Names are not type safety

2020-11-01 haskell, types, functional programming

Haskell programmers spend a lot of time talking about *type safety*. The Haskell school of program construction advocates "capturing invariants in the type system" and "making illegal states unrepresentable," both of which sound like compelling goals, but are rather vague on the techniques used to achieve them. Almost exactly one year ago, I published Parse, Don't Validate as an initial stab towards bridging that gap.

The ensuing discussions were largely productive and right-minded, but one particular source of confusion quickly became clear: Haskell's `newtype` construct. The idea is simple enough—the `newtype` keyword declares a wrapper type, nominally distinct from but representationally equivalent to the type it wraps—and on the surface this *sounds* like a simple and straightforward path to type safety. For example, one might consider using a `newtype` declaration to define a type for an email address:

```haskell
newtype EmailAddress = EmailAddress Text
```

This technique can provide *some* value, and when coupled with a smart constructor and an encapsulation boundary, it can even provide some safety. But it is a meaningfully distinct *kind* of type safety from the one I highlighted a year ago, one that is far weaker. On its own, a newtype is just a name.

And names are not type safety.

Intrinsic and extrinsic safety

To illustrate the difference between constructive data modeling (discussed at length in my previous blog post) and newtype wrappers, let's consider an example. Suppose we want a type for "an integer between 1 and 5, inclusive." The natural constructive modeling would be an enumeration with five cases:

```
data OneToFive
  = One
  | Two
  | Three
  | Four
  | Five
```

We could then write some functions to convert between `Int` and our `OneToFive` type:

```
toOneToFive :: Int -> Maybe OneToFive
toOneToFive 1 = Just One
toOneToFive 2 = Just Two
toOneToFive 3 = Just Three
toOneToFive 4 = Just Four
toOneToFive 5 = Just Five
toOneToFive _ = Nothing

fromOneToFive :: OneToFive -> Int
fromOneToFive One   = 1
fromOneToFive Two   = 2
fromOneToFive Three = 3
fromOneToFive Four  = 4
fromOneToFive Five  = 5
```

This would be perfectly sufficient for achieving our stated goal, but you'd be forgiven for finding it odd: it would be rather awkward to work with in practice. Because we've invented an entirely new type, we can't reuse any of the usual numeric functions Haskell provides. Consequently, many programmers would gravitate towards a newtype wrapper, instead:

```
newtype OneToFive = OneToFive Int
```

Just as before, we can provide `toOneToFive` and `fromOneToFive` functions, with identical types:

```
toOneToFive :: Int -> Maybe OneToFive
toOneToFive n
  | n >= 1 && n <= 5 = Just $ OneToFive n
  | otherwise        = Nothing

fromOneToFive :: OneToFive -> Int
fromOneToFive (OneToFive n) = n
```

If we put these declarations in their own module and choose not to export the `OneToFive` constructor, these APIs might appear entirely interchangeable. Naïvely, it seems that the newtype version is both simpler and equally type-safe. However—perhaps surprisingly—this is not actually true.

To see why, suppose we write a function that consumes a `OneToFive` value as an argument. Under the constructive modeling, such a function need only patternmatch against each of the five constructors, and GHC will accept the definition as exhaustive:

```
ordinal :: OneToFive -> Text
ordinal One   = "first"
ordinal Two   = "second"
ordinal Three = "third"
ordinal Four  = "fourth"
ordinal Five  = "fifth"
```

The same is not true given the newtype encoding. The newtype is opaque, so the only way to observe it is to convert it back to an `Int` —after all, it *is* an `Int`. An `Int` can of course contain many other values besides `1` through `5`, so we are forced to add an error case to satisfy the exhaustiveness checker:

```
ordinal :: OneToFive -> Text
ordinal n = case fromOneToFive n of
  1 -> "first"
  2 -> "second"
  3 -> "third"
  4 -> "fourth"
  5 -> "fifth"
  _ -> error "impossible: bad OneToFive value"
```

In this highly contrived example, this may not seem like much of a problem to you. But it nonetheless illustrates a key difference in the guarantees afforded by the two approaches:

- The constructive datatype captures its invariants in such a way that they are *accessible* to downstream consumers. This frees our `ordinal` function from worrying about handling illegal values, as they have been made unutterable.

- The newtype wrapper provides a smart constructor that *validates* the value, but the boolean result of that check is used only for control flow; it is not preserved in the function's result. Accordingly, downstream consumers cannot take advantage of the restricted domain; they are functionally accepting `Int` s.

Losing exhaustiveness checking might seem like small potatoes, but it absolutely is not: our use of `error` has punched a hole right through our type system. If we were to add another constructor to our `OneToFive` datatype,[1] the version of `ordinal` that consumes a constructive datatype would be immediately detected non-exhaustive at compile-time, while the version that consumes a newtype wrapper would continue to compile yet fail at runtime, dropping through to the "impossible" case.

All of this is a consequence of the fact that the constructive modeling is *intrinsically* type-safe; that is, the safety properties are enforced by the type declaration itself. Illegal values truly are unrepresentable: there is simply no way to represent `6` using any of the five constructors. The same is not true of the newtype declaration, which has no intrinsic semantic distinction from that of an `Int` ; its meaning is specified extrinsically via the `toOneToFive` smart constructor. Any semantic distinction intended by a newtype is thoroughly invisible to the type system; it exists only in the programmer's mind.

Revisiting non-empty lists

Our `OneToFive` datatype is rather artificial, but identical reasoning applies to other datatypes that are significantly more practical. Consider the `NonEmpty` datatype I've repeatedly highlighted in recent blog posts:

```
data NonEmpty a = a :| [a]
```

It may be illustrative to imagine a version of `NonEmpty` represented as a newtype over ordinary lists. We can use the usual smart constructor strategy to enforce the desired non-emptiness property:

```
newtype NonEmpty a = NonEmpty [a]

nonEmpty :: [a] -> Maybe (NonEmpty a)
nonEmpty [] = Nothing
nonEmpty xs = Just $ NonEmpty xs

instance Foldable NonEmpty where
  toList (NonEmpty xs) = xs
```

Just as with `OneToFive`, we quickly discover the consequences of failing to preserve this information in the type system. Our motivating use case for `NonEmpty` was the ability to write a safe version of `head`, but the newtype version requires another assertion:

```
head :: NonEmpty a -> a
head xs = case toList xs of
  x:_ -> x
  []  -> error "impossible: empty NonEmpty value"
```

This might not seem like a big deal, since it seems unlikely such a case would ever happen. But that reasoning hinges entirely on trusting the correctness of the module that defines `NonEmpty`, while the constructive definition only requires trusting the

GHC typechecker. As we generally trust that the typechecker works correctly, the latter is a much more compelling proof.

Newtypes as tokens

If you are fond of newtypes, this whole argument may seem a bit troubling. It may seem like I'm implying newtypes are scarcely better than comments, albeit comments that happen to be meaningful to the typechecker. Fortunately, the situation is not quite that grim—newtypes *can* provide a sort of safety, just a weaker one.

The primary safety benefit of newtypes is derived from abstraction boundaries. If a newtype's constructor is not exported, it becomes opaque to other modules. The module that defines the newtype—its "home module"—can take advantage of this to create a *trust boundary* where internal invariants are enforced by restricting clients to a safe API.

We can use the `NonEmpty` example from above to illustrate how this works. We refrain from exporting the `NonEmpty` constructor, and we provide `head` and `tail` operations that we trust to never actually fail:

```haskell
module Data.List.NonEmpty.Newtype
  ( NonEmpty
  , cons
  , nonEmpty
  , head
  , tail
  ) where

newtype NonEmpty a = NonEmpty [a]
```

```
cons :: a -> [a] -> NonEmpty a
cons x xs = NonEmpty (x:xs)

nonEmpty :: [a] -> Maybe (NonEmpty a)
nonEmpty [] = Nothing
nonEmpty xs = Just $ NonEmpty xs

head :: NonEmpty a -> a
head (NonEmpty (x:_)) = x
head (NonEmpty [])    = error "impossible: empty NonEmpty value"

tail :: NonEmpty a -> [a]
tail (NonEmpty (_:xs)) = xs
tail (NonEmpty [])     = error "impossible: empty NonEmpty value"
```

Since the only way to construct or consume `NonEmpty` values is to use the functions in `Data.List.NonEmpty.Newtype`'s exported API, the above implementation makes it impossible for clients to violate the non-emptiness invariant. In a sense, values of opaque newtypes are like *tokens*: the implementing module issues tokens via its constructor functions, and those tokens have no intrinsic value. The only way to do anything useful with them is to "redeem" them to the issuing module's accessor functions, in this case `head` and `tail`, to obtain the values contained within.

This approach is significantly weaker than using a constructive datatype, since it is theoretically possible to screw up and accidentally provide a means to construct an invalid `NonEmpty []` value. For this reason, the newtype approach to type safety does not on its own constitute a *proof* that a desired invariant holds. However, it restricts the "surface area" where an invariant violation can occur to the defining module, so reasonable confidence the invariant really does hold can be achieved by thoroughly testing the module's API using fuzzing or property-based testing techniques.[2]

This tradeoff may not seem all that bad, and indeed, it is often a very good one! Guaranteeing invariants using constructive data modeling can, in general, be quite difficult, which often makes it impractical. However, it is easy to dramatically underestimate the care needed to avoid accidentally providing a mechanism that permits violating the invariant. For example, the programmer may choose to take advantage of GHC's convenient typeclass deriving to derive a `Generic` instance for `NonEmpty`:

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics (Generic)

newtype NonEmpty a = NonEmpty [a]
  deriving (Generic)
```

However, this innocuous line provides a trivial mechanism to circumvent the abstraction boundary:

```
ghci> GHC.Generics.to @(NonEmpty ()) (M1 $ M1 $ M1 $ K1 [])
NonEmpty []
```

This is a particularly extreme example, since derived `Generic` instances are fundamentally abstraction-breaking, but this problem can crop up in less obvious ways, too. The same problem occurs with a derived `Read` instance:

```
ghci> read @(NonEmpty ()) "NonEmpty []"
NonEmpty []
```

To some readers, these pitfalls may seem obvious, but safety holes of this sort are remarkably common in practice. This is especially true for datatypes with more sophisticated invariants, as it may not be easy to determine whether the invariants are actually upheld by the module's implementation. Proper use of this technique demands caution and care:

- All invariants must be made clear to maintainers of the trusted module. For simple types, such as `NonEmpty`, the invariant is self-evident, but for more sophisticated types, comments are not optional.

- Every change to the trusted module must be carefully audited to ensure it does not somehow weaken the desired invariants.

- Discipline is needed to resist the temptation to add unsafe trapdoors that allow compromising the invariants if used incorrectly.

- Periodic refactoring may be needed to ensure the trusted surface area remains small. It is all too easy for the responsibility of the trusted module to accumulate over time, dramatically increasing the likelihood of some subtle interaction causing an invariant violation.

In contrast, datatypes that are correct by construction suffer none of these problems. The invariant cannot be violated without changing the datatype definition itself, which has rippling effects throughout the rest of the program to make the consequences immediately clear. Discipline on the part of the programmer is unnecessary, as the typechecker enforces the invariants automatically. There is no "trusted code" for such datatypes, since all parts of the program are equally beholden to the datatype-mandated constraints.

In libraries, the newtype-afforded notion of safety via encapsulation is useful, as libraries often provide the building blocks used to construct more complicated data structures. Such libraries generally receive more scrutiny and care than application code does, especially given they change far less frequently. In application code, these techniques are still useful, but the churn of a production codebase tends to weaken encapsulation boundaries over time, so correctness by construction should be preferred whenever practical.

Other newtype use, abuse, and misuse

The previous section covers the primary means by which newtypes are useful. However, in practice, newtypes are routinely used in ways that do not fit the above pattern. Some such uses are reasonable:

- Haskell's notion of typeclass coherency limits each type to a single instance of any given class. For types that permit more than one useful instance, newtypes are the traditional solution, and this can be used to good effect. For example, the `Sum` and `Product` newtypes from `Data.Monoid` provide useful `Monoid` instances for numeric types.

- In a similar vein, newtypes can be useful for introducing or rearranging type parameters. The `Flip` newtype from `Data.Bifunctor.Flip` is a simple example, flipping the arguments of a `Bifunctor` so the `Functor` instance may operate on the other side:

```haskell
newtype Flip p a b = Flip { runFlip :: p b a }
```

Newtypes are needed to do this sort of juggling, as Haskell does not (yet) support type-level lambdas.

- More simply, transparent newtypes can be useful to discourage misuse when the value needs to be passed between distant parts of the program and the intermediate code has no reason to inspect the value. For example, a `ByteString` containing a secret key may be wrapped in a newtype (with a `Show` instance omitted) to discourage code from accidentally logging or otherwise exposing it.

All of these applications are good ones, but they have little to do with *type safety.* The last bullet in particular is often confused for safety, and to be fair, it does in fact take advantage of the type system to help avoid logical mistakes. However, it would be a mischaracterization to claim such usage actually *prevents* misuse; any part of the program may inspect the value at any time.

Too often, this illusion of safety leads to outright newtype abuse. For example, here's a definition from the very codebase I work on for a living:

```haskell
newtype ArgumentName = ArgumentName { unArgumentName :: GraphQL.Name }
  deriving ( Show, Eq, FromJSON, ToJSON, FromJSONKey, ToJSONKey
           , Hashable, ToTxt, Lift, Generic, NFData, Cacheable )
```

This newtype is useless noise. Functionally, it is completely interchangeable with its underlying `Name` type, so much so that it derives a dozen typeclasses! In every location it's used, it's immediately unwrapped the instant it's extracted from its enclosing record, so there is no type safety benefit whatsoever. Worse, there isn't even any clarity added by labeling it an `ArgumentName`, since the enclosing field name already makes its role clear.

Newtypes like these seem to arise from a desire to use the type system as a taxonomy of the external world. An "argument name" is a more specific concept than a generic "name," so surely it ought to have its own type. This makes some intuitive sense, but it's rather misguided: taxonomies are useful for documenting a domain of interest, but not necessarily helpful for modeling it. When programming, we use types for a different end:

- Primarily, types distinguish *functional* differences between values. A value of type `NonEmpty a` is *functionally* distinct from a value of type `[a]`, since it is fundamentally structurally different and permits additional operations. In this sense, types are *structural*; they describe what values *are* in the internal world of the programming language.

- Secondarily, we sometimes use types to help ourselves avoid making logical mistakes. We might use separate `Distance` and `Duration` types to avoid accidentally doing something nonsensical like adding them together, even though they're both representationally real numbers.

Note that both these uses are *pragmatic*; they look at the type system as a tool. This is a rather natural perspective to take, seeing as a static type system *is* a tool in a literal sense. Nevertheless, that perspective seems surprisingly unusual, even though the use of types to classify the world routinely yields unhelpful noise like `ArgumentName`.

If a newtype is completely transparent, and it is routinely wrapped and unwrapped at will, it is likely not very helpful. In this particular case, I would eliminate the distinction altogether and use `Name`, but in situations where the different label adds genuine clarity, one can always use a type alias:[3]

```haskell
  type ArgumentName = GraphQL.Name
```

Newtypes like these are security blankets. Forcing programmers to jump through a few hoops is not type safety—trust me when I say they will happily jump through them without a second thought.

Final thoughts and related reading

I've been wanting to write this blog post for a long time. Ostensibly, it's a very specific critique of Haskell newtypes, and I've chosen to frame things this way because I write Haskell for a living and this is the way I encounter this problem in practice. Really, though, the core idea is much bigger than that.

Newtypes are one particular mechanism of defining *wrapper types*, a concept that exists in almost any language, even those that are dynamically typed. Even if you don't write Haskell, much of the reasoning in this blog post is likely still relevant in your language of choice. More broadly, this is a continuation of a theme I've been trying to convey from different angles over the past year: type systems are tools, and we should be more conscious and intentional about what they actually do and how to use them effectively.

The catalyst that got me to finally sit down and write this was the recently-published Tagged is not a Newtype. It's a good blog post, and I wholeheartedly agree with its general thrust, but I thought it was a missed opportunity to make a larger point. Indeed, `Tagged` *is* a newtype, definitionally, so the title of the blog post is something of a misdirection. The real problem is a little deeper.

Newtypes are useful when carefully applied, but their safety is not intrinsic, no more than the safety of a traffic cone is somehow contained within the plastic it’s made of. What matters is being placed in the right context—without that, newtypes are just a labeling scheme, a way of giving something a name.

And a name is not type safety.

1. Admittedly rather unlikely given its name, but bear with me through the contrived example. ↩

2. In theory, it is still possible to thoroughly prove the invariant holds using external verification techniques, such as by writing a pen-and-paper proof or by using program extraction in combination with a proof assistant/theorem prover. However, these techniques are extremely uncommon in general programming practice. ↩

3. As it happens, I think type aliases are often also more harmful than helpful, so I would caution against overusing them, too, but that is outside the scope of this blog post. ↩

← An introduction to typeclass metaprogramming

Types as axioms, or: playing god with static types →