Theories and Data Structures

—Draft—

ANONYMOUS AUTHOR(S)

ACM Reference format:
Anonymous Author(s). 2020. Theories and Data Structures
—Draft—. 1, POPL, Article 1 (January 2020), 5 pages.

DOI:

1 BACKGROUND

In this section, we introduce our problem and our exposition language, Agda.

```
module POPL19 where

open import Helpers.DataProperties

open import Function using (_∘_)
open import Data.Nat
open import Data.Fin  as Fin hiding (_+_)
open import Data.Vec as Vec hiding (map)
open import Relation.Binary.PropositionalEquality
```

A *(non-dependently-typed) signature* is what programmers refer to as an 'interface': It is a collection of 'sort', or type, symbols; along with a collection of 'function' symbols, and a collection of 'relation' symbols. For our discussion, we are only interested in single-sorted non-relational signatures, which means there is only one anonoymous sort symbol and a collection of function symbols. In Agda:

```
record Signature₀ : Set₁ where
  field
    FuncSymbs : Set
    arity     : FuncSymbs → ℕ
```

In a dependently-typed language, we may simplify this further since a pair `A × (A → B)` corresponds to a *single* dependent type `A' : B → Set`.

```
record Signature₁ : Set₁ where
  field
    {- “FunSymbs n” denotes the function symbols of arity “n”. -}
    FuncSymbs : (n : ℕ) → Set
```

Of-course a one-field record is silly, so we simplify further:

```
Signature : Set₁
Signature = ℕ → Set
```

Here is a few examples of signatures:

2020. XXXX-XX/2020/1-ART1 $15.00
DOI:

```
data Type𝒮 : Signature where

data Unary𝒮 : Signature where
  next : Unary𝒮 1

data Magma𝒮 : Signature where
  _⨾_ : Magma𝒮 2

data Pointed𝒮 : Signature where
  point : Pointed𝒮 0
```

```
{- Pointed Magma -}
data Semigroup𝒮 : Signature where
  Id  : Semigroup𝒮 0
  _⨾_ : Semigroup𝒮 2

{- Alias -}
Monoid𝒮 : Signature
Monoid𝒮 = Semigroup𝒮

{- Pointed Unary Magma -}
data Group𝒮 : Signature where
  Id : Group𝒮 0
  _ : Group𝒮 1
  _⨾_ : Group𝒮 2
```

Just as a signature can be construed as an interface, an *algebra* can be thought of as an implementation:

```
record _Algebra (FuncSymbs : Signature) : Set₁ where
  field
    Carrier : Set
    reify   : ∀ {n : ℕ} (f : FuncSymbs n) → Vec Carrier n → Carrier

open _Algebra

{- syntactic sugar -}
infixl 5 ⟦_⟧_$_
⟦_⟧_$_ : {FuncSymbs : Signature} {n : ℕ} → FuncSymbs n → (𝒜 : FuncSymbs Algebra)
     → Vec (Carrier 𝒜) n → Carrier 𝒜
⟦ f ⟧ 𝒜 $ xs = reify 𝒜 f xs
```

Here are two examples algebra:

```
numbers : Type𝒮 Algebra
numbers = record { Carrier = ℕ ; reify = λ { () _ } }

additive-ℕ : Semigroup𝒮 Algebra
additive-ℕ = record { Carrier = ℕ
                    ; reify = λ { Id [] → 0; _⨾_ (x ∷ y ∷ []) → x + y}
                    }
```

To define the notion of *free algebra*, we only need the concept of *homomorphism*: Functions that preserve the interpretations of the function symbols.

```
record _Homomorphism (FuncSymbs : Signature) (Src Tgt : FuncSymbs Algebra) : Set where
  field
    map          : Carrier Src → Carrier Tgt
    preservation :   ∀ {n : ℕ} {f : FuncSymbs n} {xs : Vec (Carrier Src) n}
                   → map (⟦ f ⟧ Src $ xs) ≡ ⟦ f ⟧ Tgt $ Vec.map map xs

open _Homomorphism

{- Syntactic Sugar -}
_⟨$⟩_ : {𝒮 : Signature} {Src Tgt : 𝒮 Algebra}
      → (𝒮 Homomorphism) Src Tgt → Carrier Src → Carrier Tgt
h ⟨$⟩ xs = map h xs
```

We are now in a position for our prime definition: One says 𝒜 *is a free* 𝒮*-algebra for a type* G *of 'generators'* provided 𝒜 is an 𝒮-algebra that 'contains' G and every 𝒮-homomorphisms 𝒜 → ℬ correspond to functions G → Carrier ℬ.

```
record _free-for_ {𝒮 : Signature} (𝒜 : 𝒮 Algebra) (G : Set) : Set₁ where
  field
    embed      : G → Carrier 𝒜
    extend     : {ℬ : 𝒮 Algebra} → (G → Carrier ℬ) → (𝒮 Homomorphism) 𝒜 ℬ
    {- "Homomorphisms are determined by their behaviour on embeded elements." -}
    uniqueness : {ℬ : 𝒮 Algebra} (H : (𝒮 Homomorphism) 𝒜 ℬ) → H ≡ extend (map H ∘ embed)

  restrict : {ℬ : 𝒮 Algebra} (H : (𝒮 Homomorphism) 𝒜 ℬ) → G → Carrier ℬ
  restrict H g = H ⟨$⟩ embed g
```

This paper aims to solve 𝒜 free-for G where 𝒜 is the unknown.

An interface generally comes with a collection of coherence laws enforcing desirable behaviour. Likewise, we want to speak of "equational algebras". This requires we speak of "equations", which are pairs of "terms":

```
data _Term-over_ (𝒮 : Signature) (X : Set) : Set where
  var : X → 𝒮 Term-over X
  _$_ : {n : ℕ} (f : 𝒮 n) → Vec (𝒮 Term-over X) n → 𝒮 Term-over X

{- Example semigroup term -}
x⨾Id⨾y : Semigroup𝒮 Term-over (Fin 2)
x⨾Id⨾y = _⨾_ $ (_⨾_ $ (var  :: Id $ [] :: []) :: var  :: [])
  where  = zero ;  = suc zero

{- Ever term is a function of its variables -}
arity : {X : Set} {𝒮 : Signature} → 𝒮 Term-over X → ℕ
arity _ = 0
-- arity (var x) = 1
-- arity (f $ xs) = sum (Vec.map arity xs)
-- Fails termination checking.

⟦_⟧t : {X : Set} {𝒮 : Signature} → (t : 𝒮 Term-over X) → (𝒜 : 𝒮 Algebra)
     → Vec (Carrier 𝒜) (arity t) → Carrier 𝒜
⟦ t ⟧t 𝒜 = {!!}

data _Equation-over_ (𝒮 : Signature) (X : Set) : Set where
  _≈_ : (lhs rhs : 𝒮 Term-over X) → 𝒮 Equation-over X

lhs rhs : {𝒮 : Signature} {X : Set} → 𝒮 Equation-over X → 𝒮 Term-over X
lhs (l ≈ r) = l
rhs (l ≈ r) = r

{- Example semigroup axiom -}
sg-assoc : Semigroup𝒮 Equation-over (Fin 3)
sg-assoc =   (_⨾_ $ ( :: (_⨾_ $ ( ::  :: [])) :: []))
           ≈ (_⨾_ $ ((_⨾_ $ ( ::  :: [])) ::  :: []))
  where  = var zero ;  = var (suc zero) ;  = var (suc (suc zero))
```

We can now define an equational theory:

```
record EquationalSpecfication : Set₁ where
  field
    -- Interface
    FuncSymbs : Signature
    -- Constraints, with numbers as variables
    Axioms    : FuncSymbs Equation-over ℕ → Set

open EquationalSpecfication

record _Theory (ℰ : EquationalSpecfication) : Set₁ where
  field
    Carrier' : Set
    reify'   : ∀ {n : ℕ} (f : FuncSymbs ℰ n) → Vec Carrier' n → Carrier'

  algebra : (FuncSymbs ℰ) Algebra
  algebra = record { Carrier = Carrier' ; reify = reify' }

  field
    satisfy : ∀ {e} {_ : Axioms ℰ e} → ⟦ lhs e ⟧t algebra ≡ ⟦ rhs e ⟧t algebra
```

2 WE WANT TO BE SYSTEMATIC ABOUT

Exploring Magma-based theories: see https://en.wikipedia.org/wiki/Magma_(algebra) where we want to at least explore all the properties that are affine. These are interesting things said at https://en.wikipedia.org/wiki/Category_of_magmas which should be better understood.

Pointed theories: There is not much to be said here. Although I guess 'contractible' can be defined already here.

Pointed Magma theories: Interestingly, non-associative pointed Magma theories don't show up in the nice summary above. Of course, this is where Monoid belongs. But it is worth exploring all of the combinations too.

unary theories: wikipedia sure doesn't spend much time on these (see https://en.wikipedia.org/wiki/Algebraic_structure) but there are some interesting ones, because if the unary operation is 'f' things like forall x. f (f x) = x is **linear**, because x is used exactly once on each side. The non-linearity of 'f' doesn't count (else associativity wouldn't work either, as _*_ is used funnily there too). So "iter 17 f x = x" is a fine axiom here too. [iter is definable in the ground theory]

This is actually where things started, as 'involution' belongs here.

And is the first weird one.

Pointed unary theories: E.g., the natural numbers

Pointer binary theories: need to figure out which are expressible

more: semiring, near-ring, etc. Need a sampling. But quasigroup (with 3 operations!) would be neat to look at.

Also, I think we want to explore

- ⋄ Free Theories
- ⋄ Initial Objects
- ⋄ Cofree Theories (when they exist)

Then the potential 'future work' is huge. But that can be left for later. We want to have all the above rock solid first.

3 RELATIONSHIP WITH 700 MODULES

To make it a POPL paper, as well as related to your module work, it is also going to be worthwhile to notice and abstract the patterns. Such as generating induction principles and recursors.

A slow-paced introduction to reflection in Agda:
https://github.com/alhassy/gentle-intro-to-reflection

4 TIMELINE

Regarding POPL:
https://popl20.sigplan.org/track/POPL-2020-Research-Papers#POPL-2020-Call-for-Papers
There is no explicit Pearl category, nor any mention of that style. Nevertheless, I think it's worth a shot, as I think by being systematic, we'll "grab" in a lot of things that are not usually considered part of one's basic toolkit.

However, to have a chance, the technical content of the paper should be done by June 17th, and the rest of the time should be spent on the presentation of the material. The bar is very high at POPL.

5 TASK LIST ITEMS BELOW

- ☒ JC start learning about org mode
- ☒ JC Figure out how to expand collapsed entries
- ☐ JC See §4, first code block, of https://alhassy.github.io/init/ to setup `:ignore:` correctly on your machine.
  - ∘ This may require you to look at sections 2.1 and 2.2.
    This also shows you how to get 'minted' colouring.
- ☐ JC Write introduction/outline
- ☐ MA To read: *From monoids to near-semirings: the essence of MonadPlus and Alternative*, https://usuarios.fceia.unr.edu.ar/~mauro/pubs/FromMonoidstoNearsemirings.pdf.

6 DONE LITERATE AGDA IN ORG-MODE

JC, for now, use "haskell" labelled src blocks to get basic colouring, and I will demonstrate org-agda for you in person, if you like. Alternatively, I can generate coloured org-agda on my machine at the very end.

- ⋄ A basic setup for *actually* doing Agda development within org-mode can be found at: https://alhassy.github.io/literate/
- ⋄ Example uses of org-agda include
  - ∘ https://alhassy.github.io/next-700-module-systems-proposal/PackageFormer.html ; also · · · .org
    - ★ Shallow use of org-agda merely for colouring ;; Prototype for Package Formers
      - · Source mentions org-agda features that have not been pushed to the org-agda repo.
  - ∘ https://alhassy.github.io/PathCat/
    - ★ Large development with categories ;; Graphs are to categories as lists are to monoids
  - ∘ https://github.com/alhassy/gentle-intro-to-reflection
    - ★ Medium-sized development wherein Agda is actually coded within org-mode.