A Functional Programming Language with Patterns and Copatterns

Shams A. Alkhulaif

Submitted in partial fulfilment
of the requirements for the degree of

Master of Science

Department of Computer Science
Brock University
St. Catharines, Ontario

Abstract

Since the emergence of coinductive data types in functional programming languages, various languages such as Haskell and Coq tried different ways in dealing with them. Yet, none of them dealt with coinductive data types properly. In lazy languages such as Haskell, both inductive data types and coinductive data types are gathered and mixed in one list. Moreover, some languages such as Coq used the same constructors that are used for inductive data types as a tool to tackle coinductive data types, and while other languages such as Haskell did use destructors, they did not use them properly. Coinductive data types behave differently than inductive data types and therefore, it is more appropriate to deal with them differently. In this thesis, we propose a new functional programming language where coinductive data types are dealt with in a dual approach in which coinductive data types are defined by observation and inductive data types are defined by constructors. This approach is more appropriate in dealing with coinductive data types whose importance comes from their role in creating a safer and more sophisticated software.

Acknowledgements

I would like to express my profound and sincere gratitude to my supervisor, Dr. Michael Winter for his supervision, continuous and endless guidance, advice, support, motivation, encouragement from the very beginning of this research work, and for helpful comments on the text. I am also grateful for suggesting the topic of the thesis, reading my numerous revisions and giving me lots of valuable comments. I am truly fortunate to have Dr. Winter as my supervisor, as his knowledge and leadership have allowed me to draw inspiration to overcome future obstacles, and develop goals. Without his encouragement, guidance and support from the initial to the final level this thesis would not have been possible. I am indebted to him more than he knows.

I would like to thank my supervisory committee members Dr. Brian Ross and Dr. Beatrice Ombuki-Berman, Graduate Program Director, as well as external examiner Dr. Gunther Schmidt for their support, guidance and helpful suggestions. Their guidance has served me well and I owe them my heartfelt appreciation.

I would like also to thank my family starting with my exceedingly generous parents who encouraged my continued academic studies, my parents who supported me from afar as much as they had previously done from close by. I would like to show my appreciation and to thank my supportive husband for his help, encouraging, and all the continuous support he has given me during my years of graduate studies at Brock University. I thank him for believing in me, and for sharing my wish to reach the goal of completing this task.

I extend my gratitude to my sponsors at Al-Jouf University as well as the The Saudi Arabian Cultural Bureau who helped me all the way regardless of the circumstances I have been through.

I thank Brock University and its staff that helped with hardware and software support, and Kevin Bacon for the original template of this thesis.

In the end, I like to dedicate my endeavour and my thesis to my son in anticipation and hope that one day he will do a better job than his Mom did.

Contents

List of Figures

Chapter 1

Introduction

When dealing with recursive data types in functional programming languages we may distinguish two main kinds, inductive and coinductive. In inductive data types, the elements of the data type are finitely generated which means they can be defined by constructors. Constructors are basically the mechanism an inductive data type provides to define finite data. For example, the elements of the data type $List\ a$ are finite lists with elements from a. Any list can be constructed starting from the empty list by adding a new head element finitely many times. If we denote by $[\,] : List\ a$, the empty list, and by cons : $a \to List\ a \to List\ a$, the constructor that produces a new list from an already given list with an additional head element, then every list can be written using the constructors $[\,]$ and cons. The concept of inductive data types or finite data is dealt with in many functional languages such Haskell and others. However, while inductive data types are very clearly explained and constructed in the main implementation fields of functional programming languages, one may notice that the case is not the same when it comes to infinite data or coinductive data types.

On the contrary from finite data types as the name would indicate, elements of coinductive data types are infinite in nature. For example, the data type $Stream\ a$ of streams of a elements is the type of all infinite lists with elements from a. Elements of such a data type cannot be defined by constructors, i.e. they cannot be generated by applying constructors finitely many times. Therefore, infinite data types are defined by observation. Destructors as a dual concept to constructors provide a convenient mechanism to define infinite data types by observation. For example, the codata type $Stream\ a$ can be defined by using the two destructors hd and tl, returning the head element respectively the tail of a stream. Just as constructors define data type elements uniquely, destructors define codata type elements uniquely. For example, a stream has the number 1 as a head and a tail equal to itself is uniquely determined,

i.e., it is the infinite sequence of 1's. Infinite data may play an important role in the creation of more sophisticated and safer software. Our desire is to establish behavioural and liveness properties by understanding infinite data more clearly and less vaguely.

1.1 Infinite Data in Haskell

The concept of infinite data being dually different from finite data has been missing in many languages that tried to deal with infinite data types; which had led to treating infinite data types in an improper way. The lack of this understanding was a major reason why some of these languages dealt with infinite data in an ad-hoc manner. For instance, in some lazy languages such as Haskell, there is only one kind of lists which includes both finite and infinite lists, i.e. it is the combination of lists and streams. In other words, Haskell does not have infinite data types in a separate form and so it includes them in list data type. Functions with a parameter of a data type are usually defined by pattern matching. A pattern is a term constructed from variables and constructors, i.e., a term that describes how the element was constructed. Haskell provides top-level pattern matching as implicit program construction, i.e., pattern matching can be used, for each parameter of a function, by providing separate definitions for each pattern. Consider the example:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:l) = (fx):(map f l)
```

The first line defines the type of the new function map. Please note that lower case letters denote type variables in Haskell which are implicitly quantified. Therefore, the map is a polymorphic function that takes two parameters. The first parameter is a function mapping elements from a to elements of b ($a- > b$), and the second parameter is a list with elements from a ($[a]$). Finally map returns a list of elements from b ($[b]$). The following two lines define the function by pattern matching on its second parameter. The second line defines map in the case that the list is the empty list ([ ]), and the third line defines map in the case that the list is constructed using cons (:).

In Haskell, it is easy to define functions that compute the same result as destructors of the corresponding codata type. For example, the destructors hd and tl of streams can be implemented in Haskell as [3]:

```
hd :: [a] -> a
hd (x:_) = x

tl :: [a] -> [a]
tl (_:l) = l
```

In this syntax, the first part (the head) is meant to extract the first element of an infinite list, and the second part (the tail) is to extract the elements after the head. In other words, the head would take the first element in a non-empty list, while the tail takes everything else but the head. Please note that hd and tl are only partially defined. They are not defined for the empty list, i.e., a defining statement for the constructor [ ] is missing in both definitions. Furthermore, hd and tl are just regular functions and cannot be used in pattern matching. A dual concept to pattern matching is not available in Haskell.

1.2 Infinite Data in Dependently-Typed Languages

In other dependently-typed languages such as Coq or Agda, dealing with infinite data is not any better. While the proof assistant Coq, for instance, does have coinductive types, it uses constructors to define infinite data. In Coq, coinductive streams are defined as follows:

```
CoInductive Stream A :=
| Cons : A -> Stream A -> Stream A.
```

Please note that coinductive type in Coq represents infinite elements that are not finitely generated by constructors but still uses constructors to define them. As an example we have the following two definitions of the stream 0,0,0,... and 1,1,1,... in Coq [1]:

```
CoFixpoint zeroes : Stream nat := Cons nat 0 zeroes.
CoFixpoint ones : Stream nat := Cons nat 1 ones.
```

Similarly to Haskell, functions have to be defined using pattern matching. As an example, consider the following definition of map for streams. Please note that Coq

does not provide implicit top-level pattern matching. Instead it uses an explicit match statement. The destructors hd and tl can be defined analogously to Haskell. However, in Coq these functions are total since the stream coinductive type does not have a constructor for the empty list [1].

```
CoFixpoint map {A B} (f: A -> B) (s : Stream A) : Stream B :=
match s with
| Cons _ h t => Cons _ (f h) (map f t)
end.
```

In Coq, it is possible to define the (dependent) type of proofs that two given streams are equal. Since streams are coinductive, this type has to be coinductive as well. The type defines two streams to be equal if the heads and the tails are equal [1].

```
CoInductive stream_eq {A} : Stream A -> Stream A -> Prop :=
| Stream_eq : forall x1 x2 t1 t2, x1 = x2 ->
stream_eq t1 t2 -> stream_eq (Cons _ x1 t1) (Cons _ x2 t2).
```

Now, we can prove that the two streams ones and map S zeroes are equal. Please note that proving a property in Coq is just defining a function of the appropriate type. The Coq language just provides suitable tactics implementing a proof language, or system, creating the required function.

```
Lemma Test1 : stream_eq (map S zeroes) ones.
Proof.
cofix Hyp.
assumption.
Qed.

Lemma Test2 : stream_eq (map S zeroes) ones.
Proof.
(*cofix Hyp.
assumption.*)
cofix Hyp.
replace ones with
((fun s => match s with Cons _ n s' => Cons _ n s' end) ones)
by (destruct ones; trivial); simpl.
replace (map S zeroes) with
((fun s => match s with Cons _ n s' => Cons _ n s' end)
(map S zeroes)) by (destruct (map S zeroes); trivial); simpl.
constructor.
trivial.
assumption.
Qed.
```

What we want to do is to show that ones and map S zeroes (S is successor) are equal. The Cofix is the dual of induction. However, after doing Cofix, the hypothesis and the goal are equal and after using the assumption, the system says that there is nothing left to show. Also, closing the proof with Qed does not work which is due to the fact that the hypothesis is about the tails of the two streams which happen to be the same as the whole streams. So, using the assumption is not correct. Therefore, in order to get it right we have to "unfold" ones and map S zeroes first so that both terms start with a constructor. This is necessary since the conclusion of Stream_eq requires the two streams to be in that form. This "unfolding" can only be done by a trick (as shown above) Then we can use the constructor Stream_eq and finish. If streams would be defined by destructors, then Stream_eq would become:

```
Stream_eq : forall s1 s2, hd s1
= hd s2 -> stream_eq (tl sq) (tl s2) -> stream_eq s1 s2.
```

Here, there is no need to "unfold" the two streams since the conclusion of Stream_eq does not require the terms to be in a specific form. Moreover, "in the Calculus of (Co)Inductive Constructions, the core theory underlying Coq, coinduction is broken, since computation does not preserve types". [4, 9, 15]. In Agda for instance, one can never mix inductive and coinductive types in a compositional way according to Abel and colleagues. In other words, while encoding the property "infinitely often" from temporal logic is possible, its dual "eventually forever" is not possible. The common fact of all these languages is simply that none of them seems to be dealing with infinite data or coinductive data types in the best way possible.

1.3 Our Approach

In our language, we want to avoid all problems indicated above when dealing with finite and infinite data. The first major problem to be treated is the problem of mixing finite and infinite data types. As their properties are different, they both should have been differently defined and treated. Therefore, this problem must be fixed by separating them from each other. The other major problem is that constructors are always used to define infinite data when observation is what should have been used to define it instead of giving its value directly. Therefore, in our language:

- We will have finite and infinite data types separated. This way will assure us a more proper way of defining them.
- Since we separated them from each other, finite elements of data types will be defined by constructors using pattern matching and infinite elements of codata types will be defined by destructors (destructors) using copattern matching.

Chapter 2

Our Language

As a functional programming language, our language will have both finite and infinite data. The main difference to other functional programming languages is the treatment of infinite data. Therefore, when dealing with finite data, our language will not look very different from Haskell. This way we can focus more on the creation of a language that deals best with coinductive data types.

2.1 Lists

As indicated above, dealing with finite data will not be very different from the way languages like Haskell deal with it. For example, in our language, lists are defined as follows:

```
data List a {
empty : List a;
cons  : a -> List a -> List a;
}
```

In Haskell, the list is defined as follows:

```
data [a] =
[]
| (:) a [a]
```

It is worth mentioning that the list data type in Haskell is actually implemented internally without an explicit definition in Haskell itself for efficiency reasons. However, the internal implementation is equivalent to the one given above.

In both languages, the definition starts with the keyword data. Then, the name of the new data type and its potential parameters are listed, i.e., *List* a, $[a]$ respectfully. This is followed by the constructors for the new data type. The only difference here is that in Haskell we do not need to mention the return type of the constructors. For example, the constructor [ ] in Haskell takes no parameters (constant) and is of type $[a]$ by definition. Similarly, the infix constructor : takes parameters of type a and $[a]$ and returns $[a]$ by definition. In our language, we have decided that the full type of every constructor has to be provided with the requirement that its return type is equal to the type that we are currently defining.

2.2 Streams

When it comes to coinductive types with infinite data, our language would differ from other languages that tried to deal with infinite data. As mentioned before, many existing languages use constructors for infinite data as well. In our language, destructors are used to define infinite data which cannot be defined by constructors. In other words, they will not be constructed from the empty set, rather, they will be defined by **how they behave**. For example, if we look at the head of an infinite list that we have, we will get an element while we also can look at the remaining list of the tail. This is the way infinite objects are described. In one word, *observation*. In our language, we define streams as follows:

```
codata Stream a {
hd : Stream a -> a;
tl : Stream a -> Stream a;
}
```

As shown in the definition, infinite data (codata) is dealt with as another data type with a different mechanism to define it (destructors).

2.3 Functions

After data and codata types are defined, the next type of data to be defined is *Functions*. Functions are the kinds of data that allow us to work on the data types we have. In his book *The Haskell School of Expression*, Hudak wonders "what should the type of a function be?" before he defines it as "It seems that it should at least convey the fact that a function takes values of one type ... as input and returns values

of (possibly) some other type ... as output"[13]. This definition explains the type of functions and what it basically does. In our language, data and codata types are separated, and the mechanisms used to define finite and infinite data (constructors and destructors) are not the same. Considering that, just as constructors cannot be used for finite and infinite data, functions that deal with finite data cannot be the same as those dealing with codata types. Therefore, functions as well will be defined in accordance to the data or codata type.
Let us take a map function for instance. In our language, the map function for lists is defined as follows:

```
map : forall a b, (a -> b) -> List a -> List b;
map = fun (f : a -> b) (l : List a) =>
match l with
empty       => empty;
cons x l1   => cons (f x) (map f l1);
end
end;
```

This map is used with finite lists as it is dealing with empty lists. It permits us to compensate for the missing eliminations for positive types [4]. This will not work properly when dealing with streams as it is using **Pattern Matching** which is about mapping elements of a list to elements of another. In streams where data is infinite this will not work.
In this definition, the map is defined for empty lists, which makes it not effective for streams. Therefore, we defined what we call *mapS* as follows:

```
mapS : forall a b, (a -> b) -> Stream a -> Stream b;
mapS = fun (f : a -> b) (s : Stream a) =>
comatch as Stream b by
hd _ => f (hd s);
tl _ => mapS f (tl s);
end
end;
```

The *function mapS* uses what we call **Copattern Matching** which is more about heads and tails than it is about individual elements. Instead of compensating for the missing eliminations for positive types, **Copatterns** are compensating for the missing introductions for negative types.

2.4 Copattern Matching Vs. Pattern Matching

Upon dealing with data, patterns appear in many areas such as lambda abstractions, function definitions, pattern bindings, list comprehensions, do expressions, and case expressions [17]. Yet, as we proposed a dual approach in dealing with finite and infinite data types, it is important to deal with them dually as well as matching their patterns and copatterns. When dealing with finite data, *pattern matching*, which is comparing an observed pattern with an expected pattern to determine whether they match or not, is to be used [29]. This concept of pattern matching is used by languages, such as Haskell, in a proper way. However, when it comes to infinite data types, there is no dual tool such as pattern matching being used as mentioned earlier. Therefore, we used *copattern matching* which was proposed by Abel et al. where instead of values being matched against patterns, evaluation contexts are being matched against copatterns [4]. According to Abel et al., copatterns may be regarded as a *definition scheme* for infinite data types where it could be seen as *experiments* on blackbox infinite objects such as functions and streams [4]. Pairing a copattern with its defining outcome is what is known as a covering observation, and a finite set of those observations is what defines infinite objects [4]

2.5 Syntax of The Grammar of Program

2.5.1 Backus Naur Form

Backus Naur Form is a notation of the scheme of syntactic definition of ALGOL 60 which was proposed by John Backus in 1959 [21]. This proposed scheme made clear distinctions between Syntax and Semantics which were two of the *semiotics* proposed by Charles Morris in his book *Signs, Language, and Behavior [21, 20].* BNF grammar also does agree with the Context-free Grammar of Noam Chomsky [21]. Moreover, according to *The Computer Science and Communications Dictionary* Backus Naur form is "A metalanguage used to specify or describe the syntax of a language in which each unique symbol represents a single set of symbol strings. Common abbreviation BNF."[30, 16]. Therefore, we chose to use BNF to describe the grammar of our language as it is used to describe a language using a context free grammar.

Table 2.1: Notational Conventions

Macro name	Description
<Syntax>	Nonterminal is denoted by surrounding symbol with <>
Syntax	Terminal is unadorned symbol
|	Alternation is denoted by | (choice)
'Syntax'	Use as is
::=	Replacement is denoted by ::= (the production)
(Syntax)	For grouping, for a list (must pick one)

While the right hand side (rhs) of the production is the sequence of terminal and non-terminal symbols to the right of the ::=, the left hand side (lhs) of the production is the non-terminal symbol to the left of the ::=. Since it is a context free grammar, it is defined by a set of productions where the lhs is to be replaced by the rhs anywhere lhs exists despite the context. This is what makes it context free [7].

2.5.2 Application of Backus Naur Form in the Grammar of our Language

In this section, we use BNF to describe the syntax of the grammar of our language as it is used to describe a language using a context free grammar. And before showing the syntax, we presented some notational conventions that are used for presenting the syntax in Table 2.1.

Program Grammar

$$\begin{aligned} \langle Program \rangle \quad &::= \langle Declaration \rangle \\ &\mid \langle Declaration \rangle \langle Program \rangle \end{aligned}$$

Declaration Grammar

$$\begin{aligned} \langle Declaration \rangle \quad &::= \langle TypeDeclaration \rangle \\ &\mid \langle TermDeclaration \rangle \end{aligned}$$

Type Declaration Grammar

⟨*TypeDeclaration*⟩ ::= ⟨*TypeData*⟩
| ⟨*TypeCoData*⟩

⟨*TypeData*⟩ ::= '`data`' ⟨*Name*⟩ ⟨*ListTypeVariable*⟩ '`{`' ⟨*ListConstructor*⟩ '`}`'

⟨*TypeCoData*⟩ ::= '`codata`' ⟨*Name*⟩ ⟨*ListTypeVariable*⟩ '`{`' ⟨*ListDestructor*⟩ '`}`'

⟨*ListTypeVariable*⟩ ::= ⟨*Empty*⟩
| ⟨*TypeVariable*⟩ ⟨*ListTypeVariable*⟩

⟨*ListConstructor*⟩ ::= ⟨*Empty*⟩
| ⟨*ConstructorDefinition*⟩ ⟨*ListConstructor*⟩

⟨*ListDestructor*⟩ ::= ⟨*Empty*⟩
| ⟨*DestructorDefinition*⟩ ⟨*ListDestructor*⟩

⟨*ConstructorDefinition*⟩ ::= ⟨*Name*⟩ ':' ⟨*TypeBody*⟩ ';'

⟨*DestructorDefinition*⟩ ::= ⟨*Name*⟩ ':' ⟨*TypeBody*⟩ ';'

⟨*Empty*⟩ ::= nil

⟨*Name*⟩ ::= String

Type Grammar

⟨*Type*⟩ ::= ⟨*TypeBody*⟩
| ⟨*TypePolymorphic*⟩

⟨*TypeBody*⟩ ::= ⟨*TypeVariable*⟩
| ⟨*TypeFunction*⟩
| ⟨*TypeDefined*⟩
| ⟨*TypeApply*⟩

⟨*TypePolymorphic*⟩ ::= '`forall`' ⟨*ListTypeVariable*⟩',' ⟨*TypeBody*⟩

⟨*TypeVariable*⟩ ::= ⟨*Identifier*⟩

⟨*TypeFunction*⟩ ::= ⟨*TypeBody*⟩ '`->`' ⟨*TypeBody*⟩

⟨*TypeDefined*⟩ ::= ⟨*Identifier*⟩

⟨*TypeApply*⟩ ::= ⟨*TypeBody*⟩ ' ' ⟨*TypeBody*⟩

⟨*Identifier*⟩ ::= ⟨*Letter*⟩
| ⟨*Letter*⟩⟨*Identifier*⟩
| ⟨*Identifier*⟩⟨*Digit*⟩
| ⟨*Name*⟩

⟨*Letter*⟩ ::= $a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z \mid A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid Z$

⟨*Digit*⟩ ::= $0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Term Declaration Grammar

⟨*TermDeclaration*⟩ ::= ⟨*Name*⟩ ':' ⟨*Type*⟩';'
⟨*Name*⟩ '=' ⟨*Term*⟩';'

Term Grammar

⟨*Term*⟩ ::= ⟨*TermVariable*⟩
| ⟨*TermFunction*⟩
| ⟨*TermDefined*⟩
| ⟨*TermMatch*⟩
| ⟨*TermCoMatch*⟩
| ⟨*TermApply*⟩

⟨*TermVariable*⟩ ::= '('⟨*Identifier*⟩':'⟨*Type*⟩')'

⟨*TermFunction*⟩ ::= 'fun' ⟨*ListTermVariable*⟩ '=>' ⟨*Term*⟩ 'end'

⟨*TermDefined*⟩ ::= ⟨*Identifier*⟩

⟨*TermMatch*⟩ ::= 'match' ⟨*Identifier*⟩ 'with'
⟨*ListCase*⟩
'end'

⟨*TermCoMatch*⟩ ::= 'comatch as' ⟨*TypeBody*⟩ 'by'
⟨*ListCoCase*⟩
'end'

⟨*TermApply*⟩ ::= ⟨*Term*⟩ ' ' ⟨*Term*⟩
| '('⟨*Term*⟩ ' ' ⟨*Term*⟩')'

⟨*Case*⟩ ::= ⟨*Constructor*⟩ '=>' ⟨*Term*⟩ ';'

⟨*CoCase*⟩ ::= ⟨*Destructor*⟩ '_'' =>' ⟨*Term*⟩';'

⟨*ListCase*⟩ ::= ⟨*Case*⟩
| ⟨*Case*⟩ ⟨*ListCase*⟩

⟨*ListCoCase*⟩ ::= ⟨*CoCase*⟩
| ⟨*CoCase*⟩ ⟨*ListCoCase*⟩

⟨*Constructor*⟩ ::= ⟨*Name*⟩

⟨*Destructor*⟩ ::= ⟨*Name*⟩

⟨*ListTermVariable*⟩ ::= ⟨*Empty*⟩
| ⟨*TermVariable*⟩ ⟨*ListTermVariable*⟩

2.5.3 Parse tree

"A parse tree is a tree-based notation for describing the way a sentence could be built from a grammar"[7]. An example of a context free grammar using BNF for data type declaration is the following syntax tree:

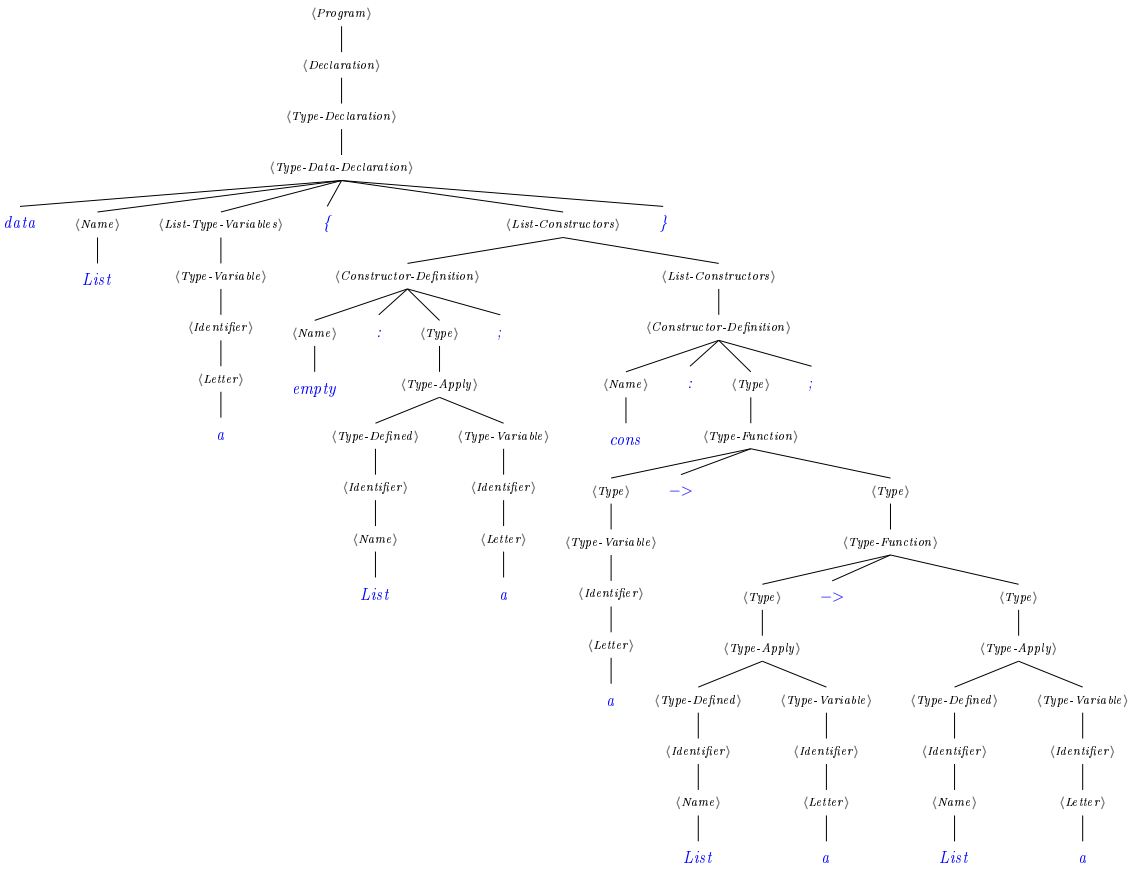

Figure 2.1: Data Parse Tree

This data type declaration in this example is the result of the parsing tree above:

```
data List a {
empty : List a;
cons  : a -> List a -> List a;
}
```

Another example of a context free grammar using BNF for codata type declaration is the following syntax tree:

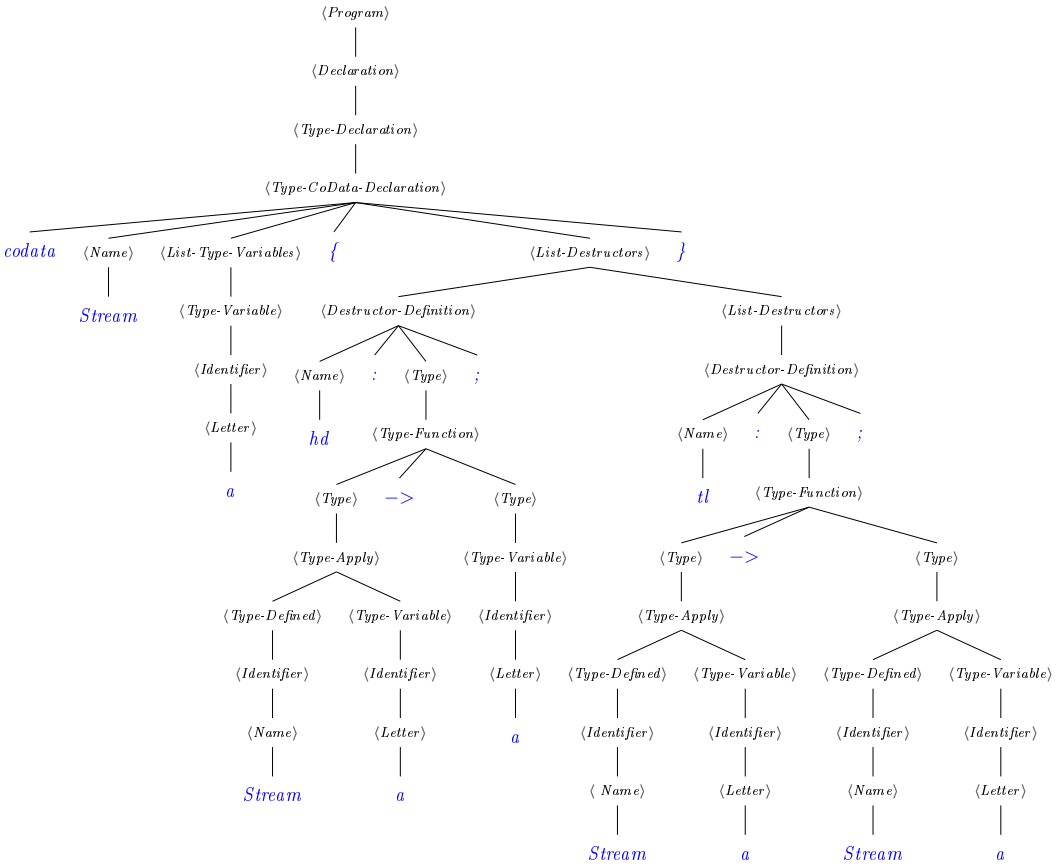

Figure 2.2: CoData Parse Tree

This codata type declaration in this example is the result of the parsing tree above:

```
codata Stream a {
hd : Stream a -> a;
tl : Stream a -> Stream a;
}
```

Generally, parse trees and their derivations have one main difference. This difference is that the parse tree, or syntax tree, does not state the order in which some of the derivations are derived [7]. However, in what is called *leftmost derivation,* where the left most non-terminal is always replaced first [7], that difference between parse trees and their derivations appears to be non-existent.

2.6 Typing Rules

The typing rules are based on a context Γ. Such a context is a list of declarations of the form $x : A$ where x is a variable and A is a type. If $x : A$ is in Γ, it is assumed that x has type A. All typing rules are based on such a Γ.

A typing judgement has the form $\Gamma \vdash t : A$ where Γ is a context, t is a term, and A is a type. It states that under the assumptions in Γ, i.e., that the variables have the type as listed, the term t has type A. We present the typing rules in the form:

$$\frac{A_1 \cdots A_n}{C}$$

where A_1,..., A_n and C are typing judgements. The rule says that if we already derived the judgements $A_1, \ldots, A_n$, then we may conclude the judgement C. Please note that in the special case $n = 0$, i.e., the number of judgements above the line is 0, the rule simply says that a certain term always has the given type in the given context. Please note that while some typing rules such as *TypeFunction* and *TypeApply* may exist in other languages [22, 23], *TypeCoData* is particularly written for our language.

2.6.1 Typing Rules

To define the typing rules we assume that D is a data type of kind $\underbrace{\star \to \cdots \to \star}_{n+1 \text{ times}}$ taking n parameters, and:
$cons_i$: `forall` $A_1 \cdots A_n,\ B_1 \to \cdots \to B_m \to D \quad A_1 \cdots A_n$ is a constructor of D,

and that C is a codata type of kind $\underbrace{\star \to \cdots \to \star}_{n+1 \text{ times}}$ taking n parameters and

$destr_i$: `forall` $A_1 \cdots A_n :\ C \quad A_1 \cdots A_n \to D$ is a destructor of C.

The typing rules of our language are as follows:

$$\Gamma_1, x : A,\ \Gamma_2 \vdash x : A \qquad \text{(TypeVariable Rule)}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\ t : B} \qquad \text{(TypeApply Rule)}$$

$$\frac{\Gamma,\ x : A\ \ \vdash\ t : B}{\Gamma \vdash \text{fun } x : A \texttt{ => } t \texttt{ end} : A \to B} \qquad \text{(TypeFunction Rule)}$$

$$\frac{\Gamma \vdash t : D \quad A_1 \cdots A_n \qquad \Gamma,\ x_1 : B_1, \cdots, x_m : B_m \vdash t_i : C}{\begin{array}{c}\Gamma\ \vdash\ \texttt{match}\ t\ \texttt{with} \\ cons_i\ x_1 \cdots x_m \texttt{ => } t_i \\ \texttt{end} : C\end{array}} \qquad \text{(TypeData Rule)}$$

$$\frac{\Gamma \vdash\ t_i : D_i}{\begin{array}{c}\Gamma\ \vdash\ \texttt{comatch as}\ C\quad A_1\ \cdots\ A_n\ \texttt{by} \\ destr_i\ \texttt{\_ => } t_i \\ \texttt{end} : C \quad A_1 \cdots A_n\end{array}} \qquad \text{(TypeCoData Rule)}$$

Figure 2.3: Typing Rules

2.6.2 Reduction Rules

As usually we denote by $t[t'/x]$ the substitution of the term t' for every free occurrence of x in t. Please note that this may require to rename bounded variables in t in order to avoid binding free variable in t' during substitution.
Reduction rules are the mechanism for executing a functional program. The reduction rules include the regular β-reduction and two rules related to matching and comatching. Informally, a program reduces if a constructor (destructor) meets a match (comatch) statement. The rules are as follows:

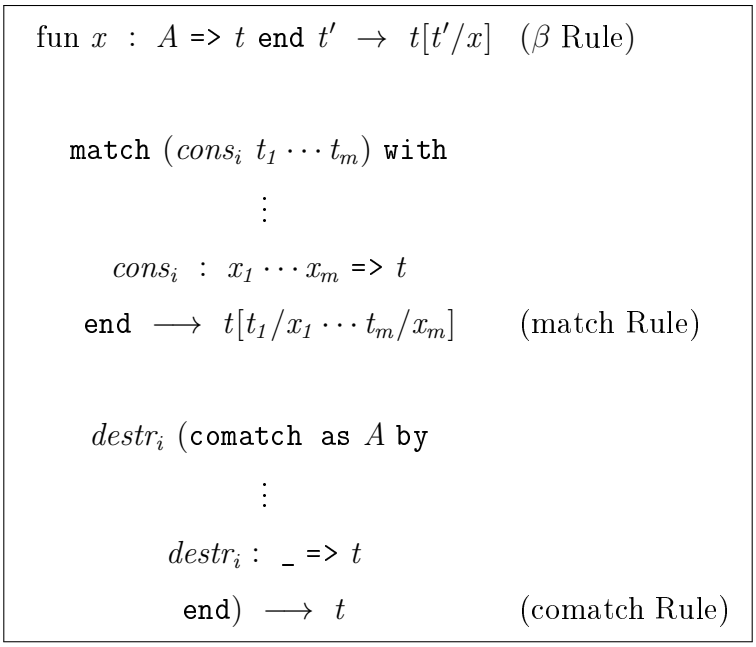

Figure 2.4: Reduction Rules

Chapter 3

Implementation

The parsing and interpretation of our language was done using *JPARSEC* as a parser [2]. *JPARSEC* is "Java library initially developed to implement a complete set of algorithms to compute ephemerides" that was meant to improve scientific research [28]. As an object oriented language, Java is relatively fast, cross-platform and cross-architecture language. the abstraction in Java, where simple or complex programs are implemented and kept easy to maintain and scale, is offered at a high level. These features of Java were the reason why the developer of *JPARSEC* has chosen it to be the language of his project [28]. Moreover, Java has a rich type system and flexible subclass mechanism and is without primitive types [19]. This helps Java avoid having problems such as those with higher-order interpretation of matching, particularly in the context of bounded quantification that pure object-oriented languages with primitive types have [19].

In our language, there are three main hierarchies for Kinds, Types, and Terms that are implemented using *Composite Design Pattern*. According to Riehle, *the Composite Design Pattern* is a design pattern which is an "abstraction from a concrete recurring solution that solves a problem in a certain context", that "can be best explained as the composition of further atomic or composite patterns."[24]. Constituting patterns that are integrating with one another in order to achieve a high level of cooperation and collaboration is what gives *Composite Design Pattern* its identity that distinguishes it as more than just a sum of some patterns [24].

3.1 Kinds

3.1.1 Hierarchy of Kinds

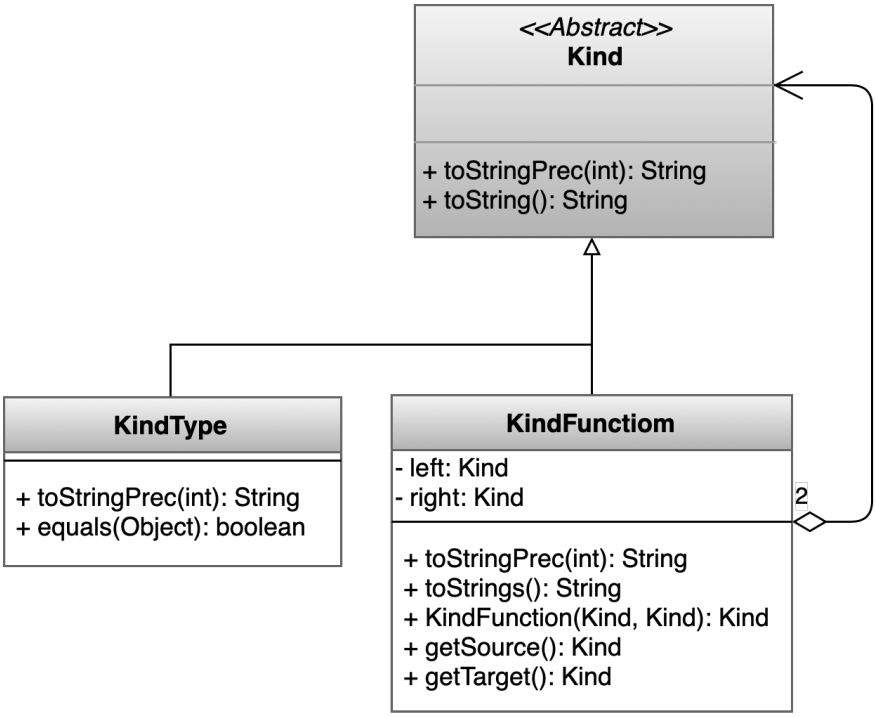

Figure 3.1: Hierarchy of Kinds

As different values and terms can be of different types, the types of these values and terms have a type, called *Kind,* as well. For instance, `Bool` and `Int` are two different types that have their own values and terms such as *True and False* of type `Bool`. The type itself is a plain type and has kind $\star$, i.e., the kind $\star$ indicates that the entity in question is a type. If we consider the list data type from the previous chapter, then List `Bool` is also a type, and, hence, has kind $\star$. On the other hand List itself is not a type. It is a type constructor, i.e., a function that maps types to types. Therefore, List has kind $\star \rightarrow \star$. Unlike types, Kinds cannot appear on the surface syntax as they are to be inferred by the compiler [5]. There are only two Kinds of types, *KindType* and *KindFunction*. *KindType* is denoted as a $\star$ and includes basic types such as `Bool, Int`, and `Char`. Each one of these types is of *Kind* $\star$. *KindFunction*, on the other hand, is more complicated. It can be where we have a function that takes a type and returns a type. *KindFunction* can also be taking two arguments of types of

KindType and returning a type of *KindType*. An example of that would be *Pair* of kind $\star \rightarrow \star \rightarrow \star$.

```
data pair a b = pair a b
```

Therefore, kinds are relatively easy when defined and here is the definition of Kind:

$$\kappa \quad ::= \star \mid \kappa \rightarrow \kappa$$

3.1.2 Implementation of Kinds

Similar to the implementation of other syntactical components of our language, kinds are implemented using the composite design pattern [10]. We start our description with the abstract class *Kind*. *Kind* here is an abstract class, which means it does not have a body. This superclass is considered a parent to the other subclasses -its children- which will provide it with the body. *Kind* has two subclasses *KindType* and *KindFunction*, which we explained earlier, as well as two methods which are *toStringPrec(int)* and *toString()*. Starting with the methods, the first method, *toStringPrec(int),* which is defined in *Kind* and implemented in its subclasses in order to convert an element of the class *Kind* into a string. The parameter *prec* is used to model precedences of the operations and place brackets accordingly. It returns an object of a string which is representing the precise integer.

Implementation:

```
    public abstract String toStringPrec(int prec);
```

The second method, *toString()*, is implemented in *Kind* by calling *toStringPrec(0)*. It returns a *toStringPrec(0)*.

Implementation:

```
    public String toString()
```

Another method, *equals(Object)*, is implemented in the obvious way for all kinds. It returns true if object and *Kind* are equal.

Implementation:

```
    public boolean equals(Object obj)
```

3.2 Types

3.2.1 Hierarchy of Types

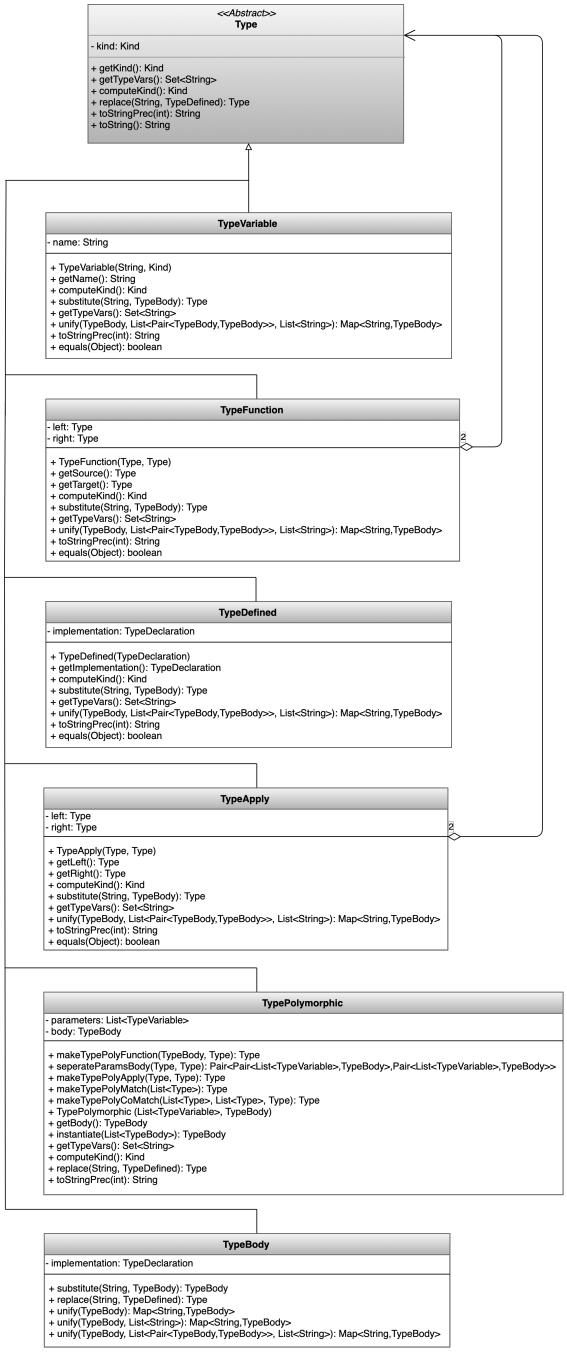

Figure 3.2: Hierarchy of Types

Similar to kinds, types include values, expressions, and/or *terms*. The type of these values is called *Type*. Although this concept looks similar to the one in *Kind*, there are still some major differences between the two. The first difference is the fact that *Types* can appear on the surface syntax as they are to be inferred by the compiler [5]. *Types* include simple types such as `Nat, Bool`, and `Char`. They also include *polymorphic* types which are the types of values and terms that can come formed in two or more ways [12]. In functional programming languages, data types can sometimes be created by the user. Yet, there would be some built-in data types like the ones mentioned earlier. In our language, there are six types of types as shown in Figure 3.2. They are *TypeVariable, TypeFunction, TypeDefined, TypeBody, TypePolymorphic,* and *TypeApply*.

TypeVariable for instance, is all the types that are considered universally quantified [18]. As explained earlier, *polymorphic* can be written in several ways, yet they are either *parametric* or *constrained*. Being *parametric* simply means it can work for any type as it is not using information that are specified to a type or a set of types, and being *constrained* means to be bounded to the set of types that have instances of its typeclasses [6]. For *TypeFunction*, a *function* in its simplest forms is when it takes values of one type and returns values of same or other types, and a *TypeFunction* is a function that can be as simple as that or even more complex yet the main concept remains the same [14].

3.2.2 Implementation of Types

When implementing the class *Type*, there are many methods as shown in Figure 3.2. However, two of them are worthy of explaining here which are **Unification and Substitution.** As for **Unification**, in general, "Unification tries to identify two symbolic expressions by replacing certain sub-expressions (variables) by other expressions"[26]. In other words, these expressions can unify a unifier (substitution of terms for variables) that makes them equal [8]. Although this is a general statement about unification, it is not so different from what it is doing in our language. **Unification** in our language computes the **most general unifier** (mgu) of two types by using John A. Robinson's algorithm [25]. In types, the method **Unification**, or as it is written in the Hierarchy of Types (Figure 3.2) *unify*, gets two variables to be the same. On the other hand, **Substitution** is "a mapping from variables to terms which is almost everywhere equal to the identity"[26]. In our language, substitution

methods are in both Types and Terms. they constantly changes expressions that they are applied to, and can be the way as which solutions of **Unification** are denoted. In other words, they replace a free type or a free term variable by a given type or term respectfully.

3.3 Terms

3.3.1 Hierarchy of Terms

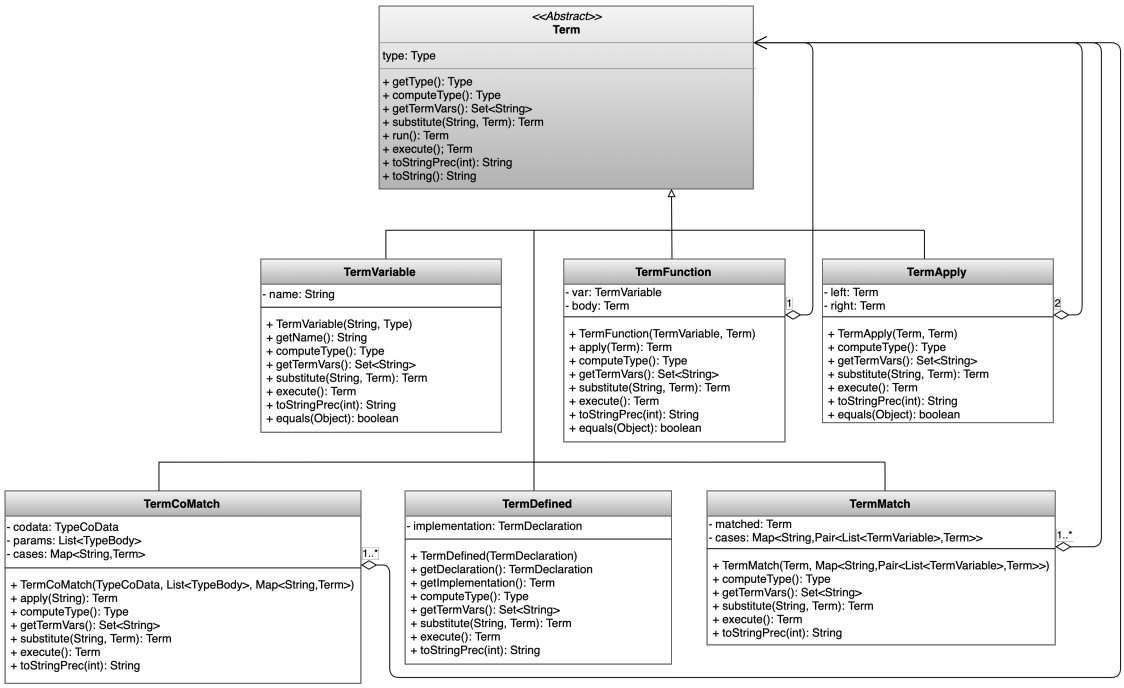

Figure 3.3: Hierarchy of Terms

Similar to the way kinds consist of types, types do consist of terms. Therefore, terms could be regarded as the blocks of types. In functional programming, terms may sometimes be used interchangeably with expressions, and although it may depend on the meaning a programmer has of terms, they are quite different. According to Granström, the word term can be defined as *an expression taken together with its meaning* [11]. Terms and expressions also can be either simple or complex, depending on their parts. Also, terms can be built from function symbols as well as variable symbols [26]. There can be expressions that have multiple meanings and these are called *Polymorphic* expressions [11]. When describing the hierarchy of terms in our

language, there are six types of terms: *TermVariable, TermFunction, TermDefined, TermMatch, TermCoMatch* and *TermApply*.

3.3.2 Implementation of Terms

Similar to types, when terms are implemented, they have many methods as shown in Figure 3.3. There are three of these methods that are of higher importance: *toString(), toStringPrec(int)*, and *substitute (String, Term): Term*.

toStringPrec(int), is defined in Term and implemented in its subclasses in order to convert an element of the class *Term*. The second method, *toString()*, is implemented in *Term* by calling *toStringPrec(0)*, returns a *toStringPrec(0)*.

3.4 Program

3.4.1 Hierarchy of Type Declaration

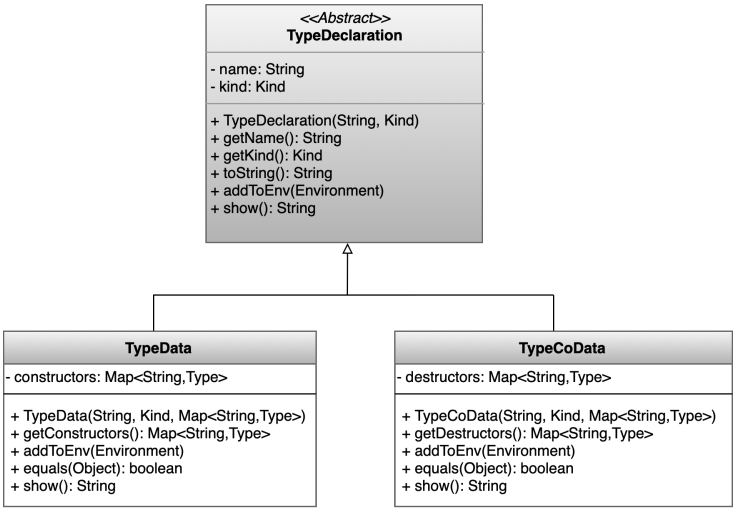

Figure 3.4: Hierarchy of Type Declaration

As we are dually treating data and codata, we have two subclasses under *TypeDeclaration* and can be seen in Figure 3.4. Its subclasses consist of *TypeData* and *TypeCoData*. While the subclass *TypeData* provides *getConstructors()* to access the constructors of this data type, the subclass *TypeCoData* provides *getDestructors()* to access the destructors of this data type. This is, of course, because data is defined by constructors and codata is defined by destructors.

3.4.2 Hierarchy of Term Declaration

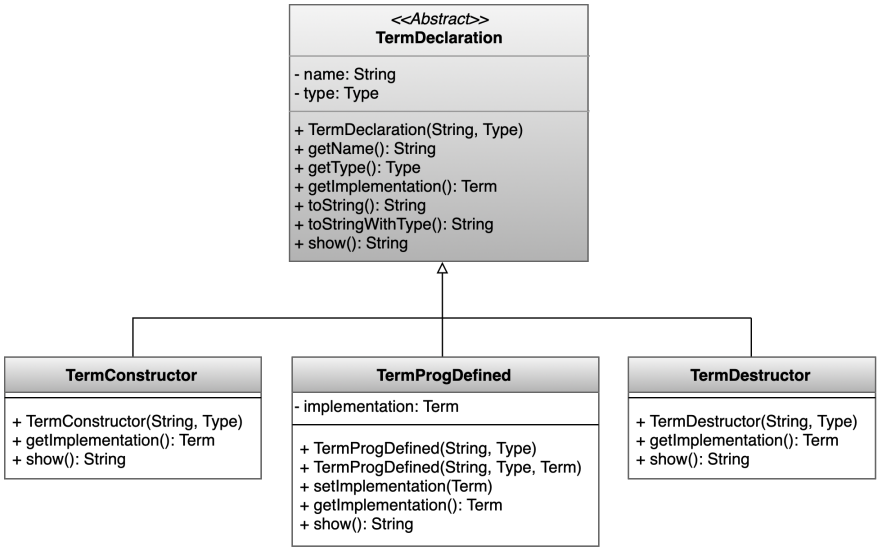

Figure 3.5: Hierarchy of Term Declaration

In term declaration, the hierarchy of Term Declaration has three subclasses as shown in Figure 3.5, which are *TermConstructor, TermDestructor* and *TermProgDefined*. As for *TermProgDefined*, it is the implementation of *Term* and is basically the main program.

Chapter 4

Execution

After the language is constructed, comes the time of execution of the program. In this chapter, we focus on the method *execute* in Terms. This method finds reductions (left hand side of a reduction rule) and replaces it by the right hand side. This is done recursively until no reduction rule can be applied. The main program will automatically execute the term main defined in a user defined program. After the execution is done, the results will be given.

4.1 Execution Rules

When executing, the program uses reduction rules (or execution rules) recursively until no reduction can be found. Being recursively applied would make it hard to detect the order in which the execution takes place. However, going over the reduction rules shown in Figure 2.4 should clarify the steps in which the program performs the reductions. Starting with the β rule,

$$\text{fun } x \ : \ A \texttt{ => } t \texttt{ end } t' \ \rightarrow \ t[t'/x]$$

if a function is applied to an argument, then one simply takes the body of the function and executes it. In this rule, the left side, until the *end*, will be applied to the parameter. However, one would get the body where the variable by a parameter is placed. This computes its left side to the right hand side.

If it is a *match*, in which case this rule will be used is:

$$\begin{array}{c} \texttt{match}\ (cons_i\ t_1 \cdots t_m)\ \texttt{with} \\ \vdots \\ cons_i\ :\ x_1 \cdots x_m\ \texttt{=>}\ t \\ \texttt{end} \longrightarrow t[t_1/x_1 \cdots t_m/x_m] \end{array}$$

and the term being matched is a term that starts with a constructor, then the corresponding line is taken out of the match statement and placed in the concrete parameter t_a all the way to t_m for the variables. Again, the left side will be constantly replaced with the right hand side until no reductions are possible.

If it is a *comatch*, in which case this rule will be used is:

$$\begin{array}{c} destr_i\ (\texttt{comatch as}\ A\ \texttt{by} \\ \vdots \\ destr_i : \texttt{\_ =>}\ t \\ \texttt{end}) \longrightarrow t \end{array}$$

and instead of starting with a *constructor* we start with a *destructor* that is applied to comatch, then the corresponding line will be taken from the comatch and placed in the variable. Therefore, a while loop runs everything within its block, which is defined by curly braces, as long as the conditional test is true. When the conditional test is false, the while loop code block will not run, and execution will move down to the code immediately after the loop block. These executions will recursively be applied.

4.1.1 Method Execute

For terms, we execute using method *execute*. Here is *execute* method in Term:

```
public abstract Term execute();
```

The method *execute* has a type *Term*, has no parameters, has no body of the method, and is simply an abstract which is different than other methods of term. *TermVariable, TermDefined, TermCoMatch* and *TermFunction* for instance have no param-

eters, and they return a Term in accordance to the initial term. Other terms are *TermMatch* and *TermApply*.

When *TermMatch* matches a *Term* and this *Term* is equal to *TermApply*, the right parameter of *TermApply* will be added to a list, then *TermMatch* will get the left parameter of *TermApply*. This loop will continue running as long as the conditional test is true. Afterwards, if the conditional test is false, execution will move to the code that comes after the loop. If *TermMatch* matches a *Term* and this *Term* is equal to *TermDefined*, and if implementation of *TermDeclaration* is equal to *TermConstructor*, then *Term* will get a list name of *TermDeclaration*, and will do a substitution for a list of *TermVariable* in *a* continuously for a loop until the conditional test is false. It will return the result of the execution of the *Term* if all the conditions are met, otherwise it will return a *Term*.

When a *TermApply* left is equal to *TermDefined* it will execute the implementation of *TermDefined*. If that is not the case, it will execute the *TermApply* on the left. If the result of the execution is equal to a *TermFunction*, will execute the right parameter of *TermApply* of *TermFunction*. Else, when the result of the execution is equal to a *TermDefined*, the implementation of the result of the execution, which is a *TermDefined*, is equal to *TermDestructor*, and the running of the *TermApply* on the right is equal to *TermCoMatch*, then the running of applying destructor to the *TermCoMatch* results. Else, it will result a new *TermApply* with the result of the running of *TermApply* as a right parameter and the result of the execution of *TermApply* left as a left parameter.

If the result of the execution and *TermDefined* are not equal or if the implementation of the result of the execution which is a *TermDefined* and *TermDestructor* are not equal, the result will be a new *TermApply* with two parameters: the result of the execution of *TermApply* left as a left parameter and execution of *TermApply* on the right as a right parameter.

At the end of execution it should return a *Term*.

The main program of our language, which is a subclass of *TermDeclaration*, is called *TermProgDefined* as shown in Figure (3.5). It has one parameter implementation of type *Term*, two constructor methods, and three other methods. The first constructor method is *TermProgDefined(String, Type)* with two parameters that have types *String* and *Type* which it got from *TermDeclaration*. The second constructor method is a *TermProgDefined(String, Type, Term)* method with three parameters

that have types *String, Type*, and *Term*. The second constructor method got its first two parameters from *TermDeclaration* while it produces the third itself. For the other methods, the first method is called *setImplementation(Term)* with one parameter of type *Term*. This method returns nothing. Another method it has is a *getImplementation()* method with no parameters. This method always returns a *Term*. The last method, is called *show()*. It does not have any parameters, and always returns a *String*.

4.1.2 Test Program

Here is the test program file that was executed which contains *TypeDeclaration* and *TermDeclaration*, and *Functions*:

```
data nat {
    zero : nat;
    succ : nat -> nat;
}

five : nat;
five = succ (succ (succ (succ (succ zero))));

data List a {
    empty : List a;
    cons  : a -> List a -> List a;
}

map : forall a b, (a -> b) -> List a -> List b;
map = fun (f : a -> b) (l : List a) =>
        match l with
            empty       => empty;
            cons x l1   => cons (f x) (map f l1);
        end
      end;
```

```
codata Stream a {
    hd : Stream a -> a;
    tl : Stream a -> Stream a;
}

mapS : forall a b, (a -> b) -> Stream a -> Stream b;
mapS = fun (f : a -> b) (s : Stream a) =>
            comatch as Stream b by
                hd _ => f (hd s);
                tl _ => mapS f (tl s);
            end
       end;

get : forall a, nat -> Stream a -> List a;
get = fun (n : nat) (s : Stream a) =>
        match n with
            zero   => empty;
            succ m => cons (hd s) (get m (tl s));
        end
    end;

zeroes : Stream nat;
zeroes = comatch as Stream nat by
           hd _ => zero;
           tl _ => zeroes;
         end;

main : List nat;
main = get five (mapS succ zeroes);
```

In this file we write a *DataDeclaration* of `Nat`, *DataDeclaration* of *List a*, and a *CoDataDeclaration* of Stream *a*. Also we write a *TermDeclaration* and function definitions for the three declarations. Every declaration in the example has constructors or destructors as the case in *CoDataDeclaration*. Every constructor has a *Type*. Also, every destructor in codata declaration has a *Type*, so head has a *Type Stream a* $\rightarrow$ *a*, and tail has a *Type Stream a* $\rightarrow$ *Stream a* which is a *TypeFunction*.

```
data nat {
    zero : nat;
    succ : nat -> nat;
}
```

```
data List a {
    empty : List a;
    cons  : a -> List a -> List a;
}
```

```
codata Stream a {
    hd : Stream a -> a;
    tl : Stream a -> Stream a;
}
```

Also we write a *TermDeclaration* for the three declarations, we write:

```
five : nat;
five = succ (succ (succ (succ (succ zero))));
```

```
zeroes : Stream nat;
zeroes = comatch as Stream nat by
           hd _ => zero;
           tl _ => zeroes;
         end;
```

```
main : List nat;
main = get five (mapS succ zeroes);
```

And for the function definitions, we write:

```
map : forall a b, (a -> b) -> List a -> List b;
map = fun (f : a -> b) (l : List a) =>
        match l with
            empty       => empty;
            cons x l1   => cons (f x) (map f l1);
        end
      end;
```

```
mapS : forall a b, (a -> b) -> Stream a -> Stream b;
mapS = fun (f : a -> b) (s : Stream a) =>
            comatch as Stream b by
                hd _ => f (hd s);
                tl _ => mapS f (tl s);
            end
       end;
```

```
get : forall a, nat -> Stream a -> List a;
get = fun (n : nat) (s : Stream a) =>
        match n with
            zero   => empty;
            succ m => cons (hd s) (get m (tl s));
        end
    end;
```

Term of program has two lines. The first line is: a name with the *Type* of the program. The second line is: the name =, and then a *Term* as shown below:

```
five : nat;
five = succ (succ (succ (succ (succ zero))));
```

After codata is defined, the next step is to define a function:

```
mapS : forall a b, (a -> b) -> Stream a -> Stream b;
mapS = fun (f : a -> b) (s : Stream a) =>
            comatch as Stream b by
                hd _ => f (hd s);
                tl _ => mapS f (tl s);
            end
       end;
```

Function *mapS* will return a stream destructed by applying a function (the first argument) to all items in a stream passed as (the second argument).

4.1.3 Main Program

In the main class, the program will be run using the main method as shown in Figure 4.1.

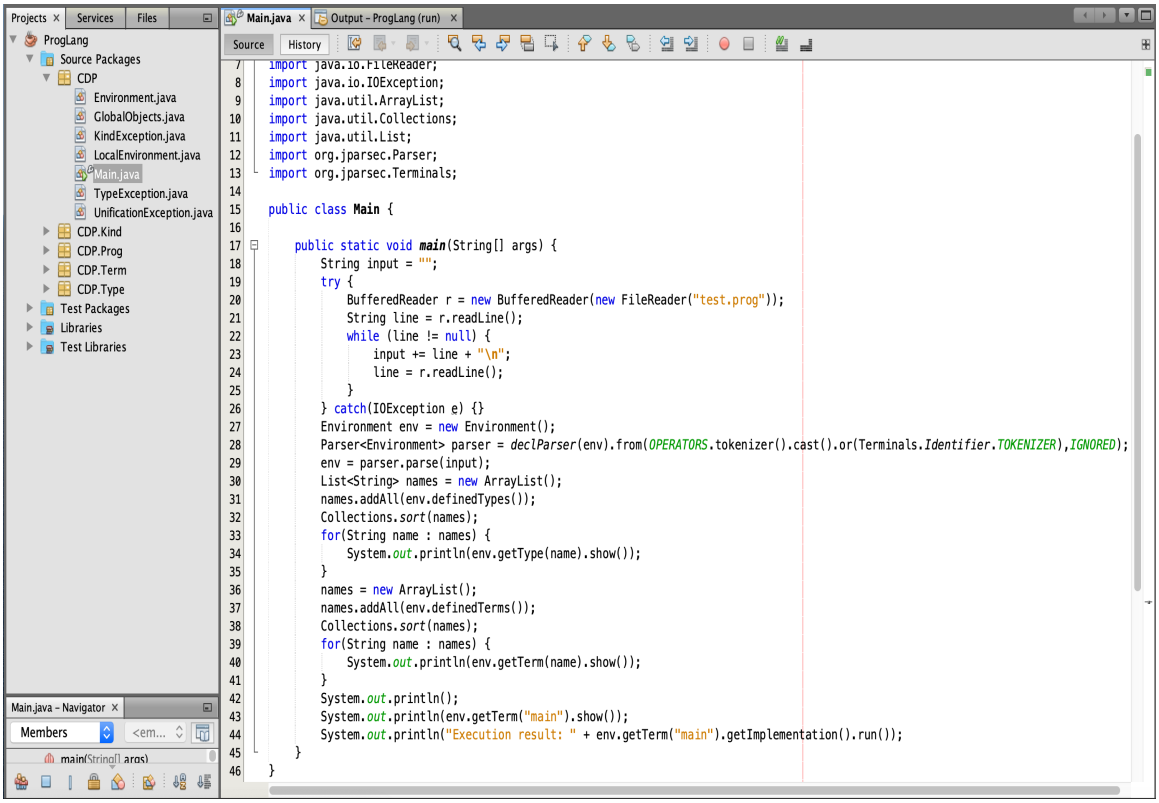

Figure 4.1: Execution of Main

4.1.4 Execution Result

When the program is run, the result of the example as shown below:

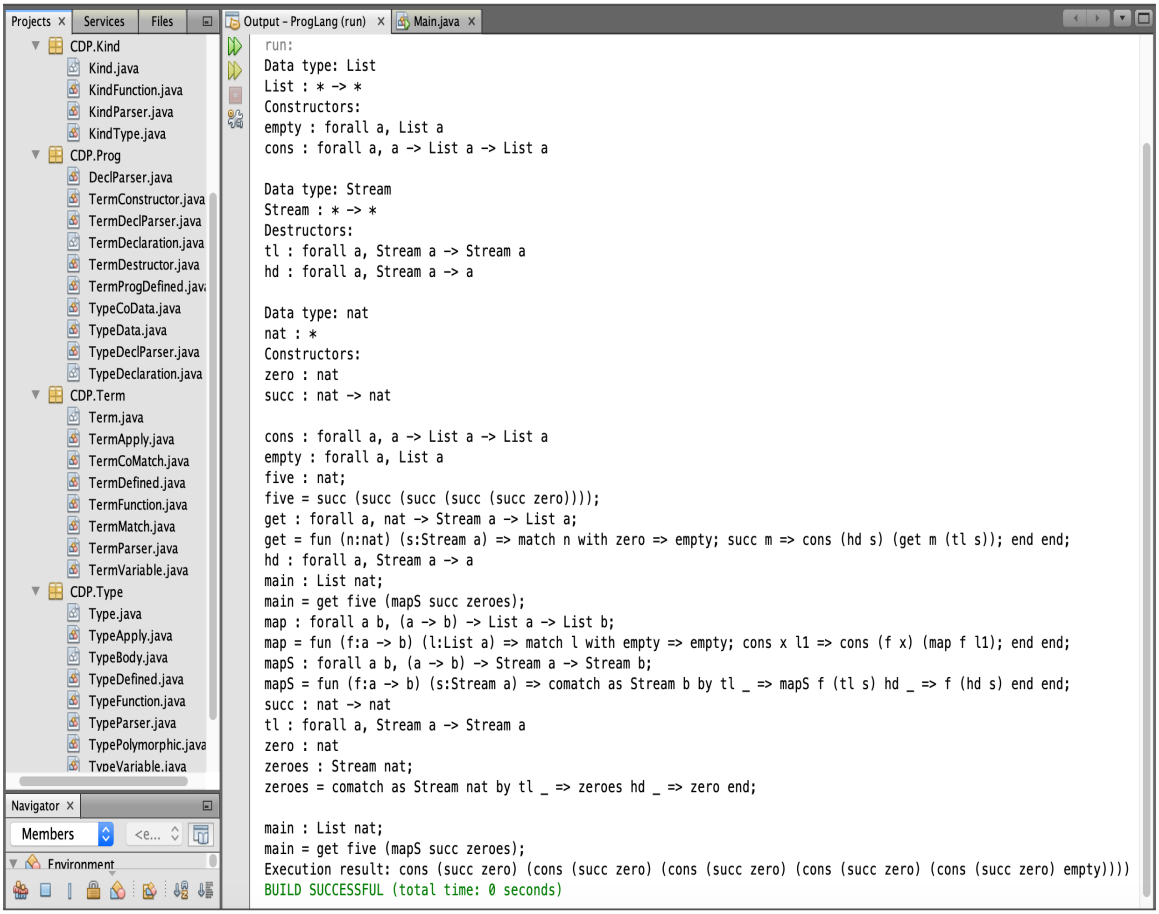

Figure 4.2: Execution Result of The Example

The program shows *Types, Kinds, Terms, Constructors* and *Destructors* with their types, *Functions*, and the execution results as shown in Figure 4.2. In Figure 4.2, *Type* of data *List* can be seen first, then *Kind* of *List*, and the *Constructors*. For codata. It is almost the same order starting with codata *Stream, Type, Kind*, and *Destructors*. With `Nat`, it is the same as data *List*. after it shows the functions, it lastly shows the execution results as below:

```
1 Execution result: cons (succ zero) (cons (succ zero) (cons (succ
  ↪  zero) (cons (succ zero) (cons (succ zero) empty))))
```

Chapter 5

Conclusion

5.1 Summary

In conclusion, while constructors can be used to define finite elements of data types when dealing with inductive data types, they cannot be used to define infinite elements of data types in coinductive data types. In addition, although functional programming languages such as Haskell do understand the need of using destructors to define infinite data types, they do not use it dually with constructors. Moreover, in dependently-typed languages such as Coq, constructors are what is being used to deal with infinite data types. In other words, there is no language that dealt with coinductive data types in a proper manner.

Therefore, we proposed, in this thesis, a language where finite and infinite data types are dealt with separately in a dual approach in which finite elements of data types are defined using constructors and infinite elements of data types are defined using observation. This approach is most appropriate as it takes into consideration the nature of the two data types (finite and infinite) and deals with them accordingly. We then defined the language and presented its grammar using BNF, after which we presented the implementation and execution of the program.

5.2 Future Work

For future work, we hope to see programming language environments that are safer and more convenient for programmers. This would be a very important inspiration for programmers to be more creative and productive than they are in a non-convenient and non-safe environments.

In terms of libraries in programming languages, we hope to see more libraries developed in the future with additional syntactic shorthands that make programming more convenient for the programmer such as putting nested matches and/or comatches in just one construction.

Bibliography

[1] Coq the coq proof assistant. `https://coq.inria.fr/library/index.html`. Accessed: 2020-03-23.

[2] Github jparsec. `https://github.com/jparsec/jparsec`. Accessed: 2018-08-29.

[3] Haskell a gentle introduction to haskell version 98. `https://www.haskell.org/tutorial/index.html`. Accessed: 2020-03-23.

[4] Abel A., Pientka B., Thibodeau D., and Setzer A. Copatterns programming infinite structures by observation. *ACM SIGPLAN Notices*, 48(1):27–38, 2013.

[5] M. Adriaan, F. Piessens, and M. Odersky. Generics of a higher kind. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, Oct 2008.

[6] Christopher. Allen and Julie Moronuki. Haskell programming from first principles. *Gumroad (ebook)*, 2017.

[7] Heckendorn R. B. A practical tutorial on context free grammars. 2015. `http://marvin.cs.uidaho.edu/Handouts/grammar.pdf`.

[8] Kees Doets. *From logic to logic programming.* Mit Press, 1994.

[9] Carlos Eduardo Gimenéz Enez. *Un Calcul de Constructions Infinies et son application a la vérification de systemes communicants.* PhD thesis, Ecole Normale Superieure de Lyon, 1996.

[10] E. Gamma. *Design Patterns*: *Elements of Reusable Object-Oriented Software.* Addison Wesley, Boston, MA, 1995.

[11] Johan Georg Granström. *Treatise on intuitionistic type theory*, volume 22. Springer Science & Business Media, 2011.

[12] Daume H. Yet another haskell tutorial. *School of Computing University of Utah*, 2004. `https://www.cs.utah.edu/-hal/htut/tutorial.pdf`.

[13] Paul Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge University Press, New York, USA, 2000.

[14] Paul Hudak and Donya Quick. *The Haskell School of Music: From signals to Symphonies*. Cambridge University Press, 2014.

[15] INRIA. *The Coq Proof Assistant Reference Manual*, 2010.

[16] Earley J. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970. `https://dl.acm.org/doi/pdf/10.1145/362007.362035`.

[17] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[18] Simon Marlow et al. Haskell 2010 language report. *Available on: https://www.haskell. org/onlinereport/haskell2010*, 2010.

[19] Winter Michael. On problems in polymorphic object-oriented languages with self types and matching. *Fundamenta Informaticae*, 71(4):477–491, 2006.

[20] C. Morris. *Signs, language and behavior*. Prentice-Hall, 1946.

[21] Lucas P. On the formalization of programming languages: Early history and main approaches. *The Vienna Development Method: The Meta-Language*, pages 1–23, 1978. `https://link.springer.com/content/pdf/10.1007/3-540-08766-4_8.pdf`.

[22] B.C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press. MIT Press, 2005.

[23] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[24] Dirk Riehle. Composite design patterns. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 218–228, 1997.

[25] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[26] F. Baader-W. Snyder. Unification theory. *Handbook of automated reasoning*, pages 447–533, 2001. `http://www.cs.bu.edu/~snyder/publications/UnifChapter.pdf`.

[27] Simon Thompson. *Haskell the craft of functional programming*. Addison-Wesley, Reading, Massachusetts, 2011.

[28] Alonso-Albi TomáS. Jparsec: a java package for astronomy with twelve years of development and use. *arXiv preprint arXiv:1806.03088*, 2018. `https://arxiv.org/pdf/1806.03088.pdf`.

[29] Hak Tony. and Jan Dul. Pattern matching. 2009.

[30] Martin H. Weik. *Backus Naur form*, pages 99–99. Springer US, Boston, MA, 2001.