Setting the Record Straight with Singletons

by Reed Mullanix / University of Minnesota / WITS '22

on January 22, 2022

» Definitions

* A large part of the praxis of using a proof assistant effectively is the careful choice of definitions.

» Definitions

- A large part of the praxis of using a proof assistant effectively is the careful choice of definitions.
- In fact, there often isn't a single "correct" choice!

» Definitions

- A large part of the praxis of using a proof assistant effectively is the careful choice of definitions.
- In fact, there often isn't a single "correct" choice!
- One way this manifests is when we try to define an algebraic heirarchy.

» Bundled and Unbundled Structures

As a case study, let's look at 2 possible defintions of a Monoid:

» Bundled and Unbundled Structures

As a case study, let's look at 2 possible defintions of a Monoid:

* The first "bundles up" the carrier type inside of the structure.

$$\mathit{Monoid} = \mathit{record}\ \{X : \mathcal{U}, \circ : X \times X \to X, \ldots\}$$

» Bundled and Unbundled Structures

As a case study, let's look at 2 possible defintions of a Monoid:

* The first "bundles up" the carrier type inside of the structure.

$$\mathit{Monoid} = \mathit{record}\ \{X : \mathcal{U}, \circ : X \times X \to X, \ldots\}$$

* On the other hand, we could parameterize over all the relevant data!

$$\mathit{Monoid} = \forall (X : \mathcal{U})(\circ : X \times X \to X)(e : X),\ \mathit{isMonoid}(X, \circ, e)$$

» Bundled and Unbundled Structures

As a case study, let's look at 2 possible defintions of a Monoid:

* The first "bundles up" the carrier type inside of the structure.

$$\mathit{Monoid} = \mathit{record}\ \{X : \mathcal{U}, \circ : X \times X \to X, \ldots\}$$

* On the other hand, we could parameterize over all the relevant data!

$$\mathit{Monoid} = \forall (X : \mathcal{U})(\circ : X \times X \to X)(e : X),\ \mathit{isMonoid}(X, \circ, e)$$

The parameterized version lets us (defintionally) constrain some components of the structure, and the bundled version lets us more easily talk about the type of *all* monoids.

» Bundled and Unbundled Structures

* To use the parameterized definition, we need to have unification and implicits to have any hope.

» Bundled and Unbundled Structures

* To use the parameterized definition, we need to have unification and implicits to have any hope.
* However, unification is *hard*.

» Bundled and Unbundled Structures

- To use the parameterized definition, we need to have unification and implicits to have any hope.
- However, unification is *hard*.
- We've managed to avoid writing a unifier in `cooltt` so far, can we keep going?

» Bundled and Unbundled Structures

- To use the parameterized definition, we need to have unification and implicits to have any hope.
- However, unification is *hard*.
- We've managed to avoid writing a unifier in `cooltt` so far, can we keep going? (Spoiler: the answer is yes!)

» Singleton Types

What unification buys is is the ability to specify that two structures are defined on the *same* data.

» Singleton Types

What unification buys is is the ability to specify that two structures are defined on the *same* data. What if we could do that another way?

» Singleton Types

What unification buys is is the ability to specify that two structures are defined on the *same* data. What if we could do that another way?

* Enter singleton types!

» Singleton Types

What unification buys is is the ability to specify that two structures are defined on the *same* data. What if we could do that another way?

- Enter singleton types!
- Singleton types allow us to quantify over terms that are determined up to *definitional* equality.

» Singleton Types

What unification buys is is the ability to specify that two structures are defined on the *same* data. What if we could do that another way?

- Enter singleton types!
- Singleton types allow us to quantify over terms that are determined up to *definitional* equality.
- $[\, \mathbb{N} \mid 4 \,]$, $[\, \mathcal{U} \mid \mathit{Bool} \to \mathit{Bool} \,]$, *etc...*

» Singleton Types

What unification buys is is the ability to specify that two structures are defined on the *same* data. What if we could do that another way?

- Enter singleton types!
- Singleton types allow us to quantify over terms that are determined up to *definitional* equality.
- $[\, \mathbb{N} \mid 4 \,]$, $[\, \mathcal{U} \mid \mathit{Bool} \to \mathit{Bool} \,]$, *etc...*
- Typechecking these is easy with NbE!
- (In fact, we already have a more general form of type in `cooltt` called an extension type for various cubical reasons).

» Elaborating Records

Armed with our new types, we eliminate the need parameterized representation!

* We add some syntactic sugar for elaborating a “patched record”

» Elaborating Records

Armed with our new types, we eliminate the need parameterized representation!

- We add some syntactic sugar for elaborating a "patched record"
- $M : \mathit{Monoid} \vdash M\,[\ \mathit{Carrier} := \mathbb{N},\ \circ := +,\ \ldots\ ]$

» Elaborating Records

Armed with our new types, we eliminate the need parameterized representation!

- We add some syntactic sugar for elaborating a "patched record"
- $M : \mathit{Monoid} \vdash M\,[\ \mathit{Carrier} \mathrel{.=} \mathbb{N},\ \circ \mathrel{.=} +,\ \ldots\ ]$
- This elaborates out to a record type with singleton types constraining the relevant fields.

» Elaborating Records

Armed with our new types, we eliminate the need parameterized representation!

- We add some syntactic sugar for elaborating a "patched record"
- $M : \mathit{Monoid} \vdash M\,[\ \mathit{Carrier} \mathrel{.=} \mathbb{N},\ \circ \mathrel{.=} +,\ \ldots\ ]$
- This elaborates out to a record type with singleton types constraining the relevant fields.
- We then always work with the bundled representation, and use these "patches" in place of the parameterized version.

» Unification Strikes Back!

However, this is only half the story!

* We often want to define data as a family of types.

» Unification Strikes Back!

However, this is only half the story!

* We often want to define data as a family of types.
* For instance, the type $hom : obj \to obj \to \mathcal{U}$ for a category.

» Unification Strikes Back!

However, this is only half the story!

- We often want to define data as a family of types.
- For instance, the type $hom : obj \to obj \to \mathcal{U}$ for a category.
- When defining any operations involving multiple *homs*, we end up with a huge explosion of arguments! As an example, *assoc* would need 4 redundant arguments!

» Unification Strikes Back!

However, this is only half the story!

* We often want to define data as a family of types.
* For instance, the type $hom : obj \to obj \to \mathcal{U}$ for a category.
* When defining any operations involving multiple *homs*, we end up with a huge explosion of arguments! As an example, *assoc* would need 4 redundant arguments!
* This is normally handled by adding implicit arguments, but that requires unification! What are we to do?

» Unification Strikes Back!

However, this is only half the story!

- We often want to define data as a family of types.
- For instance, the type $hom : obj \to obj \to \mathcal{U}$ for a category.
- When defining any operations involving multiple *homs*, we end up with a huge explosion of arguments! As an example, *assoc* would need 4 redundant arguments!
- This is normally handled by adding implicit arguments, but that requires unification! What are we to do?
- Hmm, that family looks a lot like some sort of parameterized representation...

» Automatic Bundling

If we can extend the elaborator with the ability to convert a type family of the form:

* $P : \mathit{record}\ \{...\} \to \mathcal{U}$

» Automatic Bundling

If we can extend the elaborator with the ability to convert a type family of the form:

* $P : \mathit{record}\ \{...\} \to \mathcal{U}$

Into it's "bundled" representation:

* $\mathit{record}\ \{..., \mathit{fibre} : P(\mathit{struct}\ \{...\})\}$

» Automatic Bundling

If we can extend the elaborator with the ability to convert a type family of the form:

* $P : \mathit{record}\ \{...\} \to \mathcal{U}$

Into it's "bundled" representation:

* $\mathit{record}\ \{..., \mathit{fibre} : P(\mathit{struct}\ \{...\})\}$

Then we can re-use the record patching machinery from before to solve the problem!

» Automatic Bundling

If we can extend the elaborator with the ability to convert a type family of the form:

* $P : \mathit{record}\ \{...\} \to \mathcal{U}$

Into it's "bundled" representation:

* $\mathit{record}\ \{..., \mathit{fibre} : P(\mathit{struct}\ \{...\})\}$

Then we can re-use the record patching machinery from before to solve the problem!

* IE: given $\mathit{hom} : \mathit{record}\ \{s : \mathit{obj},\ t : \mathit{obj}\} \to \mathcal{U}$...

» Automatic Bundling

If we can extend the elaborator with the ability to convert a type family of the form:

* $P : \mathit{record}\ \{...\} \to \mathcal{U}$

Into it's "bundled" representation:

* $\mathit{record}\ \{..., \mathit{fibre} : P(\mathit{struct}\ \{...\})\}$

Then we can re-use the record patching machinery from before to solve the problem!

* IE: given $\mathit{hom} : \mathit{record}\ \{s : \mathit{obj},\ t : \mathit{obj}\} \to \mathcal{U}$...
* We can then bundle this family into $\mathit{record}\ \{s : \mathit{obj},\ t : \mathit{obj},\ \mathit{fibre} : \mathit{hom}(\mathit{struct}\ \{s = s, t = t\})\}$...

» Automatic Bundling

If we can extend the elaborator with the ability to convert a type family of the form:

* $P : \mathit{record}\ \{...\} \to \mathcal{U}$

Into it's "bundled" representation:

* $\mathit{record}\ \{..., \mathit{fibre} : P(\mathit{struct}\ \{...\})\}$

Then we can re-use the record patching machinery from before to solve the problem!

* IE: given $\mathit{hom} : \mathit{record}\ \{s : \mathit{obj},\ t : \mathit{obj}\} \to \mathcal{U}$...
* We can then bundle this family into $\mathit{record}\ \{s : \mathit{obj},\ t : \mathit{obj},\ \mathit{fibre} : \mathit{hom}(\mathit{struct}\ \{s = s, t = t\})\}$...
* And then constrain the fields using singleton types.

» Automatic Bundling

Pulling it all together, we can define a category like so:

```
def category : type :=
  sig (ob : type)
      (hom : sig (s : ob) (t : ob) → type)
      (id  : (x : ob) → hom [ s .= x | t .= x ])
      (seq : (f : hom) (g : hom [ s .= f.t ]) →
             hom [ s .= f.s | t .= g.t ])
      ...
```

(Excuse the cooltt syntax)

» Implementation

This is roughly the same level of brevity that unification and implicits would provide us, but at a fraction of the implementation cost.

» Implementation

This is roughly the same level of brevity that unification and implicits would provide us, but at a fraction of the implementation cost.

* It took under 100 lines of code in `cooltt` to add all the elaborator features required.

Questions?