QuickChecking with dependent types

Adventures in automatic generation of dependently typed values and derivation of such generators

Denis Buzdalov

Idris Developer Meeting
08.12.2022

About this talk

About this talk

- Three perspectives
  - user perspective, not so familiar with property-based testing

About this talk

- Three perspectives
  - user perspective, not so familiar with property-based testing
  - user perspective, sold to property-based testing, specifics of dependent types

About this talk

- Three perspectives
  - user perspective, not so familiar with property-based testing
  - user perspective, sold to property-based testing, specifics of dependent types
  - implementer's perspective, elaboration reflection and derivation design

About this talk

- Three perspectives
  - user perspective, not so familiar with property-based testing
  - user perspective, sold to property-based testing, specifics of dependent types
  - implementer's perspective, elaboration reflection and derivation design
- Overlays and lists ;-)

About this talk

- Three perspectives
  - user perspective, not so familiar with property-based testing
  - user perspective, sold to property-based testing, specifics of dependent types
  - implementer's perspective, elaboration reflection and derivation design
- Overlays and lists ;-)
- Questions at any time

Why? PBT and applications

Property-based testing roughly

- Property
- Search space
- Infrastructure

Property-based testing roughly

- Property
  - generally, a function `Input` → `Property`, must hold for any `Input`
- Search space
- Infrastructure

Property-based testing roughly

- Property
  - generally, a function `Input` → `Property`, must hold for any `Input`
  - can
    - call pure functions and compare results
  - target under test
    - pure functions and their combinations
- Search space
- Infrastructure

Property-based testing roughly

- Property
  - generally, a function `Input → Property`, must hold for any `Input`
  - can
    - call pure functions and compare results
    - perform `IO` actions
  - target under test
    - pure functions and their combinations
    - external systems
- Search space
- Infrastructure

Property-based testing roughly

- Property
  - generally, a function `Input → Property`, must hold for any `Input`
  - can
    - call pure functions and compare results
    - perform `IO` actions
    - update local models
  - target under test
    - pure functions and their combinations
    - external systems
    - ...including stateful APIs
- Search space
- Infrastructure

Property-based testing roughly

- Property
  - generally, a function `Input → Property`, must hold for any `Input`
  - can
    - call pure functions and compare results
    - perform `IO` actions
    - update local models
  - target under test
    - pure functions and their combinations
    - external systems
    - ...including stateful APIs
- Search space
  - generation `Seed → a`
- Infrastructure

Property-based testing roughly

- Property
  - generally, a function `Input → Property`, must hold for any `Input`
  - can
    - call pure functions and compare results
    - perform `IO` actions
    - update local models
  - target under test
    - pure functions and their combinations
    - external systems
    - ...including stateful APIs
- Search space
  - generation `Seed → a`
  - co-generation `a → Seed`
- Infrastructure

Property-based testing roughly

- Property
  - generally, a function `Input → Property`, must hold for any `Input`
  - can
    - call pure functions and compare results
    - perform `IO` actions
    - update local models
  - target under test
    - pure functions and their combinations
    - external systems
    - ...including stateful APIs
- Search space
  - generation `Seed → a`
  - co-generation `a → Seed`
  - shrinking `a → a`
- Infrastructure

Property-based testing roughly

- Property
  - generally, a function `Input → Property`, must hold for any `Input`
  - can
    - call pure functions and compare results
    - perform `IO` actions
    - update local models
  - target under test
    - pure functions and their combinations
    - external systems
    - ...including stateful APIs
- Search space
  - generation `Seed → a`
  - co-generation `a → Seed`
  - shrinking `a → a`
- Infrastructure
  - various test conditions
  - state machine models
  - derivation
  - ...

Property-based testing roughly

- Property
  - generally, a function `Input → Property`, must hold for any `Input`
  - can
    - call pure functions and compare results
    - perform `IO` actions
    - update local models
  - target under test
    - pure functions and their combinations
    - **external systems**
    - ...including stateful APIs
- Search space
  - **generation** `Seed → a`
  - co-generation `a → Seed`
  - shrinking `a → a`
- Infrastructure
  - various test conditions
  - state machine models
  - **derivation**
  - ...

Why dependent types?

Why dependent types?

- Target: pure functions

Why dependent types?

- Target: dependently-typed pure functions

Why dependent types?

- Target: dependently-typed pure functions
- Target: external systems

Why dependent types?

- Target: dependently-typed pure functions
- Target: external systems
  - e.g. compiler
    - say, black-box testing of a backend

Why dependent types?

- Target: dependently-typed pure functions
- Target: external systems
  - e.g. compiler
    - say, black-box testing of a backend
    - or, monolithic multistage compiler

Why dependent types?

- Target: dependently-typed pure functions
- Target: external systems
  - e.g. compiler
    - say, black-box testing of a backend
    - or, monolithic multistage compiler
    - input: well-typed program

Why dependent types?

- Target: dependently-typed pure functions
- Target: external systems
  - e.g. compiler
    - say, black-box testing of a backend
    - or, monolithic multistage compiler
    - input: well-typed program
  - e.g. runtime (a-la JVM)
    - say, specific OEM compiler only

Why dependent types?

- Target: dependently-typed pure functions
- Target: external systems
  - e.g. compiler
    - say, black-box testing of a backend
    - or, monolithic multistage compiler
    - input: well-typed program
  - e.g. runtime (a-la JVM)
    - say, specific OEM compiler only
    - or, complex internal invariant

Why dependent types?

- Target: dependently-typed pure functions
- Target: external systems
  - e.g. compiler
    - say, black-box testing of a backend
    - or, monolithic multistage compiler
    - input: well-typed program
  - e.g. runtime (a-la JVM)
    - say, specific OEM compiler only
    - or, complex internal invariant
    - input: well-typed program (high- or low-level)

Why dependent types?

- Target: dependently-typed pure functions
- Target: external systems
  - e.g. compiler
    - say, black-box testing of a backend
    - or, monolithic multistage compiler
    - input: well-typed program
  - e.g. runtime (a-la JVM)
    - say, specific OEM compiler only
    - or, complex internal invariant
    - input: well-typed program (high- or low-level)
  - e.g. OS driver
    - complex environment
    - follows sophisticated protocol

Why dependent types?

- Target: dependently-typed pure functions
- Target: external systems
  - e.g. compiler
    - say, black-box testing of a backend
    - or, monolithic multistage compiler
    - input: well-typed program
  - e.g. runtime (a-la JVM)
    - say, specific OEM compiler only
    - or, complex internal invariant
    - input: well-typed program (high- or low-level)
  - e.g. OS driver
    - complex environment
    - follows sophisticated protocol
    - input: sequence of protocol-respecting API calls

Why dependent types?

- Target: dependently-typed pure functions
- Target: external systems
  - e.g. compiler
    - say, black-box testing of a backend
    - or, monolithic multistage compiler
    - input: well-typed program
  - e.g. runtime (a-la JVM)
    - say, specific OEM compiler only
    - or, complex internal invariant
    - input: well-typed program (high- or low-level)
  - e.g. OS driver
    - complex environment
    - follows sophisticated protocol
    - input: sequence of protocol-respecting API calls
  - the idea:
    - describe sparse input domain with dependent types
    - split description and generation

Open questions

- Does it look like repairing car engine through exhaust?

Open questions

- Does it look like repairing car engine through exhaust?
- Is this route easier?

Open questions

- Does it look like repairing car engine through exhaust?
- Is this route easier?
- What class of dependent types can be supported this way?

Tiny example

Sequence of abstract name **definitions** and abstract **usages** of previously defined names

Tiny example

Sequence of abstract name **definitions** and abstract **usages** of previously defined names, no redefinitions allowed

Tiny example

Sequence of abstract name **definitions** and abstract **usages** of previously defined names, no redefinitions allowed

```
data Stmts : (defsBefore, defsAfter : List Name) → Type where

  Def : (new : Name) →
        All (\def ⇒ So $ def /= new) defs ⇒
        Stmts defs (new :: defs)

  Use : (usd : Name) →
        Any (\def ⇒ So $ def == usd) defs ⇒
        Stmts defs defs

  (>>) : Stmts pre mid → Stmts mid post → Stmts pre post
```

Tiny example

```
x : Stmts [] ["x"]
x = do
  Def "x"
  Use "x"

xyz : Stmts [] ["z","y","x"]
xyz = do
  Def "x"
  Use "x"
  Def "y"
  Def "z"
  Use "x"
```

```
bad1 : Stmts [] ["x"]
bad1 = do
  Use "x"
  Def "x"

bad2 : Stmts [] ["x","x"]
bad2 = do
  Def "x"
  Use "x"
  Def "x"
```

Slightly bigger example: primitive imperative language

```
data Statement : (preV  : Variables) → (preR  : Registers rc) →
                 (postV : Variables) → (postR : Registers rc) →
                 Type where

  Nop  : Statement vars regs vars regs

  (.)  : (ty : Type') → (n : Name) → Statement vars regs ((n, ty)::vars) regs

  (#=) : (n : Name) → (0 lk : Lookup n vars) ⇒ (v : Expression vars regs lk.reveal) →
         Statement vars regs vars regs

  (%=) : {0 preR : Registers rc} → (reg : Fin rc) → Expression vars preR ty →
         Statement vars preR vars $ preR `With` (reg, Just ty)

  For  : (init : Statement preV preR insideV insideR) → (cond : Expression insideV insideR Bool') →
         (upd  : Statement insideV insideR insideV updR) → (0 _ : updR =%= insideR) ⇒
         (body : Statement insideV insideR postBodyV bodyR) → (0 _ : bodyR =%= insideR) ⇒
         Statement preV preR preV insideR

  If__ : (cond : Expression vars regs Bool') → Statement vars regs varsThen regsThen →
         Statement vars regs varsElse regsElse → Statement vars regs vars $ Merge regsThen regsElse

  (*>) : Statement preV preR midV midR → Statement midV midR postV postR → Statement preV preR postV postR

  Block : Statement preV preR insideV postR → Statement preV preR preV postR

  Print : Show (idrTypeOf ty) ⇒ Expression vars regs ty → Statement vars regs vars regs
```

Shape of generators

Some arbitrary choices

- QuickCheck (Haskell, Idris 1)
  - `Gen a` is data and `Arbitrary a` is typeclass
  - `f :: Gen a → Gen [a]`, but
  - `instance Arbitrary a ⇒ Arbitrary [a]`

Some arbitrary choices

- QuickCheck (Haskell, Idris 1)
  - `Gen a` is data and `Arbitrary a` is typeclass
  - `f :: Gen a → Gen [a]`, but
  - `instance Arbitrary a ⇒ Arbitrary [a]`
- Hedgehog (Haskell, Idris 2)
  - `Gen a` is data
  - `list : Range Nat → Gen a → Gen (List a)`

Some arbitrary choices

- QuickCheck (Haskell, Idris 1)
  - `Gen a` is data and `Arbitrary a` is typeclass
  - `f :: Gen a → Gen [a]`, but
  - `instance Arbitrary a ⇒ Arbitrary [a]`
- Hedgehog (Haskell, Idris 2)
  - `Gen a` is data
  - `list : Range Nat → Gen a → Gen (List a)`
- Validity (Haskell)
  - `Gen a` is data and `GenUnchecked a`, `Validity a` and `GenValid a` are typeclasses

Some arbitrary choices

- QuickCheck (Haskell, Idris 1)
  - `Gen a` is data and `Arbitrary a` is typeclass
  - `f :: Gen a → Gen [a]`, but
  - `instance Arbitrary a ⇒ Arbitrary [a]`
- Hedgehog (Haskell, Idris 2)
  - `Gen a` is data
  - `list : Range Nat → Gen a → Gen (List a)`
- Validity (Haskell)
  - `Gen a` is data and `GenUnchecked a`, `Validity a` and `GenValid a` are typeclasses
- DepTyCheck (Idris 2)
  - `Gen a` technically is data

Some arbitrary choices

- QuickCheck (Haskell, Idris 1)
  - `Gen a` is data and `Arbitrary a` is typeclass
  - `f :: Gen a → Gen [a]`, but
  - `instance Arbitrary a ⇒ Arbitrary [a]`
- Hedgehog (Haskell, Idris 2)
  - `Gen a` is data
  - `list : Range Nat → Gen a → Gen (List a)`
- Validity (Haskell)
  - `Gen a` is data and `GenUnchecked a`, `Validity a` and `GenValid a` are typeclasses
- DepTyCheck (Idris 2)
  - `Gen a` technically is data, pragmatically used as both

Some arbitrary choices

- QuickCheck (Haskell, Idris 1)
  - `Gen a` is data and `Arbitrary a` is typeclass
  - `f :: Gen a → Gen [a]`, but
  - `instance Arbitrary a ⇒ Arbitrary [a]`
- Hedgehog (Haskell, Idris 2)
  - `Gen a` is data
  - `list : Range Nat → Gen a → Gen (List a)`
- Validity (Haskell)
  - `Gen a` is data and `GenUnchecked a`, `Validity a` and `GenValid a` are typeclasses
- DepTyCheck (Idris 2)
  - `Gen a` technically is data, pragmatically used as both
  - `list : Gen a ⇒ (length : Gen Nat) → Gen (List a)`

Some arbitrary choices

- QuickCheck (Haskell, Idris 1)
  - `Gen a` is data and `Arbitrary a` is typeclass
  - `f :: Gen a → Gen [a]`, but
  - `instance Arbitrary a ⇒ Arbitrary [a]`
- Hedgehog (Haskell, Idris 2)
  - `Gen a` is data
  - `list : Range Nat → Gen a → Gen (List a)`
- Validity (Haskell)
  - `Gen a` is data and `GenUnchecked a`, `Validity a` and `GenValid a` are typeclasses
- DepTyCheck (Idris 2)
  - `Gen a` technically is data, pragmatically used as both
  - `list : Gen a ⇒ (length : Gen Nat) → Gen (List a)`
  - ```
    %hint
    DefaultList : Gen a ⇒ Gen (List a)
    DefaultList = list progressiveNat
    ```
    ```

Garçon! Generator, please

- “Give me a generator of `Fin` s”

Garçon! Generator, please

- “Give me a generator of `Fin` s”

```
Fin : Nat → Type
```

Garçon! Generator, please

- “Give me a generator of `Fin` s”
  - `fins : {n : Nat} → Gen (Fin n)`

```
Fin : Nat → Type
```

Garçon! Generator, please

- “Give me a generator of `Fin` s”
  - `fins : {n : Nat} → Gen (Fin n)`
  - `fins : (n : Nat) → Gen (Fin n)`

```
Fin : Nat → Type
```

Garçon! Generator, please

- “Give me a generator of `Fin` s”

```
Fin : Nat → Type
```

  - `fins : {n : Nat} → Gen (Fin n)`
  - `fins : (n : Nat) → Gen (Fin n)`
  - `fins : Gen (n ** Fin n)`

Garçon! Generator, please

- "Give me a generator of `Fin` s" `Fin : Nat → Type`
  - `fins : {n : Nat} → Gen (Fin n)`
  - `fins : (n : Nat) → Gen (Fin n)`
  - `fins : Gen (n ** Fin n)`
- "... of statements" `Stmts : (bef, aft : List Name) → Type`

Garçon! Generator, please

- "Give me a generator of `Fin` s" `Fin : Nat → Type`
  - `fins : {n : Nat} → Gen (Fin n)`
  - `fins : (n : Nat) → Gen (Fin n)`
  - `fins : Gen (n ** Fin n)`
- "... of statements" `Stmts : (bef, aft : List Name) → Type`
  - `stmts : (bef, aft : List Name) → Gen (Stmts bef aft)`

Garçon! Generator, please

- "Give me a generator of `Fin` s" `Fin : Nat → Type`
  - `fins : {n : Nat} → Gen (Fin n)`
  - `fins : (n : Nat) → Gen (Fin n)`
  - `fins : Gen (n ** Fin n)`
- "... of statements" `Stmts : (bef, aft : List Name) → Type`
  - `stmts : (bef, aft : List Name) → Gen (Stmts bef aft)`
  - `stmts : (bef : List Name) → Gen (aft ** Stmts bef aft)`

Garçon! Generator, please

- "Give me a generator of `Fin` s" `Fin : Nat → Type`
  - `fins : {n : Nat} → Gen (Fin n)`
  - `fins : (n : Nat) → Gen (Fin n)`
  - `fins : Gen (n ** Fin n)`
- "... of statements" `Stmts : (bef, aft : List Name) → Type`
  - `stmts : (bef, aft : List Name) → Gen (Stmts bef aft)`
  - `stmts : (bef : List Name) → Gen (aft ** Stmts bef aft)`
  - `stmts : (aft : List Name) → Gen (bef ** Stmts bef aft)`

Garçon! Generator, please

- "Give me a generator of `Fin` s" `Fin : Nat → Type`
  - `fins : {n : Nat} → Gen (Fin n)`
  - `fins : (n : Nat) → Gen (Fin n)`
  - `fins : Gen (n ** Fin n)`
- "... of statements" `Stmts : (bef, aft : List Name) → Type`
  - `stmts : (bef, aft : List Name) → Gen (Stmts bef aft)`
  - `stmts : (bef : List Name) → Gen (aft ** Stmts bef aft)`
  - `stmts : (aft : List Name) → Gen (bef ** Stmts bef aft)`
  - `stmts : Gen (bef ** aft ** Stmts bef aft)`

Garçon! Generator, please

- "Give me a generator of `Fin` s" `Fin : Nat → Type`
  - `fins : {n : Nat} → Gen (Fin n)`
  - `fins : (n : Nat) → Gen (Fin n)`
  - `fins : Gen (n ** Fin n)`
- "... of statements" `Stmts : (bef, aft : List Name) → Type`
  - `stmts : (bef, aft : List Name) → Gen (Stmts bef aft)`
  - `stmts : (bef : List Name) → Gen (aft ** Stmts bef aft)`
  - `stmts : (aft : List Name) → Gen (bef ** Stmts bef aft)`
  - `stmts : Gen (bef ** aft ** Stmts bef aft)`
  - `stmts : Gen (List Name) ⇒ Gen (bef ** aft ** Stmts bef aft)`

Garçon! Generator, please

- "Give me a generator of `Fin` s" `Fin : Nat → Type`
  - `fins : {n : Nat} → Gen (Fin n)`
  - `fins : (n : Nat) → Gen (Fin n)`
  - `fins : Gen (n ** Fin n)`
- "... of statements" `Stmts : (bef, aft : List Name) → Type`
  - `stmts : (bef, aft : List Name) → Gen (Stmts bef aft)`
  - `stmts : (bef : List Name) → Gen (aft ** Stmts bef aft)`
  - `stmts : (aft : List Name) → Gen (bef ** Stmts bef aft)`
  - `stmts : Gen (bef ** aft ** Stmts bef aft)`
  - `stmts : Gen (List Name) ⇒ Gen (bef ** aft ** Stmts bef aft)`
  - ```
    stmts : ({a : _} → Gen (List a)} ⇒
            Gen (bef ** aft ** Stmts bef aft)
    ```
    ```

Garçon! Generator, please

- “Give me a generator of `Vect` s”

```
Vect : Nat → Type → Type
```

Garçon! Generator, please

- “Give me a generator of `Vect` s” `Vect : Nat → Type → Type`
  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`

Garçon! Generator, please

- "Give me a generator of `Vect` s" `Vect : Nat → Type → Type`
  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : Gen a ⇒ (n : Nat) → {a : Type} → Gen (Vect n a)`

Garçon! Generator, please

- “Give me a generator of `Vect` s” `Vect : Nat → Type → Type`
  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : (n : Nat) → {a : Type} → Gen a ⇒ Gen (Vect n a)`

Garçon! Generator, please

- “Give me a generator of `Vect` s” `Vect : Nat → Type → Type`
  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : (n : Nat) → {a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : (n : Nat) → {0 a : Type} → Gen a ⇒ Gen (Vect n a)`

Garçon! Generator, please

- “Give me a generator of `Vect` s” `Vect : Nat → Type → Type`
  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : (n : Nat) → {a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : (n : Nat) → {0 a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : Gen a ⇒ (n : Nat) → Gen (Vect n a)`

Garçon! Generator, please

- “Give me a generator of `Vect` s” `Vect : Nat → Type → Type`
  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : (n : Nat) → {a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : (n : Nat) → {0 a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : Gen a ⇒ (n : Nat) → Gen (Vect n a)`
  - `anyVect : Gen (n ** a ** Vect n a)`

Garçon! Generator, please

- “Give me a generator of `Vect` s” `Vect : Nat → Type → Type`

  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : (n : Nat) → {a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : (n : Nat) → {0 a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : Gen a ⇒ (n : Nat) → Gen (Vect n a)`

  - `anyVect : Gen (n ** a ** Vect n a)`
  - `anyVect : Gen a ⇒ Gen (n ** a ** Vect n a)`

Garçon! Generator, please

- "Give me a generator of `Vect` s" `Vect : Nat → Type → Type`

  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : (n : Nat) → {a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : (n : Nat) → {0 a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : Gen a ⇒ (n : Nat) → Gen (Vect n a)`

  - `anyVect : Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (a ** a) ⇒ Gen (n ** a ** Vect n a)`

Garçon! Generator, please

- “Give me a generator of `Vect` s” `Vect : Nat → Type → Type`

  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : (n : Nat) → {a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : (n : Nat) → {0 a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : Gen a ⇒ (n : Nat) → Gen (Vect n a)`

  - `anyVect : Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (a ** a) ⇒ Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (a ** Gen a) ⇒ Gen (n ** a ** Vect n a)`

Garçon! Generator, please

- "Give me a generator of `Vect` s" `Vect : Nat → Type → Type`

  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : (n : Nat) → {a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : (n : Nat) → {0 a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : Gen a ⇒ (n : Nat) → Gen (Vect n a)`

  - `anyVect : Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (a ** a) ⇒ Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (a ** Gen a) ⇒ Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (Exists$\a⇒Gen a)⇒Gen (n ** Exists$\a⇒Vect n a)`

Garçon! Generator, please

- “Give me a generator of `Vect` s” `Vect : Nat → Type → Type`

  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : (n : Nat) → {a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : (n : Nat) → {0 a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : Gen a ⇒ (n : Nat) → Gen (Vect n a)`

  - `anyVect : Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (a ** a) ⇒ Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (a ** Gen a) ⇒ Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (Exists$\a⇒Gen a)⇒Gen (n ** Exists$\a⇒Vect n a)`
  - `anyVect : Gen (Exists Gen) ⇒ Gen (n ** Exists $ Vect n)`

Garçon! Generator, please

- "Give me a generator of `Vect` s" `Vect : Nat → Type → Type`

  - `vect : (n : Nat) → {a : Type} → Gen (Vect n a)`
  - `vect : (n : Nat) → {a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : (n : Nat) → {0 a : Type} → Gen a ⇒ Gen (Vect n a)`
  - `vect : Gen a ⇒ (n : Nat) → Gen (Vect n a)`

  - `anyVect : Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (a ** a) ⇒ Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (a ** Gen a) ⇒ Gen (n ** a ** Vect n a)`
  - `anyVect : Gen (Exists$\a⇒Gen a)⇒Gen (n ** Exists$\a⇒Vect n a)`
  - `anyVect : Gen (Exists Gen) ⇒ Gen (n ** Exists $ Vect n)`
  - `anyVect : Gen (Exists Gen) ⇒ Gen (Exists $ Exists . Vect)`

Difficult issue of emptiness

- QuickCheck
  - `Gen` can be empty
  - supports filtration
  - retries when failed to generate

Difficult issue of emptiness

- QuickCheck
  - `Gen` can be empty
  - supports filtration
  - retries when failed to generate
- Hedgehog (Idris version)
  - `Gen` intentionally cannot be empty
  - does not support filtration

Difficult issue of emptiness

- QuickCheck
  - `Gen` can be empty
  - supports filtration
  - retries when failed to generate
- Hedgehog (Idris version)
  - `Gen` intentionally cannot be empty
  - does not support filtration
- DepTyCheck
  - has split `Gen` and `NonEmptyGen`
  - supports filtration for `Gen`
  - distinguishes between static and dynamic emptiness
    - `(n : Nat) → Gen (Fin n)`
    - `Gen a ⇒ (length : Gen Nat) → Gen (List a)`
  - retries on dynamic failure

Adventures with derivation

What do you derive?

```
mutual
  data A : Type where ...

  data B : Type where ...
```

What do you derive?

```
mutual
  data A : Type where ...

  data B : Type where ...
```

```
mutual -- really working derivation below
  %hint ShowA : Show A
  ShowA = %runElab derive {mutualWith = [`{B}]}

  %hint ShowB : Show B
  ShowB = %runElab derive {mutualWith = [`{A}]}
```

What do you derive?

```
data Stmts : (defsBefore, defsAfter : List Name) → Type where ...
```

What do you derive?

```
data Stmts : (defsBefore, defsAfter : List Name) → Type where ...


mutual -- hypothetical derivation below

  %hint S_gg : Gen (pre ** post ** Stmts pre post)
  S_gg = %runElab derive {mutualWith =
    [ Giv [0, 1] `{Stmts}, Giv [0] `{Stmts}, Giv [1] `{Stmts} ]}

  %hint S_Gg : {pre : _} → Gen (post ** Stmts pre post)
  S_Gg = %runElab derive {mutualWith =
    [ Giv [0, 1] `{Stmts}, Giv [1] `{Stmts}, Giv [] `{Stmts} ]}

  %hint S_gG : {post : _} → Gen (pre ** Stmts pre post)
  S_gG = %runElab derive {mutualWith =
    [ Giv [0, 1] `{Stmts}, Giv [0] `{Stmts}, Giv [] `{Stmts} ]}

  %hint S_GG : {pre, post : _} → Gen (Stmts pre post)
  S_GG = %runElab derive {mutualWith =
    [ Giv [0] `{Stmts}, Giv [1] `{Stmts}, Giv [] `{Stmts} ]}
```

Now, imagine more...

```
mutual

  data Stmts : (defsBefore, defsAfter : List Name) → Type where ...

  data AuxDef : Ctx1 → Ctx2 → ... → Type where ...

  data MoreDefs : Ctx1 → Type where ...

  data WellTypedBlock : Ctx → Type where
```

Now, imagine more...

```
mutual

  data Stmts : (defsBefore, defsAfter : List Name) → Type where ...

  data AuxDef : Ctx1 → Ctx2 → ... → Type where ...

  data MoreDefs : Ctx1 → Type where ...

  data WellTypedBlock : Ctx → Type where
```

Say, give me `(ctx : _) → Gen (WellTypedBlock ctx)`!

Now, imagine more...

```
mutual

  data Stmts : (defsBefore, defsAfter : List Name) → Type where ...

  data AuxDef : Ctx1 → Ctx2 → ... → Type where ...

  data MoreDefs : Ctx1 → Type where ...

  data WellTypedBlock : Ctx → Type where
```

Say, give me `(ctx : _) → Gen (WellTypedBlock ctx)`!

- a lot

Now, imagine more...

```
mutual

  data Stmts : (defsBefore, defsAfter : List Name) → Type where ...

  data AuxDef : Ctx1 → Ctx2 → ... → Type where ...

  data MoreDefs : Ctx1 → Type where ...

  data WellTypedBlock : Ctx → Type where
```

Say, give me `(ctx : _) → Gen (WellTypedBlock ctx)`!

- a lot
- depends on implementation!

More decisions to make

```
data X : Nat → Nat → Type where ...
data Y : Nat → Nat → Type where ...

data Z : Type where
  MkZ : X n m → Y n k → Z
```

More decisions to make

```
data X : Nat → Nat → Type where ...
data Y : Nat → Nat → Type where ...

data Z : Type where
  MkZ : X n m → Y n k → Z
```

Which generators of `X` would you use for `Y`?

More decisions to make

```
data X : Nat → Nat → Type where ...
data Y : Nat → Nat → Type where ...

data Z : Type where
  MkZ : X n m → Y n k → Z
```

Which generators of `X` would you use for `Y`?

- `(n, m : _) → Gen (X n m)` and `(n, m : _) → Gen (Y n m)`?

More decisions to make

```
data X : Nat → Nat → Type where ...
data Y : Nat → Nat → Type where ...

data Z : Type where
  MkZ : X n m → Y n k → Z
```

Which generators of `X` would you use for `Y`?

- `(n, m : _) → Gen (X n m)` and `(n, m : _) → Gen (Y n m)`?
- `Gen (n ** m ** X n m)` and `(n : _) → Gen (m ** Y n m)`?

More decisions to make

```
data X : Nat → Nat → Type where ...
data Y : Nat → Nat → Type where ...

data Z : Type where
  MkZ : X n m → Y n k → Z
```

Which generators of `X` would you use for `Y`?

- `(n, m : _) → Gen (X n m)` and `(n, m : _) → Gen (Y n m)`?
- `Gen (n ** m ** X n m)` and `(n : _) → Gen (m ** Y n m)`?
- `Gen (n ** m ** Y n m)` and `(n : _) → Gen (m ** X n m)`?

More decisions to make

```
data X : Nat → Nat → Type where ...
data Y : Nat → Nat → Type where ...

data Z : Type where
  MkZ : X n m → Y n k → Z
```

Which generators of `X` would you use for `Y`?

- `(n, m : _) → Gen (X n m)` and `(n, m : _) → Gen (Y n m)`?
- `Gen (n ** m ** X n m)` and `(n : _) → Gen (m ** Y n m)`?
- `Gen (n ** m ** Y n m)` and `(n : _) → Gen (m ** X n m)`?

What if

```
data X : Nat → Nat → Type where
  X4 : {n : _} → X 4 n
  X8 : {n : _} → X 8 n
```

More decisions to make

```
data X : Nat → Nat → Type where ...
data Y : Nat → Nat → Type where ...

data Z : Type where
  MkZ : X n m → Y n k → Z
```

Which generators of `X` would you use for `Y`?

- `(n, m : _) → Gen (X n m)` and `(n, m : _) → Gen (Y n m)`?
- `Gen (n ** m ** X n m)` and `(n : _) → Gen (m ** Y n m)`?
- `Gen (n ** m ** Y n m)` and `(n : _) → Gen (m ** X n m)`?

What if

```
data X : Nat → Nat → Type where
  X4 : {n : _} → X 4 n
  X8 : {n : _} → X 8 n

data X : Nat → Nat → Type where
  MkX : {n, m : _} → LTE n m ⇒ X n m
```

On current DepTyCheck

Thank you!

Questions?

https://github.com/buzden/deptycheck