Master's Degree in Formal Methods in Computer Science

Facultad de Informática,

Universidad Complutense de Madrid

Master's Thesis

Academic year 2023 / 2024

Automatic test-case generation for Haskell based on dependent types

Generación automática de casos de prueba para Haskell basada en tipos dependientes

Author: Pablo Castellanos García

Advisors: Julio Mariño Carballo, Ignacio Ballesteros González

ETSI Informáticos, Universidad Politécnica de Madrid

Defended on 10th July, 2024

Obtained a calification of 10/10

Contents

1 Introduction **1**

1.1 Motivation 3

1.2 The solution strategy 4

1.3 Document structure 4

2 Preliminaries **5**

2.1 QuickCheck 5

2.1.1 Generators 5

2.1.2 Testing programs 7

2.2 Template Haskell 8

2.2.1 Quotations 8

2.2.2 The internal representation 9

2.2.3 Splices 13

2.3 Prolog 13

2.3.1 Horn clauses 14

2.3.2 Prolog syntax 15

2.3.3 Prolog semantics 18

2.3.4 Some predefined predicates 21

3 Producing test case generators **25**

3.1 Representation of data declarations . . . 25

3.2 Translation . . . 28

3.2.1 First approach . . . 28

3.2.2 Second approach: sized generation . . . 39

3.2.3 Third approach: uniform generation . . . 53

4 Simulating dependent types **57**

4.1 GADTs . . . 57

4.2 Applications . . . 60

4.2.1 Height-balanced red-black trees . . . 61

5 Test case generators for GADTs **65**

5.1 The supported syntax . . . 65

5.2 The new translation rules . . . 67

5.2.1 First approach: adapting the old rules . . . 67

5.2.2 Second approach: the new strategy . . . 69

6 Conclusions **80**

6.1 Future work . . . 80

Bibliography **82**

A The full implementation **84**

A.1 General auxiliary functions . . . 84

A.2 First stage: standard ADTs . . . 85

A.2.1 First approach . . . 86

A.2.2 Second approach: sized generation 89

A.2.3 Third approach: uniform generation 93

A.3 Second stage: GADTs 98

Automatic test-case generation for Haskell based on dependent types

Abstract

Software testing is the simplest and most widespread approach to finding bugs in computer programs, where *bugs* are understood as a discrepancy between the *expected* output of the program and its *actual* output. Testing has the disadvantage of requiring a previous generation of a suite of test cases—inputs to the program that are representative of the whole input space—together with the expected outputs. *Property-based* testing addresses this issue by requiring only a formal specification of the intended behavior of the code and generating test cases *automatically*: then, the functional correctness of the program can be tested empirically by checking if the output of the program when given the generated test cases as input satisfies the specification.

A naïve implementation of property-based testing behaves poorly when the tested program has non-trivial *preconditions*: these are usually written in the specification as a conditional statement of the form "*if* the input meets some condition, *then* the output should satisfy some other condition", but values generated at random frequently do not satisfy the precondition, meaning that the specification holds vacuously—a false premise makes any implication true, regardless of the conclusion.

Our work aims to solve the above problem by using a dependent type system to encode the desired preconditions at the type level. We start by developing a mechanism to automatically construct test case generators from data type declarations. Then, we show how dependent types can be simulated in Haskell using advanced features such as GADTs and data type promotion, and finally we extend our mechanism to translate (simulated) dependent type declarations into test case generators that only produce values satisfying the specified precondition.

Keywords Haskell, Property-Based Testing, GADTs, Dependent types, Prolog

Generación automática de casos de prueba para Haskell basada en tipos dependientes

Resumen

El *testing* es el método más sencillo y extendido para encontrar errores en programas de ordenador, entendiendo un *error* como una discrepancia entre la salida que se *espera* del programa y la salida que *realmente produce* éste. No obstante, el *testing* tiene el inconveniente de que requiere haber producido previamente una serie de casos de prueba—valores de entrada al programa que sean representativos de todo el espacio de posibles entradas—así como las salidas esperadas. El *testing basado en propiedades* (*property-based testing* o PBT, en inglés) aborda este problema requiriendo únicamente una especificación formal del comportamiento esperado del programa, y generando los casos de prueba de manera automática: así, la corrección funcional del programa se verifica empíricamente comprobando si la salida que produce programa al recibir como entrada los valores generados satisface o no la especificación.

Una implementación ingenua de PBT puede dar malos resultado cuando el programa analizado tiene *precondiciones* no triviales: éstas normalmente se codifican en la especificación como un enunciado condicional de la forma "*si* la entrada satisface alguna condición, *entonces* la salida debería satisfacer alguna otra ccondición", pero es común que los valores generados al azar no cumplan la precondición, haciendo que la especificación se verifique de manera trivial, ya que una premisa falsa hace cierta cualquier implicación, sin importar cuál sea la conclusión.

Este trabajo pretende resolver el problema descrito usando un sistema de tipos dependientes para codificar las precondiciones. En primer lugar, desarrollamos un mecanismo para construir automáticamente generadores de casos de prueba a partir de declaraciones de tipos de datos. A continuación, mostramos cómo se puede simular un sistema de tipos dependientes en Haskell usando características avanzadas como GADTs o promoción de tipos de datos, y finalmente extendemos el mecanismo de traducción para transformar estas declaraciones de tipos dependientes simulados en generadores de casos de prueba que solo produzcan valores que satisfagan la precondición establecida.

Palabras clave Haskell, *Property-Based Testing*, GADTs, Tipos dependientes, Prolog

Chapter 1

Introduction

It is safe to say that software is omnipresent in the modern world, and an increasingly big part of our everyday lives depends on it: the systems in charge of distributing the workloads in power grids, the on-board computers in airplanes, the controllers for medical machines like fMRI scanners and the personal computers most of us use for work or entertainment are just a few examples of pieces of technology we rely on on a daily basis, and all of them are based on extremely complex software systems. Naturally, there is no need to emphasise the importance of having strong guarantees that these systems contain no errors and behave as we expect them to, even in unforeseeable situations. Unfortunately, this is not usually the case in the real world: in practice, software bugs are unavoidable unless a lot of effort is placed on looking for them.

The simplest approach to *systematically* looking for software bugs is *testing*, i.e. running the program on many different inputs (*test cases*) and checking whether it returns the expected outputs. This method is frequently used in practice during the software development cycle because it is very easy to implement, but it has the disadvantage that the test cases need to be produced somehow and, more importantly, the expected outputs also need to be calculated for each input. A bigger problem with testing, however, is that it can never be used to prove that a program is correct, only that it is incorrect: if an error is found then the program is clearly incorrect, but if no problems are detected one cannot know if it is because the program is correct or because it is incorrect and the bug just happens to be unrelated to the tested inputs—unless, of course, the input space is so small that it can be searched exhaustively, but this situation is the exception rather than the rule.

Another, more complex approach is that of *formal verification*, that is, proving—in the mathematical sense—that the program satisfies some *specification*, written in a formal

language with well-defined semantics like mathematical logic. The obvious benefit of this approach over testing is that it can give absolute certainty in the correctness of a program, and even when the program is incorrect it can often help in detecting the error. On the other hand, it is not possible, in general, to prove properties about the *semantics* of a program automatically from a static description (i.e. its source code). Even though some tools have been developed to assist with producing these proofs [1, 2], it is not an easy task and usually requires a large amount of user interaction.

Property-Based Testing (PBT, also known as *generative testing*) is, in a sense, a meeting point for the two previous approaches: it is a form of testing in which the expected behaviour of the program is described by means of a formal specification; this allows for the automatic generation of test cases and eliminates the need of producing the expected output corresponding to each input. The Haskell library *QuickCheck* [3, 4] was a pioneer in the development of this type of testing, and its popularisation led to the creation of other similar PBT libraries for other programming languages [5]. This approach is superior to traditional testing methods in that it can run completely automatically, since it is not based on comparing *actual* output values with *expected* ones. Instead, the difficulties are placed in (a) generating appropiate test cases, in a way that is representative of the actual input space distribution, and (b) writing precise specifications that closely model the desired properties. The first issue relates to the implementation of the testing infrastructure, while the second one relates to its use.

A different method for finding errors in programs is *static analysis*, that is, the analysis of the behaviour of programs without ever executing them—in this sense, it is closely related to formal verification, although its focus is much more restricted: instead of attempting to prove the overall correctness of the program (which, as we mentioned before, cannot be done automatically), it only tries to prove a limited set of properties, like absence of dead code or divisions by zero (which can be proven automatically in many cases).

A particular kind of static analysis is *type* analysis, which is frequently used to avoid many common programming mistakes. In the last decades, many advances have been made on the topic of type systems, greatly increasing their expressiveness and therefore their ability to catch bugs *at compile time*. In particular, two closely related trends in type system research are *refinement* types and *dependent* types. The former are regular types that are "refined" with a logical predicate, and the latter are types whose definition is allowed to depend on a value; for instance, within these type systems one could refer to the type of *integers between 0 and 7*, the type of *lists containing the number 42* or the type of *increasingly sorted lists*. In recent years, many programming languages have emerged that are based on these ideas [6, 7, 8, 1, 9], showing the academic interest in the topic. All of these belong to the functional programming paradigm, and most are either based

on Haskell or strongly influenced by it. As a proof of their expressive power, note that many are advertised as programming languages *and proof assistants*; in fact, some of them have helped in proving and/or checking proofs of cutting-edge mathematical theorems and include libraries with common proof tactics and mathematical results.

1.1 Motivation

Property-based testing is based on the idea of writing precise *specifications* of the intended behavior of the code, as properties that are expected to hold. Such specifications very often take the form of *preconditions* for a function call: properties that any given value should satisfy before being passed to the function.

For instance, consider the classical example of the *merge sort* algorithm: a given list to be sorted is decomposed into two equal-size lists, these are sorted recursively, and the results are merged back together in a way that preserves their relative order. The splitting and the recursive sorting do not have any particular precondition, in the sense that they could be applied to any list and their output value would still be correct; the `merge` function, on the other hand, requires that its two arguments are sorted for the overall result to be sorted as well.

Informally, one could specify the correctness of this function in a logical language as "*if* the two arguments are sorted, *then* the output is also sorted". To empirically test the correctness of an implementation of `merge`, a naïve PBT implementation would generate many pairs of lists to be passed as arguments—which would *not* be sorted, with high probability—and then check whether the correctness condition is satisfied. Since this property is defined as an implication and the premise is false, the implication would hold vacuously and so the implementation of `merge` would pass the tests even though nothing was actually checked about its behaviour.

QuickCheck is somewhat aware of this problem and issues a warning when some number of verification conditions hold vacuously in this sense; this is a good thing because it informs the user that the valid tests are not actually giving any guarantees at all about the correctness of the program, but it is still unsatisfactory because nothing can be done, when receiving the warning, other than trying to generate more test cases.

For precisely this reason, it is desirable to define more powerful test generators for specialised data types that have a more constrained structure. These stricter data types include types such as sorted lists, but also type-safe data structures like red-black trees or even well-typed expressions in a language with several incompatible types.

QuickCheck does allow the user to define custom generators for specific data types, instead of relying on the built-in ones, but these generators need to be written by hand and in a case-by-case basis. The goal of this work, therefore, is to create test case generators in the spirit of QuickCheck but with as little user intervention as possible.

1.2 The solution strategy

Our main approach for *automatically* creating test case generators is to inspect the Haskell code containing the data type definitions and use it to mechanically write a *program* to produce the test cases. In particular, we use Template Haskell [10] to programatically inspect Haskell code and Prolog to carry out the generation.

1.3 Document structure

We start by presenting the basic background knowledge needed to understand the exposition in Chapter 2, providing references for the interested reader who wishes to learn more; specifically, we introduce the basics of QuickCheck in § 2.1, of Template Haskell in § 2.2 and of Prolog in § 2.3.

In Chapter 3 we present, in three successive stages, the main translation mechanism from standard Haskell data type declarations to Prolog code. Then, in Chapter 4, we show how one can use a generalised version of Haskell's type system to encode stronger information about data values at the type level, and finally in, Chapter 5, we extend the translation mechanism from Chapter 3 to deal with the new types introduced in Chapter 4.

The developed code can be found in § A.2 and § A.3 for Chapters 3 and 5 respectively.

Chapter 2

Preliminaries

2.1 QuickCheck

QuickCheck [3, 4] is a Haskell library for property-based testing; in fact, it is the project that introduced this testing technique and has even been called 'the grandfather of property-based testing libraries' [11]. We include here a brief introduction to its usage; the interested reader can find a more in-depth exposition about its design and its more advanced features in online tutorials such as [11, 12].

2.1.1 Generators

QuickCheck deals with random value generation by introducing, for each type `a`, the polymorphic *generator* type, `Gen a`, which essentially contains wrappers around functions that return a value of type `a` given a random number generator and an integer (representing a 'size' parameter):

```
newtype Gen a = MkGen {
  unGen :: QCGen -> Int -> a
}
```

Elements of the generator type `Gen a` can be used through the `generate` function, which takes a value of type `Gen a` and returns randomly generated values of type `a` (wrapped inside the `IO` monad),

```
generate :: Gen a -> IO a
```

To identify the types that can be used in testing (i.e. those that can be generated automatically), QuickCheck provides a type class `Arbitrary`, with the following definition:

```
class Arbitrary a where
  arbitrary   :: Gen a
  coarbitrary :: a -> Gen b -> Gen b
```

The class method `arbitrary` guarantees that any type `a` belonging to the class `Arbitrary` can be used for automatic testing, since values of this type can be generated through the returned `Gen a`. On the other hand, the optional class method `coarbitrary` guarantees that *functions returning values of type* `a` can be generated automatically, though we will not go deeper into this.

QuickCheck provides instances of class `Arbitrary` for the usual basic types (integers, booleans, pairs, triples, etc.) To define a custom data type as an instance of class `Arbitrary`, the user must provide, at least, an implementation of the `arbitrary` method, which means returning values of type `Gen a`. This could be done completely manually but QuickCheck includes some combinators to create `Gen a` values more easily, some of which are listed below:

- `choose :: Random a => (a,a) -> Gen a`, which produces a random element in the range specified as the argument.
- `elements :: [a] -> Gen a`, which picks an element at random from the list.
- `oneof :: [Gen a] -> Gen a`, which randomly picks one generator from a list of previously defined generators.
- `frequency :: [(Int, Gen a)] -> Gen a`, which works in a similar way to `oneof` but assigning *weights* to each of the generators.

Moreover, the type constructor `Gen` is an instance of the type class `Monad` (and therefore also `Applicative`), so different generators can be easily combined to produce new ones.

Still, even if QuickCheck provides some tools to help, the user does need to write the Haskell code specifying how to generate test cases for every custom data type that needs to be an instance of `Arbitrary`, which means making some decisions. For instance, the documentation explicitly states that the user is free to treat the notion of 'size' in any way: the generators can be more or less sensitive to the size value, or even ignore it completely.

2.1.2 Testing programs

Property-based testing is based on the idea of running tests only on particular aspects of a program, called *properties*, which need to be specified by the user. QuickCheck implements these by defining a data type `Result` (which contains, among other information, whether the test was successful or not) and encoding properties as a `Gen` of a collection of `Result`; that is, a property to be checked is nothing more than a way of generating *results* about the program. A limitation with properties is that they cannot be defined polymorphically, so the user needs to specify what type should be used for generating the test cases.

To distinguish the types that can be tested in this way, QuickCheck also provides a type class `Testable` which can be used to simplify the process of defining properties. The following instances of `Testable` are included in QuickCheck:

```
instance Testable Bool
instance (Arbitrary a, Show a, Testable prop) => Testable (a -> prop)
```

Finally, properties of a program can be tested throught the `quickCheck` function,

```
quickCheck :: Testable prop => prop -> IO ()
```

This function runs the tests a configurable number of times (100, by default) while gradually increasing the size of the generated values, and aggregates all the results to produce a report. Some of the information it can produce is

- Whether all the tests succeeded, and a counterexample in case one failed. The counterexamples are automatically shrunk so that the report only includes the smallest value found that violates the property.
- Whether the generated values represent a representative enough sample of the input space (as defined by the user).
- Whether a property with a precondition passes the tests *vacuously*, that is, because the precondition does not hold.

2.2 Template Haskell

Template Haskell (TH) [10] is an extension to the Haskell programming language that adds support for *compile-time meta-programming*. According to the abstract of the original paper describing TH,

> the purpose of the system is to support the algorithmic construction of programs at compile-time. The ability to generate code at compile time allows the programmer to implement such features as polytypic programs, macro-like expansion, user directed optimization (such as inlining), and the generation of supporting data structures and functions from existing data structures and functions.

In particular, the ability to automatically *inspect the internal representation* of Haskell programs—although only a small fraction of what Template Haskell has to offer—will be very useful for our purposes, since we can use it to extract information about Haskell data declarations within Haskell itself and therefore easily work with the declarations.

To be more precise, TH supports two main operations [13]: *(quasi-)quotations*, writen as (some variation of) `[| ... |]` and *splices*, written as `$( ... )`. It also provides a type class `Quote` to work with "quoted" programs, and a monad `Q` that is an instance of it.

2.2.1 Quotations

Quotations are the mechanism that TH provides to refer to Haskell code inside Haskell programs. For this purpose, TH defines a series of algebraic data types (ADTs) to represent syntactically-correct Haskell programs [14]. The result of applying any of the available *quotation brackets* to a well-written program is an object representing its abstract syntax tree (AST) using these ADTs.

More concretely, Template Haskell defines four quotation brackets, each serving a particular purpose:

- The *expression* bracket, written `[| ... |]` or `[e| ... |]`, receives a Haskell expression and returns an object of type `Quote m => m Exp`.
- The *declaration* bracket, written `[d| ... |]`, receives a series of top-level declaration and returns an object of type `Quote m => m [Dec]`.

- The *type* bracket, written `[t| ... |]`, receives a Haskell type and returns an object of type `Quote m => m Type`.
- The *pattern* bracket, written `[p| ... |]`, receives a pattern (as in *pattern-matching*) and returns an object of type `Quote m => m Pat`.

Before presenting the definitions of the algebraic data types used by Template Haskell, let us look at an example of how these quotation brackets can be used in actual Haskell code:

Example 2.2.1. Consider the following quoted expression, which introduces a local definition for the pattern `x`, of type `Integer`, to be used in another expression:

```
[| let x = 7 :: Integer in x + 5 |]
```

This is expanded *at compile time* to the following data value wrapped inside the `Q` monad:

```
LetE [ValD (VarP x_1) (NormalB (SigE (LitE (IntegerL 7))
                                     (ConT GHC.Num.Integer.Integer))) []]
     (InfixE (Just (VarE x_1)) (VarE GHC.Num.+) (Just (LitE (IntegerL 5))))
```

Note that this object contains all four categories: expressions, declarations, types and patterns; it may be helpful to revisit this example after the following subsection to compare how the complete ADT values relate to the original Haskell code. ♦

2.2.2 The internal representation

The full description of the ADTs representing the Haskell syntax is included in [14]; we present here a summary of the most important grammatical categories for our purposes, with some minor details—that are irrelevant to our implementation—omitted for simplicity.

`Name` Abstract data type representing names in the syntax tree; these include data type names, data constructor names, function names...

`TyVar` Abstract data type representing type variables; this is used to introduce new type variables to describe parametric polymorphism.

- The data constructor `PlainTV` expects a `Name` representing the name of the type variable.

`Type` Abstract data type representing types. The data constructors most relevant to our work are the following:

- `VarT`, expecting a `Name`, for a type variable. The difference with `TyVar` above is that this is used when a type is expected, while `TyVar` is used for introducing polymorphic parameters (notably in data declarations).
- `ConT`, expecting a `Name`, for a type constant. For example, this could be `Int`, `Char` or a user-defined type.
- `ListT`, for list types. This represents the standard Haskell type constructor `[]`, with kind `* -> *`.
- `TupleT`, expecting an integer value, for tuples. This family of constructors (one for each non-negative argument) corresponds to the family of standard Haskell type constructors `()`, `(,)`, `(,,)`, etc. For example, `TupleT 0` represents `()` (also known as the *unit* type), with kind `*`, while `TupleT 2` represents `(,)` (the constructor for tuples with two elements), with kind `* -> * -> *`.
- `AppT`, for type applications, expecting two types. This is used for building complex types from types that have a kind different from `*`. For example,
  - the type '`[Int]`' (explicitly, '`[] Int`') is represented in TH as '`AppT ListT (ConT Int)`'.
  - the type '`(Int, Bool)`' (explicitly, '`(,) Int Bool`') is represented in TH as '`AppT (AppT (TupleT 2) (ConT Int)) (ConT Bool)`'. As usual, in Haskell syntax this is left-associative, so that '`(,) Int Bool`' is really shorthand for '`((,) Int) Bool`'; of course, this must be encoded explicitly in the AST.
- `ArrowT` represents the *arrow* type constructor (for constructing functional types), with kind `* -> * -> *`.

`Con` Abstract data type representing a single data constructor. These support both classical Haskell syntax and GADT syntax.

- The data constructor `NormalC` describes the simplest—and most common—data constructor. It expects a `Name` for the constructor itself and a list of `Type` for its arguments. For example, the *cons* list constructor '`Cons a (List a)`' is encoded as '`NormalC Cons [VarT a, AppT (ConT List) (VarT a)]`'.
- The data constructor `GadtC` describes simple constructors using GADT syntax.

Note that Template Haskell's terminology does not strictly align with the usual terminology found in the literature. To illustrate the difference, consider the usual constructors for lists: *nil* and *cons*. In the case of *nil*, both uses of the term "constructor" align. In the case of *cons*, however, there is a slight inconsistency:

- In Template Haskell terminology, `Cons` is only the name, and the whole constructor is `Cons a (List a)`, encoded as

  ```
  NormalC Cons [VarT a, AppT (ConT List) (VarT a)]
  ```

  as previously described.
- In the usual terminology, the constructor itself is *cons*, and it expects two arguments of types `a` and `List a`.

Throughout this work, we will adhere to the usual terminology: by *constructor*, we will mean only the name (with its associated arity and argument types). To refer to constructors that are fully instantiated, we will exclusively use the term *constructions* (corresponding to the `Con` TH data type).

`Clause` Abstract data type to encode a single function rule. It has a single (homonymous) constructor that expects a list of `Pat` for pattern matching on the arguments, a `Body` and a list of `Dec` for 'where' clauses.

`Dec` Abstract data type representing declarations; these include function or value declarations, data type declarations, class declarations, instance declarations...

- The data constructor `ValD` encodes a value declaration. It expects a `Pat` for pattern matching, a `Body` and a list of `Dec` for 'where' clauses.
- The data constructor `FunD` encodes a function declaration. It expects a `Name` for the function and a list of `Clause` for the function rules.
- The data constructor `DataD` corresponds to a data type declaration, using the `data` Haskell keyword. Among others arguments, it expects a `Name` for the new type being defined, a list of `TyVar` for the polymorphic parameters to the type, and a list of `Con` for the constructors of the new type.

In this work, the main interest is in *data* declarations; to conclude this section, let us see a few examples in some detail (focusing only on the relevant arguments to the various constructors and leaving out the rest):

Example 2.2.2 (Custom lists)**.** Even though Template Haskell has built-in support for native Haskell lists, it can be illustrative to look at a declaration for user-defined ones; the standard way to define them would be '`data List a = Nil | Cons a (List a)`', which would be encoded in TH as

```
DataD List
      [PlainTV a]
      [NormalC Nil [],
       NormalC Cons [VarT a, AppT (ConT List) (VarT a)]]
```

The relevant arguments to `DataD` are:

- the name `List`,
- the type variable `a`,
- the two constructors: `Nil` (with no arguments) and `Cons` (which has one argument of type `a` and another of type `List a`).

♦

Example 2.2.3 (Wrapper for pairs)**.** To illustrate the use of tuples in Template Haskell, let us take a look at a data type `Pair` that is just a wrapper around standard Haskell pairs: '`data Pair a b = MkPair (a, b)`'; this would be encoded in TH as

```
DataD Pair
      [PlainTV a, PlainTV b]
      [NormalC MkPair [AppT (AppT (Tuple 2) (VarT a)) (VarT b)]]
```

The arguments to `DataD` are:

- the name `Pair`,
- the type variables `a` and `b`,
- the unique constructor `MkPair`, with a single argument of type `(a,b)`.

♦

Example 2.2.4 (Rose trees)**.** Rose trees are a more interesting example than the ones we have seen before. They are trees with a variable number of children, and can be described in Haskell as '`data RoseTree a = Rose a [RoseTree a]`': each node of the tree stores a value (of type `a`) and has a list of children (each of them being another rose tree). This declaration is parsed by Template Haskell as

```
DataD RoseTree
      [PlainTV a]
      [NormalC Rose [VarT a,
                     AppT ListT (AppT (ConT RoseTree) (VarT a))]]
```

The arguments to `DataD` are now:

- the name `RoseTree`,
- the type variable `a`,
- the unique constructor `Rose`, with one argument of type `a` and another of type `[RoseTree a]`.

♦

In the following chapters, the discussion will often revolve around grammars based on this one, but modified to fit our needs: we will work only with the subset of grammatical categories that are required at each stage, and we will often omit the arguments to the TH constructors that are not needed for our purposes. Any major difference between the grammars we define and Template Haskell's grammar will be stated in the text; the interested reader can find the full documentation for Template Haskell's internal representation in the bibliography.

2.2.3 Splices

Finally, Template Haskell provides the mechanism of *splices* to interpolate the internal representation presented in 2.2.2 to actual Haskell code. Splices are introduced by preprending a Haskell expression with the `$` character, without any space between them.

Splices act as the inverse of quotation brackets. In particular, splices can occur in place of an expression, a pattern, a type or a list of declarations; the spliced expression should then be of type `Q Exp`, `Q Pat`, `Q Type` or `Q [Dec]`, respectively.

2.3 Prolog

Prolog (from *Programmation en Logique*) is a general purpose logic programming language. This section is intended as a brief introduction to the language (both its syntax and its semantics), as well as a description of the logic formalism it is based on. In particular, we will focus mostly on those aspects that have been used in our work and are needed for the exposition; the interested reader can find all the details in [15, 16].

2.3.1 Horn clauses

Logic programming languages in general, and Prolog in particular, are based on a subset of first-order predicate logic known as *Horn clauses*. These are a very restricted class of logic formulas that nevertheless are expressive enough to encode many interesting computations in a natural way, making them very appealing for their use in programming. More precisely, Horn clauses are defined formally as follows:

Definition 2.3.1 (Horn clause)**.**

1. A *literal* is either an atomic predicate, or the negation of one. We say that a literal is *positive* if it is unnegated, and *negative* otherwise.
2. A *disjunctive clause* is a (finite) disjunction of literals.
3. A *Horn clause* is a disjunctive clause where at most one literal is positive.

▲

Observe that Horn clauses can be naturally split into four categories:

- No positive literals
  - No negative literals. This is the *empty* clause, whose truth value is always *false*.
  - Some negative literals. These are known as *goal* clauses, because of their use in computer-assisted theorem proving.
- Exactly one positive literal
  - No negative literals. These are known as *facts*.
  - Some negative literals. These are known as *definite* or *strict* Horn clauses, and they are generally the most useful in logic programming.

To motivate the use of definite clauses in logic programming, let us consider some positive literals $l, l_1, l_2, \ldots, l_r$. Then, a definite Horn clause using these literals could be

$$\neg l_1 \vee l_2 \vee \cdots \vee \neg l_r \vee l.$$

By de Morgan's laws, the previous expression is logically equivalent to

$$\neg(l_1 \wedge l_2 \wedge \cdots \wedge l_r) \vee l,$$

which, using the general fact that $p \to q$ is equivalent to $\neg p \vee q$, can be rewritten as

$$(l_1 \wedge l_2 \wedge \cdots \wedge l_r) \to l.$$

In logic programming, this implication is usually written in the reverse order, namely

$$l \leftarrow (l_1 \wedge l_2 \wedge \cdots \wedge l_r).$$

This should be understood as "a sufficient condition for l to hold is that $l_1, l_2, \ldots, l_r$ all hold simultaneously". This way, *sets* of Horn clauses (or, alternatively, *sequences* of Horn clauses) can closely model a *rule-based system*. This is the essence of logic programming.

A *set* of Horn clauses should be understood as a *disjunction* of its elements, in the sense that any clause (rule) can be applied; for this reason, given Horn clauses $c_1, \ldots, c_n$, we will write $\{c_1, \ldots, c_n\}$ and $c_1 \vee \cdots \vee c_n$ interchangeably. A *sequence*, in contrast, is the non-commutative analogue: the order in which clauses appear is important, because only *the first* applicable clause can be used. We will write $c_1 \veebar \cdots \veebar c_n$ for a non-commutative disjunction.

To be more precise, we will understand a *logic program*, in the abstract sense, to be either a set or a sequence (depending on our needs) of Horn clauses with precisely one positive literal, that is, either *facts* or *definite clauses*:

$$\begin{aligned} \langle \mathit{clause} \rangle \ ::= \ & \langle \mathit{predicate} \rangle \\ \mid \ & \langle \mathit{predicate} \rangle \leftarrow \langle \mathit{predicate} \rangle_1 \wedge \ldots \wedge \langle \mathit{predicate} \rangle_n \quad (n \geq 1) \end{aligned}$$

Figure 2.1: Horn clause syntax

We will define $\mathcal{H}$ to be the set of all Horn clauses, and we will work with sets of clauses (elements of $\mathcal{P}(\mathcal{H})$) or with sequences of clauses (elements of $\mathcal{H}^*$) depending on the requirements of the problem.

2.3.2 Prolog syntax

A Prolog program is modeled as a (possibly empty) list of Horn clauses followed by a *query*; the list of clauses is usually written in a file which is then loaded into the interpreter, whereas the query is usually entered directly into the interpreter to get an answer (see § 2.3.3 for more details).

The full Prolog syntax is presented in figure 2.2 below [17].

⟨*program*⟩ ::= ⟨*clause list*⟩ ⟨*query*⟩ | ⟨*query*⟩

⟨*clause list*⟩ ::= ⟨*clause*⟩ | ⟨*clause list*⟩ ⟨*clause*⟩

⟨*clause*⟩ ::= ⟨*predicate*⟩ `.` | ⟨*predicate*⟩ `:-` ⟨*predicate list*⟩ `.`

⟨*predicate list*⟩ ::= ⟨*predicate*⟩ | ⟨*predicate list*⟩ `,` ⟨*predicate*⟩

⟨*predicate*⟩ ::= ⟨*atom*⟩ | ⟨*atom*⟩ `(` ⟨*term list*⟩ `)`

⟨*term list*⟩ ::= ⟨*term*⟩ | ⟨*term list*⟩ `,` ⟨*term*⟩

⟨*term*⟩ ::= ⟨*numeral*⟩ | ⟨*atom*⟩ | ⟨*variable*⟩ | ⟨*structure*⟩

⟨*structure*⟩ ::= ⟨*atom*⟩ `(` ⟨*term list*⟩ `)`

⟨*query*⟩ ::= `?-` ⟨*predicate list*⟩ `.`

⟨*atom*⟩ ::= ⟨*small atom*⟩ | `'` ⟨*string*⟩ `'`

⟨*small atom*⟩ ::= ⟨*lowercase letter*⟩ | ⟨*small atom*⟩ ⟨*character*⟩

⟨*variable*⟩ ::= ⟨*uppercase letter*⟩ | ⟨*variable*⟩ ⟨*character*⟩

⟨*lowercase letter*⟩ ::= `a` | `b` | `c` | `d` | ⋯ | `x` | `y` | `z`

⟨*uppercase letter*⟩ ::= `A` | `B` | `C` | `D` | ⋯ | `X` | `Y` | `Z` | `_`

⟨*numeral*⟩ ::= ⟨*digit*⟩ | ⟨*numeral*⟩ ⟨*digit*⟩

⟨*digit*⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

⟨*character*⟩ ::= ⟨*lowercase letter*⟩ | ⟨*uppercase letter*⟩ | ⟨*digit*⟩ | ⟨*special*⟩

⟨*special*⟩ ::= `+` | `-` | `*` | `/` | `\` | `^` | `~` | `:` | `.` | `?` | `@` | `#` | `$` | `&`

Figure 2.2: Prolog syntax

There are a few important remarks that must be made about this syntax:

Remark 2.3.1. There are only two rules for *clauses*, corresponding to *facts* and to *definite rules* (where the characters `:-` should be interpreted as logical implication, ←). ■

Remark 2.3.2. *Variables* and *small atoms* are constructed in almost the same way, the only difference being that variables always start with an uppercase letter (or an underscore, `_`) while small atoms always start with a lowercase letter. ■

Prolog clauses are made up of *predicates*, which can be of two types: simple atoms (which should be thought of as logical propositions) and structures, formed by an atom (called the *functor* or *function symbol*) and a list of terms (the *arguments*). The terms can be of one of four types: numbers, atoms, variables or other structures.

Example 2.3.1. Let us illustrate the syntax definition with an example of a Prolog program:

```
1   founder('José Arcadio Buendía').
2   founder('Úrsula Iguarán').
3
4   mother('José Arcadio', 'Úrsula Iguarán').
5   father('José Arcadio', 'José Arcadio Buendía').
6
7   mother('Aureliano', 'Úrsula Iguarán').
8   father('Aureliano', 'José Arcadio Buendía').
9
10  mother('Arcadio', 'Pilar').
11  father('Arcadio', 'José Arcadio').
12
13  mother('Aureliano José', 'Pilar').
14  father('Aureliano José', 'Aureliano').
15
16  parent(X, Y) :- mother(X, Y).
17  parent(X, Y) :- father(X, Y).
18
19  ancestor(X, Z) :- parent(X, Z).
20  ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).
```

Note that this program contains facts (e.g. `father('Arcadio', 'José Arcadio').`) and definite rules (e.g. `parent(X, Y) :- mother(X, Y).`).

In this setting, an example of a query could be `mother('Aureliano', M).` (which, intuitively, should return 'true' if the variable `M` is bound to the mother of Aureliano). ♦

Finally, we introduce a standard definition and notation that is useful when discussing Prolog code:

Definition 2.3.2 (Arity)**.** The number of arguments in a predicate is called its *arity*; more concretely, if the predicate is constructed as a structure then this number is the number

of arguments of the structure, and if it is a plain atom, then the number of arguments is taken to be 0.

To concisely refer to some predicate where we do not wish to mention the arguments explicitly, we use the notation `<atom>/<arity>`, where `atom` is either the functor of the structure or the atom that makes up the predicate. ▲

Example 2.3.2. The code shown in example 2.3.1 mentions, among others, the predicates `founder/1` and `parent/2`. ♦

2.3.3 Prolog semantics

In this subsection, we present an informal description of Prolog's semantics, which are heavily based on the underlying logic.

The Prolog semantics clearly distinguish between predicates and queries. To be more precise, the list of predicates *defines* a relation between the various atoms and variables that appear in the code, while queries ask the interpreter to *answer* some question using the predicate definitions.

Example 2.3.3. In the example 2.3.1, the facts state relationships between the various atoms (e.g. `'Arcadio'` and `'Pilar'` are related through the `mother/2` predicate). The definite rules, on the other hand, are used here to define predicates not intensionally, but extensionally through some other predicates. For example, lines 16 and 17 state that, *if* `X` and `Y` are bound to any two terms related through the `mother/2` predicate, *then* they are also related through the `parent/2` predicate. ♦

As evidenced by this example, predicates are allowed to be defined recursively, and the scope of a variable is the clause it is defined in.

The interaction with the program is done through *queries*. When the user inputs a query (which is nothing more than a list of predicates) into the prompt, the interpreter tries to determine if said predicate holds using exactly the information present in the defined predicates. If there are any free variables in the query predicates, they are assumed to be *existentially qualified*, meaning that the interpreter will bind them to any value that makes the predicate hold, if possible.

Example 2.3.4 (Checking information)**.** Going back to the previous example, these are some queries we might ask the Prolog interpreter:

- `?- mother('Arcadio', 'Pilar').` Since this can be deduced from the information

given in the predicate definitions (it is given explicitly as a fact), the interpreter returns `true.`

- `?- ancestor('Aureliano José', 'José Arcadio Buendía').` It is easy to see intuitively that `'José Arcadio Buendía'` is the grandfather of `'Aureliano José'`, so this query should return `true .` as well.

  To check this automatically, the interpreter uses *backtracking*: it first tries to use the first rule for `ancestor/2`, on line 19, which states that a sufficient condition for being an ancestor is being a parent. Since this does not hold for this particular case, the interpreter then tries to use the next available rule, on line 20. It then tries to check if each predicate in the clause holds: to do this, it starts by binding the variable `Y` to some term that is related to `'Aureliano José'` through the predicate `parent/2`; again, this is done by backtracking, first trying (and eventually failing) with `Y = 'Pilar'` and finally succeeding with `Y = 'Aureliano'`. The interpreter then returns the value `true`, as expected:

  ```
  ?- ancestor('Aureliano José', 'José Arcadio Buendía').
  true.
  ```

- On the other hand, if after all this backtracking the interpreter finds that the provided query does *not* hold, it returns `false.`:

  ```
  ?- ancestor('Aureliano José', 'Arcadio').
  false.
  ```

  Note that the value `false` is *only* given once all possibilities have been ruled out.

♦

One of the most useful features of Prolog is that, if a query with any free variables is satisfiable, the interpreter returns values for the variable(s) that satisfy the predicate:

Example 2.3.5 (Retrieving information)**.** Revisiting once again the previous example,

- To find out Aureliano's mother, we can give the following query:

  ```
  ?- mother('Aureliano', M).
  M = 'Úrsula Iguarán'.
  ```

  As expected, the interpreter replies that there is a valid solution and gives the value that the variable `M` needs to have so that the predicate is satisfied.

- Suppose we want to learn all the descendants of `'José Arcadio Buendía'`; we could ask Prolog in the same way:

```
?- ancestor(X, 'José Arcadio Buendía').
X = 'José Arcadio'
```

However, in this case the interpreter does not immediately return to the prompt; this indicates that there is more than one solution, which we can request by interactively using the *semicolon* `;`, like so:

```
?- ancestor(X, 'José Arcadio Buendía').
X = 'José Arcadio' ;
X = 'Aureliano' ;
X = 'Arcadio' ;
X = 'Aureliano José' ;
false.
```

The final `false.` means that, when asking Prolog for additional solutions, it was not able to find any.

♦

The cut operator

This behaviour is possible because, after one rule is successfully applied, Prolog will try the next ones if the user keeps asking for more solutions. In some situations this may be desirable, but in some others it can give superfluous information. For example, in a query like `?- ancestor('Aureliano José', 'José Arcadio Buendía').` we get a result of `true`, but the interpreter indicates that there may be other results (there can be multiple correct choices for the internal variable `Y`, and there is more than one rule than can be applied to conclude something like `ancestor('Aureliano José', 'José Arcadio Buendía').`, namely the ones in lines 19 and 20 in example 2.3.1). However, when asking for more solutions, the interpreter returns the result `false`, since there is only one way in which `'José Arcadio Buendía'` is an ancestor of `'Aureliano José'`.

To avoid this kind of situations, Prolog provides the *cut* operator, implemented as a predicate of arity zero, `!/0`. Whenever it is encountered in a rule, it signals the interpreter not to backtrack beyond this point; therefore, it can be used to avoid getting multiple answers when we only care about the first one, or to stop the interpreter from trying to backtrack when we know there is only one solution.

In particular, placing the cut as the final predicate in a definite clause, we can model *non-associative* disjunction: the first clause that can be applied successfully is used, and the rules that are written after are not tried by the interpreter, even when the user explicitly asks for more solutions. This implies that the order of clauses matters when using the cut operator.

Wildcard and singleton variables

Recall from figure 2.2 that variables can start with an underscore, '_'. Variables starting with _ are interpreted in a special way, in that they are treated as "one-time" variables:

- The variable '_' (the *wildcard* variable) is special because it is the only variable that can be bound to a different value each time it appears in a program. This is often useful in practice, for example for unifying a term where some of its arguments are not used elsewhere. While it is possible to use a fresh variable for each such argument, using the wildcard variable conveys the programmer's intentions more clearly.
- Every other variable starting with an underscore is used to denote a *singleton* variable, that is, a variable that is meant to be used only once in a program. This is useful in practice for the same reason as the wildcard while including a variable name for documentation purposes.

  In contrast to the wildcard variable, singleton variables *can* be used several times with the same value within the same scope; however, the Prolog interpreter checks whether this is the case and issues a warning if it detects a variable marked as 'singleton' used more than once within the same scope.

2.3.4 Some predefined predicates

To conclude this section, we present here some built-in predicates that will be useful in the following chapters.

Random choices

Predicates for choosing values at random will be vital for automatic test case generation; more specifically, we will need the following two predicates:

- `random_between/3` is true if and only if the third argument is equal to a value chosen uniformly at random between the first two (greater than or equal to the first and

less than or equal to the second). This is most useful when the third argument is a variable, since then the variable is bound to a random value in the desired range:

```
?- random_between(1, 10, X).
X = 9.
```

Although significantly less useful, it can also be used as a predicate that holds with some probability; for example, `random_between(1, 3, 2)` is true one out of three times (when the randomly chosen value is equal to 2).

- `random_member/2` is true if and only if the first argument is a value chosen at random from the list given as a second argument. For example, to choose a day of the week uniformly at random, we could use the following query:

```
?- random_member(Day, [mon, tue, wed, thu, fri, sat, sun]).
Day = sat.
```

Structure inspection

One predicate that will be very useful in the following chapters is `=../2`, which can be used to either *inspect* or *generate* structures programatically (recall from figure 2.2 that a structure is formed by a "base" *atom* applied to a list of *terms*). The first argument to `=../2` is the structure and the second one is a list containing at its head the base atom and at its tail the list of terms.

Although it can be written as any other predicate, it can also be written using infix notation, which leads to clearer expressions:

```
?- ancestor('Aureliano José', 'José Arcadio Buendía') =.. [Atom | Terms].
Atom = ancestor,
Terms = ['Aureliano José', 'José Arcadio Buendía'].

?- T =.. [f | [x, g(y)]].
T = f(x, g(y)).
```

Control predicates

There are some special predicates that influence the interpreter's behaviour, so they can be classified as "control-flow" predicates. The most relevant are:

- The *cut* predicate, `!`, which was already described in § 2.3.3.
- The `repeat` predicate, which can be understood as a "try again" primitive: it provides an infinite amount of *choice points*, so if a predicate that is written after it fails, it can be given (infinitely) more chances.

  It is generally used in conjunction with the cut operator as a guard against a possibly failing predicate: it is given an infinite number of chances, but the first that is successful is "commited to" with the cut:

  Example 2.3.6 (*Try again*)**.** In the following query, a random element is sampled from the list `[1,2,3]` and the result is unified with the value `1`:

  ```
  ?- repeat, random_member(X, [1,2,3]), X = 1, !.
  X = 1.
  ```

  The predicate `random_member(X, [1,2,3]), X = 1` succeeds with a probability of 1/3, and fails with a probability of 2/3. By including the `repeat`, we make sure that it is given enough chances to succeed, and the final `!` ensures that when it first succeeds, the exploration halts by stopping the backtracking. Without the cut, we would get an infinite loop, with the variable `X` always being unified with `1`:

  ```
  ?- repeat, random_member(X, [1,2,3]), X = 1.
  X = 1 ;
  X = 1 ;
  X = 1 ;
  X = 1 ;
  X = 1 .
  ```

  ♦

- The `;` predicate, which combines two predicates to form their logical disjunction.

Aggregation predicates

Aggregation predicates are sometimes useful when a query has many answers, since they can group all of them together producing one single list. These predicates—`findall/3`, `setof/3` and `bagof/3`—have subtle differences between them, but they are mostly used in the same way:

Example 2.3.7. Going back to example 2.3.5 above, when querying the descendants of `'José Arcadio Buendía'`, instead of asking Prolog for the answers one by one, we

might wish to populate a list containing all of them, that is, all values `X` such that `ancestor(X, 'José Arcadio Buendía')` holds. This can be done like so:

```
?- findall(X, ancestor(X, 'José Arcadio Buendía'), Results).
Results = ['José Arcadio', 'Aureliano', 'Arcadio', 'Aureliano José'].
```

♦

Type-related predicates

Prolog includes some predicates for working with values of a specific type. The ones we will need in the following chapters are presented below:

- Predicate `char_code/2` relates a single unicode character (an atom of length 1) to its character code (the integer representing it):

  ```
  ?- char_code(Char, 65).
  Char = 'A'.
  ```

- Predicate `atom_string/3` performs a conversion between an atom and the string containing its "name":

  ```
  ?- atom_string(ancestor, Str).
  Str = "ancestor".

  ?- atom_string(Atom, "This is a string").
  Atom = 'This is a string'.
  ```

- Predicate `maplist/3` is useful for working with lists: it applies the predicate given as the first argument to the values in the lists given in the remaining arguments, instantiating any variable if possible:

  ```
  ?- maplist(char_code, [X,Y,Z], [70,71,72]).
  X = 'F',
  Y = 'G',
  Z = 'H'.

  ?- maplist(mother, ['Aureliano','Aureliano José'], [M1,M2]).
  M1 = 'Úrsula Iguarán',
  M2 = 'Pilar'.
  ```

  When used in this way, it is similar to Haskell's `zipwith` function.

Chapter 3

Producing test case generators from data declarations

The main goal of this project is to automatically create test case generators for non-trivial data types. To achieve this, there are two fundamental components:

(a) A way to represent the data type declarations, along with their desired (static) invariants.

(b) A way to produce some *program* to carry out the type-based test case generation.

3.1 Representation of data declarations

The first step, then, will be to encode the Haskell data declarations in a way that facilitates working with them to produce the generators. *Template Haskell* (see § 2.2) has been used for this purpose, since it offers mechanisms to automatically produce Haskell data values representing some piece of actual Haskell code. In particular, top-level declarations can be placed inside *declaration brackets* `[d| ... |]` to produce a list of 'declaration' objects inside a quotation monad, that is, an object of type '`Quote m => m [Dec]`' (see § 2.2.1 for more information on TH quotation brackets).

These objects can then be processed just as any other Haskell object; this makes it easy to implement *syntax-directed* functions for the translation of the objects constructed by Template Haskell into the code that will carry out the test case generation.

The complete grammar that is supported by our translation mechanism at this stage

is presented schematically in figure 3.1 below.

$$
\begin{aligned}
\langle Dec \rangle &::= \texttt{DataD}\ \langle Name \rangle\ \langle TyVar \rangle^*\ \langle Con \rangle^+ \\
\langle Con \rangle &::= \texttt{NormalC}\ \langle Name \rangle\ \langle Type \rangle^* \\
\langle Type \rangle &::= \texttt{VarT}\ \langle Name \rangle \\
&\quad\ |\ \ \texttt{ConT}\ \langle Name \rangle \\
&\quad\ |\ \ \texttt{AppT}\ \langle Type \rangle\ \langle Type \rangle \\
&\quad\ |\ \ \texttt{ListT} \\
&\quad\ |\ \ \texttt{TupleT}\ \langle Int \rangle \\
\langle TyVar \rangle &::= \texttt{PlainTV}\ \langle Name \rangle
\end{aligned}
$$

Figure 3.1: Supported syntax in the first stage

Before we proceed, some comments must be made about this grammar:

Remark 3.1.1. Note that this grammar is heavily inspired by the internal representation defined by Template Haskell, but it only needs a small subset of that (cf. § 2.2.2 and [14]). In particular, note that

(a) the only supported declarations are *data declarations*,
(b) *function types* (i.e. those constructed with the *arrow* operator) are unsupported,
(c) only the usual Haskell syntax is supported for constructors (and not GADT syntax).

■

Remark 3.1.2. As mentioned in § 2.2.2, some of these constructors (most notably, `DataD`) expect some additional arguments that are omitted here for simplicity, since they are irrelevant to our work; the interested reader can find the documentation for the full grammar defined by TH in [14]. ■

Notation 3.1.1. Where Template Haskell defines the type of some argument to be a list, we have chosen to write a *Kleene star* instead, as is customary in the literature. Still, in the presentation below we will enclose the objects of these types between square brackets and will separate the elements inside with commas (resembling standard Haskell syntax for lists) for better readability. ■

Remark 3.1.3. The syntax described in figure 3.1 makes repeated use of the $Name$ grammatical category (following Template Haskell's internal representation) but it does not include a precise definition for it. The reason for this is that it distracts from the more important aspects of the grammar and adds little value, as the names are only relevant to us as a label for the various identifiers.

In the following discussion (as well as in the actual Haskell code that performs the translation), the only use for names is extracting a string representation of the identifier. Because of this, *Name*'s are treated directly like strings in the rest of the text.

By extension, type variables are also implicitly used as strings, given that (for our purposes) they are nothing but a wrapper around a *Name*. ■

Notation 3.1.2. The syntax described above can be awkward to work with, especially in the case of types and constructions, because it is very different from the way we think about these concepts. Because of this, it can be useful to introduce a more intuitive notation to refer to them—without changing the previous definition, since it is desirable to stay as close to Template Haskell's grammar as possible.

This grammar is based on *currying*, as is the case in most modern functional programming languages. Take as an example the type of triples of integers, characters and boolean values. While intuitively we could write this as `(Tuple 3)(Int, Char, Bool)`, in Template Haskell's grammar (and in ours, by extension), this is defined by successive partial applications of type constructors:

```
AppT (AppT (AppT (Tuple 3) (ConT Int)) (ConT Char)) (ConT Bool).
```

or, synthetically, `(((Tuple 3) (ConT Int)) (ConT Char)) (ConT Bool)`, by writing `AppT` implicitly as juxtaposition. Our goal is to bridge the gap between these two notations; we do this in two steps:

1. we omit the parentheses with the assumption that `AppT` is left-associative, and
2. we take the first type in such a chain to be the *base* type and the rest to be its *arguments*.

In the previous example, the base type would be `Tuple 3` and its arguments would be `ConT Int`, `ConT Char` and `ContT Bool`. Finally, the distinction between type constants and type variables is not critical for our exposition; we can simply use the name to refer to either construct. With this, our example can be written with an intuitive notation in a straightforward way.

Crucially, this conversion is easy to do automatically, and a Haskell function implementing this it included in our code. Therefore, from now on, we will use this simpler notation freely.

Similarly, in the case of constructions, we can omit the `NormalC` and write only the

name of the constructor followed by its list of arguments inside parentheses, as usual. This is merely a cosmetic change, so we will also use this new notation freely in the following. ■

To conclude this section, let us take a look at how this syntax is used to encode a few examples of declarations:

Example 3.1.1 (Custom lists)**.** The usual way to define lists in a functional language is '`data List a = Nil | Cons a (List a)`', which is encoded in our syntax as

```
DataD List
      [a]
      [Nil,
       Cons(a, List(a))]
```

♦

Example 3.1.2 (Rose trees)**.** Rose trees can be defined in standard Haskell syntax as '`data RoseTree a = Rose a [RoseTree a]`', which is encoded in our syntax as

```
DataD RoseTree
      [a]
      [Rose(a, ListT(RoseTree(a)))]
```

♦

3.2 Translation

3.2.1 First approach

For the purpose of actually generating the data values, Prolog has been chosen for two main reasons: because of the ease of specifying syntax-oriented rules within it, and because of its flexibility for using specialised search strategies, which is particularly useful for generating data values where the generation is not completely "free"—consider, for instance, a data structure like *red-black trees*, where the colors in the nodes need to satisfy some concrete conditions for the structure as a whole to be correct. The translation functions, then, transform a Template Haskell object representing a data declaration into a Prolog program describing exactly how to generate randomised values of these types.

In particular, a data declaration stating that a value of a (possibly parametric) type can be created with one or more constructors is translated into one or more *rules*, one for each construction. Each of these rules says that, for generating a value of the specified type, one possible way to do it is to use that construction. This is done via the `gen/2` Prolog predicate; more concretely, `gen(Type, Val)` means that `Val` is a value of type `Type`. For example, all of the following are possible values that could be generated:

- `gen(bool, false).`
- `gen(list(int), nil).`
- `gen(list(int), cons(1, cons(2, cons(3, nil)))).`
- `gen(pair(int, bool), mkpair(7, true)).`

Note that all the second arguments to predicate `gen/2` are instances of a construction for the type that appears in the first argument (e.g. `false` is one of the constructions for `bool`, `nil` is one of the constructions for `list(int)`, etc.).

For 'base' types (those predefined in Haskell like `Int`, `Bool`, `Char` and `String`), as well as lists and tuples, the necessary rules for predicate `gen/2` are provided by default. These are the following:

- For `Int`, the Prolog predicate `random_between/3` is used to generate a random integer between the bounds of Haskell integers (as given by `minBound :: Int` and `maxBound :: Int`). Then, the resulting number `X` is one possible way to construct an integer; the corresponding Prolog code is

```
gen(int, X) :- random_between(-9223372036854775808,
                              9223372036854775807,
                              X).
```

- For `Char`, a similar approach is taken to produce a *character code*, and then this code is used to generate the character itself with the Prolog predicate `char_code/2`. The resulting character `C` is returned by an auxiliary predicate `gen_char/1`, and this predicate is used to say that `C` is one possible way to construct a `Char`; the corresponding Prolog code is

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

- For `String`, a random length `N` is chosen between 0 and 99, then a list `L` of `N` characters is generated using the auxiliary predicate `gen_char/1` and the default Prolog predicate `maplist/2`, and finally a string `S` is created from `L` with the default predicate

`atom_string/2`. The resulting string is a way to construct a `String`, as described by this Prolog code:

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

- For `Bool`, two rules are provided directly:

```
gen(bool, false).
gen(bool, true).
```

- For lists, we follow the usual approach with *nil* and *cons* constructors since it has native support in Prolog. The rules say that `[]` is of type `listt(A)` (for any `A`) and, for creating `[X|XS]` as a value of type `listt(A)` one should require that (a) `X` is of type `A`, and (b) `XS` is of type `listt(A)`. Formally,

$$\begin{aligned} &\texttt{gen}(\texttt{listt}(\texttt{A}), \texttt{[]}) \vee \\ &\texttt{gen}(\texttt{listt}(\texttt{A}), \texttt{[X|XS]}) \leftarrow \texttt{gen}(\texttt{A}, \texttt{X}) \wedge \texttt{gen}(\texttt{listt}(\texttt{A}), \texttt{XS}) \end{aligned} \tag{3.1}$$

  and, as a Prolog program,

```
gen(list__(A), []).
gen(list__(A), [X|XS]) :- gen(A, X), gen(list__(A), XS).
```

  This will be used as the translation of the standard Haskell lists (i.e. those encoded in Template Haskell as the type `ListT`). The type is named `list__` in the code to avoid name clashes with other types that could be defined by the user.

- For tuples, a similar approach is taken; the main difference is that a tuple contains many types, so the 'type constructor' `tuple` receives a list of types, instead of a single one. In the Prolog code, the actual tuple of values is modelled using a list. Three rules are provided for constructing tuples:

  1. For a tuple of only one element, the value is generated directly.
  2. For a tuple of more than one element, the first element is generated directly and the tail is generated recursively.
  3. An exceptional rule is provided for the tuple with no elements, `()`, usually known as the *unit* type; this type only contains one element (also called unit and written `()`). In the Prolog code, an ad-hoc data constructor `unit` is given for it.

Formally, the rules are

$$(\texttt{gen}(\texttt{tuplet}(\texttt{[T]}), \texttt{[X]}) \leftarrow \texttt{gen}(\texttt{T}, \texttt{X})) \vee$$
$$(\texttt{gen}(\texttt{tuplet}(\texttt{[T|TS]}), \texttt{[X|XS]}) \leftarrow \texttt{gen}(\texttt{T}, \texttt{X}) \wedge \texttt{gen}(\texttt{tuplet}(\texttt{TS}), \texttt{XS})) \vee$$
$$(\texttt{gen}(\texttt{tuplet}(\texttt{[]}), \texttt{unit}))$$

which are translated into Prolog code as follows:

```
gen(tuple__([T]),    [X])    :- gen(T, X).
gen(tuple__([T|TS]), [X|XS]) :- gen(T, X), gen(tuple__(TS), XS).
gen(tuple__([]),     unit).
```

Note that, in the case of constructors that expect one or more parameters (for example, the *cons* list constructor), the predicate `gen/2` is used again to recursively generate the values to be passed to the constructor.

Recall from § 2.3.2 that, in Prolog, variables must start with an upper-case letter and all other identifiers (constants and predicate names) must start with a lower-case letter. To generate Prolog code more efficiently, we introduce two functions, `mkPrologVar` and `mkPrologConst`, to create a string representing a Prolog variable or constant, respectively, given a Template Haskell *Name*.

Recall also from remark 3.1.3 that, to simplify the presentation, we do not distinguish between Template Haskell *Name*'s and the underlying string representing the identifier. Likewise, the use of the conversion functions `mkPrologVar` and `mkPrologConst` is left implicit in the rest of the text, since we will make frequent use of both. To be more precise, given a *Name* that represents a Prolog constant (for example, the name of a type being defined), its translation into Prolog will be written using the same name, changed if necessary to begin with a lower-case letter, and in typewriter font; for example,

$$\texttt{mkPrologConst}(typeName) = \texttt{typeName}.$$

Similarly, the translation of a *Name* that represents a Prolog variable (for example, a constructor parameter that needs to be instantiated with some value) will be written using the same name, starting with an upper-case letter, and in typewriter font; for example,

$$\texttt{mkPrologVar}(ctorParam) = \texttt{CtorParam}.$$

By using this convention, there is no ambiguity because identifiers always start with a letter and its case is enough to classify it as a variable or a constant.

Types, however, are not so simple, as they may include both type variables and type constants; as an example, it could be natural in Haskell code to use a type like `Either a String` to represent the result of some computation that could fail, in which case an error message is returned. In this example, a way to encode this as a Prolog term may be `either(A, string)`, including again a variable and a constant. This conversion can be easily done automatically but it distracts from the exposition without adding much value; therefore, we will abbreviate this conversion from Haskell to Prolog code with curly braces, like so:

$$\{Either\ a\ String\} = \texttt{either(A, string)}.$$

With this, we are ready to describe the translation procedure from Haskell data declarations to the Prolog program that generates the test cases. As before, for clarity this is presented using a simplified notation for Template Haskell objects and Horn clauses instead of actual Prolog code (while maintaining the above convention for differentiating between Prolog constants and variables). Abstracting away from these details, the translation rules are presented in figure 3.2 below; the complete Haskell program that performs the transformation from the Template Haskell representation to actual Prolog code can be found in appendix A.2.1 and in the public repository associated to the project [18], and is released as free software under the GNU General Public License (version 3).

$$\begin{aligned}
&\text{parseDec} : \texttt{Dec} \to \mathcal{P}(\mathcal{H}) \\
&\text{parseDec } [\![\texttt{DataD } typeName\ typeVars\ [con_1, \ldots, con_n]]\!] = \\
&\quad = \bigvee_{i=1}^{n} \text{rule } [\![typeName\ typeVars\ con_i]\!] \\
\\
&\text{rule} : Name \times TyVar^* \times Con \to \mathcal{H} \\
&\text{rule } [\![typeName\ typeVars\ ctorName(ctorParam_1, \ldots, ctorParam_r)]\!] = \\
&\quad = \texttt{gen(typeName(TypeVars), ctorName(FreshVar}_1, \ldots, \texttt{FreshVar}_r\texttt{))} \\
&\qquad \leftarrow \bigwedge_{i=1}^{r} \texttt{gen(\{}ctorParam_i\texttt{\}, FreshVar}_i\texttt{)} \\
&\quad \text{where } [\texttt{FreshVar}_1, \ldots, \texttt{FreshVar}_r] = freshVars(ctorName, r)
\end{aligned}$$

Figure 3.2: Translation rules for data type generation

The "parseDec" translation rule works by adding a new Horn clause for every constructor, as we mentioned previously. On the other hand, "rule" specifies that the given data constructor can be used for creating values of the given type when it is completetely instantiated with values of the appropiate types. To be more specific, it does this by introducing

one fresh variable for each expected parameter and binding each new variable to a value of the appropiate type by recursively using the `gen/2` predicate.

These fresh variables are all named after the constructor itself, but they are numbered starting from 1 up until the number of expected parameters to the constructor. As an example, the *cons* constructor for lists expects two arguments (of types `a` and `List(a)`), so "rule" introduces two fresh variables for instantiating them: $\texttt{Cons}_1$ (to be instantiated with a value of type `a`) and $\texttt{Cons}_2$ (to be instantiated with a value of type `List(a)`). To generate the fresh variables, we introduce a function $freshVars$ that receives the base name and the number of needed variables to be generated; for example,

$$freshVars(Cons, 2) = [\texttt{Cons}_1, \texttt{Cons}_2].$$

Examples

To better understand how these translation rules work, it is useful to see them applied to a few examples of real data structures:

Example 3.2.1 (Custom lists)**.** Recall from example 3.1.1 that lists have one type variable `a` and two constructors, `Nil` (with 0 arguments) and `Cons` (with 2 arguments, one `a` and one `List(a)`). The translation rule "parseDec" says that we need to use "rule" for each constructor:

1. For the constructor `Nil`, "rule" works as follows:

$$\text{rule } [\![\texttt{List [a] Nil}]\!] = \texttt{gen}(\texttt{list}(\texttt{A}), \texttt{nil})$$

   This is precisely saying that `nil` is a value of type `list(A)` (for any type `A`).

2. For the constructor `Cons`, "rule" works as follows:

$$\begin{aligned}\text{rule } [\![\texttt{List [a] Cons(a, List(a))}]\!] &= \\ &= \texttt{gen}(\texttt{list}(\texttt{A}), \texttt{cons}(\texttt{CONS}_1, \texttt{CONS}_2)) \\ &\leftarrow \texttt{gen}(\texttt{A}, \texttt{CONS}_1) \wedge \texttt{gen}(\texttt{list}(\texttt{A}), \texttt{CONS}_2)\end{aligned}$$

   This rule is saying that $\texttt{cons}(\texttt{CONS}_1, \texttt{CONS}_2)$ is a value of type `list(A)`, provided that $\texttt{CONS}_1$ can be generated as a value of type `A` and $\texttt{CONS}_2$ can be generated as a value of type `list(A)`.

Once we have the two rules needed to build lists, we can combine them using a dis-

junction according to "parseDec", resulting in

$$\begin{aligned}&\text{parseDec } [\![\texttt{DataD List [a] [Nil, Cons(a, List(a))]}]\!] = \\ &\quad = \texttt{gen}(\texttt{list}(\texttt{A}), \texttt{nil}) \vee \\ &\quad\quad \texttt{gen}(\texttt{list}(\texttt{A}), \texttt{cons}(\texttt{CONS}_1, \texttt{CONS}_2)) \leftarrow \texttt{gen}(\texttt{A}, \texttt{CONS}_1) \wedge \texttt{gen}(\texttt{list}(\texttt{A}), \texttt{CONS}_2)\end{aligned}$$

Note the similarities between this generator, which was produced completely automatically from the data type declaration, and the one in (3.1), which was produced by hand based on an intuitive understanding of how lists are made up in a functional setting. ♦

Example 3.2.2 (Rose trees)**.** Let us continue from where we left example 3.1.2. Since now we only have one constructor, we apply "rule" only once:

$$\begin{aligned}&\text{rule } [\![\texttt{RoseTree [a] Rose(a, ListT(RoseTree(a)))}]\!] = \\ &\quad = \texttt{gen}(\texttt{rosetree}(\texttt{A}), \texttt{rose}(\texttt{ROSE}_1, \texttt{ROSE}_2)) \\ &\quad\quad \leftarrow \texttt{gen}(\texttt{A}, \texttt{ROSE}_1) \wedge \texttt{gen}(\texttt{listt}(\texttt{rosetree}(\texttt{A})), \texttt{ROSE}_2)\end{aligned}$$

Now, using "parseDec" yields

$$\begin{aligned}&\text{parseDec } [\![\texttt{DataD RoseTree [a] [Rose(a, ListT(RoseTree(a)))]}]\!] = \\ &\quad = \texttt{gen}(\texttt{rosetree}(\texttt{A}), \texttt{rose}(\texttt{ROSE}_1, \texttt{ROSE}_2)) \\ &\quad\quad \leftarrow \texttt{gen}(\texttt{A}, \texttt{ROSE}_1) \wedge \texttt{gen}(\texttt{listt}(\texttt{rosetree}(\texttt{A})), \texttt{ROSE}_2)\end{aligned}$$

♦

Limitations

The generators that are created at this stage model very closely how we intuitively think about algebraic data types, as evidenced by the resemblance between the hand-made generator for lists and the one produced automatically with our translation rules. However, they are far from ideal when it comes to actually producing data values, in at least two ways. Let us illustrate the issues with some examples:

Example 3.2.3 (Selecting a rule)**.** Consider the generator for lists presented in example 3.2.1 above. It is straightforward to translate it directly into an equivalent Prolog program as follows:

```
gen(list(A), nil).
gen(list(A), cons(CONS1, CONS2)) :- gen(A, CONS1), gen(list(A), CONS2).
```

Now, let us use this generator to produce lists of integers. To do this, we could use the generator for integers described above; however, for the sake of exposition, it is better to restrict the generation of integers to a smaller range instead of using the whole `Int` type. For this reason we set the bounds for integers to -10 and 10, leaving us with the following simplified generator:

```
gen(int, X) :- random_between(-10, 10, X).
```

Now, writing these rules in a Prolog file and loading it into an interpreter, we can use a query like `gen(list(int), L)` to automatically produce random lists of integers; a possible output to the query could be

```
?- gen(list(int), L).
L = nil ;
L = cons(5, nil) ;
L = cons(5, cons(-6, nil)) ;
L = cons(5, cons(-6, cons(-4, nil))) ;
L = cons(5, cons(-6, cons(-4, cons(10, nil)))) ;
L = cons(5, cons(-6, cons(-4, cons(10, cons(10, nil))))) ;
L = cons(5, cons(-6, cons(-4, cons(10, cons(10, cons(-1, nil)))))) ;
```

At first glance, this may seem like the intended behaviour: the Prolog interpreter is answering the query with randomly generated lists of integers that are inside the specified range. However, there are two details that are worth pointing out here:

1. The user needs to actively *ask* for more answers to the query to get interesting enough list values (note the semicolons at the end of each line; see. 2.3.3).
2. The prefixes of the generated lists are always the same; the only difference between each line and the previous one is that the former has one more element appended at the end with respect to the latter.

♦

These may look like minor inconveniences: one could try to solve the first issue by using one of Prolog's *aggregation predicates*, and the second could be avoided by choosing a big enough list from the output of each query and discarding the rest. However, this is unsatisfactory, as will become apparent after the following example. Before that, let us look into *why* this is happening in the first place.

Both of these behaviours can be understood by reasoning about Prolog's search strategy when answering a query.

By default, Prolog tries to apply the rules in the program in the order in which they are given. When a rule is successfully applied, it presents the chosen variable assignment to the user, who then can ask Prolog to give another solution. This forces Prolog to backtrack and continue the search, producing the solutions as they are being found. In the case of the previous example, this takes the following form:

1. `nil` is given as a valid element of type `list(int)`.
2. When the user asks for another solution, Prolog looks for another rule; this yields `cons(CONS1, CONS2)`, provided that `gen(int, CONS1)` and `cons(list(int), CONS2)` both hold. The first can be resolved by using the generator for integers, and the second by the `nil` rule for lists.
3. When the user asks for another solution, Prolog first backtracks in the *last* rule it used, which in this case is the `nil` constructor for lists. After this, it tries to look for another rule to apply instead of that, yielding `cons(CONS1, CONS2)`; as before, `CONS1` is generated as a random integer and `CONS2` is generated as the `nil` list.

This way, the process continues generating longer and longer lists in place of the tail of the previous list, producing the behaviour observed in the previous example.

Note that, if instead of the code generated by our translation rules (and presented above) we had written an alternative version like this,

```
gen(list(A), nil).
gen(list(A), cons(CONS1, CONS2)) :- gen(list(A), CONS2), gen(A, CONS1).
```

(where the only difference is that the order in which `CONS1` and `CONS2` are generated is swapped), then the behaviour of the generator would be qualitatively different: now the generated lists no longer share all their prefixes, as can be seen in the following sample output to the same query:

```
?- gen(list(int), L).
L = nil ;
L = cons( 4, nil) ;
L = cons(-3, cons( 0, nil)) ;
L = cons(-5, cons(-8, cons(-6, nil))) ;
```

```
L = cons( 6, cons( 3, cons( 6, cons(10, nil)))) ;
L = cons( 6, cons( 3, cons(-7, cons(-9, cons(-5, nil))))) ;
L = cons( 3, cons( 8, cons(-7, cons( 6, cons( 5, cons(1, nil)))))) ;
```

The reason is that now the backtracking would first try to undo the generation of the head of the list, which is an integer; since, in this case, there is only one rule for generating an integer, the backtracking fails on this first rule, and Prolog tries to backtrack on the next rule, which is the `nil` constructor for lists. Therefore, instead of using `nil` it tries to use `cons` (producing a list of length 1, instead of length 0), and *then* tries to re-apply the rule for generating an integer. This results in completely new lists being generated at each step (at the cost of quadratic complexity, instead of linear like before).

This solves the second issue (not the first), but it is still unsatisfactory because it required changing the generated code with some intuitive understanding of Prolog's search strategy and, more importantly, an understanding of the data structure being generated. This is something that cannot be done automatically in an easy way (besides not being a full solution to the problem at hand), so we choose not to explore this possibility further.

Let us look next at the other—more important—reason why this approach to producing the generators is insufficient:

Example 3.2.4 (Biased generation)**.** Consider an algebraic data type for encoding binary trees, `data BinTree a = Leaf | Node (BinTree a) a (BinTree a)`, that is, each node contains a value and has exactly two (possibly empty) children. The encoding of this type in our syntax would be as follows,

```
DataD BinTree
      [a]
      [Leaf,
       Node(BinTree(a), a, BinTree(a))]
```

which yields the following generators when using "parseDec":

$$\begin{aligned}&\text{parseDec } [\![\texttt{DataD BinTree [a] [Leaf, Node(BinTree(a), a, BinTree(a))]}]\!] = \\ &\quad = \texttt{gen}(\texttt{bintree}(\texttt{A}), \texttt{leaf}) \vee \\ &\qquad \texttt{gen}(\texttt{bintree}(\texttt{A}), \texttt{node}(\texttt{NODE1}, \texttt{NODE2}, \texttt{NODE3})) \\ &\qquad\quad \leftarrow \texttt{gen}(\texttt{bintree}(\texttt{A}), \texttt{NODE1}) \wedge \texttt{gen}(\texttt{A}, \texttt{NODE2}) \wedge \texttt{gen}(\texttt{bintree}(\texttt{A}), \texttt{NODE3})\end{aligned}$$

This can be easily translated into actual Prolog code as follows:

```
gen(bintree(A), leaf).
gen(bintree(A), node(NODE1, NODE2, NODE3)) :- gen(bintree(A), NODE1),
                                              gen(A, NODE2),
                                              gen(bintree(A), NODE3).
```

Using the same generator for integers as before, a query like `gen(bintree(int), T)` can produce an output like the following:

```
?- gen(bintree(int), T).
T = leaf ;
T = node(leaf, -8, leaf) ;
T = node(leaf, -8, node(leaf, -7, leaf)) ;
T = node(leaf, -8, node(leaf, -7, node(leaf, 6, leaf))) ;
T = node(leaf, -8, node(leaf, -7, node(leaf, 6, node(leaf, 2, leaf)))) ;
```

Notice that the issue of the prefixes being replicated across all structures remains. More importantly, a new issue arises when considering a data structure that is not linear like the lists from the previous example: the generated data values are completely biased towards one side. In all of the generated values starting from the second, the root of the tree contains a value of `-8` and has an empty left child; then, starting from the third value, all right children contain a value of `-7` at the root and have an empty left child... the same pattern continues through all the generated binary trees.

Figure 3.3: Graphical representation of the generated trees

The reason for this is that the backtracking never reaches the left child; Prolog always finds a way to continue with the generation just by creating different root values or right children, so the first constructor that is used for the left child (in this case, `leaf`) is kept across all solutions found by Prolog's backtracking algorithm.

♦

3.2.2 Second approach: sized generation

All of the issues described at the end of the previous subsection can be solved by taking a notion of *size* into account when generating the data values. In the case of lists, this 'size' measure intuitively corresponds to the length, so generating a list of some size that is fixed in advance eliminates the need to ask Prolog for more values until an interesting enough data value is produced. This way, Prolog answers each query with only one list value, so there is no issue with repeated prefixes. In the example of binary trees, the generator can make sure that both children have (exactly or approximately) the same size, so it is guaranteed that biased trees are never generated.

Before describing our solution, we first need to formally define what we mean by the *size* of a value belonging to an algebraic data type:

Definition 3.2.1 (ADT size)**.** Let v be a value belonging to an ADT, so that it is built using some (fully instantiated) constructor. Then,

- If the constructor has no arguments, then its size is defined to be 0:

$$size(ctorName()) \coloneqq 0.$$

- If the constructor has a positive number of arguments, then its size is defined to be one more than the greatest size among all the arguments:

$$size(ctorName(arg_1, \ldots, arg_r)) \coloneqq 1 + \max_{1 \leq i \leq r} size(arg_i).$$

In this context, we take literals for basic types (integers, booleans, strings, etc.) to be constructors with 0 arguments, meaning that they have a size of 0. ▲

In the discussions that follow we will often need to distinguish between constructors that accept no arguments and constructors that require at least one; for this reason, we introduce the following definition:

Definition 3.2.2 (Terminals and non-terminals)**.** A constructor that accepts no arguments is said to be *terminal*. In contrast, a constructor that expects at least one argument is called *non-terminal*. ▲

Intuitively, this definition of "size" corresponds to the height of the abstract tree representing the *structure* of the value. However, in many standard data structures, this notion of size can be better understood in some other way; let us see it with some examples:

Example 3.2.5 (List length)**.** Consider the list `[42, 13, 7, 6]`, which can be written out explicitly as `Cons(42, Cons(13, Cons(7, Cons(6, Nil))))`. This ADT value can be represented graphically as the tree shown in figure 3.4:

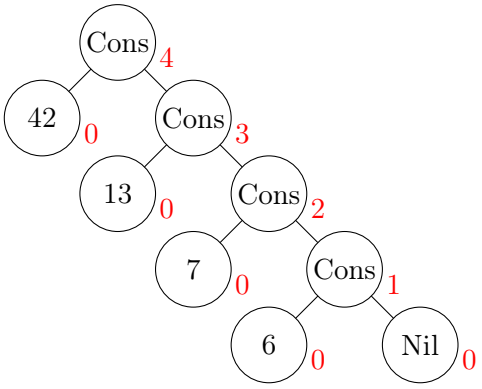

Figure 3.4: Graphical representation of the example list

Each node has its corresponding size written next to it in red. Notice that the terminal constructor `Nil` and the integer values have a size of 0, and the non-terminal constructors correspond to the nodes where the size increases.

Note as well that the size of the whole structure is 4, which is exactly equal to the *length* of the list. ♦

Example 3.2.6 (Binary tree height)**.** Consider the following binary tree:

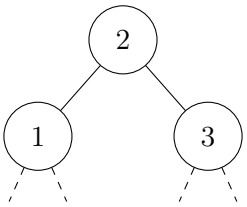

Figure 3.5: Binary tree example

If binary trees are defined as `data Tree a = L | N (Tree a) a (Tree a)`, the example tree can be encoded as `N (N L 1 L) 2 (N L 3 L)`.

The abstract tree modelling this binary tree can be represented graphically as depicted in figure 3.6. Now, the size of the example binary tree is 2, which coincides with its *height* in the usual sense. ♦

Remark 3.2.1. Even if this notion of "size" corresponds in many cases to a well-known concept about concrete data structures, this is not always the case. For instance, in a more

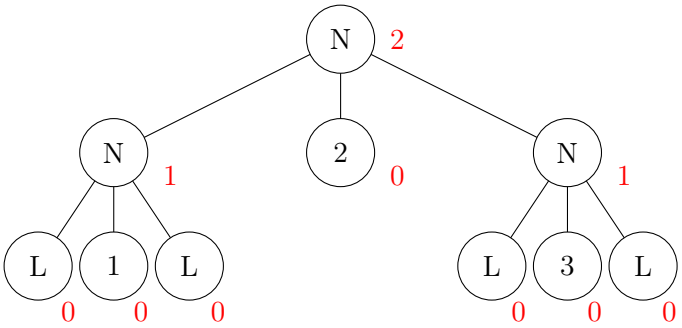

Figure 3.6: Graphical representation of the example tree

complex data structure like *rose trees*, the size in this sense no longer corresponds to the height of the tree. ■

With this, we are ready to extend the generator described in § 3.2.1 to produce values of a specific size: to do this, we can add one more argument to the predicate `gen/2` specifying the size that the generated value should have; then, the recursive calls to the new predicate `gen/3` are made with a size that is smaller than the original size by one.

This way, the generated values have abstract trees that are "complete", in the sense that *all* of the children of an internal node of size $s \geq 1$ have size $s - 1$; this can be a problem because it may happen that one data structure does not have values of a particular size. Before presenting our solution to these problems, it can be illustrative to see a few informal examples to better understand the issues and the proposed solutions:

Example 3.2.7 (Sized lists)**.** To generate a list of size $s = 0$, the only option is to use the *nil* constructor, because the *cons* constructor always produces lists of size at least 1. Conversely, to produce a list of size $s \geq 1$, we must use *cons*, and this requires first generating the head of the list (a value of the base type) and the tail (another list of the same type), both of size $s - 1$.

Suppose we wanted to generate a list of $s = 2$ integers: in this case, to create the head we would need to create an integer of size $s - 2 = 1$, which is impossible with our definition of size; therefore, the generation rule would always fail to produce a list of two integers. Naturally, there is nothing special about the number 2 here; the same applies for any size greater than 1. ♦

Note that this is not an issue with our definition of size, because the only reasonable size for a terminal constructor is 0; moreover, any other choice of 'size' for terminal constructors would still produce the same behaviour, so changing the definition for the case of terminals is not a valid solution to the problem at hand.

The problem actually arises because the requirement that *every* child of a node of size s should have size $s - 1$ is too strict. See figure 3.4: in a well-formed list, in *every* node corresponding to the *cons* constructor, the size is one more than the *right* child, while the left child always has a size of 0 (since we are dealing with lists of integers; lists of more complex types could be different). This observation suggests that *terminal constructors* (including literals of the base types) should be allowed to generate values of any size, not only size 0. This means that the translation rules should distinguish between terminal and non-terminal constructors to produce the generators.

Still, this does not solve the problem completely, as shown in the next example:

Example 3.2.8 (Deep recursion)**.** Some complex data structures are defined recursively, but the recursive call is hidden deeply in the structure; for example, rose trees can be defined in Haskell as `data RoseTree a = R a [RoseTree a]`, so the constructor for rose trees has an argument that is a *list* of other rose trees (in contrast to, for instance, binary trees, where the recursive constructor expects other binary trees *directly*). Essentially, this means that there is always a gap of at least 2 between the size of a rose tree and the size of its biggest child.

To see a concrete example, consider the rose tree `R 1 [R 11 [], R 12 []]`, which is represented visually like this:

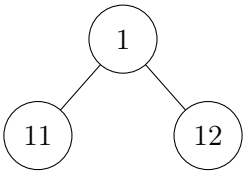

Figure 3.7: Rose tree example

The abstract tree encoding this value can be represented graphically as

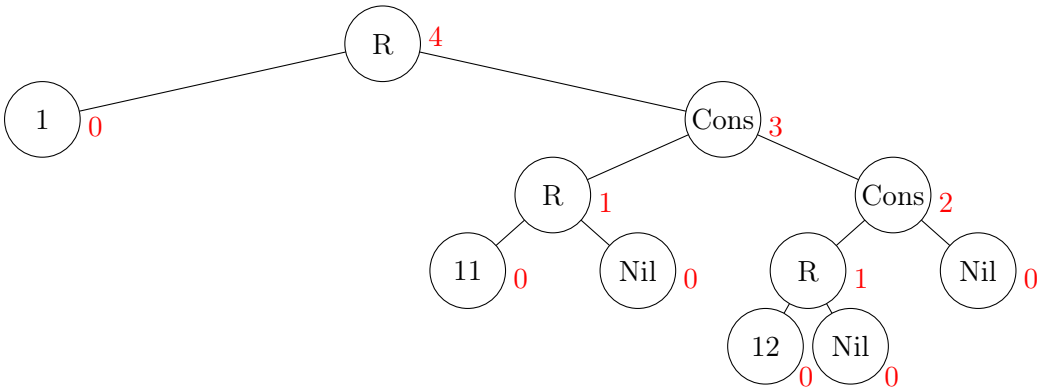

Figure 3.8: Graphical representation of the example rose tree

The value in the root is a 1, and the (non-empty) list of children is constructed using *cons*; this needs a head (another rose tree) and a tail (which is again constructed using *cons*), etc.

To generate a rose tree of size 4 like this automatically, we first need to generate a value for the root (which we already solved) and a list of children, of size 3. To generate this list, using *cons*, we should first generate the head of the list, which should be a rose tree of size 2 (note that, in the above abstract tree, the rose tree that occupies this position has size 1, instead of 2).

To generate this rose tree of size 2, we need to create a list of children of size 1, necessarily using the *cons* operator. This requires constructing a head, which should be a rose tree *of size 0*, which is impossible since there is only one construtor for rose trees, and it is non-terminal. ♦

This example shows that our simplistic generation rules can never produce a rose tree of size 4, even though *there are* rose trees of this size. Again, this is because the requirement that children have a size exactly one unit smaller is too strict. However, in this case, the problem does not arise with a terminal constructor, but with a non-terminal that cannot exist (there are no rose trees of size 2, as the above informal argument shows).

This example also illustrates the complex relationships that can appear between *different* ADT's when using recursion. The misstep in the generation was when we tried to generate a list of rose trees with size 1, and this cannot be easily detected during the value generation because lists do not (and should not) know about the definition of rose trees, or any other data type that *uses* lists.

Fortunately, there is a simple solution to this problem that leverages Prolog's search strategy. The idea is to always place the generation rules for non-terminals before those for terminals. This way, if a non-terminal rule is applicable, it will be used first (which is desirable, because then the generated values will be as large as possible while staying inside the bounds defined by the specified size). Otherwise, it should be allowed to use the rules for terminals (which are always applicable), meaning that they act as a "default" value.

In particular, the order in which the generation rules appear in the code is now crucial, so that latter rules should *only* be used if the ones that came before were not applicable. For this reason, we will frequently use the *cut* operator (see § 2.3.3) in the new rules and we will use *sequences* of Horn clauses ($\mathcal{H}^*$) to model Prolog programs instead of *sets* of clauses ($\mathcal{P}(\mathcal{H})$).

The generation rules for basic types need to be updated with the notion of size. Given the preceding discussion this is an easy task: for almost every type, it is enough to add the 'size' argument to the `gen` predicate using a wildcard variable `_`, indicating that the produced value can be valid for any desired size; for example, for integers we have

```
gen(int, _, X) :- random_between(-9223372036854775808,
                                 9223372036854775807,
                                 X).
```

which is almost identical to the one in § 3.2.1. The only types that need to be updated significantly are lists and tuples:

- For lists, the terminal constructor *nil* can be extended with a wildcard `_` as the rest of the basic types. The non-terminal constructor *cons* needs to ensure that its size is at least one, and generate the head and tail with a size smaller in one. Formally,

$$\begin{aligned}&\texttt{gen}(\texttt{listt}(\texttt{A}), \texttt{N}, [\texttt{X}|\texttt{XS}]) \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \\ &\quad \texttt{gen}(\texttt{A}, \texttt{M}, \texttt{X}) \wedge \texttt{gen}(\texttt{listt}(\texttt{A}), \texttt{M}, \texttt{XS}) \veebar \\ &\texttt{gen}(\texttt{listt}(\texttt{A}), \_, []) \end{aligned}$$

  and, as a Prolog program,

```
gen(list__(A), N, [X|XS]) :- N >= 1, M is N-1,
                             gen(A, M, X), gen(list__(A), M, XS), !.
gen(list__(A), _, []) :- !.
```

  Again, notice that this is very similar to the generator from 3.2.1, but including sizes and non-commutative rules, and placing the non-terminal constructor first.

- In the case of tuples, the approach is the same as for lists. Formally,

$$\begin{aligned}&\texttt{gen}(\texttt{tuplet}([\texttt{T}]), \texttt{N}, [\texttt{X}]) \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \\ &\quad \texttt{gen}(\texttt{T}, \texttt{M}, \texttt{X}) \veebar \\ &\texttt{gen}(\texttt{tuplet}([\texttt{T}|\texttt{TS}]), \texttt{N}, [\texttt{X}|\texttt{XS}]) \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \\ &\quad \texttt{gen}(\texttt{T}, \texttt{M}, \texttt{X}) \wedge \texttt{gen}(\texttt{tuplet}(\texttt{TS}), \texttt{N}, \texttt{XS}) \veebar \\ &\texttt{gen}(\texttt{tuplet}([]), \_, \texttt{unit}) \end{aligned}$$

  and, as a Prolog program,

```
gen(tuple__([T]),    N, [X]) :- !, N >= 1, M is N-1,
                                   gen(T, M, X).
```

```
gen(tuple__([T|TS]), N, [X|XS]) :- N >= 1, M is N-1,
                                   gen(T, M, X),
                                   gen(tuple__(TS), N, XS), !.
gen(tuple__([]),     _, unit).
```

One detail worth noting is that, to construct a tuple where every element has the same size, we need to use the size `M` (that is, `N-1`) for the size of the head and `N` itself for the tail; if the tail used `M` as well, the elements in the tuple would get smaller by (at least) one unit, which is not the desired result. A cleaner and more uniform approach to solve this issue is presented in 3.2.3.

We can finally present the new translation rules for size-aware generators, shown in figure 3.9—the complete Haskell code performing the translation can be found in appendix A.2.2.

$$\begin{aligned}
&\text{parseDec} : \texttt{Dec} \to \mathcal{H}^* \\
&\text{parseDec } [\![\texttt{DataD } typeName\ typeVars\ [con_1, \ldots, con_n]]\!] = \\
&\quad = \bigvee_{\substack{1 \le i \le r \\ con_i \text{ non-terminal}}} \text{ruleNonTerminal } [\![typeName\ typeVars\ con_i]\!] \veebar \\
&\qquad \bigvee_{\substack{1 \le i \le r \\ con_i \text{ terminal}}} \text{ruleTerminal } [\![typeName\ typeVars\ con_i]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{ruleNonTerminal} : Name \times TyVar^* \times Con \to \mathcal{H} \\
&\text{ruleNonTerminal } [\![typeName\ typeVars\ ctorName(ctorParam_1, \ldots, ctorParam_r)]\!] = \\
&\quad = \texttt{gen}(\texttt{typeName}(\texttt{TypeVars}),\ \texttt{N},\ \texttt{ctorName}(\texttt{FreshVar}_1, \ldots, \texttt{FreshVar}_\texttt{r})) \\
&\qquad \leftarrow \texttt{N} \ge 1 \wedge \texttt{M} := \texttt{N} - 1 \ \wedge \bigwedge_{i=1}^{r} \texttt{gen}(\{ctorParam_i\}, \texttt{M}, \texttt{FreshVar}_\texttt{i}) \\
&\quad \text{where } [\texttt{FreshVar}_1, \ldots, \texttt{FreshVar}_\texttt{r}] = freshVars(ctorName, r)
\end{aligned}$$

$$\begin{aligned}
&\text{ruleTerminal} : Name \times TyVar^* \times Con \to \mathcal{H} \\
&\text{ruleTerminal } [\![typeName\ typeVars\ ctorName]\!] = \\
&\quad = \texttt{gen}(\texttt{typeName}(\texttt{TypeVars}),\ _,\ \texttt{ctorName})
\end{aligned}$$

Figure 3.9: Translation rules for sized data type generation

The rule "parseDec" splits the constructors into terminals and non-terminals and uses the corresponding rule on each one (placing all the non-terminals first). "ruleNonTerminal" is very similar to "rule" from figure 3.2, but it describes how to generate values of some

size `N`: it first ensures that `N` is at least 1, and then it recursively uses the predicate `gen/3` to generate all the arguments with a size smaller in one unit. "ruleTerminal" is simpler, because it does not need to generate the arguments for the constructor: it simply states that the constructor alone is enough to generate a value of the specified type, for any size (note the wildcard variable `_`).

Note that the variables for representing the sizes of the generated data values (`N` and `M` in the translation rules presented in figure 3.9) are respectively `N__` and `M__` in the actual generated code, to avoid name clashes with the type variables appearing in the declarations provided by the user; they are written without underscores in this text to simplify the notation.

Let us look at a few examples to show how these translation rules work in practice:

Example 3.2.9 (Custom lists)**.** Consider the data type `data L a = N | C a (L a)`, which defines the usual list structure (where the constructor names *nil* and *cons* have been shortened for convenience). The constructor `N` is clearly a terminal, while `C` is non-terminal. Let us work with each individually before putting them all together:

- For `N` (*nil*), the translation rule "ruleTerminal" works as follows:

$$\text{ruleTerminal } [\![L\ a\ N]\!] = \texttt{gen}(\texttt{l}(\texttt{A}),\ \_,\ \texttt{n}),$$

  stating that `n` is a valid element of type `l(A)` for any size.

- For `C` (*cons*), the translation rule "ruleNonTerminal" works as follows:

$$\begin{aligned}&\text{ruleNonTerminal } [\![L\ a\ C(a, L(a))]\!] = \\ &\quad = \texttt{gen}(\texttt{l}(\texttt{A}),\ \texttt{N},\ \texttt{c}(\texttt{C1}, \texttt{C2})) \\ &\qquad \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \texttt{gen}(\texttt{A}, \texttt{M}, \texttt{C1}) \wedge \texttt{gen}(\texttt{l}(\texttt{A}), \texttt{M}, \texttt{C2})\end{aligned}$$

Now, according to "parseDec", we have

$$\begin{aligned}&\text{parseDec } [\![\texttt{DataT}\ L\ a\ [N,\ C(a, L(a))]]\!] = \\ &\quad = \texttt{gen}(\texttt{l}(\texttt{A}),\ \texttt{N},\ \texttt{c}(\texttt{C1}, \texttt{C2})) \\ &\qquad \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \texttt{gen}(\texttt{A}, \texttt{M}, \texttt{C1}) \wedge \texttt{gen}(\texttt{l}(\texttt{A}), \texttt{M}, \texttt{C2}) \veebar \\ &\quad\ \ \texttt{gen}(\texttt{l}(\texttt{A}),\ \_,\ \texttt{n})\end{aligned}$$

These generation rules can be readily translated into valid Prolog code (notice the use of the *cut* operator, `!`, to model the non-commutative disjunction):

```
gen(l(A), N, c(C1, C2)) :- N >= 1, M is N-1,
                           gen(A, M, C1), gen(l(A), M, C2), !.
gen(l(A), _, n) :- !.
```

This can be directly loaded into a Prolog interpreter together with a sized generator for some simple type like integers,

```
gen(int, _, X) :- random_between(-10, 10, X).
```

This way, we can automatically generate random integer lists of some pre-specified length with a query like the following:

```
?- gen(l(int), 4, L).
L = c(-2, c(3, c(-6, c(6, n)))).

?- gen(l(int), 4, L).
L = c(-10, c(-5, c(-1, c(-9, n)))).

?- gen(l(int), 4, L).
L = c(-6, c(-4, c(-5, c(0, n)))).
```

Notice that each query produces one and only one list of exactly the required length, as opposed to the behaviour of the previous generator (cf. example 3.2.3). ♦

Example 3.2.10 (Binary trees)**.** Consider the type of binary trees, which can be defined in Haskell as `data Tree a = L | N (Tree a) a (Tree a)`; again, there is one terminal and one non-terminal constructor, and we deal with them separately:

- The *leaf* terminal constructor `L` is translated as

$$\text{ruleTerminal } [\![Tree\ a\ L]\!] = \texttt{gen(tree(A), \_, l)}$$

- The *internal node* non-terminal constructor `N` is translated as

$$\begin{aligned}\text{ruleNonTerminal } [\![Tree\ a\ N(Tree(a), a, Tree(a))]\!] = \\ = \texttt{gen(tree(A), N, n(N1, N2, N3))} \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \\ \texttt{gen(tree(A), M, N1)} \wedge \texttt{gen(A, M, N2)} \wedge \texttt{gen(tree(A), M, N3)}\end{aligned}$$

Finally, according to "parseDec", we have

$$\begin{aligned}&\text{parseDec } [\![\texttt{DataD } Tree\ a\ [L, N(Tree(a), a, Tree(a))]]\!] = \\ &\quad = \texttt{gen(tree(A), N, n(N1, N2, N3))} \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \\ &\qquad \texttt{gen(tree(A), M, N1)} \wedge \texttt{gen(A, M, N2)} \wedge \texttt{gen(tree(A), M, N3)} \veebar \\ &\quad \texttt{gen(tree(A), \_, l)}\end{aligned}$$

As before, it is straightforward to translate this into Prolog:

```
gen(tree(A), N, n(N1, N2, N3)) :- N >= 1, M is N-1,
                                  gen(tree(A), M, N1),
                                  gen(A, M, N2),
                                  gen(tree(A), M, N3), !.
gen(tree(A), _, l) :- !.
```

Now, with the same sized generator for integers as before, we can automatically generate binary trees with a simple query:

```
?- gen(tree(int), 4, T).
T = n(n(n(n(l, -6, l), 10, n(l, 7, l)), -9, n(n(l, -10, l), -5, n(l, -9, l))),
  -5, n(n(n(l, -7, l), -8, n(l, 2, l)),  9, n(n(l,   4, l), -7, n(l,  3, l)))).
```

For reference, this binary tree can be represented graphically as follows:

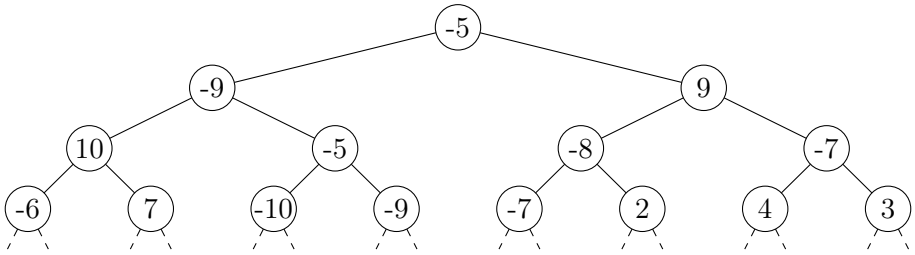

Figure 3.10: Graphical representation of the generated tree

As mentioned just after the definition of our notion of 'size', the structures that are generated by our rules are *complete*, whenever possible. In the case of binary trees, this corresponds perfectly to the standard idea of complete trees, as can be seen in the above figure. Contrast this with the degenerate trees from figure 3.3, which also reached a height of 4 but where generated without taking into account the size of the children. ♦

Example 3.2.11 (Deep recursion)**.** Let us try to generate values from a more complex data structure; in particular, let us focus on *rose trees*, `data RoseTree a = R a [RoseTree a]`. As previously noted, these do not have any terminal construtor, which poses a challenge as the value generation should reach a "base case" at some point. Before seeing how that is resolved, let us calculate the generation rules:

$$\begin{aligned}&\text{ruleNonTerminal } [\![\mathit{RoseTree},\ a,\ R(a,[\mathit{RoseTree}(a)])]\!] = \\ &\quad = \texttt{gen}(\texttt{rosetree}(\texttt{A}),\ \texttt{N},\ \texttt{r}(\texttt{R1},\texttt{R2})) \\ &\qquad \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \texttt{gen}(\texttt{A},\texttt{M},\texttt{R1}) \wedge \texttt{gen}(\texttt{listt}(\texttt{rosetree}(\texttt{A})),\texttt{M},\texttt{R2})\end{aligned}$$

$$\begin{aligned}&\text{parseDec } [\![\mathit{RoseTree}\ a\ [R(a,[\mathit{RoseTree}(a)])]]\!] = \\ &\quad = \texttt{gen}(\texttt{rosetree}(\texttt{A}),\ \texttt{N},\ \texttt{r}(\texttt{R1},\texttt{R2})) \\ &\qquad \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \texttt{gen}(\texttt{A},\texttt{M},\texttt{R1}) \wedge \texttt{gen}(\texttt{listt}(\texttt{rosetree}(\texttt{A})),\texttt{M},\texttt{R2})\end{aligned}$$

As usual, the translation of these generation rules to Prolog code is straightforward:

```
gen(rosetree(A), N, r(R1, R2)) :- N >= 1, M is N-1,
                                  gen(A, M, R1),
                                  gen(list__(rosetree(A)), M, R2), !.
```

Let us try to generate rose trees of integers by hand to better understand how the generation works even in the case where there are no terminal constructors:

- It is not possible to produce a rose tree of **size 0**, since both rules have the requirement that the size `N` is at least one. The Prolog interpreter confirms this,

  ```
  ?- gen(rosetree(int), 0, X).
  false.
  ```

  Note that this is not a limitation of the generation rules; with our definition of "size", *there are no* rose trees of size 0.

- To produce a rose tree of **size 1**, we start with the first rule: it requires generating an integer and a list of rose trees, both of size 0. Since integers always have size 0, this can be done immediately; suppose the generated value is `3`. For lists, recall that the generators provided for standard Haskell lists are the following:

  ```
  gen(list__(A), N, [X|XS]) :- N >= 1, M is N-1,
                               gen(A, M, X), gen(list__(A), M, XS), !.
  gen(list__(A), _, []) :- !.
  ```

To generate a list of rose trees of size 0, the first rule is not applicable; then we try the second one, which can always be used, producing the empty list. (Note that the third rule for lists is not tried once the second succeeds because of the cut operator.)

Therefore, the generation rule for rose trees succeeds, producing the value `r(3, [])`. In the Prolog interpreter,

```
?- gen(rosetree(int), 1, X).
X = r(3, []).
```

- To produce a rose tree of **size 2**, we try to use the first rule; as before, this requires generating an integer and a list of rose trees, both of size 1. In the case of the integer there is no problem, but generating the list is a bit more delicate. The first rule tries to produce a list with a head and a tail; the head should be a rose tree and the tail should be a list of rose trees, both of size 0; however, as previously noted, there are no rose trees of size 0, meaning that this rule for generating the list will fail. After this, the next rule is tried, giving the empty list as a result (which has size 0, and not 1). This way, the result is again something like `r(3, [])` and, again, the Prolog interpreter confirms this.

Notice that the internal representation of the structures can get quite large even for relatively small values:

```
?- gen(rosetree(int), 5, X).
X = r(7, [r(1, [r(-3, [])]), r(7, []), r(0, [])]).
```

This structure can be represented graphically as in the tree in figure 3.11 (where, as usual, `:` and `[]` are the *cons* and *nil* constructors for lists, respectively), and it is an encoding of the relatively simple rose tree presented in figure 3.12 below.

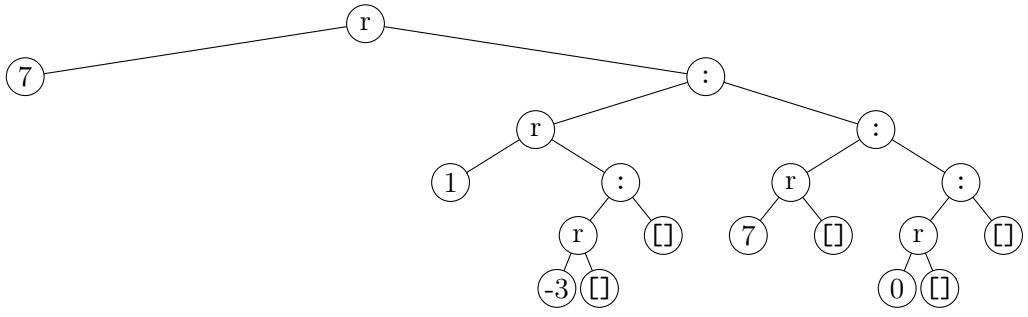

Figure 3.11: Structure of the generated rose tree

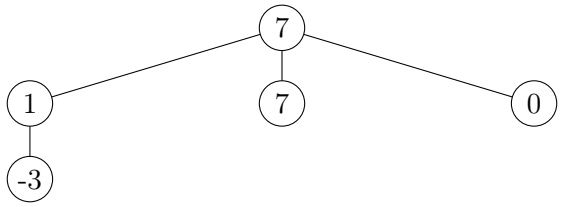

Figure 3.12: Graphical representation of the generated rose tree

Let us conclude with a bigger example:

```
?- gen(rosetree(int), 8, X).
X = r(-5, [r(2,  [r(2,  [r(-4, []), r(5, [])]),
                  r(3,  [r(5, [])]),
                  r(-4, []), r(4,  [])]),
           r(-4, [r(8,  [r(1, [])]),
                  r(3,  []), r(9,  [])]),
           r(3,  [r(-4, []), r(6, [])]),
           r(4,  [r(4,  [])]),
           r(6,  []), r(-8, [])]).
```

where the output has been formatted for clarity. Finally, here is a visualisation of the tree:

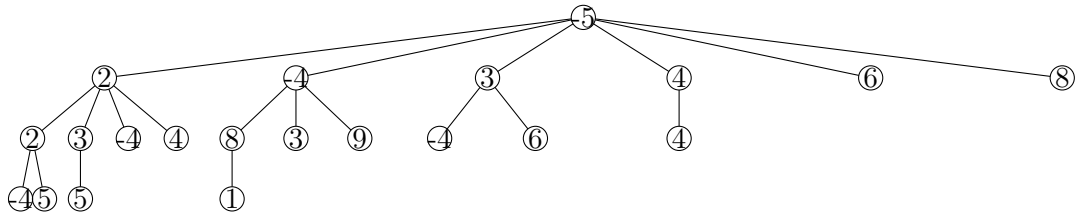

Figure 3.13: Generated rose tree of size 8

♦

Remark 3.2.2. Notice that, for complex data structures like rose trees, our notion of "size" does not correspond to any meaningful concept about the data structure (the tree height, the number of elements it contains...) Still, it provides a uniform way to produce values that are neither too small nor too big for a very wide range of data types, completely automatically.

Notice too that the generated rose trees seem to be leaning towards the left. This is a consequence of the way lists are usually constructed in a functional setting (the *cons*

constructor builds up a list using a *head* and a *tail*) that is hard to avoid. One option could be to assign different weights to the different arguments of each constructor, but this cannot be easily done in a way that is both automatic and meaningful for every data structure. ■

Limitations

Our generation rules can already produce interesting values—that are neither too small nor too big—for many real-world data structures, as shown in the above examples. However, all of these have one characteristic in common: they have at most one terminal and at most one non-terminal constructor. This is an issue because of the non-associativeness introduced in the rules since the first iteration, as illustrated by the following examples:

Example 3.2.12 (Either)**.** One of the simplest examples of a data type with more than one non-terminal constructor is the *either* type, `data Either a b = Left a | Right b`. The translation procedure creates these generation rules:

$$\texttt{gen(either(A,B), N, left(LEFT1))} \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \texttt{gen(A,M,LEFT1)} \veebar$$
$$\texttt{gen(either(A,B), N, right(RIGHT1))} \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \texttt{gen(B,M,RIGHT1)}$$

The fact that the different generation rules are aggregated with a *non-associative* disjunction means that the first rule is always tried first and, if it is successful (which, in general, it is), the second one is not used at all. This can be confirmed empirically with a Prolog interpreter by loading the code representing the rules,

```
gen(either(A, B), N, left(LEFT1)) :- N >= 1, M is N-1,
                                     gen(A, M, LEFT1), !.
gen(either(A, B), N, right(RIGHT1)) :- N >= 1, M is N-1,
                                       gen(B, M, RIGHT1), !.
```

and asking a simple query:

```
?- gen(either(int, int), 7, X).
X = left(10).

?- gen(either(int, int), 7, X).
X = left(-8).
```

```
?- gen(either(int, int), 7, X).
X = left(-7).

?- gen(either(int, int), 7, X).
X = left(-2).

?- gen(either(int, int), 7, X).
X = left(-8).

?- gen(either(int, int), 7, X).
X = left(5).
```

♦

Example 3.2.13 (Booleans)**.** Of course, the same problem arises with data types with more than one terminal constructor, like the basic type of boolean values. The (predefined) rules for generating booleans are as follows,

$$\texttt{gen}(\texttt{bool}, \_, \texttt{false}) \veebar$$
$$\texttt{gen}(\texttt{bool}, \_, \texttt{true})$$

which, in this case, only produce the value `false`. ♦

3.2.3 Third approach: uniform generation

The issues presented at the end of the previous section can be solved in a very natural way, by choosing randomly a non-terminal constructor in "ruleNonTerminal" and a terminal constructor in "ruleTerminal".

This can be done by splitting all the constructors in terminals and non-terminals, and simplifying each rule to choose an element from the appropiate list using the standard Prolog predicate `random_member/2` (see 2.3.4). Finally, a value of the selected type can be generated automatically and uniformly with a new predicate called `create_values/3`.

This predicate has three arguments: the randomly chosen construction, the desired size, and the value that is generated. It first decomposes the construction into the constructor name and its arguments using the standard `=..` operator. Then, it uses the `gen/3` predicate to recursively generate all the needed arguments, and finally it instantiates the constructor with the generated values using the operator `=..` once again.

To generate the needed arguments, a new "internal" data type `all__` is introduced to simultaneously generate arbitrarily many data values. It is similar to tuples, but it always contains at least one value (to avoid dealing with the special case of `unit`), since we will only use it for non-terminal constructors, and the size of the construction is the same as the sizes of its arguments, because it is not regarded as a data constructor in itself, but rather as a way of simultaneously generating values of other types.

As Prolog code, this takes the following form:

```
gen(all__([T]),    N, [X]) :- !, gen(T, N, X).
gen(all__([T|TS]), N, [X|XS]) :- gen(T, N, X), gen(all__(TS), N, XS).

create_values(Con, N, X) :- Con =.. [Ctor|Args],
                            gen(all__(Args), N, Values),
                            X =.. [Ctor|Values].
```

Notice that the `create_values/3` predicate is completely general in that it can be used uniformly with any non-terminal construction and that it removes the burden of recursively generating the children from the generation rule "ruleNonTerminal".

The only nuance is that the generation rules produced by "ruleNonTerminal" and "ruleTerminal" will always fail to generate a value when they are passed an empty sequence of constructions. To avoid including useless generation rules that would always fail, we distinguish cases in the translation rules "ruleNonTerminal" and "ruleTerminal": when the sequence of constructions is non-empty they work as before, and they produce no rule when it is empty.

The new translation rules are presented in figure 3.14—the full Haskell code that implements them is shown in appendix A.2.3.

To conclude, let us see how this solves the issues presented at the end of the previous subsection:

Example 3.2.14 (Booleans)**.** Booleans are the only pre-defined data types with more than one terminal constructor; in this iteration, their two generation rules are merged into one in the same spirit as the new "ruleTerminal", like this:

```
gen(bool, _, B) :- random_member(B, [true, false]).
```

♦

$$
\begin{aligned}
&\text{parseDec} : \texttt{Dec} \to \mathcal{H}^* \\
&\text{parseDec } [\![\texttt{DataD } typeName\ typeVars\ cons]\!] = \\
&\quad = \text{ruleNonTerminal } [\![typeName\ typeVars\ nonTerminals]\!] \veebar \\
&\qquad \text{ruleTerminal } [\![typeName\ typeVars\ terminals]\!] \\
&\quad \text{where } terminals = (con \in cons \mid con \text{ terminal}), \\
&\qquad\quad nonTerminals = (con \in cons \mid con \text{ not terminal})
\end{aligned}
$$

$$
\begin{aligned}
&\text{ruleNonTerminal} : Name \times TyVar^* \times Con^* \to \mathcal{H} \\
&\text{ruleNonTerminal } [\![typeName\ typeVars\ cons]\!] = \\
&\quad = \texttt{gen}(\texttt{typeName}(\texttt{TypeVars}),\ \texttt{N},\ \texttt{X}) \\
&\qquad \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \texttt{random\_member}(\texttt{C}, \texttt{cons}) \wedge \texttt{create\_values}(\texttt{C}, \texttt{M}, \texttt{X})
\end{aligned}
$$

$$
\begin{aligned}
&\text{ruleTerminal} : Name \times TyVar^* \times Con^* \to \mathcal{H} \\
&\text{ruleTerminal } [\![typeName\ typeVars\ cons]\!] = \\
&\quad = \texttt{gen}(\texttt{typeName}(\texttt{TypeVars}),\ \texttt{N},\ \texttt{X}) \\
&\qquad \leftarrow \texttt{random\_member}(\texttt{X}, \texttt{cons})
\end{aligned}
$$

Figure 3.14: Translation rules for uniform data type generation

Apart from booleans, the other basic type whose generation rules can be improved is tuples: as noted before, the new `all__` construction is very similar to tuples in that it generates all its arguments "simultaneously"; therefore, it can be used in the generation rules for tuples in this way:

```
gen(tuple__(TS), N, XS) :- N >= 1, M is N-1,
                           gen(all__(TS), M, XS), !.
gen(tuple__([]), _, unit).
```

Notice that in these generation rules the desired size of $\texttt{N} - 1$ is computed once and it is used to generate all the arguments of this size, as opposed to the rules from § 3.2.2.

Example 3.2.15 (Either)**.** The *either* data type introduced in example 3.2.12 has two non-terminal constructors (*left* and *right*) and zero terminal constructors. The terminal

rule produces no generation rules, while the non-terminal rule gives

$$\begin{aligned}&\text{ruleNonTerminal } [\![\mathit{Either},\ a,\ b,\ [\mathit{Left}(a), \mathit{Right}(b)]]\!] = \\ &\quad = \texttt{gen}(\texttt{either}(\texttt{A},\texttt{B}),\ \texttt{N},\ \texttt{X}) \\ &\qquad \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} \coloneqq \texttt{N} - 1 \wedge \texttt{random\_member}(\texttt{C}, [\texttt{left}(\texttt{A}), \texttt{right}(\texttt{B})]) \wedge \\ &\qquad\quad \texttt{create\_values}(\texttt{C}, \texttt{M}, \texttt{X})\end{aligned}$$

Therefore, the full generator is given by

$$\begin{aligned}&\text{parseDec } [\![\texttt{DataD } \mathit{Either}\ a\ b\ [\mathit{Left}(a), \mathit{Right}(b)]]\!] = \\ &\quad = \texttt{gen}(\texttt{either}(\texttt{A},\texttt{B}),\ \texttt{N},\ \texttt{X}) \\ &\qquad \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} \coloneqq \texttt{N} - 1 \wedge \texttt{random\_member}(\texttt{C}, [\texttt{left}(\texttt{A}), \texttt{right}(\texttt{B})]) \wedge \\ &\qquad\quad \texttt{create\_values}(\texttt{C}, \texttt{M}, \texttt{X})\end{aligned}$$

The translation into Prolog is

```
gen(either(A, B), N, X) :- N >= 1, M is N-1,
                           random_member(C, [left(A), right(B)]),
                           create_values(C, M, X), !.
```

With this, our generation rules can uniformly produce values of the *either* type:

```
?- gen(either(int, int), 5, X).
X = left(-7).

?- gen(either(int, int), 5, X).
X = left(10).

?- gen(either(int, int), 5, X).
X = right(-1).

?- gen(either(int, int), 5, X).
X = right(-8).

?- gen(either(int, int), 5, X).
X = left(6).
```

♦

Chapter 4

Simulating dependent types through GADTs

Despite its rich type system, especially when compared to mainstream programming languages, standard Haskell falls short for enforcing—and even expressing—a certain kind of properties about data values at the type level. Still, there is a practical interest in doing so, given that many of these properties arise naturally in the implementation of some common data structure. Two examples of data structure properties that cannot be expressed in standard Haskell types are fixed-length vectors (used, for instance, in the implementation of bound-safe vector indexing) and the self-balancing properties of trees like AVL or red-black trees.

Some modern programming languages [6, 7, 8, 1] include a *dependent type system*, in which *types* are allowed to depend on arbitrary *values*. Such a strong type system enables the programmer to express invariants like these inside the data type definitions themselves, meaning that the type checker can be used to statically enforce the desired properties.

Generalised Algebraic Data Types (GADTs) can be seen as a step towards supporting dependent types in Haskell, especially in combination with other modern Haskell features like *data type promotion*.

4.1 Generalised Algebraic Data Types

Algebraic Data Types (ADTs) in standard Haskell can be regarded as a simultaneous generalisation of both *product types* (tuples, structs, classes, etc.) and *sum types* (unions, option types, etc.). This makes them useful for defining a wide range of data types:

Example 4.1.1 (Simple expressions)**.** A common use for ADTs is to encode expressions in a programming language:

```
data Expr = I Int
          | Add Expr Expr
          | Mul Expr Expr
          | B Bool
          | And Expr Expr
          | Or  Expr Expr
          | Eql Expr Expr
```

This type contains values representing valid expressions like

```
(I 3 `Mul` I 4) `Eql` (I 5 `Add` I 7)
```

which means it can be used to encode an Abstract Syntax Tree (AST) or to implement an Embedded Domain-Specific-Language (EDSL).

♦

ADTs in Haskell can be parameterised by one or more *type variables* to produce a family of types through what is known as *parametric polymorphism*; this enables the programmer to define, for example, the type of arbitrary lists (along with functions to perform computations on them) which can then be instantiated with arbitrary types, producing lists of integers, lists of booleans, lists of binary trees of strings, etc.

Standard parametric polymorphism, although very useful in a wide range of applications, has a few shortcomings. In the first place, it does not allow type *constants* to appear in the left-hand side of definitions (e.g. the user cannot provide one constructor for specifically creating lists of *integers*). In the second place, it does not support even simple *pattern matching* for types: it is not possible to give a different implementation of a function for specific types (e.g. one cannot define a function on lists that treats list of integers differently). *Generalised* Algebraic Data Types (GADTs) are an extension of standard ADTs adding support for these features. This is achieved using a more explicit syntax for data definitions, as the next example shows:

Example 4.1.2. Lists are usually defined in Haskell as follows:

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

An equivalent definition using GADT syntax could be

```
data List a where
  Nil :: List a
  Cons :: a -> List a -> List a
```

This definition is stating explicitly that `Nil` is a value of type `List a` (for *any* type `a`), while `Cons` is a data constructor that, given a value of type `a` and a list of type `List a`, creates another list of type `List a`.

The above definition can be extended with another constructor for producing exclusively lists of integers, without needing to define a new type only for this specific type of lists:

```
data List a where
  Nil :: List a
  Cons :: a -> List a -> List a
  ICons :: Int -> List Int -> List Int
```

This way, functions defined over the type `List a` by pattern matching can use the extra type information included in constructor `ICons`:

```
remove0 :: List a -> List a
remove0 Nil = Nil
remove0 (Cons x xs) = Cons x xs
remove0 (ICons x xs) = if x == 0 then xs else (ICons x (remove0 xs))
```

The previous function definition is well-typed because, when Haskell encounters the constructor `ICons`, it already knows that `x` is of type `Int` and `xs` is of type `List Int`, so `x == 0` is a valid expression. As an example,

```
ghci> remove0 (ICons 3 (ICons 0 (ICons 5 Nil)))
ICons 3 (ICons 5 Nil)
```

Note as well that `Nil` can be used to produce an (empty) list of any type, in particular of integers, so it can be used with `ICons`. ♦

4.2 Applications

An interesting application of GADTs is to add more information to an existing data structure through the use of *phantom types*—type parameters that are not reflected in the *values* of the structure, but act as type annotations for the various data constructors. Let us see an example:

Example 4.2.1 (Typed expressions)**.** The simple expressions from example 4.1.1 can be classified—intuitively—as either of type *integer* or type *boolean*, but this difference is not stated explicitly nor enforced by Haskell's type system. The declaration for the data type `Expr` can be rewritten using GADTs as follows to include this extra information:

```
1  data Expr t where
2    I :: Int -> Expr Int
3    Add, Mul :: Expr Int -> Expr Int -> Expr Int
4    B :: Bool -> Expr Bool
5    And, Or :: Expr Bool -> Expr Bool -> Expr Bool
6    Eql :: Expr t -> Expr t -> Expr Bool
```

For example, the constructor `I` receives a single integer and produces an expression of type `Int`, while `Eql` takes two expressions of the same type and creates an expression of type `Bool`.

Note how, for instance, `Add` or `Mul` are *labeled* as constructors producing expressions of type `Int`, but do not contain themselves any integer value (they do not have any argument of type `Int`)—this is the reason why the type `t` in the new declaration for `Expr` at line 1 above is known as a *phantom* type.

Crucially, this data type contains well-typed expressions and *only* well-typed expressions. The value from example 4.1.1,

```
(I 3 `Mul` I 4) `Eql` (I 5 `Add` I 7)
```

is an element of `Expr Bool`, since `I 3 `Mul` I 4` and `I 5 `Add` I 7` are elements of type `Expr Int`, and the data constructor `Eql` combines any expressions of the same type (`Int`, in this case) to produce an expression of type `Bool`. However, a Haskell expression like

```
I 3 `Or` B False
```

is *ill-typed*, and therefore does not correspond to an expression as defined by the `Expr t` data type. ♦

GADTs are especially useful when combined with automatic *type promotion*, that is, is the process of creating a new data *type* from an existing data *value*. Type promotion is not enabled by default in the Haskell compiler, but must be activated with the `DataKinds` pragma.

Example 4.2.2 (Data type promotion)**.** Given the data declaration below,

```
data Quality = Good | Bad
```

the data *values* `Good` and `Bad` can be promoted automatically to the *type* level (while the *type* `Quality` is made into a new *kind*), allowing us to use them as labels for constructors using GADTs:

```
data Var t where
  GoodVar :: Var Good
  BadVar  :: Var Bad
```

♦

Data type promotion can be used to simulate dependent type declarations inside Haskell's type system: wherever a type needs to depend on a *value*, it uses instead the *type* corresponding to the value.

As an example showing the importance of this technique, GHC includes a definition of one type for each natural number—along with some utilities for working with them, like *type constraints* expressing an inequality relation between two types—, for use in *type-level arithmetic*. An useful application of type-level arithmetic is implementing the type of *integers modulo n*, which can be defined polymorphically, using only one parameterised data type declaration, and instantiating it with any desired type-level natural.

Let us conclude this chapter with a case-study of how to combine these two features of Haskell—GADTs and data type promotion—to simulate a dependent type that encodes a type-safe version of *red-black trees*.

4.2.1 Height-balanced red-black trees

Red-black trees are interesting because they are a data structure with a complex enough invariant that it cannot be fully expressed in standard Haskell's type system. Let us introduce the precise definition:

Definition 4.2.1 (Red-black trees)**.** A *red-black tree* [19] is a binary search tree where each node is colored either red or black in such a way that the following conditions hold:

(a) The root should be black.

(b) Every leaf should be black.

(c) Every red node should have only black children.

(d) From every node, every path that ends in a leaf should have the same number of black nodes.

These conditions together ensure that, for each node, its two children are not too different in height, effectively guaranteeing that the tree as a whole is balanced. ▲

Here, we will focus exclusively on the color invariants, leaving aside the binary search property.

A Haskell data type representing red-black trees can be defined as follows:

```
data Color = Red | Black
data RBTree a = Leaf a | RB Color a (RBTree a) (RBTree a)
```

However, this naïve definition has the same problem as the simple expressions from example 4.1.1: it contains every correct value, but also some *incorrect* values that violate the invariant.

Example 4.2.3. Consider the following value of type `RBTree Int`:

```
RB Black 2 (Leaf 1) (RB Red 5 (Leaf 4) (RB Red 7 (Leaf 6) (Leaf 13)))
```

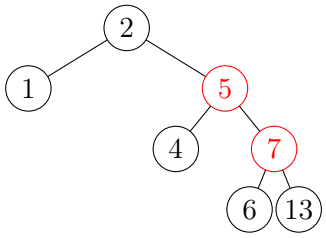

Figure 4.1: Graphical representation of the incorrect red-black tree

It is a valid value from the point of view of the Haskell definition, but it does not satisfy the invariants—in particular, it is missing the third condition—, so the resulting tree is *not* guaranteed to be balanced. ♦

One possibility to include the color invariants in the type itself is given in [19]. The idea is to add two phantom types, one reflecting the color of each node and another simulating a natural number that keeps track of how many black nodes are encountered until the leaves:

```
data Color = Red | Black

data Nat where
   Z :: Nat
   S :: Nat -> Nat

data RBTree :: Type -> Color -> Nat -> Type where
  L  :: RBTree a Black Z
  TR :: RBTree a Black n -> a -> RBTree a Black  n -> RBTree a Red n
  TB :: RBTree a c1    n -> a -> RBTree a c2     n -> RBTree a Black (S n)
```

Notice that this type definition enforces all four constraints:

(a) The root being black is the only invariant that is not included explicitly in the type in this way—even if it could be, by adding a fourth constructor for generating exclusively the root. Still, that is not necessary because this condition can be checked easily by the type checker wherever a red-black tree is used, by requiring that it is of type `RBTree a Black n`.

(b) The leaves being black is reflected in the fact that `L`, the only constructor for leaves, only returns black trees.

(c) Red nodes having only black children is enforced because the only constructor targeting red nodes (`TR`) has two arguments of type `RBTree a Black n`.

(d) The fact that from each node, every path ending in a leaf passes through the same number of black nodes holds because (1) leafs have no children, so the condition holds vacuously for them, and (2) the constructors for internal nodes (`TR` and `TB`) both require that their children have the same number `n` of black nodes in each path to a leaf. In the case of `TR` (the constructor for *red* internal nodes), this number is preserved, while in the case of `TB` (the constructor for *black* internal nodes), it is increased in one unit.

Example 4.2.4 (Correct red-black tree)**.** As an example, consider the following value representing a red-black tree:

```
TB 2 (L 1) (TR 5 (L 4) (L 7))
```

The fact that this expression is well-typed guarantees that it encodes a *correct* red-black tree because of the stronger information stored in the type definition. In particular, it is necessarily a balanced tree, as can be seen by representing it graphically: ♦

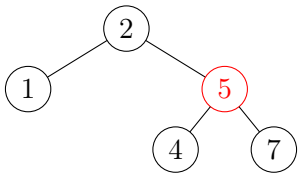

Figure 4.2: Graphical representation of the correct red-black tree

Note, however, how the incorrect red-black tree from 4.2.3 can *not* be encoded as a value of type `RBTree Int c n` for any color `c` and numeral `n`, since the red node containing the number 5 cannot be constructed from a black tree (the leaf containing 4) and a red tree (the subtree with a 7 in its root).

Chapter 5

Test case generators for GADTs

The goal of this chapter is to extend the generators described in Chapter 3 to be able to deal with GADTs and data type promotion, to add support for automatic generation of complex structures like those described in Chapter 4.

5.1 The supported syntax

At this second stage, our translation function accepts all of the syntactic constructs from the previous iteration, adding support for only two new ones: GADT-style constructors and promoted data types. More concretely, the entire supported grammar is presented in figure 5.1 below (cf. figure 3.1).

$$
\begin{array}{lll}
\langle Dec \rangle & ::= & \texttt{DataD}\ \langle Name \rangle\ \langle TyVar \rangle^*\ \langle Con \rangle^+ \\
\\
\langle Con \rangle & ::= & \texttt{NormalC}\ \langle Name \rangle\ \langle Type \rangle^* \\
 & | & \texttt{GadtC}\ \langle Name \rangle^+\ \langle Type \rangle^*\ \langle Type \rangle \\
\\
\langle Type \rangle & ::= & \texttt{VarT}\ \langle Name \rangle \\
 & | & \texttt{ConT}\ \langle Name \rangle \\
 & | & \texttt{AppT}\ \langle Type \rangle\ \langle Type \rangle \\
 & | & \texttt{ListT} \\
 & | & \texttt{TupleT}\ \langle Int \rangle \\
 & | & \texttt{PromotedT}\ \langle Name \rangle \\
\\
\langle TyVar \rangle & ::= & \texttt{PlainTV}\ \langle Name \rangle
\end{array}
$$

Figure 5.1: Supported syntax in the second stage

As in Chapter 3, in the following we will freely use a simplified notation to help readability. In the case of promoted types, we will follow Haskell's notation and precede the name of the value by an apostrophe to refer to the type associated with it (e.g. by promoting the value `Red` we get the type `'Red`). In the case of GADT constructors, we will use a similar notation as the one used before for normal constructors (those using the `NormalC` syntactic category), but stating explicitly the type of the generated value. Also, note that `GadtC` has a sequence of names—instead of only one like `NormalC` does—since GADT syntax allows to simultaneously define several data constructors as long as they have the same types as arguments and return value.

Example 5.1.1 (Typed expressions)**.** Recall the typed expressions from example 4.2.1:

```
data Expr t where
  I :: Int -> Expr Int
  Add, Mul :: Expr Int -> Expr Int -> Expr Int
  B :: Bool -> Expr Bool
  And, Or :: Expr Bool -> Expr Bool -> Expr Bool
  Eql :: Expr t -> Expr t -> Expr Bool
```

This data declaration can be encoded in our syntax as follows:

```
DataD Expr
      [t]
      [I :: Int -> Expr(Int),
       Add, Mul :: (Expr(Int), Expr(Int)) -> Expr(Int),
       B :: Bool -> Expr(Bool),
       And, Or :: (Expr(Bool), Expr(Bool)) -> Expr(Bool),
       Eql :: (Expr(t), Expr(t)) -> Expr(Bool)]
```

To be more explicit, let us take a closer look at, for example, the second element in the list of GADT constructions that are provided for the type `Expr`:

```
Add, Mul :: (Expr(Int), Expr(Int)) -> Expr(Int)
```

This corresponds to a `GadtC` value where:

(a) The list of names is `[Add, Mul]`.

(b) The list of argument types is `[Expr(Int), Expr(Int)]`.

(c) The return type is `Expr(Int)`.

Essentially, this is encoding the definition of two new data constructors `Add` and `Mul` that, given two expressions of type `Int`, return another expression of type `Int`.

♦

5.2 The new translation rules

5.2.1 First approach: adapting the old rules

The generation strategy implemented in the first stage works well when polymorphic types are only parameterised by type *variables*. As an example, polymorphic lists are defined in a generic way as `List a = Nil | Cons a (List a)`, where the type variable `a` can be instantiated with any type at all. More importantly, both of its constructors `Nil` and `Cons` are equally suitable for producing a value of type `List a` for any concrete type that is used in place of the variable `a`.

In contrast, the typed expressions from example 4.2.1 are only defined for two concrete type *constants*: `Int` and `Bool`. In other words, for *any* value of type `Expr t`, the type variable `t` is necessarily instantiated to either `Int` or `Bool`, since all of the constructors target either `Expr Int` or `Expr Bool` as the return type. Moreover, not all the constructors are valid in any of the two situations: some produce exclusively integer expressions while others can only be used to create boolean expressions.

These crucial differences between standard polymorphic types and GADTs mean that the previous approach is no longer applicable for dealing with the generalised types, without doing first a few modifications.

One option could be to maintain the same technique from Chapter 3 but including some preprocessing to classify all of the constructors according to the types they are able to generate. In the case of typed expressions, this would amount to grouping together the constructors `I`, `Add` and `Mul` as those that generate values of type `Expr Int`, and separately the constructors `B`, `And`, `Or` and `Eql` as those that generate values of type `Expr Bool`. Each of these classes of constructors would then be used in a different generation rule: one for generating integer expressions and another for generating boolean expressions.

Classifying the constructors in this way before producing the generators is viable for such a simple type, but it easily becomes unmanageable for more complex types with

several parameters that may be either variables or constants:

Example 5.2.1. Let us take a look at the red-black trees from Chapter 4:

```
data RBTree :: Type -> Color -> Nat -> Type where
  L  :: RBTree a Black Z
  TR :: RBTree a Black n -> a -> RBTree a Black  n -> RBTree a Red n
  TB :: RBTree a c1    n -> a -> RBTree a c2     n -> RBTree a Black (S n)
```

Notice that the first type parameter is always instantiated with a type variable `a`, so we can safely treat it as we did in the previous stage.

In contrast to what happens with typed variables, in this case it is not very clear how one could classify the constructors `L`, `TR` and `TB` according to the types they produce. As an example, to generate a red-black tree with a `Black` root, there are two applicable constructors: `L` and `TB`. The first only generates trees with a "black height" of zero, while the second only generates trees with a "black height" of at least one, so they produce *disjoint* sets of trees. As the generation targets *concrete* types—only parameterised with constants, not variables—, this separation of the two constructors means that we should provide two generation rules: one for black trees with a black height of zero and another for black trees with a black height of at least one. In a sense, generation rules should be targeted at constructing the most precise type possible wherever there are constants.

The situation is different for trees with a `Red` root. There is only one constructor `TR` targeting them, and the black height of a tree built with it is the same as that of its children: red nodes can have a black height of zero or more than zero, and all of them are constructed in the same way (with the same constructor). The fact that the "black height" of the constructed value is given by a variable means that we could give only one generation rule for that; generation rules should be as general as possible wherever the parameter is instantiated with variables.

Consider, however, what would happen if we added a fourth constructor for red leafs,

```
LR :: RBTree a Red Z
```

Now, red nodes can be built with an arbitrary black height (with the `TR` constructor), or with a black height of precisely zero (with `LR`). What this means for the purposes of automatic generation is that we now need *two* rules for building red trees, as was the case for black tres:

(a) One rule targets the case where the black height is at least one (using exclusively the `TR` constructor). For this generation rule to be compatible with the general strategy presented in Chapter 3, constructor `TR` should be refined to produce red trees of black height `S(n)`: since these are disjoint from those with a height of precisely zero, the generation rules should reflect that.

(b) The other rule targets the case where the black height is exactly zero (using constructors `LR` and `TR`). Again, constructor `TR` should be refined to produce only red trees of black height `Z`.

Thus, we arrive at the following two generation rules (using the notation from Chapter 3):

$$\begin{aligned}&\texttt{gen}(\texttt{rbtree}(\texttt{A},\texttt{red},\texttt{s}(\texttt{H})),\ \texttt{N},\ \texttt{X}) \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \\ &\quad \texttt{random\_member}(\texttt{C}, [\texttt{TR}(\texttt{rbtree}(\texttt{A},\texttt{black},\texttt{s}(\texttt{H})),\texttt{A},\texttt{rbtree}(\texttt{A},\texttt{black},\texttt{s}(\texttt{H})))]) \wedge \\ &\quad \texttt{create\_values}(\texttt{C},\texttt{M},\texttt{X})\end{aligned}$$

$$\begin{aligned}&\texttt{gen}(\texttt{rbtree}(\texttt{A},\texttt{red},\texttt{z}),\ \texttt{N},\ \texttt{X}) \leftarrow \texttt{N} \geq 1 \wedge \texttt{M} := \texttt{N} - 1 \wedge \\ &\quad \texttt{random\_member}(\texttt{C}, [\texttt{TR}(\texttt{rbtree}(\texttt{A},\texttt{black},\texttt{z}),\texttt{A},\texttt{rbtree}(\texttt{A},\texttt{black},\texttt{z})),\ \texttt{LR}]) \wedge \\ &\quad \texttt{create\_values}(\texttt{C},\texttt{M},\texttt{X})\end{aligned}$$

Note that these are all the generation rules needed to produce red nodes, and we would need two other rules for producing black ones. ♦

In conclusion, we could in theory use the same approach as in Chapter 3 for the generation of GADTs, but we would need to be careful in the translation phase to ensure that we are classifying the constructors correctly according to the type that they produce, all type constants are addressesd and type variables are used when there is no concrete constant appearing in that position. This would complicate the translation phase and the number of generation rules would grow with respect to the previous stage (where there were, at most, two generation rules for each type).

5.2.2 Second approach: the new strategy

Example 5.2.1 illustrates why it is desirable to depart from the technique presented in Chapter 3 in search for a simpler strategy, even if it is not strictly necessary.

Our new approach is based on pairing together, in the generation rules, each construction with all of the type parameters in the type of the values it generates (whether type

variables or type *constants*), and letting Prolog's unification algorithm determine which are applicable. Let us see how this pairing works in some concrete examples:

Example 5.2.2 (Expressions)**.** In the typed expressions defined as

```
data Expr t where
  I :: Int -> Expr Int
  Add, Mul :: Expr Int -> Expr Int -> Expr Int
  B :: Bool -> Expr Bool
  And, Or :: Expr Bool -> Expr Bool -> Expr Bool
  Eql :: Expr t -> Expr t -> Expr Bool
```

the defined type `Expr t` has a single type parameter. The following table shows all of the constructions together with the value for the type parameter each of them produces:

Type parameter	Construction
`Int`	`I(Int)`
`Int`	`Add(Expr(Int), Expr(Int))`
`Int`	`Mul(Expr(Int), Expr(Int))`
`Bool`	`B(Bool)`
`Bool`	`And(Expr(Bool), Expr(Bool))`
`Bool`	`Or(Expr(Bool), Expr(Bool))`
`Bool`	`Eql(Expr(t), Expr(t))`

Figure 5.2: Type parameters for `Expr`

Now, to determine if any particular construction can be used to generate values of type, for instance, `Expr Int`, it is enough to check if the type parameter can be unified with `Int`. Crucially, this unification is done automatically by Prolog, and it can be used uniformly for both type variables and constants. ♦

The previous example shows the basic idea for the new technique for the generation rules, but the type parameters are simple enough that determining if a given construction is applicable for some type parameter is straightforward. The usefulness of our new approach is better illustrated with a more intricate type definition:

Example 5.2.3 (Red-black trees)**.** In the definition of type-safe red-black trees presented in Chapter 4,

```
data RBTree :: Type -> Color -> Nat -> Type where
  L  :: RBTree a Black Z
```

```
TR :: RBTree a Black n -> a -> RBTree a Black  n -> RBTree a Red n
TB :: RBTree a c1    n -> a -> RBTree a c2     n -> RBTree a Black (S n)
```

the type `RBTree a c n` has three type parameters: the type that is stored inside, the color and the black height. The following table again shows the constructions together with the values for the type parameters each produces:

Type parameter	Construction
`a, Black, Z`	`L`
`a, Red, n`	`TR(RBTree(a, Black, n), a, RBTree(a, Black, n))`
`a, Black, S(n)`	`TB(RBTree(a, c1, n), a, RBTree(a, c2, n))`

Figure 5.3: Type parameters for `RBTree`

In this case, determining if a given construction is applicable for generating values of type, for instance, `RBTree a c Z` is reduced to checking if the list `[a, c, Z]` can be unified with the associated type parameter. In this particular example, it is easy to check that it can be unified with `[a, Black, Z]` and with `[a, Red, n]`, but not the third. Therefore, the first and second constructions are applicable, while the third one is not. ♦

The crucial aspect is that producing these pairings is straightforward, since it is only a syntactic transformation. The base for the generation rules in this new approach, therefore, is very easy to write directly from the type declaration; this contrasts with the complex generation rules that would be needed if we followed the strategy from § 5.2.1.

Not needing to classify the constructions according to their output type greatly simplifies the translation rules, as we have shown. However, having all of them grouped together complicates *choosing* a construction that is suitable for generating a value of the desired type. While, in the first stage, we could be sure that any construction for a type would be equally valid, and we could distinguish between "terminal" and "non-terminal" constructors to get values as large as possible, the generation in this second stage is not as straightforward and comes with a few limitations.

As we mentioned before, Prolog's unification is an effective method to automatically determine if a given construction is applicable for generating values of a specific type. Therefore, one easy approach for automatic generation is to choose one construction at random and automatically check if its associated type parameters are compatible with those in the target type. In case they are not, another construction is selected and the process continues.

However, even when a suitable construction has been selected, generation may still fail

because of the enforced sizes:

Example 5.2.4 (Sized expressions)**.** Let us consider once again the data type of type-safe expressions (example 5.2.2). If the user asks to generate an integer expression of size 1, then the constructor `Add` can be selected, since it creates a value of type `Expr Int`. Choosing this constructor requires generating both its arguments, which should be of size 0, but there is no way of generating a value of size 0. Thus, even if constructor `Add` was applicable from the *type* perspective, generation fails because it only generates values of size at least 2. ♦

Our new strategy, therefore, requires that generation is attempted indefinitely until a value is successfuly produced, even after having chosen a valid constructor. More concretely, the generation strategy is implemented in the following Prolog code: The target

```
gen(Type, N, Val) :- N >= 1, M is N-1,
                     Type =.. [BaseType | TypeArgs],
                     constructions(BaseType, Options),
                     (
                         repeat,
                         choose_construction(TypeArgs, Options, Cons),
                         create_values(Cons, M, Val), !
                     ).
```

Figure 5.4: Generic `gen/3` predicate for GADTs

type is split into the base type and the type arguments using operator `=..`, and then the list of all available constructions for this base type is placed in variable `Options`, using the pairings as shown in examples 5.2.2 and 5.2.3. Finally, a construction is chosen from the list of available options and it is used to generate the values; if this fails, another construction is chosen at random, and this continues until the values are successfully generated. The predicate `choose_constructions/3` is defined like this:

```
choose_construction(TypeArgs, Options, Choice) :-
  repeat, random_member((TypeArgs, Choice), Options), !.
```

Among all of the `Options`, a pairing is selected at random but trying to unify the first element of the pair with the desired type arguments; if this unification fails, another pairing is selected at random, and the process continues until a succesful unification takes place.

Importantly, both of these predicates (the version of `gen/3` for GADTs and `choose_construction`) are written to be completely independent of the value to be generated; therefore, they can be included verbatim in the Prolog code, reducing the translation rule

for producing the generators to the relatively simple step of creating the pairings to be "stored" in the `constructions` predicate. Consequently, the only translation rules needed for automatically creating GADT generators are the following:

$$\begin{aligned}&\text{parseGadtDec } [\![\texttt{DataD } typeName\ typeVars\ cons]\!] = \\ &\quad = \texttt{constructions}(\texttt{typeName},\ options) \\ &\qquad \text{where } options = \bigoplus_{con \in cons} \text{parseGadtCon } [\![con]\!]\end{aligned}$$

$$\begin{aligned}&\text{parseGadtCon } [\![\texttt{GadtC } names\ args\ baseType(typeArgs)]\!] = \\ &\quad = \bigoplus_{name \in names} [(\{typeArgs\}, \texttt{name}(\{args\}))]\end{aligned}$$

Figure 5.5: Translation rules for GADT generation

where $\oplus$ means list concatenation, e.g. $[1,2,3] \oplus [4,5] = [1,2,3,4,5]$. The Haskell code implementing these translation rules is presented in appendix A.3.

Let us see how these translation rules work in practical examples:

Example 5.2.5 (Typed expressions)**.** For simplicity, let us consider a subset of the constructors for the previously discussed expressions:

```
data Expr t where
  I :: Int -> Expr Int
  Add, Mul :: Expr Int -> Expr Int -> Expr Int
  Eql :: Expr t -> Expr t -> Expr Bool
```

In this reduced data declaration, the list of constructions contains three elements:

- `I :: Int -> Expr Int`, which is encoded in our syntax as `GadtC` $[I]$ $[Int]$ $Expr(Int)$. Rule "parseGadtCon" then gives

$$\bigoplus_{name \in [I]} [(\texttt{[int]},\ \texttt{name(int)})] = [(\texttt{[int]},\ \texttt{i(int)})].$$

- `Add, Mul :: Expr Int -> Expr Int -> Expr Int`, which is encoded as

$$\texttt{GadtC}\ [Add, Mul]\ [Expr(Int), Expr(Int)]\ Expr(Int).$$

Rule "parseGadtCon" then gives

$$\bigoplus_{name \in [Add, Mul]} [(\texttt{[int]},\ \texttt{name(expr(int), expr(int))})] =$$
$$= [(\texttt{[int]},\ \texttt{add(expr(int), expr(int))}),$$
$$(\texttt{[int]},\ \texttt{mul(expr(int), expr(int))})].$$

- `Eql :: Expr t -> Expr t -> Expr Bool`, which is encoded in our syntax as

$$\texttt{GadtC}\ [Eql]\ [Expr(t), Expr(t)]\ Expr(Bool).$$

Rule "parseGadtCon" then gives

$$\bigoplus_{name \in [Eql]} [(\texttt{[bool]},\ \texttt{name(expr(T), expr(T))})] = [(\texttt{[bool]},\ \texttt{eql(expr(T), expr(T))})].$$

Finally, grouping all three lists together, rule "parseGadtDec" gives

$$\texttt{constructions(expr}, [(\texttt{[int]}, \texttt{i(int)}),$$
$$(\texttt{[int]}, \texttt{add(expr(int), expr(int))}),$$
$$(\texttt{[int]}, \texttt{mul(expr(int), expr(int))}),$$
$$(\texttt{[bool]}, \texttt{eql(expr(T), expr(T))})])$$

Compare this list of constructions with the pairings listed in example 5.2.2, of which this is a subset.

If we wanted to generate a boolean expression of size 5, we would use a query like `gen(expr(bool), 5, E)`, as in the previous stage. Now, the `gen/3` predicate from figure 5.4 would be used, decomposing the type `expr(bool)` into the "base type" `expr` and the list of type arguments `[bool]` and unifying the variable `Options` with the above pairing. Then, the only option whose first element unifies with the desired type arguments `[bool]` is the last one, so it will be the only one that is chosen by predicate `choose_construction`. The values are then created recursively, as they were in the first stage.

In the Prolog interpreter, we may get the following answers to the query:

```
?- gen(expr(bool), 5, E).
E = eql(eql(add(add(i(8), i(7)), mul(i(10), i(3))), i(-1)),
        eql(eql(add(i(-6), i(6)), mul(i(10), i(-4))), eql(i(3), i(4)))).
```

```
?- gen(expr(bool), 5, E).
E = eql(i(5), i(-10)).
```

which correspond to the expressions represented in the following trees:

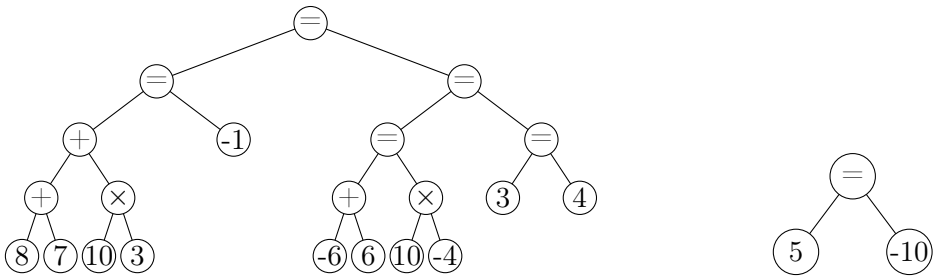

Figure 5.6: Graphical representation of the generated expressions

Notice that there is a very big variation in the sizes of the generated expressions, since we can no longer guarantee that non-terminal constructors are used whenever possible. Note as well that the equality operator can be used both with integers and with booleans, since the type variable `t` can be instantiated with any type whatsoever.

For completeness, below are presented two of the expressions that can be produced with the full generators, i.e. including the boolean operations:

```
?- gen(expr(bool), 5, E).
E = or(eql(mul(mul(i(-1), i(-1)), mul(i(-2), i(-5))), i(-2)),
       eql(add(i(9), add(i(-4), i(6))), mul(i(6), mul(i(9), i(3))))).

?- gen(expr(bool), 5, E).
E = or(b(true),
       eql(or(and(b(false), b(true)), eql(i(0), i(1))),
           or(eql(b(false), b(true)), or(b(false), b(false))))).
```

which can be represented graphically as in figure 5.7 below. ♦

Example 5.2.6 (Red-black trees)**.** Recall the definition of safe red-black trees, given by

```
data RBTree :: Type -> Color -> Nat -> Type where
  L  :: RBTree a Black Z
  TR :: RBTree a Black n -> a -> RBTree a Black  n -> RBTree a Red n
  TB :: RBTree a c1    n -> a -> RBTree a c2     n -> RBTree a Black (S n)
```

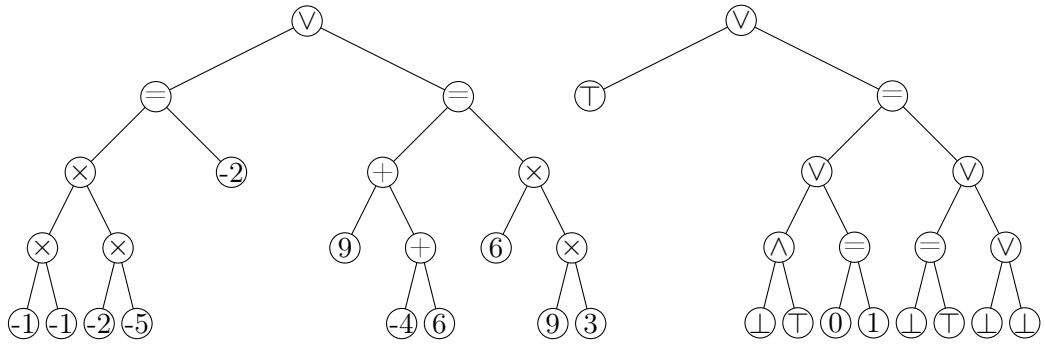

Figure 5.7: Graphical representation of the generated expressions

We have three constructors to deal with:

- `L :: RBTree a Black Z`, which is encoded in our syntax as

  $$\texttt{GadtC}\ [L]\ [\,]\ RBTree(a, Black, Z)$$

  Applying rule "parseGadtCon" then gives

  $$\bigoplus_{name \in [L]} [(\texttt{[A, black, z]},\ \texttt{name}())] = [(\texttt{[A, black, z]},\ \texttt{l}())]$$

- `TR :: RBTree a Black n -> a -> RBTree a Black n -> RBTree a Red n`, which is encoded as

  $$\texttt{GadtC}\ [TR]\ [RBTree(a, Black, n),\ a,\ RBTree(a, Black, n)]\ RBTree(a, Red, n)$$

  Applying rule "parseGadtCon" then gives

  $$\bigoplus_{name \in [TR]} [(\texttt{[A, red, N]},\ \texttt{name}(\texttt{rbtree}(\texttt{A}, \texttt{black}, \texttt{N}),\ \texttt{A},\ \texttt{rbtree}(\texttt{A}, \texttt{black}, \texttt{N})))] =$$
  $$= [(\texttt{[A, red, N]},\ \texttt{tr}(\texttt{rbtree}(\texttt{A}, \texttt{black}, \texttt{N}),\ \texttt{A},\ \texttt{rbtree}(\texttt{A}, \texttt{black}, \texttt{N})))]$$

- `TB :: RBTree a c1 n -> a -> RBTree a c2 n -> RBTree a Black (S n)`, which is encoded as

  $$\texttt{GadtC}\ [TB]\ [RBTree(a, c1, n),\ a,\ RBTree(a, c2, n)]\ RBTree(a, Black, S(n))$$

Applying "parseGadtCon" one last time gives

$$\bigoplus_{name \in [TB]} [([\texttt{A, black, s(N)}], \texttt{name}(\texttt{rbtree}(\texttt{A}, \texttt{C1}, \texttt{N}), \texttt{A}, \texttt{rbtree}(\texttt{A}, \texttt{C2}, \texttt{N})))] =$$
$$= [([\texttt{A, black, s(N)}], \texttt{tb}(\texttt{rbtree}(\texttt{A}, \texttt{C1}, \texttt{N}), \texttt{A}, \texttt{rbtree}(\texttt{A}, \texttt{C2}, \texttt{N})))]$$

The last step is to group all three together according to rule "parseGadtDec":

$$\begin{aligned}\texttt{constructions}(\texttt{expr}, [&([\texttt{A, black, z}], \texttt{1}()),\\ &([\texttt{A, red, N}], \texttt{tr}(\texttt{rbtree}(\texttt{A}, \texttt{black}, \texttt{N}), \texttt{A}, \texttt{rbtree}(\texttt{A}, \texttt{black}, \texttt{N}))),\\ &([\texttt{A, black, s(N)}], \texttt{tb}(\texttt{rbtree}(\texttt{A}, \texttt{C1}, \texttt{N}), \texttt{A}, \texttt{rbtree}(\texttt{A}, \texttt{C2}, \texttt{N})))])\end{aligned}$$

Compare these automatically-generated pairings with those from example 5.2.3.

Finally, below is presented a correct red-black tree (from the point of view of the color invariants and therefore the height-balancing) that can be automatically generated from this pairings:

```
?- gen(rbtree(int, black, s(s(s(z)))), 6, T).
T = tb(tb(tr(tb(tr(e, -9, e), -3, e), -5, tb(tr(e, 3, e), 9, e)),
          3,
          tb(e, 5, tr(e, 6, e))),
       2,
       tb(tr(tb(e, -3, e), 5, tb(tr(e, -1, e), 1, e)),
          8,
          tr(tb(tr(e, 2, e), -4, e), -3, tb(e, 9, tr(e, -7, e))))).
```

which can be represented graphically as in figure 5.8 below:

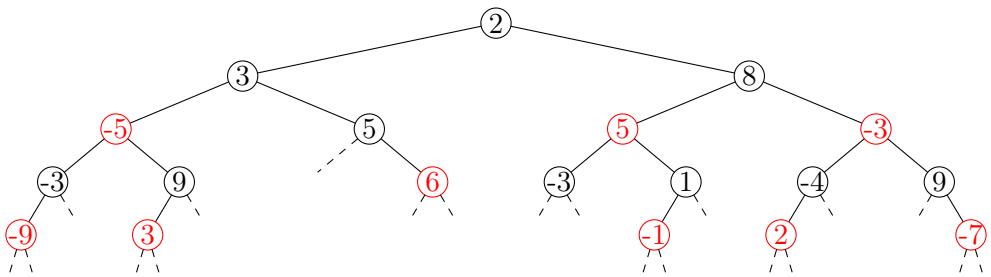

Figure 5.8: Graphical representation of the generated red-black tree

It is easy to see that the "black height" of the tree is indeed 3 (that is, `s(s(s(z)))`), as specified in the query.

Also note that the generated tree does have a height of 6 (which, in this case, coincides with the notion of *size* as defined in Chapter 3), as requested, because the leaves have a size of 1, even if they are not shown explicitly as nodes since they contain no values. ♦

Remark 5.2.1. The new generation strategy for GADTs is fully compatible with the previous one for standard Haskell types, both in the *translation phase* that creates the generators and in the *generation phase* itself:

The translation phase transforms a data declaration into the Prolog code that actually carries out the value generation. Since a data declaration cannot mix standard Haskell constructors with GADT-style constructors, there is no confusion as to what translation style to use for each particular data declaration.

The generation phase produces random values in response to a query, which is done through the `gen/3` predicate. As long as there are no two data types with the same name (which is ensured by the Haskell compiler), the old `gen/3` can be used for standard ADTs and the specialised `gen/3` can be used for GADTs; importantly, the user can run queries through this predicate in a completely uniform way, whether the type is defined using a GADT declaration or a standard Haskell one. ■

Limitations

Our approach for random value generation supporting GADTs relies heavily on the predicate `random_member/2`, both for *choosing* an applicable construction and for *backtracking*, i.e. choosing a different one if the generation failed because of the size restrictions. The use of this predicate greatly simplifies the generation, as illustrated by § 5.2.1, but it also has some disadvantages.

One issue is the fact that, since we are no longer splitting the constructions into "terminal" and "non-terminal", the generated values can be much smaller than what was requested in the query (see the expressions from example 5.2.5). The distinction between terminals and non-terminals helps avoid this problem by trying first the non-terminal constructions, i.e. those that have at least one argument, thus ensuring that the trees representing the structure of the value are as close to complete as possible (see example 3.2.10). By not being able to distinguish between terminals and non-terminals anymore, any given terminal is just as valid as any non-terminal, cutting the trees short even if they could still be extended.

A bigger problem with this approach is that some types have a small amount of values for a given size, in some cases none at all. For instance, there are no rose trees of size 2 (as

shown in example 3.2.8), and red-black trees of a height that is too "tight" are relatively sparse: consider as an example red-black trees with a black height of 3 and a size (height) of 4; there must be precisely one red node on each path from the root to a leaf to make up for the difference, so there are not many ways to randomly construct a red-black tree in a way that will satisfy this restriction.

Since our technique is to keep trying at random until a valid value can be generated, this relative scarcity—or absence—of valid values means that the generation phase can reach exponentially long average running times, and even enter into an infinite loop. This is what happens when trying to generate a rose tree of size 2 (provided the type of rose trees has been defined using GADT syntax) or when trying to generate a red-black tree with a black height of 3 and a size of 4 or even 5: the Prolog interpreter hangs—or seems to hang—without giving the user any kind of feedback.

As far as we know, the only way to fully address these issues is to bring the generation of GADTs closer to the one for standard Haskell types, as suggested in § 5.2.1, although this approach significantly complicates the translation function. A workaround while keeping the simple translation is to raise or lower the requested size whenever the generation seems to take a long time—it is possible to generate rose trees with a size value larger than 2, and it is easier to generate red-black trees if the height of the tree is farther from the black height, giving a bigger margin to the randomised generation. Still, this solution is far from ideal, so we leave the improvement of the worst-case performance as future work.

Chapter 6

Conclusions

The main goal of this project has been to design and implement an automatic test case generator for Haskell that requires as little user intervention as possible, with a special focus on producing data values with non-trivial properties, such as correctly colored red-black trees.

To achieve our goal, we started by implementing a completely automatic translation mechanism that transforms a Haskell data declaration into a Prolog program that is able to generate values of the specified type (Chapter 3). All this is accomplished while taking into account the size of the generated elements, thus ensuring that the produced test cases are neither too simple to help in detecting errors nor too large to be usable in practice.

In Chapter 4, we showed how *Generalised* Algebraic Data Types (GADTs) are more expressive than standard Haskell ones, especially in combination with advanced features like *data type promotion*, and how we can leverage their expressiveness to simulate *dependent types* to encode stronger properties directly in the type system. Finally, in Chapter 5 we extended our translation functions to deal with GADTs and demonstrated with two examples how our generators help in producing values with a more rigid structure.

6.1 Future work

Our implementation is working correctly for generating random values, but we acknowledge that it can be improved in, at least, four aspects:

In the first place, it currently does not support an automatic way of translating the random test cases, generated in Prolog, to Haskell values. We propose two main alternatives

for implementing this Prolog-to-Haskell value conversion:

(a) Writing a parser in Haskell to process Prolog structures and build Haskell values from them.

(b) Making every data declaration in Haskell automatically derive the `Read` type class, and writing a simple translator predicate in Prolog to convert a Prolog structure to a string that can be parsed by the automatically-created `read` Haskell function.

In the second place, our implementation cannot be regarded as a testing library because it handles only the test case generation. This could be solved by integrating our generation phase into QuickCheck or some other similar testing framework, or extending our implementation with testing capabilities.

Moreover, we rely heavily on the *simulation* of dependent types in a type system that does *not* support them natively. Because of this, the expressive power of the type system—even if extended beyond the capabilities of standard Hindley-Milner—complicates the encoding of many interesting properties.

Finally, the produced generators could be made more efficient through the use of *memoisation*—reusing already computed values—, following the paradigm of *Tabled Logic Programming* (TLP) [20]. In partice, this means that there would be a high degree of repetition in the generated test cases, which may be undesirable in some circumstances that require a more extensive exploration of the input space, but could nevertheless be useful when more performance is needed.

Bibliography

[1] INRIA. *The Coq proof assistant*. 2022. URL: https://coq.inria.fr/.

[2] Microsoft Research. *The Dafny programming and verification language*. http://dafny.org/. URL: http://dafny.org/.

[3] Koen Claessen and John Hughes. 'QuickCheck: a lightweight tool for random testing of Haskell programs'. In: *SIGPLAN Not.* 35.9 (Sept. 2000), pp. 268–279. ISSN: 0362-1340. DOI: 10.1145/357766.351266. URL: https://doi.org/10.1145/357766.351266.

[4] Koen Claessen, Björn Bringert and Nick Smallbone. *QuickCheck: Automatic testing of Haskell programs*. 2024. URL: https://hackage.haskell.org/package/QuickCheck (visited on 09/06/2024).

[5] David R. MacIver. *QuickCheck in every language*. 16th Apr. 2016. URL: https://hypothesis.works/articles/quickcheck-in-every-language/ (visited on 09/04/2024).

[6] Leonardo de Moura and Sebastian Ullrich. 'The Lean 4 Theorem Prover and Programming Language'. In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79876-5.

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

[8] Ulf Norell. 'Dependently typed programming in Agda'. In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*. TLDI '09. Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 1–2. ISBN: 9781605584201. DOI: 10.1145/1481861.1481862. URL: https://doi.org/10.1145/1481861.1481862.

[9] Niki Vazou et al. 'Refinement types for Haskell'. In: *SIGPLAN Not.* 49.9 (Aug. 2014), pp. 269–282. ISSN: 0362-1340. DOI: 10.1145/2692915.2628161. URL: https://doi.org/10.1145/2692915.2628161.

[10] Tim Sheard and Simon Peyton Jones. 'Template meta-programming for Haskell'. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2002, pp. 1–16. ISBN: 1581136056. DOI: 10.1145/581690.581691. URL: https://doi.org/10.1145/581690.581691.

[11] Joe "*begriffs*" Nelson. *The Design and Use of QuickCheck*. 14th Jan. 2017. URL: https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html (visited on 30/04/2024).

[12] John Hughes. *QuickCheck manual*. URL: https://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html (visited on 30/04/2024).

[13] GHC Team. *Template Haskell user's guide*. 2023. URL: https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/template_haskell.html (visited on 12/04/2024).

[14] *Template Haskell documentation*. URL: https://downloads.haskell.org/~ghc/9.6.4/docs/libraries/template-haskell-2.20.0.0/Language-Haskell-TH.html#g:18 (visited on 14/04/2024).

[15] Patrick Blackburn, Johan Bos and Kristina Striegnitz. *Learn Prolog Now!* 2012. URL: http://lpn.swi-prolog.org/ (visited on 22/04/2024).

[16] Wikibooks contributors. *Prolog*. 2020. URL: https://en.wikibooks.org/wiki/Prolog (visited on 22/04/2024).

[17] Ranjit Jhala. *Prolog tutorial*. 2009. URL: https://cseweb.ucsd.edu/classes/fa09/cse130/misc/prolog/prolog_tutorial.pdf (visited on 22/04/2024).

[18] Pablo Castellanos. *Automatic test-case generation for Haskell based on dependent types—GitLab repository*. 2024. URL: https://gitlab.com/pacastega/auto-test-gen (visited on 28/05/2024).

[19] Stephanie Weirich. *Depending on types*. 2014. URL: https://www.cis.upenn.edu/~sweirich/talks/icfp14.pdf (visited on 08/05/2024).

[20] Terrance Swift and David S. Warren. 'XSB: Extending Prolog with Tabled Logic Programming'. In: *Theory and Practice of Logic Programming* 12.1–2 (2012), pp. 157–187. DOI: 10.1017/S1471068411000500.

Appendix A

The full implementation

This appendix contains all the relevant code developed for the project, which can also be found in the public GitLab repository associated to the project [18]. This includes both the Haskell functions to translate data declarations into random generators in Prolog and the manually written generators for the basic types.

A.1 General auxiliary functions

The code below defines the general auxiliary functions that are used throughout the iterations:

```haskell
module Utils where

import Data.List (intercalate)
import Data.Char (toLower, toUpper, isAlpha, isDigit)
import Language.Haskell.TH (Name, nameBase)

lowerBase :: Name -> String
lowerBase = map toLower . nameBase

upperBase :: Name -> String
upperBase = map toUpper . nameBase

parens :: String -> String
parens s = "("  ++ s ++ ")"
```

```haskell

sqbrackets :: String -> String
sqbrackets s = "["  ++ s ++ "]"

mkPrologConst :: Name -> String
mkPrologConst = map aux . nameBase where
  aux :: Char -> Char
  aux c
    | isDigit c = c
    | isAlpha c = toLower c
    | otherwise = '_'

mkPrologVar :: Name -> String
mkPrologVar = map aux . nameBase where
  aux :: Char -> Char
  aux c
    | isDigit c = c
    | isAlpha c = toUpper c
    | otherwise = '_'

mkList :: [String] -> String
mkList = intercalate ", "

mkPair :: String -> String -> String
mkPair a b = parens $ a ++ ", " ++ b

mkTerm :: String -> [String] -> String
mkTerm name args = if null args then name else name ++ parens (mkList args)

freshVars :: String -> Int -> [String]
freshVars str n = map (\x -> str ++ (show x)) [1..n]
```

A.2 First stage: standard ADTs

This section contains the code implementing the translation rules for the three iterations in the first stage (described in Chapter 3), supporting only the standard Haskell data types.

A.2.1 First approach

Basic type generation (Prolog code)

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

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

gen(int, X) :- random_between(-9223372036854775808, 9223372036854775807, X).

gen(bool, false).
gen(bool, true).

gen(list__(A), []).
gen(list__(A), [X|XS]) :- gen(A, X), gen(list__(A), XS).

gen(tuple__([T]),    [X])    :- !, gen(T, X).
gen(tuple__([T|TS]), [X|XS]) :- gen(T, X), gen(tuple__(TS), XS).
gen(tuple__([]),     unit).
```

Translation function (Haskell code)

```
module Translation where

import Utils
import Language.Haskell.TH

parseDec :: Dec -> [String]
parseDec (DataD _ tyName tyVars _ cons _) = map (rule tyName tyVars) cons
parseDec _ = error "Only data declarations are supported"


generate :: String -> String -> String
generate typeName con = mkTerm "gen" [typeName, con]
```

```
13
14
15 rule :: Name -> [TyVarBndr flag] -> Con -> String
16 rule tyName tyVars (NormalC ctorName ctorTyParams) =
17   (generate dataType instantiatedCtor) ++ hypotheses ++ "."
18   where
19     -- the name of the new type constructor with type arguments
20     dataType = mkTerm (mkPrologConst tyName) arguments
21     arguments = map parseTV tyVars
22
23     -- the fully instantiated constructor
24     instantiatedCtor = mkTerm (mkPrologConst ctorName) ctorArgs
25     ctorArgs = freshVars (mkPrologVar ctorName) (length ctorTyParams)
26
27     -- every constructor argument must be of the declared type
28     hypotheses  = if null hyp then "" else " :- " ++ mkList hyp
29     hyp = zipWith genFn ctorTyParams ctorArgs
30     genFn tyParam arg = generate (parseBT tyParam) arg
31 rule _ _ _ = error "This constructor is still unsupported"
32
33
34 parseBT :: BangType -> String
35 parseBT = parseTy . snd
36
37
38 -- Recover the base type and its arguments
39 parseAppT :: Type -> (String, [String])
40 parseAppT t = parseAppT' t [] where
41   parseAppT' (AppT t1 t2) acc = parseAppT' t1 (parseTy t2:acc)
42   parseAppT' (VarT name)  acc = (mkPrologVar   name, acc)
43   parseAppT' (ConT name)  acc = (mkPrologConst name, acc)
44   parseAppT' (ListT)      acc = ("list__",           acc)
45   parseAppT' (TupleT _)   acc = ("tuple__",          acc)
46   parseAppT' _            _   = error "This type is still unsupported"
47
48
49 parseTy :: Type -> String
50 parseTy (VarT name)  = mkPrologVar name
51 parseTy (ConT name)  = mkPrologConst name
```

```
52 parseTy t@(AppT _ _) = case parseAppT t of
53   ("tuple__", args) -> "tuple__" ++ parens (sqbrackets (mkList args))
54   (base,      args) -> mkTerm base args
55 parseTy (ListT)      = "list__"
56 parseTy (TupleT _)   = "tuple__"
57 parseTy _            = error "This type is still unsupported"
58
59
60 parseTV :: TyVarBndr flag -> String
61 parseTV (PlainTV name _) = mkPrologVar name
62 parseTV (KindedTV name _ _) = mkPrologVar name
```

Example

To automatically create the test case generators, one can simply write the desired types inside a *declaration bracket*, written `[d| ... |]`. This produces a value of type `[Dec]` wrapped inside the `Q` monad; the value of type `[Dec]` can be passed to the `parseDec` function to produce a list of strings, which can then be concatenated and transformed into a Template Haskell expression (a value of type `Exp`) representing the string, all within the monadic context. Finally, the expression can be "taken out" of the monadic context by splicing it back into an actual Haskell string and used e.g. for printing into a file:

```
decs :: Q [Dec]
decs = [d|
        data Either' a b = Left' a | Right' b
        data BinTree a = BinEmpty | Bin (BinTree a) a (BinTree a)

        data Pair a b = Pair (a, b)

        data Color = Red | Black
        data RBTree a = RBEmpty | RB Color a (RBTree a) (RBTree a)

        data Trie a = Trie [Trie a] Bool a
        data Rose a = Rose a [Rose a]
        |]

parseDQ :: Q Exp
parseDQ = ListE . map (LitE . StringL . unlines . parseDec) <$> decs
```

The Prolog code produced by splicing `parseDQ` is presented below (formatted for typographical reasons):

```
gen(either_(A, B), left_(LEFT_1)) :- gen(A, LEFT_1).
gen(either_(A, B), right_(RIGHT_1)) :- gen(B, RIGHT_1).

gen(bintree(A), binempty).
gen(bintree(A), bin(BIN1, BIN2, BIN3)) :- gen(bintree(A), BIN1),
                                          gen(A, BIN2),
                                          gen(bintree(A), BIN3).

gen(pair(A, B), pair(PAIR1)) :- gen(tuple__([A, B]), PAIR1).

gen(color, red).
gen(color, black).

gen(rbtree(A), rbempty).
gen(rbtree(A), rb(RB1, RB2, RB3, RB4)) :- gen(color, RB1),
                                          gen(A, RB2),
                                          gen(rbtree(A), RB3),
                                          gen(rbtree(A), RB4).

gen(trie(A), trie(TRIE1, TRIE2, TRIE3)) :- gen(list__(trie(A)), TRIE1),
                                           gen(bool, TRIE2),
                                           gen(A, TRIE3).

gen(rose(A), rose(ROSE1, ROSE2)) :- gen(A, ROSE1),
                                    gen(list__(rose(A)), ROSE2).
```

A.2.2 Second approach: sized generation

Basic type generation (Prolog code)

```
1 gen_char(C) :- repeat, random_between(0, 7935, Code),
2                (char_code(C, Code), !; fail).
3 gen(char, _, C) :- gen_char(C).
4
5 gen(string, _, S) :- repeat, random_between(0, 99, N),
```

```
                    (length(L, N), maplist(gen_char, L),
                       atom_string(L, S), !; fail).

gen(int, _, X) :- random_between(-9223372036854775808, 9223372036854775807, X).

gen(bool, _, false).
gen(bool, _, true).

gen(list__(A), 0, []) :- !.
gen(list__(A), N__, [X|XS]) :- N__ >= 1, M__ is N__-1,
                                gen(A, M__, X), gen(list__(A), M__, XS), !.
gen(list__(A), N__, L) :- N__ >= 1, M__ is N__-1, gen(list__(A), M__, L), !.

gen(tuple__([T]),    N__, [X])    :- !, gen(T, N__, X).
gen(tuple__([T|TS]), N__, [X|XS]) :- gen(T, N__, X), gen(tuple__(TS), N__, XS).
gen(tuple__([]),     _, unit).
```

Translation function (Haskell code)

```
module Translation where

import Data.List (partition)
import Utils
import Language.Haskell.TH

parseDec :: Dec -> [String]
parseDec (DataD _ tyName tyVars _ cons _) =
  map (ruleNonTerm tyName tyVars) nonTerminals ++
  map (ruleTerm    tyName tyVars) terminals
  where
    (terminals, nonTerminals) = partition isTerminal cons
parseDec _ = error "Only data declarations are supported"


generate :: String -> String -> String -> String
generate tyName size con = mkTerm "gen" [tyName, size, con]


```

```haskell
-- Determine if a constructor is a terminal or not. Terminal constructors do not
-- have arguments, so they need no extra hypotheses to be generated.
isTerminal :: Con -> Bool
isTerminal (NormalC _ []) = True
isTerminal (NormalC _ _)  = False
isTerminal _              = error "This constructor is still unsupported"


ruleTerm :: Name -> [TyVarBndr flag] -> Con -> String
ruleTerm tyName tyVars (NormalC ctorName ctorTyParams) =
  generate dataType "_" instantiatedCtor ++ ":- !."
  where
    -- the name of the new type constructor with type arguments
    dataType = mkTerm (mkPrologConst tyName) arguments
    arguments = map parseTV tyVars

    -- the fully instantiated constructor
    instantiatedCtor = mkTerm (mkPrologConst ctorName) ctorArgs
    ctorArgs = freshVars (mkPrologVar ctorName) (length ctorTyParams)
ruleTerm _ _ _ = error "This constructor is still unsupported"


ruleNonTerm :: Name -> [TyVarBndr flag] -> Con -> String
ruleNonTerm tyName tyVars (NormalC ctorName ctorTyParams) =
  generate dataType "N__" instantiatedCtor ++ hypotheses
  where
    -- the name of the new type constructor with type arguments
    dataType = mkTerm (mkPrologConst tyName) arguments
    arguments = map parseTV tyVars

    -- the fully instantiated constructor
    instantiatedCtor = mkTerm (mkPrologConst ctorName) ctorArgs
    ctorArgs = freshVars (mkPrologVar ctorName) (length ctorTyParams)

    -- every constructor argument must be of the correct type and size
    hypotheses = " :- " ++ "N__ >= 1, M__ is N__-1,\n  " ++
      mkList hyp ++ ", " ++ "!."
    hyp = zipWith genFn ctorTyParams ctorArgs
    genFn tyParam arg = generate (parseBT tyParam) "M__" arg
```

```
59 ruleNonTerm _ _ _ = error "This constructor is still unsupported"
60
61
62 parseCon :: Con -> String
63 parseCon (NormalC name args) = mkTerm (mkPrologConst name) (map parseBT args)
64 parseCon _ = error "This constructor is still unsupported"
65
66
67 parseBT :: BangType -> String
68 parseBT = parseTy . snd
69
70
71 -- Recover the base type and its arguments
72 parseAppT :: Type -> (String, [String])
73 parseAppT t = parseAppT' t [] where
74   parseAppT' (AppT t1 t2) acc = parseAppT' t1 (parseTy t2:acc)
75   parseAppT' (VarT name)  acc = (mkPrologVar   name, acc)
76   parseAppT' (ConT name)  acc = (mkPrologConst name, acc)
77   parseAppT' (ListT)      acc = ("list__",           acc)
78   parseAppT' (TupleT _)   acc = ("tuple__",          acc)
79   parseAppT' _            _   = error "This type is still unsupported"
80
81
82 parseTy :: Type -> String
83 parseTy (VarT name)  = mkPrologVar name
84 parseTy (ConT name)  = mkPrologConst name
85 parseTy t@(AppT _ _) = case parseAppT t of
86   ("tuple__", args) -> "tuple__" ++ parens (sqbrackets (mkList args))
87   (base,      args) -> mkTerm base args
88 parseTy (ListT)      = "list__"
89 parseTy (TupleT _)   = "tuple__"
90 parseTy _            = error "This type is still unsupported"
91
92
93 parseTV :: TyVarBndr flag -> String
94 parseTV (PlainTV name _) = mkPrologVar name
95 parseTV (KindedTV name _ _) = mkPrologVar name
```

Example

The declarations from before can now be processed and spliced into the following Prolog code (which, again, has been slightly formatted):

```
gen(either_(A, B), N__, left_(LEFT_1)) :- N__ >= 1, M__ is N__-1,
  gen(A, M__, LEFT_1), !.
gen(either_(A, B), N__, right_(RIGHT_1)) :- N__ >= 1, M__ is N__-1,
  gen(B, M__, RIGHT_1), !.

gen(bintree(A), N__, bin(BIN1, BIN2, BIN3)) :- N__ >= 1, M__ is N__-1,
  gen(bintree(A), M__, BIN1), gen(A, M__, BIN2), gen(bintree(A), M__, BIN3), !.
gen(bintree(A), _, binempty):- !.

gen(pair(A, B), N__, pair(PAIR1)) :- N__ >= 1, M__ is N__-1,
  gen(tuple__([A, B]), M__, PAIR1), !.

gen(color, _, red):- !.
gen(color, _, black):- !.

gen(rbtree(A), N__, rb(RB1, RB2, RB3, RB4)) :- N__ >= 1, M__ is N__-1,
  gen(color, M__, RB1),
  gen(A, M__, RB2), gen(rbtree(A), M__, RB3), gen(rbtree(A), M__, RB4), !.
gen(rbtree(A), _, rbempty):- !.

gen(trie(A), N__, trie(TRIE1, TRIE2, TRIE3)) :- N__ >= 1, M__ is N__-1,
  gen(list__(trie(A)), M__, TRIE1), gen(bool, M__, TRIE2), gen(A, M__, TRIE3), !.

gen(rose(A), N__, rose(ROSE1, ROSE2)) :- N__ >= 1, M__ is N__-1,
  gen(A, M__, ROSE1), gen(list__(rose(A)), M__, ROSE2), !.
```

A.2.3 Third approach: uniform generation

Basic type generation (Prolog code)

```
1 create_values(Con, N, X) :- Con =.. [Ctor|Args],
2                             gen(all__(Args), N, Values),
3                             X =.. [Ctor|Values].
```

```
4
5  gen_char(C) :- repeat, random_between(0, 7935, Code),
6                 (char_code(C, Code), !; fail).
7  gen(char, _, C) :- gen_char(C).
8  gen(string, _, S) :- repeat, random_between(0, 99, N),
9                       (length(L, N), maplist(gen_char, L),
10                         atom_string(L, S), !; fail).
11
12 gen(int, _, X) :- random_between(-9223372036854775808, 9223372036854775807, X).
13
14 gen(bool, _, B) :- random_member(B, [true, false]).
15
16 gen(list__(A), N, [X|XS]) :- N >= 1, M is N-1,
17                               gen(A, M, X), gen(list__(A), M, XS), !.
18 gen(list__(A), _, []) :- !.
19 gen(tuple__(TS), N, XS) :- N >= 1, M is N-1,
20                             gen(all__(TS), M, XS), !.
21 gen(tuple__([]), _, unit).
22
23 gen(all__([T]),    N, [X]) :- !, gen(T, N, X).
24 gen(all__([T|TS]), N, [X|XS]) :- gen(T, N, X), gen(all__(TS), N, XS).
```

Translation function (Haskell code)

```
1  module Translation where
2
3  import Data.List (partition)
4  import Utils
5  import Language.Haskell.TH
6
7  parseDec :: Dec -> String
8  parseDec (DataD _ tyName tyVars _ ctors _) = unlines $
9    [ruleNonTerm typeStr nonTerminals, ruleTerm typeStr terminals]
10     where
11       (terminals, nonTerminals) = partition isTerminal ctors
12       typeStr = mkTerm (mkPrologConst tyName) (map parseTV tyVars)
13 parseDec _ = error "Only data declarations are supported"
14
```

```
15
16 generate :: String -> String -> String -> String
17 generate tyName size varName = mkTerm "gen" [tyName, size, varName]
18
19
20 -- Determine if a constructor is a terminal or not. Terminal constructors do not
21 -- have arguments, so they need no extra hypotheses to be generated.
22 isTerminal :: Con -> Bool
23 isTerminal (NormalC _ []) = True
24 isTerminal (NormalC _ _)  = False
25 isTerminal _              = error "This constructor is still unsupported"
26
27
28 ruleTerm :: String -> [Con] -> String
29 ruleTerm _       []    = "%% [NO TERMINALS]"
30 ruleTerm typeStr ctors =
31   "%% Terminals:\n" ++
32   generate typeStr "_" "X" ++ " :- " ++
33   "random_member(X, " ++ sqbrackets (mkList $ map parseCon ctors) ++ "), !."
34
35
36 ruleNonTerm :: String -> [Con] -> String
37 ruleNonTerm _       []    = "%% [NO NON-TERMINALS]"
38 ruleNonTerm typeStr ctors =
39   "%% Non-terminals:\n" ++
40   generate typeStr "N__" "X__" ++ " :- " ++
41   "N__ >= 1, M__ is N__-1,\n  " ++
42   "random_member(C__, " ++ sqbrackets (mkList $ map parseCon ctors) ++ "),\n" ++
43   "  create_values(C__, M__, X__), !."
44
45
46 parseCon :: Con -> String
47 parseCon (NormalC name args) = mkTerm (mkPrologConst name) (map parseBT args)
48 parseCon _ = error "This constructor is still unsupported"
49
50
51 parseBT :: BangType -> String
52 parseBT = parseTy . snd
53
```

```haskell

-- Recover the base type and its arguments
parseAppT :: Type -> (String, [String])
parseAppT t = parseAppT' t [] where
  parseAppT' (AppT t1 t2) acc = parseAppT' t1 (parseTy t2:acc)
  parseAppT' (VarT name)  acc = (mkPrologVar   name, acc)
  parseAppT' (ConT name)  acc = (mkPrologConst name, acc)
  parseAppT' (ListT)      acc = ("list__",           acc)
  parseAppT' (TupleT _)   acc = ("tuple__",          acc)
  parseAppT' _            _   = error "This type is still unsupported"


parseTy :: Type -> String
parseTy (VarT name)  = mkPrologVar name
parseTy (ConT name)  = mkPrologConst name
parseTy t@(AppT _ _) = case parseAppT t of
  ("tuple__", args) -> "tuple__" ++ parens (sqbrackets (mkList args))
  (base,      args) -> mkTerm base args
parseTy (ListT)      = "list__"
parseTy (TupleT _)   = "tuple__"
parseTy _            = error "This type is still unsupported"


parseTV :: TyVarBndr flag -> String
parseTV (PlainTV name _) = mkPrologVar name
parseTV (KindedTV name _ _) = mkPrologVar name
```

Example

The same declarations now produce the following Prolog code:

```prolog
%% Non-terminals:
gen(either_(A, B), N__, X__) :- N__ >= 1, M__ is N__-1,
  random_member(C__, [left_(A), right_(B)]),
  create_values(C__, M__, X__), !.
%% [NO TERMINALS]

%% Non-terminals:
```

```
gen(bintree(A), N__, X__) :- N__ >= 1, M__ is N__-1,
  random_member(C__, [bin(bintree(A), A, bintree(A))]),
  create_values(C__, M__, X__), !.
%% Terminals:
gen(bintree(A), _, X) :- random_member(X, [binempty]), !.

%% Non-terminals:
gen(pair(A, B), N__, X__) :- N__ >= 1, M__ is N__-1,
  random_member(C__, [pair(tuple__([A, B]))]),
  create_values(C__, M__, X__), !.
%% [NO TERMINALS]

%% [NO NON-TERMINALS]
%% Terminals:
gen(color, _, X) :- random_member(X, [red, black]), !.

%% Non-terminals:
gen(rbtree(A), N__, X__) :- N__ >= 1, M__ is N__-1,
  random_member(C__, [rb(color, A, rbtree(A), rbtree(A))]),
  create_values(C__, M__, X__), !.
%% Terminals:
gen(rbtree(A), _, X) :- random_member(X, [rbempty]), !.

%% Non-terminals:
gen(trie(A), N__, X__) :- N__ >= 1, M__ is N__-1,
  random_member(C__, [trie(list__(trie(A)), bool, A)]),
  create_values(C__, M__, X__), !.
%% [NO TERMINALS]

%% Non-terminals:
gen(rose(A), N__, X__) :- N__ >= 1, M__ is N__-1,
  random_member(C__, [rose(A, list__(rose(A)))]),
  create_values(C__, M__, X__), !.
%% [NO TERMINALS]
```

A.3 Second stage: GADTs

This second section contains the code from A.2.3 extended with the basic generators and the translation rules implemented in the second stage (described in Chapter 5).

Basic type generation (Prolog code)

```
create_values(Con, N, X) :- Con =.. [Ctor|Args],
                            gen(all__(Args), N, Values),
                            X =.. [Ctor|Values].

choose_construction(TypeArgs, Options, Choice) :-
  repeat, random_member((TypeArgs, Choice), Options), !.

gen_char(C) :- repeat, random_between(0, 7935, Code),
               (char_code(C, Code), !; fail).
gen(char, _, C) :- gen_char(C).
gen(string, _, S) :- repeat, random_between(0, 99, N),
                     (length(L, N), maplist(gen_char, L),
                        atom_string(L, S), !; fail).

gen(int, _, X) :- random_between(-9223372036854775808, 9223372036854775807, X).

gen(bool, _, B) :- random_member(B, [true, false]).

gen(list__(A), N, [X|XS]) :- N >= 1, M is N-1,
                             gen(A, M, X), gen(list__(A), M, XS), !.
gen(list__(A), _, []) :- !.
gen(tuple__(TS), N, XS) :- N >= 1, M is N-1,
                           gen(all__(TS), M, XS), !.
gen(tuple__([]), _, unit).

gen(all__([]),     _, []) :- !.
gen(all__([T|TS]), N, [X|XS]) :- gen(T, N, X), gen(all__(TS), N, XS).

gen(Type, N__, Val) :- N__ >= 1, M__ is N__-1,
                       Type =.. [BaseType | TypeArgs],
                       constructions(BaseType, Options),
```

```
                    (
                        repeat,
                        choose_construction(TypeArgs, Options, Cons),
                        create_values(Cons, M__, Val), !
                    ).
```

Translation function (Haskell code)

```haskell
{-# LANGUAGE DataKinds #-}
module Translation where

import Data.List (partition)
import Utils
import Language.Haskell.TH

parseDec :: Dec -> String
parseDec (DataD _ tyName tyVars _ ctors _)
  | all gadtCon ctors = let options = sqbrackets (mkList $ map parseCon ctors)
    in constructions (mkPrologConst tyName) options
  | otherwise = let (terminals, nonTerminals) = partition isTerminal ctors
                    typeStr = mkTerm (mkPrologConst tyName) (map parseTV tyVars)
    in unlines [ruleNonTerm typeStr nonTerminals, ruleTerm typeStr terminals]
parseDec _ = error "Only data declarations are supported"


sort :: [Dec] -> [Dec]
sort ds = normal ++ gadt where
  (gadt, normal) = partition isGadtDec (filter isDataDec ds)

  isDataDec :: Dec -> Bool
  isDataDec (DataD {}) = True
  isDataDec _          = False

  isGadtDec :: Dec -> Bool
  isGadtDec (DataD _ _ _ _ ctors _) = any gadtCon ctors
  isGadtDec _ = error "Only data declarations are supported"

gadtCon :: Con -> Bool
```

```
31 gadtCon (NormalC _ _) = False
32 gadtCon (GadtC _ _ _) = True
33 gadtCon _             = error "This constructor is still unsupported"
34
35
36 generate :: String -> String -> String -> String
37 generate tyName size varName = mkTerm "gen" [tyName, size, varName]
38
39
40 constructions :: String -> String -> String
41 constructions tyName options = mkTerm "constructions" [tyName, options] ++ ".\n"
42
43
44 -- Determine if a constructor is a terminal or not. Terminal constructors do not
45 -- have arguments, so they need no extra hypotheses to be generated.
46 isTerminal :: Con -> Bool
47 isTerminal (NormalC _ []) = True
48 isTerminal (NormalC _ _)  = False
49 isTerminal _              = error "This constructor is still unsupported"
50
51
52 ruleTerm :: String -> [Con] -> String
53 ruleTerm _       []    = "%% [NO TERMINALS]"
54 ruleTerm typeStr ctors =
55   "%% Terminals:\n" ++
56   generate typeStr "_" "X" ++ " :- " ++
57   "random_member(X, " ++ sqbrackets (mkList $ map parseCon ctors) ++ "), !."
58
59
60 ruleNonTerm :: String -> [Con] -> String
61 ruleNonTerm _       []    = "%% [NO NON-TERMINALS]"
62 ruleNonTerm typeStr ctors =
63   "%% Non-terminals:\n" ++
64   generate typeStr "N__" "X__" ++ " :- " ++
65   "N__ >= 1, M__ is N__-1,\n  " ++
66   "random_member(C__, " ++ sqbrackets (mkList $ map parseCon ctors) ++ "),\n" ++
67   "  create_values(C__, M__, X__), !."
68
69
```

```
70  parseCon :: Con -> String
71  parseCon (NormalC name args) = mkTerm (mkPrologConst name) (map parseBT args)
72  parseCon (GadtC names args ty) = mkList $ map processCtor names where
73    (_, targetTypes) = parseAppT ty
74    processCtor name = mkPair (sqbrackets $ mkList targetTypes)
75                              (mkTerm (mkPrologConst name) (map parseBT args))
76  parseCon _ = error "This constructor is still unsupported"
77
78
79  parseBT :: BangType -> String
80  parseBT = parseTy . snd
81
82
83  -- Recover the base type and its arguments
84  parseAppT :: Type -> (String, [String])
85  parseAppT t = parseAppT' t [] where
86    parseAppT' (AppT t1 t2)     acc = parseAppT' t1 (parseTy t2:acc)
87    parseAppT' (VarT name)      acc = (mkPrologVar   name, acc)
88    parseAppT' (ConT name)      acc = (mkPrologConst name, acc)
89    parseAppT' (PromotedT name) acc = (mkPrologConst name, acc)
90    parseAppT' (ListT)          acc = ("list__",           acc)
91    parseAppT' (TupleT _)       acc = ("tuple__",          acc)
92    parseAppT' t'               _   = error $
93      "Type " ++ show t' ++ " is still unsupported"
94
95
96  parseTy :: Type -> String
97  parseTy (VarT name)      = mkPrologVar name
98  parseTy (ConT name)      = mkPrologConst name
99  parseTy (PromotedT name) = mkPrologConst name
100 parseTy t@(AppT _ _)     = case parseAppT t of
101   ("tuple__", args) -> "tuple__" ++ parens (sqbrackets (mkList args))
102   (base,      args) -> mkTerm base args
103 parseTy (ListT)          = "list__"
104 parseTy (TupleT _)       = "tuple__"
105 parseTy t               = error $ "Type " ++ show t ++ " is unsupported"
106
107
108 parseTV :: TyVarBndr flag -> String
```

```
109 parseTV (PlainTV name _) = mkPrologVar name
110 parseTV (KindedTV name _ _) = mkPrologVar name
```

Example

In addition to the previous data declaration, we include now the following GADT declarations:

```
data Nat where
   Z :: Nat
   S :: Nat -> Nat

data RBTree :: Type -> Color -> Nat -> Type where
  E  :: RBTree a Black Z
  TR :: RBTree a Black n -> a -> RBTree a Black  n -> RBTree a Red n
  TB :: RBTree a c1    n -> a -> RBTree a c2     n -> RBTree a Black (S n)

data Expr t where
   I :: Int -> Expr Int
   B :: Bool -> Expr Bool
   Add, Sub, Mul :: Expr Int -> Expr Int -> Expr Int
   Eql :: Expr t -> Expr t -> Expr Bool
   Leq, Geq :: Expr Int -> Expr Int -> Expr Bool
   Not :: Expr Bool -> Expr Bool
   And, Or :: Expr Bool -> Expr Bool -> Expr Bool
```

These are automatically transformed into the following Prolog generators; all the previous generators, except for those for the type `color`, are omitted for brevity, although they can coexist with them:

```
%% [NO NON-TERMINALS]
%% Terminals:
gen(color, _, X) :- random_member(X, [red, black]), !.

constructions(nat, [([], z),
                    ([], s(nat))]).
```

```
constructions(rbtree, [([A, black, z], e),
                       ([A, red, N], tr(rbtree(A, black, N), A, rbtree(A, black, N))),
                       ([A, black, s(N)], tb(rbtree(A, C1, N), A, rbtree(A, C2, N)))]).

constructions(expr, [([int], i(int)),
                     ([bool], b(bool)),
                     ([int], add(expr(int), expr(int))),
                     ([int], sub(expr(int), expr(int))),
                     ([int], mul(expr(int), expr(int))),
                     ([bool], eql(expr(T), expr(T))),
                     ([bool], leq(expr(int), expr(int))),
                     ([bool], geq(expr(int), expr(int))),
                     ([bool], not(expr(bool))),
                     ([bool], and(expr(bool), expr(bool))),
                     ([bool], or(expr(bool), expr(bool)))]).
```