University of Applied Sciences and Arts Northwestern Switzerland FHNW

Master of Science in Engineering

Projekt 7

Autor	Tobias Wyss, tobias.wyss@students.fhnw.ch
Supervisor	Prof. Dierk König
Start	20.09.2024
Abgabe	07.02.2025

Abstract

Damit Entwickler einen Typ in Property-based Testing-Frameworks verwenden können, müssen sie unabhängige Implementationen für das Generieren und Vereinfachen (Shrinking) für die Werte dieses Typs bereitstellen. Da beide Algorithmen oft Logik teilen, führt dies zu Codeduplikaten und erhöht somit das Fehlerpotenzial. Diese Arbeit beschreibt zwei Ansätze, um dieses Problem zu lösen: Statt einzelner Werte erzeugt Integrated-Shrinking Bäume von Werten, wobei jede Ebene im Baum die Shrinks des Werts im übergeordneten Knoten enthält. Die strukturierte Generierung verzichtet hingegen vollständig auf Shrinking, indem sie komplexe Werte aus einfachen Startwerten und wiederholter Anwendung geeigneter Transformationsfunktionen ableitet.

Inhaltsverzeichnis

1 Einleitung

1.1 Propertybased Testing

Das verbreitete Testing-Framework „QuickCheck“ generiert automatisiert Testdaten und überpüft definierte Eigenschaften des Programms auf ihre Gültigkeiten. Dadurch kann QuickCheck oftmals nicht bedachte Edgecases finden, in denen eine Eigenschaft nicht hält. Dieses Verfahren nennt man „Propertybased Testing“.

Listing 1 zeigt eine Property für binäre Suchbäume, die besagt, dass das Resultat von zwei verschiedenen Operationen identisch sein muss, unabhängig davon, ob zuerst ein Element aus einem Baum gelöscht wird und dieser anschliessend in eine Liste umgewandelt wird, oder ob der Baum zuerst in eine Liste umgewandelt und dann ein Element daraus gelöscht wird. John Hughes nennt Propertys dieser Art „Model based Properties“, da sie ein Modell (in diesem Falle die Liste) verwenden, um die Invariante zu definieren. [1]

```
1  p_deleteModel :: Int -> Tree -> Property
2  p_deleteModel k t = property $
3    bstToList (delete k t)
4    ==
5    (L.sort $ deleteKey k $ bstToList t)
```

Listing 1: Eine Invariante um die Funktion `delete` auf `Tree`s zu testen

1.2 Generierte Werte shrinken

Da QuickCheck die Werte anhand eines Seeds zufällig generiert, kommt es oft vor, dass eine Eigenschaft für einen komplexen Wert nicht hält. Da es schwierig ist, einen komplexen Wert zu interpretieren, versucht QuickCheck diesen Wert anhand definierter Regeln zu vereinfachen. Der Wert wird solange vereinfacht, bis ein Wert gefunden wird, für den die Invariante wieder hält. Der Wert davor ist das Resultat dieses Prozesses, den man „Shrinking“ nennt.

1.2.1 Shrinking Regeln definieren

QuickCheck erlaubt solche Regeln pro Typ zu erstellen und stellt für diesen Zweck eine Typklasse `Arbitrary` zur Verfügung. Für einen einfachen Datentyp `newtype`

`Nat = Nat Int`, welcher natürliche Zahlen repräsentiert, definiert Listing 2 eine Instanz dazu:

```haskell
instance Arbitrary Nat where
  arbitrary      = Nat <$> (arbitrary `suchThat` (>= 0))
  shrink (Nat a) = [Nat s | s <- shrink a, s >= 0]
```

Listing 2: Die `Arbitrary`-Instanz von `Nat`

Da `Int` bereits eine `Arbitrary`-Instanz hat, können wir auf dieser Basis die Instanz zu `Nat` definieren: Wir generieren Werte des Typs `Int` und wenden den Newtype Wrapper an. Für das Shrinking greifen wir ebenfalls auf die `Arbitrary`-Instanz von `Int` zurück. Mit der Funktion `\x -> x >= 0` stellen wir bei beiden Funktionen sicher, dass nur Werte grösser gleich `0` generiert werden.

Pro Typ kann es maximal ein `Arbitrary` und somit auch nur einen einzigen Shrinking Prozess geben. Um für einen Typen verschiedene `Arbitrary`s zur Verfügung zu stellen, verwendet man in der Regel einen `newtype` Wrapper, analog wie in diesem Beispiel. Aufgrund des Bezugs, den der Shrinking Prozess zum Typ hat, wird er „Typbasiertes Shrinking“ genannt.

1.3 Die Probleme von typbasiertem Shrinking

Typbasiertes Shrinking ist zwar einfach, aber nicht optimal. Die Nachteile sieht man bereits am obigen Beispiel: man muss sowohl beim Generieren als auch beim Shrinken der Werte ungültige Werte entfernen - bei beiden Funktionen müssen wir negative Zahlen explizit ausschliessen. Das führt schnell zu Fehlern, vor allem wenn das Shrinken etwas unübersichtlicher wird. Listing 3 zeigt anhand eines binären Suchbaums ein etwas komplexeres Beispiel:

```haskell
data BST k v = Leaf | Branch (BST k v) k v (BST k v)

instance (Ord k, Arbitrary k, Arbitrary v) => Arbitrary (BST k v) where
    arbitrary = do
      kvs <- arbitrary
      return $
        foldr (uncurry insert) nil kvs
    shrink Leaf = []
    shrink (Branch l k v r) =
      [Leaf] ++
      [l, r] ++
      [Branch l k' v r | k' <- shrink k] ++
      [Branch l' k v r | l' <- shrink l] ++
      [Branch l k v r' | r' <- shrink r]
```

Listing 3: BST mit einer zugehörigen `Arbitray`-Instanz

Die Funktion `arbitrary` generiert eine Liste von Key-Value-Paaren, die mittels `insert` in einen leeren Suchbaum eingefügt werden. `shrink` kommt zu einfacheren Bäumen indem es

1. den Baum direkt in einen leeren Baum umwandelt,
2. den Root Knoten entfernt und die beiden Teilbäume links und rechts zurückgibt
3. den Key verkleinert
4. oder den jetzigen Knoten unberührt lässt und einen der beiden Teilbäumen verkleinert.

Blätter können nicht weiter verkleinert werden.

Wenn `insert` korrekt implementiert ist, ist `arbitrary` ebenfalls korrekt. Was aber ist mit der Funktion `shrink`? Sie ist im obigen Beispiel nicht korrekt implementiert, da wir den Key `k` verkleinern. Dies kann dazu führen, dass die Invariante eines binären Suchbaumes verletzt wird, welche lautet: alle Keys im linken Teilbaum sind kleiner oder gleich dem Key in der Wurzel und alle Keys im rechten Teilbaum sind grösser als der Key in der Wurzel.

Somit müssen wir also noch überprüfen, dass beim Shrinken des Keys diese Invariante nicht verletzt wird. Listing 4 definiert eine überarbeite Version der Funktion `shrink`. Sie verwendet die Funktion `valid`, die für jeden Binären Suchbaum den Wert `True` zurückgibt, wenn er das vorherig erwähnte Prädikat erfüllt.

```
1       shrink Leaf = []
2       shrink (Branch l k v r) =
3         [Leaf] ++
4         [l, r] ++
5         (filter valid $
6           [Branch l k' v r | k' <- shrink k] ++
7           [Branch l' k v r | l' <- shrink l] ++
8           [Branch l k v r' | r' <- shrink r]
9         )
```

Listing 4: Die korrigierte `Arbitrary`-Instanz von `BST`

Die Funktion ist analag zur ersten Version. Allerdings muss an der richtigen Stelle überprüft werden, ob die geshrinkten Bäume nach wie vor valide sind.

Beim automatisierten Testen von Sourcecode möchte man sich sicher sein, dass die Testcases korrekt implementiert sind. Deshalb sollten sie so wenig Raum für Fehler wie möglich bieten.

1.4 Ziele dieser Arbeit

Diese Arbeit untersucht zwei alternative Arten zur Generierung von Testdaten. Beide beheben die im Abschnitt 1.3 *„Die Probleme von typbasiertem Shrinking“* beschriebenen Nachteile und bieten zusätzlich neue Vorteile, ohne dabei den Charakter von Propertybased Testing zu verlieren.

Kapitel 3 *„Integrated-Shrinking“* erklärt einen alternativen Shrinking Ansatz und Kapitel 4 *„Generatoren“* führt aus, wie man zufällig generierte Testdaten mit strukturiert generierten ersetzen kann, so dass kein Shrinking mehr nötig ist.

2 Projektumgebung

2.1 Frege

QuickCheck ist ein Testing-Framework, das John Hughes für Haskell geschrieben hat. Mittlerweile gibt es ähnliche Frameworks für verschiedenste Programmiersprachen, unter anderem auch für die funktionale Programmiersprache „Frege“. Frege bringt rein funktionale Programmierung auf die JVM und erschliesst so neue Konzepte für Java-Entwickler. [2]

Dieses Projekt verwendet Frege für die Implementation aller beschriebenen Konzepte und auch für sämtliche Code-Beispiele dieses Berichts. In den meisten Fällen kann man die Code-Beispiele ohne Änderungen in Haskell ausführen.

2.2 Ähnliche Arbeiten

2.2.1 Propertybased Testing in Java

Jqwik ist ein Propertybased Testing-Framework, basierend auf der JUnit 5 Plattform. Somit kann es mit Java verwendet werden. [3] Im Gegensatz zu QuickCheck, verwendet Jqwik Integrated-Shrinking. Johannes Link, der Autor von Jqwik, schrieb eine Blogpost-Serie, die einen einfachen Einstieg in Propertybased Testing erlaubt. [4] Der gut verständliche Sourcecode von Jqwik zeigt auf, wie Integrated-Shrinking grundsätzlich implementiert werden kann und bildete so eine wichtige Grundlage für diese Arbeit.

2.2.2 Integrated-Shrinking in Haskell

Als Alternative zu QuickCheck gibt es das Haskell Testing-Framework „Hedgehog“. [5] Hedgehog verwendet Integrated anstelle von Typebased-Shrinking. Basierend auf den Ideen von Hedgehog erklärt Edsko de Vries in seinem Blogpost „Integrated versus Manual Shrinking“, wie Integrated Shrinking in Haskell implementiert werden kann und wie es sich von Typebased Shrinking unterscheidet. [6] Dieser Blogpost liefert viele Ansätze, die mehrere Konzepte dieser Arbeit inspirierten. Somit ist dieser Post ebenfalls eine wichtige Grundlage für diese Projektarbeit.

3 Integrated-Shrinking

Die Konzepte Typebased und Intgegrated Shrinking sind ähnlich, allerdings wird bei Integrated-Shrinking der Shrinkingprozess in den Wertgenerierungsprozess „integriert“. [7]

Durch diese Integration wird das im Abschnitt 1.3 *„Die Probleme von typbasiertem Shrinking“* beschriebene Problem gelöst. So lassen sich bereits bestehende Instanzen auch einfacher zu neuen Instanzen kombinieren. Integrated-Shrinking lässt also einfachere Code-Wiederverwendung zu.

3.1 Die Typklasse `IntegratedArbitrary`

Integrated-Shrinking kann ebenfalls als Typklasse modelliert werden. Im Gegensatz zu der im Abschnitt 1.2.1 *„Shrinking Regeln definieren“* vorgestellten Klasse `Arbitrary`, bietet `IntegratedArbitrary` nur eine einzelne Methode. Listing 5 definiert die Implementation:

```
1 data Shrinkable a = Node { root :: a, shrinks :: [Shrinkable a] }
2
3 class IntegratedArbitrary where
4   gen :: StdGen -> Shrinkable a
```

Listing 5: Die `IntegratedArbitrary`-Instanz von `Nat`

Die Funktion `gen` nimmt einen Zufallszahlgenerator und gibt ein `Shrinkable a` zurück. `Shrinkable` ist eine Implementation eines Rosetrees - ein Tree mit beliebig vielen Nachfolgern. Die Wurzel des Trees ist dabei der generierte Testwert, die direkten Nachfolger sind die geshrinkten Werte, die sich durch Applizieren der Shrinkregeln auf die Wurzel ergeben. Jeder dieser Werte wird nun durch rekursives Anwenden dieser Regeln weiter geshrinkt. So entsteht ein potentiell unendlich tiefer Baum.

3.2 `IntegratedArbitrary` implementieren

Um Instanzen von `IntegratedArbitrary` zu definieren, verwenden wir eine Hilfsfunktion `shrinkTree`. Listing 6 definiert diese:

```haskell
shrinkTree :: forall a. Eq a => (a -> [a]) -> a -> Shrinkable a
shrinkTree shrink root = Node root shrinkRoot
  where
    shrinkRoot :: (Eq a) => [Shrinkable a]
    shrinkRoot = map (removeDuplicates . shrinkTree shrink) (shrink root)
```

Listing 6: `shrinkTree` wendet eine Funktion rekursiv auf einen gebenen Wert an

Die Funkion nimmt einen Startwert `root` und eine Funktion `shrink`, die Variationen des Startwerts erstellt. Diese Funktion wird nun auf `root` angewendet. So entsteht die erste Ebene des Trees. Durch rekursives Anwenden der Funktion entsteht der ganze Baum.

3.2.1 `IntegratedArbitrary` von `Int`

Listing 7 zeigt am Beispiel der `IntegratedArbitrary`-Instanz von `Int` die Verwendung der Funktion `shrinkTree`:

```haskell
import System.Random

instance IntegratedArbitrary Int where
  gen :: StdGen -> Shrinkable Int
  gen seed =
    let root = randomValue seed in
      shrinkTree shrinkInt root
    where
      shrinkInt a | a > 0     = [a `quot` 2, a - 1]
                  | a < 0     = [a `quot` 2, a + 1]
                  | otherwise = []

      randomValue :: Random a => StdGen -> a
      randomValue seed = fst (Random.random seed)
```

Listing 7: Die `IntegratedArbitrary`-Instanz von `Int`

Wichtig dabei ist vor allem die lokale Definition `shrinkInt`. Sie nähert einen gegebenen Wert dem Wert `0` an. Der Wert `0` kann nicht weiter vereinfacht werden.

Aus dieser Definition ergeben sich für die Werte `4` und `-4` folgende Shrinktrees:

```
4                       -4
├─ 2                    ├─ -2
│  └─ 1                 │  └─ -1
│     └─ 0              │     └─ 0
└─ 3                    └─ -3
   ├─ 1                    ├─ -1
   │  └─ 0                 │  └─ 0
   └─ 2                    └─ -2
      └─ 1                    └─ -1
         └─ 0                    └─ 0
```

Listing 8: Die Shrinktrees für `4` respektive `-4`

Mit analogem Vorgehen können auch Instanzen für andere einfache Datentypen wie `Bool` erstellt werden.

3.3 Kombinieren von Instanzen

3.3.1 Instanzen für Tuples und Records

Listing 9 zeigt auf Zeile 11 am Beispiel von Tuples, wie mit Hilfe der `Applicative`-Instanz von `Shrinkable` das Kombinieren von `IntegratedArbitrary`s einfach möglich ist:

```
1  import System.Random
2
3  instance (IntegratedArbitrary a, IntegratedArbitrary b) =>
4             IntegratedArbitrary (a,b) where
5    gen :: (IntegratedArbitrary a, IntegratedArbitrary b) => StdGen ->
6             Shrinkable (a,b)
7    gen seed =
8      let
9        (s1, s2) = Random.split seed
10     in
11      (,) <$> gen s1 <*> gen s2
```

Listing 9: Die `IntegratedArbitrary`-Instanz für Tuples

Das gleiche Muster lässt sich für Records verwenden. Die Erstellung von `IntegratedArbitrary`s für Records ist besonders interessant, da viele Projekte für verschiedenste Datenstrukturen Records verwenden: Listing 10 definiert den Record `Person` und die zugehörige Arbitrary Instanz:

```haskell
data Person = Person { age :: Int, name :: String }

instance Arbitrary Person where
  gen :: StdGen -> Shrinkable Person
  gen seed =
    let
      (s1, s2) = Random.split seed
    in
     Person <$> gen s1 <*> gen s2
```

Listing 10: Die `IntegratedArbitrary`-Instanz für den Record `Person`

Mit `Random.split` können beliebig viele Seeds generiert werden und somit ist es einfach, Instanzen für Records mit beliebig vielen Feldern zu erstellen.

3.3.2 `Nat` revisited

Die `Arbitrary`-Instanz von `Nat`, die Listing 2 zeigt, lässt sich leicht als `IntegratedArbitrary` definieren. `Nat` ist definiert als `newtype Nat = Nat Int`. Listing 11 zeigt die neue Implementation:

```haskell
instance IntegratedArbitrary Nat where
  gen = fmap Nat . (generateSatisfying pos)
    where pos = (>= 0)
```

Listing 11: Die `IntegratedArbitrary`-Instanz für `Nat`

`generateSatisfying` ist eine Hilfsfunktion mit der Signatur `IntegratedArbitrary a => (a -> Bool) -> StdGen -> Shrinkable a`. Sie nimmt ein Prädikat und einen Generator und gibt ein `Shrinkable` zurück. Im Hintergrund verwendet sie die `IntegratedArbitrary`-Instanz von `a`. Allerdings werden nur Elemente zurückgegeben, die das übergebene Prädikat erfüllen. `fmap` appliziert den Konstruktor von `Nat` auf die Elemente des generierten Trees.

Anders als in Listing 2 erlaubt es uns `IntegratedArbitrary`, die generierten Werte zusammen mit dem Shrinktree zu filtern. Somit verwenden wir das Prädikat `(>= 0)` nur noch an einer Stelle. Das verringert die Wahrscheinlichkeit, Fehler zu machen.

3.4 Integrated-Shrinking auf Listen

Listing 12 leitet aus der `IntegratedArbitrary`-Instanz für Tuples eine Instanz für Listen ab. Bei der Liste kommt zusätzlich eine Länge hinzu. Um eine Länge zu generieren, verwenden wir auf Zeile 8 die Instanz von `Nat`:

```
1  import System.Random
2
3  instance IntegratedArbitrary a => IntegratedArbitrary [a] where
4    gen :: IntegratedArbitrary a => StdGen -> Shrinkable [a]
5    gen seed =
6      let
7        (s1, s2)  = Random.split seed
8        (Nat len) = Shrinkable.root $ gen s1
9      in
10       genEls s2 len
11     where
12       genEls :: IntegratedArbitrary a => StdGen -> Int -> Shrinkable [a]
13       genEls seed amt
14                     | amt <= 0  = pure []
15                     | otherwise =
16                       let (s, s') = Random.split seed
17                       in (:) <$> gen s <*> genEls s' (amt - 1)
```

Listing 12: Eine problematische `IntegratedArbitrary`-Instanz für `List`

Analog zu Tuples und Records verwenden wir nun in der Funktion `genEls` die `Applicative`-Instanz und erhalten so den Shrinktree für die Liste.

Diese Implementation ist allerdings nicht korrekt. Listing 13 zeigt den generierten Shrinktree für die Liste `[0, 1, 1]` und somit das Problem dieser Implementation:

```
1  [0, 1, 1]
2  ├─ [0, 0, 1]
3  │  └─ [0, 0, 0]
4  ├─ [0, 0, 1]
5  │  └─ [0, 0, 0]
6  ├─ [0, 1, 0]
7  │  └─ [0, 0, 0]
8  └─ [0, 1, 0]
9     └─ [0, 0, 0]
```

Listing 13: Das Resultat einer geshrinkten Liste

Die Implementation shrinkt zwar die Elemente in der Liste korrekt, allerdings wird die Länge der Liste nicht mitgeshrinkt. Die geshrinkten Listen haben immer exakt drei Elemente.

Eine korrekte Implementation muss auch die Länge der Liste shrinken. Somit kann die `Applicative`-Instanz nicht dazu verwendet werden.

3.4.1 Korrektes Shrinking für Listen

Da wir nicht einfach bestehende `Shrinkable`s für die Liste kombinieren können, müssen wir eine spezifische Implementation bereitstellen, die die Elemente der Liste generiert und die geshrinkten Werte mit der geshrinkten Listenlänge kombiniert. Dabei müssen wir beachten, dass das Shrinking in die Generierung der Werte integriert ist, somit können wir diese beiden Prozesse nicht voneinander trennen. Damit wir die Werte der Liste später noch shrinken können, müssen wir deshalb beim Generieren der Werte deren Shrinktrees aufbewahren. Tun wir das nicht, haben wir später keine Möglichkeit mehr, die Werte selbst zu shrinken.

Dies führt zu einem Algorithmus, den Anbieter einer Datenstruktur schreiben müssen, um Integrated-Shrinking zu verwenden. Für eine beliebige Datenstruktur `Functor f => f a` sieht er so aus:

1. Generiere so viele `Shrinkable a`s wie benötigt und speichere sie in der Datenstruktur `f`, für die das Arbitrary erstellt werden soll
2. Definiere eine Shrink-Funktion, die:
   1. die Länge von `f` shrinkt
   2. jedes Element aus `f` shrinkt
3. Map `Shrinkable.root` über `f`, um ein gültiges Wurzel-Element zu erhalten.
4. Kombiniere die Wurzel und seine Shrinks zu einem Shrinktree.

Listing 14 implementiert diesen Algorithmus für die Liste:

```haskell
instance IntegratedArbitrary a => IntegratedArbitrary [a] where
  gen :: IntegratedArbitrary a => StdGen -> Shrinkable [a]
  gen seed =
    let
      (s1, s2)  = Random.split seed
      -- len is the amount of elements in the list we generate
      (Nat len) = Shrinkable.root $ gen s1
      elements  = genElements s2 len
    in
      shrinkList elements
    where
      -- Step 1: generate as many elements as needed
      genElements :: IntegratedArbitrary a => StdGen -> Int -> [Shrinkable a]
      genElements _ 0 = []
      genElements seed l = let (s1, s2) = Random.split seed in
          (gen s1) : genElements s2 (l - 1)

      -- Step 2: provide the shrinking function
      shrinkList :: [Shrinkable a] -> Shrinkable [a]
      shrinkList ts =
          -- Step 3: map `Shrinkable.root` over the elements
          -- Step 4: combine the root with the shrink tree
          Node (map Shrinkable.root ts) $
              -- Shrink the length of the list
              [shrinkList (as ++ cs)
                  | (as, _, cs) <- select [] ts]
              ++
              -- shrink the focused element, leave the others untouched
              [shrinkList (as ++ [b'] ++ cs)
                  | (as, b, cs)  <- select [] ts,
                    b'           <- Shrinkable.shrinks b]

      -- | moves each element of the *second* list into the focus
      select :: [a] -> [a] -> [([a], a, [a])]
      select _ [] = []
      select as (b:bs) = (as, b, bs) : select (as ++ [b]) (bs)
```

Listing 14: Korrektes Integrated-Shrinking für Listen

Dieses Kapitel zeigt, dass Integrated-Shrinking hervorragend für einfache Datentypen und Typen mit fixer Länge funktioniert. Allerdings ist es komplexer, Instanzen zu schreiben, die eine Abhängigkeit zu einer Länge oder anderen dynamischen Eigenschaften haben, da sich diese nicht mehr einfach über die `Applicative`-Instanz kombinieren lassen. Dies führt zu Implementationen von Integrated-Shrinking, die um einiges komplexer sind als die Implementationen von Typebased-Shrinking.

Man sieht also, dass Typebased-Shrinking Vorteile für den Anbieter einer Datenstruktur bietet - denn es lassen sich sehr einfach neue Instanzen bauen. Integrated-

Shrinking dagegen eignet sich gut für Verwender der Datenstrukturen, da man sich keine Gedanken über Shrinking machen muss.

3.5 Kombinieren von Typebased & Integrated-Shrinking

Um das beste aus beiden Welten zu nehmen, können wir diese Ansätze kombinieren. Die Idee liegt darin, dass der Anbieter eines Datentyps oder einer Datenstruktur ein typbasiertes `Arbitrary` anbietet. Daraus kann er ohne zusätzliche Logik (mittels Default-Implementationen) ein `IntegratedArbitrary` ableiten. Dieses lässt sich vom Testing-Framework und gleichzeitig als Basis für andere `IntegratedArbitrary` verwenden.

Abbildung 1 zeigt diese Kombination schematisch anhand der Liste. Diese bietet bereits eine `Arbitrary`-Instanz aus QuickCheck und nun zusätzlich auch eine Implementation von `IntegratedArbitrary`. Daraus können wir nun mit wenig Aufwand für den Typen `newtype SortedList a = SortedList [a]` eine `IntegratedArbitrary`-Instanz anbieten.

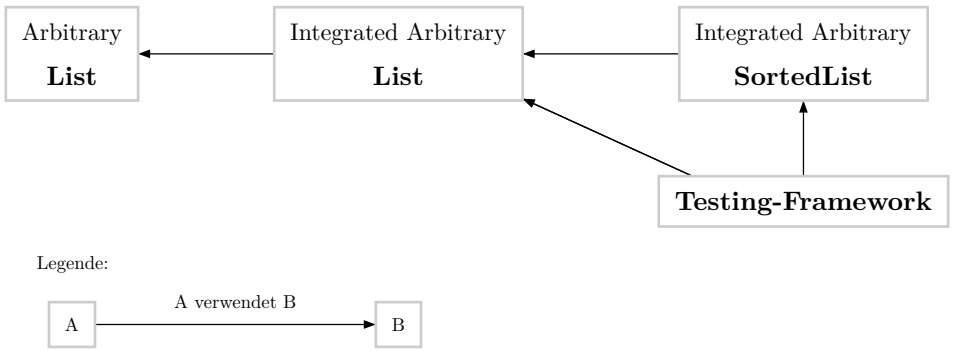

Abbildung 1: Kombination von `Arbitrary` mit `IntegratedArbitrary`

3.5.1 Implementation

Dazu fügen wir `IntegratedArbitrary a` eine Defaultimplementation für `gen` hinzu, falls es für `a` bereits eine `QuickCheck.Arbitrary`-Instanz gibt. Listing 15 definiert diese Implementation:

```haskell
import Test.QuickCheck as QC()

size = 1

class (QC.Arbitrary a, Eq a) => IntegratedArbitrary a where
  gen :: StdGen -> Shrinkable a
  gen seed =
    let
      root = Gen.unGen QC.Arbitrary.arbitrary seed size
    in
      shrinkTree QC.Arbitrary.shrink root
```

Listing 15: Defaultimplementierung für die `gen` Methode

In ihr verwenden wir die Methoden `Arbitrary.arbitrary`, um einen Wert zu generieren und `Arbitrary.shrink`, um einen gegebenen Wert zu shrinken. Listing 6 definierte die Funktion `shrinkTree` bereits , die aus einem Wert und einer Funktion ein `Shrinkable` baut.

3.5.1.1 Implementation für `List` und `SortedList`

Listing 16 zeigt nun die Implementation für `IntegratedArbitrary` des Beispiels `SortedList`. Zeile 1 definiert die Instanz für die Liste. Für sie muss keine Implementation geliefert werden, sondern sie kann direkt die Defaultimplementierung verwenden.

Die Implementation für `Sortedlist` ist auch einfach: wir verwenden wie zuvor die Funktion `generateSatisfying`, um generierte Daten zu filtern und mappen anschliessend den Konstuktor `SortedList` über jedes Element.

```haskell
instance (IntegratedArbitrary a, Eq a) => IntegratedArbitrary [a]

derive (Eq a) => Eq (SortedList a)

instance (Ord a, IntegratedArbitrary a) => IntegratedArbitrary (SortedList a) where
  gen :: (Ord a, IntegratedArbitrary a) => StdGen -> Shrinkable (SortedList a)
  gen = fmap SortedList . (generateSatisfying isSortedList)
    where
      isSortedList :: (Ord a) => [a] -> Bool
      isSortedList []  = True
      isSortedList [_] = True
      isSortedList (x:y:xs) = x <= y && isSortedList (y:xs)
```

Listing 16: Die Implementation von `IntegratedArbitrary` für `SortedList`

3.6 Diskussion

Dieses Kapitel zeigt, dass Integrated-Shrinking eine sinnvolle Ergänzung zu Type-based-Shrinking ist. Indem wir die Definitionen, die QuickCheck bereits bietet, wiederverwenden, können wir auf einfache Weise die Vorteile von Integrated-Shrinking ausnutzen. Zusätzlich ist es für Anbieter von Datenstrukturen sehr einfach, Integrated-Shrinking anzubieten, da es nur eine kleine Codeergänzung mit sich bringt. Durch die Natur der Typklassen können Verwender sogar selbständig beliebige Datenstrukturen kompatibel mit Integrated Shrinking machen und so von den Vorteilen profitieren.

4 Generatoren

4.1 Probleme von zufälligen Testdaten

Damit Propertybased Testing-Frameworks funktionieren, müssen sie Testdaten generieren. QuickCheck macht das automatisiert über einen Zufallsgenerator. Zufällige Testdaten haben mehrere Nachteile:

1. Bei jedem Testlauf wird mit anderen Daten getestet, das führt zu Inkonsistenzen zwischen einzelnen Testdurchläufen.
2. Es ist nicht garantiert, dass die zufälligen Testdaten auch sinnvoll sind für den Test, in dem sie verwendet werden.
3. Es ist schwierig, Äquivalenzklassen abzubilden.

Alle Gründe können dazu führen, dass ausgeführte Tests unter Umständen keine Aussagekraft haben und deshalb nutzlos sind. Hinzu kommt das Problem, dass Werte die eine Property nicht erfüllen, oft sehr komplex sind. Diese müssen mit dem in Abschnitt 1.2 *„Generierte Werte shrinken“* erklärten Prozess „Shrinking“ vereinfacht werden.

4.2 Strukturierte Generierung von Testdaten

Um diese Probleme zu beheben, können wir eine strukturierte Vorgehensweise beim Generieren von Testdaten verwenden. Anstelle von zufälligen Werten verwenden wir einen Wert und Funktionen, die diesen Wert strukturiert verändern. Den Wert nennen wir „Startwert“ und die Funktionen „Transformationsfunktionen“. Wenn wir die Transformationsfunktionen rekursiv auf alle generierten Werte anwenden, entsteht ein unendlich grosser Baum von Werten.

Listing 17 zeigt anhand des Startwerts `0` und den Transformationsfunktionen `(+1)`, `(\x -> x - 1)` und `(*3)` die ersten drei Ebenen des Baumes:

```
0
├─ 1
│  ├─ 2
│  ├─ 0
│  └─ 3
├─ -1
│  ├─ 0
│  ├─ -2
│  └─ -3
└─ 0
   ├─ 1
   ├─ -1
   └─ 0
```

Listing 17: Baum von strukturiert generierten Werten

Statt, wie bei zufälliger Generierung von Daten, komplexe Werte zu generieren und diese dann zu vereinfachen, kehren wir den Prozess um: wir nehmen einfache Werte und machen sie zunehmend komplexer.

Mit diesen sehr einfachen Transformationsfunktionen kommen wir zu einer Vielzahl an Werten. Dabei können wir bereits gesehene Werte bedenkenlos entfernen, da die Subtrees von zwei gleichen Werten immer gleich sind.

4.3 Implementation

4.3.1 Die Funktion `generate`

Listing 18 definiert das Herzstück des Generators, die Funktion `generate`. Sie produziert aus einer Liste von Transformationsfunktionen und einem Startwert einen `Tree` von Elementen. Für den `Tree` verwenden wir die Implementation aus der Frege Standard Library:

```
1 import Data.Tree
2
3 generate :: [a -> a] -> a -> Tree a
4 generate fs r = Node r $ subs
5   where
6     subs = map (generate fs) (map ($ r) fs)
```

Listing 18: Die Funktion `generate`

Die Funktion setzt den Startwert als Wurzel des Trees, und erstellt die direkten Nachfolger, indem sie alle Funktionen in `fs` auf ihn anwendet. Daraus entsteht

eine Liste von Werten. Auf jeden dieser Werte wendet sie nun wieder `generate fs` an, so bildet sich der Baum.

4.3.2 Bäume mittels BFS in eine Liste umwandeln

Um die generierten Werte in einem Testing-Framework zu verwenden, wandeln wir den Baum in eine Liste um. Bäume lassen sich gut mit rekursiven DFS-Algorithmen traversieren. Da der Baum, den die Funktion `generate` erstellt, unendlich tief ist, eignen sich DFS-Algorithmen in diesem Fall allerdings nicht. Zusätzlich sind Werte näher der Wurzel interessanter, da diese mehr Nachfolger haben als Werte tiefer im Baum. Um dem Benutzer zu überlassen, wie viele Tests das Testing-Framework ausführen soll, muss der Algorithmus zusätzlich lazy sein. Deshalb kommt hier ein queue-basierter Algorithmus zum Einsatz. [8]

4.3.3 Die Typklasse `Generator`

Damit wir leichter Propertys für verschiedene Typen schreiben können, führt Listing 19 eine neue Typklasse `Generator` ein:

```
1 type Label = String
2
3 class Eq a => Generator a where
4   seeds :: [a]
5   fs    :: [(Label, a -> a)]
```

Listing 19: Die Typklasse `Generator`

Dabei definiert `seeds` verschiedene Startwerte und `fs` die Transformationsfunktionen. Zusätzlich labeln wir die Transformationsfunktionen, um bei fehlschlagenden Tests Auskunft darüber zu geben, wie dieser Wert generiert wurde. Der `Eq`-Constraint ist wichtig, da wir doppelte Werte erkennen und entfernen möchten.

Listing 20 zeigt die Implementation von `Generator Int`:

```
1 instance Generator Int where
2   fs    :: [(Label, Int -> Int)]
3   fs    = [("inc", (+1)), ("dec", (\x -> x - 1)), ("*3", (*3))]
4   seeds = [-1,0,1]
```

Listing 20: Die Implementation von `Generator Int`

4.3.4 Generatoren in einem Testing-Framework verwenden

Mit diesen Bausteinen können wir ein Testing-Framework bauen. Dazu ändern wir die Funktion `generate` so ab, dass sie mit der Typklasse `Generator` umgehen kann und statt eines `Trees` eine `List` zurückgibt. Listing 21 zeigt die Signatur der neuen Funktion:

```
1 generate :: Generator a => a -> [TestCase a]
```

Listing 21: Die Signatur der Funktion `generate`

`generate` erwartet bewusst nach wie vor ein Seed als Parameter, damit der Verwender von aussen eines der potentiell mehreren Seeds verwenden kann. Der Rückgabetyp ist nicht mehr bloss `[a]` sondern nun `[TestCase a]`, welcher für jeden generierten Wert den Weg vom Seed aus in einer Liste mitführt.

Listing 22 definiert diesen Typ:

```
1 type Label = String
2
3 data TestCase a = TestCase { value :: a, seed :: a, path :: [(Label, a)] }
```

Listing 22: Der Typ `TestCase a`

Wenn ein Wert für eine Property nicht gilt, kann ein Testing-Framework nun die Labels der Operationen, die zu diesem Wert geführt haben, und die Zwischenwerte ausgeben:

```
1 TEST FAILED! With value: 5
2 5: 2 -> 6 -> 5 (seed -> *3 -> dec)
```

Listing 23: Beispielausgabe eines fehlschlagenden Tests

Mit diesen Mitteln kann ein Testing-Framework nun eine unendliche Menge an `TestCase`s, basierend auf einem Seed, generieren und mit ihnen Propertys beliebig oft mittels verschiedenen Werten überprüfen.

4.4 Kombinieren von Generatoren

4.4.1 Generatoren für Records und Tuples

Durch die Kombination von Generatoren lassen sich Instanzen für Tuples oder Records bauen.

Anhand des Records `data MyRecord a = MyRecord { a :: a, b :: Int }` zeigt Listing 24, wie wir für solche Datentypen `Generator`-Instanzen erstellen können:

```haskell
instance (Eq a, Generator a) => Generator (MyRecord a) where
  seeds =
    MyRecord <$> seeds <*> seeds

  fs = let
   changeA =
    [ ("a = " ++ lbl, \r -> r.{ a = fA (MyRecord.a r) }) | (lbl, fA) <- fs ]
   changeB =
    [ ("b = " ++ lbl, \r -> r.{ b = fB (MyRecord.b r) }) | (lbl, fB) <- fs ]
  in
   changeA ++ changeB
```

Listing 24: `Generator`-Instanz für einen Record

Die `seeds` kombiniert Listing 24 mittels der `Applicative`-Instanz der Liste. Daraus entstehen alle Kombinationen der Startwerte. Transformationsfunktionen erstellt es, indem es zuerst alle `fs` des ersten und anschliessend alle des zweiten Feldes anwendet. Mittels Konkatentation der beiden Listen erhalten wir eine Liste mit allen Permutationen aller Felder. Eine zusätzliche Transformation, die beide Werte gleichzeitig verändert, ist nicht nötig, da die Funktionen rekursiv auf alle Werte in jeder Ebene angewendet werden. Diese Werte entstehen deshalb automatisch in tieferen Ebenen.

4.4.2 `Generator` für Listen

Um Listen zu modifizieren, haben wir zwei Möglichkeiten:

1. Elemente zur Liste hinzufügen
2. Elemente der Liste verändern

Listing 25 zeigt wie die Instanz dazu aussieht:

```
1  instance (Generator a) => Generator [a] where
2    seeds :: Generator a => [[a]]
3    seeds = [[]]
4    fs :: Generator a => [(Label, [a] -> [a])]
5    fs    =
6      let
7        addSeed = [ ("addSeed", \l -> l ++ [seed]) | seed <- seeds ]
8        applyFsToEach =
9          [ (label lA idx,
10           \xs -> [ if i == idx then fA x else x | (x,i) <- indexed xs])
11         | (lA, fA) <- fs
12         , idx <- [0..lastPermutationIndex]]
13     in
14       addSeed ++ applyFsToEach
15     where
16       label lA idx = "apply_" ++ lA ++ "_to_elem_" ++ show idx
17       indexed xs = zip xs [0..]
18       lastPermutationIndex = 4
```

Listing 25: `Generator`-Instanz für Listen

Mittels `addSeed`, definiert auf Zeile 8, erzeugen wir neue Listen, indem wir alle verfügbaren Startwerte hinzufügen. `applyFsToEach`, definiert ab Zeile 9, permutiert die Elemente innerhalb der Liste.

Bei jeder Transformation möchten wir nur so wenig wie möglich verändern, damit wir möglichst genau bestimmen können, warum ein Test fehlschlägt. Um das zu erreichen, müssten wir aus der gegebenen Liste mit n Elementen und m -Transformationsfunktionen $n \cdot m$ neue Listen generieren. Allerdings lässt es der Typ von `Generator.fs` nicht zu, dass wir aus einem Element mehrere Elemente erstellen. Um das zu umgehen, müssen wir das gleiche Element also mehrmals verändern. Nur wie oftmals? Da wir zu diesem Zeitpunkt nicht wissen können wie viele Elemente die Liste hat, definiert Listing 25 auf den Zeilen 12 und 18, dass nur die ersten 5 Elemente transformiert werden. Wir können stattdessen auch nicht `[0..]` verwenden, da die Evaluierung der Liste in diesem Kontext nicht mehr terminiert.

4.4.3 `takeWhen`

Damit es einfacher wird Generatoren zu bauen, die eine spezifische Submenge von anderen Generatoren erzeugen, fügt Listing 26 der Typklasse `Generator` die Methode `takeWhen` hinzu:

```
1 class Eq a => Generator a where
2   seeds :: [a]
3   fs    :: [(Label, a -> a)]
4   takeWhen :: a -> Bool
5   takeWhen = const True
```

Listing 26: `takeWhen` als Erweiterung von `Generator`

Für viele Generatoren ist `takeWhen` nicht relevant. Wir geben der Methode deshalb die Defaultimplementation `const True`. Somit muss sie nirgends zwingend implementiert werden.

Listing 27 verwendet `takeWhen` und `Generator Int` und definiert so eine `Generator`-Instanz für `Nat`:

```
1 instance Generator Nat where
2   seeds              = fmap Nat seeds
3   fs                 = map (fmap (\f -> \(Nat x) -> Nat (f x))) fs
4   takeWhen (Nat v)   = v >= 0
```

Listing 27: Die `Generator`-Instanz für `Nat`

Es reicht also, bloss den newtype-Wrapper `Nat` auf die Startwerte und die Transformationsfunktionen anzuwenden und zusätzlich ein Prädikat zu definieren, um diese Instanz korrekt zu implementieren.

4.5 Diskussion

Mit Generatoren starten wir mit kleinen Werten, die mit Transformationsfunktionen komplexer werden. Shrinking ist so nicht mehr nötig, denn wenn ein Test scheitert, geschieht das bereits beim einfachsten Wert, für den die Property nicht hält.

Durch passende Transformationsfunktionen können wir den Lösungsraum gezielt durchschreiten und so Äquivalenzklassen sehr einfach abbilden. Dies führt zu einer soliden Testbasis, die sich nur ändert, wenn sich der Generator ändert. Somit sind die Tests immer reproduzierbar und frei von Seiteneffekten. Durch die Generierung von Daten ist trotzdem sichergestellt, dass nicht nur vorgesehene Werte überprüft werden.

4.5.1 Nachteile von Generatoren

Generatoren generieren schrittweise Werte. Da es kein Shrinking gibt, darf dies nicht in zu grossen Schritten geschehen. Das führt dazu, dass die Generatoren den Lösungsraum nur sehr langsam durchschreiten. Um viel Aussagekraft zu haben, braucht es deshalb eine grössere Anzahl an Testdaten, mit denen die Propertys überprüft werden. Das verlangsamt die Tests.

Zusätzlich führen die kleinen Schritte auch dazu, dass viele ähnliche Werte generiert werden, welche zum Teil keine weitere Aussagekraft haben als die Werte davor.

4.5.2 Offene Punkte

Wie Abschnitt 4.4.2 *„`Generator` für Listen“* anhand der Liste zeigt, ist der Typ von `Generator.gen` mit `a -> a` nicht gut gewählt - denn er lässt nicht zu, dass aus einem Element mehrere Elemente erzeugt werden. Sicherlich ist der Typ `a -> [a]` eine bessere Variante. Allerdings ist nicht klar, ob dieser Typ tatsächlich die optimale Lösung ist, oder ob sich eine andere Datenstruktur wie zum Beispiel ein Baum besser eignen würde. Welche Datenstruktur am besten geeignet ist, wird sich durch Hinzufügen von `Generator`-Instanzen von weiteren Typen zeigen. Bei der Wahl der Datenstruktur muss man allerdings beachten, dass das Schreiben von Generatoren nicht an Einfachheit einbüsst, um die Verwendung so simpel wie möglich zu gestalten.

Es existiert lange nicht für jeden Datentypen aus der Frege-Standardlibrary eine `Generator`-Instanz. Sowohl für diese als auch für die Generatoren, die bereits existieren, gilt es herauszufinden, was geeignete Startwerte und Transformations-

funktionen sind. Wählt man diese gut, können wir gängige Edgecases, wie zum Beispiel Grenzwerte, unserer Datentypen direkt abbilden. Somit werden diese kritischen Werte in jedem Property, das den `Generator` verwendet, automatisch überprüft. Momentan wählt das Testing-Framework des Generators einen Startwert aus und generiert nur Testdaten basierend auf diesem. Somit werden ganze Äquivalenzklassen, die durch die anderen Seeds abgedeckt werden, ignoriert. Das Testing-Framework sollte also die generierten Werte auf Basis der einzelnen Startwerte mischen, um so die ganze Breite eines Datentyps abzudecken.

5 Abschliessende Gedanken

Kapitel 4 *„Generatoren“* zeigt, dass es strukturierte Generierung möglich macht, Äquivalenzklassen und Edgecases abzubilden und somit sämtliche Propertys automatisch auf häufige Fehlerquellen überprüft werden. Das führt zweifellos zu aussagekräftigen Tests.

Trotz den Nachteilen von zufällig generierten Testdaten bieten diese auch einen positiven Aspekt: so können Werte gefunden werden, an die kein Entwickler gedacht hat. Zusätzlich zeigt schon allein die Verbreitung von QuickCheck, dass diese Art von Testdaten-Generierung gut funktioniert.

Um die Vorteile beider Ansätze zu nutzen, wäre es spannend, sie in einem zukünftigen Projekt in einer einzelnen Typklasse zu vereinen. Somit kann ein Testing-Framework eine Teilmenge an Tests zufällig und eine weitere Teilmenge strukturiert generieren.

Es ist allerings zu überprüfen, ob dieser Ansatz so tatsächlich eine weitere Verbesserung bringt oder ob er das Schreiben von Tests bloss erschwert.

Bevor wir aber an eine Kombination denken können, müssen wir noch ein anderes Problem lösen: sowohl der Generator als auch der gezeigte Ansatz zu Integrated-Shrinking unterstützen nur Propertys, die einen einzigen Parameter haben. Man muss also einen geeigneten Weg finden, um Tests auszuführen, die komplexer sind. Vielleicht ist für die Lösung dieses Problems ein Blick in die Implementation von QuickCheck hilfreich, das genau die selbe Problematik lösen musste.

Abbildungsverzeichnis

Listingverzeichnis

Bibliographie

[1] J. Hughes, „How to Specify It!: A Guide to Writing Properties of Pure Functions“, *Trends in Functional Programming*, Bd. 12053. Springer International Publishing, Cham, S. 58–83, 2020. doi: 10.1007/978-3-030-47147-7_4.

[2] „frege/README.md at master · Frege/frege“. Zugegriffen: 20. Januar 2025. [Online]. Verfügbar unter: https://github.com/Frege/frege/blob/master/README.md

[3] J. Link, „jqwik“. Zugegriffen: 20. Januar 2025. [Online]. Verfügbar unter: https://jqwik.net/

[4] J. Link, „Property-based Testing in Java: Introduction“. Zugegriffen: 20. Januar 2025. [Online]. Verfügbar unter: https://blog.johanneslink.net/2018/03/24/property-based-testing-in-java-introduction/

[5] „hedgehogqa/haskell-hedgehog“. Zugegriffen: 20. Januar 2025. [Online]. Verfügbar unter: https://github.com/hedgehogqa/haskell-hedgehog

[6] E. de Vries, „Integrated versus Manual Shrinking“. Zugegriffen: 20. Januar 2025. [Online]. Verfügbar unter: https://well-typed.com/blog/2019/05/integrated-shrinking/

[7] R. Draper, „Proposal: free shrinking with QuickCheck“. Zugegriffen: 13. Januar 2025. [Online]. Verfügbar unter: https://mail.haskell.org/pipermail/libraries/2013-November/021674.html

[8] C. Okasaki, „Breadth-first numbering: lessons from a small exercise in algorithm design“, *SIGPLAN Not.*, Bd. 35, Nr. 9, S. 131–136, Sep. 2000, doi: 10.1145/357766.351253.