Read the mode and stay positive

Towards internal positivity annotations

Malin Altenmüller[1] Joris Ceulemans[2] Lucas Escot[3]
Andreas Nuyts[2] **Josselin Poiret**[4]

[1]University of Strathclyde, Scotland

[2]imec-DistriNet, KU Leuven, Belgium

[3]TU Delft, Netherlands

[4]ENS de Lyon, France

June 15, 2023

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

- Pros: very readable and natural definition, constructors are explicit;

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

- Pros: very readable and natural definition, constructors are explicit;
- Cons: induction scheme lives outside of the type theory, can't abstract over it.

W-types

First internal approximation: W-types[1] aka. containers[2].

[1] Martin-Löf, “Intuitionistic type theory”.
[2] Abbott, Altenkirch, and Ghani, “Categories of Containers”.

W-types

First internal approximation: W-types[1] aka. containers[2].

Define the following primitive

```
data W (S : Set) (P : S → Set) : Set where
  sup : (cons : S) → (P cons → W S P) → W S P
```

[1] Martin-Löf, "Intuitionistic type theory".

[2] Abbott, Altenkirch, and Ghani, "Categories of Containers".

W-types

First internal approximation: W-types[1] aka. containers[2].

Define the following primitive

```
data W (S : Set) (P : S → Set) : Set where
  sup : (cons : S) → (P cons → W S P) → W S P
```

S is the type of *shapes* while P is the type of *positions*, ie. arities of the recursive references of the constructors.

[1]Martin-Löf, "Intuitionistic type theory".

[2]Abbott, Altenkirch, and Ghani, "Categories of Containers".

```
ℕShapes : Set
ℕShapes = Bool

ℕPositions : ℕShapes → Set
ℕPositions (inl tt) = ⊥ -- zero constructor
ℕPositions (inr tt) = ⊤ -- succ constructor

ℕ = W ℕShapes ℕPositions
```

- Pros: Historic approach, role is well-understood;

- Pros: Historic approach, role is well-understood;
- Cons: Not very intuitive, doesn't have great internal features.

- Pros: Historic approach, role is well-understood;
- Cons: Not very intuitive, doesn't have great internal features.

```
List : Set → Set
List A = W ListShapes ListPositions
  where ListShapes : Set
        ListShapes = ⊤ ⊎ A

        ListPositions : ListShapes → Set
        ListPositions (inl _) = ⊥
        ListPositions (inr _) = ⊤
```

We have one shape per element of A!

```
def-naturals : (n : ℕ)
             → (n ≡ zero)
               ⊎ (Σ[ m ∈ ℕ ] (n ≡ (succ m)))
```

```
def-naturals : (n : ℕ)
             → (n ≡ zero)
               ⊎ (Σ[ m ∈ ℕ ] (n ≡ (succ m)))


def-naturals (sup (inl tt) f) = {!!}
-- need to show f ≡ (λ ())
def-naturals (sup (inr tt) g) = {!!}
-- need to show g ≡ (λ tt → g tt)
```

```
def-naturals : (n : ℕ)
             → (n ≡ zero)
               ⊎ (Σ[ m ∈ ℕ ] (n ≡ (succ m)))
```

```
def-naturals (sup (inl tt) f) = {!!}
-- need to show f ≡ (λ ())
def-naturals (sup (inr tt) g) = {!!}
-- need to show g ≡ (λ tt → g tt)
```

`funExt` needed!

Fixed points

Can we take inspiration from the categorical semantics? Initial algebras of endofunctors are the least fixed points of type formers.

Fixed points

Can we take inspiration from the categorical semantics? Initial algebras of endofunctors are the least fixed points of type formers.

```
ℕ-type-former : Set → Set
ℕ-type-former X = ⊤ ⊎ X
```

How do we take least fixed points of type formers?

```
μ : (Set → Set) → Set
μ F = {!!}
```

How do we take least fixed points of type formers?

```
μ : (Set → Set) → Set
μ F = {!!}
```

Not all type formers have fixed points!

```
oops-type-former : Set → Set
oops-type-former X = X → ⊥
```

Proof assistants already use a syntactic criterion: strict positivity.

```
data I : Set where
  ok : (⊥ → I) → I → I
  not-ok : {! ((I → ⊥) → ⊥) → I !}
```

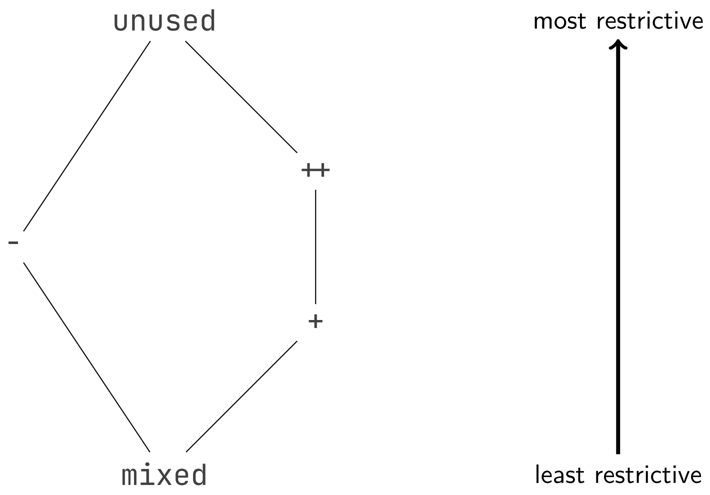
However, we have no way of saying inside the type system that a type former is strictly positive, so we still can't type μ.

Let's just extend the type system then!

```
id : @++ Set → Set
id X = X

not-positive : @++ Set → Set
not-positive X = {! X → ⊥ !}
```

New modal type theory, inspired by Abel, "Polarized Subtyping for Sized Types" and following the framework of Gratzer et al., "Multimodal Dependent Type Theory".



unused
most restrictive
++
-
+
mixed
least restrictive

```
spositive : @++ Set → Set
spositive A = ⊥ → A

negative : @- Set → Set
negative A = A → ⊥
```

Modalities compose!

```
f : @+ Set → Set
f A = negative (spositive (negative A))
```

```
spositive : @++ Set → Set
spositive A = ⊥ → A

negative : @- Set → Set
negative A = A → ⊥
```

Modalities compose!

```
f : @+ Set → Set
f A = negative (spositive (negative A))
```

Very importantly

```
pi : (@- A : Set) → (@++ B : Set) → Set
pi A B = A → B
```

After fiddling with the positivity checker, we can write down:

After fiddling with the positivity checker, we can write down:

```
data μ (@++ F : @++ Set → Set) : Set where
  fix : F (μ F) → μ F
```

Back to our running examples:

```
ℕ : Set
ℕ = μ (λ X → ⊤ ⊎ X)
```

Back to our running examples:

```
ℕ : Set
ℕ = μ (λ X → ⊤ ⊎ X)
```

```
List : @++ Set → Set
List A = μ (λ X → ⊤ ⊎ A × X)
```

Back to our running examples:

```
ℕ : Set
ℕ = μ (λ X → ⊤ ⊎ X)
```

```
List : @++ Set → Set
List A = μ (λ X → ⊤ ⊎ A × X)
```

```
W : (A : Set) (B : A → Set)
  → @++ Set → Set
W A B X =
  Σ[ a ∈ A ] (B a → X)
```

```
pattern zero = fix (inl tt)
pattern succ n = fix (inr n)

pattern nil = fix (inl tt)
pattern cons a b = fix (inr (a , b))

length : {A : Set}
       → List A
       → ℕ
length nil = zero
length (cons _ b) = succ (length b)
```

Further work

Subtyping

```
id : (@++ Set → Set) → (Set → Set)
id F = {!F!}
```

Need to eta-expand!

Reduces composability.

Generic fmap

```
fmap : (F : @+ Set → Set) {A B : Set}
     → (f : A → B)
     → (F A → F B)
fmap f fa = {!!}
```

Sound familiar?

Generic fmap

```
fmap : (F : @+ Set → Set) {A B : Set}
     → (f : A → B)
     → (F A → F B)
fmap f fa = {!!}
```

Sound familiar?

```
equivmap : (F : Type → Type) {A B : Type}
         → (eq : A ≃ B)
         → F A ≃ F B
equivmap F eq = pathToEquiv (cong F (ua eq))
```

```
{-# TERMINATING #-}
μelim : (F : @++ Set → Set) {A : Set}
      → (alg : F A → A)
      → (μ F → A)
μelim F alg (fix x) = alg (fmap F (μelim F alg) x)
```

→ Internalize some generic programming.

Directed Type Theory

```
data Hom[_,_] {A : Set ℓ}
      : @- A → @+ A → Set ℓ
      where
  id : ∀ {@unused x} → Hom[ x , x ]
```

Directed Type Theory

```
data Hom[_,_] {A : Set ℓ}
      : @- A → @+ A → Set ℓ
      where
  id : ∀ {@unused x} → Hom[ x , x ]



elim : {A : Set ℓ}
         (F : @- A → @+ A → Set ℓ')
      → ((@unused x : A) → F x x)
      ------------------------------
      → (@unused x y : A)
      → (Hom[ x , y ] → F x y)
elim F F-id x .x id = F-id x
```

```
fmap : {A B : Set ℓ}
     → (F : @+ Set ℓ → Set ℓ')
     → Hom[ A , B ] → Hom[ F A , F B ]
fmap F id = id
```

```
fmap : {A B : Set ℓ}
     → (F : @+ Set ℓ → Set ℓ')
     → Hom[ A , B ] → Hom[ F A , F B ]
fmap F id = id


HomToFun : {A B : Set ℓ}
         → Hom[ A , B ] → A → B
HomToFun {A = A} {B = B} =
  elim (λ X Y → X → Y) (λ X x → x) A B
```

```
fmap : {A B : Set ℓ}
     → (F : @+ Set ℓ → Set ℓ')
     → Hom[ A , B ] → Hom[ F A , F B ]
fmap F id = id
```

```
HomToFun : {A B : Set ℓ}
         → Hom[ A , B ] → A → B
HomToFun {A = A} {B = B} =
  elim (λ X Y → X → Y) (λ X x → x) A B
```

```
postulate ua : {A B : Set ℓ}
             → (A → B) → Hom[ A , B ]
```

Replacing positivity checks entirely

Remove syntactic positivity check and replace with type checking.

Replacing positivity checks entirely

Remove syntactic positivity check and replace with type checking.

Add μ as a primitive and make data declarations desugar into a use of it.

Replacing positivity checks entirely

Remove syntactic positivity check and replace with type checking.

Add μ as a primitive and make data declarations desugar into a use of it.

But we still don't know exactly how it interacts with funky inductive types like inductive-inductive or inductive-recursive ones!

Annotation inferrer

Recontextualize positivity checking as an annotation elaboration algorithm.

$\rightarrow$ Flexibility of type annotations + comfort of automation.

Semantics

- We're now in directed type theory, so we need a directed model like in North, "Towards a Directed Homotopy Type Theory".

Semantics

- We're now in directed type theory, so we need a directed model like in North, "Towards a Directed Homotopy Type Theory".
- Some types have a fix-point operator while most don't. Maybe they should belong to their own mode.

Semantics

- We're now in directed type theory, so we need a directed model like in North, "Towards a Directed Homotopy Type Theory".
- Some types have a fix-point operator while most don't. Maybe they should belong to their own mode.

Fix-point mode: categorical analogue of dcpos, $< \varkappa$-locally presentable categories with $< \varkappa$-accessible functors between them.

Complication: The rank of accessibility of $-^A$ depends on the cardinality of A.

Semantics

- We're now in directed type theory, so we need a directed model like in North, "Towards a Directed Homotopy Type Theory".
- Some types have a fix-point operator while most don't. Maybe they should belong to their own mode.

Fix-point mode: categorical analogue of dcpos, $< \varkappa$-locally presentable categories with $< \varkappa$-accessible functors between them.

Complication: The rank of accessibility of $-^A$ depends on the cardinality of A.

No clear semantics for the so-called lock operator on contexts in MTT for our modalities.

Takeaway

A prototype Agda implementation at `https://github.com/agda/agda/pull/6385`. Some work already merged!

Takeaway

A prototype Agda implementation at `https://github.com/agda/agda/pull/6385`. Some work already merged!

- Internalize induction schemes synthetically;

Takeaway

A prototype Agda implementation at `https://github.com/agda/agda/pull/6385`. Some work already merged!

- Internalize induction schemes synthetically;
- Use type-checking instead of syntactical checks for inductive types;

Takeaway

A prototype Agda implementation at `https://github.com/agda/agda/pull/6385`. Some work already merged!

- Internalize induction schemes synthetically;
- Use type-checking instead of syntactical checks for inductive types;
- Real-world use of directed type theory for programmers!

Abbott, Michael, Thorsten Altenkirch, and Neil Ghani. "Categories of Containers". In: *Foundations of Software Science and Computation Structures*. Ed. by Andrew D. Gordon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 23–38. ISBN: 978-3-540-36576-1.

Abel, Andreas. "Polarized Subtyping for Sized Types". In: *Computer Science – Theory and Applications*. Ed. by Dima Grigoriev, John Harrison, and Edward A. Hirsch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 381–392. ISBN: 978-3-540-34168-0.

Gratzer, Daniel et al. "Multimodal Dependent Type Theory". In: *Logical Methods in Computer Science* Volume 17, Issue 3 (July 2021). ISSN: 1860-5974. DOI: 10.46298/lmcs-17(3:11)2021. URL: http://dx.doi.org/10.46298/lmcs-17(3:11)2021.

Martin-Löf, Per. "Intuitionistic type theory". In: *Studies in proof theory*. 1984.

North, Paige Randall. "Towards a Directed Homotopy Type Theory". In: *Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics, MFPS*

Thank you!

Any questions? Suggestions?