Proving the Correctness of Functional Programs using Mizar

Yatsuka Nakamura

Shinshu University
Faculty of Engineering
Information Engineering
380-8553 Nagano-city, Japan
ynakamur@cs.shinshu-u.ac.jp

Abstract. A method for proving the correctness of functional programs (e.g., Haskell 98 [1]) using the Mizar system is introduced. The functions in a functional program are translated to Mizar functions and the proofs of existence and uniqueness of Mizar functions are done semi-automatically using schemes in Mizar. Types of variables are also reduced to Mizar types. It is also shown how to design concrete Haskell programs using Mizar. Some problems remain in completing such a method, but they can be overcome and are also discussed here.

1 Introduction

Until now, the designing methods of programs have been a kind of art, not a science. However, as software programs become more and more huge, the work of removing bugs becomes very difficult. Some theoretical method for developing software is necessary.

Recently, people have become aware of the importance of functional programs like Haskell [1]. The reason for this is that we can resolve huge programs into basic and simpler functions with functional programs.

Functions of a functional program are similar to mathematical functions so it may be possible to prove the correctness of each function of a functional program.

If one uses only pencil and paper to try to prove the correctness of a program, it would not be a reliable proof because the level of accuracy is different between computer programs and usual handwritten mathematics.

Fortunately, we now have very strong tools called proof checkers to do mathematics in a rigid way. One of the most popular proof checkers is Mizar, which was originated by Dr. A. Trybulec and his group.

The extent of the library of a proof checker is important when it is used to prove the correctness of programs because the semantics of programs relates to mathematical models in the library.

Fortunately, the Mizar Mathematical Library (MML) is currently the largest [4]. It contains nearly 1,000 articles and approximately 32,000 lemmas and theorems.

In the Mizar library, there are some attempts to prove the correctness of programs. One is the "PRGCOR" series [2] by the author where the concept of "non overwriting" is introduced and the correctness of some C programs written in non overwriting style can be proven.

Most functional programs are essentially "non destructive" in the sense that all values in variables are not destroyed by overwriting. So, the "non destructive" property is the same as our concept of "non overwriting". If we restrict programs to functional ones, the discussion becomes very simple.

The description of functions of 1^{st} order logic in Haskell is very near to the definition of functors in Mizar. For example, the "max" function in Haskell is written as

```
   let max a b = if a>=b then a else b
```

On the other hand, the max functor in Mizar is written as

```
   definition let a,b;
   func max(a,b) equals
   ::XXREAL_0:def 9
    a if b <= a otherwise b;
```

So, with a little modification of sentences, we can convert the definitions of Mizar functors into Haskell functions.

With the completion of such a correspondence, we have the following possibility:

The names of functions which are installed to Haskell are given as

```
       $H_max
```

in Mizar. If the same function is already defined in Mizar, the description of a synonym will be enough, as follows.

```
   notation let x,y be Element of REAL;
    synonym $H_max(x,y) for max(x,y);
end;
```

Basic functions in Haskell (functions in PRELUDE) are defined in a Mizar article. New programs which a programmer wants to write in Haskell form can be written in Mizar. For example, from the following Mizar definition

```
   definition let a,b,c;
   func $H_max3(a,b,c) equals
   :AA: $H_max($H_max(a,b),c);
```

we can derive the corresponding program through some automated method as

```
   let max3 a b c = max(max a b) c
```

As a Mizar theorem, we know

```
theorem :: XXREAL_0:34
 max(max(a,b),c) = max(a,max(b,c));
```

Hence we can see that for a new Haskell program

```
max3 a b c = max a (max b c)
```

is valid.

With only a number of theorems about the new function, its correctness is guaranteed. The function can be defined without contradiction in Mizar, but whether or not the definition is intuitively proper is another problem.

More advanced Haskell programs can also be written using $H_ form functions in Mizar. If all variables in the right hand side of a formula in the definition of a new functor are $H_ form functions, then it can be converted to a Haskell function.

When a new Mizar article is verified by the checker, there is a step of "accommodation", where the checker gathers all previous articles used in the new article. By the same method, one can gather all necessary old definitions of $H_ form functors used to define a new function and one can get the necessary library file to run it.

In such a situation, the Mizar library containing $H_ form functors will become a very good manual for describing their use because in the definitions the type of arguments are described and the conditions for using the functor are also described by the clause of "assume". The theorems about such a functor show the conditions for using it as well. For example, the functor to get the roots of an algebraic equation gives the correct answer only if the coefficient of the equation is not zero.

By changing all operations in Haskell to a function form, for example

```
(<=) a b,
```

instead of

```
a <= b,
```

the correspondence of program functions and functors will become very simple.

But, the situation is more difficult because there are additional important capabilities for functional program languages like Haskell which are difficult to correspond to Mizar functors.

One of them is the treatment of recurrent descriptions like

```
let sum0 i = if ((<=)i 0) then 0 else (+) i (sum0 ((-)i 1))
```

There is no corresponding way of making such a description for Mizar functors. One must use Mizar functions in such a case. Definitions of Mizar functions are rather difficult as compared to the case of functors.

There are more important cases where one must use functions.

2 Mizar Functions and Mizar Functors

In the definition of a Mizar function, one must prove the existence and uniqueness of the function itself, not for a value of the function. The following functor "$H_double" is a simple functor introduced as an example.

```
definition let x be Element of REAL;
::PROG. Example.
func $H_double x equals
:BB: $H_(*) (x,2);
correctness;
end;
```

The Mizar verifier accepts the above definition without proof, where we assume the function $H_(*) is already introduced. From the above definition we can automatically get a Haskell program as follows (parts following a :: sign are comments in Mizar. A "PROG." comment means that this definition corresponds to a Haskell program):

```
let double x = (*) x 2
```

We can also define $H_double as a function in Mizar as follows:

```
definition
func $H_double -> Function of REAL,REAL means
:CC: for x being Element of REAL holds
     it.x = $H_(*) (x,2);
existence;
::>    *4
uniqueness;
::>     *4
end;
```

Two *4 comments were imposed on the article when it was checked by the Mizar verifier. These comments indicate incorrect lines of induction. We must prove the existence of the $H_double function, not the existence of the value x*2 which is easier. About 50 lines are necessary to prove the existence and uniqueness of the $H_double function.

The function "double" is the simplest one, but to define it as a function is not easy in Mizar.

Let us return to the recurrent program example shown in the previous section. The Haskell program was:

```
let sum0 i = if ((<=)i 0) then 0 else (+) i (sum0 ((-)i 1))        (1)
```

As stated already, the Haskell function sum0 above cannot be represented by a Mizar functor. As a function of Mizar however this can be defined. To show the definition, we give two definitions of functors as follows:

```
definition let D be set;let x be Element of BOOLEAN,
 z,u be Element of D;
::PROG. Basic
func $H_if_then_else(x,z,u) -> Element of D equals
:AE508d: z if (x=$H_True) otherwise u;
correctness;
end;

definition let n,m be Element of NAT;
::PROG.
func $H_nsub(n,m) -> Element of NAT equals
:AE508e: $H_max($H_(-)(n,m),0);
correctness
proof
 per cases;
 suppose $H_(-)(n,m)<=0;
  hence $H_max($H_(-)(n,m),0) is Element of NAT
   by XXREAL_0:def 9;
 end;
 suppose B0: $H_(-)(n,m)>0;then
   B1: $H_(-)(n,m)=n-m & n-m>0 by AA173;then
   B2: $H_max($H_(-)(n,m),0)=n-m by XXREAL_0:def 9;
   n-m=n-'m by BINARITH:def 3,B1;
  hence $H_max($H_(-)(n,m),0) is Element of NAT by B2;
 end;
end;
end;
```

The second definition of $H_nsub has a proof of correctness. Usually, the proof of correctness of a functor is simple, but sometimes a proof of 10 or 20 lines is necessary.

Using the above two functors, we can give a definition of a $H_sum0 functor as:

```
definition
::PROG. Summing from 0 to l
func $H_sum0 -> Function of NAT, NAT means
:AE508f: for l being Element of NAT holds
 it.l=$H_if_then_else(($H_(<=)(l, 0)), 0, $H_(+)(l,
  it.($H_nsub(l,1))));
existence;
::>    *4
uniqueness;
::>     *4
end;
```

The proof of existence and uniqueness are rather complicated and will be discussed in the next section.

We obtain the Haskell program by extracting the last formula as:

```
let sum0 l = if_then_else ((<=) l 0) 0 ((+)l (sum0 (nsub l 1)))
```

This is slightly different from formula (1) on the page 204, but it is a more theoretical formula.

3 Schemes in Mizar to Prove the Correctness of Functions

This section is a technical discussion on how to prove the correctness of Mizar functions. As stated in Section 1, proving correctness (existence + uniqueness) for functions is more difficult than for functors. It is rather doubtful that the proofs of existence and uniqueness are necessary in actual application levels because existence is clear since the functions are really calculated and uniqueness is no concern for people who use them.

But to prove that the functions in a program are the same as mathematical functions, we must do the proofs of correctness.

The proof of correctness for a function which is essentially a functor is done semi-automatically using a special scheme. A scheme is a special device in Mizar used to prove formulas of higher order logics.

Let us consider the following scheme:

```
scheme Lambdax{X, Y() -> non empty set, F(set)->set}:
 ex f being Function of X(),Y() st (for x being Element of X()
 holds f.x = F(x));
```

The proof of this scheme is not difficult and can be done by using the fundamental scheme in the article of functions (FUNCT_2:sch 1 in MML). Some examples of proofs of existence of functions using the above scheme are shown below:

```
definition
func $H_succ -> Function of REAL,REAL means
:AA509: for n being Element of REAL holds it.n=n+1;
existence
proof
  deffunc F(Element of REAL)=$1+1;
  A1: for x being Element of REAL holds F(x) is Element of REAL;
  consider f being Function of REAL,REAL such that
  A2: (for x being Element of REAL
    holds f.x = F(x)) from Lambdax(A1);
 thus thesis by A2;
end;
uniqueness;
end;
```

This is a definition of the “succ” function in Haskell.

```
definition
func $H_abs -> Function of REAL,REAL means
:AA176: for x being Element of REAL holds it.x= |. x .|;
existence
proof
  deffunc F(Element of REAL)= |. $1 .|;
  A1: for x being Element of REAL
       holds F(x) is Element of REAL;
  consider f being Function of REAL,REAL such that
  A2: (for x being Element of REAL
    holds f.x = F(x)) from Lambdax(A1);
  thus thesis by A2;
end;
uniqueness;
end;
```

By comparing the two proofs above, the lines are almost the same except for the first line of defining the function F. So, the proof is semi–automatic. Similarly, uniqueness is also proved easily using the following scheme.

```
scheme Lambdau{X, Y() -> non empty set, F(set)->set}:
 for f1,f2 being Function of X(),Y() st (for x being
  Element of X()
 holds f1.x = F(x))&(for x being Element of X()
 holds f2.x = F(x)) holds f1=f2;
```

For recursive functions, existence and uniqueness are also proved easily by the following two schemes.

```
scheme Lambdas{Y() -> non empty set,Z() -> Element of Y(),
 G(Element of NAT,Element of Y())->Element of Y()}:
 ex f being Function of NAT,Y() st (for x being Element of NAT
 holds f.x=$H_if_then_else($H_(<=)(x,0),Z(),
  G(x,f.($H_nsub(x,1)))));
```

This scheme is a variation of an old scheme named "LambdaRecExD". The following is used for proofs of recurrent functions.

```
scheme Lambdasu{Y() -> non empty set,Z() -> Element of Y(),
 G(Element of NAT,Element of Y())->Element of Y()}:
 for f1,f2 being Function of NAT,Y() st (for x being
   Element of NAT
 holds f1.x=$H_if_then_else($H_(<=)(x,0),Z(),
   G(x,f1.($H_nsub(x,1)))))&
 (for x being Element of NAT
  holds f2.x=$H_if_then_else($H_(<=)(x,0),Z(),
    G(x,f2.($H_nsub(x,1)))))
  holds f1=f2;
```

4 Types and Classes

There are many classes and types in Haskell. We can translate them to Mizar. There are multiple meanings of "correctness" for programs. One of them is that correctness means that a program runs without overflow (and underflow) errors.

Here, we are concerned with semantic correctness, which means that the program achieves some calculation on a proper mathematical model.

If we are not interested in the size of numbers, the following types of Haskell need not be distinguished:

```
Int,    Integer.
```

These two types correspond to the following type of Mizar:

```
Element of INT
```

Also, the types of Haskell

```
Float and Double
```

correspond to Mizar type

```
Element of REAL,
```

and

```
Bool
```

corresponds to Mizar type

```
Element of BOOLEAN.
```

User defined types in Haskell are also defined in Mizar. In both the cases of basic and user defined types, in Mizar we use the form

```
Element of X,
```

where X is a non empty set. If we restrict the type to such a form, we can define a function of the form

```
Function of X,Y
```

easily.

If we put the definition as

```
definition let D,E be non empty set;
let f be Function of D,E;let x be Element of D;
redefine func f.x -> Element of E;
coherence by FUNCT_2:7;
end;
```

the value of a function from D to E is automatically interpreted as being an Element of E, so that functions work as if they were functors.

5 Functions of More than Two Arguments

Functions of more than two arguments in Haskell correspond to functions of the following:

```
    (<=) x y

definition
func $H_(<=) -> Function of REAL,Funcs(REAL,BOOLEAN) means
:AA200: for x being Element of REAL holds
 for y being Element of REAL holds
  (x<=y implies (it.x).y =$H_True)&
  (x>y implies (it.x).y =$H_True);
existence;
uniqueness;
end;
```

If we fix x to be an element of REAL, $H_`(<=).x` becomes a Function of REAL,BOOLEAN. This relates to the following Haskell program ([1,2,3,4,5] is a list which will be discussed in the next section):

```
Prelude> map ((<=) 2) [1,2,3,4,5]
[False,True,True,True,True]
```

The function “map” claims a function as its first argument. The term ((¡=) 2) represents a function from Num to Bool.

6 Modeling Pairs and Lists

The notion of pairs in Haskell is almost the same as the notion of pairs in Mizar, where a pair of x and y is written as [x,y] (it is written as (x,y) in Haskell). There is no problem in making a correspondence between them. But, the “fst” function and “snd” function give one problem when we want to translate them to Mizar.

What is the type of a pair [x,y]? We can choose any data of any type for x and y. Then, we must consider a set of all sets Ω as the range of “fst”, which is impossible because the existence of such a set causes Cantor’s paradox. We must choose a set smaller than Ω. As a Mizar functor, “fst” can be defined easily, because the functors z’1 and z’2 already exist in Mizar library. The following is clear:

```
$H_fst z = z'1
```

and

```
$H_snd z=z'2
```

But to define “fst” and “snd” as functions of Mizar, some other device will be necessary. We do not discuss this further here.

There if another concept called “List” in Haskell, like

```
    [1,2,3,4,5],
```

which is similar to a finite sequence of D (D is a set) in Mizar. This can be represented as

```
<* 1,2,3,4,5*>,
```

which is a function from {1,2,3,4,5} to D.

We can also use the concept of "XFinSequence of D" in Mizar [3], which is a function from {0,1,2,3,4} to D. With the Haskell function (!!), the n-th location is shown as

```
(!!) 0 [1,2,3,4,5] = 1.
```

So, it seems more natural to use XfinSequence, but the theory for FinSequence is well developed in Mizar.

There is another form of list like:

```
[1..]
```

or

```
[1,3..].
```

Both are infinite sequences which have infinitely many elements. Hence, we must construct a new type "LIST of D" in Mizar.

7 λ–Calculus

There is no corresponding concept of λ–calculus in Mizar. The function λ–calculus in Haskell is as follows:

```
Prelude> (\x -> (+) 2 ((*)x x)) 5
27
```

In the above, the "`(\x -> (+) 2 ((*)x x))`" part acts as a function of one variable (x=5 here. The value is calculated as 2+x*x). If we divide it into two lines, it is not necessary to use λ–calculus.

```
Prelude> let ll2 x=(+)2 ((*)x x)
Prelude> ll2 5
27
```

Here "ll2" is a temporarily introduced function. If we can substitute ll2 in the second line with the first line, then these two lines are mathematically the same as λ–calculus.

So, there is no essential difficulty for returning λ–calculus to Mizar formulas.

References

1. Johns, S.P (ed), *Haskell 98 Language and Libraries: the Revised Report*, Cambridge University Press, (2003).
2. Nakamura, Y., *Correctness of Non Overwriting Programs. Part I*, Formalized Mathematics, Vol. 12, No. 1 (2004), pp. 29–32.
3. Tsunetou, T., Bancerek, G. and Nakamura, Y., *Zero-Based Finite Sequences*, Formalized Mathematics, Vol. 9, No. 4 (2001), pp. 825–829.
4. Wiedijk, F., *Comparing Mathematical Provers*, Mathematical Knowledge Management: Second International Conference, MKM 2003, Bertinoro, Italy, February 16-18, 2003. Proceedings, Springer Berlin / Heidelberg (2004), pp. 188–202.