Universiteit Utrecht

Faculty of Science
Dept. of Information and Computing Sciences

Thesis title

Author

C.R. van der Rest

Supervisor

Dr. W.S. Swierstra
Dr. M.M.T. Chakravarty
Dr. A. Serrano Mena

June 6, 2019

Contents

Declaration

Thanks to family, supervisor, friends and hops!

Abstract

Abstract

1 Introduction

2 Background

In this section, we will briefly discuss some of the relevant theoretical background for this thesis. We assume the reader to be familiar with the general concepts of both Haskell and Agda, as well as functional programming in general. We shortly touch upon the following subjects:

- *Type theory* and its relationship with *classical logic* through the *Curry-Howard correspondence*
- Some of the more advanced features of the programming language *Agda*, which we use for the formalization of our ideas: *Codata*, *Sized Types* and *Universe Polymorphism.*
- *Datatype generic programming* using *type universes* and the design patterns associated with datatype generic programming.

We present this section as a courtesy to those readers who might not be familiar with these topics; anyone experienced in these areas should feel free to skip ahead.

2.1 Type Theory

Type theory is the mathematical foundation that underlies the *type systems* of many modern programming languages. In type theory, we reason about terms and their *types*. We briefly introduce some basic concepts, and show how they relate to our proofs in Agda.

2.1.1 Intuitionistic Type Theory

In Intuitionistic type theory consists of terms, types and judgements $a : A$ stating that terms have a certain type. Generally we have the following two finite constructions: $\mathbb{0}$ or the *empty type*, containing no terms, and $\mathbb{1}$ or the *unit type* which contains exactly 1 term. Additionally, the *equality type*, $=$, captures the notion of equality for both terms and types. The equalit type is constructed from *reflexivity*, i.e. it is inhabited by one term $refl$ of the type $a = a$.

Types may be combined using three constructions. The *function type*, $a \to b$ is inhabited by functions that take an element of type a as input and produce something of type b. The *sum type*, $a + b$ creates a type that is inhabited by *either* a value of type a *or* a value of type b. The *product type*, $a * b$, is inhabited by a pair of values, one of type a and one of type b. In terms of set theory, these operations correspond respectively to functions, *cartesian product* and *tagged union.*

2.1.2 The Curry-Howard Equivalence

The *Curry-Howard equivalence* establishes an isomorphism between *propositions and types* and *proofs and terms* [?]. This means that for any type there is a corresponding proposition, and any

Classical Logic	Type Theory
False	$\bot$
True	$\top$
$P \vee Q$	$P + Q$
$P \wedge Q$	$P * Q$
$p \Rightarrow Q$	$P \rightarrow Q$

Table 2.1: Correspondence between classical logic and type theory

term inhabiting this type corresponds to a proof of the associated proposition. Types and propositions are generally connected using the mapping shown in section 2.1.2.

Example We can model the proposition $P \wedge (Q \vee R) \Rightarrow (P \wedge Q) \vee (P \wedge R)$ as a function with the following type:

```
tautology : ∀ {P Q R} → P * (Q + R) → (P * Q) + (P * R)
```

We can then prove that this implication holds on any proposition by supplying a definition that inhabits the above type:

```
tautology (fst , inj₁ x) = inj₁ (fst , x)
tautology (fst , inj₂ y) = inj₂ (fst , y)
```

In general, we may prove any proposition that captured as a type by writing a programin that inhabits that type. Allmost all constructs are readily translatable from proposition logic, except boolean negation, for which there is no corresponding construction in type theory. Instead, we model negation using functions to the empty type $\bot$. That is, we can prove a property P to be false by writing a function $P \rightarrow \bot$. This essentially says that P is true, we can derive a contradiction, hence it must be false. Alowing us to prove many properties including negation.

Example For example, we might prove that a property cannot be both true and false, i.e. $\forall P . \neg(P \wedge \neg P)$:

```
P∧¬P→⊥ : ∀ {P} → P * (P → ⊥) → ⊥
P∧¬P→⊥ (P , P→⊥) = P→⊥ P
```

However, there are properties of classical logic which do not carry over well through the Curry-Howard isomorphism. A good example of this is the *law of excluded middle*, which cannot be proven in type theory:

```
P∨¬P : ∀ {P} → P + ¬ P
```

This implies that type theory is incomplete as a proof system, in the sense that there exist properties wich we cannot prove, nor disprove.

2.1.3 Dependent Types

Dependent type theory allows the definition of types that depend on values. In addition to the constructs introduced above, one can use so-called Π-types and Σ-types. Π-types capture the idea of *dependent function types*, that is, functions whose output type may depend on the values of its input. Given some type A and a family P of types indexed by values of type A (i.e. P has type $A \to Type$), Π-types have the following form:

$$\Pi_{(x:A)}P(x) \equiv (x : A) \to P(x)$$

In a similar spirit, Σ-types are ordered *pairs* of which the type of the second value may depend on te first value of the pair:

$$\Sigma_{(x:A)}P(x) \equiv (x : A) \times P(x)$$

The Curry-Howard equivalence extends to Π- and Σ-types as well: they can be used to model universal and existential quantification [?] (??).

Classical Logic	Type Theory
$\exists\ x\ .\ P\ x$	$\Sigma_{(x:A)}P(x)$
$\forall\ x\ .\ P\ x$	$\Pi_{(x:A)}P(x)$

Table 2.2: Correspondence between quantifiers in classical logic and type theory

Example we might capture the relation between universal and negated existential quantification ($\forall\ x\ .\ \neg P\ x \Rightarrow \neg\exists\ x\ .\ P\ x$) as follows:

```
∀¬⟶¬∃ : ∀ {P} ⟶ ((x : Set) ⟶ P x ⟶ ⊥) ⟶ Σ Set P ⟶ ⊥
∀¬⟶¬∃ ∀x¬P (x , Px) = ∀x¬P x Px
```

The correspondence between dependent pairs and existential quantification quite beautifullly illustrates the constructive nature of proofs in type theory; we prove any existential property by presenting a term together with a proof that the required property holds for that term.

2.2 Agda

Agda is a programming language based on Intuitionistic type theory[?]. Its syntax is broadly similar to Haskell's, though Agda's type system is arguably more expressive, since types may depend on term level values.

Due to the aforementioned correspondence between types and propositions, any Agda program we write is simultaneously a proof of the proposition associated with its type. Through this mechanism, Agda serves a dual purpose as a proof assistent.

2.2.1 Codata and Sized Types

All definitions in Agda are required to be *total*, meaning that they must be defined on all possible inputs, produce a result in finite time. To enforce this requirement, Agda needs to check whether the definitions we provide are terminating. As stated by the *Halting Problem*, it is not possible to define a general procedure to perform this check. Instead, Agda uses a *sound approximation*, in which it requires at least one argument of any recursive call to be *syntactically smaller* than its corresponding function argument. A consequence of this approach is that there are Agda programs that terminate, but are rejected by the termination checker. This means that we cannot work with infinite data in the same way as in the same way as in Haskell, which does not care about termination.

Example The following definition is perfectly fine in Haskell:

```
nats :: [Int]
nats = 0 : map (+1) nats
```

Meanwhile, an equivalent definition in Agda gets rejected by the Termination checker. The recursive call to *nats* has no arguments, so none are structurally smaller, thus the termination checker flags this call.

```
nats : List ℕ
nats = 0 :: map suc nats
```

However, as long as we use *nats* sensibly, there does not need to be a problem. Nonterminating programs only arise with improper use of such a definition, for example by calculating the length of *nats*. We can prevent the termination checker from flagging these kind of operations by making the lazy semantics explicit, using *codata* and sized types. Codata is a general term for possible inifinite data, often described by a co-recursive definition. Sized types extend the space of function definitions that are recognized by the termination checker as terminating by tracking information about the size of values in types [?]. In the case of lists, this means that we explicitly specify that the recursive argument to the _ :: _ constructor is a *Thunk*, which should only be evaluated when needed:

```
data Colist (A : Set) (i : Size) : Set where
  [] : Colist A i
  _::_ : A → Thunk (Colist A) i → Colist A i
```

We can now define *nats* in Agda by wrapping the recursive call in a thunk, explicitly marking that it is not to be evaluated until needed.

```
nats : ∀ {i : Size} → Colist ℕ i
nats = 0 :: λ where .force → map suc nats
```

Since colists are possible infinite structures, there are some functions we can define on lists, but not on colists.

Example Consider a function that attempts to calculate the length of a *Colist*:

```
length : ∀ {a : Set} → Colist a ∞ → ℕ
length [] = 0
length (x :: xs) = suc (length (xs .force))
```

In this case *length* is not accepted by the termination checker because the input colist is indexed with size ∞, meaning that there is no finite upper bound on its size. Hence, there is no guarantee that our function terminates when inductively defined on the input colist.

There are some cases in which we can convince the termination checker that our definition is terminating by using sized types. Consider the folowing function that increments every element in a list of naturals with its position:

```
incpos : List ℕ ⟶ List ℕ
incpos [] = []
incpos (x :: xs) = x :: incpos (map suc xs)
```

The recursive call to *incpos* gets flagged by the termination checker; we know that *map* does not alter the length of a list, but the termination checker cannot see this. For all it knows *map* equals *const* [1], which would make *incpos* non-terminating. The size-preserving property of *map* is not reflected in its type. To mitigate this issue, we can define an alternative version of the *List* datatype indexed with *Size*, which tracks the depth of a value in its type.

```
data SList (A : Set) : Size ⟶ Set where
  []    : ∀ {i} ⟶ SList A i
  _::_ : ∀ {i} ⟶ A ⟶ SList A i ⟶ SList A (↑ i)
```

Here $\uparrow\ i$ means that the depth of a value constructed using the :: constructor is one deeper than its recursive argument. Incidently, the recursive depth of a list is equal to its size (or length), but this is not necessarily the case. By indexing values of *List* with their size, we can define a version of *map* which reflects in its type that the size of the input argument is preserved:

```
map : ∀ {i} {A B : Set} ⟶ (A ⟶ B) ⟶ SList A i ⟶ SList B i
```

Using this definition of *map*, the definition of *incpos* is no longer rejected by the termination checker.

2.2.2 Universe Polymorphism

Contrary to Haskell, Agda does not have separate notions for *types*, *kinds* and *sorts*. Instead it provides an infinite hierarchy of type universes, where level is a member of the next, i.e. $Set\ n : Set\ (n+1)$. Agda uses this construction in favor of simply declaring $Set : Set$ to avoid the construction of contradictory statements through Russel's paradox.

This implies that every construction in Agda that ranges over some $Set\ n$ can only be used for values that are in $Set\ n$. It is not possible to define, for example, a *List* datatype that may contain both *values* and *types* for this reason.

We can work around this limitation by defining a *universe polymorphic* construction for lists:

```
data List {ℓ} (a : Set ℓ) : Set ℓ where
  [] : List a
  _::_ : a ⟶ List a ⟶ List a
```

Allowing us to capture lists of types (such as $\mathbb{N} :: Bool :: []$) and lists of values (such as $1 :: 2 :: []$) using a single datatype. Agda allows for the programmer to declare that $Set : Set$ using the {-# OPTIONS --type-in-type #-} pragma. Throughout the development accompanying this thesis, we will refrain from using this pragma wherever possible. The examples included in this thesis are often not universe-polymorphic, since the universe level variables required often pollute the code, and obfuscate the concept we are trying to convey.

2.3 Generic Programming and Type Universes

In *Datatype generic programming*, we define functionality not for individual types, but rather by induction on *structure* of types. This means that generic functions will not take values of a particular type as input, but a *code* that describes the structure of a type. Haskell's **deriving** mechanism is a prime example of this mechanism. Anytime we add **deriving** Eq to a datatype definition, GHC will, in the background, convert our datatype to a structural representation, and use a *generic equality* to create an instance of the Eq typeclass for our type.

2.3.1 Design Pattern

Datatype generic programming often follows a common design pattern that is independent of the structural representation of types involved. In general we follow the following steps:

1. First, we define a datatype $\mathcal{U}$ representing the structure of types, often called a *Universe*.
2. Next, we define a semantics $[\![\_]\!] : \mathcal{U} \to K$ that associates codes in $\mathcal{U}$ with an appropriate value of kind K. In practice this is often a functorial representation of kind $Set \to Set$.
3. Finally, we (often) define a fixed point combinator of type $(u : \mathcal{U}) \to Set$ that calculates the fixpoint of $[\![u]\!]$.

This imposes the implicit requirement that if we want to represent some type T with a code $u : \mathcal{U}$, the fixpoint of u should be isomorphic to T.

Given these ingredients we have everything we need at hand to write generic functions. Section 3 of Ulf Norell's *Dependently Typed Programming in Agda* [?] contains an in depth explanation of how this can be done in Agda. We will only give a rough sketch of the most common design pattern here. In general, a datatype generic function is supplied with a code $u : \mathcal{U}$, and returns a function whose type is dependent on the code it was supplied with.

Example Suppose we are defining a generic procedure for decidable equality. We might use the following type signature for such a procedure:

$$\_\overset{?}{=}\_ : \forall\, \{u : \mathcal{U}\} \longrightarrow (x : \mathbf{Fix}\ u) \longrightarrow (y : \mathbf{Fix}\ u) \longrightarrow x \equiv y \uplus \neg\, x \equiv y$$

If we now define $\overset{?}{=}$ by induction over u, we have a decision procedure for decidable equality that may act on values on any type, provided their structure can be described as a code in $\mathcal{U}$.

2.3.2 Example Universes

There exist many different type universes. We will give a short overview of the universes used in this thesis here; they will be explained in more detail later on when we define generic generators for them. The literature review in section 3.3 contains a brief discussion of type universes beyond those used we used for generic enumeration.

Regular Types Although the universe of regular types is arguably one of the simplest type universes, it can describe a wide variaty of recursive algebraic datatypes `[citation]`, roughly corresponding to the algebraic types in Haskell98. Examples of regular types are *natural numbers*, *lists* and *binary trees*. Regular types are insufficient once we want to have a generic representation of mutually recursive or indexed datatypes.

Indexed Containers The universe of *Indexed Containers* [?] provides a generic representation of large class indexed datatypes by induction on the index type. Datatypes we can describe using this universe include *Fin* (appendix A.2), *Vec* (appendix A.3) and closed lambda terms (appendix A.8).

Indexed Descriptions Using the universe of *Indexed Descriptions* [?] we can represent arbitrary indexed datatypes. This allows us to describe datatypes that are beyond what can be described using indexed containers, that is, datatypes with recursive subtrees that are interdependent or whose recursive subtrees have indices that cannot be uniquely determined from the index of a value.

3 Literature Review

In this section, we discuss some of the existing literature that is relevant in the domain of generating test data for property based testing. We take a look at some existing testing libraries, techniques for generation of constrained test data, and a few type universes beyond those we used that aim to describe (at least a subset of) indexed datatypes.

3.1 Property Based Testing

Property Based Testing aims to assert properties that universally hold for our programs by parameterizing tests over values and checking them against a collection of test values. Libraries for property based testing often include some kind of mechanism to automatically generate collections of test values. Existing tools take different approaches towards generation of test data: *QuickCheck* [?] randomly generates values within the test domain, while *SmallCheck* [?] and *LeanCheck* [?] exhaustively enumerate all values in the test domain up to a certain point.

3.1.1 Existing Libraries

There exist many libraries for property based testing. For brevity, we constrain ourselves here to those that are relevant in the domain of functional programming and/or haskell.

QuickCheck

Published in 2000 by Claessen & Hughes [?], QuickCheck implements property based testing for Haskell. As mentioned before, test values are generated by sampling randomly from the domain of test values. QuickCheck supplies the typeclass `Arbitrary`, whose instances are those types for which random values can be generated. A property of type $a \rightarrow Bool$ can be tested if a is an instance of `Arbitrary`. Instances for most common Haskell types are supplied by the library. If a property fails on a testcase, QuickCheck supplies a counterexample. Consider the following faulty definition of $reverse$:

$$\begin{aligned}
&reverse :: Eq\ a \Rightarrow [\,a\,] \rightarrow [\,a\,] \\
&reverse\ [\,] \qquad = [\,] \\
&reverse\ (x : xs) = nub\ ((reverse\ xs) +\!\!+ [\,x, x\,])
\end{aligned}$$

If we now test our function by calling $quickCheck\ reverse\_preserves\_length$, we get the following output:

```
Test.QuickCheck> quickCheck reverse_preserves_length
*** Failed! Falsifiable (after 8 tests and 2 shrinks):
[7,7]
```

We see that a counterexample was found after 8 tests *and 2 shrinks*. Due to the random nature of the tested values, the counterexamples that falsify a property are almost never minimal counterexamples. QuickCheck takes a counterexample and applies some function that produces a collection of values that are smaller than the original counterexample, and attempts to falsify the property using one of the smaller values. By repeatedly *Shrinking* a counterexample, QuickCheck is able to find much smaller counterexamples, which are in general of much more use to the programmer.

Perhaps somewhat surprising is that QuickCheck is also able randomly generate values for function types by modifying the seed of the random generator (which is used to generate the function's output) based on it's input.

(Lazy) SmallCheck

Contrary to QuickCheck, SmallCheck [?] takes an *enumerative* approach to the generation of test data. While the approach to formulation and testing of properties is largely similar to QuickCheck's, test values are not generated at random, but rather exhaustively enumerated up to a certain *depth*. Zero-arity constructors have depth `0`, while the depth of any positive arity constructor is one greather than the maximum depth of its arguments. The motivation for this is the *small scope hypothesis*: if a program is incorrect, it will almost allways fail on some small input [?].

In addition to SmallCheck, there is also *Lazy* SmallCheck. In many cases, the value of a property is determined only by part of the input. Additionally, Haskell's lazy semantics allow for functions to be defined on partial inputs. The prime example of this is a property `sorted :: Ord a => [a] -> Bool` that returns `false` when presented with `1:0:⊥`. It is not necessary to evaluate $\perp$ to determine that the input list is not ordered.

Partial values represent an entire class of values. That is, `1:0:⊥` can be viewed as a representation of the set of lists that have prefix `[1, 0]`. By checking properties on partial values, it is possible to falsify a property for an entire class of values in one go, in some cases greatly reducing the amount of testcases needed.

LeanCheck

Where SmallCheck uses a value's *depth* to bound the number of test values, LeanCheck uses a value's *size* [?], where size is defined as the number of construction applications of positive arity.

Both SmallCheck and LeanCheck contain functionality to enumerate functions similar to QuickCheck's `Coarbitrary`.

Hegdgehog

Hedgehog [?] is a framework similar to QuickCheck, that aims to be a more modern alternative. It includes support for monadic effects in generators and concurrent checking of properties.

Feat

A downside to both SmallCheck and LeanCheck is that they do not provide an efficient way to generate or sample large test values. QuickCheck has no problem with either, but QuickCheck generators are often more tedious to write compared to their SmallCheck counterpart. Feat [?] aims to fill this gap by providing a way to efficiently enumerate algebraic types, employing memoization techniques to efficiently find the n^{th} element of an enumeration.

QuickChick

QuickChick is a QuickCheck clone for the proof assistant Coq [?]. The fact that Coq is a proof assistant enables the user to reason about the testing framework itself [?]. This allows one, for example, to prove that generators adhere to some distribution.

3.1.2 Generating Constrained Test Data

Defining a suitable generation of test data for property based testing is notoriously difficult in many cases, independent of whether we choose to sample from or enumerate the space of test values. Writing generators for mutually recursive datatypes with a suitable distribution is especially challenging.

We run into prolems when we desire to generate test data for properties with a precondition. If a property's precondition is satisfied by few input values, it becomes unpractical to test such a property by simply generating random input data. Few testcases will be relevant (meaning they satisfy the precondition), and the testcases that do are often trivial cases. The usual solution to this problem is to define a custom test data generator that only produces data that satisfies the precondition. In some cases, such as the *insert_preserves_sorted* from section **??**, a suitable generator is not too hard to define:

```
gen_sorted :: Gen [Int]
gen_sorted = arbitrary >>= return ∘ diff
  where diff :: [Int] → [Int]
        diff []     = []
        diff (x:xs) = x : map (+x) (diff xs)
```

However, for more complex preconditions defining suitable generators is all but trivial.

3.1.3 Automatic Generation of Specifications

A surprising application of property based testing is the automatic generation of program specifications, proposed by Claessen et al. [?] with the tool *QuickSpec*. QuickSpec automatically generates a set of candidate formal specifications given a list of pure functions, specifically in the form of algebraic equations. Random property based testing is then used to falsify specifications. In the end, the user is presented with a set of equations for which no counterexample was found.

3.2 Techniques for Generating Test Data

This section discusses some existing work regarding the generation of test data satisfying invariants, such as well-formed λ-terms.

3.2.1 Lambda Terms

A problem often considered in literature is the generation of (well-typed) lambda terms [?, ?, ?]. Good generation of arbitrary program terms is especially interesting in the context of testing compiler infrastructure, and lambda terms provide a natural first step towards that goal.

Claessen and Duregaard [?] adapt the techniques described by Duregaard [?] to allow efficient generation of constrained data. They use a variation on rejection sampling, where the space of values is gradually refined by rejecting classes of values through partial evaluation (similar to SmallCheck [?]) until a value satisfying the imposed constrained is found.

An alternative approach centered around the semantics of the simply typed lambda calculus is described by Pałka et al. [?]. Contrary to the work done by Claessen and Duregaard [?], where typechecking is viewed as a black box, they utilize definition of the typing rules to devise an algorithm for generation of random lambda terms. The basic approach is to take some input type, and randomly select an inference rule from the set of rules that could have been applied to arrive at the goal type. Obviously, such a procedure does not guarantee termination, as repeated application of the function application rule will lead to an arbitrarily large goal type. As such, the algorithm requires a maximum search depth and backtracking in order to guarantee that a suitable term will eventually be generated, though it is not guaranteed that such a term exists if a bound on term size is enforced [?].

Wang [?] considers the problem of generating closed untyped lambda terms.

3.2.2 Inductive Relations in Coq

An approach to generation of constrained test data for Coq's QuickChick was proposed by Lampropoulos et al. [?] in their 2017 paper *Generating Good Generators for Inductive Relations*. They observe a common pattern where the required test data is of a simple type, but constrained by some precondition. The precondition is then given by some inductive dependent relation indexed by said simple type. The *Sorted* datatype shown in section **??** is a good example of this

They derive generators for such datatypes by abstracting over dependent inductive relations indexed by simple types. For every constructor, the resulting type uses a set of expressions as indices, that may depend on the constructor's arguments and universally quantified variables. These expressions induce a set of unification constraints that apply when using that particular constructor. These unification constraints are then used when constructing generators to ensure that only values for which the dependent inductive relation is inhabited are generated.

3.3 Generic Programming & Type Universes

Datatype generic programming concerns techniques that allow for the definition of functions by inducting on the *structure* of a datatype. Many approaches towards this goal have been developed, some more expressive than others. This section discusses a few of them.

3.3.1 SOP (Sum of Products)

On of the more simple representations is the so called *Sum of Products* view [?], where datatypes are respresented as a choice between an arbitrary amount of constructors, each of which can have any arity. This view corresponds to how datatypes are defined in Haskell. As we will see (for example in section **??**), other universes too employ sum and product combinators to describe the structure of datatypes, though they do not necessarily enforce the representation to be in disjunctive normal form.

Sum of Products, in its simplest form, cannot represent mutually recursive families of datatypes. An extension that allows this has been developed in [?].

4
A Combinator Library for Generators

4.1 The Type of Generators

We have not yet specified what it is exactly that we mean when we talk about *generators*. In the context of property based testing, it makes sense to think of generators as entities that produce values of a certain type; the machinery that is responsible for supplying suitable test values. As we saw in section **??**, this can mean different things depending on the library that you are using. *SmallCheck* and *LeanCheck* generators are functions that take a size parameter as input and produce an exhaustive list of all values that are smaller than the generator's input, while *QuickCheck* generators randomly sample values of the desired type. Though various libraries use different terminology to refer to the mechanisms used to produce test values, we will use *generator* as an umbrella term to refer to the test data producing parts of existing libraries.

4.1.1 Examples in Existing Libraries

When comparing generator definitions across libraries, we see that their definition is often more determined by the structure of the datatype they ought to produce values of than the type of the generator itself. Let us consider the Nat datatype (definition **??**). In QuickCheck, we could define a generator for the Nat datatype as follows:

```
genNat :: Gen Nat
genNat = oneof [pure Zero, Suc <$> genNat]
```

QuickCheck includes many combinators to finetune the distribution of values of the generated type, which are omitted in this case since they do not structurally alter the generator. Compare the above generator to its SmallCheck equivalent:

```
instance Serial m Nat where
  series = cons0 Zero \/ Cons1 Suc
```

Both generator definitions have a strikingly similar structure, marking a choice between the two available constructors ($Zero$ and Suc) and employing a appropriate combinators to produce values for said constructors. Despite this structural similarity, the underlying types of the respective generators are wildly different, with $genNat$ being an IO operation that samples random values and the $Serial$ instance being a function taking a depth and producing all values up to that depth.

4.1.2 Separating Structure and Interpretation

The previous example suggests that there is a case to be made for separating a generators structure from the format in which test values are presented. Additionally, by having a single datatype representing a generator's structure, we shift the burden of proving termination from a generator's definition to its interpretation, which in Agda is a considerable advantage. In practice this means that we define some datatype *Gen a* that marks the structure of a generator, and a function *interpret* : *Gen a* → *T a* that maps an input structure to some *T a*, where *T* which actually produces test values. In our case, we will almost exclusively consider an interpretation of generators to functions of type ℕ → *List a*, but we could have chosen *T* to by any other type of collection of values of type *a*. An implication of this separation is that, given suitable interpretation functions, a user only has to define a single generator in order to be able to employ different strategies for generating test values, potentially allowing for both random and enumerative testing to be combined into a single framework.

This approach means that generator combinators are not functions that operate on a a generator's result, such as merging two streams of values, but rather a constructor of some abstract generator type; *Gen* in our case. This datatype represents generators in a tree-like structure, not unlike the more familiar abstract syntax trees used to represent parsed programs.

4.1.3 The *Gen* Datatype

We define the datatype of generators, *Gen a t*, to be a family of types indexed by two types [1]. One signifying the type of values that are produced by the generator, and one specifying the type of values produced by recursive positions.

Listing 4.1: Definition of the *Gen* datatype

```
data Gen : (a : Set) → (t : Set) → Set where
  Or   : ∀ {a t   : Set} → Gen a t → Gen a t → Gen a t
  Ap   : ∀ {a t b : Set} → Gen (b → a) t → Gen b t → Gen a t
  Pure : ∀ {a t   : Set} → a → Gen a t
  None : ∀ {a t   : Set} → Gen a t
  μ    : ∀ {a     : Set} → Gen a a
```

Closed generators are then generators produce that produce the values of the same type as their recursive positions:

```
𝔾 : Set → Set
𝔾 a = Gen a a
```

The *Pure* and *Ap* constructors make *Gen* an instance of *Applicative*, meaning that we can (given a fancy operator for denoting choice) denote generators in way that is very similar to their definition:

```
nat : 𝔾 ℕ
nat = ⦇ zero  ⦈
    ∥ ⦇ suc μ ⦈
```

[1] The listed definition will not be accepted by Agda due to inconsistencies in the universe levels. This is also the case for many code examples to come. To keep things readable, we will not concern ourselves with universe levels throughout this thesis.

This serves to emphasize that the structure of generators can, in the case of simpler datatypes, be mechanically derived from the structure of a datatype. We will see how this can be done in chapter ??.

The question remains how to deal with constructors that refer to *other* types. For example, consider the type of lists (definition ??). We can define an appropriate generator following the structure of the datatype definition:

```
list : ∀ {a : Set} → 𝔾 a → 𝔾 (List a)
list a = ⦇ []         ⦈
       ∥ ⦇ { }? :: μ ⦈
```

It is however not immediately clear what value to supply to the remaining interaction point. If we inspect its goal type we see that we should supply a value of type *Gen a (List a)*: a generator producing values of type *a*, with recursive positions producing values of type *List a*. This makes little sense, as we would rather be able to invoke other *closed generators* from within a generator. To do so, we add another constructor to the *Gen* datatype, that signifies the invokation of a closed generator for another datatype:

```
Call : ∀ {a t : Set} → Gen a a → Gen a t
```

Using this definition of *Call*, we can complete the previous definition for *list*:

```
list : ∀ {a : Set} → 𝔾 a → 𝔾 (List a)
list a = ⦇ []              ⦈
       ∥ ⦇ (Call a) :: μ ⦈
```

4.1.4 Generator Interpretations

We can view a generator's interpretation as any function mapping generators to some type, where the output type is parameterized by the type of values produced by a generator:

```
Interpretation : (Set → Set) → Set
Interpretation T = ∀ {a t : Set} → 𝔾 t → Gen a t → T a
```

From this definition of *Interpretation*, we can define concrete interpretations. For example, if we want to behave our generators similar to SmallCheck's *Series*, we might define the following concrete instantiation of the *Interpretation* type:

```
GenAsList : Set
GenAsList = Interpretation λ a → ℕ → List a
```

We can then define a generator's behiour by supplying a definition that inhabits the *GenAsList* type:

```
asList : GenAsList
asList gen = { }?
```

The goal type of the open interaction point is then ℕ→ *List a*. We will see in section 4.3 how we can flesh out this particular interpretation. We could however have chosen any other result type, depending on what suits our particular needs. An alternative would be to interpret generators as a *Colist*, omitting the depth bound altogether:

```
GenAsColist : Set
GenAsColist = ∀ {i : Size} → Interpretation λ a → Colist a i
```

4.2 Generalization to Indexed Datatypes

A first approximation towards a generalization of the *Gen* type to indexed types might be to simply lift the existing definition from *Set* to $I \to Set$.

```
𝔾ᵢ : ∀ {I : Set} → (I → Set) → Set
𝔾ᵢ {I} P = (i : I) → 𝔾 (P i)
```

However, by doing so we implicitly impose the constraint that the recursive positions of a value have the same index as the recursive positions within it. Consider, for example, the *Fin* type (definition ??). If we attempt to define a generator using the lifted type, we run into a problem.

```
fin : 𝔾ᵢ Fin
fin zero    = None
fin (suc n) = ⦇ zero       ⦈
            ∥ ⦇ suc { }?  ⦈
```

Any attempt to fill the open interaction point with the constructor fails, as it expects a value of *Gen (Fin n) (Fin suc n)*, but requires both its type parameters to be equal. We can circumvent this issue by using direct recursion.

```
fin : 𝔾ᵢ Fin
fin zero    = None
fin (suc n) = ⦇ zero ⦈
            ∥ ⦇ suc (Call (fin n)) ⦈
```

It is however clear that this approach becomes a problem once we attempt to define generators for datatypes with recursive positions which have indices that are not structurally smaller than the index they target. To overcome these limitations we resolve to a separate deep embedding of generators for indexed types.

And consequently the type of closed indexed generators.

```
𝔾ᵢ : ∀ {I : Set} → (I → Set) → Set
𝔾ᵢ {I} P = (i : I) → Genᵢ (P i) P i
```

Notice how the Ap_i constructor allows for its second argument to have a different index. The reason for this becomes clear when we

With the same combinators as used for the *Gen* type, we can now define a generator for the *Fin* type.

```
fin : 𝔾ᵢ Fin
fin zero    = empty
fin (suc n) = ⦇ zero       ⦈
            ∥ ⦇ suc (μᵢ n) ⦈
```

Now defining generators for datatypes with recursive positions whose indices are not structurally smaller than the index of the datatype itself can be done without complaints from the termination checker, such as well-scoped λ-terms (definition ??).

```
term : 𝔾ᵢ WS
term n = ⦇ var (Callᵢ {i = n} fin n) ⦈
       ∥ ⦇ abs (μᵢ (suc n))          ⦈
       ∥ ⦇ app (μᵢ n) (μᵢ n)          ⦈
```

It is important to note that it is not possible to call indexed generators from simple generators and vice versa with this setup. We can allow this by either parameterizing the *Call* and *iCall* constructors with the datatype they refer to, or by adding extra constructors to the *Gen* and Gen_i datatypes, making them mutually recursive.

Listing 4.2: Definitiong of the Gen_i datatype

```
data Genᵢ {I : Set} : Set → (I → Set) → I → Set where
  Pureᵢ  : ∀ {a : Set} {t : I → Set} {i : I} → a → Genᵢ a t i

  Apᵢ    : ∀ {a b : Set} {t : I → Set} {x : I} {y : I}
           → Genᵢ (b → a) t x → Genᵢ b t y → Genᵢ a t x

  Orᵢ    : ∀ {a : Set} {t : I → Set} {i : I}
           → Genᵢ a t i → Genᵢ a t i → Genᵢ a t i

  μᵢ     : ∀ {a : I → Set} (i : I) → Genᵢ (a i) a i

  Noneᵢ : ∀ {a : Set} {t : I → Set} {i : I} → Genᵢ a t i

  Callᵢ  : ∀ {t : I → Set} {i : I} {J : Set} {s : J → Set}
           → ((j : J) → Genᵢ (s j) s j) → (j : J) → Genᵢ (s j) t i
```

4.3 Interpreting Generators as Enumerations

We will now consider an example interpretation of generators where we map values of the *Gen* or Gen_i datatypes to functions of type ℕ→ *List a*. The constructors of both datatypes mimic the combinators used Haskell's *Applicative* and *Alternative* typeclasses, so we can use the *List* instances of these typeclasses for guidance when defining an enumerative interpretation.

Listing 4.3: Interpretation of the *Gen* datatype as an enumeration

```
toList : Interpretation λ a → ℕ → List a
toList _ _           zero    = []
toList g (Or g1 g2)  (suc n) = merge (toList g g1 (suc n)) (toList g g2 (suc n))
toList g (Ap g1 g2)  (suc n) =
  concatMap (λ f → map f (toList g g2 (suc n))) (toList g g1 (suc n))
toList _ (Pure x)    (suc n) = x :: []
toList _ None        (suc n) = []
toList g μ           (suc n) = toList g g n
toList _ (Call g)    (suc n) = toList g g (suc n)
```

Similarly, we can define such an interpretation for the Gen_i datatype similar to listing 4.3 with the only difference being the appropriate indices getting passed to recursive calls. Notice how our generator's behaviour - most notably the intended semantics of the input depth bound - is entirely encoded within the definition of the interpretation. In this case by decrementing n anytime a recursive position is encountered.

4.4 Properties for Enumerations

4.5 Generating Function Types

4.6 Monadic Generators

There are some cases in which the applicative combinators are not expressive enough to capture the desired generator. For example, if we were to define a construction for generation of Σ types, we encounter some problems.

```
gen-Σ : ∀ {I : Set} {P : I → Set} → 𝔾 I → ((i : I) → 𝔾 (P i)) → 𝔾 (Σ[ i ∈ I ] P i)
gen-Σ gi gp = ⦇ (λ x y → x , { }? ) (Call gi) (Call (gp { }? )) ⦈
```

We can extend the *Gen* datatype with a *Bind* operation that mimics the monadic bind operator (≫=) to allow for such dependencies to exist between generated values.

```
gen-Σ : ∀ {I : Set} {P : I → Set} → 𝔾 I → ((i : I) → 𝔾 (P i)) → 𝔾 (Σ[ i ∈ I ] P i)
gen-Σ gi gp = (Call gi) »= λ i → (Call (gp i)) »= λ p → Pure (i , p)
```

5
Generic Generators for Regular types

A large class of recursive algebraic data types can be described with the universe of *regular types*. In this section we lay out this universe, together with its semantics, and describe how we may define functions over regular types by induction over their codes. We will then show how this allows us to derive from a code a generic generator that produces all values of a regular type. We sketch how we can prove that these generators are indeed complete.

5.1 The universe of regular types

Though the exact definition may vary across sources, the universe of regular types is generally regarded to consist of the *empty type* (or $\mathbb{0}$), the unit type (or $\mathbb{1}$) and constants types. It is closed under both products and coproducts [1]. We can define a datatype for this universe in Agda as shown in lising 5.1

Listing 5.1: The universe of regular types

```
data Reg : Set where
  Z   : Reg
  U   : Reg
  _⊕_ : Reg → Reg → Reg
  _⊗_ : Reg → Reg → Reg
  I   : Reg
```

The semantics associated with the *Reg* datatype, as shown in listing 5.1, map a code to a functorial representation of a datatype, commonly known as its *pattern functor*. The datatype that is represented by a code is isomorphic to the least fixpoint of its pattern functor. We fix pattern functors using the following fixpoint combinator:

```
data Fix (c : Reg) : Set where
  In : ⟦ c ⟧ (Fix c) → Fix c
```

Example The type of natural numbers (see listing A.1) exposes two constructors: the nullary constructor *zero*, and the unary constructor *suc* that takes one recursive argument. We may thus view this type as a coproduct (i.e. choice) of either a *unit type* or a *recursive subtree*:

[1]This roughly corresponds to datatypes in Haskell 98

Listing 5.2: Semantics of the universe of regular types

```
⟦_⟧ : Reg ⟶ Set ⟶ Set
⟦ Z       ⟧ r = ⊥
⟦ U       ⟧ r = ⊤
⟦ c₁ ⊕ c₂ ⟧ r = ⟦ c₁ ⟧ r ⊎ ⟦ c₂ ⟧ r
⟦ c₁ ⊗ c₂ ⟧ r = ⟦ c₁ ⟧ r × ⟦ c₂ ⟧ r
⟦ I       ⟧ r = r
```

```
ℕ' : Set
ℕ' = Fix (U ⊕ I)
```

We convince ourselves that ℕ′ is indeed equivalent to ℕ by defining conversion functions, and showing their composition is extensionally equal to the identity function, shown in listing 5.1.

Listing 5.3: Isomorphism between ℕ and ℕ′

```
fromℕ : ℕ ⟶ ℕ'
fromℕ zero    = In (inj₁ tt)
fromℕ (suc n) = In (inj₂ (fromℕ n))

toℕ : ℕ' ⟶ ℕ
toℕ (In (inj₁ tt)) = zero
toℕ (In (inj₂ y))  = suc (toℕ y)

ℕ-iso₁ : ∀ {n} ⟶ toℕ (fromℕ n) ≡ n
ℕ-iso₁ {zero}  = refl
ℕ-iso₁ {suc n} = cong suc ℕ-iso₁

ℕ-iso₂ : ∀ {n} ⟶ fromℕ (toℕ n) ≡ n
ℕ-iso₂ {In (inj₁ tt)} = refl
ℕ-iso₂ {In (inj₂ y)}  = cong (In ∘ inj₂) ℕ-iso₂

ℕ≃ℕ' : ℕ ≃ ℕ'
ℕ≃ℕ' = record { from = fromℕ ; to = toℕ ; iso₁ = ℕ-iso₁ ; iso₂ = ℕ-iso₂ }
```

We may then say that a type is regular if we can provide a proof that it is isomorphic to the fixpoint of some c of type Reg. We use a record to capture this notion, consisting of a code and an value that witnesses the isomorphism.

```
record Regular (a : Set) : Set where
  field
    W : Σ[ c ∈ Reg ] (a ≃ Fix c)
```

By instantiating *Regular* for a type, we may use any generic functionality that is defined over regular types.

5.1.1 Non-regular data types

Although there are many algebraic datatypes that can be described in the universe of regular types, some cannot. Perhaps the most obvious limitation the is lack of ability to caputure data families indexed with values. The regular univeres imposes the implicit restriction that a datatype is uniform in the sens that all recursive subtrees are of the same type. Indexed families, however, allow for recursive subtrees to have a structure that is different from the structure of the datatype they are a part of.

Furethermore, any family of mutually recursive datatypes cannot be described as a regular type; again, this is a result of the restriction that recursive positions allways refer to a datatype with the same structure.

5.2 Generic Generators for regular types

We can derive generators for all regular types by induction over their associated codes. Furthermore, we will show in section ?? that, once interpreted as enumerators, these generators are complete; i.e. any value will eventually show up in the enumerator, provided we supply a sufficiently large size parameter.

5.2.1 Defining functions over codes

If we apply the approach described in section 2.3.1 without care, we run into problems. Simply put, we cannot work with values of type $Fix\ c$, since this implicitly imposes the restriction that any I in c refers to $Fix\ c$. However, as we descent into recursive calls, the code we are working with changes, and with it the type associated with recursive positions. For example: the I in $(U \oplus I)$ refers to values of type $Fix\ (U \oplus I)$, not $Fix\ I$. We need to make a distinction between the code we are currently working on, and the code that recursive positions refer to. For this reason, we cannot define the generic generator, $deriveGen$, with the following type signature:

```
deriveGen : (c : Reg) → Gen (Fix c) (Fix c)
```

If we observe that $[\![c]\!](Fix\ c) \simeq Fix\ c$, we may alter the type signature of $deriveGen$ slightly, such that it takes two input codes instead of one

```
deriveGen : (c c' : Reg) → Gen (⟦ c ⟧ (Fix c')) (⟦ c' ⟧ (Fix c'))
```

This allows us to induct over the first input code, while still being able to have recursive positions reference the correct *top-level code*. Notice that the first and second type parameter of Gen are different. This is intensional, as we would otherwise not be able to use the μ constructor to mark recursive positions.

5.2.2 Composing generic generators

Now that we have the correct type for $deriveGen$ in place, we can start defining it. Starting with the cases for Z and U:

```
deriveGen Z c' = empty
deriveGen U c' = pure tt
```

Both cases are trivial. In case of the Z combinator, we yield a generator that produces no elements. As for the U combinator, $[\![U]\!](Fix\ \ c')$ equals $\top$, so we need to return a generator that produces all inhabitants of $\top$. This is simply done by lifting the single value tt into the generator type.

In case of the I combinator, we cannot simply use the μ constructor right away. In this context, μ has the type $Gen\ ([\![c']\!](Fix\ \ c'))\ ([\![c']\!](Fix\ \ c'))$. However, since $[\![I]\!](Fix\ \ c)$ equals $Fix\ \ c$, the types do not lign up. We need to map the In constructor over μ to fix this:

```
deriveGen I c' = ⦇ In μ ⦈
```

Moving on to products and coproducts: with the correct type for $deriveGen$ in place, we can define their generators quite easily by recursing on the left and right subcodes, and combining their results using the appropriate generator combinators:

```
deriveGen (cₗ ⊕ cᵣ) c' = ⦇ inj₁ (deriveGen cₗ c') ⦈ ∥ ⦇ inj₂ (deriveGen cᵣ c') ⦈
deriveGen (cₗ ⊗ cᵣ) c' = ⦇ deriveGen cₗ c' , deriveGen cᵣ c' ⦈
```

Although defining $deriveGen$ constitutes most of the work, we are not quite there yet. Since the the $Regular$ record expects an isomorphism with $Fix\ \ c$, we still need to wrap the resulting generator in the In constructor:

```
genericGen : (c : Reg) → Gen (Fix c) (Fix c)
genericGen c = ⦇ In (Call (deriveGen c c)) ⦈
```

The elements produced by $genericGen$ can now readily be transformed into the required datatype through an appropriate isomorphism.

Example We derive a generator for natural numbers by invoking $genericGen$ on the appropriate code $U \oplus I$, and applying the isomorphism defined in listing ?? to its results:

```
genℕ : Gen ℕ ℕ
genℕ = ⦇ (_≃_.to ℕ≃ℕ') (Call (genericGen (U ⊕ I))) ⦈
```

In general, we can derive a generator for any type A, as long as there is an instance argument of the type $Regular\ \ A$ in scope:

```
isoGen : ∀ {A} → ⦃ p : Regular A ⦄ → Gen A A
isoGen ⦃ record { W = c , iso } ⦄ = ⦇ (_≃_.to iso) (Call (genericGen c)) ⦈
```

5.3 Constant Types

In some cases, we describe datatypes as a compositions of other datatypes. An example of this would be lists of numbers, $List\ \mathbb{N}$. Our current universe definition is not expressive enough to do this.

Example Given the code representing natural numbers ($U \oplus I$) and lists ($U \oplus (C \otimes I)$, where C is a code representing the type of elements in the list), we might be tempted to try and replace C with the code for natural numbers in the code for lists:

```
listℕ : Set
listℕ = Fix (U ⊕ ((U ⊗ I) ⊗ I))
```

This code does not describe lists of natural numbers. The problem here is that the two recursive positions refer to the *same* code, which is incorrect. We need the first I to refer to the code of natural numbers, and the second I to refer to the entire code.

5.3.1 Definition and Semantics

In order to be able to refer to other recursive datatypes, the universe of regular types often includes a constructor marking *constant types*:

$$\mathrm{K} : \mathrm{Set} \longrightarrow \mathbf{Reg}$$

The K constructor takes one parameter of type Set, marking the type it references. The semantics of K is simply the type it carries:

$$[\![\ \mathrm{K}\ s\ ]\!]\ r = s$$

Example Given the addition of K, we can now define a code that represents lists of natural numbers:

$$\begin{aligned}&\mathbf{list}\mathbb{N} : \mathrm{Set}\\&\mathbf{list}\mathbb{N} = \mathbf{Fix}\ (\mathrm{U} \oplus (\mathrm{K}\ (\mathbf{Fix}\ (\mathrm{U} \oplus \mathrm{I})) \otimes \mathrm{I}))\end{aligned}$$

With the property that $list\mathbb{N} \simeq List\ \mathbb{N}$.

5.3.2 Generic Generators for Constant Typse

When attempting to define $deriveGen$ on $K\ \ s$, we run into a problem. We need to return a generator that produces values of type s, but we have no information about s whatsoever, apart from knowing that it lies in Set. This is a problem, since we cannot derive generators for arbitrary values in Set. This leaves us with two options: either we restrict the types that K may carry to those types for which we can generically derive a generator, or we require the programmer to supply a generator for every constant type in a code. We choose the latter, since it has the advantage that we can generate a larger set of types.

We have the programmer supply the necessary generators by defining a *metadata* structure, indexed by a code, that carries additional information for every K constructor used. We then parameterize $deriveGen$ with a metadata structure, indexed by the code we are inducting over. The definition of the metadata structure is shown in listing 5.3.2.

Listing 5.4: Metadata structure carrying additional information for constant types

```
data KInfo (P : Set → Set) : Reg → Set where
  Z~    : KInfo P Z
  U~    : KInfo P U
  _⊕~_ : ∀ {cₗ cᵣ} → KInfo P cₗ → KInfo P cᵣ → KInfo P (cₗ ⊕ cᵣ)
  _⊗~_ : ∀ {cₗ cᵣ} → KInfo P cₗ → KInfo P cᵣ → KInfo P (cₗ ⊗ cᵣ)
  I~    : KInfo P I
  K~    : ∀ {S} → P S → KInfo P (K S)
```

We then adapt the type of $deriveGen$ to accept a parameter containing the required metadata structure:

```
deriveGen : (c c' : Reg) → KInfo (λ S → Gen S S) c → Gen (⟦ c ⟧ (Fix c')) (⟦ c' ⟧ (Fix c'))
```

We then define *deriveGen* as follows for constant types. All cases for existing constructors remain the same.

```
deriveGen (K x) c' (K~ g) = Call g
```

5.4 Complete Enumerators For Regular Types

By applying the *toList* interpretation shown in listing 4.3 to our generic generator for regular types we obtain a complete enumeration for regular types. Obviously, this relies on the programmer to supply complete generators for all constant types referred to by a code.

We formulate the desired completeness property as follows: *for every code c and value x it holds that there is an n such that x occurs at depth n in the enumeration derived from c.* In Agda, this amounts to proving the following statement:

```
genericGen-Complete : ∀ {c x} → ∃[ n ] (x ∈ toList (genericGen c) (genericGen c) n)
```

Just as was the case with deriving generators for codes, we need to take into the account the difference between the code we are currently working with, and the top level code. To this end, we alter the previous statement slightly.

```
deriveGen-Complete : ∀ {c c' x} → ∃[ n ] (x ∈ toList (deriveGen c c') (deriveGen c' c') n)
```

If we invoke this lemma with two equal codes, we may leverage the fact that *In* is bijective to obtain a proof that *genericGen* is complete too. The key observation here is that mapping a bijective function over a complete generator results in another complete generator.

The completeness proof roughly follows the following steps:

- First, we prove completeness for individual generator combinators
- Next, we assemble a suitable metadata structure to carry the required proofs for constant types in the code.
- Finally, we assemble the individual components into a proof of the statement above.

5.4.1 Combinator Correctness

We start our proof by asserting that the used combinators are indeed complete. That is, we show for every constructor of *Reg* that the generator we return in *deriveGen* produces all elements of the interpretation of that constructor. In the case of *Z* and *U*, this is easy.

```
deriveGen-Complete {Z} {c'} {()}
deriveGen-Complete {U} {c'} {tt} = 1 , here
```

The semantics of *Z* is the empty type, so any generator producing values of type $\perp$ is trivially complete. Similarly, in the case of *U* we simply need to show that interpreting *pure tt* returns a list containing *tt*.

Things become a bit more interesting once we move to products and coproducts. In the case of coproducts, we know the following equality to hold, by definition of both *toList* and *deriveGen*:

```
toList (deriveGen (cₗ ⊕ cᵣ) c') (deriveGen c' c') n
  ≡ merge (toList ⦇ inj₁ (deriveGen cₗ c') ⦈ (deriveGen c' c') n)
          (toList ⦇ inj₂ (deriveGen cᵣ c') ⦈ (deriveGen c' c') n)
```

Basically, this equality unfolds the *toList* function one step. Notice how the generators on the left hand side of the equation are *almost* the same as the recursive calls we make. This means that we can prove completeness for coproducts by proving the following lemmas, where we obtain the required completeness proofs by recursing on the left and right subcodes of the coproduct.

```
merge-complete-left  : ∀ {A} {xsₗ xsᵣ : List A} {x : A} → x ∈ xsₗ → x ∈ merge xsₗ xsᵣ
merge-complete-right : ∀ {A} {xsₗ xsᵣ : List A} {x : A} → x ∈ xsᵣ → x ∈ merge xsₗ xsᵣ
```

Similarly, by unfolding the toList function one step in the case of products, we get the following equality:

```
toList (deriveGen (cₗ ⊗ cᵣ) c') (deriveGen c' c') n
  ≡ ⦇ (toList (deriveGen cₗ c') (deriveGen c' c') n)
    , (toList (deriveGen cᵣ c') (deriveGen c' c') n) ⦈
```

We can prove the right hand side of this equality by proving the following lemma about the applicative instance of lists:

```
×-complete : ∀ {A B} {x : A} {y : B} {xs ys} → x ∈ xs → y ∈ ys → (x , y) ∈ ⦇ xs , ys ⦈
```

Again, the preconditions of this lemma can be obtained by recursing on the left and right subcodes of the product.

5.4.2 Completeness for Constant Types

Since our completeness proof relies on completeness of the generators for constant types, we need the programmer to supply a proof that the supplied generators are indeed complete. To this end, we add a metadata parameter to the type of *deriveGen-complete*, with the following type:

```
ProofMD : Reg → Set
ProofMD c = KInfo (λ S → Σ[ g ∈ Gen S S ] (∀ {x} → ∃[ n ] (x ∈ toList g g n))) c
```

In order to be able to use the completeness proof from the metadata structure in the *K* branch of *deriveGen-Complete*, we need to be able to express the relationship between the metadata structure used in the proof, and the metadata structure used by *deriveGen*. To do this, we need a way to transform the type of information that is carried by a value of type *KInfo*:

```
KInfo-map : ∀ {c P Q} → (∀ {s} → P s → Q s) → KInfo P c → KInfo Q c
KInfo-map f (K~ x) = K~ (f x)
```

Given the definition of *KInfo-map*, we can take the first projection of the metadata input to *deriveGen-Complete*, and use the resulting structure as input to *deriveGen*:

```
ProofMD : Reg → Set
ProofMD c = KInfo (λ S → Σ[ g ∈ Gen S S ] (∀ {x} → ∃[ n ] (x ∈ toList g g n))) c
```

This amounts to the following final type for *deriveGen-Complete*, where ◀m = *KInfo-map* $proj_1$ m:

```
deriveGen-Complete : (c c' : Reg) → (i : ProofMD c) → (i' : ProofMD c')
  → ∀ {x} → ∃[ n ] (x ∈ toList (C.deriveGen c c' (◀ i)) (C.deriveGen c' c' (◀ i')) n)
```

Now, with this explicit relation between the completeness proofs and the generators given to *deriveGen*, we can simply retrun the proof contained in the metadata of the *K* branch.

5.4.3 Generator Monotonicity

The lemma $\times$-*complete* is not enough to prove completeness in the case of products. We make two recursive calls, that both return a dependent pair with a depth value, and a proof that a value occurs in the enumeration at that depth. However, we need to return just such a dependent pair stating that a pair of both values does occur in the enumeration at a certain depth. The question is what depth to use. The logical choice would be to take the maximum of both dephts. This comes with the problem that we can only combine completeness proofs when they have the same depth value.

For this reason, we need a way to transform a proof that some value x occurs in the enumeration at depth n into a proof that x occurs in the enumeration at depth m, given that $n \leq m$. In other words, the set of values that occurs in an enumeration monotoneously increases with the enumeration depth. To finish our completeness proof, this means that we require a proof of the following lemma:

```
n ≤ m ⟶ x ∈ toList (C.deriveGen c c’ (◀ i)) (C.deriveGen c’ c’ (◀ i’)) n
      ⟶ x ∈ toList (C.deriveGen c c’ (◀ i)) (C.deriveGen c’ c’ (◀ i’)) m
```

We can complete a proof of this lemma by using the same approach as for the completeness proof.

5.4.4 Final Proof Sketch

By bringing all these elements together, we can prove that *deriveGen* is complete for any code c, given that the programmer is able to provide a suitable metadatastructure. We can transform this proof into a proof that *isoGen* returns a complete generator by observing that any isomorphism $A \simeq B$ establishes a bijection between the types A and B. Hence, if we apply such an isomorphism to the elements produced by a generator, completeness is preserved.

We have the required isomorphism readily at our disposal in *isoGen*, since it is contained in the instance argument *Regular a*. This allows us to have *isoGen* return a completeness proof for the generator it derives:

```
isoGen : ∀ {A} ⟶ ⦃ p : Regular A ⦄ ⟶ Σ[ g ∈ Gen A A ] ∀ {x} ⟶ ∃[ n ] (x ∈ toList g g n)
```

6 Deriving Generators for Indexed Containers

7 Deriving Generators for Indexed Descriptions

7.1 Universe Description

We utilize the generic description for indexed datatypes proposed by Dagand [?] in his PhD thesis.

7.1.1 Definition

Indexed descriptions are not much unlike the codes used to describe regular types (that is, the *Reg* datatype), with the differences being:

1. A type parameter $I : Set$, describing the type of indices.
2. A generalized coproduct, `σ, that denotes choice between n constructors, in favor of the $\oplus$ combinator.
3. Recursive positions storing the index of recursive values
4. Addition of a combinator to encode Σ types which is a generalization of the K combinator.

This amounts to the definition of indexed descriptions described in listing 7.1.1.

Listing 7.1: The Universe of indexed descriptions

```
data IDesc (I : Set) : Set where
  ‘var : (i : I) → IDesc I
  ‘1   : IDesc I
  _‘×_ : (A B : IDesc I) → IDesc I
  ‘σ   : (n : ℕ) → (T : Sl n → IDesc I) → IDesc I
  ‘Σ   : (S : Set) → (T : S → IDesc I) → IDesc I
```

The *Sl* datatype is used to select the right branch from the generic coproduct, and is isomorphic to the *Fin* datatype.

```
data Sl : ℕ → Set where
  · : ∀ {n} → Sl (suc n)
  ▻_ : ∀ {n} → Sl n → Sl (suc n)
```

7.1.2 Examples

7.2 Generic Generators for Indexed Descriptions

8
Program Term Generation

9
Implementation in Haskell

10
Conclusion & Further Work

A
Datatype Definitions

A.1 Natural numbers

Listing A.1: Definition of natural numbers in Haskell and Agda

```
data Nat = Zero
         | Suc N
```

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

A.2 Finite Sets

Listing A.2: Definition of finite sets in Agda

```
data Fin : ℕ → Set where
  zero : ∀ {n : ℕ} → Fin (suc n)
  suc  : ∀ {n : ℕ} → Fin n → Fin (suc n)
```

A.3 Vectors

Listing A.3: Definition of vectors (size-indexed listst) in Agda

```
data Vec (a : Set) : ℕ ⟶ Set where
  []    : Vec a zero
  _::_ : ∀ {n : ℕ} ⟶ a ⟶ Vec a n ⟶ Vec a (suc n)
```

A.4 Simple Types

Listing A.4: Definition of simple types in Haskell and Agda

```
data Type = T
          | Type : − >: Type
```

```
data Ty : Set where
  ‘τ      : Ty
  _‘⟶_ : Ty ⟶ Ty ⟶ Ty
```

A.5 Contexts

Listing A.5: Definition of contexts in Haskell and Agda

```
data Ctx = Empty
         | Cons Id Type Ctx
```

```
data Ctx : Set where
  ∅ : Ctx
  _,_:_ : Ctx ⟶ Id ⟶ Ty ⟶ Ctx
```

A.6 Raw λ-Terms

Listing A.6: Definition of raw λ-terms in Haskell and Agda

```
data RT = Var Id
        | Abs Id RT
        | App RT RT
```

```
data RT : Set where
  $_     : Id → RT
  Λ_⇒_ : Id → RT → RT
  _⊠_    : RT → RT → RT
```

A.7 Lists

Listing A.7: Definition lists and Agda

```
data List (a : Set) : Set where
  []   : List a
  _::_ : a → List a → List a
```

A.8 Well-scoped λ-terms

Listing A.8: Definition well-scoped λ-terms in Agda

```
data WS : ℕ → Set where
  var : ∀ {n : ℕ} → Fin n → WS n
  abs : ∀ {n : ℕ} → WS (suc n) → WS n
  app : ∀ {n : ℕ} → WS n → WS n → WS n
```

Code listings

Listing of tables