To type or not to type*

Piotr Rudnicki †
Department of Computing Science, The University of Alberta,
Edmonton, Alberta, Canada T6J 2H1.
email: `piotr@cs.ualberta.ca`

1 Introduction

N. G. de Bruijn in his letter to R. S. Boyer of August 19, 1994 which appeared at the QED mailing list wrote:

> ... in type systems like AUTOMATH the notion of elementhood can be seen as typing. Customers who like typed sets (let me call these customers TSC's) introduce a real variable by a single block opener, like
>
> `let x be real number`
>
> where `real number` is taken as a type. So for the TSC's it is the introduction of a typed variable. The customers who prefer to think in terms of untyped sets (let me call them USC's), however, have to order by means of two block openers:
>
> `let x be a set`
>
> and
>
> `assume this x is a real number.`
>
> In their case we have to realize that `x is a real number` is the property of the set `x`. The first one of the two block openers is the introduction of a typed variable (where the type is `set`), the second one is the assumption.

*This is a short summary of my presentation at the QED Workshop II, Warsaw, 20-22 July 1995.

†Supported in part by NSERC Grant No. OGP9207.

The word *type* is used with a number of different meanings and someone noticed that for the use of *type* as above, probably one should use *sort*. In MIZAR this notion is named *syntactic type*.

The question that we would like to address is: *What are the advantages and disadvantages of using syntactic types?*

In MIZAR one can work without syntactic types (types for short). However, the MIZAR data base has been developed using a hierarchy of types as it seems closer to everyday mathematical practice and the developers of MIZAR articles usually followed that practice as they knew it.

2 What are syntactic types in MIZAR?

Let us look at an example of how the type of *finite sequence of X* is defined. In article FINSEQ_1 we have the following definition:

```
definition let D be set;
 mode FinSequence of D -> FinSequence means
  rng it c= D ;
 existence  proof
    :: Here we have to demonstrate that there is an object of type
    :: FinSequence whose range is a subset of D
 end;
end;
```

FinSequence of is a type constructor which given a specific set as a parameter results in a type expression which can be used then to type objects. Later on we can then write:

```
let FST be FinSequence of NAT;
```

The denotation of FinSequence of NAT is the set of all finite sequences whose range is a subset of NAT. There is a substantial difference between the above introduction of x and the following:

```
let FSU be set;
assume FSU is FinSequence of NAT;
```

which can be characteristic of the USC at the de Bruijn's restaurant. In MIZAR, the typing expressed at the time of introducing an object (let ...) is processed differently then the typing expressed in an assumption. The type information contained in the object declaration is always automatically taken into account by the MIZAR semantic analyzer and inference checker, while the information contained in an assumption must be explicitly referenced and is effective only for the inference checker.

In the MIZAR abstract of article `FINSEQ_1` the definition of `FinSequence of ...` appears as:

```
definition let D be set;
 mode FinSequence of D -> FinSequence means
:: FINSEQ_1: def 4
rng it c= D ;
end;
```

and the label `FINSEQ_1:def 4` is used to make a reference to the definiens of `FinSequence of D` when needed. (The proofs do not appear in MIZAR abstracts.) The above definition says that `FinSequence of D` is a `FinSequence` with an additional condition given by the definiens. `FinSequence` is the mother type of `FinSequence of D` and `FinSequence of D` is a subtype of `FinSequence`. Let us have a look at the definition of `FinSequence` (in its abstract format, without proof of existence):

```
definition
  mode FinSequence is FinSequence-like Function;
end;
```

`FinSequence` turns out to be a shorthand for a `Function` with the attribute `FinSequence-like`. This attribute, applicable to `Function` is defined as follows:

```
definition
 mode FinSequence-like -> Function means
:: FINSEQ_1: def 2
ex n st dom it = Seg n ;
end;
```

The above is read: the domain of a `FinSequence-like Function` is equal to an initial segment of natural numbers. And below we quote the sequence of definitions that lead from the primitive notion of `set` to the notion of `Function`, again we skip the correctness conditions:

```
definition
 mode Relation-like -> set means
:: RELAT_1: def 1
  x ∈ it implies ex y,z st x = [y,z];
end;
```

```
definition
 mode Relation is Relation-like set;
end;

definition let X be set;
 pred X is Function-like means
:: FUNCT_1: def 1
 for x,y1,y2 st [x,y1] ∈ X & [x,y2] ∈ X holds y1 = y2;
end;

definition
 cluster Relation-like Function-like set;
end;

definition
 mode Function is Function-like Relation-like Any;
end;
```

The MIZAR semantic analyzer takes into account the entire chain of mother types. In particular, the object `FST` introduced above besides being a `FinSequence of D`, is also a `FinSequence`, a `Function`, a `Relation`, and of course a `set` as `set` is the root of the tree of types. All operations defined for a mother type are applicable also to the subtype. For instance, function application is defined as follows:

```
 definition let f be Function, x be Any;
  func f.x -> Any means
 ::  FUNCT_1:  def 4
 [x,it] ∈ f if x ∈ dom f otherwise it = 0;
end;
```

And therefore we can write `FST.10` which is always denoting. Note, that we must not write `FSU.10` as function application requires that the left argument of the dot is a function which does not follow from the declaration of `FSU` and no assumption regarding `FSU` is taken into account by the semantic analyzer.

Note that although `FST.10` is denoting its type is `Any`, (`Any` and `set` are synonyms). Can we force `FST.10` to be of type `Nat`? We found the definition of certain functor π in the MIZAR data base:

```
definition
  let X be set, D be non empty set, p be PartFunc of X,
      D, i be Any;
 assume  i ∈ dom p;
 func π(p,i) -> Element of D means
:: FINSEQ_4: def 4
   it = p.i;
end;
```

However, we must not write π(FST,10) yet, as FinSequence of NAT is not a partial function (PartFunc of NAT,NAT) in the MIZAR sense. In fact, PartFunc does not have Function in the chain of its mother types! The mode (type constructor) PartFunc has been derived as follows:

```
definition
 let X,Y;
 mode Relation of X,Y -> Subset of [:X,Y:] means
:: RELSET_1: def 1
  not contradiction;
end;

definition
 let X,Y;
 cluster -> Relation-like Subset of [:X,Y:];
end;

definition let X,Y;
  cluster Function-like Relation of X,Y;
 end;

definition let X,Y;
  mode PartFunc of X,Y is Function-like Relation of X,Y;
 end;
```

How can we make MIZAR treat a FinSequence of D as a PartFunc? For this we use a mechanism known as redefinition of modes. First, we show that the attribute FinSequence-like is applicable to partial functions (article FINSEQ_4):

```
definition let D be set;
 cluster FinSequence-like PartFunc of NAT,D:
```

```
  existence proof
    :: Here we have to show that there exists a partial function
    :: which is FinSequence-like.
end;
```

And now we can redefine `FinSequence of D` as a `PartFunc of NAT, D`.

```
definition let D be set;
 redefine mode FinSequence of D -> FinSequence-like PartFunc
     of NAT,D;
  coherence proof
    :: Here we have to show that every FinSequence of D is
    :: a PartFunc of NAT, D
end;
```

With this redefinition `FST` besides being whatever it was before is also a `PartFunc of NAT, NAT` and the expression π`(FST, 10)` is correctly typed.

3 Disadvantages of using syntactic types

- Syntactic types are an additional complication of any proof-checking system and as such decrease the reliability of the system and of its implementations.
- There is the temptation to define new notions on types that are too narrow.
- Type definitions need to be proven correct:
  - Since types have non-empty denotations we have to prove existence of an object of a newly defined type.
  - When we redefine the meaning of a type we have to prove coherence, namely that the type being redefined is coherent with the newly added mother type.
- The language must provide means for changing type of an object. In Mizar this is done by the `reconsider` construction. For example:

```
let x be Nat;
. . .
reconsider x as Integer by INT_1:9;
```

and from now on `x` is treated as an `Integer`.

Many a time the type change is performed by some artificially looking manipulations like introduction of otherwise not needed casting function or using existential quantifier and equality within a formula to “smuggle” information that an object of a type is equal to some object of a different type.

4 Advantages of using syntactic types

- A collection of some objects may be too big to be a set; however, we may introduce a type to name the collection. For example, we can define the type `Group` and objects of this type may have an arbitrary set as their carrier, while there is no possibility to define the set of all groups in MIZAR.

- The information carried by types of objects can be processed more efficiently by a specialized machine rather than the general inference engine.

  - Computing the possible types of an object and its attributes (subtypes) involves only sentential calculus.
  - The number of possible unifications when the types are taken into account is substantially reduced.

- Types facilitate overloading of symbols. For example, multiplication of real numbers is announced as (in `HIDDEN`):

```
definition let x,y be Element of REAL;
 func x · y -> Element of REAL;
  commutativity;
end;
```

  and later on characterized axiomatically in `AXIOMS`. However, we can redefine this multiplication for natural numbers with a natural result. This is done in article `NAT_1` as follows:

```
definition let n,k;
redefine
 func n · k -> Nat ;
  coherence proof
    :: Here we have to prove that indeed when we multiply two
    :: natural numbers we obtain a natural number as a result.
  end;
end;
```

- Hidden arguments. Not all arguments of a function or predicate have to be explicitly mentioned—some may be inferred from types of explicit arguments. In CAT_1 we find the following definition of the composition of morphisms in a category:

```
definition let C be Category;
           let a,b,c be Object of C;
           let f be Morphism of a, b;
           let g be Morphism of b, c;
 assume A: Hom(a,b)<>∅ & Hom(b,c)<>∅;
   func g·f -> Morphism of a,c means
    .  .  .  Definiens omitted
 existence proof Proof omitted end;
 uniqueness;
end;
```

The pattern of the defined operation has two explicit arguments, the morphisms in a category, and four implicit arguments that have to be reconstructed from the explicit arguments to fit what is stated in the definition. It is hard to imagine stating explicitly all six arguments when we use the composition of morphisms, which seems unavoidable if we do not use syntactic types.

- Reservations. With types, we can announce that certain identifiers are used to denote objects of specific types, and later when using the identifiers, we do not have to repeat the typing information. This feature of MIZAR saves a lot on writing, although some authors do not always use it.