Institut für Informatik
Lehrstuhl für Theoretische Informatik
LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

Towards a Prelude for Agda

Frederic Kettelhoit

Aufgabensteller:	Prof. Martin Hofmann, PhD
Betreuer:	Dr. Andreas Abel
Abgabetermin:	28. März 2012

Ich versichere hiermit eidesstattlich, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 28. März 2012

.....

(Unterschrift des Kandidaten)

Zusammenfassung

Abhängige Typen versprechen dem aufgeschlossenen Programmierer eine Fülle von mächtigen Möglichkeiten, um Eigenschaften seiner Programme zu beschreiben, allerdings ist der Preis dafür hoch: Viele abhängig getypte "Programmiersprachen" sind mehr Theorembeweiser als Programmiersprache und bieten daher wenig Komfort für alle, die nicht über tiefgehende Kenntnisse der jeweiligen Standardbibliothek verfügen.

Agda ist eine abhängig getypte Sprache, welche explizit versucht, auch fürs Programmieren und nicht nur fürs Beweisen nützlich zu sein. Die zugehörige Standardbibliothek bietet tatsächlich das meiste der dafür notwendigen Funktionalität, doch leider gestalten eine Unmenge an Namenskonflikten zwischen verschiedenen Modulen das Leben angehender Agda Programmierer nicht gerade angenehm.

Diese Arbeit zielt darauf ab, die Situation durch die Implementierung von Typklassen, welche die Namenskonflikte auflösen, zu verbessern. Zwar sind Typklassen im Kontext nicht-abhängig getypter Programmiersprachen bereits gut verstanden, allerdings machen die ausgefeilten Typen in Agda die Situation interessanter und erfordern einige Hilfskonstruktionen. Eine Reihe von Typklassen werden zusammen mit weiterer Funktionalität zu einer "Prelude" zusammengefasst, welche alle nötigen Datentypen und Funktionen für alltägliche Agda Programme mitbringt. Neben der Prelude selber werden zwei Beispielprogramme vorgestellt, die den erfolgreichen Einsatz der Prelude im Kontext IO-lastiger Programme demonstrieren.

Abstract

Dependent types promise the open-minded programmer a wealth of powerful ways to describe properties about their programs, but the cost of this blessing is high: Many dependently typed "programming languages" are more theorem provers than programming languages and lack convenience for anyone not well versed in the depths of their standard libraries.

Agda is a dependently typed language which explicitly intends to be useful also for programming and not just for proving. Its standard library indeed offers most of the necessary functionality, but unfortunately a myriad of name clashes between the different modules make the life of an aspiring Agda programmer rather unpleasant.

This thesis aims to improve the situation by implementing type classes which resolve many of the name clashes. Even though type classes are well understood in the context of non-dependently typed languages, the more sophisticated types in Agda often make the situation more interesting and require some workarounds. A number of type classes together with additional functionality are bundled into a "Prelude", which provides all the necessary data types and functions for day-to-day Agda programming. Apart from the Prelude itself, two example programs will be discussed, that show the successful usage of the Prelude in the context of IO-related programs.

Acknowledgements

I would like to thank first of all Andreas Abel for introducing me to Agda and the strange world of dependent types, for his patience, his invaluable advice and finally his support during my numerous battles with Agda's type- and termination-checker.

Thanks also to the members of the Agda Implementor's Meeting XV, especially Dominique Devriese and Nils Anders Danielsson. The many discussions proved tremendously helpful to get a feel for the needs of other Agda programmers and helped shape the overall structure of the Prelude.

Contents

Chapter 1

Introduction

1.1 Motivation

Strong static type systems offer a lot of benefits to programmers. Not only can they help make a program execute much faster by avoiding type checks at run-time, they also help to catch bugs much earlier in the development process. Programming languages such as Standard ML [MTM97] and Haskell [HHPJW07] have shown that sophisticated static type systems do work in practice even for large programs.

But in most typed programming languages there is still a fundamental divide between terms and types. Types can usually only be indexed by types and as a result there are often properties that a programmer would like to express but which are impossible to specify directly through types. An example is the head function on lists; a function that only makes sense on lists with a length of 1 or greater. Most typed languages throw an exception on run-time if head is called on an empty list, but it would obviously be of much help to check for such errors already at compile time. The multiplication of matrices is another example where we would like to make sure that a matrix with shape $\mathsf{m} \times \mathsf{n}$ is only multiplied with a matrix of shape $\mathsf{n} \times \mathsf{p}$. Not only do these guarantees lead to more bugs found earlier in development, but a smart compiler can also omit any array bounds checking for cases where such properties can be statically proved. Once again, the type system helps make the program more efficient. Types which can be indexed by terms and therefore allow to express these kinds of properties are called "dependent types" (because they *depend* on one or more terms) and form the basis of a few highly expressive type systems [McK06][OS08].

To a limited degree the encoding of these types can be "faked" even in languages with a term-type-separation, such as Haskell [McB02]. In fact, Haskell has probably carried this art of simulating dependent types in a rich but non-dependent type system further than any language before. Unfortunately this often requires to reimplement constructs such as natural numbers on the level of types, an approach that is neither very elegant nor able to encode the whole wealth of properties that fully dependently typed languages can express.

Not surprisingly, there do exist some programming languages that harness the full power of truly dependent types [Nor07] [dt04] [Mcb05]. But because any expression in the term language can also be used in the type language, type checking quickly becomes undecidable in the presence of general recursion. Many dependently typed programming systems therefore restrict recursion and require all functions to be total. The resulting system often looks and feels more like a logic than a language, blurring the distinction between theorem prover and programming language.

Having the power of a theorem prover always at one's finger tips is definitely nice for any logician or researcher in the area of type theory, but it does not help mere programmers who would like to use the nice properties of dependent types to write programs. The underlying logic systems may be useful for both audiences, but a programmer often requires a large set of functions and libraries that are not necessary to formalize logical properties in a theorem prover. As a result the support for any kind of programming that goes beyond (in other languages) trivial programs is often lacking or non existent in today's dependently typed languages.

One dependently typed language that is intended to be suitable for programming and not only for proving is Agda [Nor07]. It ships with a fairly decent standard library [DNM+09], which provides most of the data types and functions needed for day-to-day programming. On top of that, it has the ability to interface with Haskell libraries, so even IO related programming (which is often an area where pure functional languages struggle) is possible and is indeed provided through the standard library as well.

Most experienced Agda programmers import all the necessary modules from the standard library by hand into their programs. This approach is feasible if the programmer knows the standard library very well and the program does not use too many modules at once. Otherwise such a manual import quickly gets burdensome, because many Agda modules export the same name, for example Data.List and Data.Colist, which both provide a _++_ method to append two lists or colists. A newcomer to Agda is especially affected by this, because not only does he have to figure out where to find basic functionality in the standard library, but also how to resolve conflicts between the different modules. This situation is only made worse by the large number of data types compared to a non dependently typed language: non-dependent types such as lists have dependent counterparts such as vectors and both types naturally export many functions with the same names and hence lead to an even larger number of name conflicts. A programmer with no prior Agda experience therefore has to face a significant challenge before ever having a chance to experience the beauty and power of dependent types.

1.2 Objectives

This thesis sets out to improve the situation for Agda newcomers by alleviating the problem of name clashes through extensive use of type classes and a Prelude which exports bundled functionality. Type classes were not possible prior to Agda 2.3.0 due to the lack of instance arguments (further discussed in the Related Work chapter) and therefore Agda does not use any type classes yet. The aim is to provide a type class for any function that could be considered "basic functionality" (an arguably vague term) and to implement instances for all the data types that already offer comparable functionality through the standard library. This task is unfortunately complicated by Agda's sophisticated type system, which makes the definition of type classes and their instances a fair bit more complex than in non-dependently typed languages.

Apart from the type classes itself, a further aim of this thesis is to bundle all the type classes into a Prelude comparable to the one in Haskell and also export useful functions and data types from the standard library through this Prelude. The ultimate goal is to provide a collection that acts as a "one line import" in most situations; meaning that for the run-of-the-mill Agda program a single statement to import the Prelude should be enough. Apart from the exporting of type classes and other functionality, this entails providing a few "helper-functions" that are not yet included in the standard library but have been proven useful during tests. These should ultimately be moved into the right

files in the standard library but are included in the larger "Prelude project" as of now.

At last this thesis is also meant to "stress-test" the implemented Prelude using two accompanying Agda programs which make heavy use of it. Both programs are deliberately IO-focused, because this is an area that usually requires a large number of different modules and is likely to be a stumbling block for Agda beginners. In both cases the emphasis is placed not on creating a useful program, but on showing an exemplary use of the Prelude, which is easy to understand.

1.3 Outline

This thesis begins with a discussion of related work in chapter 2, which also introduces the concrete type class mechanism used from then on. The implementation of type classes for a variety of different types and purposes is discussed in chapter 3. Building on these type classes, the structure of a Prelude for Agda is discussed in chapter 4. In chapter 5 two example programs are then presented which demonstrate the example usage of Agda's type classes and the Prelude in practice. Chapter 6 draws several conclusions.

Chapter 2

Related Work

2.1 Haskell's Prelude

The Prelude presented in this thesis is heavily influenced by the Standard Prelude of Haskell, which is no surprise considering that Agda itself is heavily influenced by Haskell and borrows a large amount of syntax from it. A specification of Haskell's Prelude is defined as part of the Haskell 98 standard [P+03] and contains many commonly used data types, functions and a number of type classes. Apart from several basic functions, types and their associated type classes, Haskell's Prelude provides numerous list operations, conversions to- and from String, and basic input and output types and operations. The Agda Prelude presented in this thesis mirrors this structure almost exactly, simply because this collection of types and operations is indeed required for most programs (although the Agda Prelude also exports a few more proof-related types and functions due to Agda's strong logical foundations). The Prelude outlined in this thesis exports much less than the Haskell version though, in part because doing otherwise would go beyond the scope of a master thesis and in part because Agda's standard library itself still does not provide all the equivalent functionality.

One important difference between Haskell's Prelude and the one outlined hereafter is the level of integration: While Haskell's version is automatically imported into every Haskell program which does not explicitly declares the opposite, the Prelude for Agda is simply a normal module that has to be imported manually. One reason for this is that in Haskell the Prelude really *provides* all the functionality it exports, whereas the Agda version simply bundles and re-exports existing functionality from other modules in the standard library. Both approaches work well and result in a very similar end result from the perspective of a user.

2.2 Type classes in Haskell

Type classes were introduced in Haskell to address the issue of ad-hoc polymorphism in a unified way [WB89]. The primary motivation for ad-hoc polymorphism was originally to provide overloaded operations on Integers and Floats as well as an overloaded equality function. Earlier functional languages such as Standard ML [MTM97] or Miranda [Tur85] had used different approaches for both of these areas and neither of them provided a method that could be used to address the problem of ad-hoc polymorphism in general.

Haskell instead uses records containing the overloaded functionality and passes these as an additional argument to any caller of the overloaded operations. The record type

acts as the type class definition itself whereas an inhabitant of the record type acts as an instance that provides all the specified functions and constants. The record parameter is inferred by Haskell, only a class constraint in the type signature of the caller signals the use of overloaded functionality.

The benefit of this approach is that it can be translated into a Hindley/Milner type system without any notion of ad-hoc polymorphism rather easily and at compile time. All that is needed is to augment the callers with an additional parameter for the type class record and to access the relevant field of the record to get the overloaded function or constant. As an example, the next block of code defines a type class Example (containing two functions and one constant) and two instances, one for Char and one for Int in typical Haskell code, *before* translation:

```
class Example a where
  fun :: a → Bool → Bool
  const :: a
instance Example Char where
  fun = funChar
  const = constChar
instance Example Int where
  fun = funInt
  const = constant
```

A user of this type class would now exhibit a class constraint of the form Example a ⇒ in the type signature. An example is the following:

```
useExample1 :: Example a ⇒ a → Bool → Bool
useExample1 x = fun x
```

Whenever an instance of the type class is directly used, the right instance is inferred by Haskell automatically:

```
useExample2 :: Bool
useExample2 = fun 'c' true
```

After translation into code without ad-hoc polymorphism, the passing of the record (which in Haskell is only syntactic sugar for an algebraic data type) becomes apparent:

```
data ExampleR a = ExampleRecord (a → Bool → Bool) a
fun (ExampleRecord f c) = f
const (ExampleRecord f c) = c
exampleRChar :: ExampleR Char
exampleRChar = ExampleRecord funChar constChar
exampleRInt :: ExampleR Int
exampleRInt = ExampleRecord funInt constInt
useExample1 :: ExampleR a → a → Bool → Bool
useExample1 exampleRa x y = fun exampleRa x y
useExample2 :: Bool
useExample2 = fun exampleRchar 'c' true
```

Even though this translation seems rather well behaved, there is one significant source of complexity in Haskell's implementation of type classes: recursive resolution of

type class constraints. This allows Haskell to infer even instances of type classes that depend on instances of the same or other type classes themselves. An example is the implementation of the Show type class (used to convert arbitrary data types to strings) for lists of values: To convert a list to a string, all the elements of the list have to be an instance of the Show type class themselves. The type of the Show instance for lists is hence Show a ⇒ Show [a]. Such a recursive resolution of type class instances has turned out to be a powerful but complex tool and can be used to perform compile-time computations on the level of types [KS04].

2.3 Type classes in Agda

At the heart of Agda's approach to type classes lies the insight that a full-blown translation of type classes into records using special syntax is orthogonal to the issue of passing a suitable instance to a caller. As a consequence Agda relies on ordinary records as a structuring mechanism for type classes and their instances; a translation like in Haskell is not necessary. But a way to pass the right record instance implicitly to the caller is still required and Agda provides "instance arguments" [DP11] for that purpose.

Instance arguments are very similar to Agda's implicit arguments. Both are inferred automatically by Agda, if possible. The crucial difference is that Agda searches both context and scope to find possible candidates for instance arguments and picks one if no constraints are invalidated and no ambiguous choice for candidates exists. Just like implicit arguments, instance arguments are first class citizens in Agda, meaning that they can be used in any place where normal arguments or implicit arguments can be used, including lambdas. An example of a function using an instance argument is the following:

```
module InstanceTest where

open import Data.Bool using (Bool; true)
open import Data.Bool.Show using (show)
open import Data.String using (String)

boolValue : {{b : Bool}} → Bool
boolValue {{b}} = b

showBool : String
showBool = show boolValue
  -- normalized: "true"
```

In the above code, exactly one value of type Bool is in scope: true. The function boolValue expects an instance argument of type Bool and Agda will pass the value true for the argument {{b}} automatically, as there is no ambiguous choice. The function showBool can then call the boolValue function and convert the value into a value of type String, which results in the expected string `"true"`.

A significant difference between Haskell's type class resolution and Agda's instance arguments is that Agda's instance search is not recursive. This was a deliberate choice, made to keep the implementation complexity low and prevent the instance search from turning into an alternative computation model (type-level computations are already supported by Agda's type system anyway). The drawback is that some instances that would be legal in Haskell cannot be typed in Agda without explicitly specifying the desired instance.

Chapter 3

Type Classes in Agda

We will show the implementation of type classes in Agda using the append function _++_ as a running example. This function is both heavily used in many different types of programs and also allows a nice explanation of the difficulties that sophisticated types introduce. We will start by implementing the type class for a simple type and then progress to more complex types step by step. Every example is accompanied by another full-blown type class from the Prelude which illustrates the principles in a different context.

3.1 Simple Types

Let us begin with the basic structure of the type class and one instance for the type String, which is treated as a primitive type in Agda (not as a List Char, although it can be converted into one).

At the heart of every type class lies of course the definition of the functionality itself. We can implement this in Agda using a record with a field for each provided function. To allow different types to instantiate this type class, we parametrize the record by a type A which is of type Set, the set of small types. Since the append function takes two elements of type A and concatenates them to form another element of type A, the type of the append function is therefore simply $A \to A \to A$. Together with an operator precedence declaration, the resulting record looks like this:

```
record ++Class (A : Set) : Set where
  field
    _++_ : A → A → A
  infixr 5 _++_
```

It is now possible to use this record with different types and a concrete instantiation of the append function, but we would still need to pass the particular record as an argument to every caller of the append function. This is not very convenient and we would like the concrete instance to be inferred depending on the context. To do this we can use the aforementioned instance search mechanism and provide the record as an instance argument which is searched for by Agda automatically. The resulting code is a simple wrapper function which takes an instance of the record as an (inferred) instance argument and just opens this instance record to return the right field. In cases where there is only one unambiguous choice Agda's search will find the right instance, otherwise an error is raised.

```
_++_ : {A : Set} {{++Instance : ++Class A}} → A → A → A
_++_ {{++Instance}} = ++Class._++_ ++Instance
```

The above code not only looks like boilerplate, it *is*. The structure behind the opening of the record and the wrapping into a new function always stays the same, even the type of the wrapper function is just a duplicate of the corresponding record field. Fortunately Agda provides some special syntax which does the same as the wrapper above:

```
open ++Class {{...}} public
```

The only thing left to do now is to provide a concrete instance for the String type. This is simple: We create an inhabitant of our ++Class Record type with String as the record parameter and the _++_ function defined in the Data.String module as the value of the _++_ field of the record:

```
++InstanceString : ++Class String
++InstanceString = record { _++_ = Data.String._++_ }
```

Together with a few accompanying import statements and the module declaration of the file, the complete type class with an instance for the String type now looks like this:

```
module AppendSimple where

open import Data.String using (String)

record ++Class (A : Set) : Set where
  field
    _++_ : A → A → A
  infixr 5 _++_

open ++Class {{...}} public

++InstanceString : ++Class String
++InstanceString = record { _++_ = Data.String._++_ }
```

Even though we will add more complex append instances later on that do not fit into this schema for simple types, there are indeed a few examples of type classes that provide useful functionality in the simple way outlined above. One of these, which was implemented as part of this thesis, is the type class for decidable equality:

```
module Class.DecEq where

open import Data.Bool using (Bool)
open import Data.Char using (Char)
open import Data.Nat using (ℕ)
open import Data.String using (String)
open import Data.Unit using (⊤)

open import Relation.Binary.Core

record ≟Class (A : Set) : Set where
  field
    _≟_ : Decidable {A = A} _≡_

open ≟Class {{...}} public
```

```
≟?InstanceBool : ≟?Class Bool
≟?InstanceBool = record { _≟?_ = Data.Bool._≟?_ }
≟?InstanceChar : ≟?Class Char
≟?InstanceChar = record { _≟?_ = Data.Char._≟?_ }
≟?Instanceℕ : ≟?Class ℕ
≟?Instanceℕ = record { _≟?_ = Data.Nat._≟?_ }
≟?InstanceString : ≟?Class String
≟?InstanceString = record { _≟?_ = Data.String._≟?_ }
≟?Instance⊤ : ≟?Class ⊤
≟?Instance⊤ = record { _≟?_ = Data.Unit._≟?_ }
```

3.2 Polymorphic Types

Even though type classes for simple types are already quite useful, many functions are parametrized by types and thus do not fit into the schema outlined above. Since frequently used data types like Data.List and Data.Colist belong to this group, we would like to be able to define type classes for operations on these data types as well. The append function is again a good example, because there already exist definitions for this functionality in the Data.List and Data.Colist modules, which we will now use as instances for our type class.

One simple approach is to reuse the type class definition of the append function for simple types without any changes to the type class itself. As ++Class expects an A of type Set, we have to fit Data.List into this structure. The definition of the type List in the standard library looks like this:

```
data List {a} (A : Set a) : Set a where
  [] : List A
  _::_ : (x : A) (xs : List A) → List A
```

As we can see the type List is parametrized over a type of Set a and the resulting type is itself of Set a. A type List A though would be of type Set for every A : Set, which means that it is possible to define an instance of the append function for the type List A. The resulting code looks like this:

```
++InstanceList : {A : Set} → ++Class (List A)
++InstanceList = record { _++_ = Data.List._++_ }
```

This definition does indeed work, as we can see in a more complete example:

```
module AppendPolymorphic where

open import Data.String using (String)
open import Data.List using (List; _::_; [])

record ++Class (A : Set) : Set where
  field
    _++_ : A → A → A
  infixr 5 _++_
```

```
open ++Class {{...}} public

++InstanceString : ++Class String
++InstanceString = record {_++_ = Data.String._++_}

++InstanceList : {A : Set} → ++Class (List A)
++InstanceList = record {_++_ = Data.List._++_}

foobar = "foo" ++ "bar"
foobar2 = ('a' :: []) ++ ('a' :: [])
```

There is a problem though: The above definition only works for lists of small types, which are only a subset of all lists. A type class for the append function should be able to provide the same functionality as the implementation in Data.List itself, not just some reduced version that works similar to the original.

The necessary adjustment to the type class is rather easy: We need to replace every occurrence of Set with Set a and provide the type class with a level a in the form of an implicit argument. Such a level argument also has to be passed to the List instance, whereas the String instance stays the same:

```
module AppendPolymorphic2 where

open import Data.String using (String)
open import Data.List using (List; _::_; [])

open import Level using (Level)

record ++Class {a} (A : Set a) : Set a where
  field
    _++_ : A → A → A
  infixr 5 _++_

open ++Class {{...}} public

++InstanceString : ++Class String
++InstanceString = record {_++_ = Data.String._++_}

++InstanceList : {a : Level} {A : Set a} → ++Class (List A)
++InstanceList = record {_++_ = Data.List._++_}

foobar = "foo" ++ "bar"
foobar2 = ('a' :: []) ++ ('a' :: [])
```

The above code provides a working type class that covers all the functionality of the original _++_ functions in Data.String and Data.List. As we can see, only the List type is polymorphic, whereas the type class itself "does not know" about any form of polymorphism; the type signature of _++_ is simply A → A → A. This is the best we can do in this case, because the instance for String prohibits us from specializing the very general type signature any further.

We could get close to a polymorphic type class through the use of multiple type class parameters. While one parameter would still be the (polymorphic) type that we want to instantiate the type class for, the other one would be the type of the elements of this polymorphic type. The type class record would then look as follows:

```
record ++Class {a} (A : Set a) (F : Set a → Set a) : Set (Level.suc a) where
  field
    _++_ : F A → F A → F A
  infixr 5 _++_
```

```
open ++Class {{...}} public
```

This hypothetical type class expects a level a as an implicit argument, which is used in the type of the next two, explicit arguments; the type A and the type constructor F. Notice how we make (A : Set a) an explicit and not an implicit argument. Even though we could make it implicit and define an instance for Data.List without any problems, there is no way for the Data.String instance to figure out which A was meant, as String is a small type and Agda would complain about unsolved metavariables. But with (A : Set a) as an explicit argument, the user can decide which type A is supposed to be *for every instance separately*. In the case of String, we can choose the Unit type ⊤, which is inhabited only by a single value. We can then instantiate the type class for the type (λ _ → String), which lifts the small type String into a type constructor. As the only possible value in place of the underscore can be the single inhabitant of the Unit type, Agda can infer this value for us. The resulting type class looks as follows:

```
module AppendPolymorphic3 where

open import Data.String using (String)
open import Data.List using (List; _::_; [])
open import Data.Unit using (⊤)
open import Level using (Level)
record ++Class {a} (A : Set a) (F : Set a → Set a) : Set (Level.suc a) where
  field
    _++_ : F A → F A → F A
  infixr 5 _++_
open ++Class {{...}} public
++InstanceString : ++Class ⊤ (λ _ → String)
++InstanceString = record { _++_ = Data.String._++_ }
++InstanceList : {a : Level} {A : Set a} → ++Class A List
++InstanceList {a} = record { _++_ = Data.List._++_ {a} }
foobar = "foo" ++ "bar"
foobar2 = ('a' :: []) ++ ('a' :: [])
```

Unfortunately this more complicated type class does not buy us anything. We can still instantiate exactly the same types as with the more general type signature A → A → A, but the instance declarations become slightly longer and more cumbersome. Even worse, the type class only suggests to be of polymorphic nature, but in reality is not, as we have to pass the parameter A for every instance, defeating the whole purpose of true parametric polymorphism. In cases where instances for simple types have to be considered as well, the best approach is to simply use a type class with a very general signature as seen earlier.

The optimal solution would be to use the latter, more specialized variant and make (A : Set a) an implicit argument. Unfortunately Strings in Agda are not directly implemented as Lists of Chars and so this approach is not a viable option. But there are a indeed a few cases of overloaded functions in Agda's standard library where such a truly polymorphic type class is possible. One example is the Length class: The length function is defined for both Lists and Colists and it returns results of type ℕ for Lists and Co$\mathbb{N}$ for Colists. Because both List and Colist are truly polymorphic, we can move {A : Set a} directly into the signature of length and make it an implicit argument:

```
module Class.Length where

open import Data.List using (List)
open import Data.Colist using (Colist)

open import Level using (Level; suc)

record lengthClass {a} {N : Set} (F : Set a → Set a) : Set (suc a) where
  field
    length : {A : Set a} → (F A) → N

open lengthClass {{...}} public

lengthInstanceList : {a : Level} → lengthClass List
lengthInstanceList {a} = record {length = Data.List.length {a}}

lengthInstanceColist : {a : Level} → lengthClass Colist
lengthInstanceColist {a} = record {length = Data.Colist.length {a}}
```

One obvious drawback is that the above type class makes it impossible to define an instance for String. At the moment Agda's standard library does not provide any function to directly calculate the length of a string, so for simple overloading the above type class works well enough. To account for such instances later on, it is probably a better idea though to make this type class non-polymorphic as well. In the end, truly polymorphic type classes only make sense if the particular type class really *depends* on the instances to be polymorphic and an implementation simply does not make sense otherwise. During the work on this thesis, no such case came up.

3.3 Dependent Types

The type classes in the last two sections have demonstrated that comfortable overloading of names is indeed possible in Agda, but all of the functionality so far is available in non-dependently typed languages as well. The situation becomes more interesting when we consider true dependent types like vectors (a Vec A n is a sequence of length n with elements of type A), which are very similar to classic lists and thus exhibit many functions with the same names and behaviors as their List counterparts.

The append function is once again a good example: The append functions on vectors works exactly as the append function on lists, but it gives additional guarantees in form of more precise types, namely the promise that the size of the resulting vector will be the same as the sum of the sizes of the two input vectors, or as an Agda type signature:

```
_++_ : ∀ {a m n} {A : Set a} → Vec A m → Vec A n → Vec A (m + n)
```

Of course this guarantee should still remain intact if we extend the append type class of the previous section with an instance for the Vec data type. It is immediately obvious that the polymorphic type signature of the append class defined in the previous section does not give us this invariant. That is why we need to adjust the type class declaration accordingly and add more specific types:

```
record ++Class {f} (I : Set) (_+_ : I → I → I)
  (F : I → Set f) : Set (Level.suc f) where
  field
    _++_ : {n : I} {m : I} → F n → F m → F (n + m)
  infixr 5 _++_
open ++Class {{...}} public
```

Apart from an implicit level argument, the type class expects three explicit arguments: and index type I, a binary function _+_ over that index type and the dependent type F that constitutes both arguments and result type of the append function. In the case of the instance for Data.Vec, the type Vec is used for argument F, the type ℕ is used as I and the addition function Data.Nat._+_ is used as _+_:

```
++InstanceVec : {a : Level} {A : Set a} → ++Class ℕ _+_ (Vec A)
++InstanceVec {a} = record {_++_ = Data.Vec._++_ {a}}
```

For the instances of Data.List and Data.String we can use the same trick as in the last section: Since Unit is only inhabited by one value, it can be used as a "dummy" index for List and String and Agda can infer the value automatically. Even better, it is even possible to infer the whole binary function _+_ for us, because there is only one function of type $\top \to \top \to \top$. This keeps the instances rather succinct:

```
++InstanceList : {a : Level} {A : Set a} → ++Class ⊤ _ (λ _ → List A)
++InstanceList {a} = record {_++_ = Data.List._++_ {a}}

++InstanceString : ++Class ⊤ _ (λ _ → String)
++InstanceString = record {_++_ = Data.String._++_}
```

Examples of type classes with dependently typed signatures are abundant in the classes that were implemented in this thesis. One of these is the Drop class, which implements overloading of the drop function for the types Colist, List, Stream and Vec. The instance for Stream looks a bit different from the rest as the drop function in the standard library is only defined for streams of small types, but apart from that fact the implementation is straightforward:

```
module Class.Drop where

open import Data.Colist using (Colist)
open import Data.List using (List)
open import Data.Stream using (Stream)
open import Data.Vec using (Vec)

open import Data.Unit using (⊤)
open import Data.Nat using (ℕ; _+_)

open import Level using (Level; suc)

record dropClass {a} (I : Set) (_+_ : ℕ → I → I)
  (F : Set a → I → Set a) : Set (suc a) where
  field
    drop : {A : Set a} → (m : ℕ) → {n : I} → F A (m + n) → F A n

open dropClass {{...}} public

dropInstanceColist : {a : Level} → dropClass ⊤ _ (λ A n → Colist A)
dropInstanceColist {a} = record {drop = λ m → Data.Colist.drop {a} m}

dropInstanceList : {a : Level} → dropClass ⊤ _ (λ A n → List A)
dropInstanceList {a} = record {drop = λ m → Data.List.drop {a} m}

dropInstanceStream : dropClass ⊤ _ (λ A n → Stream A)
dropInstanceStream = record {drop = λ m → Data.Stream.drop m}

dropInstanceVec : {a : Level} → dropClass ℕ _+_ Vec
dropInstanceVec {a} = record {drop = Data.Vec.drop {a}}
```

3.4 Special Type Classes

All of the type classes described above were used to implement overloading and contained just one field per record: the overloaded function. Even though type classes of such a form are already highly useful, there are a some uses for type classes that go beyond this structure. Several of these examples shall be described in the following sections: Type classes with multiple related functions and type classes as a way to represent categorical constructors (such as Functors, Monads and Applicative Functors).

Afterwards another special type class will be presented: The Show class that can be used to convert many different Agda types to Strings. There we can see why a recursive instance search like in Haskell can sometimes be very desirable.

3.4.1 Type classes with multiple functions

All the type classes discussed so far were used to introduce simple overloading of single functions. Sometimes additional functions can be directly derived from each instance without providing another instance, the negation of the member function _∈_ is an example. This function can be easily implemented as a function directly in the type class record that by using the _∈_ function and negating it:

```
record ∈Class {a} (I : Set) (F : Set a → I → Set a) : Set (suc a) where
  field
    _∈_ : {A : Set a} → A → {n : I} → (F A n) → Set a
  infixr 5 _∈_
  _∉_ : {A : Set a} → A → {n : I} → (F A n) → Set a
  _∉_ x xs = ¬ (x ∈ xs)
  infixr 5 _∉_
open ∈Class {{...}} public
```

But sometimes this easy and elegant approach of placing the additional function directly in the type class record does not work because of slightly differing types. One example is the zipWith Class which "zips together" a sequence of elements A and a sequence of elements B into a sequence of elements C using a function of type A → B → C. The resulting type class record reflects this very general structure and requires 4 different explicit arguments as a result:

```
record zipWithClass {a} {b} {c} (I : Set) (F : Set a → I → Set a)
  (G : Set b → I → Set b)
  (H : Set c → I → Set c)
  : Set (suc a ⊔ suc b ⊔ suc c) where
  field
    zipWith : {A : Set a} {B : Set b} {C : Set c} {n : I} →
      (A → B → C) → F A n → G B n → H C n
open zipWithClass {{...}} public
```

The zip function is a special case of the zipWith as it zips 2 sequences together into a sequence of pairs. This function can not be implemented as part of the record, because the type class records' third parameter is of the general type Set c → I → Set c, which could be completely independent from Set a and Set b as used in F and G.

As a result we need to implement the zip function outside of the record – with the unfortunate effect that all the parameters of the type class record and an additional instance argument for the record have to be explicitly given in the type signature:

```
zip : ∀ {a b} {I : Set}
  {F : Set a → I → Set a}
  {G : Set b → I → Set b}
  {H : Set (a ⊔ b) → I → Set (a ⊔ b)}
  {{z : zipWithClass I F G H}}
  {A : Set a} {B : Set b} {n : I} → F A n → G B n → H (A × B) n
zip = zipWith _, _
```

The above code is a good example for the increased verbosity of dependently typed languages like Agda. Even though the actual implementation of the zip function is simple, Agda expects a type signature, which is quite hideous in this case.

3.4.2 Type Classes for Categories

As we have seen, type classes in Agda are built out of standard record types and values. This has the advantage that we can "reuse" existing records in the standard library whenever these already act as a structure similar to a type class. A good example are categorical constructors such as Monad, Functor and Applicative Functor. There already exist records in Agda's standard library which describe the general structure of these constructs as well as record instances for a few specific types. Because Agda's instance search automatically looks for suitable instances in the whole scope and context, we just need to bring these structures and their instances into scope and open them using instance arguments to get fully working type classes. An example is the Monad type class:

```
module Class.Monad where

open import Category.Monad using (RawMonad; module RawMonad)
open import Category.Monad.Identity public using (IdentityMonad)
open import Data.Maybe public using () renaming (monad to MaybeMonad)
open import Data.List public using () renaming (monad to ListMonad)

open RawMonad {{...}} public
```

The first import statement in the above code brings the type class record into scope which describes the structure of every Monad. It is opened using the usual **open** TypeClassRecord {{...}} **public** syntax. The next three import statements bring three different instances of the Monad type class into scope, renaming them if necessary.

Of course it is also possible to augment the above approach with a few custom instances for types which are instances of the type class as well, but so far lack a suitable record value in the standard library We can see an example in the Functor type class, where we can reuse the existing maybeFunctor, but have to provide custom instances for List and Colist:

```
module Class.Functor where

open import Category.Functor using (RawFunctor; module RawFunctor)
open import Data.Maybe public using () renaming (functor to maybeFunctor)
open import Data.List using (List)
```

```
open import Data.Colist using (Colist)
open import Level using (Level)
open RawFunctor {{...}} public
functorInstanceList : {a : Level} → RawFunctor {a} (List {a})
functorInstanceList {a} = record { _<$>_ = λ f l → Data.List.map f l}
functorInstanceColist : {a : Level} → RawFunctor {a} (Colist {a})
functorInstanceColist {a} = record { _<$>_ = λ f l → Data.Colist.map f l}
```

3.4.3 A Type Class for Show

One central aim of this thesis and the accompanying Prelude is to make IO-related programming easier in Agda. This will usually mean command line applications, simply because Agda does not yet provide a wealth of libraries that would make it suitable for a lot of other tasks. Writing a command line application usually involves writing to standard out as a response to command line arguments or standard input. Unfortunately Agda's standard library does not currently offer any function to convert arbitrary Agda types into a human readable string, which would be crucial to make the development of simple programs less tedious.

The Show type class provides such a function. The class itself is extremely simple and just specifies a show function which converts the instance type into a string. Instances have been implemented for 13 different (simple) types.

```
record showClass {a} (A : Set a) : Set a where
  field
    show : A → String
```

The interesting aspect of the Show class is not the class itself though, but rather the problems that arise for polymorphic types. A show function for simple types is definitely useful, but ultimately we would like to be able to convert lists or vectors into strings as well, provided that the type of the elements is an instance of the Show type class as well.

A function to show lists (or vectors) can indeed be written in Agda. The implementation relies on Agda's instance search to find a suitable instance of the Show class to convert the elements of the list (or vector) into a string.

```
showList' : {a : Level} {A : Set a} {{s : showClass A}} → List A → String
showList' [] = "]"
showList' (x :: xs) = ", " ++ show x ++ (showList' xs)
showList : {a : Level} {A : Set a} {{s : showClass A}} → List A → String
showList [] = "[]"
showList (x :: xs) = "[" ++ show x ++ showList' xs
```

But if we then try to provide this showList function as an instance of the Show class, we hit the wall of Agda's search. To find the correct instance for List would mean to not only find the showList function itself, but then also to recursively find instances for all the elements (which could turn into quite a long recursive chain for lists of lists of lists, etc.). As mentioned earlier, Agda's instance argument resolution is non-recursive, providing showList alone without a Show instance for List is the best we can do at the moment.

```
  -- the following does not work with the current instance search:
showInstanceList : {a : Level} {A : Set a} {{s : showClass A}} → showClass (List A)
showInstanceList {a} {A} {{s}} = record {show = showList {a} {A} {{s}}}

test : String
test = show ('a' :: [])
```

Chapter 4

Structure of the Prelude

Based on the type class mechanism discussed in the previous chapter, the Prelude module re-exports a number of data types and type classes to make Agda programming a bit more comfortable. This chapter presents the structure of the Prelude in detail. However, the source code presented here is not identical to the real Prelude.agda module, as the presentation here has been stripped of the **using** and **hiding** directives which are used to selectively import particular names. Including these long lists of names would only lengthen the presentation without offering any considerable benefit to the reader. As an example, the line

```
open import Data.List public
```

corresponds to the following line in the real Prelude:

```
open import Data.List public using (List; []; _::_; intersperse; takeWhile; dropWhile)
```

Apart from these directives, all the comments have been removed as well. The aim of this chapter is to give an overview on the contents of the Prelude, not to replicate the real Prelude line by line. The module starts with import statements for a few basic modules which most Agda programs are likely to need.

```
module Prelude where

open import Function public
open import Level public
open import Size public
```

The above code is followed by a few lines that import IO-related modules. The module IO.System was written as part of this thesis and provides a simply wrapper around Haskell's getArgs function.

```
open import Coinduction public
open import IO public
  -- experimental
open import IO.System public
```

Next up are the import statements for the most common data types. The list contains many types that can be found in non-dependently typed programming languages as well, but also includes types that are distinctive for dependently typed languages, such as Data.Vec and Data.Fin.

```
open import Data.Bool public
open import Data.Char public
open import Data.Colist public
open import Data.Empty public
open import Data.Fin public
open import Data.List public
open import Data.List.Any public
open import Data.Maybe public
open import Data.Nat public
open import Data.Product public
open import Data.String public
open import Data.Sum public
open import Data.Unit public
open import Data.Vec public
```

Because Agda's type system can be used as a very powerful logic, we also include different relations to export some of the most frequently used proof-related functions and types.

```
open import Relation.Binary.Core public
open import Relation.Binary.PropositionalEquality public
open import Relation.Nullary public
```

All the modules up to this point have been part of Agda's standard library. The following lines import several type classes that were implemented as part of this thesis.

```
open import Class.Functor public
open import Class.Monad public
open import Class.Append public
open import Class.Concat public
open import Class.DecEq public
open import Class.DecSetoid public
open import Class.DecTotalOrder public
open import Class.Drop public
open import Class.Eq public
open import Class.Foldl public
open import Class.Foldr public
open import Class.FromList public
open import Class.Length public
open import Class.Lines public
open import Class.ListLiteral public
open import Class.Map public
open import Class.Member public
open import Class.Null public
open import Class.Replicate public
open import Class.Reverse public
open import Class.Setoid public
open import Class.Show public
open import Class.StrictTotalOrder public
open import Class.Take public
```

```
open import Class.Unlines public
open import Class.ZipWith public
```

The Prelude ends with a few helper functions that have proven valuable during the implementation of the example programs discussed in the next chapter. These will hopefully be moved into more suitable modules in the near future, but are defined directly in the Prelude right now.

```
charToℕ : Char → Maybe ℕ
charToℕ '0' = just 0
charToℕ '1' = just 1
charToℕ '2' = just 2
charToℕ '3' = just 3
charToℕ '4' = just 4
charToℕ '5' = just 5
charToℕ '6' = just 6
charToℕ '7' = just 7
charToℕ '8' = just 8
charToℕ '9' = just 9
charToℕ _ = nothing

stringToℕ' : List Char → (acc : ℕ) → Maybe ℕ
stringToℕ' [] acc = just acc
stringToℕ' (x :: xs) acc = charToℕ x >>= λ n → stringToℕ' xs $ 10 * acc + n

stringToℕ : String → Maybe ℕ
stringToℕ s = stringToℕ' (toList s) 0
```

This is already the end of the whole Prelude. As we can see, the Prelude itself is quite short, in contrast to Haskell's Prelude for example. The guiding principle was that the Prelude itself should only bundle functionality from the standard library so that every programmer can also choose not to use the Prelude and import hand-picked modules (and type classes) instead.

Chapter 5

Example Usage of the Prelude

This chapter demonstrates the use of the Prelude in two simple Agda programs. Both are heavily IO-oriented and require a lot of different data types from the standard library. The Prelude is able to act as a "drop-in" replacement for all the manual import statements and manages the otherwise occurring name clashes automatically through type classes.

5.1 TabsToSpaces

The first example of the Prelude at work will be a simple command-line application which reads in a file and writes out the same file with all tabs converted to a specified number of spaces. Even though such a tool can be useful from time to time, Agda is definitely not the best programming language for this kind of task. Quite the opposite in fact: any program where most of the data comes from the "outside world" and is repeatedly written back without any kind of complex computation does not take advantage of the dependent types that Agda offers. Most of the time will be spent parsing the data, where Agda's type system cannot be of much help. So why attempt to write such a program in Agda, when it is obvious that it is going to be a bad fit? Exactly *because* this is Agda's biggest weakness: If the Prelude can make regular and mundane IO related programming easy or at least bearable, it will have a good chance to be of help when some input and output is needed for truly useful programs. Right now Agda programs are often only type-checked and not compiled, because this additional step requires overcoming the hurdle of communicating with the outside world. If the formalization of some system is the only purpose of the program, this may be enough. But considering that Agda explicitly tries to be a *programming language* as well as a proof assistant, the desire to embed formalized systems in a thin wrapper of IO seems reasonable.

Let us start the TabsToSpaces program with the module declaration and an import of the Prelude. This will remain the only import statement that we need, since the Prelude provides all commonly used functions in a comfortable package.

```
module TabsToSpaces where
open import Prelude
```

Next up is the heart of the conversion; the tabsToSpaces function. This function is only a wrapper around the tabsToSpaces′ function and converts between Strings and a List of Chars. The tabsToSpaces′ function itself is simple: It reads the List of Chars

character by character and replaces every match `'\t'` with `' '` replicated n times. Notice how the function uses the ++ operator to concatenate two Lists. Later on we will use the same operator to do String concatenation, made possible by the type class exported in the Prelude.

```
tabsToSpaces′ : (spaces : ℕ) → List Char → List Char
tabsToSpaces′ n [] = []
tabsToSpaces′ n (x :: xs) with x
... | '\t' = (replicate n ' ') ++ (tabsToSpaces′ n xs)
... | _    = x :: tabsToSpaces′ n xs

tabsToSpaces : (spaces : ℕ) → String → String
tabsToSpaces n s = fromList $ tabsToSpaces′ n $ toList s
```

With the conversion in place, we can now focus on the IO part of the program, first of all the parsing of the command line arguments:

```
takeExactly : ∀ {a} {A : Set a} (n : ℕ) → Colist A → Maybe (Vec A n)
takeExactly zero []          = just []
takeExactly zero l           = nothing
takeExactly (suc n) []       = nothing
takeExactly (suc n) (x :: xs) = _::_ x <$> takeExactly n (♭ xs)
```

The above function expects a colist of arguments and a number of arguments and returns a Vector of exactly this length if possible. This allows us to "cross the barrier" between coinductive data types and finite dependent types, which can be easily pattern matched later on.

Together with the helper function try_error?_ok?_ the main function can now be successfully defined. Because Agda's standard library provides two different IO types (one is the primitive IO type of Haskell, whereas the other one is a coinductive deep embedding of that type), the main′ function has to be called using the run function from main. Because the _ >>= _ function of the IO monad uses additional coinduction annotations, it has the signature {B : Set a} (m : ∞ (IO B)) (f : (x : B) → ∞ (IO A)) → IO A. As we can see, the function f returns a value of type ∞ (IO A), whereas the result type of the whole _ >>= _ function is just IO A. Hence Agda's IO monad cannot form an instance of the Monad type class and thus the Prelude renames the IO monad's _ >>= _ to _∞>>=_. Furthermore Agda does not provide any do notation like Haskell, as a result the main′ functions looks at times a bit clunky:

```
  -- A variant of Data.Maybe.maybe′ with argument ordering suited for long code
try_error?_ok?_ : ∀ {a b} {A : Set a} {B : Set b} → Maybe A → B → (A → B) → B
try (just x) error? error ok? ok = ok x
try nothing error? error ok? ok = error

main′ : IO ⊤
main′ = ♯ getArgs ∞>>= λ args →
  try takeExactly 3 args
  error? ♯ (putStrLn $ "exactly 3 arguments required:\n"
             ++ "    1. input file\n"
             ++ "    2. output file\n"
             ++ "    3. number of spaces")
  ok? λ {(infile :: outfile :: numspc :: []) →
```

```
            try stringToℕ numspc
            error? ♯ (putStrLn "The third argument needs to be a number!")
            ok? λ n →
                ♯ (♯ (readFiniteFile infile) >>= λ input →
                  ♯ (writeFile outfile ∘ unlines ∘
                     map (tabsToSpaces n) ∘ lines) input)
            }
main : _
main = run main′
```

Notice how we can pattern match on the command line arguments using the pattern (infile :: outfile :: numspc :: []). This is possible, because the call to takeExactly 3 args guarantees that the resulting vector will have length 3 (or be nothing).

Furthermore ++ can now be used to do string concatenation without having to worry about possible clashes with the usage of ++ further above for List concatenation. This kind of overloading keeps the code much cleaner than the use of explicit imports, especially in the case of mixfix operators, because otherwise Data.String._++_ would have to been used in prefix form.

The complete code can be compiled and works as expected. The following command line interaction shows how to convert a file containing tabs into a file containing 2 spaces in place of tabs:

```
> cat input
this is
        a file
                with lots
                of
        wonderful
tabs!
> ./TabsToSpaces input output 2
> cat output
this is
  a file
    with lots
    of
  wonderful
tabs!
```

5.2 PrefixCalculator

The second example will be a bit more complex than the first: a calculator that uses a Lisp-like syntax to add, subtract or multiply natural numbers. Once again this program is not exactly an indispensable companion regarding day-to-day tasks, but it is a lot closer to the kind of IO related Agda program that might be considered useful. Both the lexer and parser functions could be used in a simple Lisp interpreter without any changes, only the eval function would have to be substituted with a real Lisp evaluator. If such a Lisp was statically typed, an implementation could definitely benefit from Agda's rigorous types to implement the type system directly in Agda and prove certain properties about it. The following program therefore constitutes a first step towards an arguably useful Lisp interpreter in Agda.

Once again we begin the program with a module declaration and only one import statement for the Prelude (again, this will remain the only import statement that we need), but this time we must include an additional primitive to enable sized types [Abe10] [Abe06] and hide the existing List data type that is exported by the Prelude. The reason is that we need to define our own List data type that uses sized types to satisfy the termination checker later in the program.

```
{-# OPTIONS –sized-types –show-implicit #-}
module PrefixCalculator where
open import Prelude hiding (List; []; _::_)
```

The definition of the List data type exactly mirrors the normal definition in Data.List, albeit with additional implicit arguments for the size. Such an extension allows a limited form of subtyping, which can be helpful to reassure the termination checker that an expression involving Lists will indeed terminate. We will see an example later on.

Together with the definition of our custom List type, we will also define some functions to convert this type to- and from the normal List type exported by the Prelude:

```
infixr 5 _::_
data List {a : Level} (A : Set a) : {size : Size} → Set a where
  [] : {i : Size} → List A {↑ i}
  _::_ : {i : Size} → A → List A {i} → List A {↑ i}
toCustomList : {A : Set} → Prelude.List A → List A
toCustomList Prelude.[] = []
toCustomList (Prelude._::_ x xs) = x :: (toCustomList xs)
toOrigList : {A : Set} → List A → Prelude.List A
toOrigList [] = Prelude.[]
toOrigList (x :: xs) = Prelude._::_ x (toOrigList xs)
```

A reverse function will be useful later on as well. Since the Prelude already exports a type class for reverse, we only need to provide a record containing the custom reverse function to implement a new instance of the type class. Agda's instance search will automatically find the record when it is in context (such as in the whole following program for example).

```
reverse′ : ∀ {a} {A : Set a} → List A → List A → List A
reverse′ acc []       = acc
reverse′ acc (x :: xs) = reverse′ (x :: acc) xs
reverseInstanceCustomList : {a : Level} → reverseClass ⊤ (λ A n → List A)
reverseInstanceCustomList {a} = record {reverse = reverse′ {a} []}
```

Unfortunately all the code so far has had nothing to do with the problem domain itself, but was needed to reimplement already existing functionality in the context of sized types. This is of course not an ideal situation and will hopefully improve in the future. Sized types are still a rather experimental feature that is not used in the data types of the standard library. As Agda matures, this will probably change so that the Prelude can directly export a List type that uses sized types.

Next up is the tokenizer of the calculator. The function tokenize expects a list of chars and returns a list of tokens. Each token is either a left parenthesis "(", a right

parenthesis ")" or a symbol consisting of a string. Both parentheses and the character ' ' are treated as delimiters of tokens.

To split the list of chars into a list of tokens, we will keep all the visited characters in an accumulator argument and add this accumulator as a symbol to our result whenever we read a delimiter. Because the tokenizer could read a delimiter with no previously visited characters in the accumulator, we also need the helper function _::T_ that will make sure to add the accumulator only as a symbol if it is not empty.

```
data Token : Set where
  lPar   : Token
  rPar   : Token
  symbol : String → Token

infixr 4 _::T_
_::T_ : List Char → List Token → List Token
[]  ::T tokens = tokens
tok ::T tokens = (symbol ((fromList ∘ toOrigList ∘ reverse) tok)) :: tokens

tokenize′ : List Char → List Char → List Token
tokenize′ tok ('(' :: rest) = tok ::T (lPar :: (tokenize′ [] rest))
tokenize′ tok (')' :: rest) = tok ::T rPar :: (tokenize′ [] rest)
tokenize′ tok (' ' :: rest) = tok ::T (tokenize′ [] rest)
tokenize′ tok (c :: rest)   = tokenize′ (c :: tok) rest
tokenize′ tok []            = tok ::T []

tokenize : List Char → List Token
tokenize = tokenize′ []
```

After having defined the tokenizer, we need to parse the list of tokens into a data type of nested lists, in the Lisp world commonly called an S-Expression. A legal SExp is either an atom (such as a natural number or an operator symbol), an empty list nil or an SExp "consed" onto another SExp. Together with the data type for S-Expressions, we will also define a function _ ▷ _ to add an SExp at the end of an existing SExp.

```
data SExp : Set where
  atom : String → SExp
  nil  : SExp
  cons : SExp → SExp → SExp

_ ▷ _ : SExp → SExp → SExp
(atom x)  ▷ y = (cons (atom x) y)
nil       ▷ y = (cons y nil)
(cons x xs) ▷ y = (cons x (xs ▷ y))
```

Now follows the heart of the parser, which is split in two functions: the helper function parse′, which handles all the heavy work, and parse itself, which handles the initial cases and then calls parse′. parse′ expects to be called after an opening parenthesis has been read and tries to parse as many tokens as possible until a closing parenthesis is read, calling itself recursively in the process. It takes two input arguments; an accumulator consisting of the already parsed SExp and a list of tokens that still have to be parsed. The result type is a pair of the successfully parsed SExp and the possibly remaining tokens, wrapped in an Either type to allow the possibility of a parse error. This is where our custom List with sized types comes into play: The additional {i}

annotation assures the termination checker that the resulting list of tokens is at most as long as the input list. The different cases itself are straightforward:

- Since the function is called after reading an opening parenthesis, only a closing parenthesis constitutes a valid end-token, that is why reading [] results in an error.
- If the function comes across another opening parenthesis before, it has to call itself again, add the resulting SExp to the accumulator (if successful) and continue with the remaining tokens.
- Encountering a closing parenthesis marks a successful termination, the accumulated SExp and the remaining tokens are returned.
- Whenever a symbol is read, it can be added at the end of the accumulator and the parsing continues with the next token.

```
Error = String
parse′ : {i : Size} → SExp → List Token {i} → Either (SExp × (List Token {i})) Error
parse′ acc [] = right "unexpected end of expression"
parse′ acc (lPar :: xs) with parse′ nil xs
... | right e          = right e
... | left (exp, [])   = left $ acc ▷ exp, []
... | left (exp, rest) = parse′ (acc ▷ exp) rest
parse′ acc (rPar :: rest)       = left (acc, rest)
parse′ acc (symbol y :: rest) = parse′ (acc ▷ (atom (toString y))) rest
```

The above helper function can now be called by parse. Here we also handle the case that the input consists only of one token without any parenthesis, which could be a valid expression, if it is a number. (Or any literal in a real Lisp interpreter)

```
parse : List Token → Either SExp Error
parse []                = right "invalid expression"
parse (symbol y :: [])  = left (atom (toString y))
parse (symbol y :: rest) = right "invalid expression"
parse (lPar :: rest) with parse′ nil rest
... | right e           = right e
... | left (exp, [])    = left exp
... | left (exp, _)     = right "invalid expression"
parse (rPar :: rest)    = right "invalid expression"
```

Compared to the parsing, the eval function is rather simple and can be defined very concisely thanks to Agda's pattern matching syntax. We only handle two valid cases: An expression is either a valid number in form of an atom or it is a cons expression built out of an operator atom op and two expressions a and b. The operator can be one of "+","-" and "*", in which case the corresponding functions from the Agda standard library are called. Everything else is an invalid expression.

```
eval : SExp → Either ℕ Error
eval (atom x) with stringToℕ x
...          | just n  = left n
...          | nothing = right $ "not a number: " ++ x
```

```
eval (cons (atom op) (cons a (cons b nil))) with eval a | eval b | op
... | left x | left y | "+" = left (x + y)
... | left x | left y | "-" = left (x - y)
... | left x | left y | "*" = left (x * y)
... | right e | _ | _ = right e
... | _ | right e | _ = right e
... | _ | _ | e = right $ "unknown operator '" ++ e ++ "'"
eval _ = right "invalid expression"
```

The Read-Eval-Print-Loop, or "repl" for short, brings all of the previously defined functions together. It first converts a String to a List Char, then converts this list to our custom list data type, reads it and finally parses the resulting tokens. Because both tokenizing and parsing can result in an error, these possibilities are handled using pattern matching. In case of success, the resulting number is converted back to a String using the show function.

```
repl : String → String
repl s with (parse ∘ tokenize ∘ toCustomList ∘ toList) s
repl s | left exp with eval exp
repl s | left exp | left n = "= " ++ show n
repl s | left exp | right e = "error: " ++ e
repl s | right e = "error: " ++ e
```

The main′ function reads input from standard in, splits it into a list of lines, maps the repl function onto each of these lines (this means that it is not possible to write calculations that span multiple lines), concatenates the resulting lines back into a Costring and writes this Costring to standard out. The main function simply lifts the main′ function into Agda's IO monad.

```
main′ : _
main′ = ♯ getContents ∞>>= λ input →
        ♯ (putStrLn∞ ∘ unlines ∘ (map repl) ∘ lines) input
main : _
main = run main′
```

Once again we can compile and type-check the Agda program to verify that our Prefix Calculator indeed works as expected:

```
> ./PrefixCalculator
5
= 5
(+ (* 2 3) (+ 1 3))
= 10
(- 15 (* 3 5))
= 0
(+ 2
error: unexpected end of expression
(5)
error: invalid expression
(foo 3 4)
error: unknown operator 'foo'
```

Chapter 6

Conclusion

Even though dependently typed languages are often more complex and verbose than their non-dependently typed counterparts, the implementation of type classes in Agda is nevertheless still surprisingly manageable. Type classes that will be instantiated only for non-dependent types can be implemented almost exactly like their Haskell counterparts, the only significant difference being the explicit record passing in Agda. This approach can often be an advantage, because standard records can be reused (as seen in the Monad and Functor type classes for example). Agda's instance arguments constitute a very elegant method of decoupling the record passing from the instance search and allow for example multi-parameter type classes "out of the box", which require a special extension in Haskell.

Type classes for dependent types are only slightly more complex. Using the $\top$ type as a "dummy index" for non-dependent types allows the definition of type classes which handle dependent and non-dependent instances equally well. The only disadvantage of this approach is the greater verbosity compared to non-dependent type classes, but the extra parameters are hidden from the end user using implicit arguments and only come into play whenever new instances have to be defined.

With the help of type classes it is possible to eliminate all of the most frequently occurring name clashes in Agda. Using the different kinds of type classes outlined in this thesis, it should be possible to eliminate most (if not all) of the remaining name clashes as well. The example usages of the Prelude in the two presented programs have shown that the Agda Prelude developed here can indeed be of great help in the development of Agda programs. Not only have all the import statements been subsumed by the Prelude, the manual management of name clashes using qualified names has disappeared as well.

One considerable problem in the current situation has been the lack of a recursive instance search mechanism. Even though name clashes can be resolved just fine using the current instance search, the Show type class has shown that recursive search can be a necessity for certain instances. This is an area where Agda will hopefully improve in the near future.

Overall, the Prelude has accomplished its objectives. It is not yet clear of how much value it can be for experienced Agda programmers, but it should at least be helpful for Agda newcomers. The functionality exported by the Prelude is of course still quite small, but there is no inherent problem that would prevent it from growing into an even more valuable companion.

Bibliography

[Abe06] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[Abe10] Andreas Abel. Miniagda: Integrating sized and dependent types. In *In Partiality and Recursion (PAR*, 2010.

[DNM+09] N. Danielsson, U. Norell, S. Mu, S. Bronson, D. Doel, P. Jansson, and L. Chen. The agda standard library, 2009.

[DP11] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 143–155, New York, NY, USA, 2011. ACM.

[dt04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[KS04] Oleg Kiselyov and Chung-chieh Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 33–44, New York, NY, USA, 2004. ACM.

[McB02] Conor McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, July 2002.

[Mcb05] Conor Mcbride. The epigram prototype: a nod and two winks, 2005.

[McK06] James McKinna. Why dependent types matter. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 1–1, New York, NY, USA, 2006. ACM.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[Nor07] Ulf Norell. Towards a practical programming language based on dependent type theory, 2007.

[OS08] Nicolas Oury and Wouter Swierstra. The power of pi. *SIGPLAN Not.*, 43(9):39–50, September 2008.

[P+03] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

[Tur85] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.