Constructing Simple and Mutual Inductive Types in Agda

Stefania Damato

supervised by Prof. Thorsten Altenkirch

19th October 2020

Aims

Aims

- A **framework** of signatures for simple and mutual inductive types.

Aims

- A **framework** of signatures for simple and mutual inductive types.
- A **complete specification** of simple and mutual inductive types. For any signature, we define:
  - Algebras
  - Algebra morphisms
  - The initial algebra
  - The iterator
  - Uniqueness of the iterator.

Aims

- A **framework** of signatures for simple and mutual inductive types.
- A **complete specification** of simple and mutual inductive types. For any signature, we define:
    - Algebras
    - Algebra morphisms
    - The initial algebra
    - The iterator
    - Uniqueness of the iterator.
- A **reduction** from simple and mutual inductive types to W-types.

Inductive Types

Inductive Types

An *inductive type* is a data type defined by a list of constructors specifying how to form terms of this type.

Inductive Types

An *inductive type* is a data type defined by a list of constructors specifying how to form terms of this type.

Simple inductive types:

- ```
  data ℕ : Set where
      zero : ℕ
      suc  : ℕ → ℕ
  ```

- ```
  data InfTree : Set where
      ε∞  : InfTree
      sp∞ : (ℕ → InfTree) → InfTree
  ```

Inductive Types

An *inductive type* is a data type defined by a list of constructors specifying how to form terms of this type.

Mutual inductive types:

- ```
  data NF : Set
  data NE : Set

  data NF where
      ne : NE → NF
      lam : String → NF → NF

  data NE where
      var : String → NE
      app : NE → NF → NE
  ```
  ```

Signatures

Signatures

A signature consists of the number of sorts and constructors for each sort.

Signatures

A signature consists of the number of sorts and constructors for each sort.

```
record Sig : Set where
    constructor sig
    field
        sorts : ℕ
        cns : Fin sorts → List (Con sorts)
```

Signatures

A signature consists of the number of sorts and constructors for each sort.

```
record Sig : Set where
    constructor sig
    field
        sorts : ℕ
        cns : Fin sorts → List (Con sorts)
```

A constructor is a list of arguments.

Signatures

A signature consists of the number of sorts and constructors for each sort.

```
record Sig : Set where
    constructor sig
    field
        sorts : ℕ
        cns : Fin sorts → List (Con sorts)
```

A constructor is a list of arguments.

```
data Con (n : ℕ) : Set where
    cn : List (Arg n) → Con n
```

Signatures

A signature consists of the number of sorts and constructors for each sort.

```
record Sig : Set where
    constructor sig
    field
        sorts : ℕ
        cns : Fin sorts → List (Con sorts)
```

A constructor is a list of arguments.

```
data Con (n : ℕ) : Set where
    cn : List (Arg n) → Con n
```

An argument is either recursive or non-recursive.

Signatures

A signature consists of the number of sorts and constructors for each sort.

```
record Sig : Set where
    constructor sig
    field
        sorts : ℕ
        cns : Fin sorts → List (Con sorts)
```

A constructor is a list of arguments.

```
data Con (n : ℕ) : Set where
    cn : List (Arg n) → Con n
```

An argument is either recursive or non-recursive.

```
data Arg (n : ℕ) : Set where
    nrec : U → Arg n
    rec  : List U → Fin n → Arg n
```

Signatures

```
ℕSig : Sig
sorts ℕSig = 1
cns ℕSig = λ {zero → cn [] -- zero
                  :: cn (rec [] zero :: []) -- suc
                  :: []}
```

```
InfTreeSig : Sig
sorts InfTreeSig = 1
cns InfTreeSig = λ {zero → cn [] -- ε∞
                       :: cn (rec (nat :: []) zero :: []) -- sp∞
                       :: []}
```

Algebras

Algebras

An algebra consists of carrier types for each sort and constructors forming these types.

Algebras

An algebra consists of carrier types for each sort and constructors forming these types.

Example:

```
record ℕAlg : Set₁ where
    field
        N : Set
        z : N
        s : N → N
```

Algebras

An algebra consists of carrier types for each sort and constructors forming these types.

Example:

```
record ℕAlg : Set₁ where
    field
        N : Set
        z : N
        s : N → N
```

General:

```
record Alg (S : Sig) : Set₁ where
    field
        carriers : Fin (sorts S) → Set
        cons : (srt : Fin (sorts S)) (c : Con (sorts S)) →
                c ∈ (cns S) srt → conType srt carriers c
```

Algebra Morphisms

Algebra Morphisms

A morphism maps carrier types and preserves structure.

Algebra Morphisms

A morphism maps carrier types and preserves structure.

Example:

```
record ℕMor (n₁ n₂ : ℕAlg) : Set where
  field
    f : N n₁ → N n₂
    f_z : f (z n₁) ≡ z n₂
    f_s : (x : N n₁) → f ((s n₁) x) ≡ (s n₂) (f x)
```

Algebra Morphisms

A morphism maps carrier types and preserves structure.

Example:

```
record ℕMor (n₁ n₂ : ℕAlg) : Set where
  field
    f : N n₁ → N n₂
    f_z : f (z n₁) ≡ z n₂
    f_s : (x : N n₁) → f ((s n₁) x) ≡ (s n₂) (f x)
```

General:

```
record Mor (S : Sig) (A₁ A₂ : Alg S) : Set where
  constructor mor
  field
    f : (srt : Fin (sorts S)) → (carriers A₁) srt → (carriers A₂) srt
    eq : (srt : Fin (sorts S)) (c : Con (sorts S)) (p : c ∈ (cns S) srt)
         (xs : args srt (carriers A₁) c) →
         (f srt) (apply S A₁ srt c ((cons A₁) srt c p) xs) ≡
         apply S A₂ srt c ((cons A₂) srt c p) (map S A₁ A₂ srt c f xs)
```

Initial Algebra

Initial Algebra

The initial algebra for a signature `S` consists of carriers $[\![$ `S` $]\!]$ `srt` and their constructors.

Initial Algebra

The initial algebra for a signature $\texttt{S}$ consists of carriers $[\![\, \texttt{S} \,]\!]\, \texttt{srt}$ and their constructors.

Idea:

$[\![\mathbb{N}\texttt{Sig}]\!]$: Set
z : $[\![\mathbb{N}\texttt{Sig}]\!]$
s : $[\![\mathbb{N}\texttt{Sig}]\!] \to [\![\mathbb{N}\texttt{Sig}]\!]$

Initial Algebra

The initial algebra for a signature S consists of carriers ⟦ S ⟧ srt and their constructors.

Idea:

```
⟦ℕSig⟧ : Set
z : ⟦ℕSig⟧
s : ⟦ℕSig⟧ → ⟦ℕSig⟧
```

General:

```
Initial : (S : Sig) → Alg S
Initial S = record { carriers = λ srt → ⟦ S ⟧ srt ;
                     cons = λ srt c p → makeCons S srt c p }
```

Iterator & Uniqueness of Iterator

Iterator & Uniqueness of Iterator

A morphism from the initial S-algebra to any other S-algebra, and a proof it is unique.

Iterator & Uniqueness of Iterator

A morphism from the initial S-algebra to any other S-algebra, and a proof it is unique.

Idea:

```
InitialAlg : Initial ℕSig
⟦ ℕSig ⟧ zero : Set
zero : ⟦ ℕSig ⟧ zero
suc : ⟦ ℕSig ⟧ zero → ⟦ ℕSig ⟧ zero
```

```
BoolAlg : Alg ℕSig
Bool : Set
true : Bool
not : Bool → Bool
```

Iterator & Uniqueness of Iterator

A morphism from the initial S-algebra to any other S-algebra, and a proof it is unique.

Idea:

```
InitialAlg : Initial ℕSig
⟦ ℕSig ⟧ zero : Set
zero : ⟦ ℕSig ⟧ zero
suc : ⟦ ℕSig ⟧ zero → ⟦ ℕSig ⟧ zero
```

```
BoolAlg : Alg ℕSig
Bool : Set
true : Bool
not : Bool → Bool
```

General:

```
It : (S : Sig) (A : Alg S) → Mor S (Initial S) A
It S A = record { f = λ srt → funcs S A srt ;
                  eq = λ srt c p xs → eqProof S A srt c p xs }
```

Iterator & Uniqueness of Iterator

A morphism from the initial S-algebra to any other S-algebra, and a proof it is unique.

Idea:

```
InitialAlg : Initial ℕSig
⟦ ℕSig ⟧ zero : Set
zero : ⟦ ℕSig ⟧ zero
suc : ⟦ ℕSig ⟧ zero → ⟦ ℕSig ⟧ zero
```

```
BoolAlg : Alg ℕSig
Bool : Set
true : Bool
not : Bool → Bool
```

General:

```
It : (S : Sig) (A : Alg S) → Mor S (Initial S) A
It S A = record { f = λ srt → funcs S A srt ;
                  eq = λ srt c p xs → eqProof S A srt c p xs }
```

```
uIt : (S : Sig) (A : Alg S) (f : Mor S (Initial S) A) → f ≡ It S A
```

WI-Types

WI-Types

Given

```
I : Set
S : I → Set
P : (i : I) → S i → I → Set
```

the WI-type `WI : I → Set` is constructed by

```
sup : (i : I) (s : S i) ((j : I) → P i s j → WI j) → WI i.
```

WI-Types

Example: $\mathbb{N}$ as a WI-type.

```
I = ⊤ -- Sorts

S : I → Set -- Constructors of sorts
S tt = ⊤ ⊎ ⊤

P : (i : I) → S i → I → Set -- Recursive arguments
P tt (inj₁ tt) tt = ⊥ -- zero has no recursive arguments
P tt (inj₂ tt) tt = ⊤ -- suc has 1 recursive argument

zero' : WI I S P tt
zero' = sup tt (inj₁ tt) λ {tt → λ ()}

suc' : WI I S P tt → WI I S P tt
suc' n = sup tt (inj₂ tt) (λ {tt → λ {tt → n}})
```

WI-Type Algebra

WI-Type Algebra

An algebra constructed for the WI-type's representation of the inductive type.

WI-Type Algebra

An algebra constructed for the WI-type's representation of the inductive type.

```
WAlg : (S : Sig) → Alg S
WAlg S  = record { carriers = WI (Fin (sorts S)) (makeS S) (makeP S) ;
                   cons = λ srt c p → makeConsW S srt c p }
```

WI-Type Iterator

WI-Type Iterator

A morphism from the WI-type S-algebra to any other S-algebra.

WI-Type Iterator

A morphism from the WI-type S-algebra to any other S-algebra.

```
WIt : (S : Sig) (A : Alg S) → Mor S (WAlg S) A
WIt S A = record { f = λ srt → funcsW S A srt ;
                   eq = λ srt c p xs → {!!} }
```

Future Work

Future Work

Extend framework and results to:

- inductive families
- inductive-inductive types.