Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Master Degree in Formal Methods for Computer Science and Engineering

Master's Thesis

Type-Based Test Generation for Haskell using Constraint Logic Programming

Author: Rafael Fernández Ortiz
Director: Julio Mariño Carballo

Madrid, Julio 2023

This Master's Thesis has been deposited to the ETSI Informáticos of the Universidad Politécnica de Madrid for its defense.

Master Degree in Formal Methods for Computer Science and Engineering
Julio 2023

Title: Type-Based Test Generation for Haskell using Constraint Logic Programming

Author: Rafael Fernández Ortiz
Director: Julio Mariño Carballo
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

Construir *generadores* para crear casos de prueba es uno de los puntos importantes para poder disponer de herramientas de testing automático. Una de las técnicas más conocidas, Property-Based Testing (PBT) utiliza este sistema para probar el comportamiento del software. Sin embargo la generación de estos valores es una tarea difícil, bastante costosa y propensa a errores. Aunque la mayoría de los marcos de PBT cubren escenarios simples, suele haber problemas con los casos de prueba generados puesto que existen algunas que requieren ciertas propiedades de los datos de entrada y que a menudo estas propiedades tienen que ver con precondiciones de funciones del SW que se se está probando.

En el contexto de las PBT para lenguajes fuertes y estáticamente bien tipados, como Haskell o Scala, proponemos un enfoque para aliviar al programador de la tarea de escribir estos generadores.

Nuestro enfoque proporciona una generación eficiente y automática de valores como *inputs* siguiendo la gramática formal de Haskell, concretamente, un mecanismo de traducción sintáctica guiado por la declaración de tipos. Esta es una característica que la mayoría de las herramientas conocidas de PBT como QuickCheck, por ejemplo, aún no tienen implementada. En particular, consideramos el caso en que los valores de entrada son tipos de datos algebraicos.

El proceso de generación se realiza a través de las especificaciones de las expresiones definidas según sintaxis del lenguaje y mediante resolución simbólica en lenguajes de programación lógica de restricciones. Puesto que los lenguajes de programación funcional modernos admiten declaraciones de tipos definidos por el usuario de una cierta complejidad – e.g. tipos paramétricos o polimórficos – haremos una presentación incremental de nuestra función de traducción. En una primera fase trataremos solo aquellas declaraciones no paramétricas o monomórficas. En este escenario, los tipos de datos algebraicos no se definen mediante una representación polimórfica en su declaración, siendo este el caso más sencillo de abarcar. Después, cubriremos la versión polimórfica de nuestra función de traducción sintáctica, añadiendo la complejidad de la parametrización de los tipos en la definición del ADT. Además, debido a que los lenguajes CLP tratan de proporcionar un espacio generado infinito de solución, en este trabajo, proporcionamos algunas técnicas para limitarlo y obtener un espacio de valores generados suficientemente bueno que mantenga la completitud que necesitamos en nuestros casos de prueba. Definimos aquí un concepto de *tamaño* alrededor y en estrecha relación al significado de completitud que buscamos, e integramos ese concepto en la definición de nuestra función.

Por último, hablamos de aquellos enfoques interesantes que serían adecuados para el trabajo futuro. Por ejemplo, haciendo extensiones de sintaxis Haskell utilizando Liquid Haskell para proporcionar invariantes o restricciones complejas en el proceso de generación. Con esta nueva característica, podríamos cubrir tipos de datos algebraicos como pueden ser estructuras de arbol rojo y negro.

Abstract

Building generators to create test cases is one of the important points to be able to have automatic testing tools. One of the best known techniques, Property-Based Testing (PBT) uses this system to test software behavior. However, the generation of these values is a difficult, rather costly and error-prone task. Although most PBT frameworks cover simple scenarios, there are often problems with the generated test cases since there are some that require certain properties of the input data and often these properties have to do with preconditions of functions of the SW being tested.

In the context of PBTs for strong and statically well-typed languages, such as Haskell or Scala, we propose an approach to relieve the programmer from the task of writing these generators.

Our approach provides efficient and automatic generation of values such as *inputs* following Haskell's formal grammar, namely, a syntactic translation mechanism guided by type declaration. This is a feature that most known PBT tools such as QuickCheck, for example, do not yet have implemented. In particular, we consider the case where the input values are algebraic data types.

The generation process is performed through the specification of expressions defined according to language syntax and by symbolic resolution in constraint logic programming languages. Since modern functional programming languages support user-defined type declarations of a certain complexity – e.g. parametric or polymorphic types – we will make an incremental presentation of our translation function. In a first phase we will treat only those non-parametric or monomorphic declarations. In this scenario, algebraic data types are not defined by a polymorphic representation in their declaration, this being the simplest case to cover. Next, we will cover the polymorphic version of our syntactic translation function, adding the complexity of type parameterization in the ADT definition. Also, because CLP languages try to provide an infinite generated space of solution, in this paper, we provide some techniques to limit it and obtain a good enough generated value space that maintains the completeness we need in our test cases. We define here a concept of Ôtextitsize around and closely related to the meaning of completeness we seek, and integrate that concept into our function definition. Finally, we discussed those interesting approaches that would be suitable for future work. For example, making Haskell syntax extensions using Liquid Haskell to provide invariants or complex constraints in the generation process. With this new feature, we could cover algebraic data types such as red and black tree structures.

Acknowledgments

I would like to thank the many people who in some way have been support me and accompanying me since this thesis started.

To my family, thank you for supporting me from afar.

To my fellow master's students, for having shared this feeling of frustration, despair, success and satisfaction.

To my dog Pepino, who I have not had to sacrifice time to be with him to finish my thesis.

To Andrea, for always being there for me, supporting me through thick and thin. Being my most faithful pillar to hold on to in the hardest days. For having supported me since the beginning of the master. This thesis is thanks to you.

And finally, I wanted to give special thanks and mention to Francisco Ibañez, father of Mortadelo and Filemon, who died a few days before delivering this work and, although he is not related to the software world, he has been a reference that has marked my personality a lot. For this reason, I would like to thank him wherever he is. Rest in peace.

Contents

Chapter 1

Introduction

The absence of risks in software is essential and highlights what ensures that the software will perform as expected and not cause unintended consequences. Unfortunately, that feautere is not present most time.

For critical systems, such as those used in the medical or aerospace industries, the stakes are even higher. These systems must be thoroughly tested and validated to ensure that they will not cause harm or failure in a critical situation. For example:

- Medical systems: Medical systems such as electronic health records (EHRs) and medical devices are critical systems that can have serious consequences if they fail. A bug in an EHR system could lead to incorrect patient information being displayed, potentially leading to a misdiagnosis or other medical errors.

- Aerospace systems: Aerospace systems such as aircraft navigation systems and flight control systems are critical systems that must operate reliably at all times. A bug in an aircraft navigation system could cause the plane to fly off course, leading to a crash.

- Industrial control systems: Industrial control systems (ICS) are used to control and monitor industrial processes such as manufacturing, power generation, and oil and gas production. An issue in an ICS could cause a malfunction in a manufacturing process, leading to costly downtime or even physical damage to the equipment, or even a cyber-attack on an ICS could cause a shutdown of the whole process causing a major disruption.

The most used way to guarantee software quality and detect errors is testing, so it is essential to perform it in the best possible way.

1.1 Testing

Software testing (or simply testing) is the process of evaluating a system or its component(s) to find whether it satisfies the specified requirements. It consists of executing a program on a known pre-selected set (test suite) of inputs (test cases) and inspecting whether the outputs match the expected results.

This process validates the semantic properties of a program's behaviour [1]. It's important to test software thoroughly before deployment to ensure it functions as intended and to identify and fix bugs or other issues. Testing is an essential step in the software development process and is critical for ensuring the quality and reliability of software.

Therefore, we can define in a relaxed way that a **test** is a set of executions on a given program using different input data for each execution; its purpose is to determine if the program functions correctly. A test has a negative result if an error is detected during the test i.e., the program crashes or a property is violated. [1] [2]

A test has a positive result if a series of tests produces no error, and the series of tests is "complete" under some coverage metric. When we say in software testing that a test is "complete", it refers to the level of coverage the tests provide for the software being tested. In other words, that reflects a representative percentage of the reliability of the software with respect to expected behaviour. However, we have to consider that "reliability" is relative and it is biased and subject to the chosen test cases, which is itself subject to the criteria of the tester [2].

A test has an "incomplete" result if a series of tests produces no errors but the series is not complete under the coverage metric. In summary, a test is focused on evaluating the software to find any issues and bugs [1].

There are different types of tests that can be used during the software development process, each with a different purpose and focus. Some of the main types of tests are:

- **Unit testing**: Unit testing is a type of testing that focuses on individual components of the software, such as individual functions or methods. Unit testing aims to ensure that each component behaves as expected. Unit tests are usually automated, and they are run as part of the development process to catch any issues early.
- **Integration testing**: Integration testing is used to ensure that different software components work together correctly. It tests the interactions between different parts of the software. Integration tests are usually automated, and they are run after the unit tests to ensure that the integrated system behaves as expected.
- **Acceptance testing**: Acceptance testing is used to ensure that the software meets the needs of the end users. It is typically done by the customer or other stakeholders to ensure that the software meets their needs. Acceptance tests can be automated or manual, and they are run after system tests to ensure that the system is ready to be deployed.

Sadly, trying to find counter-examples by testing that produces bugs a behaviour non-expected is most of the time a difficult task. In simpler software, testers could find most of those cases which produce counter-examples, designing test cases one by one. However, the design process reaches those cases that one knows by experience or intuition, leaving aside very interesting and not at all intuitive cases that can hardly be imagined [1] [2]. For this reason and because it can be a very tedious task, it would be ideal to automate the generation of test cases.

Testing and Specificaions

Also, during the life-cycle of software development has used several known techniques in order to get free-bugs software in an efficient and proper way. The most common paradigm, which is also the most natural way, is test driven development (a.k.a TDD) [3].

TDD is a software development technique in which tests are developed before the code, in short and incremental cycles. This technique proposes for the developer to create a new flawed test, and then to implement a little piece of code, in order to satisfy the current test set [4]. Then, the code is refactored if necessary, to provide a better structure and architecture for the current solution.

The challenge addressed in this work is to use TDD in applications with non-deterministic behaviour as stated before. Although it is not possible to know exactly what the output will be, it is usually possible to check whether the generated output is valid or not [2].

The following factors make it difficult to develop randomized software using TDD [3]:

- Results for each execution may be different for the same inputs, which makes it difficult to validate the return value.
- Obtaining a valid return for a test case execution does not mean that valid return will be delivered on the next executions.
- The random decisions and their paths number make it not viable to create Mock Objects that return fixed results for these decisions.
- It is difficult to execute a previous failed test with the same random decisions undertaken in its former execution.

In conclusion, you have to iterate several times in case of testing fails.

On the other hand, some techniques like correctness by constructions, try to formalize some specifications and build code based on them [5].

The idea is to start with a succinct specification of the problem, which is progressively evolved into code in small, tractable refinement steps. Experience has shown that the resulting algorithms are invariably simpler and more efficient than solutions that have been hacked into correctness. Furthermore, such solutions are guaranteed to be correct (i.e. they are guaranteed to comply with their specifications) in the same sense that the proof of a mathematical theorem is guaranteed to be correct. Here you don't have to iterate too as other ones, but the formalized process is, in general, a complex task.

For many reasons, in order to get a good enough solution for testing, property-based testing was coming up.

1.2 Property-Based Testing

Property-based testing (a.k.a PBT) is a technique that uses inputs generated by specifications to test the properties of a system, rather than specific or random inputs. It helps to mitigate risks in the software industry by providing an automated way to test the software in a wide range of scenarios using randomly generated test cases. This can help to identify bugs and other issues that may not be found using traditional testing techniques such as manual testing or unit testing [6].

The use of randomly generated inputs in PBT allows for a more thorough exploration of the software's behaviour, making it more likely that any bugs or issues will be found. It also helps to ensure that the software behaves correctly in a wide range of scenarios, which is especially important for critical systems.

PBT helps to ensure that the software is robust and can handle unexpected inputs or edge cases. This is particularly important for systems where failure could have serious consequences. Also helps in testing the software performance and scalability, by testing the software with large inputs, it can identify potential performance issues that would be difficult to detect with other testing techniques.

The specification of one or more properties is the driver of the testing process, which assures that the given program meets the stated property, leaving aside the task of generating valid inputs [1].

For example, if an analyst wants to validate that a specific program correctly authenticates a user, a property-basted testing procedure tests the implementation of the authentication mechanisms in the source code to determine if the code meets the specification of *correctly authenticating the user*.

Specifications state what a system should or should not do. The advantage of using specifications is the formalism they establish for verifying proper (or improper) program behaviour. PBT validates that the final product is free of specific flaws. Because PBT concentrates on generic flaws, it is ideal for focusing on analysis late in the development cycle after program functionality has been established [7].

In a property-based framework, test cases are automatically generated and run from assertions about the logical properties of the program. Feedback is given to the user about their evaluation [1] [6].

PBT naturally is based on the logic programming paradigm. Assertions are first-order formulas and thus easily encoded as program predicates. Therefore, a property-based approach to testing is intuitive for the logic programmer.

1.3 Problem: Test cases that satisfy a given specification

Property-Based testing provides a helpful solution to generate random test cases for testing. And it is good enough for most pieces of code that want to test. However, we will put focus on those functions that assume inputs with preconditions. [6]

Example 1.3.1 (Sorted Lists)**.** Let S be a set, `xs` a list of elements of S, and consider the ordering relationship $\preceq$ over elements of S. We can say that xs is an **ordered list** if it holds one of the following invariants:

- `INV1` xs is empty.
- `INV2` $\forall xss$ sublist of xs with $xss \neq \emptyset$, $\exists a \in xss$ such that $\forall x \in xss, a \preceq x$ holds.

Let's suppose we want to test the behaviour of our `insertOrdered` function which its expected behaviour should be the following:

`PROP1` *Given an ordered list,* `insertOrdered` *inserts an element and its result is an ordered list.*

```
insertOrdered :: Ord a => a -> [a] -> [a]
insertOrdered a [] = [a]
insertOrdered a xs'@(x:xs)
  | a <= x     = a : xs'
  | otherwise = x : insertOrdered a xs
```

```
def insertOrdered[A <: Ordered[A]]: A => List[A] => List[A] =
  (a: A) => {
    case Nil => List(a)
    case as@ ::(x, xs) => if (a <= x) a :: as else x :: insertOrdered(a)(xs)
  }
```

A property-based testing framework should generate several enough random **ordered lists** to check if `PROP1` holds. This is not as easy as you could think. Let's do our own mental exercise step by step. Let's suppose we have a good PBT framework:

1. First of all, the framework has to generate randomly a set of lists.

2. Then, it has to check which one of them is an **ordered list**, i.e, it has to check if holds either `INV1` or `INV2`.

3. Finally checks if the property `PROP1` holds, which means the result has to be an **ordered list**, i.e. it has to check if holds either `INV1` or `INV2` too.

This process is more complex, but for this moment we can consider this friendly description. We will deeply get ahead in the following chapters.

Therefore, imagine that your PBT framework generates 100 randomly generated lists in every iteration. Probably, one or, if you have lucky, two lists of them are ordered lists.

On the one hand, it is so difficult to achieve so many scenarios (at least in a shorter time). Also, the framework probably brings you just the same empty list (because it is the easiest generated test case that holds one of the invariants) as the input value for checking the property which would make the results unreliable.

And on the other hand, the property `PROP1` is relatively simple but, what happens if we consider a more complex property? For example, we can consider a red-black tree.

Example 1.3.2 (Red-Black Tree)**.** A **red-black tree** is a binary search tree where each node has two labels: a color `C`, which is either `red` (`R`) or `black` (`B`), and an integer `N`. For the purpose of test generation, node values are abstracted away in the definition of the data structure:

$$
\begin{aligned}
\texttt{C} &::= \texttt{R | B} \\
\texttt{N} &::= \texttt{... | -1 | 0 | 1 | ...} \\
\texttt{Tree} &::= \texttt{nil | C N Tree Tree}
\end{aligned}
$$

A red-black tree must also satisfy the following three invariants:

`INV1` Every path from the root to a leaf has the same number of black nodes

`INV2` No red node has a red child and

`INV3` For every node n, all the nodes in the left (respectively, right) subtree of n, if any, have keys that are smaller (respectively, bigger) than the key labeling n.

Since red-black trees enjoy a weak form of balancing, operations such as inserting, deleting, and finding values are more efficient, in the worst case, than in ordinary binary search trees.

Let's suppose we want to test the behaviour of our `insertOrderedRBTree` function which its expected behaviour should be the following:

`PROP2` *Given a red-black tree,* `insertOrderedRBTree` *inserts an element and its result is a new tree which is a red-black tree.*

The following code is inspired by the Haskell ADT definition of red-black trees in [8].

```
data Color = R | B deriving Show

data Tree a = Nil | T Color a (Tree a) (Tree a) deriving Show
```

```
makeBlack :: Tree a -> Tree a
makeBlack (T _ y a b) = T B y a b
makeBlack t = t
```

```
balance :: Tree a -> Tree a
balance T B z (T R y (T R x a b) c) d = T R y (T B x a b) (T B z c d)
balance T B z (T R x a (T R y b c)) d = T R y (T B x a b) (T B z c d)
balance T B x a (T R z (T R y b c) d) = T R y (T B x a b) (T B z c d)
balance T B x a (T R y b (T R z c d)) = T R y (T B x a b) (T B z c d)
balance t = t

insert :: (Ord a) => a -> Tree a -> Tree a
insert x s = makeBlack $ insertAux s
  where insertAux Nil  = T R x Nil Nil
        insertAux (T c y a b)
          | x < y  = balance T c y (insertAux a) b
          | x == y = T c y a b
          | x > y  = balance T c y a (insertAux b)
```

```
sealed trait Color
case object R extends Color
case object B extends Color

sealed trait Tree[A]
case object Nil extends Tree[Nothing]
case class T[A](color: Color, node: A, tl: Tree[A], tr: Tree[A]) extends Tree[A]
```

```
def makeBlack[A]: Tree[A] => Tree[A] = {
  case ttree@T(_, _, _, _) => ttree.copy(color = B)
  case t => t
}

def balance[A]: Tree[A] => Tree[A] = {
  case T(B, z, T(R, y, T(R, x, a, b), c), d) => T(R, y, T(B, x, a, b), T(B, z, c, d))
  case T(B, z, T(R, x, a, T(R, y, b, c)), d) => T(R, y, T(B, x, a, b), T(B, z, c, d))
  case T(B, x, a, T(R, z, T(R, y, b, c), d)) => T(R, y, T(B, x, a, b), T(B, z, c, d))
  case T(B, x, a, T(R, y, b, T(R, z, c, d))) => T(R, y, T(B, x, a, b), T(B, z, c, d))
  case t => t
}

def insert[A <: Ordered[A]]: A => Tree[A] => Tree[A] =
  (x: A) => {
    def insertAux: Tree[A] => Tree[A] = {
      case Nil => T(R, x, Nil, Nil)
      case ttree@T(c, y, tl, tr) =>
        if (x < y) balance(T(c, y, insertAux(tl), tr))
        else if (x == y) ttree
        else balance(T(c, y, tl, insertAux(tr)))
    }

    makeBlack andThen insertAux
  }
```

Following the same reasoning that we did before, the reader can deduce how complex is the task.

In general, PBT is not prepared to generate inputs with preconditions, let alone input with complex ones.

1.4 Our Approach

Building *generators* to create test cases that property-based testing use to test software behaviour is a hard, quite costly, and error-prone task. Even though most property-based testing frameworks cover simple scenarios, there are many troubles with preconditioned generated test cases as we have just seen.

In contrast to the recent works [9] [10], what we propose is (1) to build an efficient and automatic generator of input test values that satisfy a given specification and (2) provide a language syntax-driven bijection between the origin language's expressions and the constraint logic programming language ones which can translate and map expression from one language to other. In particular, we will consider the case when the input values are algebraic data types satisfying complex constraints. The generation process is performed via symbolic execution in the CLP language of the translated expressions of those ADTs and their specifications.

We will focus on the **strong and static well-typed** language Haskell and we will use Prolog as CLP language. Although it is well known that the mainly property-based testing framework for Haskell is **QuickCheck**, and it has its own *generators*, we will provide steps to build a mechanism to generate those kinds of preconditioned input values. In particular, we will explore how to create a model that maps Haskell's ADT expressions and its specification to Prolog expression, generate symbolic Prolog expressions that hold the specifications and returns those expressions to Haskell.

Following the recent state-of-the-art approach, we use a *declarative* language such as Haskell following those works that are based on using CLP languages as generators. Being a declarative language allows us to separate the process of defining the algebra data type structure expression and their constraints from their semantics or instances. This separation helps improve the correctness of the developed test case generators, which implies that we can separate between *what* we want to generate in an expressive way [9].

But the main reason is the nature of being static-typed. It allows us to reason about types in a safe way and also we gain in static type checking in compile time, which adds us a plus about program verification. Therefore, Haskell allows to support us the fact that algebraic data types expressions are their own type (the 'ADT' type that corresponds), and then, intuitively and implicitly, we can assume and reason the ADT's invariants holds thanks to the fact that they are from that type (the 'ADT' type).

In contrast to the recent works [11] that use Z3 or some SMT solver such as a test generator and output test validator. We use constraint logic programming language such as our test generator.

This approach fits better with our purpose thanks to how flexible, expressive, and descriptive the syntax of both languages Haskell and Prolog are. In fact, we show how natural is to define an algebraic data type in Haskell and find an expression 'equivalent' (in the sense of translation) in Prolog. This leads us to find a mechanism that translates both languages' expressions which also can fit so well with the Erlang language.

However, in contrast to the recent state-of-the-art research and regarding to the mentioned Erlang language, we use **static-typed and strong-typed** language.

The use of dynamic language could provocate non-safety in its type checking, and therefore, we cannot ensure what type is working when we deal with a given formal expression. For instance, in Erlang we can have both `adt` : τ_1 and `adt` : τ_2, which their formal expression is `adt`. When we generate test cases for that expression and come back to Erlang, we could provide wrong test input values [7] [12].

There seems to be some controversy about the advantages of static vs dynamic typing [12] [7]. Type-checking provides another barrier against nonsensical programs. Moreover, whereas in a dynamically typed language, type mismatches would be discovered at runtime, in strongly typed statically checked languages type mismatches are discovered at compile time, eliminating lots of incorrect programs before they have a chance to run [7].

> Languages without type systems or even safe, dynamically checked languages, tend to offer features or encourage programming idioms that make type-checking difficult or infeasible.
> ...
> The assertion that types should be an integral part of a programming language is separate from the question of where the programmer must physically write down type annotations and where they can instead be inferred by the compiler. A well-designed statically typed

language will never require huge amounts of type information to be explicitly and tediously maintained by the programmer. [13]

In Haskell, the definition of the ADT expression is itself the declaration of the type to which it belongs. That means, we cannot type two ADT with the same expression and try belonging on different types. There is a univocal relationship between the ADT's expression and the its type.

Finally, for the purpose of this work, we will illustrate all these concepts and ideas by applying them while writing a generator for red-black trees. As we defined before:

Definition (Red-Black Tree)**.** A **red-black tree** is a binary search tree where each node has two labels: a color `C`, which is either `red` (`R`) or `black` (`B`), and an integer `N`. For the purpose of test generation, node values are abstracted away in the definition of the data structure:

$$\begin{array}{rcl} \texttt{C} & ::= & \texttt{R} \mid \texttt{B} \\ \texttt{N} & ::= & \texttt{...} \mid \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \texttt{...} \\ \texttt{Tree} & ::= & \texttt{nil} \mid \texttt{C N Tree Tree} \end{array}$$

A red-black tree must also satisfy the following three invariants:

`INV1` Every path from the root to a leaf has the same number of black nodes

`INV2` No red node has a red child and

`INV3` For every node n, all the nodes in the left (respectively, right) subtree of n, if any, have keys that are smaller (respectively, bigger) than the key labeling n.

The definition of this structure, which holds a set of complex invariants, can be defined as an algebraic data type in Haskell as follow:

```
data Color = R | B deriving Show

data Tree a = Nil | T Color a (Tree a) (Tree a) deriving Show
```

We can deduce that the expression `Nil` and `T Color a (Tree a) (Tree a)` in Haskell can be mapped to the expressions `nil` and `t(C,X,L,R)` in Prolog, respectively; where `C` is the color (either `red` or `black`), and both `L`, `R` are red-black trees too, i.e., they can be expressed as `nil` or `t(C',X',L',R')`. This brings us light on our hard way, don't you?. Let's see an example before starting this work.

Definition (Sorted Lists)**.** Let S be a set, **xs** a list of elements of S, and consider the ordering relationship $\preceq$ over elements of S. We can say that xs is an **ordered list** if it holds one of the following invariants:

`INV1` xs is empty.

`INV2` $\forall xss$ sublist of xs with $xss \neq \emptyset$, $\exists a \in xss$ such that $\forall x \in xss, a \preceq x$ holds.

First of all, we have to identify the Haskell lists' syntax expression, that is:

A list expression in Haskell could be either `[]` or `x:xs` syntax. Therefore, the list ADT could be expressed in Prolog as follows `[]` or `X:Xs`, respectively.

Then, in order to provide the invariants `INV1` and `INV2` and get a definition of a sorted list in Prolog, we should translate in someway these two (written in Haskell or, more precisely written in Liquid Haskell) or type directly in scratch the restrictions in Prolog to get the following expressions or similar:

`INV1` `sorted_list([])` which holds that an empty list is a sorted list.

INV2 sorted_list(_:[]) which holds that a one-element list is a sorted list.

INV2 sorted_list(X1:X2:Xs) :- X1 #=< X2, sorted_list(X2:Xs) which means that a list is a sorted list if it holds that every two elements hold an ordering relationship and the tail of the list is a sorted list too.

In resume, we should map the Haskell definition of a sorter list to some Prolog definition like this:

```
:- use_module(library(clpfd)).

%% Base case for an empty list
sorted_list([]).

%% Recursive case for a non-empty list
sorted_list(_:[]).
sorted_list(X1:X2:Xs) :- X1 #=< X2, sorted_list(X2:Xs).
```

With this implementation, we can get the following test cases using, for instance, the Prolog-CLI prompt and the expression sorted_list. However, readers can think that modeling list adt and the invariants for sorted list could be a little bit easy. Let's see a final challenge that shows interest in this approach.

Definition (Binary Search Tree)**.** A binary search tree (BST) is a tree in which all the nodes follow the below-mentioned properties.

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than or equal to its parent node's key.

A binary search tree must also satisfy the following invariants:

INV1 An empty tree is a binary search tree.

INV2.1 The left sub-tree is a binary search tree.

INV2.2 The right sub-tree is a binary search tree.

Here we can see that invariants not only are more complex than the sorted list example but also it holds recursively onto their inner structures. Therefore, getting a test case for this example is a hard task. But we can do as before. First of all, we have to identify the Haskell bst' syntax expression, that is:

A bst expression in Haskell could be either Nil or T a BSTree a BSTree a syntax. Therefore, the bst ADT could be expressed in Prolog as follows nil or t(N,L,R).

Then, in order to provide the invariants INV1, INV2.1 and INV2.2 and get a definition of a bst in Prolog, we do as similar as before to get the following expressions or similar:

INV1 bst(nil) which holds that an empty tree is a binary search tree.

INV2.1 **and** INV2.2 bst(t(_,nil,nil)) defines a non-empty tree with no left or right subtrees, which holds that a one-element tree is a binary search tree.

INV2.1 bst(t(N,L,nil)) :- L = t(LN,_,_), LN #< N, bst(L) defines a non-empty tree with a left subtree and no right subtree.

INV2.2 bst(t(N,nil,R)) :- R = t(RN,_,_), RN #> N, bst(R) defines a non-empty tree with a right subtree and no left subtree.

`INV2.1` **and** `INV2.2`

```
bst(t(N,L,R)):-L = t(LN,_,_), LN #< N, bst(L)
               R = t(RN,_,_), RN #> N, bst(R)
```

defines a non-empty tree with both left and right subtrees.

In resume, we should map the following Haskell definition of a binary search tree:

```
data BSTree a = Nil | T a (BSTree a) (BSTree a) deriving Show
```

to some Prolog definitions like this:

```
:- use_module(library(clpfd)).

% Base case for an empty tree
bst(nil).

% Recursive case for a non-empty tree
bst(t(_,nil,nil)).
bst(t(N,L,nil)) :- L = t(LN,_,_), LN #< N, bst(L).
bst(t(N,nil,R)) :- R = t(RN,_,_), RN #> N, bst(R).
bst(t(N,L,R)) :- L = t(LN,_,_), LN #< N, bst(L), R = t(RN,_,_), RN #> N, bst(R).
```

1.5 Related Works

1.5.1 QuickCheck: Automatic testing of Haskell programs

QuickCheck [14] is a library for random testing of program properties. The programmer provides a specification of the program, in the form of properties that functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in Haskell, using combinators provided by QuickCheck. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators.

QuickCheck is one of the original property-based testing frameworks. It was developed in 1999 as a tool for Haskell, and it has since been ported to other languages such as Erlang, Scala, and Clojure. QuickCheck uses random input generation and shrinking to automatically generate test cases for a given property. [15]

1.5.2 ScalaCheck: Automatic testing of Scala and Java programs

ScalaCheck is a library written in Scala and used for automated property-based testing of Scala or Java programs. ScalaCheck was originally inspired by the Haskell library QuickCheck but has also ventured into its own. ScalaCheck is used by several prominent Scala projects, for example, the **Scala compiler** and the **Akka** concurrency framework. [16]

1.5.3 PropEr: Property-based testing tool for Erlang

PropEr is a QuickCheck-inspired open-source property-based testing tool for Erlang, developed by Manolis Papadakis, Eirini Arvaniti, and Kostis Sagonas.

PropEr is a property-based testing tool, designed to test programs written in the Erlang programming language. Its focus is on testing the behaviour of pure functions. On top of that, it is equipped with two library modules that can be used for testing stateful code. The input domain of functions is specified through the use of a type system, modeled closely after the type system of the language itself. Properties are written using Erlang expressions, with the help of a few predefined macros. [17]

1.5.4 Hypothesis: Property-based testing tool for Python

Hypothesis is a modern property-based testing library for Python. It's similar to the previously mentioned frameworks but with some differences, it also provides features like stateful testing and advanced strategies for input generation. [18]

1.6 Test Cases with Pre-Condition(s)

Efforts to provide best-fitted generators to complex constrained test cases are becoming a very current research topic. It is faced from some perspectives.

Regarding to Fioravanti et al. research works [6], [9] and [10], we can observe a long projection to provide a similar approach. Indeed, in [6] and [9], they are based on the idea to relieve the developer from the task of writing data generators of valid inputs using CLP languages. They show how using CLP languages for that propose and prove how many benefits and how efficient is to use these kinds of logic programming languages as test generators.

Finally, in [10], they bring us some kinds of best-performances using program transformation instead of the classical and native left-to-right strategy implemented by standard CLP systems which can very inefficient. These works, in essence, propose a framework based on Constraint Logic Programming for the systematic development of generators of large sets of structurally complex test data structures.

They adopt a declarative approach too following the so-called *constrain-and-generate* approach. However, they focus on the **dynamically typed** functional programming language Erlang and refer to how arduous the task of writing a generator that satisfies all constraints defined by a filter for the PropEr framework is.

Ricardo Peña et al. research works in [11] propose a system that can automatically generate an exhaustive set of black box test cases, up to a given size, for programs under test requiring complex preconditions. The key of this research is how they translate a formal precondition into a set of constraints belonging to the decidable logic of SMT solvers. They use an **SMT solver as a test case generator** to directly generate test cases satisfying a complex precondition. Also, they describe both black-box and white-box approaches to fit searching strategies for those test cases.

1.7 Thesis Organization

We will discuss the organization of the content and provide a brief overview of the different topics that will be covered throughout the book. We will focus on two main aspects: Haskell and Prolog.

1. In this preliminary chapter 2, we will explore the basic concepts of Haskell and Prolog. We will discuss the roles these languages play in the context of the thesis and analyze their formal syntax.

    - Haskell: We will explain what Haskell is and its significance in functional programming. We will discuss its key features and how it relates to the translation approach that will be employed later on.
    - Haskell Syntax: We will analyze the specific syntax of Haskell that facilitates the translation of expressions. We will discuss the constructions and conventions used to represent.
    - Prolog: We will introduce Prolog, a logic programming language. We will describe its utility and how it applies to problem-solving based on logical rules. We will analyze its formal syntax and highlight relevant aspects for translating algebraic data type expressions to Prolog expressions such that recursive definitions.
    - Prolog Syntax: We will explore the Prolog syntax necessary for receiving and processing the translated expressions from Haskell. We will discuss how logical rules are structured in Prolog and how they can be used to manipulate and evaluate the expressions sent from Haskell.

2. In chapter 3, we will present a translation mechanism guided by Haskell syntax to send algebraic data types expressions to Prolog expressions. We will provide the necessary foundations to understand and utilize the translation mechanism from Algebra data types to Prolog expressions, laying the groundwork for subsequent chapters that delve deeper into these concepts and present practical applications.

Chapter 2

Preliminaries

In this section, we present the basic notions of the Haskell and Prolog languages. Both will be the main actors of this work and before starting we need to bring some context about both languages, identifying which one will be our scope and what will need from each one.

2.1 Haskell, the functional guy

Haskell is a purely functional programming language, lazy, statically typed, concurrent, type inference, and why not, elegant and concise language.

Purely functional programming. Every function in Haskell is a function in the mathematical sense (i.e., "pure"). Even side-effecting IO operations are but a description of what to do, produced by pure code. There are no statements or instructions, only expressions which cannot mutate variables (local or global) nor access states like time or random numbers. The following function takes an integer and returns an integer. By the type it cannot do any side-effects whatsoever, it cannot mutate any of its arguments.

```
square :: Int -> Int
square x = x * x
```

The following string concatenation is a type error:

```
"Name: " ++ getLine -- error
```

Because `getLine` has type `IO String` and not `String`, like "Name: " is. So by the type system, you cannot mix and match purity with impurity.

Lazy. Functions don't evaluate their arguments. This means that programs can compose together very well, with the ability to write control constructs (such as if/else) just by writing normal functions. The purity of Haskell code makes it easy to fuse chains of functions, allowing for performance benefits.

Statically Typed. Every expression in Haskell has a type which is determined at compile time. All the types composed together by function application have to match up. If they don't, the program will be rejected by the compiler. Types become not only a form of guarantee, but a language for expressing the construction of programs. All Haskell values have a type:

```
char = 'a'     :: Char
int = 123      :: Int
fun = isDigit :: Char -> Bool
```

You have to pass the right type of values to functions, or the compiler will reject the program:

```
isDigit 1 -- error
```

Concurrent. Haskell lends itself well to concurrent programming due to its explicit handling of effects. Its flagship compiler, GHC, comes with a high-performance parallel garbage collector and lightweight concurrency library containing a number of useful concurrency primitives and abstractions. Easily launch threads and communicate with the standard library:

```
main = do
  done <- newEmptyMVar
  forkIO (do putStrLn "I'm one thread!"
             putMVar done "Done!")
  second <- forkIO (do threadDelay 100000
                       putStrLn "I'm another thread!")
  killThread second
  msg <- takeMVar done
  putStrLn msg
```

Use an asynchronous API for threads:

```
do a1 <- async (getURL url1)
  a2 <- async (getURL url2)
  page1 <- wait a1
  page2 <- wait a2
  ...
```

Type Inference. You don't have to explicitly write out every type in a Haskell program. Types will be inferred by unifying every type bidirectionally. However, you can write out types if you choose, or ask the compiler to write them for you for handy documentation. This example has a type signature for every binding:

```
main :: IO ()
main = do line :: String <- getLine
          print (parseDigit line)
  where parseDigit :: String -> Maybe Int
        parseDigit ((c :: Char) : _) =
          if isDigit c
             then Just (ord c - ord '0')
             else Nothing
```

But you can just write:

```
main = do line <- getLine
          print (parseDigit line)
  where parseDigit (c : _) =
          if isDigit c
             then Just (ord c - ord '0')
             else Nothing
```

Elegant and concise. Because it uses a lot of high-level concepts, Haskell programs are usually shorter than their imperative equivalents. And shorter programs are easier to maintain than longer ones and have fewer bugs.

2.1.1 Grammar specification

For our purpose, it is necessary to identify which is the specification of Haskell's grammar. More precisely, which is the specification of the definition of an algebraic data type in Haskell. Then, we need to identify which elements participate in the specification, and, in the next chapter, we will see how we need to translate those elements to Prolog elements within a complete formal Prolog expression. Therefore:

Let's suppose we want to formally define an algebraic data type. Regarding [19] and [20], we can consider a subset of the grammar specification (expressed in BNF-Like syntax) for defining an ADT expression in Haskell. In our case, we won't consider either `context` or `deriving` words because we want just an approach to simple algebraic data type expressions. Therefore, for our purpose and simplicity, we have considered the following subset.

$$
\begin{array}{rcll}
\mathit{special} & ::= & \texttt{(} \mid \texttt{)} \mid \texttt{,} \mid \texttt{;} \mid \texttt{[} \mid \texttt{]} \mid \texttt{`} \mid \texttt{\{} \mid \texttt{\}} & \\
\mathit{reservedop} & ::= & \texttt{..} \mid \texttt{:} \mid \texttt{::} \mid \texttt{=} \mid \backslash \mid \texttt{|} \mid \texttt{<-} \mid \texttt{->} & \\
 & \mid & \texttt{@} \mid \sim \mid \texttt{=>} & \\
\mathit{ascDigit} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{...} \mid \texttt{9} & \\
\mathit{ascSmall} & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{...} \mid \texttt{z} & \\
\mathit{ascLarge} & ::= & \texttt{A} \mid \texttt{B} \mid \texttt{...} \mid \texttt{Z} & \\
\mathit{ascSymbol} & ::= & \texttt{!} \mid \texttt{\#} \mid \texttt{\$} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{*} \mid \texttt{+} \mid \texttt{.} \mid \texttt{/} \mid \texttt{<} & \\
 & \mid & \texttt{=} \mid \texttt{>} \mid \texttt{?} \mid \texttt{@} \mid \backslash \mid \hat{} \mid \texttt{|} \mid \texttt{-} \mid \sim & \\
\mathit{uniDigit} & ::= & \text{Any Unicode decimal digit} & \\
\mathit{uniSmall} & ::= & \text{Any Unicode lowercase letter} & \\
\mathit{uniLarge} & ::= & \text{Any uppercase or titlecase Unicode letter} & \\
\mathit{uniSymbol} & ::= & \text{Any Unicode symbol or punctuation} & \\
\mathit{digit} & ::= & \mathit{ascDigit} \mid \mathit{uniDigit} & \\
\mathit{small} & ::= & \mathit{ascSmall} \mid \mathit{uniSmall} \mid \_ & \\
\mathit{large} & ::= & \mathit{ascLarge} \mid \mathit{uniLarge} & \\
\mathit{symbol} & ::= & \mathit{ascSymbol} \mid \mathit{uniSymbol}_{<\mathit{special} \mid \_ \mid : \mid " \mid '>} & \\
\mathit{reserveid} & ::= & \texttt{case} \mid \texttt{class} \mid \texttt{data} \mid \texttt{default} \mid \texttt{deriving} \mid \texttt{do} \mid \texttt{else} & \\
 & \mid & \texttt{if} \mid \texttt{import} \mid \texttt{in} \mid \texttt{infix} \mid \texttt{infixl} \mid \texttt{infixr} \mid \texttt{instance} & \\
 & \mid & \texttt{let} \mid \texttt{module} \mid \texttt{newtype} \mid \texttt{of} \mid \texttt{then} \mid \texttt{type} \mid \texttt{where} \mid \_ & \\
\mathit{varid} & ::= & (\mathit{small}\ \{\mathit{small} \mid \mathit{large} \mid \mathit{digit} \mid \texttt{'}\})_{<\mathit{reserveid}>} & \\
\mathit{conid} & ::= & \mathit{large}\ \{\mathit{small} \mid \mathit{large} \mid \mathit{digit} \mid \texttt{'}\} & \text{(constructors)} \\
\mathit{consym} & ::= & (\ \texttt{:}\ \{\mathit{symbol} \mid \texttt{:}\}\ )_{<\mathit{reservedop}>} & \\
\mathit{tyvar} & ::= & \mathit{varid} & \text{(type variables)} \\
\mathit{tycon} & ::= & \mathit{conid} & \text{(type constructors)} \\
\mathit{simpletype} & ::= & \mathit{tycon}\ \mathit{tyvar}_1\ \ldots\ \mathit{tyvar}_k & (k \geq 0) \\
\mathit{modid} & ::= & \mathit{conid} & \text{(modules)} \\
\mathit{qtycon} & ::= & [\,\mathit{modid}\,\texttt{.}\,]\ \mathit{tycon} & \\
\mathit{gtycon} & ::= & \mathit{qtycon} & \\
 & \mid & \texttt{()} & \text{(unit type)} \\
 & \mid & \texttt{[]} & \text{(list constructor)} \\
 & \mid & \texttt{->} & \text{(function constructor)} \\
 & \mid & \texttt{(}\ \texttt{,}\ \{\texttt{,}\}\texttt{)} & \text{(tupling constructor)}
\end{array}
$$

$$
\begin{array}{rll}
\textit{btype} ::= & [\,\textit{btype}\,]\ \ \textit{atype} & \text{(type application)} \\
\textit{type} ::= & \textit{btype}\ \ [\,\texttt{->}\ \ \textit{type}] & \text{(function type)} \\
\textit{atype} ::= & \textit{gtycon} & \\
 & \mid\ \textit{tyvar} & \\
 & \mid\ \texttt{(}\textit{type}_1, ..., \textit{type}_k\texttt{)} & \text{(tuple type, } k \geq 2\text{)} \\
 & \mid\ \texttt{[}\textit{type}\texttt{]} & \text{(list type)} \\
 & \mid\ \texttt{(}\textit{type}\texttt{)} & \text{(parenthesized constructor)} \\
\textit{con} ::= & \textit{conid}\ \mid\ (\textit{consym}) & \text{(constructor)} \\
\textit{constr} ::= & \textit{con}\ \ [\texttt{!}]\ \ \textit{atype}_1 \ldots\ \ [\texttt{!}]\ \ \textit{atype}_k & \text{(arity } \textit{con} = k,\ k \geq 0\text{)} \\
\textit{constrs} ::= & \textit{constr}_1\ \mid\ \ldots\ \mid\ \textit{constr}_n & (n \geq 0) \\
\textit{topdecl} ::= & \texttt{data}\ \ \textit{simpletype}\ \ \texttt{=}\ \ \textit{constrs} &
\end{array}
$$

2.2 Prolog, the logical guy

Prolog is a logic programming language associated with artificial intelligence and computational linguistics.

Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is intended primarily as a declarative programming language: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations.

A logic program is a set of axioms, or rules, defining relations between objects. A computation of a logic program is a deduction of the consequences of the program. A program defines a set of consequences, which is its meaning The art of logic programming is constructing concise and elegant programs that have the desired meaning.

Prolog is a practical and efficient implementation of many aspects of "intelligent" program execution, such as non-determinism, parallelism, and pattern-directed procedure call. It provides a uniform data structure called the *term*, from which all data, as well as Prolog programs, are constructed. A Prolog program consists of a set of clauses, where each clause is either a fact about the given information or a rule about how the solution may relate to or be inferred from the given facts. Thus Prolog can be seen as a first step towards the ultimate goal of programming in logic.

In another way, in Prolog, a program logic is expressed in terms of relations, and a computation is initiated by running a query over these relations. Relations and queries are constructed using Prolog's single data type, the *term*. Relations are defined by *clauses*. Given a query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted, i.e., an instantiation for all free variables is found that makes the union of clauses and the singleton set consisting of the negated query false, it follows that the original query, with the found instantiation applied, is a logical consequence of the program. This makes Prolog (and other logic programming languages) particularly useful for database, symbolic mathematics, and language parsing applications.

2.2.1 Getting Started

Let's show the essential elements of the Prolog in real programs, but without becoming diverted by details, formal rules, and exceptions, before getting started with its specification.

Computer programming in Prolog consists of:

- Specifying some *facts* about object and their relationships
- Defining some *rules* about objects and their relationships
- Asking *questions* about objects and their relationships

For example, Let's suppose we told to Prolog system our rule about the *father relationship*. Let's suppose that Anakin's son is Luke, Julio's son is Anakin and there is a fourth guy, Manuel, who is just Luke's friend. Therefore, we can observe that just Anakin is related to Luke and Julio is related to Anakin by the *father relationship*. These logical rules can be written in Prolog as follow:

```
father(anakin, luke).
father(julio, anakin).
```

And if we type in Prolog-CLI `father(anakin, luke)` it returns `true`, also with `father(julio, anakin)`. However, if we try to type `father(julio, manuel)` or whatever with Manuel, (or any other combination that does not represent the relationship written before) it will return `false`.

Facts

The simplest kind of statement is called a *fact*. Facts are a means of stating that a relation holds between objects. In the example above, the expression `father(anakin, luke)` is a fact which says that `anakin` is the *father* of `luke`. Names of the individuals are known as *atoms*.

The *plus* relation can be realized via a set of facts that defines the addition table. An initial segment of the table is:

```
plus(0, 0, 0).
plus(0, 1, 1).
plus(1, 0, 1).
plus(1, 1, 2).
plus(0, 2, 2).
plus(2, 0, 2).
```

This is not the best approach to defining *plus* relation but it is good enough to catch the idea.

Queries

The second form of a statement in Prolog is a *query*. Queries are a means of retrieving information from a logic program. A query asks whether a certain relation holds between objects. For example, the query `father(anakin, luke)?` asks whether the *father* relationship holds between `anakin` and `luke`. Given the facts written before, the answer to this query is *yes*.

The Logical Variable, Substitution, and Instances

A logical variable stands for an unspecified individual and is used accordingly. Consider its use in queries. Suppose we want to know of whom `luke` is the father. One way is to ask a series of queries, `father(julio, luke)?`, `father(manuel, luke)?`, `father(anakin, luke)?`, until an answer *yes* is given.

A variable allows a better way of expressing the query as `father(X, luke)?` to which the answer is `X=anakin`. Used in this way, *variables are a means of summarizing many queries*. A query containing a variable asks whether there is a value for the variable that makes the query a logical consequence of the program.

Definition (Variable)**.** *Variables* in logic programs behave differently from variables in conventional programming languages. They stand for an unspecified but single entity rather than for a store location in memory.

When Prolog uses a variable, the variable can be either *instantiated* or *not instantiated*

Definition (Instantiation)**.** A variable is *instantiated* when there is an object that the variable stands for. A variable is not instantiated when what the variable stands for is not yet known.

Having introduced variables, we can define a *term*.

Definition (Term)**.** A *term* is the single data structure in logic programs. The definition is inductive. Constants and variables are terms. Also, compound terms or structures are terms.

Prolog can distinguish variables from terms because any term beginning with a capital letter is taken to be a variable.

Definition (Compound Term)**.** A *compound term* comprises a **functor** (called the principal functor of the term) and a sequence of one or more arguments, which are terms. A *functor* is characterized by its name, which is an atom, and its arity or number of arguments.

Sintatically, compound terms have the form $f(t_1, t_2, \ldots, t_n)$, where the functor has name f and is of arity n and the t_i with $1 \leq i \leq n$, are the arguments.

Examples of compound terms include `s(0)`, `hot(milk)`, `name(john,doe)`, `list(a,list(b,nil))`, `foo(X)`, and `tree(tree(nil,3,nil) ,5,R)`.

Queries, goals, and more general terms where variables do not occur are called *ground*. Where variables do occur, they are called *nonground*. For example, `foo(a, b)` is *ground*, whereas `bar(X)` is *nonground*.

Definition (Substitution)**.** A *substitution* is a finite set (possibly empty) of pairs of the form $X_i = t_i$, where X_i is a variable and t_i is a *term*, and $X_i \neq X_j$, for every $i \neq j$, and X_i does not occur in t_j, for any i and j.

An example of a substitution consisting of a single pair is `X=anakin`. Substitution can be applied to terms. The result of applying a substitution σ to a term W, denoted by $W\sigma$, is the term obtained by replacing every occurrence of X by t in W, for every pair $X = t$ in σ.

The result of applying `X=anakin` to the term `father(X, luke)` is the term `anakin, luke)`.

Definition (Instance)**.** X is an *instance* of W if there is a substitution σ such that $X = W\sigma$

The goal `father(anakin, luke)` is an instance of `father(X, luke)` by definition. Similarly, `plus(1,0,1)` is an (one of them) instance of `plus(1,Y,X)`.

Those definitions are enough to encourage us to talk about the grammar of Prolog.

Rules

Conjunctive queries are defining relationships in their own right. We can add to our facts the following table:

```
sith(anakin).
jedi(luke).
jedi(julio).
```

The query `father(X,Y),sith(X)?` is asking for a *father* which is also a *sith*.
The query `father(anakin,X),jedi(X)?` is asking if there exists an `anakin`'s son who is also a `jedi`.

This brings us to the third and most important statement in logic programming, a *rule*, which enables us to define new relationships in terms of existing relationships.

Definition (Rules). *Rules* are statements of the form

$$A \leftarrow B_1, B_2, \ldots B_n$$

where $n \geq 0$. The goal A is the *head* of the rule, and the conjunction of goals $B_1, B_2, \ldots B_n$ is the body of the rule.

Rules, fats, and queries are also called *Horn clauses* or *clauses* for short. Note that a fact is just a special case of a rule when $n = 0$. Facts are also called *unit clauses.* We also have a special name for clauses with one goal in the body, namely, when $n = 1$. Such a clause is called an *iterative clause.* As for facts variables appearing in rules are universally quantified, and their scope is the whole rule.

A rule expressing the *enemy* relationship could be

`enemy(X,Y)` $\leftarrow$ `jedi(X),sith(Y).`

Similarly one can define a rule expressing the *master* relationship

`master(X,Y)` $\leftarrow$ `sith(X),sith(Y).`

Rules can be viewed in two ways. First, they are a means of expressing new or complex queries in terms of simple queries. A query `enemy(luke, Y)?` to the program that contains the preceding rule for `enemy` is translated to the query `jedi(luke), sith(Y)?` according to the rule. A new query about the `enemy` relationship has been built from simple queries involving `jedi` and `sith` relationships. Interpreting rules in this way is their *procedural* reading.

The second view of rules comes from interpreting the rule as a logical axiom. The backward arrow $\leftarrow$ is used to denote logical implication. The `enemy` rule reads: *X is an* `enemy` *of Y if X is a* `jedi` *and Y is a* `sith`.

Although formally all variables in a clause are universally quantified, we will sometimes refer to variables that occur in the body of the clause, but not in its head, as if they are existentially quantified inside the body.

For example, the `enemy` rule can be read: "For all X and Y, X is an enemy of Y if there exists a couple of X and Y, such that X is a Jedi and Y is a Sith". The formal justification for this verbal transformation will not be given, and we treat it just as a convenience.

Definition. The law of *universal modus ponens* says that from the rule

$$R = (A \leftarrow B_1, B_2, \ldots B_n)$$

and the facts B_1', B_2', $\ldots$ B_n', A' can be deduced if

$$A' \leftarrow B_1', B_2', \ldots B_n'$$

is an instace of R.

Definition. A *logical program* is a finite set of rules.

Definition. An existentially quantified goal G is a logical consequence of a program P if there is a clause in P with a ground instance $A \leftarrow B_1, B_2, \cdot B_n$ with $n \geq 0$ such that $B_1, B_2, \cdot B_n$ are logical consequences of P, and A is an instance of G.

Recursive Rules

The rules described so far define new relationships in terms of existing ones. An interesting extension is the recursive definition of relationships that define relationships in terms of themselves. One way of viewing recursive rules is a generalization of a set of nonrecursive rules.

Consider the problem of testing connectivity in a directed graph. A directed can be represented as a logic program by a collection of facts. A fact `edge(Node1, Node2)` is present in the program if there is an edge from `Node1` to `Node2` in the graph. Let's suppose the following directed graph program:

```
edge(a,b).   edge(b,d).   edge(d,e).
edge(a,c).   edge(c,d).   edge(f,g).
```

Two nodes are connected if there is a series of edges that can be traversed to get from the first node to the second. That is, the relation `connected(Node1, Node2)` is `true` if `Node1` and `Node2` are connected, is the transitive closure of the `edge` relation. We could represent the connected relationship as follow:

```
connected_one(N1, N2) <- edge(N1, N), edge(N, N2).
```

However, this just shows the relationship of *one-path-connexion*, i.e. the rule `connected_one(a,c)` returns `true`, but `connected_one(a,d)` returns `false`. We should define again a new one to reach the node `d` in our *connected* relationship.

```
connected_one(N1, N2) <- edge(N1, N), edge(N, N2).
connected_two(N1, N2) <- edge(N1, N), connected_one(N, N2).
```

Again, we have now the *one-path-connexion* and *two-path-connexion* relationships, i.e. the rule `connected_one(a,c)` returns `true`, `connected_two(a,d)` returns `true` but we don't have any way to reach the node `e`.

A clear pattern can be seen, which can be expressed in a rule defining the relationship `connected(Node1, Node2)` recursively.

```
connected(N,N).
connected(N1, N2) <- edge(N1, N), conected(N, N2).
```

2.2.2 Grammar specification

Prolog is hugely flexible concerning its grammar and the expressions that one can define. This helps do syntax transformation between Haskell to Prolog, and vice versa. We will works with a subset of Prolog expressions that conforms rules, variables and predicates. That is, expressions like this:

$$rule(Var1, Var2, \ldots,\ VarK) :-\ Goal1, Goal2, \ldots,\ GoalN.$$

In the next chapter, we will see step by step how we should define the primitive type generators, then we will talk about how each part of the ADT specification should be translated into Prolog expressions, and finally, we will define a generator for the ADTs.

Chapter 3

Syntax-Translation Mechanism

In this chapter, we will define our syntax-translation function which maps Haskell expression to Prolog expression. In order to do that, I will introduce you a first approach to it and then, we will see so many examples of what kind of changes it does and how to do that.

First of all, we will see a basic scenario where algebra data types are defined with fixed types, i.e., the monomorphic type cases. We will see how to translate primitive types and I will provide so many functions that help us to get those primitive types values guided by the syntax of Haskell's types. That is, what we need to translate the syntax of the type expression `Int`, `String`, or `Bool` is usually used in the Haskell functions or ADTs declaration to a Prolog program. Then, we will see what is the equivalent expression for some trivial algebra data types like *Maybe*, *Either*, *Binary Tree*, and so on.

We will talk about what kind of problems those generations carry on and how to deal with them. And finally, we will talk about polymorphism and the way for generating a value related to the equivalent Prolog *type* for a given Haskell's algebra data type declaration with polymorphic types. The purpose is to generalize those changes based on the translation of Haskell's formal grammar step by step doing refinements from the step back to the next one in a progressive improvement process.

3.1 Monomorphic Types

For our first steps, we can deal with the most basic case: those scenarios where ADT expressions are defined with monomorphic types. To do that, we can avoid keeping in mind the type or fixing the type expression in the formal grammar definition and simply try to generate their structure with it.

Let's define the syntax translation function. Let $\mathcal{H}$, $\mathcal{P}$ be the set of well-defined expressions in Haskell and Prolog, respectively. We can consider $\mathcal{H}_{\texttt{ADT}}$ and $\mathcal{P}_{\text{Clause}}$, the subset of well-defined algebra data types expressions in Haskell contained in $\mathcal{H}$ and the subset of clauses in $\mathcal{P}$, respectively. Following the formal grammar definition for algebra data type expressions in Haskell:

$$
\begin{array}{rcl}
\texttt{data}\ \textit{simpletype} & = & \textit{constrs} \\
 & & \updownarrow \\
\texttt{data}\ \textit{tycon} & = & \textit{constr}_1 \mid \textit{constr}_2 \mid \ldots \mid \textit{constr}_n \\
 & & \updownarrow \\
\texttt{data}\ \textit{tycon} & = & \textit{con}_1\ \ \textit{atype}_{1,1}\ \ \textit{atype}_{1,2}\ \ \ldots\ \ \textit{atype}_{1,k_1} \mid \\
 & & \textit{con}_2\ \ \textit{atype}_{2,1}\ \ \textit{atype}_{2,2}\ \ \ldots\ \ \textit{atype}_{2,k_2} \mid \\
 & & \vdots \\
 & & \textit{con}_n\ \ \textit{atype}_{n,1}\ \ \textit{atype}_{n,2}\ \ \ldots\ \ \textit{atype}_{n,k_n} \mid
\end{array}
$$

We define our syntax-tranlsation semantic

$$\text{HSyntax}: \ \mathcal{H}_{\texttt{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$$

for monomorphic types as is shown in appendix B.1.1.

Now we have defined our syntax-translation semantic for monomorphic types, we will see some examples that help us to understand better. But before that, I will introduce you generation functions for primitive types.

3.1.1 Primitive Types generators

For this work, I provide some kind of Prolog programs that will be useful to us to generate primitive types. These implementations are not the best efficient ones, but they are enough for our purpose. We need those generators in the case when *tycon* is a primitive type, because

$$\text{GSyntax} \ [\![tycon]\!] = gen\_\text{Lower} \ [\![tycon]\!]$$

`Char` type generator

```
%% Char type

char(C) :- char_code(C, _).

gen_char(C) :-
    repeat,
    random_between(0, 7935, Code),
    (char_code(C, Code), !; fail).
```

`String` type generator

There is a predicate `string` in the Prolog's prelude to check if an expression is a string or not. Therefore, we will provide just the `gen_string` rule.

```
%% String type

%% string(S) in the Prolog's prelude

gen_string(S) :-
   repeat,
   random_between(0, 99, N),
   (length(L, N), maplist(gen_char, L), atom_string(L,S), string(S), !; fail).
```

`Int` type generator

```
%% Int type

int(I) :- (I >= -536870912, I =< 536870912), integer(I).

gen_int(I) :- random_between(-536870912, 536870912, I), integer(I).
```

`Integer` type generator

There is a predicate `string` in the Prolog's prelude to check if an expression is a string or not. Therefore, we will provide just the `gen_string` rule. For our purpose, we don't need all the integer type sets, so here we decided to bound for a good enough range.

```
%% Integer type

%% integer(I) in the Prolog's prelude

gen_integer(I) :- random_between(-9999999999999999999999999999,
    9999999999999999999999999999, I).
```

`Bool` type generator

```
%% Boolean type

bool(true).
bool(false).

gen_bool(B) :- bool(B).
```

`Unit` type generator

```
%% Unit type

unit(unit).

gen_unit(U) :- unit(U).
```

`Double` type generator

In Prolog, `float` predicate is a *representation type*. You can have a numeric value that has a representation both as integer and float and these representations are distinguishable: values $1 \equiv 1.0$ but expressions `1` $\neq$ `1.0`.

```
%% Double type

double(D) :- float(D).

gen_double(D) :- gen_double(-9999999, 9999999, D).

gen_double(Min, Max, D) :-
    random(F),
    D is Min + F * (Max - Min).
```

3.1.2 Examples

Example 3.1.1 (MaybeInt)**.** Let's suppose the following Haskell ADT definition:

```
data MaybeInt = None | Some Int
```

Here, using the translation function defined above as is shown in appendix B.1.2, we can translate the Haskell definition shown above to the following Prolog program:

```
maybeint(none).                          %% rule 1
maybeint(some(X21)) :- gen_int(X21).     %% rule 2
```

Now, if you type `maybeint(X)` in the Prolog's CLI, you will get a valid `MaybeInt` generated value.

Example 3.1.2 (Non-Polymorphic Either)**.** Let's suppose our non-polymorphic version of the definition of Either in Haskell for both `String` and `Int` types:

```
data Either = Left String | Right Int
```

Here, using the translation function defined above as is shown in appendix B.1.3, we can translate the Haskell definition shown above to the following Prolog program:

```
either(left(X11))   :- gen_string(X11).     %% rule 1
either(right(X21))  :- gen_int(X21).        %% rule 2
```

Now, if you type `either(X)` in the Prolog's CLI, you will get a valid `Either` generated value.

Example 3.1.3 (MyListBool)**.** Let's suppose the following Haskell ADT definition:

```
data MyListBool = Nil | Cons Bool MyListBool
```

We can observe it is a recursive definition, so the question is, what happens with the declaration of `MyListBool` in its own declaration. Well, as `MyListBool` is a Haskell algebra data type, we can deal with it as we have been doing in a recursively way or, in another hand, we can define the Prolog rule as a recursive rule (explained in section 2.2.1).

Therefore, using the translation function defined above as is shown in appendix B.1.4, we can translate the Haskell definition shown above to the following Prolog program:

```
mylistbool(nil).                                                  %% rule 1
mylistbool(cons(X21, X22))  :- gen_bool(X21), mylistbool(X22).    %% rule 2
```

Now, if you type `mylistbool(X)` in the Prolog's CLI, you will get a valid `MyListBool` generated value. But, what happens here? Let's take a look at a few more examples before.

Example 3.1.4 (Binary Search Tree)**.** Here we go again... Let's suppose the following Haskell ADT definition:

```
data BSTree = Nil | T Int BSTree BSTree
```

We can observe it is a recursive definition, so the question is, what happens with the declaration of `BSTree` in its own declaration. Well, as `BSTree` is a Haskell algebra data type, we can deal with it as we have been doing in a recursively way or, in another hand, we can define the Prolog rule as a recursive rule (explained in section 2.2.1).

Therefore, using the translation function defined above as is shown in appendix B.1.5, we can translate the Haskell definition shown above to the following Prolog program:

```
bstree(nil).                                                          %% rule 1
bstree(t(X21, X22, X23)) :- gen_int(X21), bstree(X22), bstree(X23).   %% rule 2
```

Now, if you type `bstree(X)` in the Prolog's CLI, you will get a valid `BSTree` generated value. Again, what happens here? Did you see?

Example 3.1.5 (SomeWeird)**.** Let's suppose the following (really weird) Haskell ADT definition:

```
data SomeWeird = Nil1 |
                 Nil2 |
                 Some Int |
                 Weird SomeWeird SomeWeird Bool
```

We can observe it is a recursive definition, so the question is, what happens with the declaration of `SomeWeird` in its own declaration. Well, as `SomeWeird` is a Haskell algebra data type, we can deal with it as we have been doing in a recursively way or, in another hand, we can define the Prolog rule as a recursive rule (explained in section 2.2.1).

Therefore, using the translation function defined above as is shown in appendix B.1.6, we can translate the Haskell definition shown above to the following Prolog program:

```
someweird(nil1).                                                          %% rule 1
someweird(nil2).                                                          %% rule 2
someweird(some(X31)) :- gen_int(X31).                                     %% rule 3
someweird(weird(X41, X42, X43)) :-                                        %% rule 4
  someweird(X41),
  someweird(X42),
  gen_bool(X43).
```

Now, if you type `someweird(X)` in the Prolog's CLI, you will get a valid `SomeWeird` generated value. Oh my god... it was so weird. Finally, let's take a look at one more example.

Example 3.1.6 (Rose Tree)**.** Last dance! Let's suppose the following Haskell ADT definition:

```
data RSTree = R Int [RSTree]
```

We can observe it is a recursive definition. Again, as `RSTree` is a Haskell algebra data type, we can deal with it as we have been doing in a recursively way or, in another hand, we can define the Prolog rule as a recursive rule (explained in section 2.2.1). Also, it is defined using Haskell's list type constructor which we have not yet discussed.

Haskell lists are not really either an algebra data type or a primitive type. It is a kind of algebra structure which is not formally defined as a `data`-Haskell expression. Therefore, its implementation in Prolog is different. For now, we will implement a particular generator of lists of Rose Trees, but when we talk about polymorphism, I will provide a generic generator of lists of any type. So, here using the translation function defined above as is shown in appendix B.1.7, we can translate the Haskell definition shown above to the following Prolog program:

```
rstree(r(X11, X12)) :- gen_int(X11), list(rstree, X12).     %% rule 1
```

Now, if you type `rstree(X)` in the Prolog's CLI, you will get a valid `RSTree` generated value. Again, what happens here? Did you see it? No? Okay, let's talk.

3.2 Monomorphic Types with Boundaries

When we try to generate recursively a structure in Prolog, it tries to get all the space that the symbolic variables hold for the rules that participate in the resolution process. Moreover, Prolog has default resolution strategies that imply directly in the returned solutions. We won't deep into those details, if you are interested in how to implement more efficient resolution strategies, you can check [10]. What we want to get is completeness, a good enough space of generated values for our purpose.

To be more precise:

- In the case of the `MyListBool` algebra data type, we are just getting those values with `true`. We never get either `Cons(false, Nil)` or `Cons(false, Cons(true, Cons(false, Nil)))`.
- In the case of `BSTree`, we are resolving deeply the right-branch side, getting always `nil` in the left-branch side, for example. Our Binary Search Tree Prolog program returns us:

  `T(-483997780, Nil, T(349868009, Nil, T(-265947283, Nil, Nil)))`

  and never

  `T(-483997780, T(-265947283, Nil, Nil), T(349868009, Nil, Nil))`

- In the case of `RSTree`, we are resolving deeply the inner-list side instead of getting another Rose Tree value in the most out of the list. Our Rose Tree Prolog program returns us:

  `R(-193684758, [R(-178049475, [R(186461544, [])])])`

  and never

  `R(-193684758, [R(-178049475, []), R(186461544, [])])`

An approach that fits very well is to define boundaries for recursive structures. We can define the limit of elements in a list, or the depth in the branch of a binary search tree. For instance:

Let's suppose the following Haskell ADT definition showed before:

```
data MyListBool = Nil | Cons Bool MyListBool
```

We know that using the syntax-translate mechanism HSyntax defined in 3.1, we obtaing the following Prolog program:

```
mylistbool(nil).                                              %% rule 1
mylistbool(cons(X21, X22))  :- gen_bool(X21), mylistbool(X22).    %% rule 2
```

Let's consider a new variable `Length` that represents the length of `MyListBool`. We can refine our last program into this new one

```
mylistbool(0, nil).                                           %% rule 1
mylistbool(Length, cons(X21, X22)) :-                         %% rule 2
  gen_bool(X21),
  Length > 0,
  N1 is Length - 1,
  mylistbool(N1, X22).
```

For the base case when `Length` $= 0$, our program returns Nil. Otherwise, it returns

$$Cons(b_1, Cons(b2, ...Cons(b_n, Nil)))$$

Now, if you type `mylistbool(N, X)` in the Prolog's CLI, you will get a valid `MyListBool` generated value:

- Let `N` $= 1$, our program returns:

  ```
  cons(true, nil)
  cons(false, nil)
  ```

- Let `N` $= 2$, our program returns:

  ```
  cons(true, cons(true, nil))
  cons(true, cons(false, nil))
  cons(false, cons(true, nil))
  cons(false, cons(false, nil))
  ```

We have just seen that just adding a variable allows us to get more control over the space of solutions. So, we need in some way to include this new feature in our syntax-translate mechanism as a refinement of the old one.

However, if we talk about the `MyListBool` algebra data type and how their expressions are constructed, we can define $\Gamma_{MyListBool}$ the set of expressions that define the structure of the elements belong to `MyListBool` type.

$$\Gamma_{MyListBool} = \{Nil,\ \ Cons(b, Nil),\ \ Cons(b, Cons(b, Nil)),\ \ldots\} \quad b \in \texttt{Bool}$$

And we can consider the binary operation $\preceq$ defined as, for all e_1, e_2 two expressions, we say that $e_1 \preceq e_2$ if the expression of e_1 is a subexpression contained in the expression e_2. Then, we can say that $(\Gamma_{MyListBool}, \preceq)$ is a poset, where Nil is the minimal element.

$$\begin{array}{c} \vdots \\ | \\ Con(b, Cons(b, Cons(b, Nil))) \\ | \\ Con(b, Cons(b, Nil)) \\ | \\ Cons(b, Nil) \\ | \\ Nil \end{array}$$

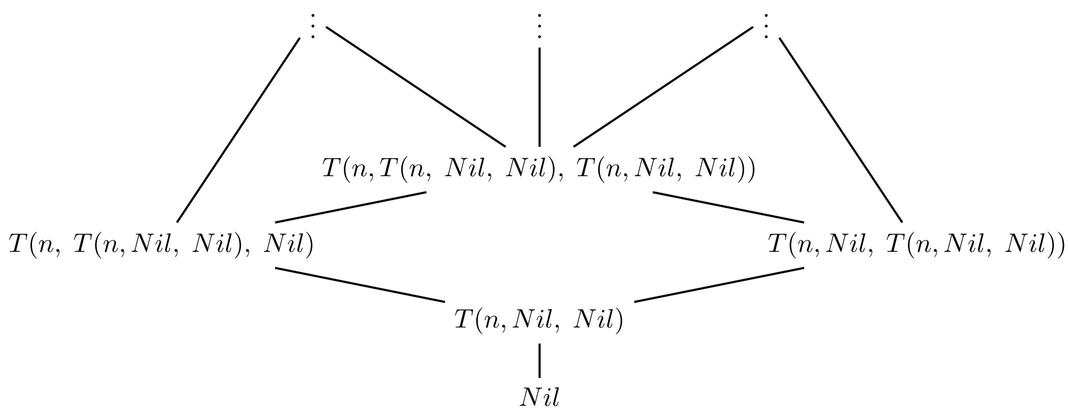
In this case, the structural induction of the `MyListBool` algebra data type grows in a linear-deep way. However, that *easy* grow is not the only one.

All right, let's take a look now at the `BSTree` expressions. In this case, we have a recursion definition in the two branches on the second constructor in the definition of `BSTree` algebra data type. The structural induction derives its shape in two possible nodes, the left one and the right one.

Let Γ_{BSTree} be the set of expressions that define the structure of an element in the `BSTree` type.

$$\begin{aligned}\Gamma_{BSTree} \;=\; &\{Nil,\;\; T(n, Nil,\; Nil),\\ &\; T(n,\; T(n, Nil,\; Nil),\; Nil),\;\; T(n, Nil,\; T(n, Nil,\; Nil)),\\ &\; T(n, T(n,\; Nil,\; Nil),\; T(n, Nil,\; Nil)),\; \ldots\} \quad n \in \texttt{Int}\end{aligned}$$

Therefore, $(\Gamma_{BSTree}, \preceq)$ is a poset, where Nil is the minimal element.



So, the concept of boundaries is not clear and, the more complex the structure more difficult defining the boundaries. In the two examples shown before, is confusing to get the idea of *HOW* grow but it seems that is easy to identify who is the minimal structure right? Both cases are the Nil element. However, what happens with the `SomeWeird` algebra data type introduced in the last section?

Here, we have a recursion definition in the two branches on the fourth constructor in the definition of `SomeWeird` algebra data type. Similar to the `BSTree` algebra data type, the structural induction derives its shape in two possible nodes, the left one and the right one. However, in this case, we have two 'Nil' elements: $Nil1$ and $Nil2$. Moreover, we have a third *intermediate* structure $Some$.

Let $\Gamma_{SomeWeird}$ be the set of expressions that define the structure of an element in the `SomeWeird` type.

$$\begin{aligned}\Gamma_{SomeWeird} \;=\; &\{Nil1,\;\; Nil2,\;\; Some(n),\\ &\; Weird(Nil1,\; Nil1,\; b),\;\; Weird(Nil1,\; Nil2,\; b),\\ &\; Weird(Nil2,\; Nil1,\; b),\;\; Weird(Nil2,\; Nil2,\; b),\\ &\; Weird(Nil1,\; Some(n),\; b),\;\; Weird(Nil2,\; Some(n),\; b),\; \ldots\} \quad n \in \texttt{Int} \;\; b \in \texttt{Bool}\end{aligned}$$

Therefore, $(\Gamma_{SomeWeird}, \preceq)$ is a poset, where $Nil1$, $Nil2$ and $Some(n)$ with $n \in \texttt{Int}$ are the minimal elements. Therefore, it wasn't also clear who the minimal element was (in fact, we see there may be several).

In order to define a general definition of boundary or a well-known structure *size* concept for getting a good-enough space of solutions that hold and providing us completeness in a systematic way, we need to identify those scenarios where an algebra data type is defined recursively and keep in mind the concept of induction. More precisely, mathematical induction over natural numbers or *recursion*, and structural induction. We will see how recursion will help us to control our concept of *size*, and the structural induction will help us to identify which ones are the minimal elements in the poset of the algebra data type expressions.

Of course, this is not the only solution. There are a lot of ways to implement the concept of *size* in Prolog, we will show an approach that fits so well with our work and more important: it will be built following the syntax of Haskell's formal grammar, getting consistency between our implementation and the mechanism that we have been defining.

Let's define the syntax translation function. Let $\mathcal{H}$, $\mathcal{P}$ be the set of well-defined expressions in Haskell and Prolog, respectively. We can consider $\mathcal{H}_{\texttt{ADT}}$ and $\mathcal{P}_{\text{Clause}}$, the subset of well-defined algebra data types expressions in Haskell contained in $\mathcal{H}$ and the subset of clauses in $\mathcal{P}$, respectively. Let HSyntax be our syntax-translate mechanism function defined before. We define HBSyntax a refinement of the function HSyntax

$$\text{HBSyntax}: \ \mathcal{H}_{\texttt{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$$

as an extension of our syntax-translation semantic for monomorphic types with boundaries as is shown in appendix B.2.1.

where:

- Let *tycon* be the algebra data type. We define Γ_{tycon} as the set of expressions that define the structure of the elements belonging to *tycon* type and let's consider the binary operation $\preceq$ defined as, for all e_1, e_2 two expressions, we say that $e_1 \preceq e_2$ if the expression of e_1 is a subexpression contained in the expression e_2. Then, we can say that $(\Gamma_{tycon}, \preceq)$ is a poset.

- For all con_i with $i \in \{1, \ldots, n\}$, there exist two $\texttt{rule}_i^0$ and $\texttt{rule}_i$ Prolog rules that hold and generate the corresponding translated expression from the Haskell's ADT constructor i. The rule $\texttt{rule}_i^0$ represents the base case in the structural induction for the constructor con_i and $\texttt{rule}_i$ represents the structural induction scenario.

  In the case of the ADT constructor, con_i doesn't have any component $atype_{i,j}$ or for all $j \in \{1, \ldots, k_i\}$ there is no exists any $atype_{i,j}$ with recursive type, therefore we can omit the inductive $\texttt{rule}_i$.

- For all con_i with $i \in \{1, \ldots, n\}$, there exists a $N_i^0 \in \mathbb{N}$, that represents the minimal *size* (on our *size* concept) of the constructor con_i. It will be the base case in the *recursion.*

  In case of the ADT constructor con_i doesn't have any component $atype_{i,j}$ or for all $j \in \{1, \ldots, k_i\}$ there is no exists any $atype_{i,j}$ with recursive type, therefore we can define $N_i^0 = 0$.

- For all con_i with $i \in \{1, \ldots, n\}$, there exists a variable N_i in Prolog, that represents the symbolic variable which will be used to do the recursion scenario from the base case $N_i^0 \in \mathbb{N}$, that is, N_i represent the $m \in \mathbb{N}$ such that if $m > N_i^0$ implies $rule_i$ hold.

- There exists a fact *base* such that for all $i \in \{1, \ldots, n\}$, $base(N_i^0)$ holds.

- For all $aytpe_{i,j}$ with $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, k_i\}$, there exists a variable $Z_{i,j}$ in Prolog, that represents the symbolic variable for the generation of the base case number in the recursion steps, N_i^0, for the position j in the constructor i which is the Haskell type $atype_{i,j}$ in the Haskell's ADT definition.

  Hence, getting the natural number which represents the minimal *size* (on our *size* concept) of the constructor j, we will obtain the expression of the minimal element that the constructor con_i can build.

- Let $m \in \mathbb{N}$ be, with $m > N_i^0$. For all $atype_{i,j}$ with $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, k_i\}$, there exists a variable $W_{i,j}$ in Prolog, that represents the symbolic variable for the generation of the natural $s \in [N_i^0, m)$ which represent the *size* equal s (on our *size* concept); and a predicate $states_i$ such that $states_i(s,\ W_{i,1},\ \ldots,\ W_{i,k_i})$ holds $\Longleftrightarrow$ for all $b, b' \in \{1, \ldots, k_i\}$, $W_{i,b} = W_{i,b'} \Rightarrow b = b'$.

For this approach, we can implement the functions $state_i$ as:

$$
\begin{aligned}
state_i(Nr_i,\ W_{i,1},\ \dots,\ W_{i,k_i}) :-\ & cases(Nr_i, W_{i,1}),\\
& cases(Nr_i, W_{i,2}),\\
& \vdots\\
& cases(Nr_i, W_{i,k_i}),\\
& diff_i(W_{i,1},\ \dots,\ W_{i,k_i}).
\end{aligned}
$$

where

$$
\begin{aligned}
& cases(\_, Y) :-\ base(Y). && diff_i(X,\ \overset{k_i}{\dots},\ X) :-\ base(X),\ !,\ \texttt{fail}.\\
& cases(X, X). && diff_i(W_{i,1},\ \dots,\ W_{i,k_i}).
\end{aligned}
$$

With that, we can obtain an intermediate expression with *size* s in the poset $(\Gamma_{tycon}, \preceq)$. An expression strictly belongs with respect $\preceq$ relation, between the $rule_i^0$ generate (i.e. the base case in the structural induction) and that one $rule_i$ generate, in the position j from the constructor i. Shortly, the expression of the element with *size* s that the constructor con_i can build.

3.2.1 Examples

Example 3.2.1 (MyListBool)**.** Let's suppose the following Haskell ADT definition:

```
data MyListBool = Nil | Cons Bool MyListBool
```

Using HSyntax, in the las examples 3.1.2 we obtain:

```
mylistbool(nil).                                                %% rule 1
mylistbool(cons(X21, X22))  :- gen_bool(X21), mylistbool(X22).     %% rule 2
```

Now, if we use the newer refinement function HBSyntax:

- Let $\Gamma_{MyListBool}$ the set of expressions that define the structure of an element in the `MyListBool` type.

  $$\Gamma_{MyListBool} = \{Nil,\ \ Cons(b, Nil),\ \ Cons(b, Cons(b, Nil)),\ \dots\} \quad b \in \texttt{Bool}$$

  We can consider $(\Gamma_{MyListBool}, \preceq)$ that is a poset.

- As we have two consutrctor con_1 and con_2, therefore we have four rules instead, $\texttt{rule}_1^0$, $\texttt{rule}_1$, $\texttt{rule}_2^0$ and $\texttt{rule}_2$.

- As `Nil` doesn't have any type in its definition, that is, there is not any $atype_{1,j}$ in its Haskell definition. In fact, by using HBSyntax we know that $\texttt{rule}_1$ is a fact `mylistbool(nil)`, therefore we don't need the $\texttt{rule}_1$ (because we won't do recursion on `nil`) and we can consider $N_1^0 = 0$, which means, we will consider Nil as the minimal structure in `MyListBool` with respect our *size* concept.

  For the second constructor, there exists two $atype_{2,1}$ and $atype_{2,2}$ related to the expression `Cons Bool MyListBool`, that is $atype_{2,1} = \texttt{Bool}$ and $atype_{2,2} = \texttt{MyListBool}$, so here we have to do structural induction in one place: $atype_{2,2}$.

  Therefore, there exists a natural N_2^0, in this case $N_2^0 = 1$, and a variables $Z_{2,2}$ such that $\texttt{rule}_2^0$ holds and is defined as follow:

$$
\begin{aligned}
\texttt{rule}_2^0:\ \ mylistbool(N_2^0,\ \ t(X_{2,1},\ \ X_{2,2})) :-\ & base(Z_{2,2}),\\
& gen_{\texttt{Bool}}(X_{2,1}),\\
& mylistbool(Z_{2,2},\ \ X_{2,2}),
\end{aligned}
$$

This means that $\texttt{rule}_2^0$ will generate our structural induction base case over the constructor `cons` and we will obtains those elements with size $N_2^0 = 1$. Also, there exists N_2 and $W_{2,2}$ variables such that $\texttt{rule}_2$ holds and is defined as follow:

$$\begin{aligned}\texttt{rule}_2:\ \ \mathit{mylistbool}(N_2,\ \ t(X_{2,1},\ \ X_{2,2})) :-\ \ & N_2 > N_2^0, \\ & Nr_2\ \texttt{is}\ N_2 - 1, \\ & \mathit{states}_2(Nr_2,\ \ W_{2,2}), \\ & \mathit{gen}_{\texttt{Bool}}(X_{2,1}), \\ & \mathit{mylistbool}(W_{2,2},\ \ X_{2,2}),\end{aligned}$$

So finally, we can translate the Haskell definition shown above to the following Prolog program:

```
%%%%%%%%%%%%%%%%%%%%%%
%% MyListBool ADT
%%%%%%%%%%%%%%%%%%%%%%

% rule01
% Base case for nil
mylistbool(0,nil).

% rule02
% Base case for cons(_,_)
mylistbool(1, cons(X21, X22)) :-
    base(Z22),
    gen_bool(X21),
    mylistbool(Z22, X22).

% rule2
% Inductive scenario for cons(_,_,_)
mylistbool(N2, cons(X21, X22)) :-
    N2 > 1,
    Nr2 is N2 - 1,
    states2(Nr2, W22),
    gen_bool(X21),
    mylistbool(W22, X22).


%%%%%%%%%%%%%%%%%
%% Auxiliar rules
%%%%%%%%%%%%%%%%%

% base rules
base(0).

% cases rule
cases(_, Y) :- base(Y).
cases(X, X).

% diff rules
diff2(X) :- base(X), !, fail.
diff2(W22).

% states rules
states2(Nr2, W22) :-
    cases(Nr2, W22),
    diff2(W22).
```

Now, if you type `mylistbool(N, X)` in the Prolog's CLI, you will get a set of valid `MyListBool` generated value which holds the completeness we have been searching for.

For instance:

- Let $\texttt{N} = 0$, our program returns:

  `nil`

- Let N $= 1$, our program returns:

```
cons(true, nil)
cons(false, nil)
```

- Let N $= 2$, our program returns:

```
cons(true, cons(true, nil))
cons(true, cons(false, nil))
cons(false, cons(true, nil))
cons(false, cons(false, nil))
```

Example 3.2.2 (Binary Search Tree)**.** Let's suppose the following Haskell ADT definition:

```
data BSTree = Nil | T Int BSTree BSTree
```

Using HSyntax, in the las examples 3.1.2 we obtain:

```
bstree(nil).                                                        %% rule 1
bstree(t(X21, X22, X23)) :- gen_int(X21), bstree(X22), bstree(X23).    %% rule 2
```

Now, if we use the newer refinement function HBSyntax:

- Let Γ_{BSTree} the set of expressions that define the structure of an element in the BSTree type.

$$\begin{aligned}\Gamma_{BSTree} \;\; = \;\; & \{Nil,\;\; T(n, Nil,\; Nil),\\ & T(n,\; T(n, Nil,\; Nil),\; Nil),\;\; T(n, Nil,\; T(n, Nil,\; Nil)),\\ & T(n, T(n,\; Nil,\; Nil),\; T(n, Nil,\; Nil)),\; \ldots\} \quad n \in \texttt{Int}\end{aligned}$$

  We can consider $(\Gamma_{BSTree}, \preceq)$ that is a poset.

- As we have two consutrctor con_1 and con_2, therefore we have four rules instead, $\texttt{rule}_1^0$, $\texttt{rule}_1$, $\texttt{rule}_2^0$ and $\texttt{rule}_2$.

- As Nil doesn't have any type in its definition, that is, there is not any $atype_{1,j}$ in its Haskell definition. In fact, by using HBSyntax we know that $\texttt{rule}_1$ is a fact bstree(nil), therefore we don't need the $\texttt{rule}_1$ (because we won't do recursion on nil) and we can consider $N_1^0 = 0$, which means, we will consider Nil as the minimal structure in BSTree with respect our *size* concept.

  For the second constructor, there exists three $atype_{2,1}$, $atype_{2,2}$ and $atype_{2,3}$ related to the expression T Int BSTree BSTree, that is $atype_{2,1} = \texttt{Int}$, $atype_{2,2} = \texttt{BSTree}$ and $atype_{2,3} = \texttt{BSTree}$, so here we have to do structural induction in two places: $atype_{2,2}$ and $atype_{2,3}$.

  Therefore, there exists a natural N_2^0, in this case $N_2^0 = 1$; two variables $Z_{2,2}$ and $Z_{2,3}$ such that $\texttt{rule}_2^0$ holds and is defined as follow:

$$\begin{aligned}\texttt{rule}_2^0:\;\; bstree(N_2^0,\;\; t(X_{2,1},\;\; X_{2,2},\;\; , X_{2,3})) :- \;\; & base(Z_{2,2}),\\ & base(Z_{2,3}),\\ & gen_{\texttt{Int}}(X_{2,1}),\\ & bstree(Z_{2,2},\;\; X_{2,2}),\\ & bstree(Z_{2,3},\;\; X_{2,3}).\end{aligned}$$

  This means that $\texttt{rule}_2^0$ will generate our structural induction base case over the constructor t and we will obtains those elements with size $N_2^0 = 1$. Also, there exists N_2, $W_{2,2}$ and $W_{2,3}$

variables such that $\texttt{rule}_2$ holds and is defined as follow:

$$
\begin{aligned}
\texttt{rule}_2:\ \ bstree(N_2,\ \ t(X_{2,1},\ \ X_{2,2},\ \ ,X_{2,3})) :-\ \ & N_2 > N_2^0, \\
& Nr_2\ \texttt{is}\ N_2 - 1, \\
& states_2(Nr_2,\ \ W_{2,2},\ \ W_{2,3}), \\
& gen_{\texttt{Int}}(X_{2,1}), \\
& bstree(W_{2,2},\ \ X_{2,2}), \\
& bstree(W_{2,3},\ \ X_{2,3}).
\end{aligned}
$$

So finally, we can translate the Haskell definition shown above to the following Prolog program:

```
%%%%%%%%%%%%%%%%%%%%%%
%% Binary Search Tree
%%%%%%%%%%%%%%%%%%%%%%

% rule01
% Base case for nil
bstree(0,nil).

% rule02
% Base case for t(_,_,_)
bstree(1, t(X21, X22, X23)) :-
    base(Z22),
    base(Z23),
    gen_int(X21),
    bstree(Z22, X22),
    bstree(Z23, X23).

% rule2
% Inductive scenario for t(_,_,_)
bstree(N2, t(X21, X22, X23)) :-
    N2 > 1,
    Nr2 is N2 - 1,
    states2(Nr2, W22, W23),
    gen_int(X21),
    bstree(W22, X22),
    bstree(W23, X23).


%%%%%%%%%%%%%%%%%
%% Auxiliar rules
%%%%%%%%%%%%%%%%%

% base rules
base(0).

% cases rule
cases(_, Y) :- base(Y).
cases(X, X).

% diff rules
diff2(X, X) :- base(X), !, fail.
diff2(W22, W23).

% states rules
states2(Nr2, W22, W23) :-
    cases(Nr2, W22),
    cases(Nr2, W23),
    diff2(W22, W23).
```

Now, if you type `bstree(N, X)` in the Prolog's CLI, you will get a set of valid `BSTree` generated values which hold the completeness we have been searching for.

For instance:

- Let $\texttt{N} = 0$, our program returns:

  `nil`

- Let $\texttt{N} = 1$, our program returns:

  `t(-245966722, nil, nil)`

- Let $\texttt{N} = 2$, our program returns:

  ```
  t(208235124, nil, t(158289014, nil, nil))
  t(308826753, t(-184602937, nil, nil), nil)
  t(302818108, t(324337197, nil, nil), t(295232001, nil, nil))
  ```

Example 3.2.3 (SomeWeird)**.** Let's suppose the following Haskell ADT definition:

```
data SomeWeird = Nil1 |
                 Nil2 |
                 Some Int |
                 Weird SomeWeird SomeWeird Bool
```

Using HSyntax, in the las examples 3.1.2 we obtain:

```
someweird(nil1).                                              %% rule 1
someweird(nil2).                                              %% rule 2
someweird(some(X31)) :- gen_int(X31).                         %% rule 3
someweird(weird(X41, X42, X43)) :-                            %% rule 4
  someweird(X41),
  someweird(X42),
  gen_bool(X43).
```

Now, if we use the newer refinement function HBSyntax:

- Let $\Gamma_{SomeWeird}$ the set of expressions that define the structure of an element in the `SomeWeird` type.

$$
\begin{aligned}
\Gamma_{SomeWeird} \quad = \quad & \{Nil1,\ Nil2,\ Some(n),\\
& Weird(Nil1,\ Nil1,\ b),\ \ Weird(Nil1,\ Nil2,\ b),\\
& Weird(Nil2,\ Nil1,\ b),\ \ Weird(Nil2,\ Nil2,\ b),\\
& Weird(Nil1,\ Some(n),\ b),\ \ Weird(Nil2,\ Some(n),\ b),\ \ldots\}\\
& n \in \texttt{Int} \ \ b \in \texttt{Bool}
\end{aligned}
$$

  We can consider $(\Gamma_{SomeWeird}, \preceq)$ that is a poset.

- As we have four consutrctors con_1, con_2, con_3 and con_4, therefore we have eight rules instead, $\texttt{rule}_1^0$, $\texttt{rule}_1$, $\texttt{rule}_2^0$, $\texttt{rule}_2$, $\texttt{rule}_3^0$, $\texttt{rule}_3$, $\texttt{rule}_4^0$ and $\texttt{rule}_4$.

- As both `Nil1` and `Nil2` don't have any type in their definitions, that is, there is not any $atype_{1,j}$ and $atype_{2,j}$ in their Haskell definition, respectively. In fact, by using HBSyntax we know that both $\texttt{rule}_1$ and $\texttt{rule}_2$ are facts `someweird(nil1)` and `someweird(nil1)`; therefore we don't need neither $\texttt{rule}_1$ nor $\texttt{rule}_2$ (because we won't do recursion on `nil1` nor `nil2`) and we can consider $N_1^0 = 0$ and $N_2^0 = 0$, which means, we will consider both `Nil1` and `Nil2` as the minimal structures in `SomeWeird` with respect our *size* concept.

  For the third constructor, there exists a $atype_{3,1}$ related to the expression `Some Int`, that is $atype_{3,1} = \texttt{Int}$. However, we won't do structural induction on that. Therefore, we work similarly as both `Nil1` and `Nil2`. That is, we can consider $N_3^0 = 0$, which means, we will consider `Some Int` as another minimal structure in `SomeWeird` concerning our *size* concept. So, at this moment, we have three minimal elements: `Nil1`, `Nil2` and `Some Int`.

For the fourth constructor, there exists three $atype_{4,1}$, $atype_{4,2}$ and $atype_{4,3}$ related to the expression `Weird SomeWeird SomeWeird Bool`, that is $atype_{4,1}$ = `SomeWeird`, $atype_{4,2}$ = `SomeWeird` and $atype_{4,3}$ = `Bool`, so here we have to do structural induction in two places: $atype_{4,1}$ and $atype_{4,2}$.

Therefore, there exists a natural N_4^0, in this case $N_4^0 = 1$; two variables $Z_{4,1}$ and $Z_{4,2}$ such that $\texttt{rule}_4^0$ holds and is defined as follow:

$$\begin{aligned}\texttt{rule}_4^0:\ \ & \mathit{someweird}(N_4^0,\ \ \mathit{weird}(X_{4,1},\ \ X_{4,2},\ \ ,X_{4,3})) :-\ \mathit{base}(Z_{4,1}),\\ & \mathit{base}(Z_{4,2}),\\ & \mathit{someweird}(Z_{4,1},\ \ X_{4,1}),\\ & \mathit{someweird}(Z_{4,2},\ \ X_{4,2}),\\ & \mathit{gen}_{\texttt{Bool}}(X_{4,3}),\end{aligned}$$

This means that $\texttt{rule}_4^0$ will generate our structural induction base case over the constructor `weird` and we will obtains those elements with size $N_4^0 = 1$. Also, there exists N_4, $W_{4,1}$ and $W_{4,2}$ variables such that $\texttt{rule}_4$ holds and is defined as follow:

$$\begin{aligned}\texttt{rule}_4:\ \ & \mathit{someweird}(N_4,\ \ \mathit{weird}(X_{4,1},\ \ X_{4,2},\ \ ,X_{4,3})) :-\ N_4 > N_4^0,\\ & Nr_4\ \texttt{is}\ N_4 - 1,\\ & \mathit{states}_4(Nr_4,\ \ W_{4,1},\ \ W_{4,2}),\\ & \mathit{someweird}(W_{4,1},\ \ X_{4,1}),\\ & \mathit{someweird}(W_{4,2},\ \ X_{4,2}),\\ & \mathit{gen}_{\texttt{Bool}}(X_{4,3}),\end{aligned}$$

So finally, we can translate the Haskell definition shown above to the following Prolog program:

```
%%%%%%%%%%%%%%%%%%%%%%
%% SomeWeird ADT
%%%%%%%%%%%%%%%%%%%%%%

% rule01
% Base case for nil1
someweird(0, nil1).

% rule02
% Base case for nil2
someweird(0, nil2).

% rule03
% Base case for some(_)
someweird(0, some(X31)) :-
    gen_int(X31).

% rule04
% Base case for weird(_,_,_)
someweird(1, weird(X41, X42, X43)) :-
    base(Z41),
    base(Z42),
    someweird(Z41, X41),
    someweird(Z42, X42),
    gen_bool(X43).

% rule4
% Inductive scenario for weird(_,_,_)
someweird(N4, weird(X41, X42, X43)) :-
    N4 > 1,
    Nr4 is N4 - 1,
    states4(Nr4, W41, W42),
    someweird(W41, X41),
    someweird(W42, X42),
```

```
    gen_bool(X43).


%%%%%%%%%%%%%%%%%
%% Auxiliar rules
%%%%%%%%%%%%%%%%%

% base rules
base(0).

% cases rule
cases(_, Y) :- base(Y).
cases(X, X).

% diff rules
diff4(X, X) :- base(X), !, fail.
diff4(W41, W42).

% states rules
states4(Nr4, W41, W42) :-
    cases(Nr4, W41),
    cases(Nr4, W42),
    diff4(W41, W42).
```

Now, if you type `someweird(N, X)` in the Prolog's CLI, you will get a set of valid `SomeWeird` generated values which hold the completeness we have been searching for.

For instance:

- Let `N` = 0, our program returns:

```
nil1
nil2
some(371056659)
```

- Let `N` = 1, our program returns:

```
weird(nil1, nil1, true)
weird(nil1, nil1, false)
weird(nil1, nil2, true)
weird(nil1, nil2, false)
weird(nil1, some(-427613704), true)
weird(nil1, some(-427613704), false)
weird(nil2, nil1, true)
weird(nil2, nil1, false)
weird(nil2, nil2, true)
weird(nil2, nil2, false)
weird(nil2, some(-266152950), true)
weird(nil2, some(-266152950), false)
weird(some(374335935), nil1, true)
weird(some(374335935), nil1, false)
weird(some(374335935), nil2, true)
weird(some(374335935), nil2, false)
weird(some(374335935), some(-27052657), true)
weird(some(374335935), some(-27052657), false)
```

3.3 Polymorfism Types

We are now in the last problem of our journey: the polymorphism types. Usually, algebra data types are defined by using parameter types that represent the polymorphism. We can obtain a greater set of elements belonging to the ADT type. In particular, we can obtain a set of instances of our ADT for every type that you can use in its implementation. So here, we have to deal with a more abstract level. Therefore, the idea here is the following:

Let's suppose our known definition of Binary Search Tree in Haskell:

```
data BSTree = Nil | T Int BSTree BSTree
```

We need to know what should look like it in a polymorphic approach. So, first of all, let's define its polymorphic version:

```
data BSTree a = Nil | T a BSTree BSTree
```

Now, our Haskell expression depends on a type parameter. Let's take a look at our first Binary Search Tree Prolog implementation results of the syntax-translation mechanism HSyntax defined in 3.1.

```
bstree(nil).                                                    %% rule 1
bstree(t(X21, X22, X23)) :- gen_int(X21), bstree(X22), bstree(X23).   %% rule 2
```

We see that by changing `gen_int` for some other more general *gen* function we could obtain a parametrized generator of types. That is, we can define a newer generator as something like that `gen(T, X)`, where `T` is the symbolic variable that parametrises the type.

```
bstree(Type, nil).                        %% rule 1
bstree(Type, t(X21, X22, X23)) :-         %% rule 2
    gen(Type, X21),
    bstree(Type, X22),
    bstree(Type, X23).
```

However, an algebra data type can be defined using as many representation polymorphic types as it needs. So, we have to generalize it also following the Haskell formal grammar. In fact, now we have to care about type variables τ_i in the left-side declaration.

$$
\begin{aligned}
\texttt{data}\ \mathit{tycon}\ \tau_1\ \tau_2\ \ldots\ \tau_k = {} & \mathit{con}_1\ \alpha_{1,1}\ \alpha_{1,2}\ \ldots\ \alpha_{1,k_1}\ \mid \\
& \mathit{con}_2\ \alpha_{2,1}\ \alpha_{2,2}\ \ldots\ \alpha_{2,k_2}\ \mid \\
& \ldots \\
& \mathit{con}_n\ \alpha_{n,1}\ \alpha_{n,2}\ \ldots\ \alpha_{n,k_n}\ \mid
\end{aligned}
$$

Let $\mathcal{H}$, $\mathcal{P}$ be the set of well-defined expressions in Haskell and Prolog, respectively. We can consider $\mathcal{H}_{\text{ADT}}$ and $\mathcal{P}_{\text{Clause}}$, the subset of well-defined algebra data types expressions in Haskell contained in $\mathcal{H}$ and the subset of clauses in $\mathcal{P}$, respectively. Let HSyntax be our syntax-translate mechanism function defined before. We define HBSyntax the refinement of the function HSyntax

$$\text{HGSyntax}: \ \mathcal{H}_{\text{ADT}} \longrightarrow \mathcal{P}_{\text{Clause}}$$

as an extension of our syntax-translation semantic for polymorphic types as is shown in appendix B.2.1.

$$
\begin{aligned}
\texttt{data}\ \mathit{tycon}\ \tau_1\ \tau_2\ \ldots\ \tau_k = {} & \mathit{con}_1\ \alpha_{1,1}\ \alpha_{1,2}\ \ldots\ \alpha_{1,k_1}\ \mid \\
& \mathit{con}_2\ \alpha_{2,1}\ \alpha_{2,2}\ \ldots\ \alpha_{2,k_2}\ \mid \\
& \ldots \\
& \mathit{con}_n\ \alpha_{n,1}\ \alpha_{n,2}\ \ldots\ \alpha_{n,k_n}\ \mid
\end{aligned}
$$

$$\Downarrow \text{HGSyntax}$$

$$
\begin{array}{ll}
\texttt{rule}_1: \ 'tycon(T_1,\ T_2,\ \ldots,\ T_k,\ 'con_1(X_{1,1},\ \ldots,\ X_{1,k_1})) :- & gen(\Delta_{1,1},\ X_{1,1}), \\
& gen(\Delta_{1,2},\ X_{1,2}), \\
& \ldots \\
& gen(\Delta_{1,k_1},\ X_{1,k_1}). \\
& pred_{1,1}(\Omega_{1,1},\ X_{1,1}), \\
& pred_{1,2}(\Omega_{1,2},\ X_{1,2}), \\
& \ldots \\
& pred_{1,k_1}(\Omega_{1,k_1},\ X_{1,k_1}).
\end{array}
$$

$$
\begin{array}{ll}
\texttt{rule}_2: \ 'tycon(T_1,\ T_2,\ \ldots,\ T_k,\ 'con_2(X_{2,1},\ \ldots,\ X_{2,k_2})) :- & gen(\Delta_{2,1},\ X_{2,1}), \\
& gen(\Delta_{2,2},\ X_{2,2}), \\
& \ldots \\
& gen(\Delta_{2,k_1},\ X_{2,k_2}). \\
& pred_{2,1}(\Omega_{2,1},\ X_{2,1}), \\
& pred_{2,2}(\Omega_{2,2},\ X_{2,2}), \\
& \ldots \\
& pred_{2,k_2}(\Omega_{2,k_2},\ X_{2,k_2}).
\end{array}
$$

$$\vdots$$

$$
\begin{array}{ll}
\texttt{rule}_n: \ 'tycon(T_1,\ T_2,\ \ldots,\ T_k,\ 'con_n(X_{n,1},\ \ldots,\ X_{n,k_n})) :- & gen(\Delta_{n,1},\ X_{n,1}), \\
& gen(\Delta_{n,2},\ X_{n,2}), \\
& \ldots \\
& gen(\Delta_{n,k_1},\ X_{n,k_n}). \\
& pred_{n,1}(\Omega_{n,1},\ X_{n,1}), \\
& pred_{n,2}(\Omega_{n,2},\ X_{n,2}), \\
& \ldots \\
& pred_{n,k_n}(\Omega_{n,k_2},\ X_{n,k_n}).
\end{array}
$$

where:

- For all τ_i with $i \in \{1, \ldots k\}$, there exists a variable T_i in Prolog, that represents the symbolic variable for the type that it will generate.

  Example 3.3.1. Let's suppose the polymorphic version of the definition of Either:

  ```
  data Either a b = Left a | Right b
  ```

  Here, you can see that $\tau_1 = \mathtt{a}$, $\tau_2 = \mathtt{b}$, therefore there exists two variables, T_1 and T_2, respectively.

  In our Prolog program, we will define a relationship between some atoms which will represent the type expression in Prolog, concerning its corresponding generation function.

- Let $P(\{T_1, T_2, \ldots,\ T_k\})$ be the *powerset* of the set of all Prolog types variables. For all $\alpha_{i,j}$ with $(i,j) \in \{1, \ldots n\} \times \{1, \ldots k_i\}$, we define $\Omega_{i,j}$ an element of $P(\{T_1, T_2, \ldots,\ T_k\})$ as the set of the those Prolog types variables that participate in the definition of $\alpha_{i,j}$ at our Haskell's algebra data type definition.

  Example 3.3.2. Let's suppose the polymorphic version of the `SomeWeird` definition:

  ```
  data SomeWeird a b = Nil1 |
                       Nil2 |
                       Some a |
                       Weird (SomeWeird a b) (SomeWeird a b) b
  ```

  Here, $\tau_1 = \texttt{a}$ and $\tau_2 = \texttt{b}$, we can define $T_1 = \texttt{A}$ and $T_2 = \texttt{B}$

  - As `Nil1` constructor doesn't require any type to be defined, therefore $\Omega_{1,1} = \emptyset \in P(\{\texttt{A}, \texttt{B}\})$.
  - As `Nil2` constructor doesn't require any type to be defined, therefore $\Omega_{2,1} = \emptyset \in P(\{\texttt{A}, \texttt{B}\})$.
  - As `Some` constructor requires `a`, that is, $\alpha_{3,1} = \texttt{a}$ and it needs `a` to be defined, therefore $\Omega_{3,1} = \{\texttt{A}\} \in P(\{\texttt{A}, \texttt{B}\})$.
  - As `Weird` constructor requires both types `SomeWeird a b` and `b`, that is, $\alpha_{4,1} = \alpha_{4,2} = \texttt{SomeWeird a b}$ and $\alpha_{4,3} = \texttt{b}$, therefore $\Omega_{4,1} = \Omega_{4,2} = \{\texttt{A}, \texttt{B}\} \in P(\{\texttt{A}, \texttt{B}\})$ and $\Omega_{4,3} = \{\texttt{B}\} \in P(\{\texttt{A}, \texttt{B}\})$.

- For all $\alpha_{i,j}$ with $(i,j) \in \{1, \ldots n\} \times \{1, \ldots k_i\}$, there exists a predicate $pred_{i,j}$ that resolve $X_{i,j}$ using $\Omega_{i,j}$.

  When $\alpha_{i,j}$ is an Haskell's primitive type, that means $\Omega_{i,j} = \{T_{\omega_{i,j}}\}$ with $T_{\omega_{i,j}} \in \{T_1, T_2, \ldots T_k\}$, the predicate $pred_{i,j}$ can be defined as the *gen* generation function. That is, $pred_{i,j}(\Omega_{i,j}, X_{i,j}) \equiv gen(T_{\omega_{i,j}}, X_{i,j})$, being the *gen* generation function our new Prolog rule that generates a value of type $\tau_{\omega_{i,j}} \in \{\tau_1, \tau_2, \ldots \tau_k\}$.

3.3.1 Generic Polymorphic Types Generator

For this work, I provide an implementation of the generic *gen* generation function of primitive types which generalize by using a type variable as a new parameter. This implementation is not the best efficient one, but it is enough for our purpose. First of all, we define the atoms which will represent the type expression in our Prolog program. That is:

$$
\begin{array}{rccc}
\text{HGSyntax}_{\texttt{PrimitiveType}}: & \mathcal{H} & \longrightarrow & \mathcal{P} \\
 & \texttt{Char} & \longrightarrow & \texttt{char} \\
 & \texttt{String} & \longrightarrow & \texttt{string} \\
 & \texttt{Int} & \longrightarrow & \texttt{int} \\
 & \texttt{Integer} & \longrightarrow & \texttt{integer} \\
 & \texttt{Double} & \longrightarrow & \texttt{double} \\
 & \texttt{Bool} & \longrightarrow & \texttt{bool} \\
 & \texttt{Unit} & \longrightarrow & \texttt{unit}
\end{array}
$$

Therefore, we can define the following relationship in Prolog:

Let *gen* be the function which generates values of primitive types. We can implement in Prolog this function as the following rule:

```
%% Generator
gen(Type, X) :-
    rel_type(Type, Gen),
    call(Gen, X).
gen(Type, X) :-
    rel_gen(Type, Gen),
    call(Gen, X).

%% Generator relationship
rel_type(char, gen_char).
```

```
rel_type(string, gen_string).
rel_type(int, gen_int).
rel_type(integer, gen_integer).
rel_type(bool, gen_bool).
rel_type(unit, gen_unit).

rel_gen(Gen, Y) :- rel_type(Gen, Y), !, fail.
rel_gen(Gen, Gen).
```

So now, we can obtain a boolean value just by typing `gen(bool, B)`. The last cases `rel_gen` are when you want to generate polymorphically by using another non-primitive generator. For instance:

Let's suppose our Prolog program that generates Binary Search Trees from the polymorphic definition.

```
bstree(Type, nil).                        %% rule 1
bstree(Type, t(X21, X22, X23)) :-         %% rule 2
    gen(Type, X21),
    bstree(Type, X22),
    bstree(Type, X23).
```

We want to generate lists of Binary Search Trees. In order to do that, we could just implement the polymorphic version of a generation of lists and use our `bstree` program:

```
list(_, 0, []).
list(Gen, Length, [X|Xs]) :-
   Length > 0,
   N1 is Length - 1,
   gen(Gen, X),
   list(Gen, N1, Xs).
```

Now, if you type `list(bstree(unit), 2, X)` in the Prolog's CLI, you will get a valid list of `BSTree` generated value:

```
[nil, nil]
[nil, t(unit, nil, nil)]
[nil, t(unit, nil, t(unit, nil, nil))]
[nil, t(unit, nil, t(unit, nil, t(unit, nil, nil)))]
```

Of course, you could have used the boundary version of `bstree` to get more control over the space of solutions.

Finally, we can show many examples using this polymorphic approach. However, before proceeding further, I will provide the refinement of HBSyntax defined in the last section 3.2 with its polymorphic version.

3.3.2 Polymorfism Types with Boundaries

Let $\mathcal{H}$, $\mathcal{P}$ be the set of well-defined expressions in Haskell and Prolog, respectively. We can consider $\mathcal{H}_{\text{ADT}}$ and $\mathcal{P}_{\text{Clause}}$, the subset of well-defined algebra data types expressions in Haskell contained in $\mathcal{H}$ and the subset of clauses in $\mathcal{P}$, respectively. Let HSyntax be our syntax-translate mechanism function defined before. We define HBSyntax the refinement of the function HGSyntax

$$\text{HGBSyntax} : \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$$

as an extension of our syntax-translation semantic for polymorphic types with boundaries as mix of B.2.1 and B.3.1.

3.3.3 Examples

For the following examples, we will use the polymorphic and bounded approach HGBSyntax. Of course, we avoid showing again the auxiliary corresponding function that accompanies the respective Prolog program in order to be briefer.

Example 3.3.3 (MyList)**.** Let's consider the polymorphic version of our `MyListBool` Haskell ADT definition:

```
data MyList a = Nil | Cons a (MyList a)
```

Using HBSyntax, in the las examples 3.2.1 we obtain:

```
mylistbool(0,nil).

mylistbool(1, cons(X21, X22)) :-
    base(Z22),
    gen_bool(X21),
    mylistbool(Z22, X22).


mylistbool(N2, cons(X21, X22)) :-
    N2 > 1,
    Nr2 is N2 - 1,
    states2(Nr2, W22),
    gen_bool(X21),
    mylistbool(W22, X22).
```

Now, if we use the newer refinement function HGBSyntax:

- We can see that there exists a τ_1. Therefore there exists a variable, T_1.
- Here, $\tau_1 = \texttt{a}$ so we can define $T_1 = \texttt{A}$ and consider the poset $P(\{\texttt{A}\})$.
- As `Nil` constructor doesn't require any type to be defined, therefore $\Omega_{1,1} = \emptyset \in P(\{\texttt{A}\})$.

  For the second constructor, there exists two $\alpha_{2,1}$ and $\alpha_{2,2}$ related to the expression `Cons a (MyList a)`, that is $\alpha_{2,1} = \texttt{a}$ and $\alpha_{2,2} = \texttt{MyList a}$, which both is defined polymorphically, therefore $\Omega_{2,1} = \{\texttt{A}\} \in P(\{\texttt{A}\})$ and $\Omega_{2,2} = \{\texttt{A}\} \in P(\{\texttt{A}\})$.

So finally, we can translate the Haskell definition shown above to the following Prolog program:

```
mylist(0, A, nil).

mylist(1, A, cons(X21, X22)) :-
    base(Z22),
    gen(A, X21),
    mylist(Z22, A, X22).


mylist(N2, A, cons(X21, X22)) :-
    N2 > 1,
    Nr2 is N2 - 1,
    states2(Nr2, W22),
    gen(A, X21),
    mylist(W22, A, X22).
```

Now, if you type `mylist(N, bool, X)` in the Prolog's CLI, you will get a set of valid `MyList a` with `a` := `Bool` generated values which is the same set than the generated one by using our old `MyListBool` program and holds the completeness we have been searching for.

Example 3.3.4 (Binary Search Tree)**.** Let's consider the polymorphic version of our `BSTree` Haskell ADT definition:

```
data BSTree a = Nil | T a BSTree BSTree
```

Using HBSyntax, in the las examples 3.2.1 we obtain:

```
bstree(0,nil).

bstree(1, t(X21, X22, X23)) :-
    base(Z22),
    base(Z23),
    gen_int(X21),
    bstree(Z22, X22),
    bstree(Z23, X23).

bstree(N2, t(X21, X22, X23)) :-
    N2 > 1,
    Nr2 is N2 - 1,
    states2(Nr2, W22, W23),
    gen_int(X21),
    bstree(W22, X22),
    bstree(W23, X23).
```

Now, if we use the newer refinement function HGBSyntax:

- We can see that there exists a τ_1. Therefore there exists a variable, T_1.
- Here, $\tau_1 =$ `a` so we can define $T_1 =$ `A` and consider the poset $P(\{\texttt{A}\})$.
- As `Nil` constructor doesn't require any type to be defined, therefore $\Omega_{1,1} = \emptyset \in P(\{\texttt{A}\})$.

  For the second constructor, there exists three $\alpha_{2,1}$, $\alpha_{2,2}$ and $\alpha_{2,3}$ related to the expression `T a (BSTree a) (BSTree a)`, that is $\alpha_{2,1} =$ `a` and $\alpha_{2,2} = \alpha_{2,3} =$ `BSTree a`, which both is defined polymorphically, therefore $\Omega_{2,1} = \{\texttt{A}\} \in P(\{\texttt{A}\})$ and $\Omega_{2,2} = \Omega_{2,3} = \{\texttt{A}\} \in P(\{\texttt{A}\})$.

So finally, we can translate the Haskell definition shown above to the following Prolog program:

```
bstree(0, A, nil).

bstree(1, A, t(X21, X22, X23)) :-
    base(Z22),
    base(Z23),
    gen(A, X21),
    bstree(Z22, A, X22),
    bstree(Z23, A, X23).

bstree(N2, A, t(X21, X22, X23)) :-
    N2 > 1,
    Nr2 is N2 - 1,
    states2(Nr2, W22, W23),
    gen(A, X21),
    bstree(W22, A, X22),
    bstree(W23, A, X23).
```

Now, if you type `bstree(N, int, X)` in the Prolog's CLI, you will get a set of valid `BSTree a` with `a` := `Int` generated values which is the same set than the generated one by using our old monomorphic type version of `BSTree` program and holds the completeness we have been searching for.

Example 3.3.5 (SomeWeird)**.** Let's consider the polymorphic version of our `SomeWeird` Haskell ADT definition:

```
data SomeWeird a b = Nil1 |
                     Nil2 |
                     Some a |
                     Weird (SomeWeird a b) (SomeWeird a b) b
```

Using HBSyntax, in the las examples 3.2.1 we obtain:

```
someweird(0, nil1).

someweird(0, nil2).

someweird(0, some(X31)) :-
    gen_int(X31).

someweird(1, weird(X41, X42, X43)) :-
    base(Z41),
    base(Z42),
    someweird(Z41, X41),
    someweird(Z42, X42),
    gen_bool(X43).

someweird(N4, weird(X41, X42, X43)) :-
    N4 > 1,
    Nr4 is N4 - 1,
    states4(Nr4, W41, W42),
    someweird(W41, X41),
    someweird(W42, X42),
    gen_bool(X43).
```

Now, if we use the newer refinement function HGBSyntax:

- We can see that there exists two τ_1 and τ_2. Therefore there exist two variables, T_1 and T_2.
- Here, $\tau_1 = \texttt{a}$ and $\tau_2 = \texttt{b}$ so we can define $T_1 = \texttt{A}$, $T_2 = \texttt{B}$ and consider the poset $P(\{\texttt{A}, \texttt{B}\})$.
- As `Nil1` constructor doesn't require any type to be defined, therefore $\Omega_{1,1} = \emptyset \in P(\{\texttt{A}\})$.

  As `Nil2` constructor doesn't require any type to be defined, therefore $\Omega_{2,1} = \emptyset \in P(\{\texttt{A}\})$.

  For the third constructor, there exists an $\alpha_{3,1}$ related to the expression `Some a`, that is $\alpha_{3,1} = \texttt{a}$, which is defined polymorphically, therefore $\Omega_{3,1} = \{\texttt{A}\} \in P(\{\texttt{A}\})$.

  For the fourth constructor, there exists three $\alpha_{4,1}$, $\alpha_{4,2}$ and $\alpha_{4,3}$ related to the expression `Weird (SomeWeird a) (SomeWeird a) b`, that is $\alpha_{4,1} = \alpha_{4,2} = \texttt{SomeWeird a b}$ and $\alpha_{4,3} = \texttt{b}$, which both is defined polymorphically, therefore $\Omega_{4,1} = \Omega_{4,2} = \{\texttt{A}, \texttt{B}\} \in P(\{\texttt{A}, \texttt{B}\})$ and $\Omega_{4,3} = \{\texttt{B}\} \in P(\{\texttt{A}, \texttt{B}\})$.

So finally, we can translate the Haskell definition shown above to the following Prolog program:

```
someweird(0, A, B, nil1).

someweird(0, A, B, nil2).

someweird(0, A, B, some(X31)) :-
    gen(A, X31).

someweird(1, A, B, weird(X41, X42, X43)) :-
    base(Z41),
    base(Z42),
    someweird(Z41, A, B, X41),
    someweird(Z42, A, B, X42),
    gen(B, X43).

someweird(N4, A, B, weird(X41, X42, X43)) :-
    N4 > 1,
    Nr4 is N4 - 1,
    states4(Nr4, W41, W42),
    someweird(W41, A, B, X41),
    someweird(W42, A, B, X42),
    gen(B, X43).
```

Now, if you type `someweird(N, int, bool, X)` in the Prolog's CLI, you will get a set of valid `SomeWeird a b` with `a` := `Int` and `b` := `Bool` generated values which is the same set than the generated one by using our old monomorphic type version of `SomeWeird` program and holds the completeness we have been searching for.

3.3.4 Red-Black Tree

Finally, we have the tools to get the Prolog program approach from a Red-Black Tree Haskell's definition. Let's pull out all the stops.

Let's suppose the following Haskell ADT definition:

```
data Color = Red | Black

data RBTree a = Nil | T Color a (RBTree a) (RBTree a)
```

Using HGBSyntax, in the las examples 3.2.1 we obtain:

```
%%%%%%%%%%%%%%%%%%%%%%
%% Color
%%%%%%%%%%%%%%%%%%%%%%

color(red).

color(black).

%%%%%%%%%%%%%%%%%%%%%%
%% Red-Black Tree
%%%%%%%%%%%%%%%%%%%%%%

rbtree(0, A, nil).

rbtree(1, A, t(X21, X22, X23, X24)) :-
    base(Z23),
    base(Z24),
    gen(color, X21),
    gen(A, X22),
    rbtree(Z23, A, X23),
    rbtree(Z24, A, X24).

rbtree(N2, A, t(X21, X22, X23, X24)) :-
    N2 > 1,
    Nr2 is N2 - 1,
    states2(Nr2, W23, W24),
    gen(color, X21),
    gen(A, X22),
    rbtree(W23, A, X23),
    rbtree(W24, A, X24).

%%%%%%%%%%%%%%%%%
%% Auxiliar rules
%%%%%%%%%%%%%%%%%

% base rules
base(0).

% cases rule
cases(_, Y) :- base(Y).
cases(X, X).

% diff rules
diff2(X, X) :- base(X), !, fail.
diff2(W23, W24).

% states rules
states2(Nr2, W23, W24) :-
    cases(Nr2, W23),
    cases(Nr2, W24),
    diff2(W23, W24).
```

And that is! right? right...

We have not finished yet in fact. Of course, now, we can generate random values from a given Haskell's ADT definition by using our syntax-translation mechanism. However, we are forgetting those structures that are valid if and only if hold so many invariants or restrictions.

Chapter 4

Conclusions and Future Works

The work done in this thesis has been quite complex in terms of agreeing on the subset of Haskell expressions for the ADTs and in developing the translation functions. In defining the translations of the subexpressions we have had to limit ourselves to those ADTs with not very complex definitions (avoiding monad declarations, for example).

On the other hand, the initial idea was to go as far as generating complex conditions from Haskell and transfer it to Prolog for the resolution and generations of the values through the translation guided by the syntaxys. However, we had to limit ourselves only to the generation of the ADT structures following the formal Haskell grammar. We leave this explained as future work below.

The result of this master's thesis is subject to further improvements. The first of all is to implement the Prolog-Haskell syntax-translate function. This will allow us to recover all values that are generated in Prolog after applying the syntax-translate mechanism defined in this work, and integrate it in some way with other PBT frameworks like QuickCheck. We also need to automatize it in a complete Haskell library in order to complement QuickCheck without switching either IDE or language. And finally, it could be improved by using some of the generation strategies defined in the recent research which are mentioned at the beginning of this work and/or improving the implementation of the primitive types generators.

Apart from the previous improvements, there is still room for new research. In this work, we did not explore the generation of those algebraic data types that are defined by invariants. We have limited just for generating ADT's structures but some ones are well-defined if and only if they hold complex constraints like red-black trees or sorted-list. For this purpose, I suggest following the Liquid-Haskell syntax as an extension of the Haskell's formal grammar. So, define pre-conditions and post-conditions and translate those sentences in some way to Prolog expressions following the concept worked in this thesis.

Finally, this field helps to create new interesting lines of research for many strong and static types languages like Scala, Rust or Solidity and explore new ways to do property-based testing not just in academic scenarios but also in industrial areas.

Bibliography

[1] G. Fink and M. Bishop, "Property-based testing: A new approach to testing for assurance", SIGSOFT Softw. Eng. Notes, vol. 22, no. 4, pp. 74–80, Jul. 1997, ISSN: 0163-5948. DOI: 10.1145/263244.263267. [Online]. Available: https://doi.org/10.1145/263244.263267.

[2] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality", in Proceedings of the 37th International Conference on Software Engineering - Volume 1, ser. ICSE '15, Florence, Italy: IEEE Press, 2015, pp. 483–493, ISBN: 9781479919345.

[3] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction", Computer, vol. 38, no. 9, pp. 43–50, Sep. 2005, ISSN: 1558-0814. DOI: 10.1109/MC.2005.314.

[4] S. Makinen and J. Münch, "Effects of test-driven development: A comparative analysis of empirical studies", vol. 166, Jan. 2014, ISBN: 978-3-319-03601-4. DOI: 10.1007/978-3-319-03602-1_10.

[5] I. Schaefer, T. Runge, L. Cleophas, and B. W. Watson, "Tutorial: The correctness-by-construction approach to programming using corc", in 2021 IEEE Secure Development Conference (SecDev), Oct. 2021, pp. 1–2. DOI: 10.1109/SecDev51306.2021.00012.

[6] E. De Angelis, F. Fioravanti, A. Palacios, A. Pettorossi, and M. Proietti, "Property-based test case generators for free", in Tests and Proofs, D. Beyer and C. Keller, Eds., Cham: Springer International Publishing, 2019, pp. 186–206, ISBN: 978-3-030-31157-5.

[7] B. Eckel, "Strong typing vs. strong testing", Jan. 2005. DOI: 10.1007/978-1-4302-0038-3_11.

[8] C. Okasaki, "Red-black trees in a functional setting", Journal of Functional Programming, vol. 9, no. 4, pp. 471–477, 1999. DOI: 10.1017/S0956796899003494.

[9] V. Senni and F. Fioravanti, "Generation of test data structures using constraint logic programming", vol. 7305, May 2012, pp. 115–131, ISBN: 978-3-642-30472-9. DOI: 10.1007/978-3-642-30473-6_10.

[10] F. Fioravanti, M. Proietti, and V. Senni, "Efficient generation of test data structures using constraint logic programming and program transformation", Journal of Logic and Computation, Nov. 2013. DOI: 10.1093/logcom/ext071.

[11] R. Peña, J. Sánchez-Hernández, M. Garrido, and J. Sagredo, "Smt-based test-case generation and validation for programs with complex specifications", in. May 2023, pp. 188–205, ISBN: 978-3-031-31475-9. DOI: 10.1007/978-3-031-31476-6_10.

[12] E. Meijer and P. Drayton, "Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages", Jan. 2004.

[13] B. C. Pierce, Types and Programming Languages, 1st. The MIT Press, 2002, ISBN: 0262162091.

[14] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs", in Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ser. ICFP '00, New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279, ISBN: 1581132026. DOI: 10.1145/351240.351266. [Online]. Available: https://doi.org/10.1145/351240.351266.

[15] Quickcheck. github reposittory, https://github.com/nick8325/quickcheck.

[16] Scalacheck. github reposittory, https://github.com/typelevel/scalacheck.

[17] Proper. github reposittory, https://github.com/proper-testing/proper.

[18] Hypothesis. github reposittory, https://github.com/HypothesisWorks/hypothesis.

[19] Haskell 98. syntax, https://www.haskell.org/onlinereport/syntax-iso.html.

[20] Haskell 98. user-defined datatypes, https://www.haskell.org/onlinereport/decls.html.

Appendix

Appendix A

Prolog Code

In this chapter, I will show the Prolog code which is used as a prelude for the translation-syntax mechanism for generating the space of solutions of a given translated algebra data type. It is introduced as modules which you can import into your Prolog program.

Primitive Types Generators

```
:- module(types,
            [
            gen_char/1,
            gen_string/1,
            gen_int/1,
            gen_integer/1,
            gen_bool/1,
            gen_unit/1
            ]).

%% Char type

char(C) :- char_code(C, _).
gen_char(C) :-
    repeat,
    random_between(0, 7935, Code),
    (char_code(C, Code), !; fail).

%% String type
%% string(S) in the Prolog's prelude

gen_string(S) :-
   repeat,
   random_between(0, 99, N),
   (length(L, N), maplist(gen_char, L), atom_string(L,S), string(S), !; fail).

%% Int type

int(I) :- (I >= -536870912, I =< 536870912), integer(I).
gen_int(I) :- random_between(-536870912, 536870912, I), integer(I).

%% Integer type
%% integer(I) in the Prolog's prelude

gen_integer(I) :- random_between(-9999999999999999999999999999,
    9999999999999999999999999999, I).

%% Boolean type

bool(true).
bool(false).
gen_bool(B) :- bool(B).

%% Unit type
```

```
unit(unit).
gen_unit(U) :- unit(U).

%% Double type

double(D) :- float(D).
gen_double(D) :- gen_double(-9999999, 9999999, D).
gen_double(Min, Max, D) :-
    random(F),
    D is Min + F * (Max - Min).
```

Polymorphic Type Generators

```
:- module(generator, [gen/2]).

% Import here the types module
% By using :- consult(path/subpath/types.pl).


%% Generator
gen(Type, X) :-
    rel_type(Type, G),
    call(G, X).

gen(Type, X) :-
    rel_gen(Type, Gen),
    call(Gen, X).

%% Generator relationship
rel_type(char, gen_char).
rel_type(string, gen_string).
rel_type(int, gen_int).
rel_type(integer, gen_integer).
rel_type(bool, gen_bool).
rel_type(unit, gen_unit).

rel_gen(Gen, Y) :- rel_type(Gen, Y), !, fail.
rel_gen(Gen, Gen).
```

Polymorphic and Bounded List Type Generator

```
:- module(list, [list/3]).

% Import here both types and generator modules
% By using :- consult(path/subpath/file.pl).

%% Bounded Polymorphic Type List
%% Example:
%%  list(char, 4, X).
%%  list(bool, 4, X).
%%  list(int, 4, X).

list(_, 0, []).

list(Gen, Length, [X|Xs]) :-
   Length > 0,
   N1 is Length - 1,
   gen(Gen, X),
   list(Gen, N1, Xs).
```

Appendix B

Translations

B.1 Monomorphic Type

B.1.1 Syntax-Translation Function

We define our syntax-tranlsation semantic

$$\text{HSyntax}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$$

for monomorphic types as follow:

$$\begin{aligned}
&\text{HSyntax}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
&\text{HSyntax}\,[\![\texttt{data}\ \mathit{simpletype}\ =\ \mathit{constrs}]\!] \ = \ \text{Rules}\,[\![\mathit{simpletype}\ =\ \mathit{constrs}]\!] \\
\\
&\text{Rules}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
&\text{Rules}\,[\![\mathit{simpletype}\ =\ \mathit{constrs}]\!] \ = \ \text{Rules}\,[\![\mathit{simpletype}\ =\ \mathit{constr}_1\ \mid\ \mathit{constr}_2\ \mid\ \ldots\ \mid\ \mathit{constr}_n]\!] \\
&\text{Rules}\,[\![\mathit{simpletype}\ =\ \mathit{constr}_1\ \mid\ \mathit{constr}_2\ \mid\ \ldots\ \mid\ \mathit{constr}_n]\!] \ = \ \bigvee_{i\in\{1,\ldots,n\}} \text{Rule}\,[\![\mathit{simpletype}\ =\ \mathit{constr}_i]\!] \\
&\text{Rule}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
&\text{Rule}\,[\![\mathit{simpletype}\ =\ \mathit{constr}_i]\!] \ = \ \text{GSimpleType}\,[\![\mathit{simpletype}]\!]\,(\ \text{Con}\,[\![\mathit{constr}_i]\!]\ )\ \ :-\ \ \text{Pred}\,[\![\mathit{constr}_i]\!]\ . \\
\\
&\text{GSimpleType}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
&\text{GSimpleType}\,[\![\mathit{simpletype}]\!] \ = \ \text{GSyntax}\,[\![\mathit{tycon}]\!] \\
\\
&\text{GType}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
&\text{GType}\,[\![\mathit{atype}_{i,j}]\!] \ = \ \text{GSyntax}\,[\![\mathit{gtycon}]\!]\,(\ \text{Var}\,[\![\mathit{atype}_{i,j}]\!]\ ) && \text{If } \mathit{atype}_{i,j} \leadsto \mathit{gtycon} \\
&\text{GType}\,[\![\mathit{atype}_{i,j}]\!] \ = \ \text{GTuple}\,[\![\mathit{atype}_{i,j}]\!] && \text{If } \mathit{atype}_{i,j} \leadsto (\mathit{type}_1,\ \ldots\ \mathit{type}_k) \\
&\text{GType}\,[\![\mathit{atype}_{i,j}]\!] \ = \ \text{GList}\,[\![\mathit{atype}_{i,j}]\!] && \text{If } \mathit{atype}_{i,j} \leadsto [\mathit{type}]
\end{aligned}$$

$$
\begin{array}{lll}
\text{GSyntax}: \mathcal{H}_{\text{ADT}} \longrightarrow \mathcal{P}_{\text{Clause}} & & \\
\text{GSyntax}\,[\![tycon]\!] = \text{Lower}\,[\![tycon]\!] & & \text{If } tycon \text{ is not a primitive type} \\
\text{GSyntax}\,[\![tycon]\!] = gen\_\text{Lower}\,[\![tycon]\!] & & \text{If } tycon \text{ is a primitive type} \\
\text{GSyntax}\,[\![type]\!] = \text{Lower}\,[\![type]\!] & & \text{If } type \text{ is not a primitive type} \\
\text{GSyntax}\,[\![type]\!] = gen\_\text{Lower}\,[\![type]\!] & & \text{If } type \text{ is a primitive type} \\
\text{GSyntax}\,[\![gtycon]\!] = \text{GSyntax}\,[\![qtycon]\!] & & \text{If } gtycon \rightsquigarrow qtycon \\
\text{GSyntax}\,[\![gtycon]\!] = \text{GSyntax}\,[\![()]\!] & & \text{If } gtycon \rightsquigarrow () \\
\text{GSyntax}\,[\![qtycon]\!] = \text{GSyntax}\,[\![tycon]\!] & & \\
\text{GSyntax}\,[\![()]\!] = gen\_unit & &
\end{array}
$$

$$
\begin{array}{l}
\text{GTuple}: \mathcal{H}_{\text{ADT}} \longrightarrow \mathcal{P}_{\text{Clause}} \\
\text{GTuple}\,[\![atype_{i,j}]\!] = \text{GTuple}\,[\![(type_1,\ \dots\ type_k)]\!] \\
\text{GTuple}\,[\![(type_1,\ \dots\ type_k)]\!] = tuple\,(\text{GSyntax}\,[\![type_1]\!],\ \dots,\ \text{GSyntax}\,[\![type_k]\!],\ \text{Var}\,[\![atype_{i,j}]\!])
\end{array}
$$

$$
\begin{array}{l}
\text{GList}: \mathcal{H}_{\text{ADT}} \longrightarrow \mathcal{P}_{\text{Clause}} \\
\text{GList}\,[\![atype_{i,j}]\!] = \text{GList}\,[\![[type]]\!] \\
\text{GList}\,[\![[type]]\!] = list(\text{GSyntax}\,[\![type]\!],\ \text{Var}\,[\![atype_{i,j}]\!])
\end{array}
$$

$$
\begin{array}{l}
\text{Con}: \mathcal{H}_{\text{ADT}} \longrightarrow \mathcal{P}_{\text{Clause}} \\
\text{Con}\,[\![constr_i]\!] = \text{Con}\,[\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \dots\ atype_{i,m}]\!] \\
\text{Con}\,[\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \dots\ atype_{i,m}]\!] = \text{Lower}\,[\![con_i]\!]\ (\ \text{Var}\,[\![atype_{i,1}]\!], \text{Var}\,[\![atype_{i,2}]\!],\ \dots,\ \text{Var}\,[\![atype_{i,m}]\!]\ )
\end{array}
$$

$$
\begin{array}{l}
\text{Pred}: \mathcal{H}_{\text{ADT}} \longrightarrow \mathcal{P}_{\text{Clause}} \\
\text{Pred}\,[\![constr_i]\!] = \text{Pred}\,[\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \dots\ atype_{i,m}]\!] \\
\text{Pred}\,[\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \dots\ atype_{i,m}]\!] = \bigwedge_{j \in \{1,\dots,m\}} \text{GType}\,[\![atype_{i,j}]\!]
\end{array}
$$

$$
\begin{array}{lll}
\text{Var}: \mathcal{H}_{\text{ADT}} \longrightarrow \mathcal{P}_{\text{Clause}} & & \\
\text{Var}\,[\![atype_{i,j}]\!] = X_{i,j} & & \text{Where } X_{i,j} \text{ is a variable} \\
\text{Var}\,[\![exp]\!] = X & & \text{Where } X \text{ is a variable}
\end{array}
$$

$$
\begin{array}{lll}
\text{Lower}: \mathcal{H}_{\text{ADT}} \longrightarrow \mathcal{P}_{\text{Clause}} & & \\
\text{Lower}\,[\![exp]\!] = {'exp} & & \text{Where } {'exp} \text{ is the expression } exp \text{ in lower case}
\end{array}
$$

The expression starting with $'$ in the evaluation of Lower, represents the same expression in Haskell with all their characters in lowercase. Let's see some examples:

Example B.1.1. Let's suppose a monotype Binary Search Tree definition in Haskell with `Int` type:

```
data BSTree = Nil | T Int BSTree BSTree
```

Here, you can see that $tycon \equiv$ `BTree`, $con_1 \equiv$ `Nil` and $con_2 \equiv$ `T`, therefore

$$
\begin{aligned}
\text{Lower}\,[\![tycon]\!] &= \text{Lower}\,[\![BTree]\!] = \texttt{btree} \\
\text{Lower}\,[\![con_1]\!] &= \text{Lower}\,[\![Nil]\!] = \texttt{nil} \\
\text{Lower}\,[\![con_2]\!] &= \text{Lower}\,[\![T]\!] = \texttt{t}
\end{aligned}
$$

respectively.

Example B.1.2. Let's suppose a non-polymorphic version of the definition of Either in Haskell for both `String` and `Int` types:

```
data Either = Left String | Right Int
```

Here, you can see that $tycon \equiv$ `Either`, $con_1 \equiv$ `Left` and $con_2 \equiv$ `Right`, therefore

$$\begin{aligned} \text{Lower } [\![tycon]\!] &= \text{Lower } [\![Either]\!] = \texttt{either} \\ \text{Lower } [\![con_1]\!] &= \text{Lower } [\![Left]\!] = \texttt{left} \\ \text{Lower } [\![con_2]\!] &= \text{Lower } [\![Right]\!] = \texttt{right} \end{aligned}$$

respectively.

B.1.2 MaybeInt

$\text{HSyntax } [\![\texttt{data } MaybeInt = None \mid Some\ Int]\!] = \text{ Rules } [\![MaybeInt = None \mid Some\ Int]\!]$

$\text{Rules } [\![MaybeInt = None \mid Some\ Int]\!] = \text{ Rule } [\![MaybeInt = None]\!] \ \vee \ \text{Rule } [\![MaybeInt = Some\ Int]\!]$

$\text{Rule } [\![MaybeInt = None]\!] = \text{ GSimpleType } [\![MaybeInt]\!](\text{Con } [\![None]\!])\ .$
$\text{Rule } [\![MaybeInt = Some\ Int]\!] = \text{ GSimpleType } [\![MaybeInt]\!](\text{Con } [\![Some\ Int]\!]) \ :- \ \text{Pred } [\![Some\ Int]\!]\ .$

$\text{GSimpleType } [\![MaybeInt]\!] = \text{ GSyntax } [\![MaybeInt]\!]$

$\text{GType } [\![Int]\!] = \text{ GSyntax } [\![Int]\!](\text{ Var } [\![Int]\!]\ )$

$\text{GSyntax } [\![MaybeInt]\!] = \text{ Lower } [\![MaybeInt]\!]$
$\text{GSyntax } [\![Int]\!] = \ gen\_\text{Lower } [\![Int]\!]$

$\text{Con } [\![None]\!] = \text{ Lower } [\![None]\!]$
$\text{Con } [\![Some\ Int]\!] = \text{ Lower } [\![Some]\!]\ (\text{ Var } [\![Int]\!]\ )$

$\text{Pred } [\![Some\ Int]\!] = \text{ GType } [\![Int]\!]$

$\text{Lower } [\![MaybeInt]\!] = \texttt{maybeint}$
$\text{Lower } [\![None]\!] = \texttt{none}$
$\text{Lower } [\![Some]\!] = \texttt{some}$
$\text{Lower } [\![Int]\!] = \texttt{int}$

$\text{Var } [\![Int]\!] = \texttt{X21}$ For $atype_{2,1} \equiv Int$

B.1.3 Either

HSyntax $[\![$`data` $Either = Left\ String \mid Right\ Int]\!] =$ Rules $[\![Either = Left\ String \mid Right\ Int]\!]$

Rules $[\![Either = Left\ String \mid Right\ Int]\!] =$ Rule $[\![Either = Left\ String]\!]\ \vee$ Rule $[\![Either = Right\ Int]\!]$

Rule $[\![Either = Left\ String]\!] =$ GSimpleType $[\![Either]\!]$(Con $[\![Left\ String]\!]$) $:-$ Pred $[\![Left\ String]\!]$.
Rule $[\![Either = Right\ Int]\!] =$ GSimpleType $[\![Either]\!]$(Con $[\![Right\ Int]\!]$) $:-$ Pred $[\![Right\ Int]\!]$.

GSimpleType $[\![Either]\!] =$ GSyntax $[\![Either]\!]$

GType $[\![String]\!] =$ GSyntax $[\![String]\!]$(Var $[\![String]\!]$)
GType $[\![Int]\!] =$ GSyntax $[\![Int]\!]$(Var $[\![Int]\!]$)

GSyntax $[\![Either]\!] =$ Lower $[\![Either]\!]$
GSyntax $[\![String]\!] =$ *gen*_Lower $[\![String]\!]$
GSyntax $[\![Int]\!] =$ *gen*_Lower $[\![Int]\!]$

Con $[\![Left\ String]\!] =$ Lower $[\![Left]\!]$ (Var $[\![String]\!]$)
Con $[\![Right\ Int]\!] =$ Lower $[\![Right]\!]$ (Var $[\![Int]\!]$)

Pred $[\![Left\ String]\!] =$ GType $[\![String]\!]$
Pred $[\![Right\ Int]\!] =$ GType $[\![Int]\!]$

Lower $[\![Either]\!] =$ `either`
Lower $[\![Left]\!] =$ `left`
Lower $[\![Right]\!] =$ `right`
Lower $[\![String]\!] =$ `string`
Lower $[\![Int]\!] =$ `int`

Var $[\![String]\!] =$ `X11` For $atype_{1,1} \equiv String$
Var $[\![Int]\!] =$ `X21` For $atype_{2,1} \equiv Int$

B.1.4 MyListBool

$$\text{HSyntax } [\![\texttt{data } \mathit{MyListBool} = \mathit{Nil} \mid \mathit{Cons\ Bool\ MyListBool}]\!] = \text{ Rules } [\![\mathit{MyListBool} = \mathit{Nil} \mid \mathit{Cons\ Bool\ MyListBool}]\!]$$

$$\begin{aligned}\text{Rules } [\![\mathit{MyListBool} = \mathit{Nil} \mid \mathit{Cons\ Bool\ MyListBool}]\!] = \ & \text{Rule } [\![\mathit{MyListBool} = \mathit{Nil}]\!] \ \vee \\ & \text{Rule } [\![\mathit{MyListBool} = \mathit{Cons\ Bool\ MyListBool}]\!]\end{aligned}$$

$$\begin{aligned}&\text{Rule } [\![\mathit{MyListBool} = \mathit{Nil}]\!] = \text{ GSimpleType } [\![\mathit{MyListBool}]\!](\text{Con } [\![\mathit{Nil}]\!]) \ .\\ &\text{Rule } [\![\mathit{MyListBool} = \mathit{Cons\ Bool\ MyListBool}]\!] = \text{ GSimpleType } [\![\mathit{MyListBool}]\!](\text{Con } [\![\mathit{Cons\ Bool\ MyListBool}]\!]) \ :- \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Pred } [\![\mathit{Cons\ Bool\ MyListBool}]\!] \ .\end{aligned}$$

$$\text{GSimpleType } [\![\mathit{MyListBool}]\!] = \text{ GSyntax } [\![\mathit{MyListBool}]\!]$$

$$\begin{aligned}&\text{GType } [\![\mathit{MyListBool}]\!] = \text{ GSyntax } [\![\mathit{MyListBool}]\!](\text{ Var } [\![\mathit{MyListBool}]\!] \ )\\ &\text{GType } [\![\mathit{Bool}]\!] = \text{ GSyntax } [\![\mathit{Bool}]\!](\text{ Var } [\![\mathit{Bool}]\!] \ )\end{aligned}$$

$$\begin{aligned}&\text{GSyntax } [\![\mathit{MyListBool}]\!] = \text{ Lower } [\![\mathit{MyListBool}]\!]\\ &\text{GSyntax } [\![\mathit{Bool}]\!] = \ \mathit{gen}\_\text{Lower } [\![\mathit{Bool}]\!]\end{aligned}$$

$$\begin{aligned}&\text{Con } [\![\mathit{Nil}]\!] = \text{ Lower } [\![\mathit{Nil}]\!]\\ &\text{Con } [\![\mathit{Cons\ Bool\ MyListBool}]\!] = \text{ Lower } [\![\mathit{Cons}]\!] \ (\text{ Var } [\![\mathit{Bool}]\!] \ , \text{ Var } [\![\mathit{MyListBool}]\!] \ )\end{aligned}$$

$$\text{Pred } [\![\mathit{Cons\ Bool\ MyListBool}]\!] = \text{ GType } [\![\mathit{Bool}]\!] \ \wedge \ \text{GType } [\![\mathit{MyListBool}]\!]$$

$$\begin{aligned}&\text{Lower } [\![\mathit{MyListBool}]\!] = \texttt{mylistbool}\\ &\text{Lower } [\![\mathit{Nil}]\!] = \texttt{nil}\\ &\text{Lower } [\![\mathit{Cons}]\!] = \texttt{cons}\\ &\text{Lower } [\![\mathit{Bool}]\!] = \texttt{bool}\end{aligned}$$

$$\begin{aligned}&\text{Var } [\![\mathit{Bool}]\!] = \texttt{X21} && \text{For } \mathit{atype}_{2,1} \equiv \mathit{Bool}\\ &\text{Var } [\![\mathit{MyListBool}]\!] = \texttt{X22} && \text{For } \mathit{atype}_{2,2} \equiv \mathit{MyListBool}\end{aligned}$$

B.1.5 BSTree

$$\text{HSyntax } [\![\texttt{data}\ BSTree = Nil \mid Cons\ Int\ BSTree\ BSTree]\!] = \text{ Rules } [\![BSTree = Nil \mid Cons\ Int\ BSTree\ BSTree]\!]$$

$$\begin{aligned}\text{Rules } [\![BSTree = Nil \mid Cons\ Int\ BSTree\ BSTree]\!] = {} & \text{ Rule } [\![BSTree = Nil]\!]\ \vee \\ & \text{ Rule } [\![BSTree = Cons\ Int\ BSTree\ BSTree]\!]\end{aligned}$$

$$\text{Rule } [\![BSTree = Nil]\!] = \text{ GSimpleType } [\![BSTree]\!](\text{Con } [\![Nil]\!])\ .$$
$$\begin{aligned}\text{Rule } [\![BSTree = Cons\ Int\ BSTree\ BSTree]\!] = {} & \text{ GSimpleType } [\![BSTree]\!](\text{Con } [\![Cons\ Int\ BSTree\ BSTree]\!])\ \ :- \\ & \text{Pred } [\![Cons\ Int\ BSTree\ BSTree]\!]\ .\end{aligned}$$

$$\text{GSimpleType } [\![BSTree]\!] = \text{ GSyntax } [\![BSTree]\!]$$

$$\text{GType } [\![BSTree]\!] = \text{ GSyntax } [\![BSTree]\!](\text{ Var } [\![BSTree]\!]\ )$$
$$\text{GType } [\![Int]\!] = \text{ GSyntax } [\![Int]\!](\text{ Var } [\![Int]\!]\ )$$

$$\text{GSyntax } [\![BSTree]\!] = \text{ Lower } [\![BSTree]\!]$$
$$\text{GSyntax } [\![Int]\!] = \ gen\_\text{Lower } [\![Int]\!]$$

$$\text{Con } [\![Nil]\!] = \text{ Lower } [\![Nil]\!]$$
$$\text{Con } [\![Cons\ Int\ BSTree\ BSTree]\!] = \text{ Lower } [\![Cons]\!]\ (\text{ Var } [\![Int]\!]\ ,\ \text{Var } [\![BSTree]\!]\ ,\ \text{Var } [\![BSTree]\!]\ )$$

$$\text{Pred } [\![Cons\ Int\ BSTree\ BSTree]\!] = \text{ GType } [\![Int]\!]\ \wedge\ \text{GType } [\![BSTree]\!]\ \wedge\ \text{GType } [\![BSTree]\!]$$

$$\text{Lower } [\![BSTree]\!] = \texttt{BSTree}$$
$$\text{Lower } [\![Nil]\!] = \texttt{nil}$$
$$\text{Lower } [\![Cons]\!] = \texttt{cons}$$
$$\text{Lower } [\![Int]\!] = \texttt{int}$$

$$\text{Var } [\![Int]\!] = \texttt{X21} \qquad \text{For } atype_{2,1} \equiv Int$$
$$\text{Var } [\![BSTree]\!] = \texttt{X22} \qquad \text{For } atype_{2,2} \equiv BSTree$$
$$\text{Var } [\![BSTree]\!] = \texttt{X23} \qquad \text{For } atype_{2,3} \equiv BSTree$$

B.1.6 SomeWeird

HSyntax $[\![$`data` $SomeWeird = Nil1 \mid Nil2 \mid Some\ Int \mid Weird\ SomeWeird\ SomeWeird\ Bool]\!] =$
Rules $[\![SomeWeird = Nil1 \mid Nil2 \mid Some\ Int \mid Weird\ SomeWeird\ SomeWeird\ Bool]\!]$

Rules $[\![SomeWeird = Nil1 \mid Nil2 \mid Some\ Int \mid Weird\ SomeWeird\ SomeWeird\ Bool]\!] =$
Rule $[\![SomeWeird = Nil1]\!]\ \vee$
Rule $[\![SomeWeird = Nil2]\!]\ \vee$
Rule $[\![SomeWeird = Weird\ Some\ Int]\!]\ \vee$
Rule $[\![SomeWeird = Weird\ SomeWeird\ SomeWeird\ Bool]\!]$

Rule $[\![SomeWeird = Nil1]\!] =$ GSimpleType $[\![SomeWeird]\!]$(Con $[\![Nil1]\!]$) .
Rule $[\![SomeWeird = Nil2]\!] =$ GSimpleType $[\![SomeWeird]\!]$(Con $[\![Nil2]\!]$) .
Rule $[\![SomeWeird = Some\ Int]\!] =$ GSimpleType $[\![SomeWeird]\!]$(Con $[\![Some\ Int]\!]$) :− Pred $[\![Some\ Int]\!]$.
Rule $[\![SomeWeird = Weird\ SomeWeird\ SomeWeird\ Bool]\!] =$
GSimpleType $[\![SomeWeird]\!]$(Con $[\![Weird\ SomeWeird\ SomeWeird\ Bool]\!]$) :−
Pred $[\![Weird\ SomeWeird\ SomeWeird\ Bool]\!]$.

GSimpleType $[\![SomeWeird]\!] =$ GSyntax $[\![SomeWeird]\!]$

GType $[\![SomeWeird]\!] =$ GSyntax $[\![SomeWeird]\!]$(Var $[\![SomeWeird]\!]$)
GType $[\![Int]\!] =$ GSyntax $[\![Int]\!]$(Var $[\![Int]\!]$)
GType $[\![Bool]\!] =$ GSyntax $[\![Bool]\!]$(Var $[\![Bool]\!]$)

GSyntax $[\![SomeWeird]\!] =$ Lower $[\![SomeWeird]\!]$
GSyntax $[\![Bool]\!] =$ *gen*_Lower $[\![Bool]\!]$

Con $[\![Nil1]\!] =$ Lower $[\![Nil1]\!]$
Con $[\![Nil2]\!] =$ Lower $[\![Nil2]\!]$
Con $[\![Some\ Int]\!] =$ Lower $[\![Some]\!]$ (Var $[\![Int]\!]$)
Con $[\![Weird\ SomeWeird\ SomeWeird\ Bool]\!] =$ Lower $[\![Weird]\!]$ (Var $[\![Bool]\!]$, Var $[\![SomeWeird]\!]$, Var $[\![SomeWeird]\!]$)

Pred $[\![Some\ Int]\!] =$ GType $[\![Int]\!]$
Pred $[\![Weird\ SomeWeird\ SomeWeird\ Bool]\!] =$ GType $[\![SomeWeird]\!]$ $\wedge$ GType $[\![SomeWeird]\!]$ $\wedge$ GType $[\![Bool]\!]$

Lower $[\![SomeWeird]\!] =$ `SomeWeird`
Lower $[\![Nil1]\!] =$ `nil1`
Lower $[\![Nil2]\!] =$ `nil2`
Lower $[\![Weird]\!] =$ `weird`
Lower $[\![Int]\!] =$ `int`
Lower $[\![Bool]\!] =$ `bool`

Var $[\![Int]\!] =$ `X31` For $atype_{3,1} \equiv Int$
Var $[\![SomeWeird]\!] =$ `X41` For $atype_{4,1} \equiv SomeWeird$
Var $[\![SomeWeird]\!] =$ `X42` For $atype_{4,2} \equiv SomeWeird$
Var $[\![Bool]\!] =$ `X43` For $atype_{4,3} \equiv Bool$

B.1.7 RSTree

$$\text{HSyntax}\, [\![\texttt{data}\ RSTree = R\ Int\ [RSTree]]\!] = \text{Rules}\, [\![RSTree = R\ Int\ [RSTree]]\!]$$

$$\text{Rules}\, [\![RSTree = R\ Int\ [RSTree]]\!] = \text{Rule}\, [\![RSTree = R\ Int\ [RSTree]]\!]$$

$$\begin{aligned}\text{Rule}\, [\![RSTree = R\ Int\ [RSTree]]\!] = \ & \text{GSimpleType}\, [\![RSTree]\!](\text{Con}\, [\![R\ Int\ [RSTree]]\!]) \quad :- \\ & \text{Pred}\, [\![R\ Int\ [RSTree]]\!]\ .\end{aligned}$$

$$\text{GSimpleType}\, [\![RSTree]\!] = \text{GSyntax}\, [\![RSTree]\!]$$

$$\text{GType}\, [\![\ [RSTree]\ ]\!] = \text{GList}\, [\![RSTree]\!]$$
$$\text{GType}\, [\![Int]\!] = \text{GSyntax}\, [\![Int]\!](\ \text{Var}\, [\![Int]\!]\ )$$

$$\text{GList}\, [\![RSTree]\!] = \mathit{list}\,(\text{GSyntax}\, [\![RSTree]\!], \text{Var}\, [\![RSTree]\!])$$

$$\text{GSyntax}\, [\![RSTree]\!] = \text{Lower}\, [\![RSTree]\!]$$
$$\text{GSyntax}\, [\![Int]\!] = \mathit{gen}\_\text{Lower}\, [\![Int]\!]$$

$$\text{Con}\, [\![R\ Int\ [RSTree]]\!] = \text{Lower}\, [\![R]\!]\ (\ \text{Var}\, [\![Int]\!]\ ,\ \text{Var}\, [\![\ [RSTree]\ ]\!]\ )$$

$$\text{Pred}\, [\![R\ Int\ [RSTree]]\!] = \text{GType}\, [\![Int]\!] \quad \wedge \quad \text{GType}\, [\![\ [RSTree]\ ]\!]$$

$$\text{Lower}\, [\![RSTree]\!] = \texttt{rstree}$$
$$\text{Lower}\, [\![R]\!] = \texttt{r}$$
$$\text{Lower}\, [\![Int]\!] = \texttt{int}$$

$$\text{Var}\, [\![Int]\!] = \texttt{X11} \qquad \text{For } atype_{1,1} \equiv Int$$
$$\text{Var}\, [\![RSTree]\!] = \texttt{X12} \qquad \text{For } atype_{1,2} \equiv RSTree$$

B.2 Monomorphic Type with Boundaries

B.2.1 Syntax-Translation Function

We define HBSyntax a refinement of the function HSyntax

$$\text{HBSyntax}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$$

as an extension of our syntax-translation semantic for monomorphic types with boundaries as follow:

$$
\begin{array}{l}
\text{HBSyntax}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{HBSyntax}\, [\![\texttt{data} \ \ \mathit{simpletype} \ = \ \mathit{constrs}]\!] \ = \ \text{Rules}\, [\![\mathit{simpletype} \ = \ \mathit{constrs}]\!] \\
\\
\text{Rules}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{Rules}\, [\![\mathit{simpletype} \ = \ \mathit{constrs}]\!] \ = \ \text{Rules}\, [\![\mathit{simpletype} \ = \ \mathit{constr}_1 \ | \ \mathit{constr}_2 \ | \ \ldots \ | \ \mathit{constr}_n]\!] \\
\text{Rules}\, [\![\mathit{simpletype} \ = \ \mathit{constr}_1 \ | \ \mathit{constr}_2 \ | \ \ldots \ | \ \mathit{constr}_n]\!] \ = \ \bigvee_{i \in \{1,\ldots,n\}} \text{Rule}\, [\![\mathit{simpletype} \ = \ \mathit{constr}_i]\!] \\
\text{Rule}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{Rule}\, [\![\mathit{simpletype} \ = \ \mathit{constr}_i]\!] \ = \ \text{GSimpleType}\, [\![\mathit{simpletype}]\!] \, (\ \text{VarRec}\, [\![\mathit{constr}_i]\!], \ \text{Con}\, [\![\mathit{constr}_i]\!] \ ) \ \ :- \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Boundary}\, [\![\mathit{constr}_i]\!] \ \text{Pred}\, [\![\mathit{constr}_i]\!] \ . \\
\\
\text{GSimpleType}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{GSimpleType}\, [\![\mathit{simpletype}]\!] \ = \ \text{GSyntax}\, [\![\mathit{tycon}]\!] \\
\\
\text{GType}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{GType}\, [\![\mathit{atype}_{i,j}]\!] \ = \ \text{GSyntax}\, [\![\mathit{gtycon}]\!] \, (\ \text{VarRec}\, [\![\mathit{constr}_i]\!], \ \text{Var}\, [\![\mathit{atype}_{i,j}]\!] \ ) \qquad \text{If } \mathit{atype}_{i,j} \rightsquigarrow \mathit{gtycon} \\
\text{GType}\, [\![\mathit{atype}_{i,j}]\!] \ = \ \text{GTuple}\, [\![\mathit{atype}_{i,j}]\!] \qquad \text{If } \mathit{atype}_{i,j} \rightsquigarrow (\mathit{type}_1, \ \ldots \ \mathit{type}_k) \\
\text{GType}\, [\![\mathit{atype}_{i,j}]\!] \ = \ \text{GList}\, [\![\mathit{atype}_{i,j}]\!] \qquad \text{If } \mathit{atype}_{i,j} \rightsquigarrow [\mathit{type}] \\
\\
\text{GSyntax}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{GSyntax}\, [\![\mathit{tycon}]\!] \ = \ \text{Lower}\, [\![\mathit{tycon}]\!] \qquad \text{If } \mathit{tycon} \text{ is not a primitive type} \\
\text{GSyntax}\, [\![\mathit{tycon}]\!] \ = \ \mathit{gen}\_\text{Lower}\, [\![\mathit{tycon}]\!] \qquad \text{If } \mathit{tycon} \text{ is a primitive type} \\
\text{GSyntax}\, [\![\mathit{type}]\!] \ = \ \text{Lower}\, [\![\mathit{type}]\!] \qquad \text{If } \mathit{type} \text{ is not a primitive type} \\
\text{GSyntax}\, [\![\mathit{type}]\!] \ = \ \mathit{gen}\_\text{Lower}\, [\![\mathit{type}]\!] \qquad \text{If } \mathit{type} \text{ is a primitive type} \\
\text{GSyntax}\, [\![\mathit{gtycon}]\!] \ = \ \text{GSyntax}\, [\![\mathit{qtycon}]\!] \qquad \text{If } \mathit{gtycon} \rightsquigarrow \mathit{qtycon} \\
\text{GSyntax}\, [\![\mathit{gtycon}]\!] \ = \ \text{GSyntax}\, [\![()]\!] \qquad \text{If } \mathit{gtycon} \rightsquigarrow () \\
\text{GSyntax}\, [\![\mathit{qtycon}]\!] \ = \ \text{GSyntax}\, [\![\mathit{tycon}]\!] \\
\text{GSyntax}\, [\![()]\!] \ = \ \mathit{gen\_unit} \\
\\
\text{GTuple}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{GTuple}\, [\![\mathit{atype}_{i,j}]\!] \ = \ \text{GTuple}\, [\![(\mathit{type}_1, \ \ldots \ \mathit{type}_k)]\!] \\
\text{GTuple}\, [\![(\mathit{type}_1, \ \ldots \ \mathit{type}_k)]\!] \ = \ \mathit{tuple}\, (\text{GSyntax}\, [\![\mathit{type}_1]\!], \ \ldots, \ \text{GSyntax}\, [\![\mathit{type}_k]\!], \ \text{Var}\, [\![\mathit{atype}_{i,j}]\!]) \\
\\
\text{GList}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{GList}\, [\![\mathit{atype}_{i,j}]\!] \ = \ \text{GList}\, [\![[\mathit{type}]]\!] \\
\text{GList}\, [\![[\mathit{type}]]\!] \ = \ \mathit{list}(\text{GSyntax}\, [\![\mathit{type}]\!], \ \text{Var}\, [\![\mathit{atype}_{i,j}]\!])
\end{array}
$$

$\text{Con}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$
$\text{Con}\ [\![constr_i]\!] \ = \ \text{Con}\ [\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \ \ldots\ \ atype_{i,m}]\!]$
$\text{Con}\ [\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \ \ldots\ \ atype_{i,m}]\!] \ = \ \text{Lower}\ [\![con_i]\!]\ (\ \text{Var}\ [\![atype_{i,1}]\!], \text{Var}\ [\![atype_{i,2}]\!],\ \ldots,\ \text{Var}\ [\![atype_{i,m}]\!]\ )$

$\text{Boundary}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$
$\text{Boundary}\ [\![constr_i]\!] \ = \ \text{Boundary}\ [\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \ \ldots\ \ atype_{i,m}]\!]$
$\text{Boundary}\ [\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \ \ldots\ \ atype_{i,m}]\!] \ = \ \bigwedge_{j\in\{1,\ldots,m\}} base(\text{VarRec}\ [\![atype_{i,j}]\!])$

If we are in the base case of the structural induction

$\text{Boundary}\ [\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \ \ldots\ \ atype_{i,m}]\!] \ =$
$\text{VarRec}\ [\![constr_i]\!] > \text{InitRec}\ [\![constr_i]\!],$
$N_{r_i}\ \text{ is }\ \text{VarRec}\ [\![constr_i]\!] - 1$
$states_i(N_{r_i},\ \text{VarRec}\ [\![atype_{i,1}]\!],\ \ldots,\ \text{VarRec}\ [\![atype_{i,m}]\!])$

$\text{Pred}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$
$\text{Pred}\ [\![constr_i]\!] \ = \ \text{Pred}\ [\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \ \ldots\ \ atype_{i,m}]\!]$
$\text{Pred}\ [\![con_i\ atype_{i,1}\ \ atype_{i,2}\ \ \ldots\ \ atype_{i,m}]\!] \ = \ \bigwedge_{j\in\{1,\ldots,m\}} \text{GType}\ [\![atype_{i,j}]\!]$

$\text{Var}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$
$\text{Var}\ [\![atype_{i,j}]\!] \ = \ X_{i,j}$ — Where $X_{i,j}$ is a variable
$\text{Var}\ [\![exp]\!] \ = \ X$ — Where X is a variable

$\text{VarRec}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$
$\text{VarRec}\ [\![constr_i]\!] \ = \ N_i^0$ — If we are in the base case of the structural induction
$\text{VarRec}\ [\![constr_i]\!] \ = \ N_i$ — If we are in the induction scenario of the structural induction
$\text{VarRec}\ [\![atype_{i,j}]\!] \ = \ N_{i,j}^0$ — If we are in the base case of the structural induction
$\text{VarRec}\ [\![atype_{i,j}]\!] \ = \ N_{i,j}$ — If we are in the induction scenario of the structural induction
$\text{VarRec}\ [\![exp]\!] \ = \ N$ — Where N is a variable which represents recursion

$\text{Lower}: \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$
$\text{Lower}\ [\![exp]\!] \ = \ {}'exp$ — Where $'exp$ is the expression exp in lower case

where

$$states_i(N_{r_i},\ W_{i,1},\ \ldots,\ W_{i,k}) :- \bigwedge_{j\in\{1,\ldots,m\}} cases(N_{r_i}, W_{i,j}) \ \wedge\ diff(W_{i,1},\ \ldots,\ W_{i,k})$$

$cases(\_, Y) :-\ base(Y).$
$cases(X, X).$
$diff(X,\ \overset{k_i}{\ldots},\ X) :-\ base(X),\ !,\ fail.$
$diff(W_{i,1},\ \ldots,\ W_{i,k}).$

B.3 Polymorphic Type

B.3.1 Syntax-Translation Function

We define HBSyntax the refinement of the function HSyntax

$$\text{HGSyntax} : \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}$$

as an extension of our syntax-translation semantic for polymorphic types as follow:

$$
\begin{array}{l}
\text{HGSyntax} : \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{HGSyntax}\, [\![\texttt{data} \ \ simpletype \ = \ constrs]\!] \ = \ \text{Rules}\, [\![simpletype \ = \ constrs]\!] \\
\\
\text{Rules} : \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{Rules}\, [\![simpletype \ = \ constrs]\!] \ = \ \text{Rules}\, [\![simpletype \ = \ constr_1 \ | \ constr_2 \ | \ \ldots \ | \ constr_n]\!] \\
\text{Rules}\, [\![simpletype \ = \ constr_1 \ | \ constr_2 \ | \ \ldots \ | \ constr_n]\!] \ = \ \bigvee_{i \in \{1,\ldots,n\}} \text{Rule}\, [\![simpletype \ = \ constr_i]\!] \\
\text{Rule} : \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{Rule}\, [\![simpletype \ = \ constr_i]\!] \ = \ \text{Rule}\, [\![tycon \ \ tyvar_1 \ \ \ldots \ \ tyvar_k \ = \ constr_i]\!]. \\
\text{Rule}\, [\![tycon \ \ tyvar_1 \ \ \ldots \ \ tyvar_k \ = \ constr_i]\!] \ = \\
\qquad \text{GSimpleType}\, [\![simpletype]\!] \,(\ \text{Var}\, [\![tyvar_1]\!], \ \ldots \ \text{Var}\, [\![tyvar_k]\!], \ \text{Con}\, [\![constr_i]\!] \ ) \ \ :- \\
\qquad\qquad \bigwedge_{t \in \{1,\ldots,k\}} \text{Gen}\, [\![tyvar_t]\!] \ \ \text{Pred}\, [\![constr_i]\!] \ . \\
\text{GSimpleType} : \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{GSimpleType}\, [\![simpletype]\!] \ = \ \text{GSyntax}\, [\![tycon]\!] \\
\\
\text{GType} : \ \mathcal{H}_{\text{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}} \\
\text{GType}\, [\![atype_{i,j}]\!] \ = \ \text{GSyntax}\, [\![gtycon]\!] \,(\ \text{Var}\, [\![atype_{i,j}]\!] \ ) \qquad \text{If } atype_{i,j} \leadsto gtycon \\
\text{GType}\, [\![atype_{i,j}]\!] \ = \ \text{GTuple}\, [\![atype_{i,j}]\!] \qquad \text{If } atype_{i,j} \leadsto (type_1, \ \ldots \ type_k) \\
\text{GType}\, [\![atype_{i,j}]\!] \ = \ \text{GList}\, [\![atype_{i,j}]\!] \qquad \text{If } atype_{i,j} \leadsto [type]
\end{array}
$$

$$\begin{aligned}
&\text{GSyntax} : \ \mathcal{H}_{\texttt{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}\\
&\text{GSyntax}\ [\![\mathit{tycon}]\!] \ = \ \text{Lower}\ [\![\mathit{tycon}]\!]\\
&\text{GSyntax}\ [\![\mathit{type}]\!] \ = \ \text{Lower}\ [\![\mathit{type}]\!]\\
&\text{GSyntax}\ [\![\mathit{gtycon}]\!] \ = \ \text{GSyntax}\ [\![\mathit{qtycon}]\!] \qquad \text{If } \mathit{gtycon} \rightsquigarrow \mathit{qtycon}\\
&\text{GSyntax}\ [\![\mathit{gtycon}]\!] \ = \ \text{GSyntax}\ [\![()]\!] \qquad \text{If } \mathit{gtycon} \rightsquigarrow ()\\
&\text{GSyntax}\ [\![\mathit{qtycon}]\!] \ = \ \text{GSyntax}\ [\![\mathit{tycon}]\!]\\
&\text{GSyntax}\ [\![()]\!] \ = \ \mathit{gen_unit}
\end{aligned}$$

$$\begin{aligned}
&\text{GTuple} : \ \mathcal{H}_{\texttt{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}\\
&\text{GTuple}\ [\![\mathit{atype}_{i,j}]\!] \ = \ \text{GTuple}\ [\![(\mathit{type}_1, \ \ldots\ \mathit{type}_k)]\!]\\
&\text{GTuple}\ [\![(\mathit{type}_1, \ \ldots\ \mathit{type}_k)]\!] \ = \ \mathit{tuple}\,(\text{GSyntax}\ [\![\mathit{type}_1]\!], \ \ldots, \ \text{GSyntax}\ [\![\mathit{type}_k]\!], \ \text{Var}\ [\![\mathit{atype}_{i,j}]\!])
\end{aligned}$$

$$\begin{aligned}
&\text{GList} : \ \mathcal{H}_{\texttt{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}\\
&\text{GList}\ [\![\mathit{atype}_{i,j}]\!] \ = \ \text{GList}\ [\![[\mathit{type}]]\!]\\
&\text{GList}\ [\![[\mathit{type}]]\!] \ = \ \mathit{list}(\text{GSyntax}\ [\![\mathit{type}]\!], \ \text{Var}\ [\![\mathit{atype}_{i,j}]\!])
\end{aligned}$$

$$\begin{aligned}
&\text{Con} : \ \mathcal{H}_{\texttt{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}\\
&\text{Con}\ [\![\mathit{constr}_i]\!] \ = \ \text{Con}\ [\![\mathit{con}_i\ \mathit{atype}_{i,1}\ \ \mathit{atype}_{i,2}\ \ \ldots\ \ \mathit{atype}_{i,m}]\!]\\
&\text{Con}\ [\![\mathit{con}_i\ \mathit{atype}_{i,1}\ \ \mathit{atype}_{i,2}\ \ \ldots\ \ \mathit{atype}_{i,m}]\!] \ = \ \text{Lower}\ [\![\mathit{con}_i]\!]\ (\ \text{Var}\ [\![\mathit{atype}_{i,1}]\!], \text{Var}\ [\![\mathit{atype}_{i,2}]\!], \ \ldots, \ \text{Var}\ [\![\mathit{atype}_{i,m}]\!]\ )
\end{aligned}$$

$$\begin{aligned}
&\text{Gen} : \ \mathcal{H}_{\texttt{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}\\
&\text{Gen}\ [\![\mathit{tyvar}_t]\!] \ = \ \mathit{gen}(\text{Var}\ [\![\mathit{tyvar}_t]\!], \ \text{Var}\ [\![\mathit{atype}_{i,j_t}]\!])
\end{aligned}$$

$$\begin{aligned}
&\text{Pred} : \ \mathcal{H}_{\texttt{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}\\
&\text{Pred}\ [\![\mathit{constr}_i]\!] \ = \ \text{Pred}\ [\![\mathit{con}_i\ \mathit{atype}_{i,1}\ \ \mathit{atype}_{i,2}\ \ \ldots\ \ \mathit{atype}_{i,m}]\!]\\
&\text{Pred}\ [\![\mathit{con}_i\ \mathit{atype}_{i,1}\ \ \mathit{atype}_{i,2}\ \ \ldots\ \ \mathit{atype}_{i,m}]\!] \ = \ \bigwedge_{j \in \{1,\ldots,m\}} \text{GType}\ [\![\mathit{atype}_{i,j}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{Var} : \ \mathcal{H}_{\texttt{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}\\
&\text{Var}\ [\![\mathit{atype}_{i,j}]\!] \ = \ X_{i,j} \qquad \text{Where } X_{i,j} \text{ is a variable}\\
&\text{Var}\ [\![\mathit{exp}]\!] \ = \ X \qquad \text{Where } X \text{ is a variable}
\end{aligned}$$

$$\begin{aligned}
&\text{Lower} : \ \mathcal{H}_{\texttt{ADT}} \ \longrightarrow \ \mathcal{P}_{\text{Clause}}\\
&\text{Lower}\ [\![\mathit{exp}]\!] \ = \ '\mathit{exp} \qquad \text{Where } '\mathit{exp} \text{ is the expression } \mathit{exp} \text{ in lower case}
\end{aligned}$$