Universiteit Utrecht

Faculty of Science

Dept. of Information and Computing Sciences

Thesis title

Author

C.R. van der Rest

Supervisor

Dr. W.S. Swierstra

Dr. M.M.T. Chakravarty

Dr. A. Serrano Mena

June 21, 2019

Contents

Declaration

Thanks to family, supervisor, friends and hops!

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

Abstract

1
Introduction

This thesis concerns itself with the generation of complex test data in the context of property based testing specifically, and generic programming for indexed datatypes in general.

1.1 Problem Statement

In *property based testing* is a technique in which the correctness of a program is asserted by defining properties that should hold over a program's output and behavior, and checking that those properties are true for a collection of input values. There exist many libraries for property based testing, of which QuickCheck [6] and SmallCheck [24] are perhaps the most notable in the realm of functional programming.

At first glance defining properties that capture the desired behavior of a program may seem like the most challenging aspect of property based testing. While this certainly can be difficult, one should not underestimate the effort that goes into generation of suitable test data. For example, suppose we are testing a function that operates on sorted lists. To do so, we would need a generator that produces sorted lists. Suppose we have a predicate that asserts sortedness:

$$
\begin{aligned}
&\mathit{isSorted} :: [\mathit{Int}] \rightarrow \mathit{Bool} \\
&\mathit{isSorted}\ [\,] \quad\quad\quad = \mathit{True} \\
&\mathit{isSorted}\ [x] \quad\quad\ \ = \mathit{True} \\
&\mathit{isSorted}\ (x : y : xs) = x \leq y \wedge \mathit{isSorted}\ (y : xs)
\end{aligned}
$$

We can use this predicate as a precondition for some property (i.e. a function $\mathit{prop} :: [\mathit{Int}] \rightarrow \mathit{Bool}$) that expects a sorted list as its input. However, this causes some problems.

```
Test.QuickCheck> quickCheck (sorted xs) ==> prop xs)
*** Gave up! Passed only 70 tests; 1000 discarded tests.
```

QuickCheck was not able to find enough lists that satisfy the predicate sorted! As it turns out, only *very few* random lists actually turn out to be sorted. The trouble does not stop there, since from those lists that QuickCheck is able to find that are sorted, most will only contain very few elements. This is simply a result of the fact that a small random list has a much higher probability of being sorted than a larger list. The way forward in this case is actually to define a custom generator that is specifically designed to produce sorted lists.

$$
\begin{aligned}
&\mathit{gen\_sorted} :: \mathit{Gen}\ [\mathit{Int}] \\
&\mathit{gen\_sorted} = \mathit{arbitrary} \gg\!= \mathit{return} \circ \mathit{diff} \\
&\quad \textbf{where}\ \mathit{diff} :: [\mathit{Int}] \rightarrow [\mathit{Int}] \\
&\quad\quad\quad \mathit{diff}\ [\,] \quad\quad = [\,] \\
&\quad\quad\quad \mathit{diff}\ (x : xs) = x : \mathit{map}\ (+x)\ (\mathit{diff}\ xs)
\end{aligned}
$$

In this case, the custom generator is not too complicated. However, as the preconditions of our properties grow more complex, so do their generators. For example, when testing a compiler, well-formedness of input programs is often a precondition of the test data. Generating well-formed programs is hard. Even synthesizing well-typed lambda terms is a surprisingly tricky problem [22, 14, 5].

We observe that the desired precondition of test data can often be expressed as using an *indexed family*. For example, the following indexed family describes sortedness for lists:

```
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 $List\ \mathbb{N}$. This means that if we are able to generate values that inhabit an indexed family such as *Sorted*, we are able to generate constrained test data. Some research has been done in this direction [15], but a generic procedure for generation of indexed families does not exist yet in the literature.

1.2 Research Question and Goals

This thesis aims to work towards an answer to the following question:

How can we obtain constrained test data by generically deriving enumeration and/or sampling mechanisms for indexed datatypes?

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.

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.

1.4 Thesis Structure

This thesis is structured as follows: in chapter 2 we discuss some relevant theoretical background and some of the work related to this thesis. Chapters 3 through 5 describe various type universes, and show how we may derive generators for any type in those universes. Additionally, we sketch how we may prove that the associated enumerations are complete. Chapter 6 is concerned with how we can implement these ideas in Haskell, and provide a comprehensive framework for the generation of well formed programs. Finally, chapter 7 provides a discussion of the work and lists some of the possible future work.

2
Background & Related Work

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

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

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

2.1 Type Theory

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

2.1.1 Intuitionistic Type Theory

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

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

2.1.2 The Curry-Howard Equivalence

The *Curry-Howard equivalence* establishes an isomorphism between *propositions and types* and *proofs and terms* [26]. This means that for any type there is a corresponding proposition, and

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

Table 2.1: Correspondence between classical logic and type theory

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

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

$$\text{tautology} : \forall \{P\ Q\ R\} \longrightarrow P * (Q + R) \longrightarrow (P * Q) + (P * R)$$

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

$$\text{tautology}\ (fst\ ,\ \text{inj}_1\ x) = \text{inj}_1\ (fst\ ,\ x)$$
$$\text{tautology}\ (fst\ ,\ \text{inj}_2\ y) = \text{inj}_2\ (fst\ ,\ y)$$

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

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

$$\text{P∧¬P⟶⊥} : \forall \{P\} \longrightarrow P * (P \longrightarrow \bot) \longrightarrow \bot$$
$$\text{P∧¬P⟶⊥}\ (P\ ,\ P{\longrightarrow}\bot) = P{\longrightarrow}\bot\ P$$

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

$$\text{P∨¬P} : \forall \{P\} \longrightarrow P + \neg\ P$$

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

2.1.3 Dependent Types

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

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

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

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

The Curry-Howard equivalence extends to Π- and Σ-types as well: they can be used to model universal and existential quantification [26] (??).

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

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

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

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

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

2.2 Agda

Agda is a programming language based on Intuitionistic type theory[21]. Its syntax is broadly similar to Haskell's, though Agda's type system is arguably more expressive, since types may depend on term level values.

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

2.2.1 Codata and Sized Types

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

> **Example** The following definition is perfectly fine in Haskell:
>
> $$nats :: [Int]$$
> $$nats = 0 : map\ (+1)\ nats$$
>
> Meanwhile, an equivalent definition in Agda gets rejected by the Termination checker. The recursive call to $nats$ has no arguments, so none are structurally smaller, thus the termination checker flags this call.
>
> ```
> nats : List ℕ
> nats = 0 :: map suc nats
> ```

However, as long as we use $nats$ sensibly, there does not need to be a problem. Nonterminating programs only arise with improper use of such a definition, for example by calculating the length of $nats$. We can prevent the termination checker from flagging these kind of operations by making the lazy semantics explicit, using *codata* and sized types. Codata is a general term for possible inifinite data, often described by a co-recursive definition. Sized types extend the space of function definitions that are recognized by the termination checker as terminating by tracking information about the size of values in types [2]. In the case of lists, this means that we explicitly specify that the recursive argument to the $\_ :: \_$ constructor is a *Thunk*, which should only be evaluated when needed:

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

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

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

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

> **Example** Consider a function that attempts to calculate the length of a $Colist$:
>
> ```
> length : ∀ {a : Set} → Colist a ∞ → ℕ
> length [] = 0
> length (x :: xs) = suc (length (xs .force))
> ```
>
> In this case $length$ is not accepted by the termination checker because the input colist is indexed with size ∞, meaning that there is no finite upper bound on its size. Hence, there is

no guarantee that our function terminates when inductively defined on the input colist.

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

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

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

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

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

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

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

2.2.2 Universe Polymorphism

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

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

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

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

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

2.3 Generic Programming and Type Universes

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

2.3.1 Design Pattern

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

1. First, we define a datatype $\mathcal{U}$ representing the structure of types, often called a *Universe*.

2. Next, we define a semantics $[\![\_]\!] : \mathcal{U} \to K$ that associates codes in $\mathcal{U}$ with an appropriate value of kind K. In practice this is often a functorial representation of kind $Set \to Set$.

3. Finally, we (often) define a fixed point combinator of type $(u : \mathcal{U}) \to Set$ that calculates the fixpoint of $[\![u]\!]$.

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

Given these ingredients we have everything we need at hand to write generic functions. Section 3 of Ulf Norell's *Dependently Typed Programming in Agda* [21] contains an in depth explanation of how this can be done in Agda. We will only give a rough sketch of the most common design pattern here. In general, a datatype generic function is supplied with a code $u : \mathcal{U}$, and returns a function whose type is dependent on the code it was supplied with.

> **Example** Suppose we are defining a generic procedure for decidable equality. We might use the following type signature for such a procedure:
>
> $$\_\stackrel{?}{=}\_ : \forall \{u : \mathcal{U}\} \longrightarrow (x : \text{Fix } u) \longrightarrow (y : \text{Fix } u) \longrightarrow x \equiv y \uplus \neg\, x \equiv y$$
>
> If we now define $\stackrel{?}{=}$ by induction over u, we have a decision procedure for decidable equality that may act on values on any type, provided their structure can be described as a code in $\mathcal{U}$.

2.3.2 Example Universes

There exist many different type universes. We will give a short overview of the universes used in this thesis here; they will be explained in more detail later on when we define generic generators for them. The literature review in section 2.6 contains a brief discussion of type universes beyond those used we used for generic enumeration.

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

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

Indexed Descriptions Using the universe of *Indexed Descriptions* [8] we can represent arbitrary indexed datatypes. This allows us to describe datatypes that are beyond what can be described using indexed containers, that is, datatypes with recursive subtrees that are interdependent or whose recursive subtrees have indices that cannot be uniquely determined from the index of a value.

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

2.4 Libraries for Property Based Testing

Property Based Testing aims to assert properties that universally hold for our programs by parameterizing tests over values and checking them against a collection of test values. Libraries for property based testing often include some kind of mechanism to automatically generate collections of test values. Existing tools take different approaches towards generation of test data: *QuickCheck* [6] randomly generates values within the test domain, while *SmallCheck* [24] and *LeanCheck* [18] exhaustively enumerate all values in the test domain up to a certain point. There exist many libraries for property based testing. For brevity we constrain ourselves here to those that are relevant in the domain of functional programming and/or haskell.

2.4.1 QuickCheck

Published in 2000 by Claessen & Hughes [6], 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`. Instances for most common Haskell types are supplied by the library. If a property fails on a testcase, QuickCheck supplies a counterexample. Consider the following faulty definition of $reverse$:

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

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

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

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

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

2.4.2 (Lazy) SmallCheck

Contrary to QuickCheck, SmallCheck [24] takes an *enumerative* approach to the generation of test data. While the approach to formulation and testing of properties is largely similar to QuickCheck's, test values are not generated at random, but rather exhaustively enumerated up to a certain *depth*. Zero-arity constructors have depth `0`, while the depth of any positive arity constructor is one greather than the maximum depth of its arguments. The motivation for this is the *small scope hypothesis*: if a program is incorrect, it will almost allways fail on some small input [4].

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

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

2.4.3 LeanCheck

Where SmallCheck uses a value's *depth* to bound the number of test values, LeanCheck uses a value's *size* [18], where size is defined as the number of construction applications of positive arity. Both SmallCheck and LeanCheck contain functionality to enumerate functions similar to QuickCheck's `Coarbitrary`.

2.4.4 Hegdgehog

Hedgehog [25] is a framework similar to QuickCheck, that aims to be a more modern alternative. It includes support for monadic effects in generators and concurrent checking of properties. Additionally it supports automatic schrinking for many datatypes. Unlike QuickCheck and SmallCheck, HedgeHog does not support (partial) automatic derivation of generators, but rather chooses to supply a comprehensive set of combinators, which the user can then use to assemble generators.

2.4.5 Feat

A downside to both SmallCheck and LeanCheck is that they do not provide an efficient way to generate or sample large test values. QuickCheck has no problem with either, but QuickCheck generators are often more tedious to write compared to their SmallCheck counterpart. Feat [12] aims to fill this gap by providing a way to efficiently enumerate algebraic types, employing memoization techniques to efficiently find the n^{th} element of an enumeration.

2.4.6 QuickChick: QuickCheck for Coq

QuickChick is a QuickCheck clone for the proof assistant Coq [11]. The fact that Coq is a proof assistant enables the user to reason about the testing framework itself [23]. This allows one, for example, to prove that generators adhere to some distribution.

2.4.7 QuickSpec: Automatic Generation of Specifications

A surprising application of property based testing is the automatic generation of program specifications, proposed by Claessen et al. [7] 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.

2.5 Generating Constrained Test Data

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

We run into prolems when we desire to generate test data for properties with a precondition. If a property's precondition is satisfied by few input values, it becomes unpractical to test such a property by simply generating random input data, and using rejection sampling to filter out those values that satisfy the desired precondition. We will often end up with very few testcases, and we will end up with a skewed distribution favoring those test values that have the largest probability to be picked at random (often these are the simplest values that satisfy the precondition).

The usual solution to this problem is to define a custom test data generator that only produces data that satisfies the precondition. There are cases in which this is not too difficult, however once we require more complex test data, such as well-formed programs, this is quite a challenging task.

2.5.1 Lambda Terms

A problem often considered in literature is the generation of (well-typed) lambda terms [22, 14, 5]. Good generation of arbitrary program terms is especially interesting in the context of testing compiler infrastructure, and lambda terms provide a natural first step towards that goal.

Claessen and Duregaard [5] adapt the techniques described by Duregaard [12] to allow efficient generation of constrained data. They use a variation on rejection sampling, where the space of values is gradually refined by rejecting classes of values through partial evaluation (similar to SmallCheck [24]) until a value satisfying the imposed constrained is found.

An alternative approach centered around the semantics of the simply typed lambda calculus is described by Pałka et al. [22]. Contrary to the work done by Claessen and Duregaard [5], 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 [20].

Wang [28] considers the problem of generating closed untyped lambda terms.

2.5.2 Inductive Relations in Coq

An approach to generation of constrained test data for Coq's QuickChick was proposed by Lampropoulos et al. [15] in their 2017 paper *Generating Good Generators for Inductive Relations*. They observe a common pattern where the required test data is of a simple type, but constrained by some precondition. The precondition is then given by some inductive dependent relation indexed by said simple type. The *Sorted* datatype shown in section ?? is a good example of this

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

2.6 Generic Programming & Type Universes

Many type universes have been developed beyond those used in this thesis, some of which are also designed to describe (a subset of) indexed datatypes. We describe a few of them here, and briefly discuss how they relate to the universes we used.

2.6.1 SOP (Sum of Products)

On of the more simple representations is the so called *Sum of Products* view [10], 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 will see (for example in section ??), other universes too employ sum and product combinators to describe the structure of datatypes, though they do not necessarily enforce the representation to be in disjunctive normal form. Sum of Products, in its simplest form, cannot represent mutually recursive families of datatypes. An extension that allows this has been developed in [19], and is available as a Haskell library through *Hackage*.

2.6.2 W-Types

Introduced by Per Martin-Löf [17], *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:

$$\begin{array}{l} \mathrm{data}\ \mathrm{WType}\ (S : \mathrm{Set})\ (P : S \longrightarrow \mathrm{Set}) : \mathrm{Set}\ \mathrm{where} \\ \quad \mathrm{sup} : (s : S) \longrightarrow (P\ s \longrightarrow \mathrm{WType}\ S\ P) \longrightarrow \mathrm{WType}\ S\ P \end{array}$$

However, we can use an alternate definition where we separate the universe into codes, semantics and a fixpoint operation (listing 2.6.2)

We take this redundant 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 emphasize the the similarities between W-types, and the universe of indexed containers, which will be further discussed in (TODO ref chapter 6)

Listing 2.1: 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
```

Example Let us look at the natural numbers (listing A.1) as an example. We can define the following W-type that is isomorphic to $\mathbb{N}$:

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

The $\mathbb{N}$ 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. The isomorphism between $\mathbb{N}$ and $W\mathbb{N}$ is established in listing ??.

Listing 2.2: Isomorphism between $\mathbb{N}$ and $W\mathbb{N}$

```
fromℕ : ℕ → Wℕ
fromℕ zero = In (false , λ())
fromℕ (suc n) = In (true , λ { tt → fromℕ n} )

toℕ : Wℕ → ℕ
toℕ (In (false , f)) = zero
toℕ (In (true , f)) = suc (toℕ (f tt))

isoℕ₁ : ∀ {n : ℕ} → toℕ (fromℕ n) ≡ n
isoℕ₁ {zero} = refl
isoℕ₁ {suc n} = cong suc isoℕ₁

isoℕ₂ : ∀ {n : Wℕ} → fromℕ (toℕ n) ≡ n
isoℕ₂ {In (false , f)} = cong (λ x → In (false , x)) (funext λ {})
isoℕ₂ {In (true , f)}  = cong (λ x → In (true , x))  (funext isoℕ₂)
```

2.6.3 Indexed Functors

Löh and Magalhães propose in their paper *Generic Programming with Indexed Functors* [16] 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 $Set \to Set$, but rather an *indexed* functor $(I \to Set) \to O \to Set$. Additionally, they add some combinators, such as first order constructors to encode isomorphisms and fixpoints as part of their universe.

2.6.4 Combinatorial Species

Combinatorial Species. Combinatorial species [29] 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 ADTs (or regular types in general) can be described by definining familiar combinators for species, such as sum and product.

3
Generic Generators for Regular types

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

3.1 The universe of regular types

Though the exact definition may vary across sources, the universe of regular types is generally regarded to consist of the *empty type* (or $\mathbb{0}$), the unit type (or $\mathbb{1}$) and constants types. It is closed under both products and coproducts [1]. We can define a datatype for this universe in Agda as shown in lising 3.1

Listing 3.1: The universe of regular types

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

The semantics associated with the *Reg* datatype, as shown in listing 3.1, map a code to a functorial representation of a datatype, commonly known as its *pattern functor*. The datatype that is represented by a code is isomorphic to the least fixpoint of its pattern functor. We fix pattern functors using the following fixpoint combinator:

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

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

[1] This roughly corresponds to datatypes in Haskell 98

Listing 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
```

```
ℕ' : Set
ℕ' = Fix (U ⊕ I)
```

We convince ourselves that ℕ′ is indeed equivalent to ℕ by defining conversion functions, and showing their composition is extensionally equal to the identity function, shown in listing 3.1.

Listing 3.3: Isomorphism between ℕ and ℕ′

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

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

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

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

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

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

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

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

3.1.1 Non-regular data types

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

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

3.2 Generic Generators for regular types

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

3.2.1 Defining functions over codes

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

$$\text{deriveGen} : (c : \text{Reg}) \rightarrow \text{Gen}\ (\text{Fix}\ c)\ (\text{Fix}\ c)$$

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

$$\text{deriveGen} : (c\ c' : \text{Reg}) \rightarrow \text{Gen}\ ([\![\ c\ ]\!]\ (\text{Fix}\ c'))\ ([\![\ c'\ ]\!]\ (\text{Fix}\ c'))$$

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

3.2.2 Composing generic generators

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

$$\begin{aligned}&\text{deriveGen Z}\ c' = \text{empty}\\&\text{deriveGen U}\ c' = \text{pure tt}\end{aligned}$$

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

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

```
deriveGen I c' = ⦇ In μ ⦈
```

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

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

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

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

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

> **Example** We derive a generator for natural numbers by invoking $genericGen$ on the appropriate code $U \oplus I$, and applying the isomorphism defined in listing ?? to its results:
>
> ```
> genℕ : Gen ℕ ℕ
> genℕ = ⦇ (_≃_.to ℕ≃ℕ') (Call (genericGen (U ⊕ I))) ⦈
> ```

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

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

3.3 Constant Types

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

> **Example** Given the code representing natural numbers ($U \oplus I$) and lists ($U \oplus (C \otimes I)$, where C is a code representing the type of elements in the list), we might be tempted to try and replace C with the code for natural numbers in the code for lists:
>
> ```
> listℕ : Set
> listℕ = Fix (U ⊕ ((U ⊗ I) ⊗ I))
> ```
>
> This code does not describe lists of natural numbers. The problem here is that the two

recursive positions refer to the *same* code, which is incorrect. We need the first I to refer to the code of natural numbers, and the second I to refer to the entire code.

3.3.1 Definition and Semantics

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

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

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

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

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

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

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

3.3.2 Generic Generators for Constant Typse

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

We have the programmer supply the necessary generators by defining a *metadata* structure, indexed by a code, that carries additional information for every K constructor used. We then parameterize $deriveGen$ with a metadata structure, indexed by the code we are inducting over. The definition of the metadata structure is shown in listing 3.3.2.

Listing 3.4: Metadata structure carrying additional information for constant types

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

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

$$\text{deriveGen} : (c\ c' : \text{Reg}) \longrightarrow \text{KInfo}\ (\lambda\ S \longrightarrow \text{Gen}\ S\ S)\ c \longrightarrow \text{Gen}\ (⟦\ c\ ⟧\ (\text{Fix}\ c'))\ (⟦\ c'\ ⟧\ (\text{Fix}\ c'))$$

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

$$\text{deriveGen}\ (\text{K}\ x)\ c'\ (\text{K\textasciitilde}\ g) = \text{Call}\ g$$

3.4 Complete Enumerators For Regular Types

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

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

$$\text{genericGen-Complete} : \forall\ \{c\ x\} \longrightarrow \exists[\ n\ ]\ (x \in \text{toList}\ (\text{genericGen}\ c)\ (\text{genericGen}\ c)\ n)$$

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

$$\text{deriveGen-Complete} : \forall\ \{c\ c'\ x\} \longrightarrow \exists[\ n\ ]\ (x \in \text{toList}\ (\text{deriveGen}\ c\ c')\ (\text{deriveGen}\ c'\ c')\ n)$$

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

The completeness proof roughly follows the following steps:

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

3.4.1 Combinator Correctness

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

$$\begin{array}{l}\text{deriveGen-Complete}\ \{\text{Z}\}\ \{c'\}\ \{()\}\\ \text{deriveGen-Complete}\ \{\text{U}\}\ \{c'\}\ \{\text{tt}\} = 1\ ,\ \text{here}\end{array}$$

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

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

```
toList (deriveGen (c_l ⊕ c_r) c') (deriveGen c' c') n
  ≡ merge (toList ⦇ inj₁ (deriveGen c_l c') ⦈ (deriveGen c' c') n)
          (toList ⦇ inj₂ (deriveGen c_r c') ⦈ (deriveGen c' c') n)
```

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

```
merge-complete-left  : ∀ {A} {xs_l xs_r : List A} {x : A} → x ∈ xs_l → x ∈ merge xs_l xs_r
merge-complete-right : ∀ {A} {xs_l xs_r : List A} {x : A} → x ∈ xs_r → x ∈ merge xs_l xs_r
```

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

```
toList (deriveGen (c_l ⊗ c_r) c') (deriveGen c' c') n
  ≡ ⦇ (toList (deriveGen c_l c') (deriveGen c' c') n)
    , (toList (deriveGen c_r c') (deriveGen c' c') n) ⦈
```

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

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

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

3.4.2 Completeness for Constant Types

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

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

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

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

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

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

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

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

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

3.4.3 Generator Monotonicity

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

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

$$\begin{aligned} n \leq m &\longrightarrow x \in \text{toList}\ (\text{C.deriveGen}\ c\ c'\ (\blacktriangleleft\ i))\ (\text{C.deriveGen}\ c'\ c'\ (\blacktriangleleft\ i'))\ n \\ &\longrightarrow x \in \text{toList}\ (\text{C.deriveGen}\ c\ c'\ (\blacktriangleleft\ i))\ (\text{C.deriveGen}\ c'\ c'\ (\blacktriangleleft\ i'))\ m \end{aligned}$$

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

3.4.4 Final Proof Sketch

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

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

$$\text{isoGen} : \forall\ \{A\} \longrightarrow \{\!\{\ p : \text{Regular}\ A\ \}\!\} \longrightarrow \Sigma[\ g \in \text{Gen}\ A\ A\ ]\ \forall\ \{x\} \longrightarrow \exists[\ n\ ]\ (x \in \text{toList}\ g\ g\ n)$$

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

4

Deriving Generators for Indexed Containers

This chapter discusses the universe of *indexed containers* [3], which provide a generic framework to describe those datatypes that can be defined by induction on their index type. Examples of datatypes we can describe using this universe include finite types ??, vectors ?? and well-scoped lambda terms. In this chapter, we give the definition for this universe together with a few examples, and show how a generic generator may be derived for indexed containers.

4.1 Universe Description

We choose to follow the representation used by Dagand in *The Essence Of Ornaments* [9], which provides an excellent introduction to indexed containers, alongside numerous examples. Just as in the previous chapter, we follow the pattern of first defining a datatype describing codes before giving the semantics and fixpoint operation.

4.1.1 Definition

Recall our definition of *W-types* in section 2.6.2. We purposefully split the canonical definition into three separate definitions for codes, semantics and fixpoint operation. If we consider the datatype describing codes in the universe of indexed descriptions (listing 4.1.1), their similarities become clear. Signatures consist of a triple of *operations*, *arities* and *typing discipline* .

Listing 4.1: 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*, describing the set of available operations. The major difference is that the operations in a signature are parameterized over the index type. Similarly, arity corresponds to position in a W-type, describing the set of recursive subtrees for a given operation. Again, a signature's arity is parameterized over the index type. The typing discipline maps arities to the indices of the corresponding subtrees.

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 a W-type, which maps a code to a value in $Set \to Set$, the semantics of a signature are parameterized over the index type, meaning they map a signature to a value in $(I \to Set) \to (I \to Set)$. The semantics are shown in listing 4.1.1.

Listing 4.2: The semantics of a signature

$$
\begin{aligned}
&[\![\_]\!] : \forall \{I\} \longrightarrow \mathsf{Sig}\ I \longrightarrow (I \longrightarrow \mathsf{Set}) \longrightarrow I \longrightarrow \mathsf{Set} \\
&[\![\ Op \lhd Ar \mid Ty\ ]\!]\ r\ i = \Sigma[\ op \in Op\ i\ ]\ ((ar : Ar\ op) \longrightarrow r\ (Ty\ ar))
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{Fix}\ \{I : \mathsf{Set}\}\ (\Sigma : \mathsf{Sig}\ I)\ (i : I) : \mathsf{Set}\ \mathsf{where} \\
&\quad \mathsf{In} : [\![\ \Sigma\ ]\!]\ (\mathsf{Fix}\ \Sigma)\ i \longrightarrow \mathsf{Fix}\ \Sigma\ i
\end{aligned}
$$

It is worth noting that, since $Set \cong \top \to Set$, we can describe non-indexed datatypes as an indexed container by choosing $\top$ as the index type. More precisely, there exists a bijection between W-types and signatures indexed with the unit type, such that for every W-type, its interpretation is isomorphic to the interpretation of the corresponding signature, and vice versa.

4.1.2 Example Signatures

Let us now consider a few examples of datatypes represented as a signature.

> **Example** We start by defining a suitable set of operations. The $\mathbb{N}$ datatype has two constructor, so we return a type with two inhabitants. We use $\top$ as the index of the signature, since $\mathbb{N}$ is a non-indexed datatype.
>
> $$
> \begin{aligned}
> &\mathsf{Op\text{-}Nat} : \top \longrightarrow \mathsf{Set} \\
> &\mathsf{Op\text{-}Nat}\ \mathsf{tt} = \top \uplus \top
> \end{aligned}
> $$
>
> 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 ($\bot$), while the suc constructor has a single recursive argument, so its arity is the unit type ($\top$).
>
> $$
> \begin{aligned}
> &\mathsf{Ar\text{-}Nat} : \mathsf{Op\text{-}Nat}\ \mathsf{tt} \longrightarrow \mathsf{Set} \\
> &\mathsf{Ar\text{-}Nat}\ (\mathsf{inj}_1\ \mathsf{tt}) = \bot \\
> &\mathsf{Ar\text{-}Nat}\ (\mathsf{inj}_2\ \mathsf{tt}) = \top
> \end{aligned}
> $$
>
> 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 $\mathbb{N}$.

$$\begin{aligned}
&\Sigma\text{-}\mathbb{N} : \mathsf{Sig}\ \top \\
&\Sigma\text{-}\mathbb{N} = \text{Op-Nat} \triangleleft \text{Ar-Nat} \mid \lambda\ _ \longrightarrow \mathsf{tt}
\end{aligned}$$

The signature for natural numbers is quite similar to how we would represent them as a W-type. This example, however, does not tell us much about how signatures enable us to represent indexed datatypes, so let us look at another example.

Example We consider the type of finite sets (listing A.2). Contrary to natural numbers, the set of available operations varies with different indices. That is, $\mathit{Fin}\ 0$ is uninhabited, so the set of operations associated with index 0 is empty. A value of type $\mathit{Fin}\ (\mathit{suc}\ n)$ can be constructed using both suc and zero, hence the set of associated operations has two elements:

$$\begin{aligned}
&\text{Op-Fin} : \mathbb{N} \longrightarrow \mathsf{Set} \\
&\text{Op-Fin zero} \quad\ = \bot \\
&\text{Op-Fin } (\mathsf{suc}\ n) = \top \uplus \top
\end{aligned}$$

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

$$\begin{aligned}
&\text{Ar-Fin} : \forall\ \{n\} \longrightarrow \text{Op-Fin}\ n \longrightarrow \mathsf{Set} \\
&\text{Ar-Fin}\ \{\mathsf{zero}\}\ \ () \\
&\text{Ar-Fin}\ \{\mathsf{suc}\ n\}\ (\mathsf{inj}_1\ \mathsf{tt}) = \bot \\
&\text{Ar-Fin}\ \{\mathsf{suc}\ n\}\ (\mathsf{inj}_2\ \mathsf{tt}) = \top
\end{aligned}$$

Recall the type of the suc constructor: $\mathit{Fin}\ n \to \mathit{Fin}\ (\mathit{suc}\ n)$. 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 $\mathit{suc}\ n$ to n, if the index is greater than 0, and the operation corresponding to the suc constructor is selected.

$$\begin{aligned}
&\text{Ty-Fin} : \forall\ \{n\}\ \{op : \text{Op-Fin}\ n\} \longrightarrow \text{Ar-Fin}\ op \longrightarrow \mathbb{N} \\
&\text{Ty-Fin}\ \{\mathsf{zero}\}\ \ \{()\}\ ar \\
&\text{Ty-Fin}\ \{\mathsf{suc}\ n\}\ \{\mathsf{inj}_1\ \mathsf{tt}\}\ () \\
&\text{Ty-Fin}\ \{\mathsf{suc}\ n\}\ \{\mathsf{inj}_2\ \mathsf{tt}\}\ \mathsf{tt} = n
\end{aligned}$$

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

$$\begin{aligned}
&\Sigma\text{-Fin} : \mathsf{Sig}\ \mathbb{N} \\
&\Sigma\text{-Fin} = \text{Op-Fin} \triangleleft \text{Ar-Fin} \mid \text{Ty-Fin}
\end{aligned}$$

One thing to keep in mind while 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 of *extensional equality* for function types:

$$\text{funext'} : \forall\ \{A : \mathsf{Set}\}\ \{B : A \longrightarrow \mathsf{Set}\} \longrightarrow (f\ g : (a : A) \longrightarrow B\ a) \longrightarrow (\forall\ \{x\} \longrightarrow f\ x \equiv g\ x) \longrightarrow f \equiv g$$

One aspect we have not yet addressed is how to represent parameterized types, such as $Vec\ a$ (listing A.3). Indexed containers do not have an explicit way to refer to other types, such as is the case with regular types, but rather include this kind of information as part of a type's operations.

Example 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

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

4.2 Generic Generators for Indexed Containers

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 implies 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 (4.2.1), where $FixR$ and InR denote the fixpoint operation for regular types. The fixpoint operation for signatures remains the same.

This implies that the definition of signatures changes slightly as well.

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
```

Listing 4.3: 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))
```

```
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 the 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 [24] *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 4.2.2.

Since part of the semantics of an indexed container is 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)
```

Listing 4.4: 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'}) ⦈
```

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 we need to make the dependency between argument and result type explicit in the type in order for Agda to be able to solve all metavariables.

4.2.3 Constructing the Generator

We are now ready to construct a the generic generator for indexed descriptions. Recall that *deriveGen* returns a generator for the regular type represented by r.

```
Σ-generate : ∀ {I : Set} → (Σ : Sig I) → (i : I) → Gen (FixΣ Σ i) (FixΣ Σ i)
Σ-generate (Op ◃ Ar ∣ Ty) i = do
  op ← ` deriveGen (Op i)
  ar ← ` Π-cogenerate (Ar op) (Ar op) λ _ → μ
  pure (InΣ (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.

Unfortunately, we have not been able to assemble a completeness proof for the enumeration derived using Σ-*generate*. 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 require induction over function types in order to complete the proof.

5

Deriving Generators for Indexed Descriptions

We use the generic description for indexed datatypes proposed by Dagand [8] in his PhD thesis. First, we give the definition and semantics of this universe, before showing how a generator can be derived from codes in this universe. Finally, we prove that the enumerations resulting from these generators are complete.

5.1 Universe Description

5.1.1 Definition

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

1. A type parameter $I : Set$, describing the type of indices.
2. A generalized coproduct, `σ, that denotes choice between n constructors, in favor of the ⊕ combinator.
3. A combinator denoting dependent pairs.
4. Recursive positions storing the index of recursive values.

This amounts to the Agda datatype describing indexed descriptions shown in listing 5.1.1.

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
```

Notice how we retain the regular product of codes as a first order construct in our universe. The *Sl* datatype is used to select the right branch from the generic coproduct, and is isomorphic to the *Fin* datatype.

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

The semantics associated with the *IDesc* universe is largely the same as the semantics of the universe of regular types. The key difference is that we do not map codes to a functor $Set \rightarrow Set$, but rather to $IDesc\ I \rightarrow (I \rightarrow Set) \rightarrow Set$. The semantics is shown in listing 6.2.3.

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 do not require a separate constructor representing the empty type, as we can encode it as a coproduct over zero constructors: $`\sigma\ 0\ \lambda\ ()$.

We calculate the fixpoint of interpreted codes using the following fixpoint combinator:

```
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, listing A.2, 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, althoug convenient in this case. A different, but equally valid description exists, in which we use the `Σ constructor to explicitly enforce the constraint that the index n is the successor of some n'.

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

Listing 5.1.1 establishes that the fixpoint of *finD* is indeed isomorphic to *Fin*.

Listing 5.3: Isomorphism between *Fix finD* and *Fin*

```
fromFin : ∀ {n} → Fin n → Fix finD n
fromFin {suc _} zero     = In (·      , tt)
fromFin {suc _} (suc fn) = In (▷ · , fromFin fn)

toFin : ∀ {n} → Fix finD n → Fin n
toFin {suc _} (In (·     , _)) = zero
toFin {suc _} (In (▷ · , r))  = suc (toFin r)

isoFin₁ : ∀ {n fn} → toFin {n} (fromFin fn) ≡ fn
isoFin₁ {suc _} {zero}  = refl
isoFin₁ {suc _} {suc _} = cong suc isoFin₁

isoFin₂ : ∀ {n fn} → fromFin {n} (toFin fn) ≡ fn
isoFin₂ {suc _} {In (·      , _)} = refl
isoFin₂ {suc _} {In (▷ ·   , _)} = cong (λ x → In (▷ · , x)) isoFin₂
```

We generalize 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 D : Σ[ φ ∈ (I → IDesc I) ] ((i : I) → A i ≃ Fix φ i)
```

5.1.2 Exmample: describing well typed lambda terms

To demonstrate the expressiveness of the *IDesc* universe, and to show how one might model a more complex datatype, we consider simply typed lambda terms as an example. We assume raw terms as described in listing A.6. We type terms using the universe described in listing A.4.

Modelling SLC in Agda

We write $\Gamma \ni \alpha : \tau$ to signify that α has type τ in Γ. Context membership is described by the following inference rules:

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

We describe these inference rules in Agda using an inductive datatype, shown in listing 5.1.2, indexed with a type and a context, whose inhabitants correspond to all proofs that a context Γ contains a variable of type 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:

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

Listing 5.4: Context membership in Agda

```
data _∋_ : Ctx → Ty → Set where
  [Pop] :   ∀ {Γ α τ}
            -------------
          → Γ , α ∶ τ ∋ τ

  [Top] :   ∀ {Γ α τ σ} → Γ ∋ τ
            ---------------------
          → Γ , α ∶ σ ∋ τ
```

We model these inference rules in Agda using a two way relation between contexts and types whose inhabitants correspond to all terms that have a given type under a given context (listing 5.1.2)

Listing 5.5: Well-typed lambda terms as a two way relation

```
data _⊢_ : Ctx → Ty → Set where
  [Var]  :  ∀ {Γ τ} → Γ ∋ τ
            ---------------
          → Γ ⊢ τ

  [Abs]  :  ∀ {Γ α σ τ} → Γ , α ∶ σ ⊢ τ
            ---------------------------
          → Γ ⊢ (σ ‘→ τ)

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

Given an inhabitant $\Gamma \vdash \tau$ of this relationship, we can write a function $toTerm$ that transforms the typing judgement to its corresponding untyped term.

$$\text{toTerm} : \forall \{\Gamma\ \tau\} \rightarrow \Gamma \vdash \tau \rightarrow \text{RT}$$

The term returned by $toTerm$ will has type τ under context Γ.

Describing 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 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. If we consider a description for lambda terms using induction on the index (listing ??), we see that it has a downside. The same constructor may yield a value with different index patterns.

Listing 5.6: A description for well typed terms using induction on the index type

```
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) } }
  where 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 , α ∶ σ , τ))
        appDesc : Ctx × Ty → IDesc (Ctx × Ty)
        appDesc (Γ , τ) = ‘Σ Ty (λ σ → ‘var (Γ , σ ‘→ τ) ‘× ‘var (Γ , σ))
```

For example, the application rule may yield both a function type as well as a ground type, we need to include a description for this constructor in both branches when pattern matching on the input type. If we compare the inductive description to a version that explicitly includes a constraint that the input type is a function type in case of the description for the abstraction rule, we end up with a much more succinct description.

However, using such a description comes at a price. The descriptions used will become more complex, hence their interpretation will too. Additionally, we delay the point at which it becomes apparent that a constructor could not have been used to create a value with the input index. This makes the generators for indexed descriptions, which we will derive in the next section, potentially more computationally intensive to run when derived from a description that uses explicit constraints, compared to an equivalent description that is defined by induction on the index.

Listing 5.7: A description for well typed terms using explicit constraints

```
wt : Ctx × Ty → IDesc (Ctx × Ty)
wt (Γ , τ) =
 ‘σ 3 λ { ·           → ‘Σ (Γ ∋ τ) λ _ → ‘1
        ; (▷ ·)       → ‘Σ (Σ (Ty × Ty) λ { (σ , τ’) → τ ≡ σ ‘→ τ’ })
                           λ { ((σ , τ’) , refl)
                              → ‘Σ ℕ (λ α → ‘var (Γ , α ∶ σ , τ’)) }
        ; (▷ (▷ ·)) → ‘Σ Ty λ {σ → ‘var (Γ , σ ‘→ τ) ‘× ‘var (Γ , σ) }
        }
```

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

$$\text{desc}\simeq : \forall \{\Gamma\ \tau\} \longrightarrow \text{Fix Inductive.wt } (\Gamma\ ,\ \tau) \cong \text{Fix Constrained.wt } (\Gamma\ ,\ \tau)$$

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:

$$\text{wt}\simeq : \forall \{\Gamma\ \tau\} \longrightarrow \text{Fix Constrained.wt } (\Gamma\ ,\ \tau) \longrightarrow \Gamma \vdash \tau$$

Using the transitivity of $\_ \simeq \_$, we can show that the inductive description also describes well typed terms.

5.2 Generic Generators for Indexed Descriptions

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:

$$\begin{aligned}\text{IDesc-gen} :\ & \forall \{I\} \{i : I\} \longrightarrow (\delta : \text{IDesc } I) \longrightarrow (\varphi : I \longrightarrow \text{IDesc } I) \\ & \longrightarrow \text{Gen}_i\ (\llbracket\ \delta\ \rrbracket\ (\text{Fix } \varphi))\ (\lambda\ i \longrightarrow \llbracket\ \varphi\ i\ \rrbracket\ (\text{Fix } \varphi))\ i\end{aligned}$$

We take a value of type $IDesc\ I$ (the description we are inducting over) and a function $I \rightarrow IDesc\ I$ (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. The definition for $`var$, $`1$ and $`\times$ can be readily transferred from the definition of $deriveGen$. The generic generators for the generalized coproduct and the $`\Sigma$ constructor are slightly more involved, since the both have to produce dependent pairs. Since the generalized coproduct is a particular instantiation of $`\Sigma$, we will consider it first.

$$\begin{aligned}&\text{IDesc-gen } \{i = i\}\ (`\sigma\ n\ T)\ \varphi = \text{do} \\ &\quad sl \longleftarrow \text{Call}_i\ \{x = i\}\ n\ \text{Sl-gen} \\ &\quad t \longleftarrow \text{IDesc-gen } (T\ sl)\ \varphi \\ &\quad \text{pure } (\_,\_\ \{B = \lambda\ sl \longrightarrow \llbracket\ T\ sl\ \rrbracket\ (\text{Fix } \varphi)\}\ sl\ t)\end{aligned}$$

Here we assume that $Sl\text{-}gen : (n : \mathbb{N}) \rightarrow Gen_i\ (Sl\ n)\ Sl\ n$ 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 pass around some metavariables in order to convince Agda that everything is in order.

We can reuse this exact same structure when defining a generator for $`\Sigma$, however since the type of its first element is chosen by the user, we cannot define a generator for it in adavance, as we did for the selector type. We use the same approach using a metadata structure as for regular types to have the programmer pass appropriate generators as input to $IDesc - gen$. We define this metadata structure as a datatype **data** $IDescM\ \{I\}\ (P : Set \rightarrow Set) : IDesc\ I \rightarrow Set$. Its constructors are largely equivalent to the metadata structure used for regular types (section 3.3.2), with the key difference being that we now require the programmer to store a piece of data depending on the type of the first element of a $`\Sigma$:

$$\begin{aligned}`\Sigma\sim :\ & \forall \{S : \text{Set}\} \{T : S \longrightarrow \text{IDesc } I\} \longrightarrow P\ S \longrightarrow ((s : S) \longrightarrow \text{IDescM } P\ (T\ s)) \\ & \longrightarrow \text{IDescM } P\ (`\Sigma\ S\ T)\end{aligned}$$

The constructor of the $IDescM$ type associated with the generalized coproduct follows the same structure as $`\Sigma\sim$, 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 $`\Sigma$ constructor, we can define a generator for its interpretation:

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

By using an instance of $Describe$, we may use the isomorphism stored within to convert the values generated by $IDesc-gen$ to the type we are describing.

5.3 Completeness Proof for Enumerators Derived From Indexed Descriptions

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 are structurally quite similar, so are their completeness proofs. This means that we can recycle a considerable portion of the proof 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) → Genᵢ (P i) P i) → Set
Complete {I} {P} i gen = ∀ {x : P i} → ∃[ n ] (x ∈ interpretᵢ gen i (gen i) n)
```

Which is essentially the same property we used for regular types, adapted for usage with indexed types. The completeness proofs for $`var$, $`1$ and $`\times$ can be transplanted from the proof for regular types with only a few minor changes. However, generators for $`\sigma$ and $`\Sigma$ are assembled using *monadic bind*, for which we have not yet proven that it satisfies our notion of completeness. Defining what completeness even means for $\gg=$ is very difficult in itself, but luckily since both usages in $IDesc-gen$ follow the same structure, we only have to prove a completeness property over our specific use of the bind operator. We replace $Complete$ with a slight variation that makes the value x we are quantifying over explicit in the type.

Listing 5.8: Completeness for the bind operator

```
»=-Complete : ∀ {I A} {P : A → Set} {T : I → Set} {x y}
                {g : Genᵢ A T x} {g' : (v : A) → Genᵢ (P v) T y}
                {x : Σ A P} {tg : (i : I) → Genᵢ (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
```

Given that the proof is supplied with a metadata structure that provides generators with completeness proofs for all `$\Sigma$ in a description, and that we have a completeness proof over the generator for the selection type in scope, we can complete the proof for the case of `σ and `Σ with a call to $bind - Complete$.

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.

6 Implementation 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.

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 $Int \to [\, a\,]$).
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 [8], and we make extensive use of both dependent pairs and dependent function types in our definition in Agda. Haskell's type system does not facilitate such relations between types. We will use a lot of *singleton types* [13] to work around this limitation. Singleton types is 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 an example. We add an extra type parameter $a :: *$ besides a parameter describing the index type, which is the raw type we will be converting to. Listing **??** contains tbe definition of $IDesc\ a\ i$, with constructors for *empty types*, *unit types*, *recursive positions* and *product types*. We purposefully omit the constructors for the generalized coproduct and the `Σ combinator, since transferring them to Haskell is slightly more involved.

Listing 6.1: Definition of $\mathit{IDesc}\ a\ i$ in Haskell

$$
\begin{array}{l}
\textbf{data}\ \mathit{IDesc}\ (a :: *)\ (i :: *)\ \textbf{where} \\
\quad \mathit{One} \quad\ \, :: \mathit{IDesc}\ a\ i \\
\quad \mathit{Empty} :: \mathit{IDesc}\ a\ i \\
\quad \mathit{Var} \quad\ \, :: i \to \mathit{IDesc}\ a\ i \\
\quad (:*:) \ \ :: \mathit{IDesc}\ a\ i \to \mathit{IDesc}\ a\ i \to \mathit{IDesc}\ a\ i
\end{array}
$$

6.2.1 Generalized coproducts

Recall that the generalized coproduct in Dagand's representation was given by a natural number n, and a function taking a finite type of size n and returning a description. We choose to use a *size indexed list* or *vector* in favor of a function from finite type to description. Assuming a type Nat is in scope, we use the following GADT to describe a vector:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Vec}\ (a :: *)\ (n :: \mathit{Nat})\ \textbf{where} \\
\quad \mathit{VNil} :: \mathit{Vec}\ a\ \mathit{Zero} \\
\quad (:::) :: a \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (\mathit{Suc}\ n)
\end{array}
$$

Given a singleton instance of the Nat datatype, $\mathit{Sing}\ n$, we can define a generalized coproduct:

$$(:+>) :: \mathit{Sing}\ n \to \mathit{Vec}\ (\mathit{IDesc}\ a\ i)\ n \to \mathit{IDesc}\ a\ i$$

Which describes a choice between n descriptions.

6.2.2 The `Σ combinator

Originally, the semantics of the `Σ combinator is a dependent pair. However, we observe that this dependency between the first and second element of the pair is not necessary to represent any of the examples we have looked at. For this reason, we choose a slightly weaker version of the `Σ combinator, which does not have a dependent pair as its semantics, but rather a regular pair, making it considerably easier to work with.

In all the examples uses of `Σ we have seen so far, the *structure* of the description returned by the function stored as its second argument remained the same each time, with only the indices stored for recursive positins depending on the choice of first element. This implies that, if we choose the semantics of a recursive position $\mathit{Var}\ i$ to be the raw type a, the semantics of the description returned by the second element will remain constant, independent of the value chosen for the first element. This means that there is no dependency between the two elements of the pair, enabling us to interpret `Σ as a regular pair.

The question that remains is what description to supply for the second element of the pair. Since the values stored in recursive positions have no effect on the semantics of a description, neither has their types. This means that two descriptions with *different* index types may map to the same interpretation, as long as their *structure* is the same. Based on this observation, we use a description of type $\mathit{IDesc}\ a\ (s \to i)$ to describe the second element of a `Σ, where $i$ is the index type of the description we are constructing, and $s$ the type stored in the first element of a `Σ. This amounts to the following constructor of the IDesc type:

$$\Sigma :: Proxy\ s \to IDesc\ a\ (s \to i) \to IDesc\ a\ i$$

It is important to note that there exist a mapping from $IDesc\ a\ (s \to i)$ to $s \to IDesc\ a\ i$, such that the interpretation 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 6.2.3, without the generalized coproduct $((:+>))$ and Σ. Here E is a type with no constructors, representing the empty type.

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)
```

For Σ, we need a type synonym to map a proxy to the type it carries:

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

Given $UnProxy$, its interpretation is now simply a pair with the type carried in the proxy, and the interpretation of the second element:

```
type instance Interpret (Σ p fd) = (UnProxy p, Interpret fd)
```

In case of the generalized coproduct, we would like to map a vector of descriptions to a type representing a choice between the interpretation of any of the descriptions carried in said 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:

```
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))
```

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 interpretation and the empty type.

6.3 Deriving Generators for Indexed Descriptions in Haskell

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

6.3.1 The generator type

We again use an abstract generator type, representing a deep embedding of the combinators we use. The definition is shown in listing 6.3.1, and is mostly the same as the definition we used in Agda. We choose to not have separate types for indexed and non-indexed generators, representing non-indexed types as types indexed by the unit type.

Listing 6.3: The *Gen* type in Haskell

```
data Gen i a t 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
```

We define the following wrapper to allow generators to be an instance of *Functor*, *Applicative*, *Monad* and *Alternative*:

```
newtype G i t a = G (Gen i a t)
```

This allows us to use the functions associated with these typeclasses to define genrators. For example:

```
bool :: G () Bool Bool
bool = pure True <|> pure False
```

6.3.2 A singleton instance for descriptions

Since our goal is eventually to define a function $idesc\_gen :: Sing\ d \to G\ i\ a\ (Interpret\ d)$, we require an appropriate singleton instance for the *IDesc* type. We again start by defining this instance for *One*, *Empty*, *Var* and $:*:$, shown in listing 6.3.2

Listing 6.4: Singleton instance for the *IDesc* type

```
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)
```

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:

$$(:+>\sim) :: SNat2\ n \to SVec\ xs \to SingIDesc\ (n :+> xs)$$

The singleton definition for the Σ constructor (listing 6.3.2) 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 $promote :: a \to Promoted\ a\ Sing$, where $Promoted$ is defined as follows:

$$\textbf{data}\ Promoted\ (a :: *)\ (f :: a \to *) = forall\ (x :: a) \circ Promoted\ (f\ x)$$

Storing singleton values, but hiding the the index.

Listing 6.5: Singleton instance for the Σ constructor

$$\begin{aligned} SSigma :: \ & Promote\ s \Rightarrow SingIDesc\ d \\ & \to G\ ()\ s\ s \\ & \to (forall\ s' \circ Sing\ s' \to Interpret\ d :\sim: Interpret\ (Expand\ d\ s')) \\ & \to SingIDesc\ (\Sigma\ (Proxy :: Proxy\ s)\ d) \end{aligned}$$

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 se when considering some examples, it is often more convenient to explicitly supply the generator to be used. 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 6.3.2.

Similarly, we use two mutually recursive functions to describe expansion for singleton descriptions (listing 6.3.2)

6.3.3 Deriving generators

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:

$$idesc\_gen :: forall\ (d :: IDesc\ a\ i) \circ (Singleton\ i) \Rightarrow SingIDesc\ d \to G\ i\ a\ (Interpret\ d)$$

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

$$\begin{aligned} & idesc\_gen\ SOne && = pure\ () \\ & idesc\_gen\ SEmpty && = empty \\ & idesc\_gen\ (SVar\ v) && = G\ (\mu\ v) \\ & idesc\_gen\ (dl :*: \sim dr) && = (,) <\$> idesc\_gen\ dl <*> idesc\_gen\ dr \end{aligned}$$

Listing 6.6: Description expansion

$$
\begin{array}{l}
\textbf{type family } \mathit{VExpand}\ (\mathit{sn} :: \mathit{SNat}\ n) \\
\quad (\mathit{xs} :: \mathit{Vec}\ (\mathit{IDesc}\ a\ (s \to i))\ n)\ (x :: s) :: \mathit{Vec}\ (\mathit{IDesc}\ a\ i)\ n \\
\textbf{type instance } \mathit{VExpand}\ \mathit{SZero}\ \mathit{VNil}\ s \qquad\qquad = \mathit{VNil} \\
\textbf{type instance } \mathit{VExpand}\ (\mathit{SSuc}\ \mathit{sn})\ (x ::: \mathit{xs})\ s = \mathit{Expand}\ x\ s ::: \mathit{VExpand}\ \mathit{sn}\ \mathit{xs}\ s \\
\textbf{type family } \mathit{Expand}\ (d :: \mathit{IDesc}\ a\ (s \to i))\ (x :: s) :: \mathit{IDesc}\ a\ i \\
\textbf{type instance } \mathit{Expand}\ \mathit{One} \qquad\qquad s = \mathit{One} \\
\textbf{type instance } \mathit{Expand}\ \mathit{Empty} \qquad\quad s = \mathit{Empty} \\
\textbf{type instance } \mathit{Expand}\ (\mathit{Var}\ i) \qquad\ \ s = \mathit{Var}\ (i\ s) \\
\textbf{type instance } \mathit{Expand}\ (\mathit{dl} :*: \mathit{dr})\ \ s = (\mathit{Expand}\ \mathit{dl}\ s) :*: (\mathit{Expand}\ \mathit{dr}\ s) \\
\textbf{type instance } \mathit{Expand}\ (\mathit{sn} :+> \mathit{xs})\ s = \mathit{sn} :+> \mathit{VExpand}\ \mathit{sn}\ \mathit{xs}\ s \\
\textbf{type instance } \mathit{Expand}\ (\Sigma\ p\ d) \qquad s = \Sigma\ p\ (\mathit{Expand}\ d\ s)
\end{array}
$$

Listing 6.7: Description expansion for singletons

$$
\begin{array}{l}
\mathit{vexpand} :: (\mathit{Singleton}\ s) \Rightarrow \mathit{Sing}\ \mathit{sn} \to \mathit{Sing}\ \mathit{xs} \to \mathit{Sing}\ s' \to \mathit{SVec}\ (\mathit{VExpand}\ \mathit{sn}\ \mathit{xs}\ s') \\
\mathit{vexpand}\ \mathit{SZero2}\ \mathit{SVNil} \qquad\qquad s = \mathit{SVNil} \\
\mathit{vexpand}\ (\mathit{SSuc2}\ \mathit{sn})\ (x ::: {\sim}\mathit{xs})\ s = \mathit{expand}\ x\ s ::: {\sim}\mathit{vexpand}\ \mathit{sn}\ \mathit{xs}\ s \\
\mathit{expand} :: (\mathit{Singleton}\ s) \Rightarrow \mathit{Sing}\ d \to \mathit{Sing}\ s' \to \mathit{Sing}\ (\mathit{Expand}\ d\ s') \\
\mathit{expand}\ \mathit{SOne} \qquad\qquad \mathit{sv} = \mathit{SOne} \\
\mathit{expand}\ \mathit{SEmpty} \qquad\quad \mathit{sv} = \mathit{SEmpty} \\
\mathit{expand}\ (\mathit{SVar}\ \mathit{ix}) \qquad \mathit{sv} = \mathit{SVar}\ (\mathit{ix}\ \mathit{sv}) \\
\mathit{expand}\ (\mathit{dl} :*: {\sim}\mathit{dr})\ \ \mathit{sv} = \mathit{expand}\ \mathit{dl}\ \mathit{sv} :*: {\sim}\mathit{expand}\ \mathit{dr}\ \mathit{sv} \\
\mathit{expand}\ (\mathit{sn} :+> {\sim}\mathit{xs})\ \mathit{sv} = \mathit{sn} :+> {\sim}\mathit{vexpand}\ \mathit{sn}\ \mathit{xs}\ \mathit{sv}
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{idesc\_gen}\ (\mathit{SZero2} :+> {\sim}\mathit{SVNil}) \qquad\qquad\qquad\quad = \mathit{empty} \\
\mathit{idesc\_gen}\ (\mathit{SSuc2}\ \mathit{SZero2} :+> {\sim}(d ::: {\sim}\mathit{SVNil})) = \mathit{idesc\_gen}\ d \\
\mathit{idesc\_gen}\ (\mathit{SSuc2}\ (\mathit{SSuc2}\ n) :+> {\sim}(d ::: {\sim}\mathit{ds})) \ \ = \\
\quad \mathit{Left} <\$> \mathit{idesc\_gen}\ d \\
\quad <|> \mathit{Right} <\$> \mathit{idesc\_gen}\ (\mathit{SSuc2}\ n :+> {\sim}\mathit{ds})
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{idesc\_gen}\ (\mathit{SSigma}\ \mathit{desc}\ \mathit{gen}\ \mathit{eq}) = \textbf{do} \\
\quad x \leftarrow G\ (\mathit{Call}\ (\lambda() \to \mathit{unG}\ \mathit{gen})\ ())
\end{array}
$$

$$\begin{aligned}
&\textbf{let } px = promote\ x \\
&\textbf{case } px \textbf{ of} \\
&\quad Promoted\ x' \rightarrow \textbf{do} \\
&\qquad p \leftarrow idesc_gen\ (expand\ desc\ x') \\
&\qquad pure\ (x, eqConv\ (eq\ x')\ p)
\end{aligned}$$

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 $Sing\ x$. 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 $Interpret\ (Expand\ desc\ s)$. We use this generator to get a value p of this type, which we can cast to a value of type $Interpret\ desc$ using the stored equality proof.

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

6.4 Examples

We consider two small 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 $run :: (i \rightarrow Gen\ i\ a\ a) \rightarrow i \rightarrow Int \rightarrow [a]$ is in scope, interpreting abstract generators as an exhaustive enumeration up to a certain depth.

6.4.1 Finite sets

We assume the definition of Fin given in listing ??. 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.

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

$$\begin{aligned}
&\textbf{type family } FinDesc\ (n :: Nat) :: IDesc\ Nat\ Nat \\
&\textbf{type instance } FinDesc\ Zero \quad = Empty \\
&\textbf{type instance } FinDesc\ (Suc\ n) = (SSuc\ (SSuc\ SZero)) :+> (One ::: Var\ n ::: VNil)
\end{aligned}$$

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:

$$\begin{aligned}
&finSDesc :: Sing\ n \rightarrow SingIDesc\ (FinDesc\ n) \\
&finSDesc\ SZero = SEmpty \\
&finSDesc\ (SSuc\ n) = SSuc2\ (SSuc2\ SZero2) :+> \sim(SOne ::: \sim SVar\ n ::: \sim SVNil)
\end{aligned}$$

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:

$$\begin{aligned}
&toFin :: Sing\ n \rightarrow Interpret\ (FinDesc\ n) \rightarrow Nat \\
&toFin\ (SSuc\ sn)\ (Left\ ()) = Zero \\
&toFin\ (SSuc\ sn)\ (Right\ n) = Suc\ n
\end{aligned}$$

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

$$
\begin{aligned}
&genFin :: forall\ (n :: Nat) \circ Nat \rightarrow G\ Nat\ Nat\ Nat \\
&genFin\ n = \\
&\quad \textbf{case}\ promote\ n\ \textbf{of} \\
&\qquad (Promoted\ sn) \rightarrow genDesc\ sn
\end{aligned}
$$

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-formed involved. We use the description shown in listing 5.1.2 as a basis, based on the representation of well-typed terms used in Phil Wadler and Wen Kokke's PLFA [27].

We model types terms and contexts with the following datatypes:

$$
\begin{aligned}
&\textbf{data}\ Type = Type : -> Type \mid T \\
&\textbf{data}\ Term = TVar\ Nat \mid TAbs\ Term \mid TApp\ Term\ Term \\
&\textbf{type}\ Ctx = [\,Type\,]
\end{aligned}
$$

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

$$
\textbf{data}\ CtxPos = Here \mid There\ CtxPos
$$

Next, we define a generator for context positions:

$$
\begin{aligned}
&genElem :: Ctx \rightarrow Type \rightarrow G\ ()\ CtxPos\ CtxPos \\
&genElem\ [\,]\ \_ = empty \\
&genElem\ (t : ts)\ t' = (\textbf{if}\ t \equiv t'\ \textbf{then}\ pure\ Here\ \textbf{else}\ empty) <|> (There <\$> genElem\ ts\ t')
\end{aligned}
$$

Here, *genElem* takes a context and a type, and returns all positions at which that type occurs in the context. Now that we have 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 6.4.2. This is a direct translation of the description shown in chapter 7 5.1.2. Since 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 6.4.2). 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.

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

$$
\begin{aligned}
&toTerm :: Sing\ i \rightarrow Interpret\ (SLTCDesc\ i) \rightarrow Term \\
&toTerm\ (SPair\ \_\ ST)\ (Left\ (n, ())) = TVar\ (toNat\ n)
\end{aligned}
$$

Listing 6.8: 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)
```

Listing 6.9: 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)
```

```
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
```

We now have everything needed in pace to start generating well-typed terms. We do this again by promoting the supplied index, and calling *genDesc* with this value:

```
termGen :: (Ctx, Type) → G (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 $check :: Ctx \to Type \to Term \to Bool$ 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
```

7 Discussion

A
Datatype Definitions

A.1 Natural numbers

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

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

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

A.2 Finite Sets

Listing A.2: Definition of finite sets in Agda

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

A.3 Vectors

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

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

A.4 Simple Types

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

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

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

A.5 Contexts

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

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

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

A.6 Raw Lambda Terms

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

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

....

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

A.7 Lists

Listing A.7: Definition lists and Agda

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

A.8 Well-scoped Lambda Terms

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

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

Code listings

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

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

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

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

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

[7] 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.

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

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

[10] 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.

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

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

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

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

[15] 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.

[16] 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.

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

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

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

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

[27] 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.

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

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