Beyond correct and fast: Inspection Testing

Joachim Breitner, University of Pennsylvania

December 4, 2017, IBM PL days, T.J. Watson Research Center

An anecdote

The generic-lens library

```
data Employee = MkEmployee String Int


ageLens :: Lens' Employee Int
ageLens f (MkEmployee name age)
  = fmap (\newAge -> MkEmployee name newAge) (f age)
```

The generic-lens library

```
data Employee = MkEmployee String Int
    deriving Generic

ageLens :: Lens' Employee Int
ageLens = typed @Int
```

The generic-lens library

```
data Employee = MkEmployee String Int
    deriving Generic

ageLens :: Lens' Employee Int
ageLens = typed @Int

-- In the libraries (simplified):
from :: Generic a => a -> Rep a
to :: Generic a => Rep a -> a
typed :: Generic s => Lens' s a
typed = ravel (dimap from (fmap to) . gtyped)
```

A promise made

Using *super* in this manner might be more convenient for programmers who are not used to programming with lenses.

7. Conclusion

Deriving lenses generically gives the programmer the best of all possible worlds. The frugality to only define whichever lenses they need to use, the confidence that their abstraction will be without cost and the flexibility to four different types of lenses. This expressive and lightweight solution will hopefully inspire other library writers to embrace `GHC.Generics` as a solid basis on which to build their libraries.

References

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed

A promise made

Using *super* in this manner might be more convenient for programmers who are not used to programming with lenses.

7. Conclusion

Deriving lenses generically gives the programmer the best of all possible worlds. The frugality to only define whichever lenses they need to use, the confidence that their abstraction will be without cost and the flexibility to four different types of lenses. This expressive and lightweight solution will hopefully inspire other library writers to embrace `GHC.Generics` as a solid basis on which to build their libraries.

References

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed

A promise broken

Manual code:

```
ageLensManual :: Lens' Employee Int
ageLensManual =
  \ (@ (f_a6t0 :: * -> *))
    ($dFunctor_a6uA :: Functor f_a6t0)
    (f1_a6ps :: Int -> f_a6t0 Int)
    (ds_d7Kk :: Employee) ->
    case ds_d7Kk of _ { MkEmployee name_a6pt age_a6pu ->
    fmap
      @ f_a6t0
      $dFunctor_a6uA
      @ Int
      @ Employee
      (\ (newAge_a6pv :: Int) -> MkEmployee name_a6pt newAge_a6pv)
      (f1_a6ps age_a6pu)
    }
```

A promise broken

Manual code:

```
ageLensManual :: Lens' Employee Int
ageLensManual =
  \ (@ (f_a6t0 :: * -> *))
    ($dFunctor_a6uA :: Functor f_a6t0)
    (f1_a6ps :: Int -> f_a6t0 Int)
    (ds_d7Kk :: Employee) ->
    case ds_d7Kk of _ { MkEmployee name_a6pt age_a6pu ->
    fmap
      @ f_a6t0
      $dFunctor_a6uA
      @ Int
      @ Employee
      (\ (newAge_a6pv :: Int) -> MkEmployee name_a6pt newAge_a6pv)
      (f1_a6ps age_a6pu)
    }
```

Generic code:

```
ageLensGeneric1
  :: forall x_X7NQ (f1_X7NS :: * -> *).
     Functor f1_X7NS =>
     (Int -> f1_X7NS Int)
     -> M1
          D
          ('MetaData "Employee" "GenericLens" "main" 'False)
          (M1
             C
             ('MetaCons "MkEmployee" 'PrefixI 'False)
             (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSo
          x_X7NQ
     -> f1_X7NS (M1
                   D
                   ('MetaData "Employee" "GenericLens" "main" 'Fa
                   (M1
                      C
                      ('MetaCons "MkEmployee" 'PrefixI 'False)
                      (S1 ('MetaSel 'Nothing 'NoSourceUnpackedn
                   x_X7NQ)
ageLensGeneric1 =
  \ (@ x_X7NQ) (@ (f1_X7NS :: * -> *)) ($dFunctor_X7NU :: Funct
    let {
      f2_a7MF
        :: f1_X7NS (M1
                      C
                      ('MetaCons "MkEmployee" 'PrefixI 'False)
```

A promise broken

Manual code:

```
ageLensManual :: Lens' Employee Int
ageLensManual =
  \ (g :: a$10 -> f_a$10 a$10)
    ($dFunctor_a$uA :: Functor f_a$10)
    (f1_a$px :: Int -> f_a$10 Int)
    (ds_d7KA :: Employee) ->
    case ds_d7KA of _ { MkEmployee name_a$gt age_a$gu ->
    fmap
      @ f_a$10
      $dFunctor_a$uA
      @ Int
      @ Employee
      (\ (newAge_a$pv :: Int) -> MkEmployee name_a$gt newAge_a$pv)
      (f1_a$px age_a$gu)
    }
```

Generic code:

```
ageLensGeneric1
  :: forall x_K7NQ (f1_K7NS :: * -> *).
     Functor f1_K7NS =>
     (Int -> f1_K7NS Int)
     -> M1
          D
          ('MetaData "Employee" "GenericLens" "main" 'False)
          (M1
             C
             ('MetaCons "MkEmployee" 'PrefixI 'False)
             (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))
          x_K7NQ
     -> f1_K7NS (M1
                   D
                   ('MetaData "Employee" "GenericLens" "main" 'False)
                   (M1
                      C
                      ('MetaCons "MkEmployee" 'PrefixI 'False)
                      (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))
                   x_K7NQ)
ageLensGeneric1 =
  \ (@ x_K7NQ) (@ (f1_K7NS :: * -> *)) ($dFunctor_K7NU :: Functor f1_K7NS) ->
    let {
      f2_a7MF
        :: f1_K7NS (M1
                      C
                      ('MetaCons "MkEmployee" 'PrefixI 'False)
                      (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int))
                      x_K7NQ)
           -> f1_K7NS (M1
                         D
                         ('MetaData "Employee" "GenericLens" "main" 'False)
                         (M1
                            C
                            ('MetaCons "MkEmployee" 'PrefixI 'False)
                            (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))
                         x_K7NQ)
      f2_a7MF =
        fmap
          @ f1_K7NS
          $dFunctor_K7NU
          @ (M1
               C
               ('MetaCons "MkEmployee" 'PrefixI 'False)
               (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))
               x_K7NQ)
          (f2_a7MF
             fmap
               @ f1_K7NS
               $dFunctor_K7NU
               @ (M1
                    D
                    ('MetaData "Employee" "GenericLens" "main" 'False)
                    (M1
                       C
                       ('MetaCons "MkEmployee" 'PrefixI 'False)
                       (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))
                    x_K7NQ)
               (\ (m1s1
                     @ (M1
                          C
                          ('MetaCons "MkEmployee" 'PrefixI 'False)
                          (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))
                          x_K7NQ)
                  `cast` ...) } in
    let {
      f4_K7Nw
        :: f1_K7NS ((:+:)
                      (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String)) (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)) x_K7NQ)
           -> f1_K7NS (M1
                         C
                         ('MetaCons "MkEmployee" 'PrefixI 'False)
                         (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))
                         x_K7NQ)
      f4_K7Nw =
        fmap
          @ f1_K7NS
          $dFunctor_K7NU
          @ ((:+:)
               (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String)) (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)) x_K7NQ)
          @ (M1
               C
               ('MetaCons "MkEmployee" 'PrefixI 'False)
               (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))
               x_K7NQ)
          ((m1s1
              @ (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)) @ x_K7NQ)
           `cast` ...) } in
    let {
      f5_K7Nx :: f1_K7NS (K1 R Int x_K7NQ) -> f1_K7NS (M1 S ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (K1 R Int) x_K7NQ)
      f5_K7Nx =
        fmap @ f1_K7NS $dFunctor_K7NU @ (K1 R Int x_K7NQ) @ (M1 S ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (K1 R Int) x_K7NQ) ((m1s1 @ (K1 R Int) @ x_K7NQ) `cast` ...) } in
    \ (x1_K7Nt :: Int -> f1_K7NS Int)
      (x2_a7MH
         :: M1
              D
              ('MetaData "Employee" "GenericLens" "main" 'False)
              (M1
                 C
                 ('MetaCons "MkEmployee" 'PrefixI 'False)
                 (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :+: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))
              x_K7NQ) ->
      f2_a7MF
        ((f4_K7Nw
            (case x2_a7MH `cast` ... of _ { :+: a1_a7N7 b1_a7N8 ->
             fmap
               @ f1_K7NS
               $dFunctor_K7NU
               @ (M1 S ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int) x_K7NQ)
               @ ((:+:)
                    (M1 S ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String)) (M1 S ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)) x_K7NQ)
               (:+:
                  @ (M1 S ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String))
                  @ (M1 S ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int))
                  @ x_K7NQ
                  a1_a7N7)
               (f5_K7Nt (fmap @ f1_K7NS $dFunctor_K7NU @ Int @ (K1 R Int x_K7NQ) ((bfGHowTypeHla1 @ Int) `cast` ...) (x1_K7Nt (b1_a7N8 `cast` ...))))
             }))

--- RHS size: terms: 0, types:
0, coercions: 4
ageLensGeneric :: Lens' Employee Int
ageLensGeneric = \ (@ (f_a6a0 :: * -> *)) ($dFunctor_a6t1 :: Functor f_a6a0) -> $wdctyped @ Employee @ Int $fGenericEmployee (ageLensGeneric1 `cast` ...) @ f_a6a0 $dFunctor_a6t1
```

A promise broken

```
proj :: Functor f => f a -> Coyoneda f a
proj fa = Coyoneda id fa

ravel :: Functor f => ((a -> Coyoneda f b) -> (s -> Coyoneda f t))
                   -> (a -> f b) -> (s -> f t)
ravel coy f s = inj $ coy (\a -> proj (f a)) s
```

mpickering commented on 3 Sep — Collaborator

Hi Joachim,

After making such a claim in the talk, I tried it out again and indeed with the most recent version of the library the situation is worse and the generated core is different. I just tried it again with an older version, and I see what is claimed with the paper.

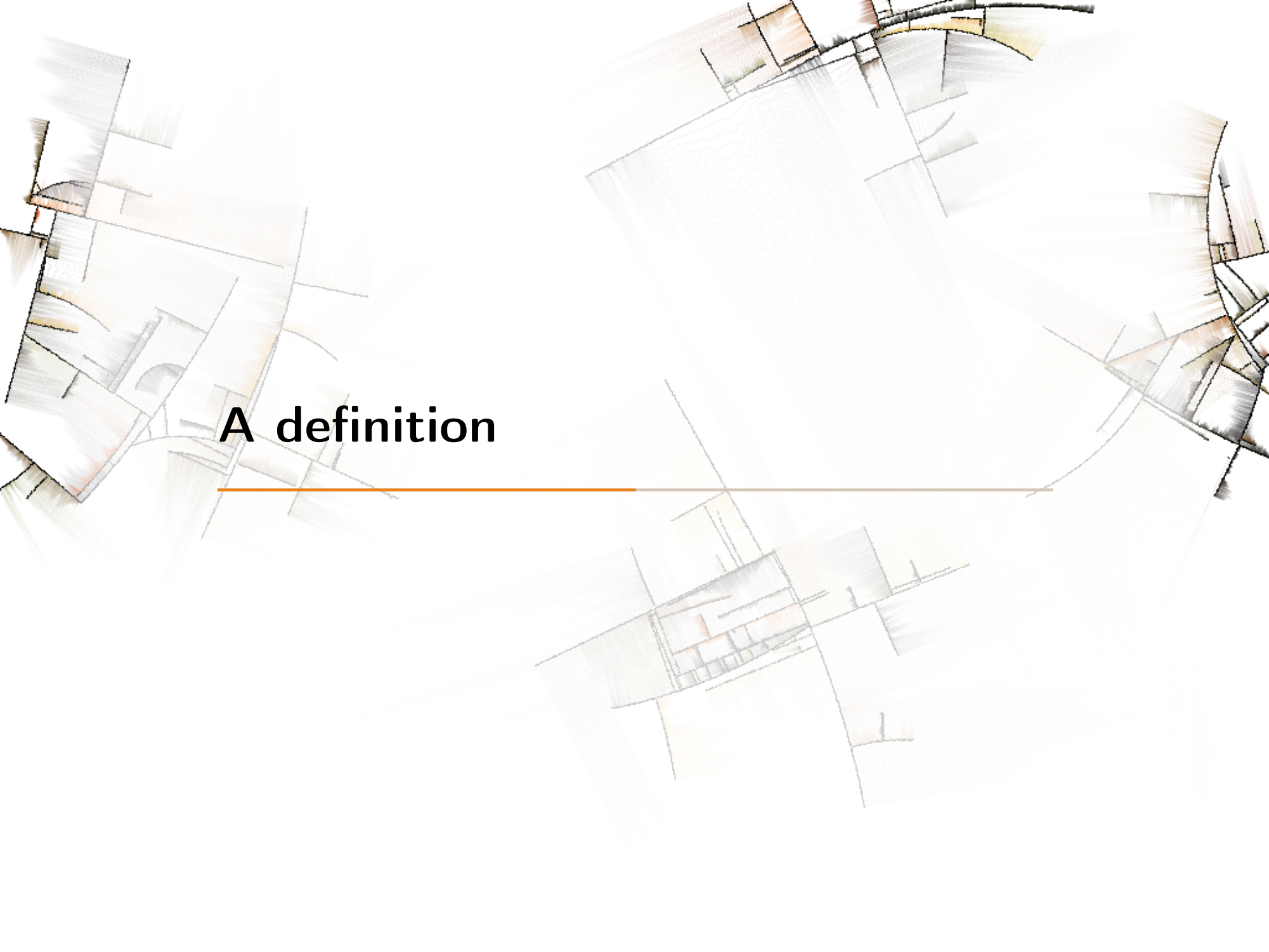
@kcsongor What have you changed since about the 0.2 version? The precise commit is in the paper repo.

kcsongor commented on 3 Sep — Owner

I'm investigating this regression and will try to get back later today

nomeata commented on 3 Sep — Contributor

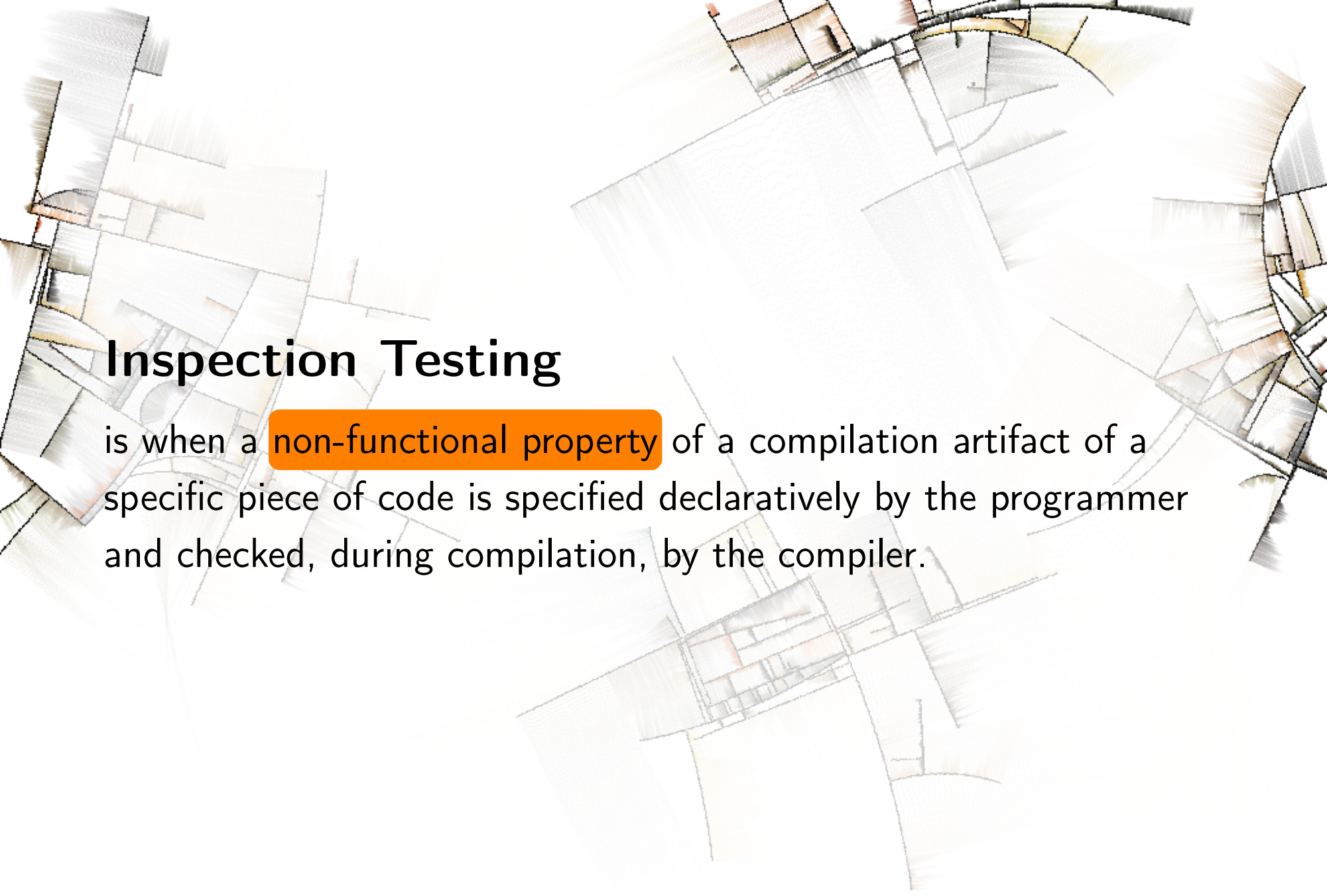
Heh. At least a good story for me to advocate for “optimization unit tests” that I envision :-)



A definition

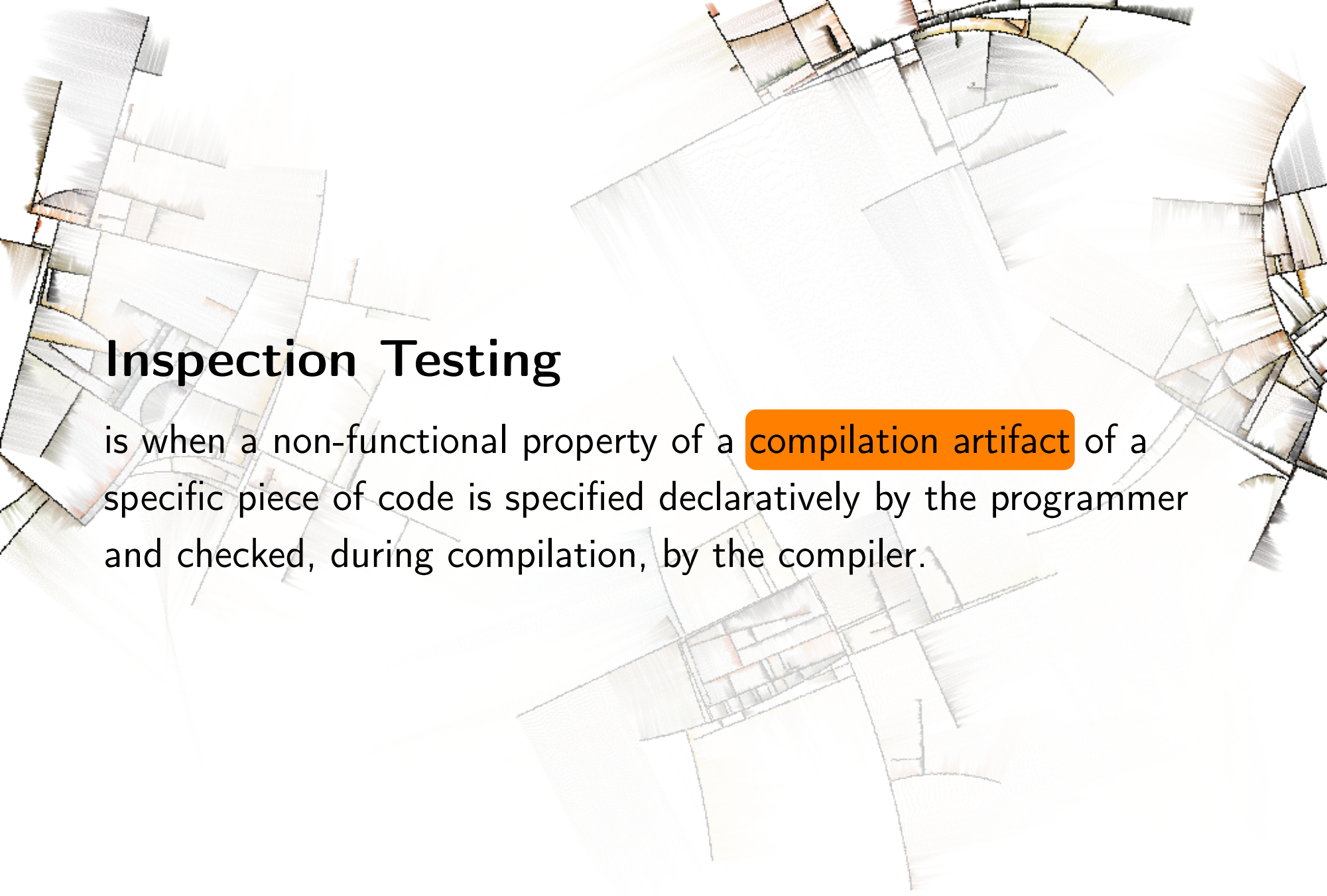
Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.



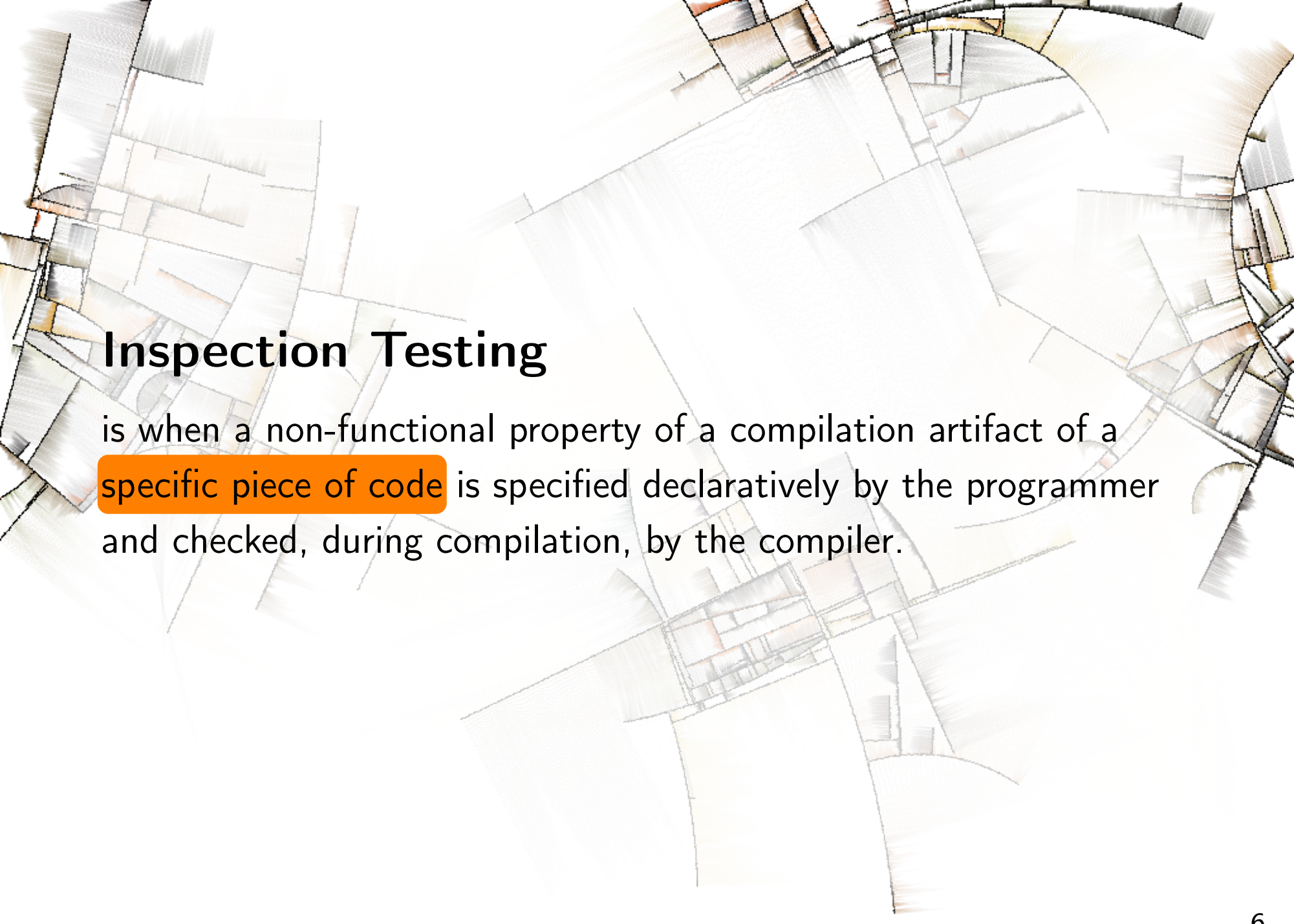
Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.



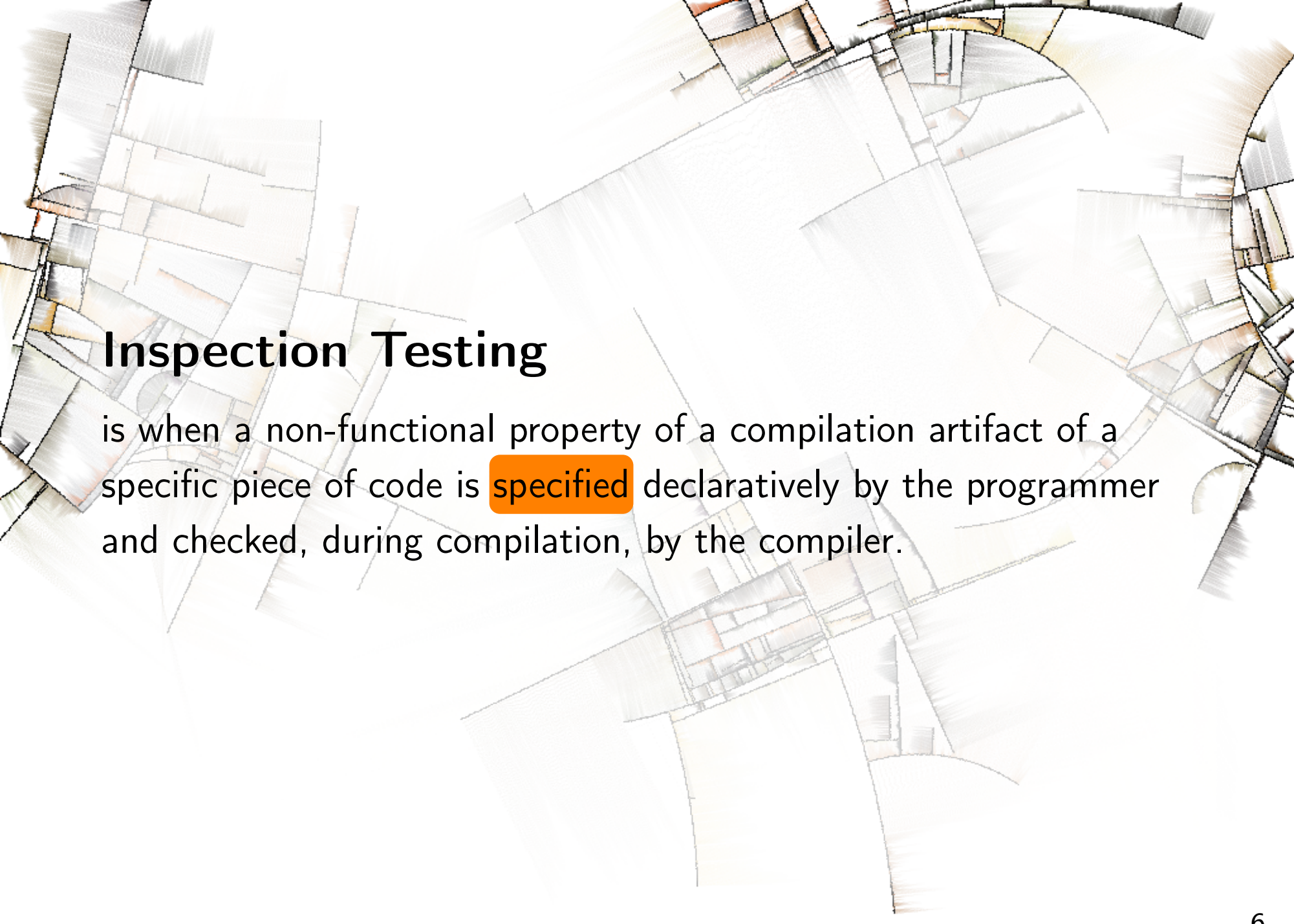
Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.



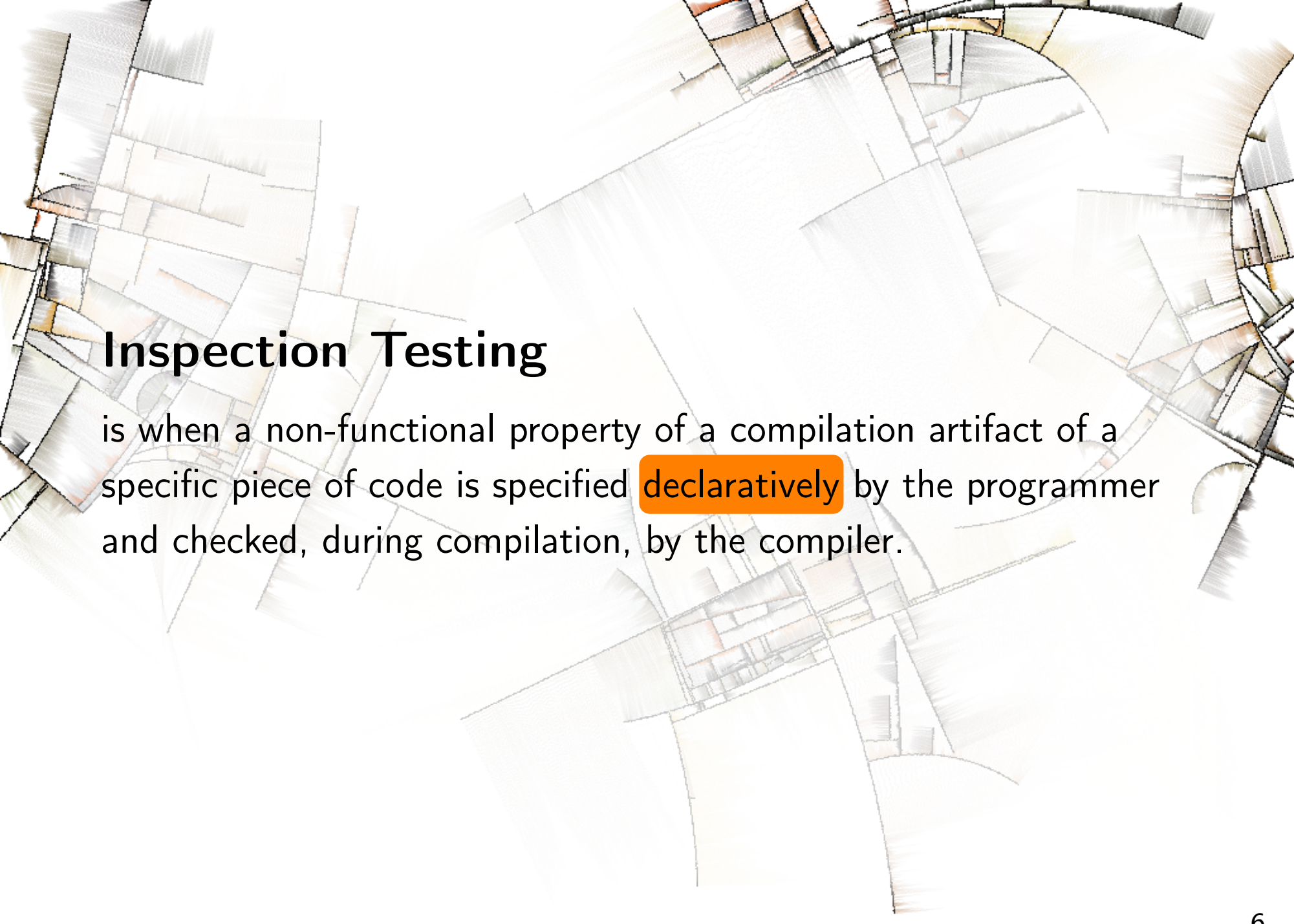
Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.



Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.



Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

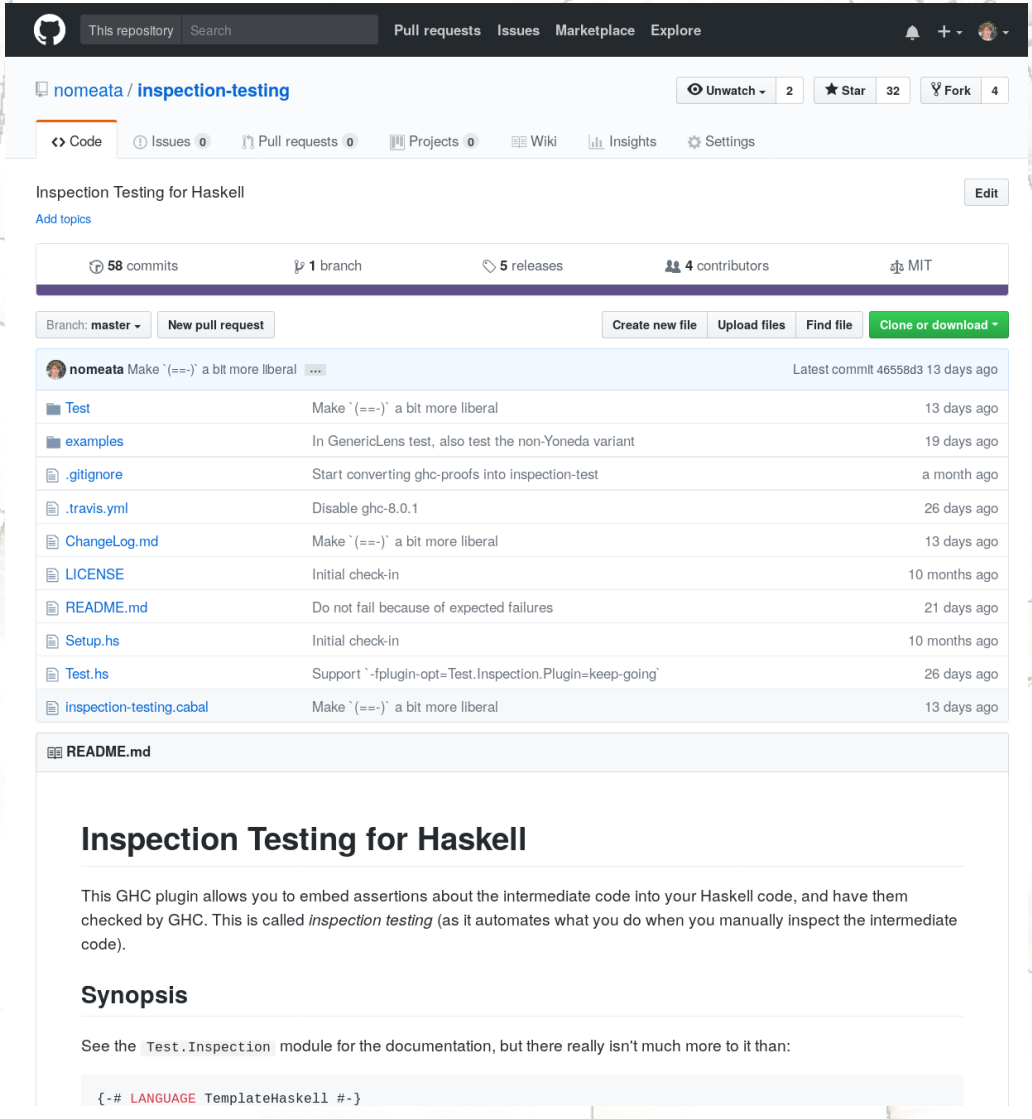
Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

Haskell Implementation



generic-lens: Happy end

```haskell
-- ------------------------------------------------------------------------
-- * Tests
-- The inspection-testing plugin checks that the following equalities hold, by
-- checking that the LHSs and the RHSs are CSEd. This also means that the
-- runtime characteristics of the derived lenses is the same as the manually
-- written ones above.

fieldALensName :: Lens' Record Int
fieldALensName = field @"fieldA"

fieldALensType :: Lens' Record Int
fieldALensType = typed @Int

fieldALensPos :: Lens' Record Int
fieldALensPos = position @1

subtypeLensGeneric :: Lens' Record Record2
subtypeLensGeneric = super

typeChangingGeneric :: Lens (Record3 a) (Record3 b) a b
typeChangingGeneric = field @"fieldA"

typeChangingGenericPos :: Lens (Record3 a) (Record3 b) a b
typeChangingGenericPos = position @1

typeChangingGenericCompose :: Lens (Record3 (Record3 a)) (Record3 (Record3 b)) a b
typeChangingGenericCompose = field @"fieldA" . field @"fieldA"

inspect $ 'fieldALensManual === 'fieldALensName
inspect $ 'fieldALensManual === 'fieldALensType
inspect $ 'fieldALensManual === 'fieldALensPos
inspect $ 'subtypeLensManual === 'subtypeLensGeneric
inspect $ 'typeChangingManual === 'typeChangingGeneric
inspect $ 'typeChangingManual === 'typeChangingGenericPos
inspect $ 'typeChangingManualCompose === 'typeChangingGenericCompose
```

Already used by:

- generic-lens
- generic-sop
- vec

Many applications

- Equality of generic vs. manual code.
- Equivalence of generic vs. manual code.
- Elimination of intermediate data structures (fusion)
- Strictness/laziness properties
- Absence of allocations
- Absence of slow function calls
- Absence of branches
- Vectorization and SIMD
- *insert more good ideas here*


Thank you