23

PROGRAMMING IN MARTIN-LÖF TYPE THEORY

Martin-Löf type theory can also be regarded as a programming language in which is possible to formally specify and prove theorems about the code itself. It is similar to the Calculus of Constructions that we mentioned earlier.

Formal proofs in MLTT can be written in this programming language and checked by a machine. Some programming languages offering this complex type system are

- **Agda** implements a variant of MLTT and can be used as a programming language or a proof assistant;
- **Coq** [04] was developed in 1984 in INRIA; it implements Calculus of Constructions and was used, for example, to check a proof of the Four Color Theorem;
- **Idris** implements a different version of identity types than that in MLTT and aims to be a useful programming language instead of a proof assistant;
- **NuPRL** [CAB+86] implements Extensional Type Theory;
- **Cubical** [CCHM16] provides an implementation of Cubical Type Theory;
- **RedPRL** is still in beta and provides an implementation of Cubical Type Theory in the NuPRL style.

In this text, we will use Agda as our proof assistant and we will mechanize proofs in Martin-Löf type theory using it. We will introduce Agda using ideas from [McB17].

23.1 MARTIN-LÖF TYPE THEORY IN AGDA

In this chapter, we will describe how to program in MLTT using Agda as proof assistant. An Agda file consists of type declarations and definitions stated with =, as in the following example.

```
- An identity function
id : ∀{l} {A : Set l} → A → A
id a = a

- A constant function
const : ∀{l} {A B : Set l} → A → B → A
```

```
const a b = a

- Composition of functions
_∘_ : ∀{l} {A B C : Set l} → (B → C) → (A → B) → A → C
(g ∘ f) x = g (f x)
```

This code exemplifies multiple idiosyncratic features of Agda when compared to our previous presentation of MLTT.

- The type of types, $\mathcal{U}$, is written as `Set`; however, it has nothing to do with our usual *set-theoretical sets*. The hierarchy of universes in Agda is written as

  $$\texttt{Set} : \texttt{Set}_1 : \texttt{Set}_2 : \ldots$$

  We will see later that it is possible to use compiler flags to postulate the collapse of this hierarchy into a single `Set : Set` and derive a contradiction from this assumption.
  If we want a definition to work at all levels of the universe hierarchy, we need to explicitly write `∀{l}...Set l` instead of simply write `Set`.
- Dependent function are implicitly built into the language. In general, the dependent function type $\prod_{a:A} B$ is written as `(a : A) → B`. In particular, we write the type `id` : $\prod_{A:\mathcal{U}} A \to A$ as `(A : Set) → A → A`.
- Agda allows a shorthand for nested dependent function types of the form $\prod_{a:A} \prod_{a':A} B$. We can write them as `(a a' : A) → B`, instead of having to write `(a : A) → (a' : A) → B`.
- Implicit arguments are declared between curly braces, `{ ... }`. Whenever we call a function with implicit arguments, the type checker will infer them from the context if we decide not to overwrite them explicitely.
- Infix operators can be defined using underscores `_`.

We have seen that Agda directly provides function types, but it also provides two mechanisms to define new types: ***data declarations*** and ***record types***.

- **Data declarations** can be thought as inductive types. They build the free type (the initial algebra) over a set of constructors; and they are better suited for defining positive types such as the void type, coproducts or natural numbers.
- **Record types** can be thought as generalized dependent n-tuples in which every element of the tuple is labeled and depends on the previous ones. They are better suited for defining negative types such as the unit type or product types. A constructor can be provided for them.

The complete type constructors of MLTT and the majority of types that we will use in this text can be defined in terms of these.

```
- An inductive datatype is determined by a (possibly empty) list of
- constructors.
```

```
data ⊥ : Set where

data _+_ (S : Set) (T : Set) : Set where
 inl : S → S + T
 inr : T → S + T

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

- A record is determined by a (possibly empty) list of fields that
- have to be filled in order to define an instance of the type.
record ⊤ : Set where
 constructor unit

record _×_ (S : Set) (T : Set) : Set where
 constructor _,_
 field
  fst : S
  snd : T

record  (A : Set)(B : A → Set) : Set where
 constructor _,_
 field
  fst : A
  snd : B fst

- Propositional equality is another example of inductive type,
- namely, the smallest type containing reflexivity.  We define it at
- all levels of the universe hierarchy.
data _==_ {l} {A : Set l} (x : A) : A → Set l where
 refl : x == x
```

Once we have defined these types, we can start profiting from the propositions as types interpretation to write and check mathematical proofs in Agda.

Functions can be defined by a (possibly empty) list of declarations that exhaustively pattern match on all the possible constructors of the type.

```
- Some simple proofs about types in Agda using pattern matching.
- Commutativity of the product.
comm-* : {A : Set}{B : Set} → A × B → B × A
comm-* (a , b) = (b , a)
```

```
- Associativity of the coproduct.
assocLR-+ : {A B C : Set} → (A + B) + C → A + (B + C)
assocLR-+ (inl (inl a)) = inl a
assocLR-+ (inl (inr b)) = inr (inl b)
assocLR-+ (inr c) = inr (inr c)

- Principle of explosion.  Ex falso quodlibet.
exfalso : {X : Set} → ⊥ → X
exfalso ()

- Projections of the dependent pair type.
```

In particular, we can check our previous proof of the Theorem of Choice defining the two projections from a dependent pair type.

```
- Projections of the dependent pair type.
proj1 : {A : Set} → {B : A → Set} →  A B → A
proj1 (fst , snd) = fst

proj2 : {A : Set} → {B : A → Set} → (p :  A B) → B (proj1 p)
proj2 (fst , snd) = snd

- Theorem of choice
tc : {A B : Set} → {R : A → B → Set} →
 ((x : A) →  B (λ y → R x y)) → ( (A → B) (λ f → (x : A) → R x (f x)))
tc {A} {B} {R} g = (λ x → proj1 (g x)) , (λ x → proj2 (g x))
```

The path induction principle follows from the definition of equality as an inductive type and can be used to prove indiscernability of identicals.

```
- Indiscernability of identicals proved by pattern matching.  We
- only have to prove the case in which the equality is reflexivity.
indiscernability : {A : Set} → {C : A → Set} →
 (x y : A) → (p : x == y) → C x → C y
indiscernability _ _ refl = id
```

23.2 TYPE IN TYPE

```
- This proof is based on Thorsten Altenkirch notes on
- Computer Aided Formal Reasoning (G53CFR, G54CFR),
- Nottingham University.
```

```
{-# OPTIONS –type-in-type #-}

module Russell where

 - Imports our previous declarations.
 open import Ctlc
 open import Data.Bool

 - Encoding of set-theoretical sets.  This is a definition of
 - a Set indexed by an index type I. This definition crucially
 - uses Set :  Set, as it is a Set taking a Set as an argument.
 - As Set is a reserved word, we use Aro = Set in esperanto.
 data Aro : Set where
  aro : (Index : Set) → (Index → Aro) → Aro

 - Definition of membership.
 - An element is a member of the set if there exists an
 - index pointing to it on the set.
 _∈_ : Aro → Aro → Set
 x ∈ (aro Index a) = Index (λ i → x == a i)

 _∉_ : Aro → Aro → Set
 a ∉ b = (a ∈ b) → ⊥

 - Paradoxical set R. It contains all the sets that do not
 - contain themselves.  The index type is the type of the
 - set that do not contain themselves.
 R : Aro
 R = aro ( Aro (λ a → a ∉ a)) proj1


 - Lemma 1.  Every set in R does not contain itself.
 lemma1 : {X : Aro} → X ∈ R → X ∉ X
 lemma1 ((X , proofX∉X) , refl) = proofX∉X

 - Lemma 2.  Every set which does not contain itself is in R.
 lemma2 : {X : Aro} → X ∉ X → X ∈ R
 lemma2 {X} proofX∉X = ((X , proofX∉X) , refl)

 - Lemma 3.  R does not contain itself
 lemma3 : R ∉ R
 lemma3 proofR∈R = lemma1 proofR∈R proofR∈R
```

```
- Russell's paradox.  We have arrived to a contradiction.
russellsparadox : ⊥
russellsparadox = lemma3 (lemma2 lemma3)
```