the thread creation fails, an exception will be through where the catch clause will return the Nothing object, otherwise after the try block, there is code to generate the tuple to return.

```
try {
  std::thread* t = new std::thread([process]() { process(); });
  addThread(pid, t);
} catch {
  ... // Returning Failure
}
... // Returning Success
```

Figure 6.27: Excerpt from C++ implementation of `spawnAndRun` function

6.3 Final Design

One final change was made to the previous design, which changed the type of the `pidSet` and `chidSet` from `List` to `Vec`. This was because when designing the parallel patterns, a proof on the length of the `pidSet` and `chidSet` was needed, so the obvious choice was to move to using the `Vec` type.

6.3.1 Updated Proofs

As the sets have changed from `List` to `Vec`, the `Elem` type will not work, hence a new type `ElemVec` was created is the `Vec` version of `Elem`.

ElemVec Type

```
data ElemVec (A:Type) (n:Nat) (x:A) (v:Vec A n) : Type where
  HereV  of [m:Nat] (xs : Vec A m) [n = Succ m] [v = ConsV [m] x xs]
  ThereV of [m:Nat] (xs : Vec A m) (y:A) (later : ElemVec A m x xs) [n =
   Succ m] [v = ConsV [m] y xs]
```

Figure 6.28: `ElemVec` datatype `pi-forall` implementation

The `ElemVec` datatype is defined in Figure 6.28 and is similar to the `Elem` type defined on Page 66, which is a proof of membership in a `List`, provides a proof of membership in a `Vec`. The `ElemVec` type is indexed by the type `a`, the length of the `Vec` named `n`, an element `n` of type `a` and a `Vec` of type `v` with length `n`. An element of the type `ElemVec A n x v` is a proof that $x \in v$.

The two constructors, `HereV` and `ThereV` are the two ways the type can be constructed. This `HereV` constructor contains two parameters and two constraints. The first is the irrelevant argument, `m`, which appears in the second relevant argument, `xs`, which is a `Vec` with length `m`. The first constraint guarantees that `m = Succ n` where `n` is the length of `v`, which appears in the type. As `m` is the length of `xs` and `n` is the length of `v`, this will ensure that the length of `xs` is one less that `v`. The second constraint creates an obligation that `v = ConsV [m] x xs`, which means that the element `x` is the head of the `v`, and the `xs` parameter is the tail of `v`. Due to `x` being the head of `v`, the `HereV` constructor is a proof that $x \in v$.

The `ThereV` constructor contains 4 parameters. The first two are the same as in the `HereV` constructor, `m` and `xs`. The remaining two arguments are `y` of type `A`, which is the head of the `v`, and `later`, which is an element of the type `ElemVec A m x xs` that is a proof that `x` is a member of `xs`. The first constraint provides a guarantee that the value of `m` is equal to `Succ n`, which similarly to the `HereV`

constructor, ensures the length of `xs` is one less than the length of `v`. The second constraint requires that `v = ConsV [m] y xs`, which means that `y` and `xs` are the head and tail of `v`, respectively. As `later` provides a proof that $x \in xs$, and the second constraint ensuring `xs` is the tail of `v`, the `ThereV` constructor gives a proof that $x \in v$.

isElemVec

Figure 6.29 defines the function `isElemVec`, which is a decidable proof for the proposition that a number `x` is a member of the `Vec` xs. This is very similar to the `isElem` function defined earlier and the code is almost the same, just slightly different to account for using `Vec` rather than `List`.

There are two lemmas used in the function `isElemVec`, namely `xNotInNilV` and `neitherHereNorThereV`, which is the `ElemVec` versions of the `xNotInNilV` and `neitherHereNorThere` used with `Elem` on Page 71.

```
1  isElemVec : [n : Nat] -> (x : Nat) -> (xs : Vec Nat n) -> Dec (ElemVec
       Nat n x xs)
2  isElemVec = \ [n] x xs . case xs of
3    NilV           -> No (xNotInNilV [Nat] [n] [x])
4    ConsV [m] y ys ->
5      case (decEqNat x y) of
6        Yes pf         -> Yes (subst (HereV [m] ys) by pf)
7        No xneqyPf   ->
8          case (isElemVec [m] x ys) of
9            Yes pf           -> Yes (ThereV [m] ys x pf)
10           No xninysPf   -> No (neitherHereNorThereVec [m] [x] [y] [ys]
       xneqyPf xninysPf)
```

Figure 6.29: `isElemVec` `pi-forall` implementation

The `isElemVec` function will not be explained as there is no difference between the `isElem` function, which was explained in detail, other than using the `ElemVec` versions of the lemmas and using `Vec` type.

6.3.2 Updated Primitives

The `link` and `spawnAndRun` functions needed to be updated when the sets were changed to `Vec`, as they needed the sets to have the `Vec` type and use the `ElemVec` type instead of the `Elem` type. The type signatures for both are shown in Figure 6.30.

A new irrelevant parameter is added to both named `n`, which is the size of the `Vec` for the sets. The return set also has a new correctness check, as the size of the returned set must be bigger by one; hence the size is `Succ n`.

```
1 link : [A : Type] -> (id : Nat) -> [n : Nat]
2     -> (idSet : Vec Nat n) -> ((ElemVec Nat n id idSet) -> Void)
3     -> IO (Maybe ({chid : Nat | (Channel A chid) * {newIdSet : Vec Nat (
    Succ n) | (id = chid) * (ConsV [n] id idSet : Vec Nat (Succ n)) =
    newIdSet}}))
4
5 spawnAndRun : (pid : Nat) -> [n : Nat] -> (pidSet : Vec Nat n)
6             -> ((ElemVec Nat n pid pidSet) -> Void) -> (process : IO Unit)
7             -> IO (Maybe ({newPid : Nat | {newPidSet : Vec Nat (Succ n) |
    (pid = newPid) * ((ConsV [n] pid pidSet : Vec Nat (Succ n)) =
    newPidSet) }}))
```

Figure 6.30: `link` and `spawnAndRun` `pi-forall` implementation

6.4 Future Work

The most obvious area to improve the concurrency primitives is designing a method of hiding the `pidSet` and `chidSet`. A potential method could be introducing a new `Process` monad, which will replace the `IO` monad. There could also be work done designing a system that will automatically generate unique process ids and channel ids so that a user would not have to provide the ids themselves.

The `PID` and `Channel` type could also be redesigned to ensure that communication is happening between the correct processes.

The actual C++ parallel runtime system could also be improved, for example, instead of every `pi-forall` process being mapped to a C++ thread, it could be represented another way and then the tasks are placed on a thread pool, which could make it more efficient.

6.5 Summary

In this chapter, the design has been laid out for the concurrency primitives. The past designs were all shown, and also reasons for the iterations so that the final design is fully justified. Proofs regarding the membership of `List` or `Vec` were also described, as well as the important type `Dec` that decides a proposition. Finally, an outline of potential areas of interest regarding exploring the concurrency primitive in more depth was given.

Chapter 7

Parallel Patterns

In this chapter, there is a discussion on the parallel patterns designed and implemented in `pi-forall`. Section 7.1 gives an overview of the overall design decisions made when designing the patterns. Section 7.2 details the farm pattern, its `pi-forall` implementation and all the proofs used to ensure correctness. It also discusses the functions used to create elements of the pattern. Section 7.3 details the pipeline pattern and its `pi-forall` implementation, much like the previous section. Section 7.4 gives an overview of some of the limitations of the parallel patterns designed and Section 7.5 discusses potential avenues for potential work that builds on the work outlined in this chapter.

7.1 Overall Design

The plan for creating the parallel patterns was to create a type representing the pattern. The types would have some correctness checking through their definition, and functions would be used to create the patterns, which would include proofs that the patterns were created correctly. This is similar to the approach taken in the previous chapter where functions like `link` and `spawnAndRun` would have proofs for correctness.

When using a pattern, it was decided that the input to a pattern would be a list, and the function to create the pattern would also run the pattern and collect the output into a list to be returned. This means that the main program will wait until the entire list has been collected, so the processes in the pattern will be fully synchronised.

It was decided that for simplicity, only the two most basic skeletons, farm and pipeline, would be implemented as patterns. However, it aimed to be designed to allow other patterns to be created using the farm and pipeline pattern as a reference.

7.1.1 Parsing Problem

When developing the patterns, an issue was discovered in `pi-forall`; the parser was extremely slow for larger functions. For example, a function with 200 lines could take many hours to parse. This can be a serious issue because of `pi-forall`'s verbosity, a function that binds 5 `IO` actions together can be over 200 lines, meaning the issue would arise. The only solution to the problem would be to re-write the parser in a much more efficient manner. However, there was not enough time to start this task, so the larger `pi-forall` functions had to be split into smaller functions even though there was no design reason.

7.1.2 Types Problem

There was another issue that arose during the design around types. The pipeline type was planned to store the channels at each stage. This would require a list of the channels; however, the type of each channel will be different as `Channel` is indexed by a type and a number, so it is impossible to have a list store them all. To get around this, storing a dependant tuple with a type such as `{A : Type | {id : Nat | Channel A id}}` would work. The issue with that solution is the `A` is relevant and not irrelevant, which, due to the limitations of the translator discussed in Section 5.11 on Page 58, is impossible to translate. This meant that an intentional limitation was created so all the channels could be contained within the type by forcing the functions at each stage to only refer to a single type, `A`, so all channels would be indexed by the same type `A`.

The limitation was carried over to the farm, so a pipeline and farm pattern could be composed together to form a more complex pattern.

7.2 Farm

The first pattern designed was the farm pattern, with a diagram of a general farm skeleton shown in Figure 7.1. There is a slight change from the skeleton when designing the farm in `pi-forall`. The emitter, workers and collector are all meant be run in parallel with each other and the main program thread. However, the `pi-forall` design instead runs the collector on the main program thread, meaning the workers, emitter and collector all run in parallel with each other. The main thread would be blocked until the pattern has been fully evaluated anyway, so to save creating another thread, the main thread would be used as the collector. Another small change from the diagram is only to have a single channel connecting all the workers to the collector. This is because the type `Channel` has no restriction that it can only communicate between two threads, which will save creating more channels.

Based on the above decisions, a farm with n workers has $n + 1$ channels; n for emitter-to-worker communication and one for worker-to-collector communication. The same will be true for the number of threads running, which would be $n + 1$ again; n for the workers and one for the emitter.

This design will also ensure that the synchronisation of the underlying threads is done. This is because when the output list has been fully formed, the termination signal must have been sent through the entire pattern, so the functions running on the threads will have all finished and are all synchronised.

The type `Farm`, shown in Figure 7.2, is the farm pattern. It is indexed by a type `A`, which is the type of messages, and a number `n`, representing the number of workers in the farm. It has a single constructor, where `m`, is the number of workers in the farm, `f` is the function to be farmed, `chs` is the list of channels. There is also a constraint ensuring the number of workers in the constructor is the same as the number in the type. There are a couple of constraints that ensure the correctness of a farm:

- The function is constrained by its type to be a function from type `A` to `A`, which is a system limitation.
- The number of channels is constrained to be one more than the number of workers, which ensures the correct number of channels is associated with the farm.
- The channel's type is indexed by the type `A` from the Farm indexes, which ensures the channels associated are also of the correct type.

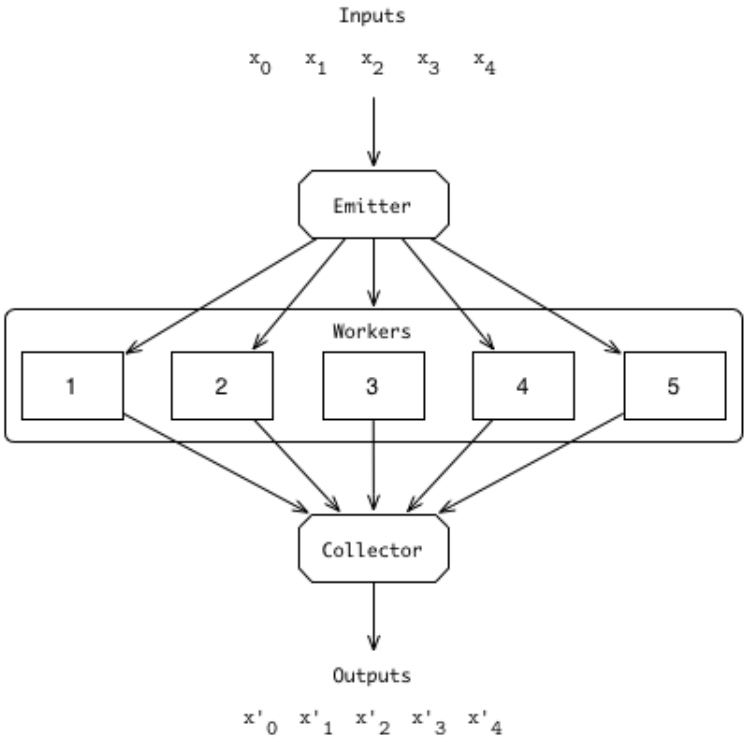


Figure 7.1: Farm Skeleton[18]

```
1 data Farm (A : Type) (n : Nat) : Type where
2   MkFarm of (m : Nat) (f : A -> A)
3             (chs : Vec {id : Nat | Channel A id} (Succ n)) [m = n]
```

Figure 7.2: `Farm` datatype `pi-forall` implementation

What the `Farm` type doesn't have constraints for is regarding the processes being spawned and also that the channels are only used in this pattern, and not another farm. Those are ensured through the function to create a farm as if it spawned the processes and created the channel, then proofs must have been supplied to show that the new ids are all unique, and if they are not passed out of the function, then the objects cannot be used elsewhere, hence ensuring that the channels and processes are not used in other patterns.

7.2.1 Proofs

As the channels and processes are created within the function to create a farm pattern, then the `pidSet` and `chidSet` need to be passed in, and also two new sets for the pids and chids that will be used for the newly created processes and channels. This means that a proof for every pid $\in$ pids must have a proof that it is not in `pidSet` and also a proof is needed to show that pids is indeed a set, where every element is unique. The same will apply to the chids and `chidSet`.

Instead of passing in a list of proofs showing that each element of one set is not in another set, a new proof was created that showed two `Vec` are disjoint.

DisjointVec

```
1 data DisjointVec (A : Type) (n : Nat) (a : Vec A n)
2                  (m : Nat) (b : Vec A m) : Type where
3   DNilV of [n = 0] [a = NilV]
4   DConsV of [n1 : Nat] (x : A) (xs : Vec A n1)
5             (pfForx : (ElemVec A m x b) -> Void)
6             (DisjointVec A n1 xs m b) [n = Succ n1] [a = ConsV [n1] x xs]
```

Figure 7.3: `DisjointVec` datatype `pi-forall` implementation

The `DisjointVec` datatype is defined in Figure 7.3 and provides a proof that two `Vec`, `a` and `b` are disjoint, or $a \cap b = \emptyset$. The `DisjointVec` is indexed by 5 values:

- The first is the type `A`, which represents the type of elements in both of the `Vec`, `a` and `b`.
- The second is the number `n`, which represents size of the `Vec`, `a`.
- The third is the `Vec a` that has the length `n`.
- The fourth is the number `m`, which represents the size of the `Vec`, `b`.
- The fifth is the `Vec b` that has the length `m`.

A method of proving that two sets are disjoint is by showing that all the elements in one of the sets do not appear in the other, which is the same way the type ensures it can only be constructed when proofs that all the elements of `a` do not appear in `b`.

There are two constructors for the `DisjointVec` as there are two cases of `a`; it is either empty with the `NilV` constructor, or it is non-empty with the `ConsV` constructor. The `DNilV` constructor can only be constructed when the length of `a` is 0, or in other words `a` is `NilV`. This handles the first case for proving two `Vec` are disjoint.

The second case is covered by the `DConsV` constructor, which proves that the head of the `Vec a` is not in `b` and also proves that the tail of `a` is disjoint to `b`. When both conditions are true, the two `Vec` are disjoint, and the `DConsV` constructor can be used. To make sure this holds, the `DconsV` constructor has many arguments, where `n1`, `x` and `xs` are the length, head and tail of `a` due to two constraints on Line 6. `pfForx` which has the type `(ElemVec A m x b) -> Void`), which provides a proof that the head of `a`, `x`, is not in `b`. The next argument has the type `DisjointVec A n1 xs m b`, which is a proof that the tail of the `a` is not in `b`. As the constructor proves the head and tail of `a` is not in `b`, then the whole of `a` is not in `b`.

decDisjointVecs

Figure 7.5 gives the definition for the function `decDisjointVecs`, which is a decidable proof for the proposition that two `Vec`, `a` and `b` are disjoint, or $a \cap b = \emptyset$. The return type of the function is `Dec (Disjoint Nat n a m b)`, which uses the `Dec` type defined in Figure 6.16 on Page 67. If a `Yes` is returned, then it will contain a proof that `a` and `b` are disjoint, otherwise, a `No` will be returned which has a counter proof proving that `a` and `b` are not disjoint.

Before explaining the `decDisjointVec` there is three lemmas that need to be explained first, which are shown in Figure 7.4.

```
lemma_y_empty_disjoint_vec : [A : Type] -> [n : Nat]
                           -> (a : Vec A n) -> [b : Vec A Zero]
                           -> (b = NilV) -> DisjointVec A n a Zero b
lemma_y_empty_disjoint_vec = \ [A] [n] a [b] pf . case a of
  NilV       -> DNilV
  ConsV [m] x xs ->
    DConsV [m] x xs (subst xNotInNilV [A] [0] [x] by pf)
           (lemma_y_empty_disjoint_vec [A] [m] xs [b] pf)

lemma_x_in_b_vec : [A : Type] -> [x : A] -> [n : Nat]
                 -> [a : Vec A n] -> [m : Nat] -> [b : Vec A m]
                 -> [ElemVec A (Succ n) x (ConsV [n] x a)]
                 -> (ElemVec A m x b)
                 -> ((DisjointVec A (Succ n) (ConsV [n] x a) m b) -> Void)
lemma_x_in_b_vec = \ [A] [x] [n] [a] [m] [b] [xInA] xInB pf . case pf of
    DConsV [n1] y ys pfFory others -> pfFory xInB

lemma_xs_in_b_vec : [A : Type] -> [x : A] -> [n : Nat]
          -> [xs : Vec A n] -> [m : Nat] -> [b : Vec A m]
          -> ((DisjointVec A n xs m b) -> Void)
          -> ((DisjointVec A (Succ n) (ConsV [n] x xs) m b) -> Void)
lemma_xs_in_b_vec = \ [A] [x] [n] [xs] [m] [b] pf p. case p of
  DConsV [n1] i is pfFori others  -> pf others
```

Figure 7.4: Lemmas for `decDisjointVecs` function `pi-forall` implementation

The first lemma is `lemma_y_empty_disjoint_vec`, which provides a proof that the two `Vec` are disjoint when the second `Vec` is empty. It does this by pattern matching on the first `Vec`, `a`, which can have two possibilities. If `a` is empty, then `DNilV` can be used to prove they are disjoint. When `a` is non-empty, the `DConsV` constructor is used, but a proof that the head of `a` is not in the empty list is required and also a proof that the tail of `a` is disjoint from `b`. It can obtain the proof that the head of `a` is not in the empty vec `b` by using the lemma `xNotInNilV`, which is the `Vec` version of the lemma `xNotInNil` defined in Figure 6.20 on Page 71, and a proof with type `ElemVec A 0 x NilV` is obtained, which the proof `b = NilV` is substituted into to provide the required proof with type `ElemVec A 0 x b`. The other proof, that the tail of `a`, `xs`, is not in the empty vec, can be obtained from recursively calling the lemma. With both proofs, it's possible to construct using the `DConsV` constructor, which proves that two `Vec` are disjoint.

The second lemma is `lemma_x_in_b_vec` provides a proof that the `Vec c` is not disjoint with the `Vec b` when given a proof that the head of the `c`, named `x`, is an element of `b`. The first proof passed to the lemma has the type `ElemVec A (Succ n) x (ConsV [n] x a)`, which shows that `x` is not an element of `c`, where `c = ConsV [n] x a` and also proves that x is the head of `c`. The other proof passed in has the type `ElemVec A m x b` which is a proof that `x` is in `b`. To prove the lemma, a pattern match is done on the variable `pf` which has the type `(DisjointVec A (Succ n) (ConsV [n] x a) m b)`, and a contradiction is found by pattern matching on it. By the definition of `DNilV`, it cannot be used as a constructor for the type of `pf`, so only `DConsV` can be matched. The contradiction is found as it could derive a proof that `x` was not in `b`, but there is already a proof named `xInB` which proves otherwise.

The final lemma `lemma_xs_in_b_vec` is very similar to the previous lemma, but instead, it provides a proof that two vecs are not disjoint based on the tail of the first list not being disjoint. It will find

a contradiction as the variable, `others` is a proof with type `((DisjointVec A n xs m b)`, however, the passed in proof has the type `((DisjointVec A n xs m b) -> Void`, hence the contradiction.

```
1  decDisjointVecs : [n : Nat] -> (a : Vec Nat n) -> [m : Nat]
2                  -> (b : Vec Nat m) -> Dec (DisjointVec Nat n a m b)
3  decDisjointVecs = \ [n] a [m] b . case a of
4    NilV        -> Yes (DNilV)
5    ConsV [n1] x xs ->
6      case b of
7        NilV        ->
8          Yes (lemma_y_empty_disjoint_vec [Nat] [n] a [b] (Refl))
9        ConsV [m1] y ys ->
10         case (decDisjointVecs [n1] xs [m] b) of
11           Yes pf     ->
12             case (isElemVec [m] x b) of
13               Yes elemPf  ->
14                 No (lemma_x_in_b_vec [Nat] [x] [n1] [xs] [m]
15                                      [b] [(HereV [n1] xs)] elemPf)
16               No p    -> Yes (DConsV [n1] x xs (p) pf)
17           No p -> No (lemma_xs_in_b_vec [Nat] [x] [n1] [xs] [m] [b] p)
```

Figure 7.5: `decDisjointVecs` function `pi-forall` implementation

The function `decDisjointVecs`(Figure 7.5) is a proof by induction on the `Vec a`. There are two base cases and an inductive case.

The first base case is shown on Line 4, where `a` is the empty vec, `NilV`. In this case, it is possible to construct a `Yes` directly using the `DNilV` constructor through the definition of `DNilV`.

The second base case is on Line 8, where `a` is non-empty, and `b` is the empty vec. Using the `lemma_y_empty_disjoint_vec`, a proof that `a` and `b` can be provided so a `Yes` can be constructed.

The third case is the inductive case and starts on Line 10. This is when both `a` and `b` are non-empty. In this case, the first thing that occurs is checking if the tail of `a`, `xs` is disjoint with `b`. If it is disjoint, then the head of `a`, `x`, is checked for membership in `b`. Line 14 is when $x \in b$ and $xs \cap b = \emptyset$, so by the lemma `lemma_x_in_b_vec` a proof that the two `Vec` are disjoint is produced, hence a `No` is returned. Line 16 is the case when `p` is a proof that $x \notin b$ and `pf` is a proof that $xs \cap b = \emptyset$, hence it is possible to construct a proof that the two `Vec` are disjoint using the `DConsV` constructor, which can be used to construct a `Yes` object. Line 17 is when the `xs` and `b` are not disjoint, so using the lemma `lemma_xs_in_b_vec` will provide a proof that `a` and `b` are disjoint, so a `No` object can be returned.

While these functions and types for proving two `Vec` being disjoint, similar types and functions were developed for proving two `List` are disjoint. However, it is a very similar difference between `Elem` and `ElemVec` so the definition is not given and doesn't appear in any patterns.

UniqueVec

Just showing that two `Vec` are disjoint is not strong enough, as if `a` contains the same element twice, but that element is not in `b`, then they can still be disjoint, but when combined will not be a set. If there is a proof that both `a` and `b` are sets, where they have unique elements, then it would be strong enough to show that when taking the union of two disjoint `Vec`, the output would be a set where each element is unique in the `Vec`.

This was done through another type called UniqueVec, which is defined in Figure 7.6, and provides a proof that a Vec is unique through its elements, meaning that there is no repeated elements in the Vec. Three values index the type: the first is the type A, which is the type of the Vec v, the second is the number n which is the length of v and finally is the Vec v.

To prove that a Vec is unique, going through every element in order and ensuring it doesn't appear later in the Vec will prove the uniqueness. This is the basis of the two constructors. UNilV is for the case when the Vec v is empty, which is always unique by definition, hence there is a constructor for it. It does have the constraint that v is empty by ensuring it is equal to NilV. The second constructor, UConsV is to prove uniqueness when a is non-empty. It first proves that v is non-empty by having x and xs be the head and tail of v through the constraint on Line 4. It then has a proof that x does not appear in the tail of v as well as a proof that the tail,xs, is unique. Only when the proofs can be created can a proof for the uniqueness of v be constructed.

```
1 data UniqueVec (A : Type) (n : Nat) (v : Vec A n) : Type where
2   UNilV of [v = NilV] [n = 0]
3   UConsV of [m : Nat] (x : A) (xs : Vec A m) ((ElemVec A m x xs) -> Void)
4     (UniqueVec A m xs) [n = Succ m] [v = ConsV [m] x xs]
```

Figure 7.6: pi-forall UniqueVec datatype definition

decUniqueVec

Figure 7.8 shows the function, decUniqueVec, which decides if a given Vec, v, is unique. The function will produce an Dec type, meaning that either Yes containing a proof of uniqueness, or a No containing a counter proof is returned.

As before, the lemmas, defined in Figure 7.7, that appear within the function are explained before decUniqueVec.

The first lemma, lemma_x_in_xs_not_unique, will provide a proof that a Vec v is not unique when the head of v, x, is an element of the tail, xs. It does this by a proof of contradiction, where it assumes that v has a proof of uniqueness, p. From p, it can derive a proof, xNInXs, that x is not in xs. This is a contradiction as the lemma has been given a proof that x is in xs, so therefore, a proof that v is not unique has been formed.

The second lemma, lemma_xs_not_unique, will provide a proof that a Vec v is not unique when the tail of the list, xs is not unique. This is again a proof by contradiction where it assumes there is a proof p where v is unique. From pattern matching on p, a proof of uniqueness for xs, named others. This is a contradiction as a proof that xs is not unique was given to the lemma, so, therefore, a proof that v is not unique has been formed.

The function decUniqueVec(Figure 7.8) is a proof by induction on the Vec v, where there is a base case and an inductive case.

The base case is when v is NilV, the empty vec and is shown on Line 3. In this case, a proof of uniqueness can be directly formed using the UNilV constructor, as an empty vec is unique. This means that using the proof, a Yes can be constructed and returned.

The inductive case is when v is a non-empty list and starts on Line 5. The first thing done is to check if the head of v, x is an element of the tail, xs, using the function isElemVec, defined in Figure 6.29 on Page 76. This will either produce a Yes or a No. In the No case, a counter proof, xInXs

```
1  lemma_x_in_xs_not_unique : [A : Type] -> [n : Nat] -> [x : A]
2      -> [xs : Vec A n] -> (ElemVec A n x xs)
3      -> ((UniqueVec A (Succ n) (ConsV [n] x xs)) -> Void)
4  lemma_x_in_xs_not_unique = \ [A] [n] [x] [xs] pf p . case p of
5    UConsV [m] x xs xNInXs others -> xNInXs pf
6
7  lemma_xs_not_unique : [A : Type] -> [n : Nat] -> [x : A]
8      -> [xs : Vec A n] -> ((UniqueVec A n xs) -> Void)
9      -> ((UniqueVec A (Succ n) (ConsV [n] x xs)) -> Void)
10 lemma_xs_not_unique = \ [A] [n] [x] [xs] pf p . case p of
11   UConsV [m] x xs xNInXs others -> pf others
```

Figure 7.7: Lemmas for `decUniqueVec` function `pi-forall` implementation

is found, which proves $x \in xs$. Line 7 shows the counter proof being used to find a contradiction using the lemma, `lemma_x_in_xs_not_unique`, which produces a proof that `v` is not unique, which gets applied to the `No` constructor and returned. If a `Yes` were found instead, then there would be a proof that $x \notin xs$;, which is named `xNInXs`. From this, a recursive call is made on the `tail` of the list to decide if it is unique. Line 10 shows the case when the tail is unique, with a proof that `xs` is unique named `xsU`. A proof that the entire `Vec v` is unique can be constructed using the `UConsV` constructor and the proof that $x \notin xs$, xNInXs, and the proof that `xs` is unique,`xsU`, which gets applied to the `Yes` constructor and returned. If the tail is not unique, as in the case of a `No` on Line 11, `xsNU` will be a proof that that `xs` is not unique. Using lemma `lemma_xs_not_unique` and the proof `xsNU` a proof that `v` is not unique can be provided, which is applied to the `No` and returned.

```
1  decUniqueVec : [n : Nat] -> (v : Vec Nat n) -> Dec (UniqueVec Nat n v)
2  decUniqueVec = \ [n] v . case v of
3    NilV            -> Yes UNilV
4    ConsV [m] x xs ->
5      case (isElemVec [m] x xs) of
6        Yes xInXs    ->
7          No (lemma_x_in_xs_not_unique [Nat] [m] [x] [xs] xInXs)
8        No xNInXs ->
9          case (decUniqueVec [m] xs) of
10           Yes xsU ->  Yes (UConsV [m] x xs xNInXs xsU)
11           No xsNU -> No (lemma_xs_not_unique [Nat] [m] [x] [xs] xsNU)
```

Figure 7.8: `decUniqueVec` function `pi-forall` implementation

7.2.2 Proof Redundancy

After designing and implementing both proofs, it became clear that only the `UniqueVec` type was needed. This is because it is possible to prove two `Vec`, `a` and `b`, are disjoint by proving that the combined `Vec`, `c` where `c` is `append a b` for some `append` function.

This is why only `UniqueVec` is used in the patterns. Another added benefit to using it is that it inherently contains proofs that the head is not in the tail of the vec, anywhere in the vec. This means that if there is two `Vec` `a` and `b`, then if `c` is appending `a` and `b`, then inside the proof for uniqueness is a proof that the last element of `a` is not in the `b`. It will then have a proof that the second last element of `a`, is not in the vec `ConsV [n] (last a) b`. For the required use, proving that a set of new pid ids are not in the current `pidSet`, it works extremely well. After spawning the

first pid, the `pidSet` will expand to include the new pid, so a proof that the next pid is not in the `pidSet` would not prove anything about the new `pidSet` with the newly created pid in it. Using the `UniqueVec` gets around the next element would have a proof for the newly created `pidSet`.

7.2.3 Spawning Channels

A function was created to create a `Vec` of channels, and the type signature is shown in Figure 7.9; the implementation is too large for a figure, so it is described instead. The arguments to the function are as follows:

- `A` is the type that should be used to index into the `Channel` type during the channel creation.
- `m` is the size of the chidSet `Vec`.
- `chidSet` is the `Vec` that contains all the currently used channel ids.
- `mDash` is the size of the chids `Vec`.
- `chids` is the `Vec` that contains all the channel ids for the channels to be created.
- The next argument is a uniqueness proof for the `Vec` where `chids` and `chidsSet` are appended. This not only proves that `chids` and `chidSet` are unique but also proves that `chids` and `chidSet` are disjoint `Vec`.

The function will return within the `IO` monad as it is an `IO` action. The result of the `IO` action will within the `Maybe` type as it is possible the construction of a `Channel` might fail. The `Maybe` type is indexed by a dependant tuple with three elements: a `Vec` of the new channels, the new `chidSet`, and a proof that the new `chidSet` is correct.

There is a couple correctness checks within the function return type. The `Vec` of channels returned has the same size as the `chids` `Vec`, as that is how many channels should be created. The returned `newChidSet` has a constraint that it is the correct size, which is `nDash + n` and the final element of the tuple ensures that the new chid set is equal to the `chids` `Vec` appended to the passed in `chidSet`.

```
1 createChannels : [A : Type]
2   -> (m : Nat) -- The size of the chidSet
3   -> (chidSet : Vec Nat m) -- The chidSet
4   -> (mDash : Nat) -- The size of the chids provided
5   -> (chids : Vec Nat mDash) -- The set of chids provided
6   -> (UniqueVec Nat (plus mDash m) (append [Nat] [mDash] [m] chids
    chidSet)) -- Proof of uniquness for chids + chidSet
7   -> IO (Maybe ((Vec {id : Nat | Channel A id} mDash) * {newChidSet : Vec
      Nat (plus mDash m) | newChidSet = ((append [Nat] [mDash] [m] chids
    chidSet) : Vec Nat (plus mDash m))}))
```

Figure 7.9: `createChannels` function `pi-forall` type signature

The functions implementation is fairly straightforward in that it is a recursive definition, where when the `chids` is empty, it will spawn a channel with the `link` function, with the proof for `link` obtained from within the `UniqueVec`, named `chidPf`. From this, a recursive call is made where the `chids` tail and the tail's proof is passed in. This will continue until the base case is reached when the `chids` is `NilV`, upon which all the channels have been generated, so the function is finished.

7.2.4 Spawning Processes

Before considering spawning the processes, it is important to think about what is going to be run on them. For example, it is not as simple as running the function on each worker, as there is no communication on the channels there will be no input. To solve this, wrapper functions will be created: the `producerWrapper` will run on the emitter/producer process and the `workerWrapper` will run on the worker processes.

Worker Wrapper

The simplest wrapper function is for the worker wrapper, shown in Figure 7.10. It will take 6 parameters:

- `A`, which is the type used in the function, `f`, and the `Channel`.
- `receiveId`, which is the channel id for the receiving `Channel`.
- `sendId`, which is the channel id for the sending `Channel`.
- The receiving channel, with id `receiveId`.
- The sending channel, with id `sendId`.
- `f` is a function with a single input of type `A`, and returns an element of the same type.

Line 6 shows the wrapper function receiving the input along the channel. As `Nothing` is the termination signal, the received message, `mx` is pattern matched on. When the terminating signal has been received, on Line 9, the `Nothing` is sent along the sending channel to propagate the termination, and then the function ends. Line 13 is when an actual input was received along the channel, so it sends `Just (f x)`, so the function's output is sent along. It will then, on Line 16, make a recursive call so that the next input can be received.

```
1  workerWrapper : [A : Type] -> (receiveId : Nat) -> (sendId : Nat)
2              -> Channel A receiveId  -> Channel A sendId
3              -> (f : A -> A) ->  IO Unit
4  workerWrapper = \ [A] receiveId sendId receiveChannel sendChannel f.
5      bindEq [Maybe A] [Unit]
6      (receive [A] receiveId receiveChannel)
7      (\ mx .
8        case mx of
9            Nothing ->
10             bind [Unit] [Unit]
11             (send [A] sendId Nothing sendChannel)
12             (end)
13           Just x  ->
14             bind [Unit] [Unit]
15             (send [A] sendId (Just (f x)) sendChannel)
16             (workerWrapper [A] receiveId sendId receiveChannel
     sendChannel f)
17     )
```

Figure 7.10: `workerWrapper` function `pi-forall` definition

Producer Wrapper

The farm producer wrapper, shown in Figure 7.11, implements a round-robin distribution strategy along all the channels. The function has two `Vec` of channels which are initially going to be the same, and a list of input. The first `Vec` is called `allChs` and will always contain all the channels that connect the producer to each worker. The second `Vec` is called `chs` and will be traversed and reset to contain all the channels when it has reached the end. It returns an `IO Unit` as the function is an `IO` action which does not return a value.

The function starts by checking if the end of the input list has been reached; if it has, then the `propgateTermination` function is called, which sends `Nothing` along all the channels. If there is still input to send, it will check to see if another channel is in the `chs` vec. If there is not, then the `chs` needs to be reset to contain all the channels, so a recursive call starts that resets `chs`. If there is a channel to send along, the input will be sent along it, and then a recursive call will be made that progresses through the `chs` vec and the input list.

```
1  farmProducerWrapper : [A : Type] -> [n : Nat] -> [m : Nat]
2                      -> (Vec ({id : Nat | Channel A id}) n)
3                      -> (Vec ({id : Nat | Channel A id}) m)
4                      -> List A -> IO Unit
5  farmProducerWrapper = \ [A] [n] [m] allChs chs list .
6    case list of
7      Nil       -> propgateTermination [A] [n] allChs
8      Cons x xs -> case chs of
9        NilV ->
10         farmProducerWrapper [A] [n] [n] allChs allChs list
11       ConsV [mDash] ch restChs ->
12         let (chid, channel) = ch in
13         bind [Unit] [Unit] (send [A] chid (Just x) channel) (
     farmProducerWrapper [A] [n] [mDash] allChs restChs xs)
```

Figure 7.11: `farmProducerWrapper` function `pi-forall` definition

spawnWorkersFarm

To spawn the workers, the `spawnWorkersFarm` function is used, which is shown in Figure 7.12, and the arguments are described in the figure. Again the key point is that the number of `pids` is the same as the number of channels passed in. This is because each worker will need their own channel linking to the producer. There is also another channel passed in, the consumer channel, which every worker will send their output from. There is also the uniqueness proof to ensure the correctness of the `pids Vec` passed in.

The return type of the function is very similar to the function `createChannels`, as they both return a value within the `IO` monad that is a `Maybe` which is indexed by a dependant tuple. However, the difference lies in the dependent tuple. In `spawnWorkersFarm` it is just a dependant pair of the new pid set, `newPidSet` and a proof as to the correctness of `newPidSet` which proves that it is `pids` appended to the `pidSet`.

The function implementation is again too long to show in a figure, so will be explained instead. It first checks if there is a process id in the `pids Vec`. If there is a process id it will then use the `spawnAndRun` function to spawn a worker with the given process id. The function run on the worker will be the `workerWrapper` where the receiving channel is taken from the set of channels passed

```
spawnWorkersFarm : [A : Type]
  -> (n : Nat) -- The size of the pidSet
  -> (pidSet : Vec Nat n) -- The pidSet
  -> (nDash : Nat) -- The size of the pids provided
  -> (pids : Vec Nat nDash) -- The set of pids provided
  -> (UniqueVec Nat (plus nDash n) (append [Nat] [nDash] [n] pids pidSet)
   ) -- Proof that the pids are valid and unique from pidSet
  -> (Vec {id : Nat | Channel A id} nDash) -- The set of channels
  -> {id : Nat | Channel A id} -- The consumer Channel
  -> (f : A -> A) -- The function
  -> IO (Maybe ({newPidSet : Vec Nat (plus nDash n) | newPidSet = ((
   append [Nat] [nDash] [n] pids pidSet) : Vec Nat (plus nDash n))}))
```

Figure 7.12: `spawnWorkersFarm` function `pi-forall` type signature

in, and the sending channel is the consumer channel. The function `f` passed in is also passed to `workerWrapper`. The proof required for this is taken from the `UniqueVec` much like in `createChannel`. After spawning, a recursive call is made to generate the next worker. This will continue until the base case when `pids` is empty and the function will terminate.

7.2.5 createFarm function

The function `createFarm`, shown in Figure 7.13, is used to create and run a farm pattern and all the arguments are described in the figure. The implementation is too long to show, so only the type signature is given.

The first key point from the signature is that the size of `pids` and `chids` is proven to be correct as it is one more than the number of workers, `numWorkers`, through the arguments, `nDash = (Succ numWorkers)` and `mDash = (Succ numWorkers)`. The two `UniqueVec` arguments prove that process ids and channel ids in `pids` and `chids` are valid and not already used. There is also a proof that the number of workers is not zero, meaning there is always a worker.

As the return type is within the `IO` monad, the function is an `IO` action. It also returns a `Maybe` as a failure could occur during the build. The `Maybe` type is indexed by a dependant tuple with 4 elements. The first is the output list that results from running the pattern. The second is `Farm` which represents the farm and is indexed by the type `A` and the number of workers, `numWorkers`. The last two elements are each a dependent pair which return the new id list and a proof that it is correct; the first pair is for the `newPidSet` and the second is for the `newChidSet`. The proofs of correctness are the same ones returned from the functions `createChannels` and `spawnWorkersFarm`.

The function's implementation starts by calling the `createChannels` function to create all the channels required for the pattern. It is then split into the head and tail, where the head of the list is used as the consumer channel, and the tail of the list is the producer channel. Next, the producer process is spawned, using the head of the `pids Vec`, and the `farmProducerWrapper` is run on the process. The tail of the `pids` is used in the `spawnWorkersFarm` function so that all the workers are generated. Finally, the function will run the function called `farmConsumerWrapper` which collects all the output from the channel. One thing to note here is that the consumer channel will be getting a termination `Nothing` from every worker, so it will need to count the number of terminations so that it stops waiting on input only when all the workers have terminated. Only when all the processes are synchronised does the `farmConsumerWrapper` return the output list, upon which the `createFarm` function will terminate and return the required return values.

```
createFarm : [A : Type] -- The type for the function
  -> (n : Nat) -- The size of the pidSet
  -> (pidSet : Vec Nat n) -- The pidSet
  -> (nDash : Nat) -- The size of the pids provided
  -> (pids : Vec Nat nDash) -- The set of pids provided
  -> (UniqueVec Nat (plus nDash n) (append [Nat] [nDash] [n] pids pidSet)
   ) -- Proof that the pids are valid and unique from pidSet
  -> (m : Nat) -- The size of the chidSet
  -> (chidSet : Vec Nat m) -- The chidSet
  -> (mDash : Nat) -- The size of the chids provided
  -> (chids : Vec Nat mDash) -- The set of chids provided
  -> (UniqueVec Nat (plus mDash m) (append [Nat] [mDash] [m] chids
   chidSet)) -- Proof that the chids are valid and unique from chisSet
  -> (numWorkers : Nat) -- Number of workers
  -> ((numWorkers = Zero) -> Void) -- Farm must have at least one worker
  -> (nDash = (Succ numWorkers)) -- Proof that the number of pids
   provided is correct
  -> (mDash = (Succ numWorkers)) -- Proof that the number of chids
   provided is correct
  -> (f : A -> A) -- The function to farm
  -> (List A) -- Input
  -> IO (Maybe ((List A) * (Farm A numWorkers) * {newPidSet : Vec Nat (
   plus nDash n) | newPidSet = ((append [Nat] [nDash] [n] pids pidSet) :
   Vec Nat (plus nDash n))} * {newChidSet : Vec Nat (plus mDash m) |
   newChidSet = ((append [Nat] [mDash] [m] chids chidSet) : Vec Nat (plus
    mDash m))}))
```

Figure 7.13: `createFarm` function `pi-forall` type signature

7.3 Pipeline

The other pattern that was implemented is the pipeline pattern, and there is only one change from the skeleton diagram, shown in Figure 7.14, which is that there is a producer which will take the inputs and pass them into the channel ready for the first stage. This is done on a new process so the pipeline can begin executing the actual stages, even though not all the inputs are in the first channel.

This means that for a pipeline with n stages, there will need to be $n+1$ processes; n for the stages and one for the producer. It will also have $n+1$ channels, n being passed out from each stage and one from the producer.

The type `Pipe` is shown in Figure 7.15 and represents the pipeline pattern. It is indexed by the type `A`, used to represent the type of messages amongst stages, and a number `n` representing the number of stages in the pipeline. It has a single constructor, `MkPipe`, which has 3 arguments. The first is `m` used to represent the number of stages in the pipeline, `funcs` is a `Vec` that contains the function for each stage, and finally `chs`, which is a `Vec` containing all of the channels that are associated with the pipeline. There is also a constraint for the constructor, which ensures that `m = n` so that the number of stages is the same. Like in the `Farm` type, there are a couple of design points that ensure the correctness of the pipeline type:

- The functions are all from type `A` to `A`, which is part of the limitation of the system.

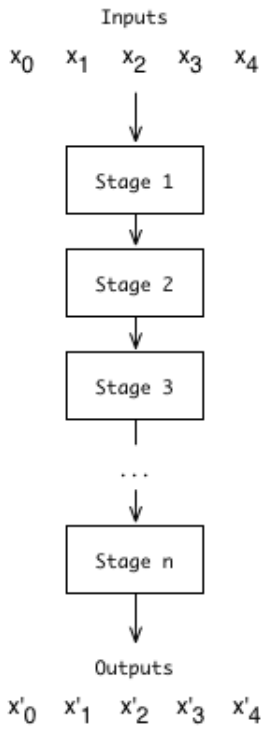


Figure 7.14: Pipeline Pattern[18]

- The number of functions is constrained to b `m`, which is the number of stages. This ensures that there is a function for each stage.
- The number of channels is constrained to be `Succ n`, which ensures that the correct number of channels are associated with the pipeline.
- The channel type is also indexed by `A`, which ensures that the channels will be of the correct type.

```
1 data Pipe (A : Type) (n : Nat) : Type where
2   MkPipe of (m : Nat) (funcs : Vec (A -> A) m)
3             (chs : Vec {id : Nat | Channel A id} (Succ n)) [m = n]
```

Figure 7.15: `Pipe` datatype `pi-forall` implementation

Much like with the `Farm`, there are no constraints that the channels or processes used for the associated pipeline are not used elsewhere. This is again done within the function that creates the pipeline.

7.3.1 Proofs

The same proofs for the `pidSet` and `chidSet` appear in the functions to create the pipeline, as that is pattern independent and purely for creating the channels and processes.

7.3.2 Spawning Channels

There is also no difference between spawning the channels for the farm pattern than in the skeletons pattern, so the same function, `createChannels` defined in Section 7.2.3 on Page 86, is used.

7.3.3 Spawning Processes

Stage Wrapper

The stage wrapper used to run on the stages is the `workerWrapper`, defined in Section 7.2.4 on Page 87, that is used in the farm pattern because there is no difference in behaviour between a worker and a stage so the same function can be used.

Producer Wrapper

The producer exhibits different behaviour in the pipeline compared to the farm; the producer will only send along a single channel. This means that the function is extremely simple and it is shown in Figure 7.16. It will recurse through the given input list and send along the given channel until the end of the list is reached, upon which it will send the termination `Nothing` and then finish.

```
1  producerWrapper : [A : Type] -> (chid : Nat) -> Channel A chid
2                  -> List A -> IO Unit
3  producerWrapper = \ [A] id ch list .
4      case list of
5          Nil          ->
6              bind [Unit] [Unit] (send [A] id Nothing ch) (end)
7          Cons x xs    ->
8              bind [Unit] [Unit]
9              (send [A] id (Just x) ch)
10             (producerWrapper [A] id ch xs)
```

Figure 7.16: `producerWrapper` function `pi-forall` definition

spawnWorkersPipe

The `spawnWorkersFarm` function will not work for the pipeline, so a new function, `spawnWorkersPipe` was created and it is shown in Figure 7.17. The type signature only has two differences: firstly, there is no longer a consumer channel passed in; instead, the channel represents the incoming channel to the next stage created. Secondly, there is not just a single function passed in, but a `Vec`, which is also the same length as the size of the `Vec` `pids`. Everything else about the type signature remains the same, so all the proof and constraints will still hold.

The difference in implementation is that when spawning the process, the receiving channel is the `in` channel argument, and the sending channel is obtained from the `Vec` of channels. This means that when recursing through the vec of `pids`, the `in` channel will become the channel used as the sending channel. By constantly updating the `in` channel, the stages will be connected correctly.

7.3.4 createPipe

The `createPipe` function, shown in Figure 7.18, is used to create and run the pipeline pattern. All of the arguments are described in the figure and only the type signature is given as, again, the function implementation is too large to display. It is also very similar to the `createFarm` function

```
spawnWorkersPipe : [A : Type]
    -> (n : Nat) -- The size of the pidSet
    -> (pidSet : Vec Nat n) -- The pidSet
    -> (nDash : Nat) -- The size of the pids provided
    -> (pids : Vec Nat nDash) -- The set of pids provided
    -> (UniqueVec Nat (plus nDash n) (append [Nat] [nDash] [n] pids
   pidSet))
    -> (Vec {id : Nat | Channel A id} nDash) -- The set of channels
    -> {in : Nat | Channel A id} -- The in Channel
    -> (fs : Vec (A -> A) nDash) -- The function
    -> IO (Maybe ({newPidSet : Vec Nat (plus nDash n) | newPidSet = ((
   append [Nat] [nDash] [n] pids pidSet) : Vec Nat (plus nDash n))}))
```

Figure 7.17: `spawnWorkersPipe` function `pi-forall` type signature

described in Section 7.2.5 on Page 89, with the only differences in the type signature being the `numWorkers` variable changing to `numStages` and a `Vec` of functions is passed in instead of just one, with the constraint that its length is `numStages`. The return type is also slightly different in that a `Pipe` is returned instead of the `Farm`, otherwise the return type still returns the output and the dependant pairs for the `newPidSet` and `newChidSet`.

Due to everything else being the same, all the other constraints gained through the type signature on `createFarm` also occur in `createPipe`.

The implementation is also slightly changed. Firstly, the function will use the `createChannels` function to create all the channels. Next, it will split that `Vec` of created channels to obtain the channel for the producer to the first stage, with all the remaining channels, which is required later for the function to spawn all stages when `spawnWorkersPipe` is called. After the split, the `pids` are also split, with the first being used to spawn the producer using the `spawnAndRun` function with the `producerWrapper` function being run on the process. Then the stages will be spawned using the function `spawnWorkersPipe` and finally, the output will collected from the final channel using the `consumerWrapper` function as in the `createFarm`. The final channel is found by retrieving the last element in the channels `Vec`. After collating the output, the other elements for the return tuple are created, such as the `Pipeline` and the dependant pairs that show the correctness of the `pidSet` and `chidSet`.

```
createPipe : [A : Type] -- The type for the functions
  -> (n : Nat) -- The size of the pidSet
  -> (pidSet : Vec Nat n) -- The pidSet
  -> (nDash : Nat) -- The size of the pids provided
  -> (pids : Vec Nat nDash) -- The set of pids provided
  -> (UniqueVec Nat (plus nDash n) (append [Nat] [nDash] [n] pids pidSet)
   ) -- Proof that the pids are valid and unique from pidSet
  -> (m : Nat) -- The size of the chidSet
  -> (chidSet : Vec Nat m) -- The chidSet
  -> (mDash : Nat) -- The size of the chids provided
  -> (chids : Vec Nat mDash) -- The set of chids provided
  -> (UniqueVec Nat (plus mDash m) (append [Nat] [mDash] [m] chids
   chidSet)) -- Proof that the chids are valid and unique from chidSet
  -> (numStages : Nat) -- Number of stages
  -> ((numStages = Zero) -> Void) -- Pipe must have at least one stage
  -> (nDash = (Succ numStages)) -- Proof that the number of pids provided
     is correct
  -> (mDash = (Succ numStages)) -- Proof that the number of chids
   provided is correct
  -> (fs : Vec (A -> A) numStages) -- The function to farm
  -> (List A) -- Input
  -> IO (Maybe ((List A) * (Pipe A numStages) * {newPidSet : Vec Nat (
   plus nDash n) | newPidSet = ((append [Nat] [nDash] [n] pids pidSet) :
   Vec Nat (plus nDash n))} * {newChidSet : Vec Nat (plus mDash m) |
   newChidSet = ((append [Nat] [mDash] [m] chids chidSet) : Vec Nat (plus
    mDash m))}))
```

Figure 7.18: `createPipe` function `pi-forall` type signature

7.4 Limitations

Although a farm and pipeline pattern have been designed and implemented in `pi-forall`, it is not without limitations.

The first major limitation is the types used as functions in the patterns. As discussed in the overall design section, Section 7.1, the same type of function must be used in a pattern, and it must be from any type `A` to the same type `A`. Without allowing relevant types, or a complete redesign, there was no way to allow any function to be used in the patterns.

Another limitation is that there was no time to extend the design to allow for pattern composition. With the current design, a farm cannot be embedded in a pipeline stage, nor may a pipeline be farmed.

There is also the issue that a user could use an empty `pidSet` or `chidSet` so that they could always prove that the given pids are unique, which could potentially ruin the patterns.

7.5 Future Work

Many areas could be explored that build on the work in this dissertation relating to the patterns.

The first would be using the current design and extending it with new patterns. This seems promising, as from looking at the functions and types used to create and run the farm and pipeline

pattern, they are extremely similar, with only a few changes needed for the function to form the other pattern. This would suggest that creating a new pattern would be relatively straightforward. The only downside to this approach is that it would still have to have the same limitations as the current patterns in that they could not be composed or must have the same types.

One area that almost got started in this dissertation but failed due to lack of time was re-designing the patterns so they could be composed. In the new design, there were also strong guarantees as to the correctness of the patterns. For example, if there were a type named `PatternType`, as shown in Figure 7.19, that is used as an index to another type, `Pattern` that is defined in Figure 7.20, then the structure of the pattern can be explicit inside the constructor of the type. The `Pattern` type is just an illustration where there would be lots more constraints and proofs contained in each constructor.

```
1 data PatternType : Type where
2     Pipeline
3     PipelineInner
4     Worker
5     Farm
```

Figure 7.19: `pi-forall` `PatternType` datatype definition

```
1 data Pattern (inType : Type) (outType : Type) (type : PatternType) : Type
      where
2     WorkerPat of ... [type = Worker]
3     PipelineInnerPat of ... [type = PipelineInner]
4     PipelinePat of ... [type = Pipeline]
5     FarmPat of ... [type = Farm]
```

Figure 7.20: `pi-forall` `Pattern` datatype definition

Using this design, a `Pattern` would be constructed one stage at a time and would be made correct through the `PatternType`. For example, a producer could be used to build an incomplete pattern to produce a `PipelineInner`. Adding stages could prove that it is a valid addition by checking what type of pattern a new stage is being added to. For example, the function to add a new worker stage to a pipeline could have the type `f : (Pattern ... PipelineInner) -> (Pattern ... Worker) -> ... -> (Pattern ... PIpelineInner)`, or a pipeline could be completed by adding a consumer with the function `g : (Pattern ... PipelineInner) -> ... -> (Pattern ... Pipeline)`. This would mean that a proof could be written about the pattern's structure as it is being built or after. This implementation managed to get fairly far, however, required types to appear relevant in functions.

Another area that could be promising is to recreate these structures in a much more popular language that has more powerful features, such as Idris. This would allow the types to appear relevant so that the types would no longer be an issue. The better design described above could also be implemented.

7.6 Summary

In this chapter, the design for the farm and skeleton pattern has been outlined. The `UniqueVec` type has been described and the function that produces proofs for it. The functions that construct the

patterns and how they ensure correctness has been shown. Finally, the design and implementation limitations have been described, and areas for future work have been outlined.

Chapter 8

Evaluation

This chapter will evaluate the parallel patterns implemented by comparing them against other pattern implementations developed in other languages. It will also critically analyse the work produced against the original requirements.

8.1 Experiments

To evaluate the `pi-forall` parallel pattern implementations, they would be compared with three other parallel pattern implementations written in other programming languages. The speedup would be what is compared, not the actual running time. This is because it is not a fair comparison, as that will evaluate the programming languages' performance rather than the parallel patterns.

The first is GrPPI[5], a C++ parallel pattern interface that can target different C++ parallel programming frameworks, such as ISO C++ Threads, OpenMP[23] and Intel TBB[24]. In the experiment, just the ISO C++ Threads backend is used.

The second is Skel[4], which is an Erlang algorithmic skeleton library.

The third is Haskel Eden[22], which extends the GHC compiler with concurrency primitives and has libraries that use the primitives to implement parallel patterns.

Each experiment would also be run sequentially, without parallelism, so a benchmark could be obtained to calculate speedup.

8.1.1 Machines

There were three machines that the experiments could run on, each with different specifications.

The first machine, PC 1, is pc8-023-l from the Jack Cole Laboratory and has the 11th Gen Intel(R) Core(TM) i5-11400 @ 2.60GHz CPU. The CPU has 6 cores but is hyperthreaded.

The second machine, PC 2, is a home computer with the AMD Ryzen 5 5600X 6-Core Processor @ 3.70GHz CPU that also has 6 Cores and the capability for simultaneous multi-threading the same as being hyperthreaded for non-intel CPUs.

The third machine, PC 3, is the Corryvreckan shared memory machine and has the Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz CPU with 28 cores and is also hyperthreaded.

8.1.2 Issues

There were a couple of issues with the experiments. The two major issues were the third parallel pattern implementation, Haskell Eden, had to be removed. This is because the Eden compiler could not be compiled. It required an old version of GHC to bootstrap it, but the required version could not be compiled either. This was extremely unfortunate, as Haskell Eden is the project that most closely resembles this project.

The second major issue was that PC 3 could not be used. This is because no `pi-forall` program can be run on it due to the limitations of `pi-forall`, which is that at least C++14 is needed when compiling, but PC 3 only had up to C++11. This was another shame, as this was a machine with the most cores, so when experimenting with a farm, it could be tested on many more than just 6 cores.

The final major issue was with the experiment design. Due to the parallel pattern limitation that the function must be of type `A -> A`, no meaningful experiments could be done, and only simple experiments could be implemented.

A minor issue was that GrPPI would not work on PC 2. The reason for this was not found, but it meant that all experiments on PC 2 would only be tested with `pi-forall` and Erlang.

8.1.3 Experiment 1

The first experiment was to farm calculating the 42nd Fibonacci number. By having an embarrassingly parallel problem, such as calculating `fib 42`, it is expected that the speedup will scale linearly with the number of cores being used.

Each language had the same implementation of a `fib` a function that would be used in each farm pattern.

8.1.4 Experiment 2

The second experiment was again for the farm pattern, but instead of calculating a Fibonacci number, a 100 by 100 was raised to the 16th power. Again this would result in the experiment being embarrassingly parallel; however, it would test if larger objects affect speedup when passed through the channels.

This experiment meant that each language needed to implement matrix multiplication so it could be farmed.

8.1.5 Experiment 3

The third experiment was for the pipeline pattern and would again use matrices. All the matrices would be 200 by 200; the first stage of the pipeline would multiply the input by a constant 200x200 matrix, and the second stage would then square the matrix.

8.1.6 Experiment 4

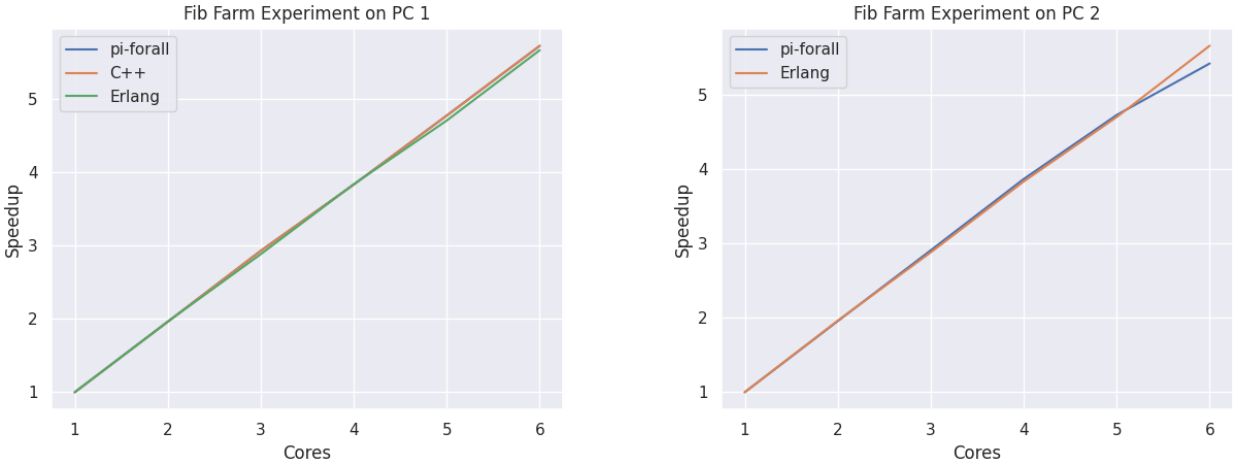
There was a plan for a fourth experiment, however, this was scrapped due to the limitations of the patterns function type again. It was to try and farm matrix multiplication, however, it would farm calculating a row of the resultant matrix, so each row is calculated in parallel. This would have been a more complex example of using the farm pattern.

8.1.7 Generating Results

To generate the results, each experiment was run on each machine (PC 1, PC 2), for every implementation for the machine(`pi-forall` C++, Erlang), and every number of cores (1-6). The experiments would then be repeated 10 times to ensure reliable results.

8.2 Analysis

8.2.1 Experiment 1



(a) PC 1

(b) PC 2

Figure 8.1: Experiment 1 Results

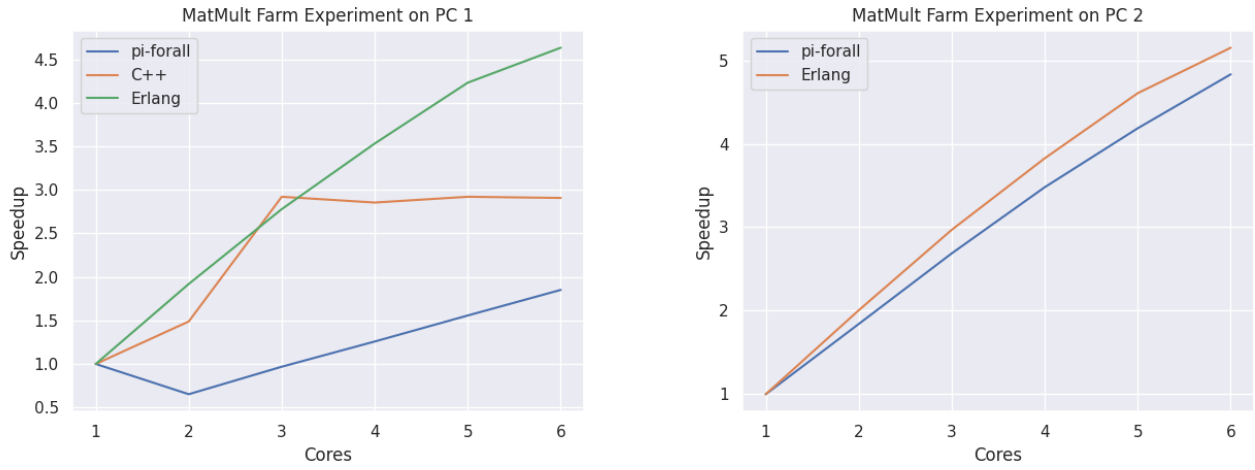
The results from experiment 1 show that the speedup on both machines, PC 1 and PC 2, scales linearly with the number of cores. This makes sense as the experiment is an embarrassingly parallel problem. 100% of the program is parallisable, so from Amdahl's Law the theoretical maximum speedup is equal to the number of cores, which is achieved. This means that the maximum speedup for this problem has been achieved.

8.2.2 Experiment 2

Experiment 2 is like experiment 1, as 100% of the problem is parallisable. This again means that the theoretical maximum speedup equals the number of cores. This is achieved on PC 2, as shown in Figure 8.2b for both the `pi-forall` and Erlang patterns.

Some bizarre results from the machine PC 1 are shown in 8.2a. The Erlang results are similar to PC 2, but the `pi-forall` is completely different. When there are two cores, there is even a slowdown instead of a speedup, and even when it uses 6 cores, a speedup of 2 is barely achieved. As PC 2 results support the expected result that speedup scales linearly with the cores, it could suggest that the reason for the poor performance of the `pi-forall` patterns in experiment 2 is due to external reasons. As PC 1 is a lab machine, other users could have been using the machine during the experiment, which would affect the run time of evaluating the pattern.

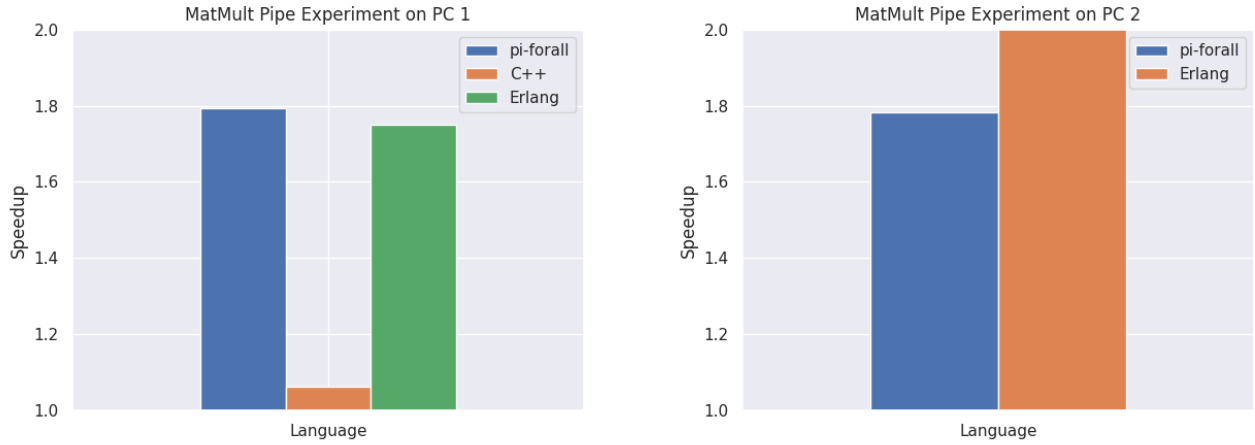
The C++ results can be explained much easier. The reason is that the experiment's granularity is too fine. This means that C++ can evaluate the functions fast enough that the overhead introduced by the patterns, such as thread creation and communication, is significant and can introduce a slowdown, hence the reason it is not scaling linearly with the cores.



(a) PC 1 (b) PC 2

Figure 8.2: Experiment 2 Results

8.2.3 Experiment 3



(a) PC 1 (b) PC 2

Figure 8.3: Experiment 3 Results

In experiment 3, as there are 2 stages in the pipeline, there is a maximum speed up of 2, because the stages are regular. This is almost reached by `pi-forall` and Erlang on both PC 1 and PC 2, so it means that the cost of using the patterns, most likely to be the communication due to the large size of the 200x200 matrix, is causing some slowdown effect.

The C++ experiment was completed almost instantly, which suggested that the granularity was again too fine, as the speedup gained from running in parallel is all lost due to the slowdown caused by introducing the patterns. There is almost no difference between the pipeline and running it sequentially.

8.3 Critical Appraisal

The primary requirements have also changed since the DOER, which can be found in Appendix B, with one of the secondary objectives, to create parallel patterns, becoming a primary objective as

it was a major focus of the dissertation.

When comparing the work produced against the primary requirements, all have been met to varying degrees of success.

8.3.1 Translation

The first objective was adding a compilation backend from the small dependently typed language, `pi-forall`[3], to create a C++ executable. This will then allow `pi-forall` programs to execute.

This objective was almost fully met in Chapter 5 with only relevant types not being able to be translated into C++. Even though it cannot translate every program, the fact that it can translate such a large subset of `pi-forall` programs as it can show that it is a large success. It also translates the `pi-forall` code so that it is straightforward to inject manually written C++ into generated code.

8.3.2 Concurrency Primitives

The second objective was to extend the `pi-forall` language and backend with concurrency primitives, allowing parallel programs to be written in the language.

This objective was fully met in Chapter 6. Using the primitives, it is possible to create a parallel program, as there are functions which can create a `pi-forall` process and run functions on it, as well as communication methods through channels. The primitives also use dependant types to help ensure the correctness of the concurrency.

8.3.3 Parallel Patterns

This primary objective was to model algorithmic skeletons using the dependent types in the language and explore the use of proofs within the patterns.

The objective was met in Chapter 7 as the farm, and pipeline patterns are implemented. There is also much investigation into proofs that can be used to ensure correctness, and in the future work section, an outline of a more complex type that more correctly models the structure of the patterns is described.

8.3.4 Evaluation

There was an objective to evaluate the parallel runtime system on a set of examples. It would involve creating a suite of parallel programs as a benchmark to compare against other, similar, parallel programs written in another language.

This objective was only partially met in this chapter. Although there were some parallel programs written, it was not able to be evaluated on PC 3, which is the large 28-core machine, nor was I able to get the Haskell Eden implementation to work so it could be compared. It was the one that most of the comparison would come from, as they both extend a compiler with concurrency primitives and then use those primitives to build a patterns library. `pi-forall` and Haskell are also very similar languages so it would have been a great comparison if it worked.

8.3.5 Secondary Objectives

As one of the secondary objectives, to create parallel patterns, was completed, even though it is now a primary, means that from the DOER, one of the secondary objectives was met.

Chapter 9

Conclusion

9.1 Project Summary

This project explored how dependant types can be used to model algorithmic skeletons, through the small, dependently typed programming language, `pi-forall`. The language was extended to have a C++ compilation backend, and then also with concurrency primitives that allowed parallel programs to be created. A farm and pipeline skeleton was modelled using the dependent types and these patterns were compared with other parallel pattern implementations so they could be evaluated.

9.2 Project Success

Overall the project can be considered a success. Creating a translator that can translate almost all `pi-forall` code into C++ code can be considered a major achievement. This is especially true when considering that there was not much documentation for the inner workings of `pi-forall`.

Modelling the parallel patterns using dependant types can also be deemed a success because, while it might have some limitations, it can still prove the correctness during its construction. Many proofs were created and refined during the project, and a second iteration, which has even more proof of correctness, is described. The work produced here has provided a good starting point to continue exploring parallel patterns being modelled with dependant types.

9.3 Project Drawbacks

Although the project is a success, there is a major drawback. The major limitation for the translation, any time the type of types, `Type`, appears relevant, it becomes impossible to translate the program. This drawback had consequences in the design of the parallel patterns, which caused a major drawback in their design, by limiting the function types to have to be over the same type `A -> A`.

Another drawback is that the patterns designed could not be composed together in any way, which was unfortunately due to running out of time.

9.4 Future Work

The future work has already been talked about a little in previous chapters, but the main areas to work on will be outlined again here.

The first area that can be worked on in the future is fully implementing the design described for the parallel patterns that would allow them to be composed. The new pattern design could also be implemented in another language, such as Idris, which would allow the limitation on the function types in the current pattern design to be removed.

This could be combined with extending Idris with a new C++ backend that would work the same way as the `pi-forall` backend, so it would be able to work with the concurrency primitives and parallel patterns that were designed. This would also have the benefit of having a richer type system and also nicer developer experience when compared to using `pi-forall`.

Another area would be to carry on with the current design and implement new patterns that are more complex than the farm and pipeline pattern; those are the only two patterns explored in this work. This could also work with extending the concurrency primitives with a new monad. The new monad would replace the `IO` monad in the concurrency use cases so it could make the `pidSet` and`chidSet`, implicit instead of explicit, as well as make generating unique ids implicit.

Finally, there is also work that can be done in the translator, such as removing the limitations by being able to translate any `pi-forall` program. As stated in Section 5.11, the translator has lots of technical debt, so it might even be worth starting again with a fresh design, that learns from what went well in this design, and also new solutions are made to deal with the current problems. If this is done, then it would be possible to use `pi-forall` to fix the limitations created by the translator in the concurrency primitives and the parallel patterns.

Appendix A

Ethics

There were no ethical considerations for the project, and the ethics form is shown below.

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

☐ **Staff Project**
☐ **Postgraduate Project**
☒ **Undergraduate Project**

Title of project

A Dependently Typed Parallel Runtime System

Name of researcher(s)

Findlay Sloan

Name of supervisor (for student research)

Christopher Brown

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☒ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

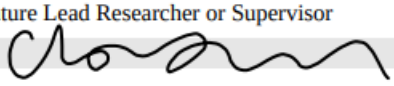
Print Name

Findlay Sloan

Date

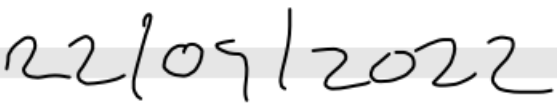
22/09/2022

Signature Lead Researcher or Supervisor



Print Name

Date



This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

Figure A.1: Signed Ethics Form

Appendix B

DOER

Findlay Sloan Supervisor: Christopher Brown

DOER

Description:

The project title is "A Dependently Typed Parallel Runtime System".

Dependently typed languages allow a programmer to create more expressive types as the types can depend on values. These dependant types can help aid a programmer's confidence in the safety of their program. Most work with dependently typed languages has explored using them as a theory solver or proof checker. This project will aim to explore improving the safety of parallel programs, as these can be difficult to reason about and debug.

The project will involve building on pi-forall, a small, dependently typed functional language. Currently, a pi-forall program can be type-checked but cannot be executed. The first area to expand pi-forall would be translating to another language so a program can run. Pi-forall also has no parallel mechanics in the language, so extensions would need to be made to write a parallel program. A parallel runtime system would also need to be written so that the program can be executed in parallel.

Objectives:

Primary:

1. Develop a translator for pi-forall to C so it can be executed. After this, non-parallel dependently typed programs can be written and run in pi-forall.
2. Extend pi-forall with parallel mechanics that take advantage of the dependent types. After this, parallel programs should be able to be written and type-checked in pi-forall.
3. Implement a parallel runtime system in C so parallel pi-forall programs can be executed in parallel.
4. Evaluate the runtime system on a set of examples. This step will involve creating a suite of parallel programs as a benchmark to compare against other, similar, parallel programs written in another language.

Secondary:

1. Provide examples of potential extensions to other, more complete, dependently typed languages to introduce parallelism.
2. Add more parallel mechanics to pi-forall, such as parallel patterns or skeletons.
3. Create a translation from pi-forall to Erlang.

Tertiary:

1. Implement the potential extensions to the other, more complete, dependently typed languages, like Idris, so that parallelism can be introduced.

Ethics:

There are no ethical considerations for this project.

Resources:

This project can be completed using standard school lab equipment.

Figure B.1: DOER

Appendix C

Running Instructions

To run the compiler, navigate to the folder that has the *pi* bash script. This script can be used as follows:

```
./pi <path to pi program>
```

Figure C.1: pi compiler usage

This will compile the program and produce a file names out.cpp, which can then be compiled with the g++ compiler using the function

```
g++ --std=c++14 -Wno-return-type -O3 path/to/out.cpp
```

Figure C.2: g++ compiler usage

This will then produce an executable, `a.out`, but another name could be specified with the `-o` argument.

Bibliography

[1] Simon Thompson. *Type theory and functional programming*. Addison Wesley, 1991.

[2] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[3] Stephanie Weirich. "Implementing Dependent Types in pi-forall". In: *arXiv preprint arXiv:2207.02129* (2022).

[4] Archibald Elliott et al. "Skel: A streaming process-based skeleton library for Erlang". In: *24th Symposium on Implementation and Application of Functional Languages, IFL*. 2012.

[5] David del Rio Astorga et al. "A generic parallel pattern interface for stream and data processing". In: *Concurrency and Computation: Practice and Experience* 29.24 (2017). e4175 cpe.4175, e4175. DOI: https://doi.org/10.1002/cpe.4175. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4175. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4175.

[6] R.R. Schaller. "Moore's law: past, present and future". In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665.

[7] J.M. Rabaey, A.P. Chandrakasan, and B. Nikolić. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall electronics and VLSI series. Pearson Education, 2003. ISBN: 9780131207646. URL: https://books.google.co.uk/books?id=%5C_7daAAAAYAAJ.

[8] Vladimir Janjic, Christopher Brown, and Adam D Barwell. "Restoration of Legacy Parallelism: Transforming Pthreads into Farm and Pipeline Patterns". In: *International Journal of Parallel Programming* 49.6 (2021), pp. 886–910.

[9] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[10] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, 1984.

[11] William A Howard. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[12] Paul Bernays. "Alonzo Church. An unsolvable problem of elementary number theory. American journal of mathematics, vol. 58 (1936), pp. 345–363." In: *The Journal of Symbolic Logic* 1.2 (1936), pp. 73–74.

[13] Edwin Brady. *Type-driven development with Idris*. Simon and Schuster, 2017.

[14] Paul Hudak et al. "Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language Version 1.2". In: *SIGPLAN Not.* 27.5 (May 1992), pp. 1–164. ISSN: 0362-1340. DOI: 10.1145/130697.130699. URL: https://doi.org/10.1145/130697.130699.

[15] Philip Wadler. "Monads for functional programming". In: *International School on Advanced Functional Programming*. Springer. 1995, pp. 24–52.

[16] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Éditeur inconnu, 1972.

[17] Heiko Eißfeldt. "POSIX: A Developer's View of Standards". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '97. Anaheim, California: USENIX Association, 1997, p. 24.

[18] Chris Brown. *Skel Tutorial*. URL: https://chrisb.host.cs.st-andrews.ac.uk/skel-test-master/tutorial/bin/tutorial.html.

[19] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5 (2013), pp. 552–593. DOI: 10.1017/S095679681300018X.

[20] Ana Bove, Peter Dybjer, and Ulf Norell. "A brief overview of Agda–a functional language with dependent types". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 73–78.

[21] Gilles Dowek et al. "The Coq proof assistant user's guide: version 5.8". PhD thesis. INRIA, 1993.

[22] RITA LOOGEN, YOLANDA ORTEGA-MALLÉN, and RICARDO PEÑA-MARÍ. "Parallel functional programming in Eden". In: *Journal of Functional Programming* 15.3 (2005), pp. 431–475. DOI: 10.1017/S0956796805005526.

[23] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[24] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[25] Marco Aldinucci et al. "Fastflow: high-level and efficient streaming on multi-core". In: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017).

[26] Richard Kaye. "Models of Peano arithmetic". In: (1991).