Utrecht University

Faculty of Science

Dept. of Information and Computing Sciences

Generating Constrained Test Data using Datatype Generic Programming

Author

C.R. van der Rest

Supervised by:

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

A thesis submitted for the degree of ”Master of Science”

July 12, 2019

Contents

Declaration

I am very grateful to my advisors Wouter Swierstra and Manuel Chakravarty, without whom this work would not have been what it is today. Their encouragement, guidance and constructive criticism has been invaluable to me, and I am glad to have had the opportunity to conduct my Master's thesis under their supervision. Furthermore, I am thankful to the members of IOHK's Plutus team for finding time in their schedule to discuss the project with me, and for the financial support provided by IOHK.

I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where stated otherwise by reference or acknowledgment, the work presented is entirely my own.

Abstract

The generation of suitable test data is an essential part of *property based testing*. Obtaining test data is simple enough when there are no additional constraints, however things become more complicated once we require data with a richer structure, for example well-formed programs when testing a compiler. We observe that we can often describe constrained data as an *indexed family*. By generating values of an indexed family that describes a set of constrained test data, we simultaneously obtain a way to generate the constrained data itself. To achieve this goal, we consider three increasingly expressive type universes: *regular types*, *indexed containers* and *indexed descriptions*. We show how generators can be derived from codes in these universes, and for *regular types* and *indexed descriptions* we show that these derived generators are *complete*. We implement the generic generator for indexed descriptions in Haskell, and use this implementation to generate constrained test data, such as well-typed lambda terms.

1

Introduction

A proven method for assuring ourselves that the programs we write behave according to our intentions is by simply running them, and observing their behavior. This approach, where we do not try to reason about a program by looking at its code, but rather treat it as a black box which we expect to have a certain behavior is commonly known as *property based testing*. Of course, this requires us to come up with suitable *input values* to run our program on. This thesis concerns itself with the question of *how* we can find these input values, specifically if our input data is subject to *constraints*. For example, if we want to test that the code generation stage of a compiler produces correct results, we need well-formed programs to run it on. We approach this problem by developing a general framework with which we can both describe and generate many kinds of constrained data, using indexed families as a uniform representation of these constraints.

The topic of this thesis was originally proposed by Wouter Swierstra, who suggested this work as a means to work towards term generation for *Plutus Core* [21], which is the compilation target of the smart contract language used by the *Cardano* [3] blockchain.

1.1 Problem statement

At first glance, defining properties that capture the desired behavior of a program may seem like the challenging aspect of property based testing. While properly defining a program's behavior in terms of a few universal properties is certainly hard, so is generating suitable test data. To illustrate this, we consider a simple example in the context of QuickCheck [10]. Suppose we are writing an implementation of *insertion sort*, and we want to make sure that we can insert an element into a sorted list while preserving its sortedness. We write the following predicate that checks whether a list of integers is sorted:

```
sorted :: [Int] → Bool
sorted []         = True
sorted [x]        = True
sorted (x:y:xs) = x ≤ y ∧ sorted (y:xs)
```

With this predicate at hand, we set out to define a property that we believe should hold for the *insert* function.

```
prop :: Int → [Int] → Property
prop x xs = sorted xs ==> sorted (insert x xs)
```

However, once we try to test this property by simply calling *quickCheck prop* we are presented with the following output:

```
> quickCheck prop
*** Gave up! Passed only 70 tests; 1000 discarded tests.
```

QuickCheck was not able to find enough suitable test cases! To obtain test values, QuickCheck uses its *Arbitrary* typeclass, meaning in this case that *xs* will just be a list of random integers. This has two important implications:

1. A random list of integers has a *very small* probability of being sorted. Hence, the antecedent of the implication will return *True* in a very small number of cases. QuickCheck will reject these test cases, since they are trivially true and thus do not really assert anything about the *merge* function. Consequently it exhausts the maximum number of 1000 tests before it finds 100 test cases that pass the precondition.

2. The test cases that QuickCheck finds that satisfy the precondition are *very likely* to be extremely short lists, often with only 0, 1 or 2 elements. This is simply a result of the fact that the probability that a random list is sorted exponentially decreases with the length of the randomly generated list. Consequently there is a heavy bias towards short lists, making it unlikely that we test the *insert* function on any larger inputs.

The common approach to deal with this problem is to take matters in our own hands and to define a generator that only generates elements that satisfy the constraints we put on our test data. For sorted lists we might write the following QuickCheck generator:

```
gen_sorted :: Gen [Int]
gen_sorted = arbitrary ≫= return . diff . map abs
  where diff :: [Int] → [Int]
        diff []     = []
        diff (x:xs) = x:map (+x) (diff xs)
```

For sorted lists, defining a custom generator is a manageable task. However if we require our test data to satisfy complicated constraints, defining these generators can become very difficult; even synthesizing well-typed lambda terms is a surprisingly tricky problem [32, 19, 9].

We present a possible solution to this problem by observing that the desired constraints on test data can often be expressed as an *indexed family* that ranges over the type of our test values. For example, we can define an indexed family that is indexed with a list of numbers, which is only inhabited if its index list is a sorted list.

```
data Sorted : (xs : List ℕ) → Set where
  nil    : Sorted []
  single : ∀ {n} → Sorted (n :: [])
  step   : ∀ {n m xs} → n ≤ m → Sorted (m :: xs)
                               → Sorted (n :: m :: xs)
```

Given a value of type `Sorted xs`, it is easy to convert it to a value of type `List ℕ`. This means that if we are able to generate values of type `Sorted xs`, we are able to generate values of type `List ℕ` for which we know that they are sorted, even though this is no longer guaranteed by the type system of the host language. By generalizing this approach and finding a way to generate values of arbitrary indexed families, we are able to generate constrained test data. There is some existing work in this context [16, 25], but we take a novel approach by considering various type universes that provide a generic description of a set of (indexed) datatypes, and deriving generators from the codes in these universes

1.2 Research question and goals

The main research question that this thesis attempts to answer is the following:

How can we derive generators for arbitrary indexed families?

By obtaining a way to generically generate values of indexed families, we hope to be able to generate constrained test data without having to define custom generation procedures. The goal is to find a generic solution which covers many different indexed families, to provide a formalization of this solution in Agda, and to implement our solution in Haskell to demonstrate the feasibility of our approach. We aim to devise a framework that is highly flexible and allows us to describe many different kinds of constrained data.

1.3 Contributions

This thesis makes the following contributions:

- A formalization in Agda of enumerative generators for *regular datatypes*, together with a proof that these generators satisfy a completeness property.

- A formalization in Agda of enumerative generators for inductive families that can be described as an *indexed container*.

- A formalization in Agda of enumerative generators for inductive families that can be described as an *indexed description*, together with a proof that they satisfy a completeness property.

- A small Haskell library that implements the enumerative generator for indexed description, and is able to generate constrained test data.

This thesis consists of two main parts: one describing the theoretical model we have developed, and one describing our implementation. The first two chapters introduce the topic and provide background and prerequisites necessary for understanding our work. Chapter 3, 4 and 5 describe our Agda model, each considering a different (and increasingly expressive) type universe. In chapter 6 we describe how the ideas developed in our theoretical model can be adapted to Haskell by developing a small library for the generation of constrained test data. Finally, we conclude in chapter 7 with a discussion of our work and the academic context surrounding it, together with some suggestions for future work.

1.4 Deliverables

This thesis is accompanied by the following deliverables:

- A Github repository containing the Agda model

- A Github repository containing the Haskell library.

- An extended abstract based on the contents of chapters 3 through 5, submitted and accepted to *TyDe 2019*.

- An extended abstract based on the contents of chapter 6, submitted and accepted to *ICFP SRC 2019*.

1.5 Methodology

We use Agda [31], a programming language/proof assistant based on Martin-Löf type theory, for the formalization of the type universes and accompanying proofs that the generators we derive from them are correct. We use the functional programming language Haskell [22] to implement a library based on this formalization. To enforce as much static guarantees as possible we use various extensions of *The Glasgow Haskell Compiler* [2], most notably *TypeFamilies* [35], *DataKinds* [43] and *GADTs* [20].

Notational conventions   This thesis contains many snippets with both Agda and Haskell code. While it will generally be clear from the context which language is shown, they can be easily be distinguished by by the fonts and color schemes used as well. Below is a small example in respectively *Haskell* and `Agda`:

```haskell
data Bin a = Leaf | Node (Bin a) a (Bin a)
```

```agda
data Bin (A : Set) : Set where
  leaf : Bin A
  node : Bin A → A → Bin A → Bin A
```

Universe polymorphism   All Agda snippets throughout this thesis have been compiled with the `{-# OPTIONS --type-in-type #-}` pragma, since this makes them much more readable than their universe polymorphic counterparts. In the Agda model accompanying this thesis we have avoided this option wherever possible in order to keep our development consistent.

2 Background & Prerequisites

In this section, we will briefly discuss some of the theoretical background that is relevant for the work presented in this thesis, and explain some aspects of our Agda development that are necessary prerequisites to understanding the remainder of this thesis. To summarize, we touch upon the following subjects:

- *Type theory* and its relationship with *logic* through the *Curry-Howard correspondence*
- *Datatype generic programming* using *type universes* and the design patterns associated with datatype generic programming.
- The type of generators used across the development, together with the completeness property we use to assert their correctness.

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*. The *Curry-Howard equivalence* establishes an isomorphism between *propositions and types* and *proofs and terms* [38]. This means that for any type there is a corresponding proposition, and any term inhabiting this type corresponds to a proof of the associated proposition. Types and propositions are generally connected using the mapping shown in table 2.1.

In general, we may prove any proposition by writing a program that inhabits its corresponding 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 $P \to \bot$. This essentially says that if P is true, then we can derive a contradiction, thus it must be false. Some properties are provable in predicate logic, but not in type theory. The most notable example of this is the *law of excluded middle*, stating that every property must be either true or false.

Logic	Type Theory
False	$\bot$
True	$\top$
$P \vee Q$	$P + Q$
$P \wedge Q$	$P * Q$
$p \Rightarrow Q$	$P \to Q$
$\exists\, x \,.\, P\, x$	$\Sigma_{(x:A)} P(x)$
$\forall\, x \,.\, P\, x$	$\Pi_{(x:A)} P(x)$

Table 2.1: Correspondence between constructive logic and type theory

2.2 Agda

Agda is a programming language and proof assistant based on Intuitionistic type theory [31]. 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. Agda uses the Curry-Howard correspondence to model propositions as types and proofs as programs.

2.3 Generic programming and type universes

In *Datatype generic programming*, we define functionality for an entire class of types at once by induction over their structure. This means that generic functions will not take *values* of a particular type as input, but rather a *code* that describes the structure of a 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 `𝒰` representing the structure of types, often referred to as the *Universe*.

2. Next, we define a semantics of the form `⟦_⟧ : 𝒰 → K` that associates codes in `𝒰` with an appropriate type of kind `K`.

3. Finally, we define a fixed point operation of type `(u : 𝒰) → Set` that calculates the fixpoint of `⟦ u ⟧`.

The semantics and fixpoint operation should be designed such that if a code `u : 𝒰` represents a type `T`, then the fixpoint of `u` is 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* [31] contains an in depth explanation of how this can be done in Agda. In general, a datatype generic function is supplied with a code `u : 𝒰`, and returns a function whose type is dependent on the code it was supplied with. For example, we might write the following function that implements decidable equality for all types in the universe 𝒰.

```
_≟_ : ∀ {u : 𝒰} → (x : Fix u) → (y : Fix u) → Dec (x ≡ y)
```

Allowing us to obtain a decision procedure for a type by instantiating ≟ with the associated code in 𝒰.

2.3.2 Isomorphisms

Throughout this thesis, we will often talk about isomorphisms between types. An isomorphism between two types `A` and `B` asserts a one to one correspondence (or a *bijection*) between their values. We use the following formal definition:

```
record _≃_ (A B : Set) : Set where
  field
    from : A → B
    to : B → A
    iso₁ : ∀ {x : A} → to (from x) ≡ x
    iso₂ : ∀ {y : B} → from (to y) ≡ y
```

Isomorphisms form an equivalence relation between types, meaning that they are *reflexive*, *symmetric* and *transitive*:

```
≃-refl  : ∀ {A} → A ≃ A
≃-sym   : ∀ {A B} → A ≃ B → B ≃ A
≃-trans : ∀ {A B C} → A ≃ B → B ≃ C → A ≃ C
```

We will refrain from explicitly including any isomorphism between types in this thesis, but wherever we mention that an isomorphism exists between two types we have formalized said isomorphism in our theoretical model.

Listing 2.1: The abstract generator type

```
data Gen {I} : (Set) → (I → Set) → I → Set where
  Pure : ∀ {A T i}       → A → Gen A T i
  Ap : ∀ {A B T i j}     → Gen (B → A) T i → Gen B T j
                         → Gen A T i
  Bind : ∀ {A B T i j}   → Gen A T j → (A → Gen B T i)
                         → Gen B T i
  Or : ∀ {A T i}         → Gen A T i → Gen A T i
                         → Gen A T i
  μ  : ∀ {A}             → (i : I) → Gen (A i) A i
  None : ∀ {A T i}       → Gen A T i
  Call : ∀ {J S T i}     → ((j' : J) → Gen (S j') S j')
                         → (j : J) → Gen (S j) T i
```

2.4 Generators

In this thesis, we use the datatype shown in listing 2.1 as an abstract representation of the concept of generators.

The abstract generator type is essentially a deep embedding of the functions exposed by the *Applicative*, *Monad* and *Alternative* typeclasses, with added constructors to mark recursive positions and invoke other generators. We can map values of this abstract generator type to any desired concrete generator type. The reason for this separation between abstract and concrete generators is twofold:

1. It becomes much easier to convince Agda's termination checker that our generators are terminating, since we do not require any recursive calls or a fixpoint operation to write recursive generators, shifting this burden to the interpretation of the abstract generator type.

2. We potentially add a bit of flexibility by delaying the point at which we have to commit to a particular generator type, which might make interfacing with existing libraries easier.

In practice, we will rarely use the constructors of the `Gen` type. Rather, we use the functions exposed by the associated typeclasses. For example, we can define an abstract generator for the `Fin` type as follows:

```
fin : (n : ℕ) → Gen (Fin n) Fin n
fin zero    = empty
fin (suc n) = ⦇ zero      ⦈
            ∥ ⦇ suc (μ n) ⦈
```

2.4.1 Generator interpretations

A consequence of this design is that we need to transform abstract generators to a concrete instantiation before we can prove properties about their behavior. We will for our proofs use an enumerative interpretation, not unlike SmallCheck's [34] *Series* typeclass, where generators are functions from recursive depth to a list of values. The definition of this interpretation is shown in listing 2.2.

Listing 2.2: Enumerative interpretation of abstract generators

```
enumerate : ∀ {I A T} → ((i : I) → Gen (T i) T i)
                      → (i : I) → Gen A T i → ℕ → List A
enumerate tg i g                zero     = []
enumerate tg i (Pure x)         (suc n)  = x ∷ []
enumerate tg i (Ap {j = j} g₁ g₂) (suc n)  =
  concatMap (λ f → map f (enumerate tg j g₂ (suc n)))
            (enumerate tg i g₁ (suc n))
enumerate tg i (Bind {j = j} g₁ fg) (suc n) =
  concatMap (λ x → enumerate tg i (fg x) (suc n))
            (enumerate tg j g₁ (suc n))
enumerate tg i (Or g₁ g₂)       (suc n)  =
  merge (enumerate tg i g₁ (suc n))
        (enumerate tg i g₂ (suc n))
enumerate tg i (μ .i)           (suc n)  =
  enumerate tg i (tg i) n
enumerate tg i None             (suc n)  = []
enumerate tg i (Call g j)       (suc n)  =
  enumerate g j (g j) (suc n)
```

2.4.2 Generator completeness

We formulate the desired completeness property, which assert that that generators do not ”skip” any values. We formalize this property as follows in Agda:

```
Complete : ∀ {I} {A : I → Set} → ((i : I) → Gen (A i) A i) → Set
Complete {I} {A} gen =
  ∀ {i : I} {x : A i} → ∃[ n ] (x ∈ enumerate gen i (gen i) n)
```

Basically this property asserts that all possible values of the type produced by a generator will occur in the enumeration at some point.

2.4.3 Generators for non-indexed types

Although our generator infrastructure is primarily designed for usage with indexed types, we can adapt it to work for non-indexed types. We do this by choosing a trivial index, and generating values of type `⊤ → A` instead of generating values of the non-indexed type `A` directly. This allows us to write a generator for natural numbers:

```
nat : Gen ℕ (λ { tt → ℕ }) tt
nat = ⦇ zero        ⦈
    ∥ ⦇ suc (μ tt) ⦈
```

We occasionally take a more liberal approach to the notation of non-indexed generators, writing `Gen A A` instead of `Gen A (λ { tt → A}) tt` and `μ` instead of `μ tt`.

Part I

Theoretical Model

3

Regular Types

We can describe a large class of recursive algebraic data types as a *regular types*. In this section we describe this universe together with its semantics, and demonstrate how we may define generators for regular types by induction over their codes. We will then prove that these derived generators satisfy our completeness property.

3.1 Universe definition

Though the exact definition may vary across sources, the universe of regular types is generally regarded to consist of the *empty type* (containing *no* inhabitants), the unit type (containing exactly *one* inhabitant) and constants types (which simply refer to another type). Regular types are closed under both *products* (representing pairing of types) and *coproducts* (representing a choice between types). Listing 3.1 shows the Agda datatype that we use to represent codes in this universe, with the associated semantics of type `Reg → Set → Set` being shown in listing 3.2.

Listing 3.1: The universe of regular types

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

The semantics map a code to a functorial representation of the datatype described by that code, 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 find this fixpoint with the following fixpoint operation:

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

Listing 3.2: 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
⟦ K s     ⟧ r = s
```

Example Let us consider the type of natural numbers:

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

Nat exposes two constructors: the nullary constructor zero, and the unary constructor suc that takes one recursive argument. We can view this type then as a coproduct (or choice) between a unit type, representing zero, and a recursive position, representing the recursive argument of the suc constructor.

```
ℕ' : Set
ℕ' = Fix (U ⊕ I)
```

We convince ourselves that ℕ' is indeed equivalent to ℕ by defining an isomorphism of type ℕ ≃ ℕ'.

In general, we say that a type is regular if and only if we can provide a proof that it is isomorphic to the fixpoint of some code c of type Reg. We use a record to capture this notion, which holds a code and an value that witnesses the isomorphism between the fixpoint of this code and the regular type A.

```
record Regular (A : Set) : Set where
  field
```

```
code : Reg
iso : A ≃ Fix code
```

By instantiating Regular for a type A, we may use any generic functionality by leveraging the isomorphism stored with the record Regular A.

3.2 Deriving generators

We can derive generators for all regular types at once by induction over their associated codes. In section 3.4 we will prove that the generators we derive from codes satisfy our completeness property under the enumerative interpretation we defined in section 2.4.

3.2.1 Performing induction over codes

In our initial approach, we might be to try to define a generator that produces values of type Fix c. Unfortunately, this will not work. By choosing Fix c as the type of elements generated, we implicitly imposes the restriction that any I in c refers to Fix c. This restriction is problematic in some cases, specifically when encountering a product or coproduct. In that case, we destruct a code c into two smaller codes c₁ and c₂. Calling our deriving function on these codes will yield two generators, one producing values of type Fix c₁ and the other producing values of type Fix c₂. It is then not possible to combine these generators into a single generator producing values of type Fix c: the recursive positions in the subgenerators refer to different types!

To remedy this, we make a distinction between the code we are doing induction over, c, and the code which describes the type that recursive positions in c refer to, c'. Furthermore, we do not produce elements of type Fix c, but rather of type ⟦ c ⟧ (Fix c') (i.e. values of the type given by the semantics of c, but recursive positions refer to the type described by c'). When calling our derivation function with two equal codes, the values produced will be isomorphic to Fix c! This results in the following type signature of our generator deriving function:

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

This step allows us to perform induction over the first input code, while still being able to have recursive positions refer to the correct *top-level code*. The first and second type parameter (respectively describing the type we are generating, and the type of recursive positions) of Gen are consequently distinct, with the second type parameter being isomorphic to Fix c'.

3.2.2 Composing generic generators

Now that we have the correct type for `deriveGen` in place, we can begin to define it We do this on a case by case basis, describing how to derive generators for each of the constructors of the `Reg` datatype.

The empty (Z) and unit (U) type

We start with the generic generators for the `Z` and `U` constructors. Recall that the generators we derive from these constructors should produce *all* inhabitants of the type given by their semantics.

```
deriveGen Z c' = empty
deriveGen U c' = ⦇ tt ⦈
```

In case of both `Z` and `U` this requirement is trivially fulfilled. For the `Z` combinator, we yield a generator that produces no elements, since its semantics is the empty type (`⊥`). As for the `U` combinator, `⟦ U ⟧ (Fix c')` equals the unit type (`⊤`), so we need to return a generator that produces all inhabitants of `⊤`, which is only the value `tt`. We get a generator that does this by lifting `tt` into the generator type.

Recursive positions (I)

We mark a recursive position in a generator with the μ constructor. However, given the previously defined type signature for `deriveGen`, μ is a generator that produces elements of type `⟦ c' ⟧ (Fix c')`. We require that the generator derived from the `I` constructor produces elements of type `⟦ I ⟧ (Fix c')`, which by definition of `⟦_⟧` equals `Fix c'`. This means that we need to apply the fixpoint wrapper `In` over the elements produced by μ:

```
deriveGen I c' = ⦇ In μ ⦈
```

Products (`⊗`) and coproducts (`⊕`)

For products and coproducts, we can quite easily define their generators by recursing on the left and right subcodes now that we have the correct type for `deriveGen` in place. We then only need to combine these generators in an appropriate way. We do this respectively building a product type out of the elements produced by the subgenerators and by marking a choice between the generators derived from the subcodes.

```
deriveGen (c₁ ⊕ c₂) c' =
  ⦇ inj₁ (deriveGen c₁ c') ⦈ ∥ ⦇ inj₂ (deriveGen c₂ c') ⦈

deriveGen (c₁ ⊗ c₂) c' =
  ⦇ deriveGen c₁ c' , deriveGen c₂ c' ⦈
```

Of course, the exact way in which the elements of subgenerators are combined still depends on how we interpret the abstract generator type; here we only describe these operations in terms of the functions exposed by *Applicative* and *Alternative*.

Wrapping up

We have defined a function that derives generators from codes in the universe of regular types (barring constant types, with with we will deal in section 3.3). We need to take one final step before we can use `deriveGen` for all regular types. Any vallue `Regular A` holds an isomorphism `A ≃ Fix c`, so we need to wrap the resulting generator in the `In` constructor, which we can only do if `deriveGen` is called *with two equal codes*. We use the following function to perform this initial call to `deriveGen`, and to wrap the values produced by the resulting generator in the fixpoint operation:

```
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 ⊕ I`, and applying an isomorphism of type `ℕ ≃ ℕ'` to the resulting generator:
>
> ```
> genℕ : Gen ℕ ℕ
> genℕ = ⦇ (_≃_.to ℕ≃ℕ') (Call (genericGen (U ⊕ I))) ⦈
> ```

We use the following function to define a generator for any type `A` for which there is an instance argument `Regular A` in scope:

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

3.3 Constant types

Constant types present a bit of a challenge, since the code `K s` can carry any type in `Set`. This means that we know nothing about the type `s` whatsoever. Since we have no general procedure for deriving generators for arbitrary types in `Set`, we need to either restrict `s` to a set of types for which we *can* derive generators (e.g. regular types), or have the user supply generators for all constant types in a code. We choose the latter approach in order to retain the flexibility that comes with the ability to refer to arbitrary types.

3.3.1 Metadata structure

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 parameterize `deriveGen` with a metadata structure that is indexed by the code we are inducting over, carrying generators for every constant type used in said code. The definition of this metadata structure is shown in listing 3.3.

Listing 3.3: 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 purposefully keep the type of information stored for constant types abstract, as we will need to record information beyond generators when proving completeness for the generators produced by `deriveGen`.

3.3.2 Deriving a generator for constant types

Given the definition of the metadata structure, we augment `deriveGen` with an extra parameter that stores generators for every constant type in a code:

```
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:

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

All cases for existing constructors remain the same, except for the fact that the metadata parameter distributes over recursive calls in the case of products and coproducts.

With this, we have completed the definition of `deriveGen`.

3.4 Proving completeness

We set out to prove that the derived generators satisfy our completeness property. Obviously, this relies on the generators supplied by the programmer being complete as well.

We start the proof by instantiating the completeness property formulated in listing 2.1 with `deriveGen` to obtain the definition of the theorem that we will prove:

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

We explicitly distinguish the codes `c` and `c'` to (again) be able to construct the proof by performing induction over the input code `c`. The reasoning behind this is very much the same as the reasoning behind the definition of `deriveGen` itself. If we invoke this lemma with two equal codes, we may utilize 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. We do not show this proof here explicitly, but we have constructed a proof of the following statement in the Agda development:

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

Which we need to generalize the proof to all types which are isomorphic to some code `c : Reg`.

3.4.1 Proof structure

The completeness proof roughly follows the following steps:

- First, we prove completeness for the individual constructors of the `Reg` type.
- Next, we assemble a suitable metadata structure to carry the required proofs for constant types in this code.
- Finally, we generalize the proof over our generic generator to a proof that ranges over all types `A` that are isomorphic to the fixpoint of some code `c : Reg`.

3.4.2 Combinator correctness

We start our proof by asserting that the generators derived from the individual constructors of the Reg datatype are complete. That is, we show that for every constructor of Reg the derived generator produces all values of the type given by the semantics of that constructor.

Empty (Z) and unit (U) types

In the case of Z and U, completing the completeness proof is relatively 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 ⊥ is trivially complete: we simply close this branch with an absurd pattern. In the case of U we simply need to show that interpreting pure tt returns a list containing tt, which we can do by returning a trivial proof that tt is an element of the singleton list [tt].

Recursive positions (I)

The proof that a recursive position μ is interpreted to a complete enumeration is simply the induction hypothesis, which states that deriveGen c' c' is complete. A subtlety here is that we *must* pattern match on In x, otherwise Agda's termination checker will flag the recursive call.

```
deriveGen-Complete {I} {c'} {In x}
  with deriveGen-Complete {c'} {c'} {x}
... | prf = {  }?
```

We can complete this definition by proving a lemma that asserts that mapping In over a generator preserves completeness:

```
lemma-In : ∀ {x g g'}
  → ∃[ n ] (x ∈ enumerate g g' n)
  → ∃[ n ] (In x ∈ enumerate (⦇ In x ⦈) g' n)
```

Products and coproducts

Things become a bit more interesting once we move to products and coproducts, since this requires us to prove that the way in which we combine subgenerators satisfies completeness under our enumerative interpretation. In both cases, this proof follows a similar structure:

1. Obtain completeness proofs for the subgenerators with recursive calls to `deriveGen-Complete`
2. Construct a lemma that asserts that the enumerative interpretation of generators preserves completeness
3. Invoke this lemma to complete the definition

Coproducts To find out what lemma we need to prove completeness for the generators derived from coproducts, we observe the following equality by unfolding the defintions of `enumerate` and `deriveGen`:

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

The generators on the right hand side of the equation are virtually the same as the recursive calls we make, modulo the `inj₁` and `inj₂` constructors we map over them to unify their result types. We can obtain a proof of completeness for the right hand side of this equality by proving the following two lemmas about the `merge` function we use to combine the results of the subgenerators of a 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
```

Proofs for these lemmas can readily be extended to a proof that if the left and right subgenerator are complete under the enumerative interpretation, then the interpretation of their coproduct (which is a call to `merge`), is also complete, simply by pairing them with the depth value returned by the recursive call.

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

We can prove completeness for 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 ⦈
```

We can again extend this lemma to a proof that the enumerative interpretation of product types is completeness preserving. In section 3.4.4 we describe in more detail how an appropriate depth value can be obtained.

3.4.3 Completeness for constant types

Since our completeness proof relies on completeness of the supplied generators for constant types, we need the programmer to supply a completness proof for the generators stored in the provided metadata structure. To this end, we parameterize the completeness proof over a metadata structure that carries both generators for all constant types in a code, and a proof that these generators are complete. We express the relation between generator and proof with a dependent pair, using the following type synonym to describe the type of this metadata parameter:

```
ProofMD : Reg → Set
ProofMD c = KInfo (λ S → Σ[ g ∈ Gen S S ]
                  (∀ {x} → ∃[ n ] (x ∈ enumerate 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 metadata structure. This will allow us to map a metadata structure containing generators and proofs to a metadata structure containing only generators.

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

We only present the case for constant types; in all other cases we simply distribute the mapping operation over all recursive positions. Given a definition of KInfo-map, we can take the first projection of the metadata input to deriveGen-Complete, and use the resulting metadata structure as input to deriveGen. We define a type synonym to describe this mapping operation:

```
◂_ : ∀ {c : Reg} → KInfo (λ A → Σ[ g ∈ Gen A A ]
                            (∀ {x} → ∃[ n ] (x ∈ enumerate g g n))) c
                  → KInfo (λ A → Gen A A) c
◂ m = KInfo-map proj₁ m
```

Which results in the following final type of deriveGen-Complete.

```
deriveGen-Complete : (c c' : Reg)
  → (i : ProofMD c) → (i' : ProofMD c')
  → ∀ {x} → ∃[ n ] (x ∈ enumerate (deriveGen c c' (◂ i))
                                  (deriveGen c' c' (◂ i')) n)
```

By expressing the relation between the metadata structure supplied to the proof and the metadata structure supplied to deriveGen explicitly in the proof's type signature, Agda is able to infer that the completeness proofs range over the generators that were supplied to deriveGen. This allows us to complete the proof for constant types simply by returning the proof that is stored in the metadata structure.

3.4.4 Generator monotonicity

There is one crucial detail we ignored when describing how to prove completeness for generators derived from product types. Since existential quantification is modelled in type theory as a dependent pair, we have to explicitly supply the depth at which an element occurs in an enumeration when proving completeness. A problem, however arises when choosing a depth value for generators derived from product types. We combine values of both subgenerators in a pair, so at what depth does this pair occur in the enumeration of the combined generator? Generally, we say that the recursive depth of a pair is the maximum of the depth of its components. Suppose the first component occurs at depth n, and the second at depth m. The depth of the pair is then `max n m`. However, the second components of the returned completeness proofs respectively have the type `x ∈ enumerate ... n` and `x ∈ enumerate ... m`. In order to unify their types, we need a lemma that transforms a proof that some value `x` occurs in the enumeration at depth `k` into a proof that `x` occurs in the enumeration at depth `k'`, given that $k \leq k'$. In other words, the set of values that occurs in an enumeration monotonously increases with the enumeration depth. We thus require a proof of the following lemma in order to finish the completeness proof:

```
n ≤ m → x ∈ enumerate (deriveGen c c' (◂ i))
                      (deriveGen c' c' (◂ i')) n
      → x ∈ enumerate (deriveGen c c' (◂ i))
                      (deriveGen c' c' (◂ i')) m
```

We do not show the definition of this proof here, but it can be completed using the exact same proof structure we used for the completeness proof.

3.4.5 Extending completeness to all regular types

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 ≃ 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 ∈ enumerate g g n)
```

With which we have shown that if a type is regular, we can derive a complete generator producing elements of that type.

Conclusion

In this chapter, we have shown how generators can be derived from codes in the universe of regular types. While this is not necessarily a new result (e.g. SmallCheck does this as well), we have also proven that these generators are complete under an enumerative interpretation, meaning that they are guaranteed to produce every inhabitant of the type they range over at some point.

Futhermore, the work done to establish this generic generator and the accompanying proof provides a solid basis for extending this result to generic generators for more expressive type universes. As we will see in the upcoming chapters, the approach described in this chapter is to a large extent applicable to other type universes as well.

Although we can describe many familiar datatypes with a code in the universe of regular types, there are some limitations. Most notably, we cannot describe any family of mutually recursive types. The way the universe is set up includes the implicit assumption that all occurrences of `I` reference the same type. If we attempt to describe a datatype that is a composite of more than one recursive algebraic datatype, such as for example the type of *rose trees*:

```
data Rose (A : Set) : Set where
  node : List (Rose A) → Rose A
```

The other obvious shortcoming is that this universe only allows us to describe non-indexed datatypes. In the following chapters we will consider two type universes that *can* do this.

4
Indexed Containers

This chapter discusses the universe of *indexed containers* [5], which provide a generic framework to describe indexed datatypes that can be defined by induction on their index type. Examples of datatypes we can describe using this universe include finite sets (`Fin`) and vectors (`Vec`). In this chapter, we give the definition of this universe together with its semantics and a few examples, and show how a generic generator may be derived for indexed containers.

Unfortunately, we were not able to construct a completeness proof for the generators derived from this universe, hence we will give a short outline of why we were unable to do so using the approach we used for regular types.

4.1 Universe definition

We choose to follow the representation used by Dagand in *The Essence Of Ornaments* [13], which provides an good introduction to indexed containers, alongside various examples. Just as in the previous chapter, we follow the pattern of first defining a datatype describing codes in the universe before giving the semantics and fixpoint operation.

4.1.1 Defintion and semantics

To give the reader some context, we will first introduce *W-types*, which are an alternative to regular types for describing non-indexed datatypes. Indexed containers describe indexed types using an approach that is very similar to how W-types describe non-indexed types. We show how these universes relate, and how we may view indexed containers as an extension to W-types.

W-types

Introduced by Per Martin-Löf [26], *W-types* abstract over tree-shaped data structures, such as natural numbers or binary trees. W-types are defined by their *shape* and *position*, describing respectively the set of constructors and the number of recursive positions.

Perhaps the best known definition of W-types is using an inductive datatype, with one constructor taking a shape value, and a function from position to W-type:

```
data WType (S : Set) (P : S → Set) : Set where
  sup : (s : S) → (P s → WType S P) → WType S P
```

However, we can use an alternate definition where we separate the universe into codes, semantics and a fixpoint operation (listing 4.1)

Listing 4.1: W-types defined with separate codes and semantics

```
record WType : Set where
  constructor _~_
  field
    S : Set
    P : S → Set

⟦_⟧sup : WType → Set → Set
⟦ S ~ P ⟧sup r = Σ[ s ∈ S ] (P s → r)

data Fix (w : WType) : Set where
  In : ⟦ w ⟧sup (Fix w) → Fix w
```

We take this extra step for two reasons:

1. To unify the definition of W-types with the design pattern for type universes we described in section 2.3.1.

2. To make the similarities between W-types and indexed containers more apparent.

Example Let us again consider natural numbers as an example. We can define the following W-type that is isomorphic to ℕ:

```
Wℕ : Set
Wℕ = Fix (Bool ~ λ { false → ⊥ ; true → ⊤ })
```

The ℕ type has two constructors, hence our shape is a finite type with two inhabitants (`Bool` in this case). We then map `false` to the empty type, signifying that `zero` has no recursive subtrees, and `true` to the unit type, denoting that `suc` has one recursive

> subtree.

Indexed containers

Indexed containers extend the notion of a W-type by parameterizing the shape and position over the index type, and including a typing discipline that describes the indices of recursive subtrees. Types are described using *Signatures*, which are a triple of *operations*, *arities* and a *typing discipline*. Their definition is shown in listing 4.2.

Listing 4.2: Signatures

```
record Sig (I : Set) : Set where
  constructor _◃_∣_
  field
    Op : (i : I) → Set
    Ar : ∀ {i} → (Op i) → Set
    Ty : ∀ {i} {op : Op i} → Ar op → I
```

The operations of a signature correspond to a W-type's *shape*, and its arity corresponds a W-type's *position*. The semantics of a signature is, just as for a W-type, a dependent pair, with the first element being a choice of operation, and the second element a function mapping arities to an appropriate recursive type. Contrary to the semantics of W-types, which map a code to a value in `Set → Set`, the semantics of a signature are parameterized over the index type, meaning they map a signature to a value in `(I → Set) → (I → Set)`. The semantics are shown in listing 4.3.

Listing 4.3: The semantics of indexed containers

```
⟦_⟧ : ∀ {I} → Sig I → (I → Set) → I → Set
⟦ Op ◃ Ar ∣ Ty ⟧ r i =
  Σ[ op ∈ Op i ] ((ar : Ar op) → r (Ty ar))
```

Consequently, the fixpoint operation needs to be lifted from `Set` to `I → Set` as well. The required adaptation follows naturally from the definition of the semantics:

```
data Fix {I : Set} (𝓢 : Sig I) (i : I) : Set where
  In : ⟦ 𝓢 ⟧ (Fix 𝓢) i → Fix 𝓢 i
```

It is worth noting that, since `Set ≅ ⊤ → Set`, we can describe non-indexed datatypes as an indexed container by choosing `⊤` as the index type. More precisely, there exists a bijection between `WType` and `Sig ⊤`, such that for every `W : WType` and `S : Sig ⊤`, `Fix W` and `Fix S tt` are isomorphic.

Proving isomorphisms

One thing to keep in mind when defining signatures for types is that part of their semantics is a dependent function type. This means that proving an isomorphism between a signature and the type it represents requires some extra work. More specifically, we need to postulate a variation on *extensional equality* for function types that establishes equality between two dependent functions:

```
funext' : ∀ {A : Set} {B : A → Set} → (f g : (a : A) → B a)
        → (∀ {x} → f x ≡ g x) → f ≡ g
```

We need this postulate to define isomorphisms both for W-types as well as indexed containers.

4.1.2 Examples

We consider a few example datatypes represented as indexed containers, in order to get a feel for how we can represent types in this universe.

Natural numbers

We start by defining a suitable set of operations. The `ℕ` datatype has two constructor, so we return a type with two inhabitants. We use `⊤` as the index of the signature, since `ℕ` is a non-indexed datatype.

```
Op-Nat : ⊤ → Set
Op-Nat tt = ⊤ ⊎ ⊤
```

Next, we map each of those operations to the right arity. The `zero` constructor has no recursive branches, so its arity is the empty type (`⊥`), while the `suc` constructor has a single recursive argument, so its arity is the unit type (`⊤`).

```
Ar-Nat : Op-Nat tt → Set
Ar-Nat (inj₁ tt) = ⊥
Ar-Nat (inj₂ tt) = ⊤
```

Since the index type has only one inhabitant, the associated typing discipline just returns tt in all cases. We bring all these elements together into a single signature, for which we can show that its fixpoint is isomorphic to ℕ.

```
ℕSig : Sig ⊤
ℕSig = Op-Nat ◃ Ar-Nat ∣ λ _ → tt
```

The signature for natural numbers is quite similar to how we would represent them as a W-type.

Finite sets

We consider the type of finite sets, Fin. Contrary to natural numbers, the set of available operations varies with different indices. That is, Fin 0 is uninhabited, so the set of operations associated with index 0 is empty. A value of type Fin (suc n) can be constructed using both suc and zero, hence the set of associated operations has two elements:

```
Op-Fin : ℕ → Set
Op-Fin zero    = ⊥
Op-Fin (suc n) = ⊤ ⊎ ⊤
```

The arity of the Fin type is exactly the same as the arity of ℕ, with the exception of an absurd pattern in the case of index zero.

```
Ar-Fin : ∀ {n} → Op-Fin n → Set
Ar-Fin {zero}  ()
Ar-Fin {suc n} (inj₁ tt) = ⊥
Ar-Fin {suc n} (inj₂ tt) = ⊤
```

Recall the type of the suc constructor: Fin n → Fin (suc n). The index of the recursive argument is one less than the index of the constructed value. The typing discipline describes this relation between index of the constructed value, and indices of recursive arguments. In the case of Fin, this means that we map suc n to n, if the index is greater than 0, and the operation corresponding to the suc constructor is selected.

```
Ty-Fin : ∀ {n} {op : Op-Fin n} → Ar-Fin op → ℕ
Ty-Fin {zero}  {()} ar
Ty-Fin {suc n} {inj₁ tt} ()
Ty-Fin {suc n} {inj₂ tt} tt = n
```

Again, we combine operations, arity and typing into a signature:

```
FinSig : Sig ℕ
FinSig = Op-Fin ◃ Ar-Fin ∣ Ty-Fin
```

Vectors

One aspect we have not yet addressed is how to represent types that refer to other types, such as `Vec a`. Indexed containers do not have an explicit way to reference other types, such as regular types. Rather they include this kind of information as part of a type's operations. We consider the `Vec` type as an example, defining the following operations:

```
Op-Vec : ∀ {A : Set} → ℕ → Set
Op-Vec {A} zero    = ⊤
Op-Vec {A} (suc n) = A
```

Notice that we map `suc n` to `A`, indicating that the `::` constructor requires an argument of type `A`. The remainder of the signature is then quite straightforward:

```
Ar-Vec : ∀ {A} {n} → Op-Vec {A} n → Set
Ar-Vec {A} {zero} tt = ⊥
Ar-Vec {A} {suc n} op = ⊤

Ty-Vec : ∀ {A} {n} {op : Op-Vec {A} n} → Ar-Vec {A} op → ℕ
Ty-Vec {A} {zero}  {tt} ()
Ty-Vec {A} {suc n} {op} tt = n

VecSig : Set → ℕ → Sig ℕ
VecSig A n = Op-Vec {A} ◃ Ar-Vec {A}
                        ∣ λ {i} {op} → Ty-Vec {op = op}
```

In fact, the arity and typing of `Vec` are almost the same as those of `Fin`.

4.2 Deriving generators

In order to be able to derive generators from signatures, there are two additional steps we need to take: restricting the set of possible operations and arities, and defining *co-generators* for regular types.

4.2.1 Restricting operations and arities

The set of operations of a signature, `Op`, is a value in `Set`. This means that we again run into the problem that we have no way to generate values of type `Op i` without any further input of the programmer. The same problem occurs with arities. We solve this problem by restricting operations and arities to regular types. By doing this, we can reuse the generators we defined for regular types to generate operations and arities. This leads to the slightly altered variation on indexed containers shown in listing 4.4, where `FixR` and `InR` denote the fixpoint operation for regular types. This implies that the definition of signatures changes slightly as well.

Listing 4.4: Indexed containers with restricted operations and arities

```
record Sig (I : Set) : Set where
  constructor _◃_∣_
  field
    Op : (i : I) → Reg
    Ar : ∀ {i} → FixR (Op i) → Reg
    Ty : ∀ {i} {op : FixR (Op i)} → FixR (Ar op) → I

⟦_⟧ : ∀ {I} → Sig I → (I → Set) → I → Set
⟦ Op ◃ Ar ∣ Ty ⟧ r i =
  Σ[ op ∈ FixR (Op i) ] ((ar : FixR (Ar op)) → r (Ty ar))
```

Example We use the following operation, arity and typing to describe the `Fin` type as a restricted signature:

```
Op-Fin : ℕ → Reg
Op-Fin zero    = Z
Op-Fin (suc n) = U ⊕ U

Ar-Fin : ∀ {n} → FixR (Op-Fin n) → Reg
Ar-Fin {zero}  (InR ())
Ar-Fin {suc n} (InR (inj₁ tt)) = Z
Ar-Fin {suc n} (InR (inj₂ tt)) = U

Ty-Fin : ∀ {n} {op : FixR (Op-Fin n)} → FixR (Ar-Fin op) → ℕ
Ty-Fin {zero}  {InR ()}
Ty-Fin {suc n} {InR (inj₁ tt)} (InR ())
Ty-Fin {suc n} {InR (inj₂ tt)} (InR tt) = n
```

This definition does not differ too much from the previous one, except that we now pattern match on the fixpoint of some code in `Reg` instead of directly on an operation or arity.

4.2.2 Generating function types

To derive a generator from a signature, we need, in addition to generic generators for regular types, a way to generate function types whose input argument is a regular type. That is, we need to define the following function:

```
cogenerate :
  ∀ {A : Set} → (r r' : Reg) → (Gen A (⟦ r' ⟧R (FixR r') → A))
  → Gen (⟦ r ⟧R (FixR r') → A) (⟦ r' ⟧R (FixR r') → A)
```

We draw inspiration from SmallCheck's [34] *CoSeries* typeclass, for which instances can be automatically derived. Co-generators for constant types are to be supplied by a programmer using a metadata structure; we choose to not make this explicit in the type signature. An example definition of `cogenerate` is included in listing 4.5.

Listing 4.5: Definition of `cogenerate`

```
cogenerate Z          r' gen = empty
cogenerate U          r' gen = ⦇ (λ { x tt → x }) gen ⦈
cogenerate (r₁ ⊕ r₂) r' gen =
  ⦇ (λ { f g (inj₁ x) → f x ; f g (inj₂ y) → g y })
    (cogenerate r₁ r' gen) (cogenerate r₂ r' gen) ⦈
cogenerate (r₁ ⊗ r₂) r' gen =
  ⦇ uncurry (cogenerate r₁ r' (cogenerate r₂ r' gen)) ⦈
cogenerate I          r' gen =
  ⦇ (λ { f (InR x) → f x }) (μ {r'}) ⦈
```

Since the semantics of an indexed container contain a *dependent* function type we need to extend `cogenerate` to work for dependent function types as well.

```
Π-cogenerate : (r r' : Reg) →
  ∀ {P : (r r' : Reg) → ⟦ r ⟧R (FixR r') → Set}
  → ((x : ⟦ r ⟧R (FixR r'))
       → Gen (P r r' x) ((x : ⟦ r' ⟧R (FixR r')) → P r' r' x))
  → Gen ((x : ⟦ r ⟧R (FixR r')) → P r r' x)
        ((x : ⟦ r' ⟧R (FixR r')) → P r' r' x)
```

The type signature of `Π-cogenerate` may look a bit daunting, but it essentially follows the exact same structure as `cogenerate`. The only real difference is that the the result type of the generated functions may depend on the code we are inducting over, and that we do not take a generator as input, but rather a function from index to generator. The definitions of `Π-cogenerate` and `cogenerate` are virtually the same, but in order to prevent Agda from getting stuck while solving metavariables describing the indices of return types, we need to make the dependency between argument and result type explicit in the generator's type.

4.2.3 Constructing generators

We are now ready to construct a the generic generator for indexed containers. Recall that `genericGen c` returns a generator for the regular type represented by `c`. We can write a function `deriveGen` that derives generators from signatures:

```
deriveGen : ∀ {I : Set} → (S : Sig I)
              → (i : I) → Gen (Fix S i) (Fix S i)
deriveGen (Op ◃ Ar | Ty) i = do
  op ← Call (genericGen (Op i))
  ar ← Call (Π-cogenerate (Ar op) (Ar op) (λ ar → μ))
  pure (In (op , λ { (InR x) → ar x }))
```

The final generator is quite simple, really. We use the existing functionality for regular types to generate operations and arities, and return them as a dependent pair, wrapping and unwrapping fixpoint operations as we go along. The dependency between the first and second element of said pair is captured using by using the monadic structure of the generator type.

Conclusion

We have showed how we can describe a large set of indexed datatypes as *indexed containers*, and how we can derive a generator from the types in this universe by constructing a generator for function types, and restricting operations and arities to regular types which enables us to reuse the generator for regular types we derived in the previous chapter.

Unfortunately, we have not been able to assemble a completeness proof for the enumeration of `deriveGen`. As was the case with the completeness proof for regular types, we need to explicitly pattern match on the value for which we are proving that it occurs in the enumeration in order for the termination checker to recognize that the proof can be constructed in finite time. However, since part of the semantics of a signature is a function type, we would need to destruct a function type such that it is clear to the termination checker that we perform the recursive calls with a value that smaller than the original argument. It was unclear to us how we should proceed with the proof at this point.

With the derived generator for indexed containers we have obtained recipe for the generation of indexed families. To motivate the need to study yet another type universe in the next chapter, we consider a datatype for binary trees, that are indexed by their number of nodes:

```
data STree (A : Set) : ℕ → Set where
```

```
leaf : STree A 0
node : ∀ {n m} → STree A n → A → STree A m
               → STree A (suc (n + m))
```

.

When we set out to define a signature for `STree`, we run into a problem:

```
Op-STree : Set → ℕ → Reg
Op-STree _ zero    = U
Op-STree A (suc n) = K A

Ar-STree : ∀ {A n} → FixR (Op-STree A n) → Reg
Ar-STree {A} {zero} (InR tt) = Z
Ar-STree {A} {suc n} (InR x) = U ⊕ U

Ty-STree : ∀ {A n} {op : FixR (Op-STree A n)}
         → FixR (Ar-STree op) → ℕ
Ty-STree {A} {zero} {InR tt} (InR ())
Ty-STree {A} {suc n} {InR x} (InR (inj₁ tt)) = {  }?
Ty-STree {A} {suc n} {InR x} (InR (inj₂ tt)) = {  }?
```

We cannot define a typing discipline! Indexed containers assume a deterministic mapping from arity to recursive index, with no dependencies between the indices of different recursive subtrees. The `STree` type violates both these assumptions, hence we need a more expressive type universe in order to generically describe it.

5 Indexed Descriptions

To be able to represent arbitrary indexed families, we use the universe of *Indexed Descriptions*, as described by Dagand [12] in his PhD thesis. We structure this chapter in the same way as the previous two chapters, by first giving the definition and semantics of the universe, before showing how a generator can be derived from codes in this universe and proving that these generators are complete under our enumerative interpretation.

5.1 Universe definition

We give the definition of the universe, together with its semantics and fixpoint operation, before considering well-typed lambda terms as an example to demonstrate how we might model a more complex datatype in this universe and to show how we can capture datatypes that cannot be described as a regular type or indexed container.

5.1.1 definition & Semantics

Where indexed containers can be viewed as an extension to W-types, indexed description take the universe of regular types as a basis and extend it to be able to deal with more complex datatypes, adding the following elements:

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 `_⊕_` constructor.
3. A combinator, `` `Σ ``, denoting dependent pairs.
4. Recursive positions, `` `var ``, storing the index of recursive values.

This amounts to the Agda datatype describing indexed descriptions shown in listing 5.1. Types are not described by a value of type `IDesc I`, but rather as a function from index to description, `I → IDesc I`. There is no explicit constructor for constant types; they can be modelled as a dependent pair consisting of the type we want to refer to, and a function returning the unit type. Similarly, the universe does not contain a constructor representing the empty type. We simply encode it as a coproduct of zero constructors: `` `σ 0 λ() ``.

Listing 5.1: 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
```

We retain the product of two descriptions as a first order construct of the universe while including a generalized notion for coproducts, which does not present a choice between 2, but rather any possible number `n` of operations. The `Sl` datatype is used to select these operations, and is isomorphic to `Fin`. We will require a lot of pattern matches this datatype to build descriptions, and by using `Sl` over `Fin` we end up with slightly more succinct descriptions. The definition of `Sl` is included below:

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

The semantics associated with the `IDesc` universe is mostly derived from the semantics of the universe of regular types, the key difference being that we do not map codes to a functor `Set → Set`, but rather to a function of type `(I → Set) → Set`. The semantics are shown in listing 6.2.

Listing 5.2: Semantics of the IDesc universe

```
⟦_⟧ : ∀ {I} → IDesc I → (I → Set) → Set
⟦ `var i   ⟧ r = r i
⟦ `1       ⟧ r = ⊤
⟦ dₗ `× dᵣ ⟧ r = ⟦ dₗ ⟧ r × ⟦ dᵣ ⟧ r
⟦ `σ n T   ⟧ r = Σ[ sl ∈ Sl n ] ⟦ T sl ⟧ r
⟦ `Σ S T   ⟧ r = Σ[ s ∈ S ] ⟦ T s ⟧ r
```

We calculate the fixpoint of a description's semantics using the following fixpoint operation:

```
data Fix {I : Set} (φ : I → IDesc I) (i : I) : Set where
  In : ⟦ φ i ⟧ (Fix φ) → Fix φ i
```

Example We can describe the Fin datatype as follows using a code in the universe of indexed descriptions:

```
finD : ℕ → IDesc ℕ
finD zero    = `σ 0 λ()
finD (suc n) = `σ 2 λ
  { •     → `1
  ; (▻ •) → `var n
  }
```

If the index is zero, there are no inhabitants, so we return a coproduct of zero choices. Otherwise, we may choose between two constructors: zero or suc. Notice that we describe the datatype by induction on the index type. This is not required, although convenient in most cases. A different, but equally valid description, exists in which we use the `Σ constructor to explicitly enforce the constraint that the index is of the form suc n.

```
finD : ℕ → IDesc ℕ
finD = λ n → `Σ ℕ λ m → `Σ (n ≡ suc m) λ { refl →
  `σ 2 λ { •     → `1
         ; (▻ •) → `var n
         }}
```

We then have that Fix finD n is isomorphic to Fin n.

Of course, we could already describe the Fin type as an indexed container. Let us reconsider the STree type (section 4.2.3), and see how it can be described as an indexed description.

Example When describing sized trees as an indexed description, the tricky part is to describe the relation between the index of a node, and the indices of its subtrees. To do this, we use the `Σ constructor, using it to include a valid decomposition of the index suc n, given by a value of type Σ (ℕ × ℕ) λ ( m , k ) → m + k ≡ n . The second element is then a description depending on such a decomposition, including two recursive positions: one of size `m` and one of size `k`. Below is the description as a whole:

```
STreeD : Set → ℕ → IDesc ℕ
STreeD A zero    = `1
STreeD A (suc n) =
  `Σ (Σ (ℕ × ℕ) λ { ( m , k ) → m + k ≡ n })
    λ { ((m , k) , _) →
      (`var m `× `Σ A λ _ → `1) `× `var k }
```

We capture the notion of datatypes that can be described in the universe of indexed descriptions by again constructing a record that stores a description and a proof that said description is isomorphic to the type we are describing:

```
record Describe {I} (A : I → Set) : Set where
  field φ : I → IDesc I
  field iso : (i : I) → A i ≃ Fix φ i
```

This allows us to use Agda's instance arguments to define functionality generically over any type that we can describe as an indexed description.

5.1.2 Example: well-typed lambda terms

To demonstrate the expressiveness of the `IDesc` universe, and to show how one might model a more complex datatype in it, we consider the *simply typed lambda calculus* as an example. We model the simply typed lambda calculus in Agda according to the representation used in Philip Wadler and Wen Kokke's PLFA [39].

Modelling well-typed terms in Agda

Wadler and Kokke use a representation using De Bruijn indices [14], which represents variables as a natural number denoting the number of lambda abstractions between the variable and the binder it refers to. Using De Bruijn indices has the clear advantage that any two α-equivalent terms have the same representation. Listing 5.3 contains the datatype definitions for raw terms, types and contexts, used to represent untyped lambda terms. Types can be either a ground type `` `τ ``, or a function type `` σ `→ τ ``. Since we are using De Bruijn indices, we do not need to store variable names in the context, only types. Hence the `Ctx` type is essentially a list of types.

We write $\Gamma \ni \tau$ to signify that a variable with type τ is bound in context Γ. Context membership is described by the following inference rules:

Listing 5.3: Datatypes for raw terms, types and contexts

```
data RT : Set where
  tvar : ℕ → RT
  tlam : RT → RT
  tapp : RT → RT → RT

data Ty : Set where
  `τ   : Ty
  _`→_ : Ty → Ty → Ty

data Ctx : Set where
  ∅    : Ctx
  _,'_ : Ctx → Ty → Ctx
```

$$\texttt{[Top]}\,\frac{}{\Gamma, \tau \ni \alpha : \tau} \qquad \texttt{[Pop]}\,\frac{\Gamma \ni \tau}{\Gamma, \sigma \ni \alpha : \tau}$$

We describe these inference rules in Agda using an inductive datatype, shown in listing 5.4, which is indexed with a pair of type and context. Its inhabitants correspond to all proofs that a context Γ contains a variable of type τ.

Listing 5.4: Context membership in Agda

```
data _∋_ : Ctx → Ty → Set where

  [Pop] : ∀ {Γ τ}
        → Γ ,' τ ∋ τ

  [Top] : ∀ {Γ τ σ} → Γ ∋ τ
        → Γ ,' σ ∋ τ
```

We write $\Gamma \vdash t : \tau$ to express a typing judgement stating that term t has type τ when evaluated under context Γ. The following inference rules determine when a term is type correct:

$$\texttt{[Var]}\,\frac{\Gamma \ni \alpha : \tau}{\Gamma \vdash \alpha : \tau} \qquad \texttt{[Abs]}\,\frac{\Gamma, \alpha : \sigma \vdash t : \tau}{\Gamma \vdash \lambda\, \alpha \,.\, t : \sigma \to \tau} \qquad \texttt{[App]}\,\frac{\Gamma \vdash t1 : \sigma \to \tau \quad \Gamma \vdash t2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}$$

We model these inference rules in Agda using a binary relation between contexts and types whose inhabitants correspond to all terms that have a given type under a given context (listing 5.5)

Listing 5.5: Well-typed lambda terms as a binary relation

```
data _⊢_ : Ctx → Ty → Set where

  [Var] : ∀ {Γ τ} → Γ ∋ τ
        → Γ ⊢ τ

  [Abs] : ∀ {Γ σ τ} → Γ ,' σ ⊢ τ
        → Γ ⊢ (σ `→ τ)

  [App] : ∀ {Γ σ τ} → Γ ⊢ (σ `→ τ) → Γ ⊢ σ
        → Γ ⊢ τ
```

Given an inhabitant of type `Γ ⊢ τ`, we can write a function `toTerm` that transforms the typing judgement to its corresponding untyped term, simply by *erasing* the indices of the proof term.

```
toVar : ∀ {Γ τ} → Γ ∋ τ → ℕ
toVar [Pop]       = zero
toVar ([Top] Γ∋τ) = suc (toVar Γ∋τ)

toTerm : ∀ {Γ τ} → Γ ⊢ τ → RT
toTerm ([Var] Γ∋τ)   = tvar (toVar Γ∋τ)
toTerm ([Abs] t)     = tlam (toTerm t)
toTerm ([App] tₗ tᵣ) = tapp (toTerm tₗ) (toTerm tᵣ)
```

The untyped term returned by `toTerm` will has type `τ` under context `Γ`, even though Agda's type system does not guarantee this anymore.

An indexed description for well-typed terms

In section 5.1.1, we saw that we can describe the `Fin` both by induction on the index, as well as by adding explicit constraints. Similarly, we can choose to define a description for well-typed terms in two ways: either by induction on the type of the terms we are describing, or by including an explicit constraint that the index type is a function type for the description of the abstraction rule. In either case, we start by defining descriptions for each of the three possible constructors (listing 5.6).

Listing 5.6: Descriptions for the constructors of the simply typed lambda calculus

```
varDesc : Ctx × Ty → IDesc (Ctx × Ty)
varDesc (Γ , τ) = `Σ (Γ ∋ τ) λ _ → `1

absDesc : Ctx × Ty × Ty → IDesc (Ctx × Ty)
absDesc (Γ , σ , τ) = `Σ ℕ (λ α → `var (Γ ,' σ , τ))

appDesc : Ctx × Ty → IDesc (Ctx × Ty)
appDesc (Γ , τ) = `Σ Ty (λ σ →  `var (Γ , σ `→ τ)
                             `× `var (Γ , σ))
```

Given the descriptions for the individual constructors, we can assemble a description for the entire datatype by pattern matchin on the index type, and returning for each branch a coproduct of the descriptions of all constructors that could have been used to create a value with that particular index (listing 5.7).

Listing 5.7: Inductive description of the simply typed lambda calculus

```
wt : Ctx × Ty → IDesc (Ctx × Ty)
wt (Γ , τ) =
  case τ of λ { `τ →
      `σ 2 λ { •     → varDesc (Γ , τ)
             ; (▻ •) → appDesc (Γ , τ) }
    ; (τ₁ `→ τ₂) →
      `σ 3 λ { •       → varDesc (Γ , τ)
             ; (▻ •)   → absDesc (Γ , τ₁ , τ₂)
             ; (▻ ▻ •) → appDesc (Γ , τ) } }
```

Alternatively, we can describe the simply typed lambda calculus as a coproduct of the descriptions of all its constructors, and adding an explicit constraint in the case of the abstraction rule that requires a proof that the index type is a function type (listing 5.8).

To convince ourselves that these descriptions do indeed describe the same type, we can show that their fixpoints are isomorphic:

```
desc≃ : ∀ {Γ τ} → Fix Inductive.wt (Γ , τ)
                ≃ Fix Constrained.wt (Γ , τ)
```

Listing 5.8: Description of the simply typed lambda calculus with explicit constraints

```
wt : Ctx × Ty → IDesc (Ctx × Ty)
wt (Γ , τ) =
  `σ 3 λ { •      → varDesc (Γ , τ)
         ; (▻ •) →
           `Σ (Σ (Ty × Ty) λ { (σ , τ') → τ ≡ σ `→ τ' })
              λ { ((σ , τ') , _) → absDesc (Γ , (σ , τ')) }
         ; (▻ ▻ •) → appDesc (Γ , τ)
         }
```

Given an isomorphism between the fixpoints of two descriptions, we can prove that they are both isomorphic to the target type by establishing an isomorphism between the fixpoint of one of them and the type we are describing. For example, we might prove the following isomorphism:

```
wt≃ : ∀ {Γ τ} → Fix Constrained.wt (Γ , τ) → Γ ⊢ τ
```

Using the transitivity of isomorphisms, we can show that the inductive description also describes well typed terms.

Both variations are equally valid descriptions of the simply typed lambda calculus (they are isomorphic), but depending on the situation one might prefer one over the other. A downside to defining descriptions by induction over the index type is that we often end up with at least some code duplication, making them unnecessarily verbose in some cases. Descriptions with explicit constraints do not have this downside. We could even substitute `varDesc`, `absDesc` and `appDesc` for their respective definitions, since they are only referred to once. This often results in descriptions that are much more succinct, but arguably less straightforward.

When looking ahead to the derivation of generators from descriptions, we see that using a description with explicit constraints has the effect of delaying the point at which we find out that a certain constructor could not have been used to construct a value with a particular index. In the case of inductive descriptions, we find out this fact relatively early; since the set of available operations explicitly depends on the index, it will never include descriptions that could not have been used to begin with. Contrary, when using a description that explicitly includes constraints, we only find that a particular constructor could not have been used when we fail to synthesize the required equality proof. In the end this means that the choice of descriptions style comes down to a tradeof between conciseness and efficiency. Throughout the remainder of this thesis, we will stick with the inductive style of defining descriptions.

5.2 Deriving generators

The process of deriving a generator for indexed descriptions is mostly the same as for regular types. There are a few subtle differences, which we will outline in this section. We define a function IDesc-gen that derives a generator from an indexed description. Let us first look at its type signature:

```
IDesc-gen : ∀ {I} {i : I} → (δ : IDesc I) → (φ : I → IDesc I)
          → Gen (⟦ δ ⟧ (Fix φ)) (λ i → ⟦ φ i ⟧ (Fix φ)) i
```

We take a value of type IDesc I (the description we are inducting over) and a function I → IDesc I (describing the type for which we are deriving a generator) as input. We return an *indexed* generator, which produces values of the type dictated by the semantics of the input description. We build this generator by defining it for the various constructors of the IDesc type.

5.2.1 Unit, product and recursive positions

The definition for `var, `1 and `× can be readily transferred from the definition of deriveGen. Their definition is included below:

```
IDesc-gen (`var i)   φ = ⦇ In (μ i) ⦈
IDesc-gen `1         φ = ⦇ tt ⦈
IDesc-gen (δₗ `× δᵣ) φ = ⦇ (IDesc-gen δₗ φ) , (IDesc-gen δᵣ φ) ⦈
```

5.2.2 Generalized coproduct

The generic generators for the generalized coproduct are slightly more involved, since we have to return a generator that produces dependent pairs. This is tricky, because the applicative combinators are not expressive enough to capture the dependency between the generated *value* of the first element, and the *type* of the second element. This means that we have to utilize the monadic structure of the generator type in order to be able to capture this dependency.

```
IDesc-gen {i = i} (`σ n T) φ = do
  sl ← Call {x = i} n Sl-gen
  t ← IDesc-gen (T sl) φ
  pure (_,_ {B = λ sl → ⟦ T sl ⟧ (Fix φ)} sl t)
```

Here we assume that `Sl-gen : (n : ℕ) → Gen`$_i$ `(Sl n) Sl n` is in scope, producing values of the selector type. We capture the dependency between the generated first element of the pair, and the type of the second element using the monadic bind of the generator type, similar to when we were defining a generator for the universe of indexed containers. The definition is pretty straightforward, although we need to explicitly pass around some metavariables, which Agda would otherwise not be able to solve.

5.2.3 Dependent pairs

We can reuse this exact same structure when defining a generator for `` `Σ ``, however since the type of its first element is chosen by the user, we need the programmer to supply a suitable generator. We use the same approach using a metadata structure as we did previously to enable the programmer to pass generators as input to `IDesc-gen`. We define this metadata structure as a datatype `data IDescM I (P : Set → Set) : IDesc I → Set`. It is largely similar to the metadata structure used for regular types (section 3.2), so we refrain from including its entire definition here. The key difference is that we now require the programmer to store a piece of data depending on the type of the first element of a `` `Σ ``:

```
`Σ~ : ∀ {S : Set} {T : S → IDesc I} → P S
    → ((s : S) → IDescM P (T s)) → IDescM P (`Σ S T)
```

The constructor of the `IDescM` type associated with the generalized coproduct follows the same structure as `` `Σ~ ``, but without a value argument, and with `S` instantiated to the selector type.

If we now assume that `IDesc-gen` is parameterized over a metadata structure containing generators for the first argument of the `` `Σ `` constructor, we can define a generator for its interpretation:

```
IDesc-gen (`Σ S T) φ (`Σ~ S~ T~) = do
  s ← Call tt (λ { tt → S~ })
  t ← IDesc-gen (T s) φ (T~ s)
  pure (_,_ {B = λ s → ⟦ T s ⟧ (Fix φ ) } s t)
```

Given a definition for *deriveGen*, we can use an instance of the `Describe` record to define a generic generator for all types that are isomorphic to the fixpoint of some indexed description.

5.2.4 Example: deriving a generator for well-typed lambda terms

of Let us look at an example in which we use `deriveGen` to derive a generator in order to get a feel for how the generic mechanism defined in this section works out when we actually try to use it. We will use the inductive description of well-typed terms to derive a generator from.

Looking at the description, we see that we use the `` `Σ `` combinator to build dependent pairs that have either a proof that some `τ` is an element of a context `Γ`, or a type as their first element. This means that we require generators that produce elements of type `Γ ∋ τ` and `Ty`:

```
gen∋ : (Γ : Ctx) → (τ : Ty) → Gen (Γ ∋ τ) (λ { tt → Γ ∋ τ }) tt

genTy : Gen Ty (λ { tt → Ty }) tt
```

How we obtain these generators is entirely up to us. We can use any of the generic derivation mechanisms described throughout this thesis, or manually define them according to our needs. The latter has the advantage that it lets us guide the generation process somewhat. In the case of lambda terms, we need to choose a new type σ when using the application rule. It might be beneficial to, for example, write a generator that will first produce a list of types that can be found in the context, and only later will exhaustively enumerate the space of all types.

Given that the required generators are in scope, we define a metadata structure indexed by the inductive description of well-typed lambda terms, which shown in listing 5.9. The structure of this metadata structure is entirely dependent on how we defined the description in the first place. We only really have a choice for the first element of `` `Σ~ ``.

Listing 5.9: Metadata structure for the inductive description of well-type lambda terms

```
wtM : (i : Ctx × Ty)
    → IDescM (λ S → Gen S (λ { tt → S }) tt) (wt i)
wtM (Γ , `τ) =
  `σ~ λ { •       → `Σ~ (gen∋ Γ `τ) λ _ → `1~
        ; (▻ •) → `Σ~ genTy λ _ → `var~ `×~ `var~
        }
wtM (Γ , (τ₁ `→ τ₂)) =
  `σ~ λ { •         → `Σ~ (gen∋ Γ (τ₁ `→ τ₂)) (λ s → `1~)
        ; (▻ •)     → `var~
        ; (▻ ▻ •) → `Σ~ genTy (λ s → `var~ `×~ `var~) }
```

The metadata structure for well-typed terms quite neatly demonstrates how our approach separates the parts of generation that can be done mechanically from the parts for which a little help from the programmer is required. Furthermore, after creating this separation we leave the programmer with complete freedom over how they provide the necessary generator, giving them the option to either reuse any of the generic derivation mechanisms we described or to define the required generators directly.

5.3 Proving completeness

We aim to prove the same completeness property for generators derived from indexed descriptions as we did for generators derived from regular types. Since both universes and the functions that we use to derive generators from their inhabitants share quite a few structural similarities, so do their respective completeness proofs. This means that we can recycle a considerable portion of the completeness proof that we wrote for regular types.

Let us first look at the exact property we aim to prove. Since we deal with indexed generators, the desired completeness property changes slightly. In natural language, we might say that our goal is to prove that *for every index* `i` *and value* `x` *of type* `P i`*, there is a depth such that* `x` *occurs in the enumeration we derive from the code describing* `P`. In Agda we formalize this property as follows:

```
Complete : ∀ {I} {P : I → Set} → (i : I)
           → ((i : I) → Gen (P i) P i) → Set
Complete {I} {P} i gen =
  ∀ {x : P i} → ∃[ n ] (x ∈ enumerate gen i (gen i) n)
```

Which is essentially the same property we used for regular types, adapted for usage with indexed types. To be able to inductively define the completeness proof, we use a slight variation on this property that distinguishes between the generator we are inducting over, and the generator describing recursive positions:

```
_|ᵢ_⇝_ : ∀ {I} {A : Set} {P : I → Set} {i : I}
         → Gen A P i → ((i : I) → Gen (P i) P i) → A → Set
```

In general, the second property is equivalent to the first if the two supplied generators are the same. We then define the following completeness lemma for the generators derived from indexed descriptions:

```
IDesc-gen-Complete :
  ∀ {δ φ x} → IDesc-gen δ φ |ᵢ (λ i → IDesc-gen (φ i) φ) ⇝ x
```

We will show how to define a proof for this lemma by considering the various constructors of the `IDesc` type.

5.3.1 Unit, product and recursive positions

The completeness proofs for `` `var``, `` `1`` and `` `×`` can again be transplanted almost without changes from the proof for regular types:

```
IDesc-gen-Complete {`var i} {φ} {In x}
  with IDesc-gen-Complete {φ i} {φ} {x}
... | prf = In-Complete prf
IDesc-gen-Complete {`1} {φ} {tt} = 1 , here
IDesc-gen-Complete {δ₁ `× δ₂} {φ} {x , y} =
  `×-Complete (IDesc-gen-Complete {δ₁})
              (IDesc-gen-Complete {δ₂})
```

Where we assume that a proofs of the following lemmas is in scope:

```
In-Complete : ∀ {g g' x}
  → g ∣ᵢ g' ↝ x → (⦇ In g ⦈) ∣ᵢ g' ↝ In x

`×-Complete : ∀ {g₁ g₂ g' x y}
  → g₁ ∣ᵢ g' ↝ x → g₂ ∣ᵢ g' ↝ y
  → ⦇ g₁ , g₂ ⦈ ∣ᵢ g' ↝ (x , y)
```

We will not go into how we can prove these lemmas, as we already discussed this when describing the completeness proof for regular types.

5.3.2 Generalized coproducts and dependent pairs

Listing 5.10: Completeness for the bind operator

```
>>=-Complete :
  ∀ {I A} {P : A → Set} {T : I → Set} {x y}
    {g : Gen A T x} {g' : (v : A) → Gen (P v) T y}
    {x : Σ A P} {tg : (i : I) → Gen (T i) T i}
  → g ∣ᵢ tg ↝ proj₁ x
  → g' (proj₁ x) ∣ᵢ tg ↝ proj₂ x
  → (g >>= λ y → ⦇ (λ v → y , v) (g' y) ⦈) ∣ᵢ tg ↝ x
```

Since the generators for `` `Σ`` and `` `σ`` are assembled using *monadic bind*, we need to prove that this operation is completeness preserving. Defining what completeness in general means for `>>=` is very difficult, but since both usages in `IDesc-gen` follow the same structure, we can get away with proving a completeness property over our specific use of the bind operator. The lemma we use is shown in listing 5.10.

With this lemma, and an appropriate metadata parameter supplied to our completeness proof, we can fill in the cases for the generalized coproduct and dependent pairs, assuming that a completeness proof for the generator producing values of the selector type is in scope.

5.3.3 Wrapping up the proof

It is worth noting that, since the universe of indexed descriptions exposes a product combinator, we require a proof of *monotonicity* for generators derived using `IDesc-gen` as well. We will not go into how to assemble this proof here (since its structure is essentially the same as the monotonicity proof for regular types), but it is obviously not possible to assemble this proof without proving the monotonicity property over our bind operation first.

Conclusion

In this chapter, we have shown how we can extend the generic derivation mechanism we used for regular types and indexed containers can be extended to a more expressive universe that is able to represent arbitrary indexed families. Furthermore, we have proved that the generators we derive from codes in this universe satisfy our completeness property. We demonstrated that this generic approach is powerful enough to generate well-typed lambda terms, relying on the programmer to supply guidance for those parts of the datatype that are too difficult to handle generically.

Given the fact that we have developed a mechanism to find inhabitants of arbitrary indexed families, we may view this result though the Curry-Howard correspondence, which implies that we have simultaneously obtained a mechanism is able to synthesize proofs of propositions in arbitrary formal systems. If we place our work in the context of proof search, we find that we can alternatively view our work as an implementation of *backward-chaining proof search* [28] parameterized over a formal system and whose proofs are correct by construction, enforced by Agda's type system. In this context, our completeness property implies that, if we enumerate these proofs, we will eventually find *all* proofs of a property, given enough time and memory. Consequently, if a theorem has a proof in a given formal system, we are guaranteed to find one in finite time. It is important to recognize that, although our approach offers a great deal of flexibility, it is not very efficient. Eventually, we exhaustively enumerate all possible proof chains that could have led to our goal type, meaning that the practical applications of this work are most likely limited until we further optimizations have been implemented.

Part II

Implementation

6 Generators for Indexed Descriptions in Haskell

We implement part of the ideas described in this thesis in Haskell to show their practical applicability. More specifically, we port the universe of indexed descriptions as described in section 5.1 together with the accompanying generic generator to Haskell. We show that it is possible using this approach to generate constrained test data by describing constrained data as an inductive datatype, and generating inhabitants of that datatype. We again consider well-typed lambda terms as an example.

6.1 General approach

The general structure of our approach is not much different from how we derived generators for indexed descriptions in Agda, and consists of the following steps:

1. First we define an abstract generator type, together with a mapping to enumerators (i.e. functions with type $Int \rightarrow [\,a\,]$).
2. Next, we define a datatype for indexed descriptions, $IDesc$, together with its semantics
3. Then we write a function that derives a generator from a value of type $IDesc$, producing elements of a type dictated by the semantics of th input description.
4. Finally, we convert the generated values to some user defined, ”raw”, datatype.

Dagand originally defines the universe in a dependently typed setting [12], and similarly we make extensive use of both dependent pairs and dependent function types in our Agda development. Haskell’s type is unfortunately not expressive enough to facilitate this approach. Instead, we will use a lot of *singleton types* [18] to work around this limitation. Singleton types are a technique to simulate a restricted for of dependent types in a non dependently typed language. They are intended to work together with the *DataKinds* extension [1]. A singleton type is indexed by some promoted datatype, and has exactly one inhabitant for every inhabitant of the type it is indexed with.

6.2 Representing indexed descriptions in Haskell

We take the datatype described in section 5.1 as our basis, adding an extra type parameter $a :: *$, describing the raw (non-indexed) type we will be converting to. Listing 6.1 contains the definition of this type, *IDesc a i*, with constructors for *empty types*, *unit types*, *recursive positions* and *product types* taken almost directly from the corresponding Agda type.

Listing 6.1: Definition of *IDesc a i* in Haskell

```
data IDesc (a :: ∗) (i :: ∗) where
  One     :: IDesc a i
  Empty   :: IDesc a i
  Var     :: i → IDesc a i
  ( :*: ) :: IDesc a i → IDesc a i → IDesc a i
  ( :+> ) :: Sing n → Vec (IDesc a i) n → IDesc a i
  Σ :: Proxy s → IDesc a (s → i) → IDesc a i
```

We use variations of the constructors for the generalized coproduct and dependent pairs, mainly because their semantics is a dependent pair, making it difficult to transfer their definitions directly to Haskell.

6.2.1 Generalized coproducts

In the original definition, the generalized coproduct was defined by a natural number n and a function that takes a value from the finite set of size n (Fin n) to a description number n. We choose to use a *vector* (or size indexed list) in favor of a function, since this wil make it considerably easier to define the semantics for this constructor. Assuming *Nat* = *Zero* | *Suc Nat* is in scope, we use the following GADT to describe a vector:

```
data Vec (a :: ∗) (n :: Nat) where
  VNil :: Vec a Zero
  (:::)  :: a → Vec a n → Vec a (Suc n)
```

We require a singleton type for natural numbers to relate the first parameter of (:+>) with the length of its second parameter.

6.2.2 The `Σ constructor

We choose to use a more restrictive form of the `Σ constructor, in order to be able to encode its semantics in Haskell's type system. Instead of having the second argument to `Σ be a function that maps values of the type s stored in its first argument to descriptions, we use a single description with with type $IDesc\ a\ (s \to i)$. This has the effect of pushing the dependency inwards: any recursive postion in this description will now store a function of type $s \to i$, meaning that the indices of recursive positions can still depend on a the chosen value of type s, while the structure of the stored description will be constant. Although theoretically this means that we can describe fewer types, we have yet to encounter an example of a type that cannot be described with this more restrictive universe definition.

The motivation behind this alteration is that it eliminates the dependency between the chosen value of type s, and the semantics of the dependent pair as a whole. This works because (as we will see in the next section, when we make the semantics precise) we use *shallow recursion*, unfolding the isomorphism between a and the semantics of the description that describes a one layer at a time. For this reason, the semantics of a recursive position is the type a, meaning that the semantics of a description is completely independent of its index type. By pushing the dependency on the chosen values of type s inwards to the recursive indices stored, we have made this value irrelevant to the semantics of the description as a whole.

It is important to note that we can map values of type $IDesc\ a\ (s \to i)$ to functions with type $s \to IDesc\ a\ i$, such that the interpretation of the resulting description is equal for all possible arguments of type s. We will make this mapping precise when we set out to derive generators from descriptions.

6.2.3 Semantics

We define the semantics of the $IDesc$ universe as a type family, mapping promoted values to their semantics. The interpretation of descriptions is relatively straightforward, and largely the same as for regular types. The semantics are shown in listing 6.2. E is a type with no constructors, representing the empty type.

We use the following type synonym to unbox the type that is stored in a proxy:

```
type UnProxy (p :: Proxy a) = a
```

In case of the generalized coproduct, we map a vector of descriptions to a type representing a choice between the interpretation of any of the descriptions carried in that vector. For example, we would map a vector $d1 ::: d2 ::: VNil$ to the type $Either\ (Interpret\ d1)\ (Interpret\ d2)$. We build the appropriate type by induction over the length of the vector. We have two base cases, one for empty vectors and one for vectors containing one element. We do so to reduce the complexity of the resulting type, preventing a vector with one element, $d ::: VNil$, to be mapped to a coproduct of its semantics and the empty type. A secondary reason is to prevent derived generators from including to many unneeded empty generators.

Listing 6.2: Semantics of the *IDesc* type

```
type family Interpret (d :: IDesc a i) :: *
type instance Interpret One                          = ()
type instance Interpret Empty                        = E
type instance Interpret (Var _ :: IDesc a i)         = a
type instance Interpret (dl :*: dr)                  =
  (Interpret dl, Interpret dr)
type instance Interpret (SZero :+> VNil)             = E
type instance Interpret (SSuc SZero :+> (x ::: VNil)) = Interpret x
type instance Interpret (SSuc (SSuc n) :+> (x ::: xs)) =
  Either (Interpret x) (Interpret (SSuc n :+> xs))
type instance Interpret (Σ p fd)                     =
  (UnProxy p, Interpret fd)
```

6.3 Deriving generators

Before we set out to describe how we derive generators from descriptions, we first briefly outline the generator type used, and describe the singleton type for descriptions needed to capture the dependency between the input description and the type of values produced by the returned generator.

6.3.1 The generator type

We take the definition of the generator type, shown in listing 6.3, straight from our Agda development. Again, we choose to not have separate generator types for indexed and non-indexed generators, representing non-indexed types as types indexed by the unit tuple, ().

We make the *Gen* type an instance of *Functor*, *Applicative*, *Monad* and *Alternative* (by relating the functions they exposed with the associated constructors), allowing us to define generators using the familiar applicative programming style:

```
tree :: Gen () Tree Tree
tree = pure Leaf <|> Node <$> μ () <*> μ ()
```

6.3.2 A singleton type for descriptions

Since our goal is eventually to define a function *idesc_gen* :: *Sing d* → *Gen i a* (*Interpret d*), we require an appropriate singleton type for the *IDesc* type. We again start by defining this instance for *One*, *Empty*, *Var* and :*: , shown in listing 6.4

Listing 6.3: The *Gen* type in Haskell

```
data Gen i t a where
  None :: Gen i a t
  Pure :: a                                → Gen i a t
  Or   :: Gen i a t → Gen i a t           → Gen i a t
  Ap   :: Gen i (b → a) t → Gen i b t     → Gen i a t
  Bind :: Gen i a t → (a → Gen i b t)     → Gen i b t
  μ    :: i                                → Gen i a a
  Call :: (j → Gen j a a) → j             → Gen i a t
```

Listing 6.4: Singleton instance for the *IDesc* type

```
SOne    ::                                   SingIDesc One
SEmpty  ::                                   SingIDesc Empty
SVar    :: forall (i' :: i) . i →            SingIDesc (Var i')
( :*: z) :: SingIDesc l → SingIDesc r →      SingIDesc (l :*: r)
```

In order to be able to define a singleton instance for the generalized coproduct, we require a singleton instance of *SNat*. We assume this instance, denoted *SNat2*, is in scope:

```
( :+>$ ) :: SNat2 n → SVec xs → SingIDesc (n :+> xs)
```

The singleton definition for the Σ constructor (listing 6.5) has a few subtleties. First, the type stored in its first element is required to be a member of the *Promote* typeclass. This typeclass describes types which are an instance of *Singleton*, and for which we know how to promote a value of type a to a value of type *Sing* a. The *Promote* class has one associated function *promote* :: a → *Promoted* a *Sing*, where *Promoted* is defined as follows:

```
data Promoted (a :: ∗) (f :: a → ∗) = forall (x :: a) . Promoted (f x)
```

Storing singleton values, but hiding the their index.

This results in the complete definition of *SingIDesc* shown in listing 6.6.

The singleton instance for Σ also stores an explicit generator for values of type s. We could have used a typeclass here, but as we will see, when considering some examples, that it is often more convenient to explicitly supply the generator to be used. This essentially has the effect that *SingIDesc* will simultaneously function to describe the dependency between

Listing 6.5: Singleton instance for the Σ constructor

```
SSigma :: Promote s ⇒ SingIDesc d
        → Gen () s s
        → (forall s′ . Sing s′ → Interpret d :~: Interpret (Expand d s′))
        → SingIDesc (Σ (Proxy :: Proxy s) d)
```

Listing 6.6: Singleton indexed description

```
data SingIDesc (d :: IDesc a i) where
  SOne :: SingIDesc One
  SEmpty :: SingIDesc Empty
  SVar :: forall (i′ :: i) . i → SingIDesc (Var i′)
  ( :*:$ ) :: SingIDesc l → SingIDesc r → SingIDesc (l :*: r)
  ( :+>$ ) :: SNat2 n → SVec xs → SingIDesc (n :+> xs)
  SSigma :: Promote s ⇒ SingIDesc d
    → Gen s
    → (forall s′ . Sing s′ → Interpret d :~: Interpret (Expand d s′))
    → SingIDesc (Σ (Proxy :: Proxy s) d)
```

the input description, and the type of values produced by the output generator, and as a metadata structure carrying generators for dependent pairs.

Finally, we require a proof that the interpretation of the *expansion* of the second argument is equal to the interpretation of the second argument, for all values of type s. We require this proof in order unify the index types of the generator derived for a Σ and the generator derived from its second argument. We define the expansion operation a the type level using a mutually recursive type family, shown in listing 6.7.

Similarly, we use two mutually recursive functions to describe expansion for singleton descriptions (listing 6.8)

6.3.3 Constructing the generator

We now have all necessary ingredients in place to define a function $idesc\_gen$ that returns a generator based on its input description. It has the following type signature:

```
idesc_gen :: forall (d :: IDesc a i) . (Singleton i)
  ⇒ SingIDesc d → Gen i a a (Interpret d)
```

Listing 6.7: Description expansion

```
type family VExpand (sn :: SNat n)
  (xs :: Vec (IDesc a (s → i)) n) (x :: s) :: Vec (IDesc a i) n
type instance VExpand SZero VNil s           = VNil
type instance VExpand (SSuc sn) (x ::: xs) s =    Expand x s
                                               ::: VExpand sn xs s


type family Expand (d :: IDesc a (s → i)) (x :: s) :: IDesc a iconven
type instance Expand One          s = One
type instance Expand Empty        s = Empty
type instance Expand (Var i)      s = Var (i s)
type instance Expand (dl :*: dr)  s = (Expand dl s) :*: (Expand dr s)
type instance Expand (sn :+> xs) s = sn :+> VExpand sn xs s
type instance Expand (Σ p d)      s = Σ p (Expand d s)
```

The definitions for the unit type, empty type, recursive positions and product type follow naturally:

```
idesc_gen SOne         = pure ()
idesc_gen SEmpty       = empty
idesc_gen (SVar v)     = μ v
idesc_gen (dl :*:$ dr) = (,) <$> idesc_gen dl <*> idesc_gen dr
```

We define a generator for the generalized coproduct by (again) inducting over the vector length, returning a choice between the generator derived from the head of the vector and the generator derived from the tail of the vector.

```
idesc_gen (SZero2 :+>$ SVNil)                = empty
idesc_gen (SSuc2 SZero2 :+>$ (d :::$ SVNil)) = idesc_gen d
idesc_gen (SSuc2 (SSuc2 n) :+>$ (d :::$ ds))  =
  Left <$> idesc_gen d
  < | > Right <$> idesc_gen (SSuc2 n :+>$ ds)
```

If we now turn our attention to the generator derived from the Σ combinator, it becomes clear why we need to define the expansion operator and the proof of equality between the interpretation of a description and the interpretation of its expansion.

```
idesc_gen (SSigma desc gen eq) = do
  x ← Call (λ() → gen) ()
```

Listing 6.8: Description expansion for singletons

```
vexpand :: (Singleton s) ⇒ Sing sn → Sing xs → Sing s′
                          → SVec (VExpand sn xs s′)
vexpand SZero2 SVNil          s = SVNil
vexpand (SSuc2 sn) (x :::$ xs) s = expand x s :::$ vexpand sn xs s


expand :: (Singleton s) ⇒ Sing d → Sing s′ → Sing (Expand d s′)
expand SOne           sv = SOne
expand SEmpty         sv = SEmpty
expand (SVar ix)      sv = SVar (ix sv)
expand (dl :*:$ dr)   sv = expand dl sv :*:$ expand dr sv
expand (sn :+>$ xs)   sv = sn :+>$ vexpand sn xs sv
```

```
let px = promote x
case px of
  Promoted x′ → do
    p ← idesc_gen (expand desc x′)
    pure (x, eqConv (eq x′) p)
```

First, we obtain a suitable value for the first element by calling the supplied generator. Next, we promote this value x to get a singleton value x' of type $\mathit{Sing}\ x$. We apply the promoted value x' to the expansion of the second argument of Σ, which returns a generator producing values which have the type $\mathit{Interpret}\ (\mathit{Expand}\ \mathit{desc}\ s)$. We use this generator to get a value p of this type, which we can cast to a value of type $\mathit{Interpret}\ \mathit{desc}$ using the stored equality proof.

With the definition of $\mathit{idesc\_gen}$ complete, we can define a function $\mathit{genDesc} :: \mathit{forall}\ (d :: \mathit{IDesc}\ a\ i) \to \mathit{Sing}\ d \to \mathit{Gen}\ i\ a\ a$ that produces elements of the raw type represented by a description. Note that we need a conversion function $\mathit{to} :: \mathit{Interpret}\ d \to a$ to convert the values produced by $\mathit{idesc\_gen}\ d$.

6.4 Examples

We consider two examples to see how we can use the approach described in this section to generate constrained test data. First we consider the type of finite sets (e.g. Fin), and after that the recurring example of well-typed lambda terms. In order to be able to test the derived generators, we assume that a function $\mathit{run} :: (i \to \mathit{Gen}\ i\ a\ a) \to i \to \mathit{Int} \to [a]$

is in scope, interpreting abstract generators as an exhaustive enumeration up to a certain depth.

6.4.1 Finite sets

We assume the canonical definition of *Fin*, shown in listing 6.9. If we erase the index of a value of type *Fin n*, we end up with a value of type *Nat*, hence *Nat* is the raw type of our description. The goal is then to derive a generator producing values of type *Nat*, which we interpret as values of type *Fin n*, but with their indices erased. This means that if we choose n as our index, the generator can only produce values that are *less than* the chosen index n. For example, index *Suc* (*Suc Zero*) should only produce the values *Suc Zero* or *Zero*, and using index *Zero* should result in a generator producing no values at all.

Listing 6.9: Finite sets in Haskell

```
data Fin (n :: Nat) :: *where
  FZero :: Fin (Suc n)
  FSuc  :: Fin n → Fin (Suc n)
```

We start by defining a type family that maps indices to descriptions:

```
type family FinDesc (n :: Nat) :: IDesc Nat Nat
type instance FinDesc Zero     = Empty
type instance FinDesc (Suc n) = (SSuc (SSuc SZero))
                                  :+> (One ::: Var n ::: VNil)
```

If the index is Zero, we return an empty description. Otherwise we have a choice between two constructors: *Suc* and *Zero*. Next, we need to a singleton value of this description:

```
finSDesc :: Sing n → SingIDesc (FinDesc n)
finSDesc SZero = SEmpty
finSDesc (SSuc n) =
  SSuc2 (SSuc2 SZero2) :+>$ (SOne :::$ SVar n :::$ SVNil)
```

In this case, the definition of *finSDesc* is completely dictated by our definition of *FinDesc*. Finally, we define a conversion function, mapping interpretations to values:

```
toFin :: Sing n → Interpret (FinDesc n) → Nat
toFin (SSuc sn) (Left ()) = Zero
toFin (SSuc sn) (Right n) = Suc n
```

We are now ready to generate values using the description for *Fin*. We do this simply by promoting the provided index, and calling *genDesc*.

```
genFin :: forall (n :: Nat) . Nat → Gen Nat Nat Nat
genFin n =
  case promote n of
    (Promoted sn) → genDesc sn
```

If we now run *genFin*, we see that it indeed produces the expected output:

```
> run genFin Zero 10
[]
> run genFin (Suc (Suc (Suc Zero))) 10
[Zero,Suc Zero,Suc (Suc Zero)]
```

6.4.2 Well-typed lambda terms

The process for generating well-typed lambda terms is exactly the same as for finite sets albeit slightly more involved due to the complexity of the datatype describing well-formedness involved. We use the description shown in listing 5.7 as a basis, modelling types, terms and contexts with the following datatypes:

```
data Type = Type :-> Type | T
data Term = TVar Nat | TAbs Term | TApp Term Term
type Ctx = [Type]
```

We use the datatype *CtxPos* to describe a position in a context:

```
data CtxPos = Here | There CtxPos
```

Next, we define a generator for context positions:

```
genElem :: Ctx → Type → Gen () CtxPos CtxPos
genElem [] _      = empty
genElem (t:ts) t′ = (if t ≡ t′ then pure Here else empty)
                    <|> (There <$> genElem ts t′)
```

Here, *genElem* takes a context and a type, and returns all positions at which that type occurs in the context. With all the necessary prerequisites in place to generate well-typed terms, we start by defining a type family that captures an appropriate description, show in listing 6.10. This is a direct translation of the description shown in chapter 7 5.7. Since

Listing 6.10: Type level description of well typed terms

```
type VarDesc = Σ (Proxy :: Proxy CtxPos) One
type AppDesc = Σ (Proxy :: Proxy Type) (Var I :*: Var I)
type family SLTCDesc (i :: (Ctx, Type)) :: IDesc Term (Ctx, Type)
type instance SLTCDesc ((,) Γ T) =
  SSuc (SSuc SZero) :+> (VarDesc ::: AppDesc ::: VNil)
type instance SLTCDesc ((,) Γ (t1 :-> t2)) =
  SSuc (SSuc (SSuc SZero)) :+>
    (VarDesc ::: Var ((,) (t1:Γ) t2) ::: AppDesc ::: VNil)
```

we never need recursive indices at the type level, we use a type family $I\ (s :: *) :: i$ as a placeholder for the recursive positions inside a Σ.

Next we define a singleton value that inhabits this description (listing 6.11). Its structure is again dictated completely by the type family *SLTCDesc*. It now becomes clear why we chose to have the programmer explicitly supply a generator to a Σ, since we can conveniently apply the index context and type to *genElem* to obtain a generator that produces the required context positions.

Listing 6.11: Singleton description of well typed terms

```
sltcDesc :: Sing i → Sing (SLTCDesc i)
sltcDesc (SPair Γ ST) = (SSuc2 (SSuc2 SZero2)) :+>$
    (  SSigma SOne (genElem Γ T) (\_ → Refl)
   :::$ SSigma (SVar (λσ → (Γ, σ :-> T)) :*:$ SVar (Γ, ))
             genType (\_ → Refl)
   :::$ SVNil)
sltcDesc (SPair Γ (t1 :->$ t2)) = (SSuc2 (SSuc2 (SSuc2 SZero2))) :+>$
    (  SSigma SOne (genElem (Γ) (t1 :-> t2)) (\_ → Refl)
   :::$ SVar (t1:Γ, t2)
   :::$ SSigma (SVar (λσ → (Γ, σ :-> (t1 :-> t2))) :*:$ SVar (Γ, ))
        genType (\_ → Refl)
   :::$ SVNil)
```

We now only have to define a convertion function that takes generated values and produces raw terms:

```
toTerm :: Sing i → Interpret (SLTCDesc i) → Term
toTerm (SPair _ ST) (Left (n, ())) = TVar (toNat n)
toTerm (SPair _ ST) (Right (_, (t1, t2))) = TApp t1 t2
toTerm (SPair _ (_ :->$ _)) (Left (n, ())) = TVar (toNat n)
toTerm (SPair _ (_ :->$ _)) (Right (Left y)) = TAbs y
toTerm (SPair _ (_ :->$ _)) (Right (Right (_, (t1, t2)))) = TApp t1 t2
```

Giving us everything we need to start generating well-typed terms. We do this again by promoting the supplied index, and calling *genDesc* with this value:

```
termGen :: (Ctx, Type) → Gen (Ctx, Type) Term Term
  termGen i =
    case promote i of
      (Promoted i′) → genDesc i′
```

We can now use *run termGen* to produce well-typed given a context and a goal type:

```
 > run termGen ([T , T :-> T] , T) 3
[TVar Zero,TApp (TVar ... ... ... (TVar (Suc (Suc (Suc Zero))))))]
```

To assert that the produced values are indeed type correct, we define a function *check* :: *Ctx* → *Type* → *Term* → *Bool* that checks whether a raw term has a certain type under certain context.

```
 > all (check [T , T :-> T] T) $ run termGen ([T , T :-> T] , T) 3
True
```

Conclusion

We have given an implementation of the generic generator for indexed descriptions in Haskell, whose produced elements satisfy the constraint described by the input description by construction. Perhaps surprisingly so, we have been able to enforce the same static guarantees in the Haskell development as in our Agda development, without sacrificing too much in terms of expressiveness. Of course, the question what exactly we lose by restricting the `Σ constructor remains a lingering question, which is definitely not easy to answer; writing a description in Agda that we cannot convert to the Haskell development is easy, but who is to say that no other description exists describing the same type, which we *can* write with the Haskell datatype.

It is worth noting that we have been as true as possible to the Agda development, forfeiting possible change in design. The primary reason for this is that it allows us to use the completeness proof we have written in Agda as an argument for the correctness of our Haskell development with greater confidence.

With this implementation, we have provided a framework in Haskell with which it is indeed possible to generate constrained test data, so long we are able to find a suitable indexed family that describes the relevant constraints. We considered the recurring example of well-typed lambda terms, and have shown how well-formed raw terms can be generated.

7 Discussion

In this final chapter, we will discuss the work presented in this thesis and place it in the context of existing literature. Will also consider some possible next steps, and reflect on the results and some of the design choices we have made.

7.1 Conclusion

We have explored various approaches to the generation of test data using datatype generic programming, with the ultimate goal being to be able to generate test data that is subject to arbitrary constraints. Based on the observation that constrained test data can often be described as an indexed family, we approached this problem by looking at how to generate values of indexed families. We have looked at three distinct type universes, starting with the universe of Regular types, which is able to describe a set of algebraic datatypes roughly equal to the algebraic datatypes in Haskell 98 [22]. We described this universe in Agda, and showed how a generator can be derived from a code in this universe. Although the exact generator type is kept abstract in this derivation, we have described an example instantiation where generators are functions of type `Int → List a`, similar to SmallCheck's *Series* [34] typeclass. For this particular generator type, we have proved that the generator derived from a code is *complete*. That is, every value of the type described by the input code will eventually show up in the enumeration.

We then looked at two more expressive type universes, which are able to describe (some) indexed datatypes: *Indexed Containers* [5] and *Indexed Descriptions* [12]. For both universes, we described how a generic generator may be constructed from codes in these universes. For the universe of indexed descriptions, we also proved that the enumerative instantiation of the generator type satisfied our completeness property. For indexed containers, we were unfortunately not able to complete this proof. Attempts to construct a proof using the same structure as used to construct the completeness proofs for regular types and indexed descriptions failed, as this approach would require induction over function types.

Having constructed a mechanism that allows generation for arbitrary indexed families in Agda, we implemented the generic generator for indexed descriptions in Haskell. Although in order to enforce correctness of the generated data in Haskell's type system we needed to impose some restrictions on the descriptions that could be used, we were still able to describe all the example datatypes. We used this implementation to generate some example constrained test data, including well-typed lambda terms. The final result is a Haskell library that is able to generate constrained test data, given that the user provides a description of an indexed family that describes the desired test data.

7.2 Reflection

While the framework we have developed is very flexible and expressive in terms of what kind of test data we can generate, it is important to recognize that it is still very much experimental, and that much improvement is needed before it can be used in any practical setting. Here, we shortly outline some of the problems that remain with our current development, and critically reflect on some of the design choices made along the way.

7.2.1 Remaining problems

First, finding a description that accurately describes an indexed family is not at all trivial. Often, there exist multiple descriptions that all describe the same datatype. These different descriptions are all mapped to distinct generators, which may not necessarily exhibit the same behavior in terms of computational efficiency, or the order in which elements are generated. The fact that Haskell's type system in no way enforces the semantics of the input description to be actually isomorphic to the datatype it describes leaves room for mistakes when defining descriptions and the conversion between their semantics and the desired datatype. Furthermore, it is hard to say how well this approach scales when we require more complex test data, especially since this would require the programmer to come up with increasingly complex descriptions. Although our Agda formalization allows us to be reasonably confident that the generators we derive indeed produce values of the intended datatype, we have no knowledge about how efficient they are at doing this, and to what extent a generator's efficiency depends on the structure of the derived generator.

7.2.2 Design choices

While we did in the end reach our goal of developing a method for generation of arbitrary indexed families, there are some aspects of our development for which, in hindsight, a different approach or design might have been beneficial.

Instantiation of the abstract generator type

The primary reason we settled for the particular enumerative interpretation of generators that we use in our model is that it is relatively convenient to work with and easy to reason about. While these are important properties when assembling a theoretical model, there are some practical downsides. Most notably, the size parameter we choose is very crude, giving the programmer little control over the amount of test values generated.

We have looked at *Colists* in combination with *sized types* [4] to use as a result type for our generators. Colists allow (possible) infinite lists to be defined in Agda (normally the termination checker does not allow this) by making the operational semantics explicit. *Sized types* further expand the space of functions we can define by including information about the size of values in the type. The nice thing about having colists as our result type is that they give the user a lot of control over the amount of elements generated. Furthermore, infinite lists are idiomatic in Haskell, making it likely that generators producing colists carry over well once we implement our development in Haskell.

Unfortunately, we were unable to make this approach work, specifically because we had trouble convincing Agda's termination checker that our notion of cartesian product between two colists was terminating, even with the usage of sized types.

Usage of existing libraries

In many places throughout our Agda model, we have performed unnecessary work by defining functionality that is already defined in Agda's standard library. Although this is perhaps unsurprising given the author's unfamiliarity with Agda previous to this work, it still harms the composability and reusability of our development. A similar concern exists with the Haskell library, which relies on our own notion of singleton types, instead of using the existing package.

Completeness property

While our completeness property guarantees that all values of a type will eventually be generated, it still leaves a lot of room for erroneous behavior of generators. Mainly, it does not guarantee anything about the order in which generators produce elements, and allows enumerations to contain the same value more than once. While the completeness property we have chosen works fine if we regard it just as a sanity check that the generators we derive do not contain any fatal mistakes, it does not guarantee that these generators are actually usable in practice.

7.3 Related Work

In this section, we briefly discuss some of the academic context surrounding our work.

7.3.1 Libraries for property based testing

Different libraries may take different approaches towards the generation of test data. Studying these libraries provides us with valuable insight into the established methods for test data generation, and how these methods relate to our work.

QuickCheck

Published in 2000 by Claessen & Hughes [10], QuickCheck implements property based testing for Haskell. 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`. Although instances for most common Haskell types are supplied by the library, QuickCheck refrains from employing any form of generic programming, instead choosing to provide a comprehensive set of combinators with which a programmer can assemble generators.

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.

SmallCheck

Contrary to QuickCheck, SmallCheck [34] 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 *recursive 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 behind this approach is the *small scope hypothesis*, which states that if a program is incorrect, then it will almost allways fail on some small input value [6].

In addition to SmallCheck, there is also *Lazy* SmallCheck, which evaluates properties on partial test values, making it possible to test entire classes of test data at once.

LeanCheck

Where SmallCheck uses a value's *depth* to bound the number of test values, LeanCheck uses a value's *size* [27], where size is defined as the number of construction applications of positive arity. This gives a more fine-grained notion of size, which helps to battle the rapid growth of the space of test values.

Feat

A downside to both SmallCheck and LeanCheck is that they they are very inefficient when it comes to the generation of large test values; they both require all that are smaller (in terms of their respective definition of size) to be enumerated first. QuickCheck has no problem with either, but generators are often more tedious to write compared to their SmallCheck counterpart. Feat [17] aims to fill this gap by providing a way to efficiently enumerate algebraic types, employing memoization techniques to efficiently index these enumerations.

Hedgehog

Hedgehog [37] is a framework similar to QuickCheck, that employs random sampling to find test values. Hedgehog has no facilities for the automatic derivation of generators, and is even more rigorous in its approach than QuickCheck, only exposing a minimal set of combinators with which users can assemble their own generator.

QuickChick

QuickChick is a QuickCheck clone for the proof assistant Coq [16]. The fact that Coq is a proof assistant enables the user to reason about the testing framework itself [33]. This allows one, for example, to prove that generators adhere to some distribution.

QuickSpec

A surprising application of property based testing is the automatic generation of program specifications, proposed by Claessen et al. [11] 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.

7.3.2 Type universes

Besides the universes of of indexed containers and indexed descriptions, other generic representations of indexed families have been developed as well. Here we briefly discuss some alternative type universes.

Indexed functors

Löh and Magalhães propose in their paper *Generic Programming with Indexed Functors* [24] a type universe for generic programming in Agda, that is able to handle a large class of indexed datatypes. Their universe takes the universe of regular types as a basis.

The semantics of the universe, however, is not a functor `Set → Set`, but rather an *indexed* functor `(I → Set) → O → Set`. Additionally, they add some combinators, such as first order constructors to encode isomorphisms and fixpoints as part of their universe.

Combinatorial species

Combinatorial species [42] were originally developed as a mathematical framework, but can also be used as an alternative way of looking at datatypes. A species can, in terms of functional programming, be thought of as a type constructor with one polymorphic argument. Haskell's algebraid datatypes (or regular types in general) can be described by defining familiar combinators for species, such as sum and product.

Mutually recursive sums of product

On of the more simple representations is the so called *Sum of Products* view [15], 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, and is closely related to the universe of regular types. As we have seen when discussing regular types, 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 [29], and is available as a Haskell library through *Hackage*.

7.3.3 Generating constrained test data

Some work in the direction of generating constrained test data has already be done. For example, an approach to generation of constrained test data for Coq's QuickChick was proposed by Lampropoulos et al. [23] in their 2017 paper *Generating Good Generators for Inductive Relations*. They observe a the same common pattern on which the work in this thesis is based, where the required test data is of a simple type, but constrained by some precondition. The precondition can then be modeled as some inductive dependent relation indexed by said simple type. The `Sorted` datatype shown in chapter 1 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.

A slightly different approach was taken by Claessen and Duregaard [9], who adapt the techniques described by Duregaard [17] 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 Lazy SmallCheck [34]) until a value satisfying the imposed constrained is found.

7.3.4 Generating well-typed lambda terms

A problem often considered in literature is the generation of (well-typed) lambda terms [32, 19, 9]. 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.

An approach centered around the semantics of the simply typed lambda calculus is described by Pałka et al. [32]. Contrary to the work done by Claessen and Duregaard [9], 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 [30].

Wang [40] considered the problem of generating closed untyped lambda terms. Furthermore, Claessen and Duregaard [9] specifically apply their work to the problem of generating well-typed lambda terms, considering this particular problem as a running example throughout their paper.

7.4 Next steps & future work

As highlighted throughout this chapter, there is plenty of room for improvement upon the current state of the work. In this section we discuss a few of the possibilities.

Generator optimizations

As of yet, no work has been done to make generators more efficient. In practice, this means that the derived generators are likely to be too slow to generate usable data for most practical applications. One of the more promising approaches to fix this is by memoization. It is likely that a generator solves the same subproblem many times, so it could greatly benefit in terms of efficiency by reusing previous solutions. For example, when generating a well-typed lambda term, the generator might encounter the same combination of goal context and type multiple times, meaning that it solves the same subproblem more often than it needs to. We might find inspiration in the work done by Claessen and Duregård [17], who devised a memoization strategy that allows for efficient indexing of the enumeration of algebraic datatypes.

Generating mutually recursive families

As of yet, the library we have developed cannot be used to generate inhabitants of mutually recursive datatypes. This is a severe limitation, as many abstract syntax datatypes utilize mutual recursion. Type universes that are able to represent mutually recursive types exist [29][41], however they are not necessarily able to represent arbitrary indexed families. Bringert and Ranta [7] propose a pattern for converting mutually recursive types to a GADT, indexed with a tag that marks which datatype of the mutually recursive family a recursive position refers to. Yakushev et al. [41] use this technique for their approach. Our Haskell library is expressive enough to generate values for these GADT's, so this appears to be a promising approach to generation of mutually recursive indexed families.

Integration with existing testing frameworks

We have provided a sample instantiation of the abstract generator type as a bounded enumeration. However, theoretically it is possible to transform the abstract generator type to any desired generator type, as long as we are able to come up with a suitable mapping. This allows our library to be potentially integrated with external testing libaries by defining a mapping between the abstract generator type, and the type of generators used by a particular library. For SmallCheck, this is simple enough, as their generator type is almost exactly equal to our example instantiation. However, when transforming abstract generators to sampling generators (such as used in QuickCheck and Hedgehog), this mapping is not at all trivial. Most notably, it is not immediately clear how we should deal with generators that produce no elements, and recursive positions. Especially deriving *sized* generators for the QuickCheck library is challenging without including additional information in the abstract generator type.

Property based testing for GADT's

Most testing frameworks for Haskell currently only include functionality to generate values of regular algebraic datatypes. If we were to test a function that has a GADT as its input type, we are only left with the possibility of defining a custom generator for the type. Since the universe of indexed descriptions is potentially expressive enough to describe any GADT, we could leverage the work from this thesis to extend existing testing libraries with the possibility to automatically derive generators for GADT's.

Increasing usability and practicality of the Haskell Library

Currently, the provided library that implements generic generators for indexed descriptions is very basic, and requires the user to supply both a type family describing the datatype, as well as a singleton value. Additionally, they need to write a conversion function that converts the generated values to a non-indexed type. In terms of practicality and usability there is much to be gained by further automating this process. Possibilities include the definition of smart constructors to abstract over common patterns, and using template Haskell [36] to (partially) automate the definition of the singleton description from the type level description, making a "bootstrapping" approach in which we single out the difficult parts of generation for a larger datatype, and reuse our facilities to derive generators for these bottlenecks much more feasible.

Generating well-formed programs in a realistic programming language

The examples presented in this thesis are mostly relatively simple indexed families. In order to further investigate the practical applicability of our work, we think that it is essential to study how our approach applies to a more complex example. A prime candidate for this purpose would be term generation for Plutus Core, which motivated our work in the first place. James Chapman's formalization in Agda, which is available through IOHK's website [8], would be a natural starting point for this.

Code listings

Bibliography

[1] Ghc user's guide - datatype promotion. `https://downloads.haskell.org/ghc/8.6.2/docs/html/users_guide/glasgow_exts.html#datatype-promotion`. Accessed on 21-06-2019.

[2] The glasgow haskell compiler. `https://www.haskell.org/ghc/`. Accessed: 2019-07-09.

[3] Cardano - home of the ada cryptocurrency and technological platform (https://www.cardano.org), July 2019.

[4] Abel, A. Miniagda: Integrating sized and dependent types. *arXiv preprint arXiv:1012.4896* (2010).

[5] Altenkirch, T., Ghani, N., Hancock, P., McBride, C., and Morris, P. Indexed containers. *Journal of Functional Programming 25* (2015).

[6] Andoni, A., Daniliuc, D., Khurshid, S., and Marinov, D. Evaluating the "small scope hypothesis". In *In Popl* (2003), vol. 2, Citeseer.

[7] Bringert, B., and Ranta, A. A pattern for almost compositional functions. In *ACM SIGPLAN Notices* (2006), vol. 41, ACM, pp. 216–226.

[8] Chapman, J., Kireev, R., and Wadler, P. System f in agda, for fun and profit (draft). `https://iohk.io/research/papers/?__hsfp=3940508650#MBGHUW3V`. Accessed: 2019-07-12.

[9] Claessen, K., Duregård, J., and Pałka, M. H. Generating constrained random data with uniform distribution. *Journal of functional programming 25* (2015).

[10] Claessen, K., and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices 46*, 4 (2011), 53–64.

[11] Claessen, K., Smallbone, N., and Hughes, J. Quickspec: Guessing formal specifications using testing. In *International Conference on Tests and Proofs* (2010), Springer, pp. 6–21.

[12] Dagand, P.-E. *A Cosmology of Datatypes*. PhD thesis, Citeseer, 2013.

[13] Dagand, P.-É. The essence of ornaments. *Journal of Functional Programming 27* (2017).

[14] De Bruijn, N. G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)* (1972), vol. 75, Elsevier, pp. 381–392.

[15] de Vries, E., and Löh, A. True sums of products. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming* (2014), ACM, pp. 83–94.

[16] Dénès, M., Hritcu, C., Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. Quickchick: Property-based testing for coq. In *The Coq Workshop* (2014).

[17] Duregård, J., Jansson, P., and Wang, M. Feat: functional enumeration of algebraic types. *ACM SIGPLAN Notices 47*, 12 (2013), 61–72.

[18] Eisenberg, R. A., and Weirich, S. Dependently typed programming with singletons. *ACM SIGPLAN Notices 47*, 12 (2013), 117–130.

[19] Grygiel, K., and Lescanne, P. Counting and generating lambda terms. *Journal of Functional Programming 23*, 5 (2013), 594–628.

[20] Hinze, R., et al. Fun with phantom types. *The fun of programming* (2003), 245–262.

[21] Input-Output-Hk. Plutus specification (https://github.com/input-output-hk/plutus/tree/master/plutus-core-spec), April 2019.

[22] Jones, S. P. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[23] Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 45.

[24] Löh, A., and Magalhaes, J. P. Generic programming with indexed functors. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming* (2011), ACM, pp. 1–12.

[25] Magalhães, J. P., and Jeuring, J. Generic programming for indexed datatypes. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming* (2011), ACM, pp. 37–46.

[26] Martin-Löf, P. *Intuitionistic type theory*, vol. 9. Bibliopolis Naples, 1984.

[27] Matela Braquehais, R. *Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing*. PhD thesis, University of York, 2017.

[28] Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied logic 51*, 1-2 (1991), 125–157.

[29] Miraldo, V. C., and Serrano, A. Sums of products for mutually recursive datatypes: the appropriationist's view on generic programming. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development* (2018), ACM, pp. 65–77.

[30] Moczurad, M., Tyszkiewicz, J., and Zaionc, M. Statistical properties of simple types. *Mathematical Structures in Computer Science 10*, 5 (2000), 575–594.

[31] Norell, U. Dependently typed programming in agda. In *International School on Advanced Functional Programming* (2008), Springer, pp. 230–266.

[32] Pałka, M. H., Claessen, K., Russo, A., and Hughes, J. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (2011), ACM, pp. 91–97.

[33] Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L., and Pierce, B. C. Foundational property-based testing. In *International Conference on Interactive Theorem Proving* (2015), Springer, pp. 325–343.

[34] Runciman, C., Naylor, M., and Lindblad, F. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices* (2008), vol. 44, ACM, pp. 37–48.

[35] Schrijvers, T., Peyton Jones, S., Chakravarty, M., and Sulzmann, M. Type checking with open type functions. *ACM Sigplan Notices 43*, 9 (2008), 51–62.

[36] Sheard, T., and Jones, S. P. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell* (2002), ACM, pp. 1–16.

[37] Stanley, J. hedgehog: Hedgehog will eat all your bugs. https://hackage.haskell.org/package/hedgehog, 2019. [Online; accessed 26-Feb-2019].

[38] Wadler, P. Propositions as types. *Communications of the ACM 58*, 12 (2015), 75–84.

[39] Wadler, P., and Kokke, W. Programming language foundations in agda, debruijn: Inherently typed de bruijn representation. `https://plfa.github.io/DeBruijn/`. Accessed: 2019-06-13.

[40] Wang, J. Generating random lambda calculus terms. *Unpublished manuscript* (2005).

[41] Yakushev, A. R., Holdermans, S., Löh, A., and Jeuring, J. Generic programming with fixed points for mutually recursive datatypes. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 233–244.

[42] Yorgey, B. A. Species and functors and types, oh my! In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 147–158.

[43] Yorgey, B. A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., and Magalhães, J. P. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation* (2012), ACM, pp. 53–66.