CLUSTER INNOVATION CENTRE, UNIVERSITY OF DELHI

Formalising Mathematics and Computing in Agda

Saransh Chopra

Supervisor: Assoc. Prof. Jacques Carette
Department of Computing and Software,
Faculty of Engineering,
McMaster University

A report submitted in partial fulfillment of the requirements for the degree of Bachelor of Technology

Submission Date: December 13, 2023

Disclaimer

I confirm that this report is my own work and I have documented all sources and material used.

New Delhi, December 13, 2023 Saransh Chopa

Acknowledgments

This work was supported by a joint grant from Mitacs and Shastri Indo-Canadian Institute (through the Government of India), under the Mitacs Globalink Research Internship Program.

Abstract

Type systems were originally devised to aid programmers in code development by validating variable passing, method interaction, and overall code structure to detect runtime errors at compile time. However, the Curry-Howard Isomorphism (Curry et al. 1980) has propelled the evolution of type systems beyond their initial purpose, transforming them into tools not only for programming assistance but also for supporting mathematicians in theorem proving. This evolution has given rise to strongly typed functional programming languages, including Agda (The Agda Team; Norell 2009), Idris 2 (Brady 2021), Lean (Moura & Ullrich 2021), and Coq (The Coq Development Team 2023; Bertot & Castran 2010), which leverage the Curry-Howard Isomorphism to act as proof assistants. In response to the growing requirement for strongly typed systems exhibiting this isomorphism, it becomes important to develop and distribute free and open-source software for researchers. This work aims to enhance Agda's standard library (The Agda Community 2023) and prepare it for version 2.0.0. The improvements encompass refactoring the library, introducing new functions and proofs, addressing mathematical bugs, streamlining the library's dependency graph to reduce compile time, and incorporating concepts of finiteness. By undertaking these enhancements, the research contributes to the ongoing development of robust tools for both programming and formal mathematical reasoning, ultimately contributing to both, the fields of computing and mathematics.

Keywords: Type Theory, Proof Theory, Functional Programming, Theory of Computation, Computational Logic, Interactive Theorem Proving, Agda, Idris2

Contents

Listings

List of Figures

List of Tables

1 Introduction

Type systems integrated into programming languages and compilers have helped programmers write better code for decades, and their use cases have been transforming rapidly in recent years. These type systems are now not restricted to compilers or compiled languages. Interpreted languages like Python now have native support for type hints, and there has been a boom in static analysis tools for type checking. These type systems have a unique mathematical interpretation, first proposed by Bertrand Russell to fill the gaps in the classical set theory. This mathematical or theoretical representation of the type systems is studied rigorously under the mathematical field of Type Theory. Alonzo Church's typed λ-calculus and Per Martin-Löf's intuitionistic type theory serve as the basis of all the modern mathematical work carried out in type theory.

The Curry-Howard correspondence enabled researchers to visualize and formalize type systems as a mirror image of type theory. Formally, the Curry–Howard correspondence shows an isomorphism between the proof systems and the models of computation. In essence, the Curry-Howard isomorphism connects computational logic with type theory, enabling researchers to write mathematical proofs as code. For instance, (Jespercockx) shows the correspondence between propositional logic and simple non-dependent types:

Propositional logic		**Type system**
proposition	P	type
proof of proposition	$p : P$	program of a type
conjunction	$P \times Q$	pair type
disjunction	$Either\ P\ Q$	either type
implication	$P \to Q$	function type
truth	$\top$	unit type
falsity	$\bot$	empty type
negation	$P \to \bot$	function to $\bot$
equivalence	$(P \to Q) \times (Q \to P)$	pair of two functions

Table 1: Correspondence between propositional logic and simple non-dependent types.

This correspondence paved the development of Coq, Lean, Agda, and Idris, computation systems that can be used as proof assistants and functional programming languages. Functional programming is a programming paradigm based on the declarative programming style originating from the typed λ-calculus. The definition of functions in a functional programming language is often similar to the mathematical definition of functions. These functions are *first-class* citizens in such languages and can be passed around like a variable, allowing *higher-order* functions to exist. Furthermore, these functions are usually *pure* (not to be confused with *total*), having no side effects at the compile or run time. Most functional programming languages are intertwined with type systems and type theory, allowing them to be used as proof assistants.

Proof assistants are currently finding applications in aiding journals during the peer-review process. Numerous mathematical journals are actively encouraging researchers to supplement

their paper submissions with a code snippet composed in any widely accepted proof assistant. The inclusion of code snippets enhances the credibility of the paper's assertions and mitigates the susceptibility to human errors. Furthermore, the foundational concepts of type systems and type theory, integral to most static analysis tools, are employed in these tools to fulfill their objectives. While languages like Agda traditionally garner interest within academic research, particularly in the realm of formalized mathematics, their utility has expanded into engineering domains. In conjunction with formal languages, automata theory, control theory, and program semantics, type theory is integral to formal methods. The application of formal methods has yielded sophisticated methodologies for verifying the accuracy of software deployed in safety-critical or security-critical systems, such as avionics software. Additionally, these languages are gaining prominence in the domain of compilers. For instance, Compcert (Leroy 2009) serves as a compiler for a subset of the C programming language, authored in Coq, and subjected to formal verification.

Through this work, we aim to improve the standard library of Agda and prepare it for its second major release, v2.0.0. The improvements encompass refactoring the library, introducing new functions and proofs, addressing mathematical bugs, streamlining the library's dependency graph to reduce compile time, and incorporating concepts of finiteness. By undertaking these enhancements, the research contributes to the ongoing development of robust tools for both programming and formal mathematical reasoning, ultimately contributing to both, the fields of Computing and Mathematics.

2 Agda and its standard library

2.1 Type Theory

Agda is a *total*, strongly typed functional programming language and an interactive proof assistant. Agda is based on the Martin-Löf Type Theory, one of the foundational type theories introduced in the 1900s. Type systems in Computer Science were first introduced to prevent program crashes that would follow from interpreting data in memory incorrectly. But now, type systems can do much more, and the popular well-typed languages, like Java, don't exploit these type systems to go beyond their initial use case. Types in Agda are *first-class* values and can be *polymorphic*. Most of the data types in Agda are inductive or recursive, using itself in its definition.

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Listing 1: Inductive definition of the data type Natural numbers.

```
data List (A : Set) : Set where
  []    : List A
  _::_  : A → List A → List A
infixr 5 _::_
```

Listing 2: Inductive definition of the data type List.

2.2 Functional Programming

Given the functional and declarative nature of Agda, the language restricts programmers to only using pure functions in their code, that is, no classes or object-oriented designs. This also means that functions should be mathematically sound, return exactly one value, and usually be recursive. Each function in Agda must have a type signature and be mathematically complete and *total*. Being a complete language, Agda can detect potential run-time errors at compile-time. Functions in Agda support both explicit and implicit, given that Agda can infer the type of implicit arguments at run-time. Lastly, Agda has in-built support for *mixfix operators*, functions with underscores in their names. These underscores represent where the arguments to a function must be passed. The mixfix operators can be assigned a *fixity* that tells Agda about the operator's precedence.

```
_+_ : ℕ → ℕ → ℕ
zero  + y = y
suc x + y = suc (x + y)

infix 4 _+_
```

Listing 3: Definition of the addition mixfix operator with fixity 4.

```
if_then_else_ : {A : Set} → Bool → A → A → A
if true  then x else y = x
if false then x else y = y
```

Listing 4: Definition of the if-then-else control flow statement with polymorphic types and implicit argument.

2.3 Dependent Types

Languages such as Java and C++ impose constraints on the return type of a function, a characteristic that remains indiscernible during compilation. In contrast, Agda exhibits first-class support for dependent types. Dependent types can reference or depend on components within a program. Incorporating dependent types in Agda facilitates the composition of significantly more precise type specifications compared to non-dependent type systems.

```
D : Bool → Set
D true  = ℕ
D false = Bool

if[_]_then_else_ : (X : Bool → Set)
                 → (b : Bool)
                 → X true
                 → X false
                 → X b
if[ X ] true  then x else y = x
if[ X ] false then x else y = y
```

```
unfamiliar : (b : Bool) → {n : ℕ} → D b
unfamiliar b {n} = if[ D ] b
                   then (suc (suc n))
                   else false
```

Listing 5: A dependent type D, dependent if-then-else control flow, and a function, unfamiliar, returning different type of values based on the input.

```
data Vec (A : Set) :ℕ → Set where
 []    : Vec A zero
 _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)

zeroes : (n : ℕ) → Vec ℕ n
zeroes zero     = []
zeroes (suc n) = zero :: zeroes n
```

Listing 6: Definition of Vec, a widely used dependent data type, and a function, zeroes, with dependently typed return value.

2.4 Interactive proof assistant

In addition to its role as a programming language, Agda serves as a proficient proof assistant. This dual functionality allows us to articulate theorems within Agda and subsequently verify their correctness through Agda's built-in proof-checking capabilities. The foundation for Agda's proof-assisting capabilities lies in the Curry-Howard isomorphism discussed earlier. The fundamental principle of the Curry-Howard correspondence posits that logical propositions, such as "P and Q," "not P," and "P implies Q," can be interpreted as types, with their valid proofs representing elements of these types. For instance, the proposition "If (P implies Q) and (Q implies R), then (P implies R)" translates into the Agda type $(P \to Q) \times (Q \to R) \to (P \to R)$:

```
proof : {P Q R : Set}
      → (P → Q) × (Q → R)
      → (P → R)
proof (f , g) = λ x → g (f x)
```

Listing 7: Proof of "If (P implies Q) and (Q implies R) then (P implies R)" in Agda.

Moreover, Agda offers the capability to engage with predicate logic by leveraging Martin-Löf's conception of the identity type, denoted as $x \equiv y$. The identity type within Agda is characterized by polymorphism and establishes a solitary constructor, $refl : x \equiv x$. The $refl$ constructor is instantiated only when the supplied arguments are equivalent; otherwise, the data type lacks any constructor, akin to the $\bot$ type:

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x
infix 4 _≡_
```

Listing 8: Propositional Equality, as proposed by Per Martin-Löf, in Agda.

The ≡ constructor can be used to equate values and treat the complete equation as a *type*. These proofs can then be solved iteratively and interactively in Agda.

```
length-is-3 : length (1 :: 2 :: 3 :: []) ≡ 3
length-is-3 = {!!}  -- a goal

-- ?0 : length (1 :: 2 :: 3 :: []) ≡ 3 -- the context - C-c C-l
-- Goal: 3 ≡ 3 -- the goal - C-c C-,

length-is-3 : length (1 :: 2 :: 3 :: []) ≡ 3
length-is-3 = {!refl!}  -- filling the goal - C-c C-<space>

length-is-3 : length (1 :: 2 :: 3 :: []) ≡ 3
length-is-3 = refl
```

Listing 9: Proving the length of the list 1 :: 2 :: 3 to be 3 with interactiveness of Agda.

3 Formalising Mathematics and Computing in Agda

The Agda standard library, while already substantial, could be expanded, particularly in computer science and mathematics. The primary aim was to expand Agda's capabilities as both a programming language and a theorem prover, emphasizing the incorporation of additional knowledge, especially in the domains of algebra and computer science.

To achieve this goal, my tasks involved implementing various mathematical and computer science structures, accompanied by the development of corresponding proofs. This work enhanced Agda's standard library and prepared it for version 2.0.0. The improvements encompassed refactoring the library, introducing new functions and proofs, addressing mathematical bugs, streamlining the library's dependency graph to reduce compile time, and incorporating concepts of finiteness.

This expansion not only reinforced Agda's role as a reliable theorem prover but also contributed to the ongoing evolution of formalized mathematics and verified computation within the Agda ecosystem. The following sub-sections present the details of the work carried out by me in the summer.

3.1 Simplifying the dependency graph

Agda's standard library is meticulously structured with a modular design, characterized by explicit boundaries between different modules. Internal dependencies among these modules are employed to mitigate redundancy and diminish compile time. However, as a user-facing software, the

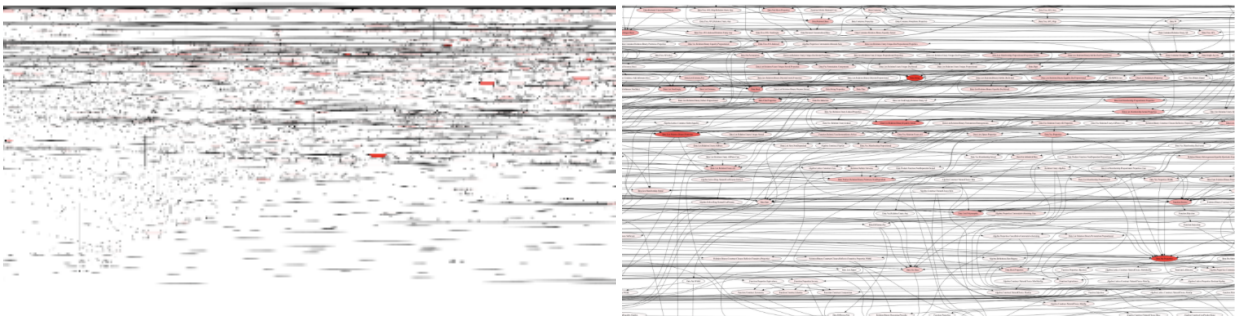
act of selectively importing specific modules can inadvertently introduce numerous unnecessary dependencies due to internal imports. This unwarranted inclusion extends compile times for end-users and amplifies the workload for developers overseeing Agda's Standard Library, as the entire library undergoes compilation in each Continuous Integration run for testing. The cumulative effect of transitive imports further exacerbates the compile time.

The library's intricate network of dependencies can be visualized as a graph, wherein module names constitute nodes and import relationships are represented by edges. Despite the enormity of this dependency graph, it can be systematically generated using the *–dependency-graph* flag during compilation. This work endeavored to streamline the dependency graph programmatically, thus curbing the library's compile time by eliminating superfluous imports.

The initial step involved assigning a score to each node in the dependency graph based on the following formula:

$$\sqrt{dependsUpon + isDependendedUpon^2}$$

Subsequently, the ten modules with the highest scores were identified, and imports within these modules, along with their transitive imports in other modules, were pruned. Notably, efforts to reduce compile time did not exhibit a linear correlation with the magnitude of pruning. Consequently, the original scoring formula was refined to account for the transitive nature of dependencies. To include transitive dependencies, the score is incremented by 1 each time a transitive dependency is discovered. This revised formula, coupled with various visualization and auxiliary functions implemented using Graphviz, was made publicly available on GitHub as a Cabal package named dot-analysis (Gallais).



(a) A portion of the dependency graph. (b) Zoomed in portion of the dependency graph.

Figure 1: Standard library's dependency graph with nodes colored against their score.

Figure 1 shows the standard library's dependency graph with nodes colored against their score - the higher the score, the brighter the node. Through the implementation of approximately 20 Pull Requests (PRs), the dependency graph underwent significant pruning, resulting in an enhanced user experience, along with notable reductions in both the overall compile time of Agda's Standard Library and the compile times of its individual modules.

3.2 Bug fixes

Similar to many software projects, Agda's standard library harbored both reported and unnoticed bugs within its codebase. Throughout the summer, my focus was on researching and introducing *fixities* for every *mixfix* operator present in the standard library. In Agda, a *mixfix* operator is characterized by having an underscore (_) in its name. Arguments for these mixfix operators are then provided in place of the added underscores. Each mixfix operator can possess its specific *fixity*, which communicates the operator's precedence to Agda. Notably, this endeavor marked the first exploration of fractional fixities within the standard library, although the proposal for their inclusion was not accepted by the maintainers.

Beyond the introduction of fixities, significant bug fixes involved the transformation of explicit arguments to implicit in functions within the *Data.Vec* and *Data.List* modules. Agda supports both *implicit* and *explicit* arguments, with implicit arguments allowing users to omit specifying the type, as Agda can infer it from the type of another argument. For instance, the explicit version of the identity function:

```
id : (A : Set) → A → A
```

Listing 10: Identity function with explicit arguments.

can be expressed with implicit arguments:

```
id : {A : Set} → A → A
```

Listing 11: Identity function with implicit arguments.

The *iterate* and *replicate* functions originally required the length of the generated *Vec* as an explicit argument, even though Agda could infer it.

```
iterate : (A → A) → A → ∀ {n} → Vec A n
replicate : ∀ {n} → A → Vec A n
```

Listing 12: Original iterate and replicate functions with explicit arguments.

To address this, both functions, along with their dependent functions, were modified to utilize implicit arguments:

```
iterate : (A → A) → A → ∀ n → Vec A n
replicate : ∀ n → A → Vec A n
```

Listing 13: Modified iterate and replicate functions with implicit arguments.

Additionally, documentation and version control system (VCS) bugs, as well as those arising during the development of new proofs and functions, were rectified within the standard library.

3.3 Refactoring the Function and Data hierarchies

Agda's standard library underwent a substantial API refactor aimed at achieving uniformity, rectifying unnoticed bugs, and enhancing overall code cleanliness. This comprehensive refactoring initiative served as a pivotal phase for the library's second major release, v2.0.0, with the overarching objective of providing users with a consistent and user-friendly experience.

A primary focus of the refactoring effort was the *Function* hierarchy, which exhibited inconsistencies compared to other record hierarchies within the library. Non-conformity with established rules in other hierarchies had deterred users from fully utilizing the Function hierarchy. To address this, a concerted effort was made to align the Function hierarchy with the established norms, enhancing its coherence and usability for end-users.

Similarly, the modules *Data.Vec*, *Data.Vec.Functional*, and *Data.List* manifested internal inconsistencies and lacked uniformity across each other. Notably, operations such as insertion, deletion, and updating were inconsistently named or implemented within these modules. This issue, persisting since 2019, had posed challenges for Agda developers and users, though it had not been accorded high priority by the Agda development team.

Over the course of the summer, I actively contributed to the refactoring of the Function hierarchy and Data modules. This intricate process involved extensive discussions, modifications to function and proof names, adjustments to type signatures, alterations to definitions, deprecation of outdated elements, assurance of backward compatibility, and resolution of bugs that surfaced during the refactoring. Apart from the Function hierarchy, the following proofs and functions were refactored or deprecated as part of this effort:

```
-- Relation.Binary.Construct.(Converse/Flip)
Relation.Binary.Construct.Converse
↠ Relation.Binary.Construct.Flip.EqAndOrd
Relation.Binary.Construct.Flip
↠ Relation.Binary.Construct.Flip.Ord

-- Codata.Guarded.Stream.Properties
drop-fusion   ↠ drop-drop

-- Codata.Sized.Colist.Properties
drop-drop-fusion   ↠ drop-drop

-- Data.List.Base
_--_ ↠ removeAt

-- Data.List.Properties
take++drop   ↠ take++drop≡id

length--  ↠ length-removeAt
map--     ↠ map-removeAt
```

```
-- Data.Vec.Base
remove   ↠ removeAt
insert   ↠ insertAt
updateAt : Fin n → (A → A) → Vec A n → Vec A n
  ↠
updateAt : Vec A n → Fin n → (A → A) → Vec A n

-- Data.Vec.Properties
take-drop-id   ↠ take++drop≡id

map-insert   ↠ map-insertAt

insert-lookup     ↠ insertAt-lookup
insert-punchIn    ↠ insertAt-punchIn
remove-PunchOut   ↠ removeAt-punchOut
remove-insert     ↠ removeAt-insertAt
insert-remove     ↠ insertAt-removeAt

-- Data.Vec.Functional.Properties

insert-lookup     ↠ insertAt-lookup
insert-punchIn    ↠ insertAt-punchIn
remove-punchOut   ↠ removeAt-punchOut
remove-insert     ↠ removeAt-insertAt
insert-remove     ↠ insertAt-removeAt

-- Data.Vec.Functional
remove : Fin (suc n) → Vector A (suc n) → Vector A n
  ↠
removeAt : Vector A (suc n) → Fin (suc n) → Vector A n

updateAt : Fin n → (A → A) → Vector A n → Vector A n
  ↠
updateAt : Vector A n → Fin n → (A → A) → Vector A n

-- Relation.Nullary.Decidable.Core
excluded-middle ↠ ¬¬-excluded-middle
```

Listing 14: Functions and proofs refactored outside of the Function hierarchy.

This comprehensive refactoring initiative aimed not only to address longstanding issues but also to enhance the overall coherence and usability of Agda's standard library, thereby laying the groundwork for a more robust and user-friendly experience in the library's second major release

3.4 New proofs and functions

During the summer, my research involved a comprehensive examination of the distinctions between Idris 2 and Agda's standard library, with a focus on identifying potential missing functionalities. Similar to Agda, Idris 2 is a purely functional programming language where types hold a primary role. While Idris 2 can serve as a proof assistant, its primary design is geared toward programming, making it imperative to delve into the independent evolution trajectories of Idris 2 and Agda.

A meticulous exploration resulted in the compilation of a detailed list enumerating functionalities absent in Agda. This list emerged from thorough discussions with my supervisor and co-intern, subsequently undergoing scrutiny wherein certain functionalities were excluded based on considerations of significance and feasibility. The finalized roster of functions, proofs, and additional features was then implemented within Agda's standard library:

```
true-xor              : true xor x ≡ not x
xor-same              : x xor x ≡ false
not-distrib^l-xor     : not (x xor y) ≡ (not x) xor y
not-distrib^r-xor     : not (x xor y) ≡ x xor (not y)
xor-assoc             : Associative _xor_
xor-comm              : Commutative _xor_
xor-identity^l        : LeftIdentity false _xor_
xor-identity^r        : RightIdentity false _xor_
xor-identity          : Identity false _xor_
xor-inverse^l         : LeftInverse true not _xor_
xor-inverse^r         : RightInverse true not _xor_
xor-inverse           : Inverse true not _xor_
∧-distrib^l-xor       : _∧_ DistributesOver^l _xor_
∧-distrib^r-xor       : _∧_ DistributesOver^r _xor_
∧-distrib-xor         : _∧_ DistributesOver _xor_
xor-annihilates-not   : (not x) xor (not y) ≡ x xor y
```

Listing 15: Properties of _xor_.

```
find^b          : (A → Bool) → List A → Maybe A
findIndex^b     : (A → Bool) → (xs : List A) → Maybe $ Fin (length xs)
findIndices^b   : (A → Bool) → (xs : List A) → List $ Fin (length xs)
find            : Decidable P → List A → Maybe A
findIndex       : Decidable P → (xs : List A) → Maybe $ Fin (length xs)
findIndices     : Decidable P → (xs : List A) → List $ Fin (length xs)

iterate : (A → A) → A → ℕ → List A

insertAt : (xs : List A) → Fin (suc (length xs)) → A → List A
updateAt : (xs : List A) → Fin (length xs) → (A → A) → List A
removeAt : (xs : List A) → Fin (length xs) → List A
```

Listing 16: Functions of List.

```
drop-all : n ≥ length xs → drop n xs ≡ []

drop-drop : drop n (drop m xs) ≡ drop (m + n) xs

lookup-replicate   : lookup (replicate n x) i ≡ x
map-replicate      : map f (replicate n x) ≡ replicate n (f x)
zipWith-replicate : zipWith _⊕_ (replicate n x) (replicate n y)
  ≡ replicate n (x ⊕ y)

length-iterate : length (iterate f x n) ≡ n
iterate-id     : iterate id x n ≡ replicate n x
lookup-iterate : lookup (iterate f x n)
  (cast (sym (length-iterate f x n)) i) ≡ ℕ.iterate f x (toℕ i)

length-insertAt    : length (insertAt xs i v) ≡ suc (length xs)
length-removeAt'   : length xs ≡ suc (length (removeAt xs k))
removeAt-insertAt : removeAt (insertAt xs i v)
  ((cast (sym (length-insertAt xs i v)) i)) ≡ xs
insertAt-removeAt : insertAt (removeAt xs i)
  (cast (sym (lengthAt-removeAt xs i)) i) (lookup xs i) ≡ xs
```

Listing 17: Proofs of List.

```
iterate-id     : iterate id x n ≡ replicate x n
take-iterate   : take n (iterate f x (n + m)) ≡ iterate f x n
drop-iterate   : drop n (iterate f x n) ≡ []
lookup-iterate : lookup (iterate f x n) i
  ≡ ℕ.iterate f x (toℕ i)
toList-iterate : toList (iterate f x n) ≡ List.iterate f x n

length-toList   : List.length (toList xs) ≡ length xs
toList-insertAt : toList (insertAt xs i v) ≡ List.insertAt
  (toList xs) (Fin.cast (cong suc (sym (length-toList xs))) i) v
```

Listing 18: Proofs of Vec.

In parallel with aligning functionalities between these sibling languages, a translation dictionary was crafted. This dictionary served as a valuable resource for developers, delineating the differences between Idris 2 and Agda and providing guidance on how code snippets from one language could be effectively translated into the other.

3.5 Finiteness of a Setoid

Agda's standard library lacked constructively well-behaved notions of finiteness for setoids. My contribution involved the formulation of record types facilitating dependently typed programming with finite setoids in Agda. Specifically, our focus was on programmatically establishing whether a setoid possesses the following characteristics:

- Strong Finiteness,
- Subfiniteness,
- Finitely Enumerable,
- Sub-finitely Enumerable, and
- Subfinitely Enumerable.

The incorporation of notions of finiteness into the standard library was deemed crucial, given that such structures were extremely helpful during the process of proving theorems (Agda):

- All finite setoids have decidable equality.
- All finite setoids can be finitely enumerated.
- Finite setoids are closed under operations like finite products and sum.

(Spiwack & Coquand 2010) defined a finite set as an enumerated set A, noted $A \in F$ when there is a list of all its elements. They termed such a list an enumeration of A and formally explored their relationships, providing combinators to define functions on finite sets and prove formulas quantified over finite sets. However, their definition proved too vague for formalization in Agda. Subsequently, we explored various definitions and notions of finiteness (Yorgey 2014; Kahl 2012; Firsov & Uustalu 2015; Bezem et al. 2012; Parmann 2014) and decided to formalize the definition provided by (Saving).

Let $[n]$ be the set $\{0,1,...,n-1\} = \{m \in \mathbb{N} \mid m < n\}$ -

- A set S is finite if and only if there exists some $n \in \mathbb{N}$ and a bijection $S \leftrightarrow [n]$.
- A set S is subfinite if and only if there is some $n \in \mathbb{N}$ and some injection $S \rightarrow [n]$.
- A set S is finitely enumerable if and only if there is some $n \in \mathbb{N}$ and some surjection $[n] \rightarrow S$.
- A set S is sub-finitely enumerable if and only if there is some finitely enumerable S' and some injection $S \rightarrow S'$.
- A set S is subfinitely enumerable if and only if there is some subfinite set S' and some surjection $S' \rightarrow S$.

The definition proposed by (Saving) was accepted by the maintainers of the standard library, and the following records were authored in pure Agda to implement this definition.

```
record IsStronglyFinite (X : Setoid c l) (n : ℕ)
       : Set (c ⊔ l) where
  field
    inv : Inverse X (setoid (Fin n))

record StronglyFinite (X : Setoid c l) (n : ℕ)
       : Set (c ⊔ l) where
  field
    isFinite : IsStronglyFinite X n
```

```
record Subfinite (X : Setoid c l) (n : ℕ)
       : Set (c ⊔ l) where
  field
    size : ℕ
    inj  : Injection X (setoid (Fin size))

record FinitelyEnumerable ((X : Setoid c l) (n : ℕ)
       : Set (c ⊔ l) where
  field
    size : ℕ
    srj  : Surjection (setoid (Fin size)) X

record InjectsIntoFinitelyEnumerable (X : Setoid c l) c′ l′
       : Set (c ⊔ l ⊔ lsuc (c′ ⊔ l′)) where
  field
    Apex : Setoid c′ l′
    finitelyEnumerable : FinitelyEnumerable Apex
    inj : Injection X Apex

  open FinitelyEnumerable finitelyEnumerable public

record SurjectionFromFinitelyEnumerable (X : Setoid c l) c′ l′
       : Set (c ⊔ l ⊔ lsuc (c′ ⊔ l′)) where
  field
    Apex : Setoid c′ l′
    subfinite : Subfinite Apex
    srj : Surjection Apex X

  open Subfinite subfinite public
```

Listing 19: Pure Agda records implementing notions of finiteness for a Setoid.

These records are currently pending integration into the standard library, awaiting approval from maintainers. The anticipated release of this work aligns with the standard library's v2.1.0 release. Additionally, further efforts are underway to incorporate necessary proofs and functions relevant to these records, ensuring the correctness and bug-free nature of the implemented code.

4 Conclusion

The focus of this work was the improvement of Agda's standard library. The incorporation of constructs enabling dependently typed programming with finite setoids added a constructive layer to Agda's capabilities. The meticulous refinement of the library's dependency graph has not only reduced compile times but has also streamlined the overall development experience. Additionally, the introduction of fixities for mixfix operators enhanced code clarity and readability. Comparative

analyses between Agda and Idris 2 unveiled opportunities for enriching Agda's standard library further. The exploration of missing functionalities and the creation of a translation dictionary served as foundational steps toward aligning these sister-like languages.

As my fellowship concluded amidst these endeavors, this thesis serves as a snapshot of the progress made and a roadmap for future enhancements to Agda's standard library. The collaborative nature of this work, involving discussions, refinements, and contributions from various stakeholders, exemplifies the collective effort required to advance the field of dependently typed programming. In conclusion, this work contributes not only to the refinement of Agda's standard library but also to the broader landscape of dependently typed programming, highlighting the ongoing synergy between programming languages and formal mathematics.

5 Future work

As my fellowship concluded, I was in the midst of incorporating notions of finiteness into Agda's standard library. There remains an opportunity to extend this work by finalizing the records for finite setoids, augmenting them with the requisite proofs and functions, and subsequently integrating them into the upcoming release of the library. Despite substantial efforts, not every mixfix operator in the library currently has an assigned fixity, presenting an avenue for gradual inclusion. Additionally, there is potential for further enrichment by replicating functionalities from Idris 2 into Agda. The Agda-Idris2 dictionary, created as part of my work, can be expanded to encompass more functionalities and foster a broader understanding of the similarities and differences between the two languages.

References

AGDA. finite setoid · issue 863 · agda/agda-stdlib. URL https://github.com/agda/agda-stdlib/issues/863#issue-481996099.

BERTOT, YVES, und PIERRE CASTRAN. 2010. *Interactive theorem proving and program development: Coq'art the calculus of inductive constructions*. 1st Ed. Springer Publishing Company, Incorporated.

BEZEM, MARC; KEIKO NAKATA; und TARMO UUSTALU. 2012. On streams that are finitely red. *Logical Methods in Computer Science* Volume 8, Issue 4. URL http://dx.doi.org/10.2168/LMCS-8(4:4)2012.

BRADY, EDWIN C. 2021. Idris 2: Quantitative type theory in practice. *CoRR* abs/2104.00480. URL https://arxiv.org/abs/2104.00480.

CURRY, HASKELL B.; J. ROGER HINDLEY; und J. P. SELDIN (Hg.) 1980. *To h.b. curry: Essays on combinatory logic, lambda calculus, and formalism*. New York: Academic Press.

FIRSOV, DENIS, und TARMO UUSTALU. 2015. Dependently typed programming with finite sets. *Proceedings of the 11th acm sigplan workshop on generic programming*, WGP 2015, 33–44. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/2808098.2808102.

GALLAIS. Github - gallais/dot-analysis: Analysing dependency graphs produced by agda. URL https://github.com/gallais/dot-analysis.

JESPERCOCKX. agda-lecture-notes/agda.pdf at master · jespercockx/agda-lecture-notes. URL https://github.com/jespercockx/agda-lecture-notes/blob/master/agda.pdf.

KAHL, WOLFRAM. 2012. Towards certifiable implementation of graph transformation via relation categories. *Relational and algebraic methods in computer science*, hrsg. von Wolfram Kahl und Timothy G. Griffin, 82–97. Berlin, Heidelberg: Springer Berlin Heidelberg.

LEROY, XAVIER. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52.107–115. URL https://doi.org/10.1145/1538788.1538814.

MOURA, LEONARDO DE, und SEBASTIAN ULLRICH. 2021. The lean 4 theorem prover and programming language. *Automated deduction – cade 28*, hrsg. von André Platzer und Geoff Sutcliffe, 625–635. Cham: Springer International Publishing.

NORELL, ULF. 2009. *Dependently typed programming in agda*, 230–266. Berlin, Heidelberg: Springer Berlin Heidelberg. URL https://doi.org/10.1007/978-3-642-04652-0_5.

PARMANN, ERIK. 2014. Some varieties of constructive finiteness. *19th int. conf. on types for proofs and programs*, 67–69.

SAVING, MARK. Is there a definition of finite sets, such that it can be used to constructively prove that a subset and a quotient set are finite? Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/4285367 (version: 2021-10-23). URL https://math.stackexchange.com/q/4285367.

SPIWACK, ARNAUD, und THIERRY COQUAND. 2010. Constructively Finite? *Contribuciones científicas en honor de Mirian Andrés Gómez*, hrsg. von Lambán Pardo, Laureano, Romero Ibáñez, Ana, Rubio García, und Julio, 217–230. Universidad de La Rioja. Link to the full book here: http://www.unirioja.es/servicios/sp/catalogo/monografias/vr77.shtml. URL https://inria.hal.science/inria-00503917.

THE AGDA COMMUNITY. 2023. Agda Standard Library. URL https://github.com/agda/agda-stdlib.

THE AGDA TEAM. Agda. URL https://agda.readthedocs.io/en/v2.6.3/index.html.

THE COQ DEVELOPMENT TEAM. 2023. The Coq reference manual – release 8.18.0. https://coq.inria.fr/doc/V8.18.0/refman.

YORGEY, BRENT. 2014. Combinatorial species and labelled structures.