Dependent Types to Push Corners of the Property-based Testing

Denis Buzdalov

ICFP, TyDe Workshop
6 September 2024
Milan, Italy

I hope you'll agree that

- Dependent types are cool

```
data AtIndex : Fin n → Vect n a → a → Type where
  Here  : AtIndex FZ (x :: xs) x
  There : AtIndex idx xs y → AtIndex (FS idx) (x::xs) y
```

I hope you'll agree that

- Dependent types are cool

```
data AtIndex : Fin n -> Vect n a -> a -> Type where
  Here  : AtIndex FZ (x :: xs) x
  There : AtIndex idx xs y -> AtIndex (FS idx) (x::xs) y
```

- Property-based testing is cool too

```
indexInserted : Property
indexInserted = property $ do
  (n, x)     <- forAll [| (nat (linear 0 100), anyInt64) |]
  (xs, idx) <- forAll [| (vect n anyInt64, fin (linearFin n)) |]
  f          <- apply <$> forAll (function anyInt32)
  insertAt idx (f x) (map f xs) === map f (insertAt idx x xs)
```

What does it mean to be type-driven?

What does it mean to be type-driven?

- We invest in expressing the intent with types

What does it mean to be type-driven?

- We invest in expressing the intent with types
- We get good things
  - less incorrect implementations
  - more confidence in the code
  - less unneeded code/checks
  - help from compilers and tools
  - ...

What does it mean to be type-driven?

- We invest in expressing the intent with types
- We get good things
  - less incorrect implementations
  - more confidence in the code
  - less unneeded code/checks
  - help from compilers and tools
  - ...
  - better and more powerful tests!

Type-driven property-based testing?

Being given two sorted lists, function `merge` *produces a sorted list*

Type-driven property-based testing?

Being given two sorted lists, function `merge` *produces a sorted list*

- Special generator of sorted lists for `List`?

```
sortedList : Gen (List Nat)
sortedList = anyList <&> foldr (\x, xs ⇒ x :: map (x+) xs) []
```

Type-driven property-based testing?

Being given two sorted lists, function `merge` *produces a sorted list*

- Special generator of sorted lists for `List`?

```
sortedList : Gen (List Nat)
sortedList = anyList <&> foldr (\x, xs => x :: map (x+) xs) []
```

- Special wrapper for sorted lists?

```
data SortedList = SL (List Nat)
sortedList : Gen SortedList
```

Type-driven property-based testing?

Being given two sorted lists, function `merge` *produces a sorted list*

- Special generator of sorted lists for `List`?

```
sortedList : Gen (List Nat)
sortedList = anyList <&> foldr (\x, xs => x :: map (x+) xs) []
```

- Special wrapper for sorted lists?

```
data SortedList = SL (List Nat)
sortedList : Gen SortedList
```

- Describe intent in type, derive generator!

Type-driven property-based testing!

```
mutual
  data SortedList : Type where
    Nil  : SortedList
    (::) : (x : Nat) -> (xs : SortedList) ->
           (0 _ : So $ canPrepend x xs) => SortedList
  canPrepend : Nat -> SortedList -> Bool
  canPrepend n = \case [] => True; x::xs => n < x

anySortedList : Gen SortedList -- derived complete generator
```

Type-driven property-based testing!

```
mutual
  data SortedList : Type where
    Nil   : SortedList
    (::) : (x : Nat) -> (xs : SortedList) ->
           (0 _ : So $ canPrepend x xs) => SortedList
  canPrepend : Nat -> SortedList -> Bool
  canPrepend n = \case [] => True; x::xs => n < x

anySortedList : Gen SortedList -- derived complete generator

toList : SortedList -> List Nat

mergeSorted : Property
mergeSorted = property $ do
  (xs, ys) <- forAll [| (anySortedList, anySortedList) |]
  assert $ sorted (merge (toList xs) (toList ys))
```

Type-driven property-based testing!

- Surely, you can still make a mistake

Type-driven property-based testing!

- Surely, you can still make a mistake
- ...but in a declarative specification

Type-driven property-based testing!

- Surely, you can still make a mistake
- ...but in a declarative specification
- Completeness and good distribution by library

Type-driven property-based testing!

- Surely, you can still make a mistake
- ...but in a declarative specification
- Completeness and good distribution by library
- ...but fine tuning is hard

Let’s try

- New language implementation
- Restricted dialect of TypeScript, static strict typing
- Interpreter + JIT; AOT compiler

Let’s try

- New language implementation
- Restricted dialect of TypeScript, static strict typing
- Interpreter + JIT; AOT compiler
- Properties
  - semantically correct programs should be successfully compilable

Let’s try

- New language implementation
- Restricted dialect of TypeScript, static strict typing
- Interpreter + JIT; AOT compiler
- Properties
    - semantically correct programs should be successfully compilable
    - among them, halting programs should run and be interpretable without unexpected crashes

Let’s try

- New language implementation
- Restricted dialect of TypeScript, static strict typing
- Interpreter + JIT; AOT compiler
- Properties
  - semantically correct programs should be successfully compilable
  - among them, halting programs should run and be interpretable without unexpected crashes
  - all these running modes should produce the same result

How specification can look like

```
data Stmts : (functions  : List (Name, FunSig)) →
             (varsBefore : List (Name, Type)) →
             (varsAfter  : List (Name, Type)) → Type where

  (.)   : (ty : Type) → (n : Name) →
          Stmts funs vars ((n, ty)::vars)

  (#=)  : (n : Name) → (0 lk : n `IsIn` vars) ⇒
          (v : Expr funs vars (found lk)) →
          Stmts funs vars vars

  If    : (cond : Expr funs vars Bool) →
          Stmts funs vars vThen → Stmts funs vars vElse →
          Stmts funs vars vars

  (>>)  : Stmts funs preV midV  → Stmts funs midV postV →
          Stmts funs preV postV
```

How specification can look like

```
record FunSig where
  constructor (==>)
  From : List Type
  To   : Type

data Expr : List (Name, FunSig) -> List (Name, Type) ->
            Type -> Type where

  C : (x : ty) -> Expr funs vars ty

  V : (n : Name) -> (0 lk : n `IsIn` vars) =>
      Expr funs vars (found lk)

  F : (n : Name) -> (0 lk : n `IsIn` funs) =>
      All (Expr funs vars) (found lk).From ->
      Expr funs vars (found lk).To
```

Semantically correct program

```
StdF : List (Name, FunSig)
StdF = [ ("+" , [Int, Int] ⟹ Int)
       , ("<" , [Int, Int] ⟹ Bool)
       , ("++", [Int] ⟹ Int)
       , ("||", [Bool, Bool] ⟹ Bool) ]
```

```
program : Stmts StdF [] ?
program = do
  Int. "x"
  "x" #= C 5
  Int. "y"; Bool. "res"
  "y" #= F "+" [V "x", C 1]
  If (F "<" [F "++" [V "x"], V "y"])
     (do "y" #= C 0; "res" #= C False)
     (do Int. "z"; "z" #= F "+" [V "x", V "y"]
         Bool. "b"; "b" #= F "<" [V "x", C 5]
         "res" #= F "||" [V "b", F "<" [V "z", C 6]])
```

Semantically incorrect programs

```
failing "Mismatch between: Int and Bool"
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    Bool. "y"; "y" #= F "+" [V "x", C 1]

failing "Mismatch between: [] and [Int]"
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    Int. "y"; "y" #= F "+" [V "x"]

failing "Mismatch between: Bool and Int"
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    Int. "y"; "y" #= F "+" [C True, V "x"]
```

Actual specification

- Using DepTyCheck library

Actual specification

- Using DepTyCheck library
- Language subset
  - Halting programs
  - Loops, `ifs`, assignments, exceptions
  - Classes without arbitrary methods, numbers, arrays

Actual specification

- Using DepTyCheck library
- Language subset
  - Halting programs
  - Loops, `if`s, assignments, exceptions
  - Classes without arbitrary methods, numbers, arrays
- Specification “hacks”
  - De Bruijn indices
  - “Continuation-passing style”-like data
  - Specialised polymorphic data types
  - Grouping type indices
  - ...

Actual specification

- Using DepTyCheck library
- Language subset
  - Halting programs
  - Loops, `if`s, assignments, exceptions
  - Classes without arbitrary methods, numbers, arrays
- Specification "hacks"
  - De Bruijn indices
  - "Continuation-passing style"-like data
  - Specialised polymorphic data types
  - Grouping type indices
  - ...
- ~330 LOC of specification + same for harness

Actual specification

- Using DepTyCheck library
- Language subset
  - Halting programs
  - Loops, `ifs`, assignments, exceptions
  - Classes without arbitrary methods, numbers, arrays
- Specification “hacks”
  - De Bruijn indices
  - “Continuation-passing style”-like data
  - Specialised polymorphic data types
  - Grouping type indices
  - ...
- ~330 LOC of specification + same for harness
- Partially derived, partially hand-written generators

Actual specification

Testing...

Actual specification

```
ASSERTION FAILED: false
IN /.../optimizations/lse.cpp:851: GetEliminationCode
Backtrace [tid=2990599]:
#0 : 0x7f876a1776c0 PrintStack(std::ostream&)
#1 : 0x7f876a177562 debug::AssertionFail(...)
#2 : 0x7f8767185e10 compiler::Lse::GetEliminationCode(...)
#3 : 0x7f8767186a20 compiler::Lse::DeleteInstructions(...)
...
```

Actual specification

```
ASSERTION FAILED: false
IN /.../optimizations/lse.cpp:851: GetEliminationCode
Backtrace [tid=2990599]:
#0 : 0x7f876a1776c0 PrintStack(std::ostream&)
#1 : 0x7f876a177562 debug::AssertionFail(...)
#2 : 0x7f8767185e10 compiler::Lse::GetEliminationCode(...)
#3 : 0x7f8767186a20 compiler::Lse::DeleteInstructions(...)
...
```

Actual specification

```
ASSERTION FAILED: false
IN /.../optimizations/lse.cpp:851: GetEliminationCode
Backtrace [tid=2990599]:
#0 : 0x7f876a1776c0 PrintStack(std::ostream&)
#1 : 0x7f876a177562 debug::AssertionFail(...)
#2 : 0x7f8767185e10 compiler::Lse::GetEliminationCode(...)
#3 : 0x7f8767186a20 compiler::Lse::DeleteInstructions(...)
...
```

Shrinking...

Actual specification

```
class C0 {
  x0: boolean
}

function main() : void {
  let x1: C0 = {x0: true}
  while(x1.x0) {
    x1.x0 = x1.x0
    x1.x0 = false
  }
}
```

Actual specification

Testing...

Actual specification

```
TypeError: Unreachable statement. [<filename>:26:34]
```

Actual specification

```
TypeError: Unreachable statement. [<filename>:26:34]
```

Shrinking...

Actual specification

```
TypeError: Unreachable statement. [<filename>:3:30]
```

```
function main() : void {
  let x1: Int = 1
  while(([false, true])[x1]) {
  }
}
```

Actual specification

Testing...

Actual specification

```
ASSERTION FAILED: block->GetGraph() == GetGraph()
IN /.../optimizer/ir/graph_cloner.h:176: GetClone
Backtrace [tid=2902033]:
#0 : 0x7fe71892b820 PrintStack(std::ostream&)
#1 : 0x7fe71892b6c2 debug::AssertionFail(...)
#2 : 0x7fe71a61ae61 compiler::GraphCloner::GetClone(...)
#3 : 0x7fe71a61162a compiler::GraphCloner::CopyLoop(...)
#4 : 0x7fe71a611839 compiler::GraphCloner::CopyLoop(...)
#5 : 0x7fe71a611173 compiler::GraphCloner::CloneAnalyses(...)
#6 : 0x7fe71a610d1f compiler::GraphCloner::CloneGraph()
#7 : 0x7fe71a5b377c compiler::GraphChecker::GraphChecker(...)
...
```

Actual specification

```
ASSERTION FAILED: block->GetGraph() == GetGraph()
IN /.../optimizer/ir/graph_cloner.h:176: GetClone
Backtrace [tid=2902033]:
#0 : 0x7fe71892b820 PrintStack(std::ostream&)
#1 : 0x7fe71892b6c2 debug::AssertionFail(...)
#2 : 0x7fe71a61ae61 compiler::GraphCloner::GetClone(...)
#3 : 0x7fe71a61162a compiler::GraphCloner::CopyLoop(...)
#4 : 0x7fe71a611839 compiler::GraphCloner::CopyLoop(...)
#5 : 0x7fe71a611173 compiler::GraphCloner::CloneAnalyses(...)
#6 : 0x7fe71a610d1f compiler::GraphCloner::CloneGraph()
#7 : 0x7fe71a5b377c compiler::GraphChecker::GraphChecker(...)
...
```

Shrinking...

Actual specification

```
class C0 {
  x0: boolean

  f() : string {
    return ""
  }
}
```

```
function main() : void {
  let x2: C0 = {x0: true}
  let fuel0 = 1
  while(fuel0 > 0) {
    do {
      fuel0--
      do {
        fuel0--
        let s = x2.f()
      } while(true && (fuel0 > 0))
    } while(true && (fuel0 > 0))
  }
}
```

Actual specification

- ...and so on
- 9 bugs in typechecker, JIT- and AOT-optimisers, code generator
- 8 more bugs during specification

What we’ve (re)learned

- Same property can find many different bugs

What we've (re)learned

- Same property can find many different bugs
- Subtle bugs are in interaction of many features

What we've (re)learned

- Same property can find many different bugs
- Subtle bugs are in interaction of many features
- PBT can find bug that are tricky to find manually

What we’ve (re)learned

- Same property can find many different bugs
- Subtle bugs are in interaction of many features
- PBT can find bug that are tricky to find manually
- PBT is cool!

What we've (re)learned

- Same property can find many different bugs
- Subtle bugs are in interaction of many features
- PBT can find bug that are tricky to find manually
- PBT is cool!
- Dependent types may be cool for PBT!

What are generators anyway?

```
data Fin : Nat → Type
```

What are generators anyway?

```
data Fin : Nat → Type
```

What's a signature for generator of `Fin`?

What are generators anyway?

```
data Fin : Nat → Type
```

What's a signature for generator of `Fin`?

```
finN   : (n : Nat) → Gen (Fin n)
finAny : Gen (n ** Fin n)
```

What are generators anyway?

```
data Fin : Nat -> Type
```

What's a signature for generator of `Fin`?

```
finN   : (n : Nat) -> Gen (Fin n)
finAny : Gen (n ** Fin n)
```

```
data LT : Nat -> Nat -> Type
```

What are generators anyway?

```
data Fin : Nat -> Type
```

What's a signature for generator of `Fin`?

```
finN   : (n : Nat) -> Gen (Fin n)
finAny : Gen (n ** Fin n)
```

```
data LT : Nat -> Nat -> Type
```

```
ltAny   : Gen (l ** r ** LT l r)
ltLeft  : (r : Nat) -> Gen (l ** LT l r)
ltRight : (l : Nat) -> Gen (r ** LT l r)
ltGiven : (l, r : Nat) -> Gen (LT l r)
```

Can generators be empty?

Can generators be empty?

- QuickCheck says “yes”, limited retry

Can generators be empty?

- QuickCheck says “yes”, limited retry
- Idris Hedgehog says “no”, disallowing even filtering

Can generators be empty?

- QuickCheck says "yes", limited retry
- Idris Hedgehog says "no", disallowing even filtering
- We can't afford saying "no", recall

```
finN : (n : Nat) → Gen (Fin n)
```

Can generators be empty?

- QuickCheck says "yes", limited retry
- Idris Hedgehog says "no", disallowing even filtering
- We can't afford saying "no", recall

  ```
  finN : (n : Nat) → Gen (Fin n)
  ```

- Generation spends most of the time in retrying

Compile-time control of emptiness

```
data Emptiness = NonEmpty | MaybeEmptyDeep | MaybeEmpty
data Gen : Emptiness → Type → Type
```

Compile-time control of emptiness

```
data Emptiness = NonEmpty | MaybeEmptyDeep | MaybeEmpty
data Gen : Emptiness → Type → Type
```

```
finN   : (n : Nat) → Gen MaybeEmpty (Fin n)
finSN  : (n : Nat) → Gen NonEmpty (Fin (S n))
finAny : Gen NonEmpty (n ** Fin n)
```

Compile-time control of emptiness

```
data Gen : Emptiness → Type → Type where
  --- ... other stuff ... ---
  OneOf : alem `NoWeaker` em ⇒ NotImmediatelyEmpty alem ⇒
          GenAlternatives True alem a → Gen em a
```

Compile-time control of emptiness

```
data GenAlternatives : (notEmpty : Bool) → Emptiness → Type → Type
```

```
data Gen : Emptiness → Type → Type where
  --- ... other stuff ... ---
  OneOf : alem `NoWeaker` em ⇒ NotImmediatelyEmpty alem ⇒
          GenAlternatives True alem a → Gen em a
```

Compile-time control of emptiness

```
data GenAlternatives : (notEmpty : Bool) → Emptiness → Type → Type
```

```
data Gen : Emptiness → Type → Type where
  --- ... other stuff ... ---
  OneOf : alem `NoWeaker` em ⇒ NotImmediatelyEmpty alem ⇒
          GenAlternatives True alem a → Gen em a

oneOf : alem `NoWeaker` em ⇒ AltsNonEmpty altsNe em ⇒
        GenAlternatives altsNe alem a → Gen em a
```

Compile-time control of emptiness

```
data GenAlternatives : (notEmpty : Bool) -> Emptiness -> Type -> Type
data AltsNonEmpty : Bool -> Emptiness -> Type
```

```
data Gen : Emptiness -> Type -> Type where
  --- ... other stuff ... ---
  OneOf : alem `NoWeaker` em => NotImmediatelyEmpty alem =>
          GenAlternatives True alem a -> Gen em a

oneOf : alem `NoWeaker` em => AltsNonEmpty altsNe em =>
        GenAlternatives altsNe alem a -> Gen em a
```

Compile-time control of emptiness

```
data GenAlternatives : (notEmpty : Bool) -> Emptiness -> Type -> Type
data AltsNonEmpty : Bool -> Emptiness -> Type
```

```
data Gen : Emptiness -> Type -> Type where
  --- ... other stuff ... ---
  OneOf : alem `NoWeaker` em => NotImmediatelyEmpty alem =>
          GenAlternatives True alem a -> Gen em a

oneOf : alem `NoWeaker` em => AltsNonEmpty altsNe em =>
        GenAlternatives altsNe alem a -> Gen em a
```

```
g1, g2 : Gen MaybeEmpty Nat
g1 = oneOf [empty, elements [0, 1, 2], empty]
g2 = elements [0, 1, 2]
```

Practical derivation

Practical derivation

- Complete
- Total

Practical derivation

- Complete
- Total
- Good distribution

Practical derivation

- Complete
- Total
- Good distribution

```
fins : Fuel -> (n : Nat) -> Gen MaybeEmpty (List (Fin n))
```

Practical derivation

- Complete
- Total
- Good distribution

```
fins : Fuel → (n : Nat) → Gen MaybeEmpty (List (Fin n))
```

- Whole-type derivation

Practical derivation

- Complete
- Total
- Good distribution

```
fins : Fuel → (n : Nat) → Gen MaybeEmpty (List (Fin n))
```

- Whole-type derivation

```
program : Fuel → (fs : _) → (vs : _) →
          Gen MaybeEmpty (vs' ** Stmts fs vs vs')
```

Practical derivation

- Complete
- Total
- Good distribution

```
fins : Fuel → (n : Nat) → Gen MaybeEmpty (List (Fin n))
```

- Whole-type derivation

```
program : Fuel → (fs : _) → (vs : _) →
          Gen MaybeEmpty (vs' ** Stmts fs vs vs')
```

- Efficient

Challenges

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

Challenges

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

zs : Fuel → Gen MaybeEmpty Z
zs = deriveGen
```

Challenges

```
data Z : Type where
  Zxy : X n m -> Y n k -> Z
  Zyx : Y n m -> X n k -> Z

zs : Fuel -> Gen MaybeEmpty Z
zs = deriveGen
```

```
genX'' : Gen0 (n ** m ** X n m)
genY' : (n:_) -> Gen0 (k ** Y n k)
genZxy : Gen0 Z
genZxy = do
  (n ** m ** x) <- genX''
  (k ** y) <- genY' n
  pure (Zxy x y)

    -- or --

genX' : (n:_) -> Gen0 (m ** X n m)
genY'' : Gen0 (n ** k ** Y n k)
genZxy' : Gen0 Z
genZxy' = do
  (n ** k ** y) <- genY''
  (m ** x) <- genX' n
  pure (Zxy x y)
```

Challenges

```
data Y : Nat -> Nat -> Type where
  MkY : (n, m : _) -> Y n m

data Z : Type where
  Zxy : X n m -> Y n k -> Z
  Zyx : Y n m -> X n k -> Z

zs : Fuel -> Gen MaybeEmpty Z
zs = deriveGen
```

```
genX'' : Gen0 (n ** m ** X n m)
genY' : (n:_) -> Gen0 (k ** Y n k)
genZxy : Gen0 Z
genZxy = do
  (n ** m ** x) <- genX''
  (k ** y) <- genY' n
  pure (Zxy x y)

    -- or --

genX' : (n:_) -> Gen0 (m ** X n m)
genY'' : Gen0 (n ** k ** Y n k)
genZxy' : Gen0 Z
genZxy' = do
  (n ** k ** y) <- genY''
  (m ** x) <- genX' n
  pure (Zxy x y)
```

Challenges

```
data X : Nat → Nat → Type where
  X45 : X 4 5
  X5m : X 5 m

data Y : Nat → Nat → Type where
  MkY : (n, m : _) → Y n m

data Z : Type where
  Zxy : X n m → Y n k → Z
  Zyx : Y n m → X n k → Z

zs : Fuel → Gen MaybeEmpty Z
zs = deriveGen
```

```
genX'' : Gen0 (n ** m ** X n m)
genY' : (n:_) → Gen0 (k ** Y n k)
genZxy : Gen0 Z
genZxy = do
  (n ** m ** x) <- genX''
  (k ** y) <- genY' n
  pure (Zxy x y)

    -- or --

genX' : (n:_) → Gen0 (m ** X n m)
genY'' : Gen0 (n ** k ** Y n k)
genZxy' : Gen0 Z
genZxy' = do
  (n ** k ** y) <- genY''
  (m ** x) <- genX' n
  pure (Zxy x y)
```

Challenges

```
f : Nat → Nat

data Yn : (n : Nat) → Fin n      → Type

data Z : Type where
  Z1 : (x : Fin (f n)) → Yn (f n) x → Z   -- n, (x ** y)
```

Challenges

```
f : Nat → Nat
```

```
data Yn : (n : Nat) → Fin n     → Type
data Yf : (n : Nat) → Fin (f n) → Type
```

```
data Z : Type where
  Z1 : (x : Fin (f n)) → Yn (f n) x → Z   -- n, (x ** y)
  Z2 : (x : Fin (f n)) → Yf n x     → Z   -- (n ** x ** y)
```

Challenges

```
f : Nat → Nat
g : Fin n → Fin n

data Yn : (n : Nat) → Fin n     → Type
data Yf : (n : Nat) → Fin (f n) → Type

data Z : Type where
  Z1 : (x : Fin (f n)) → Yn (f n) x → Z   -- n, (x ** y)
  Z2 : (x : Fin (f n)) → Yf n x     → Z   -- (n ** x ** y)
  Z3 : (x : Fin (f n)) → Yf n (g x) → Z   -- n, x, y
```

Challenges

```
f : Nat → Nat
g : Fin n → Fin n


data Yn : (n : Nat) → Fin n     → Type
data Yf : (n : Nat) → Fin (f n) → Type


data Z : Type where
  Z1 : (x : Fin (f n)) → Yn (f n) x → Z   -- n, (x ** y)
  Z2 : (x : Fin (f n)) → Yf n x     → Z   -- (n ** x ** y)
  Z3 : (x : Fin (f n)) → Yf n (g x) → Z   -- n, x, y
  Z4 : (x : Fin n)     → Yn n (g x) → Z   -- (n ** x), y
```

If you're interested

This presentation

DepTyCheck, examples

Code from the slides

Thank you!

This presentation

Code from the slides

DepTyCheck, examples

Questions?