Type checking in FrAid

NOTICE: ""

Table of contents

""

Note:

The main focus of FrAid is manipulation of mathematical functions. As such its purpose is served if everything could be just a complex number or evaluates to a complex number. In reality though, as a programming language it needs a certain number of utility functions and constructs which somewhat "polutes" the otherwise clean semantics. The following section treats this subject by introducing type and function implementation distinction (which do not exist in the "pure" language of mathematics). Again, argument types and function implementation details have no meaning in the context of defining mathematical functions in FrAid and only apply where human interaction (graphics, error handling, etc.) or Input/Output operations are involved.

1. FrAid is weakly typed

FrAid is weakly typed - there is no restriction what values could be assigned to variables or returned by functions and everything evalues to a Complex but just like other weakly typed languages the functions could place restrictions on what their arguments are.

Examples:

```
                //function which takes numbers and strings and returns the
oposite type
                f(x) = if isString(x) then 5 else
                if isComplex( x ) then "hello" else "don't know";

                //a function which does not check the types (uses the fact
that everything evaluates to a number)
                subtract( "hello", 3);                          //will
return -3 because the string evaluates to 0

                //a function which places restrictions on its arguments
                printTree(2);                                   //
returns error - Argument not a DefinedFunction
```

2. Checked types

In FrAid you can use these functions to check what is behind a returned/passed value (by a Function or a Variable):

```
                isComplex()
                isFunction()
                isString()
                typeOf()
```

In your Java code for the same purpose you you can use:

```
                org.fraid.complex.ComplexHelper.ensureString( Complex );
                org.fraid.complex.ComplexHelper.ensureComplex( Complex );
                org.fraid.complex.ComplexHelper.ensureComplexFunction(
```

```
Complex );
                org.fraid.complex.ComplexHelper.ensureDefinedFunction(
Complex );
```

Note:

You need these only if you are using FrAid from Java.

The following is-a hierarchy shows how the FrAid "types" relate to each other (the same names are used in the rest of the documentation). The asterisk denotes checked type. If a type is checked a value of a parent's type could be passed as an argument but only those of the appropriate type will be accepted:

- Complex - everything evaluates to a Complex, if not Complex* would generally evaluate to 0+0i;
  - Complex* - checked Complex, accepts only complex numbers (3+4i) or a Function which evaluates to a complex number;
    - Boolean - a checked Complex could be a Boolean;
  - String - something is considered a String if SimpleNode.m_text != null;
    - String* - checked String, accepts only strings ("hello") or a Function which evaluates to a string;
  - Function* - a single quote character followed by a function signature ('sin(x)), the names of the arguments are insignificant, the number is important;
    - Variable* - a function with no arguments (Pi(); or Pi;), the brackets can be ommited;
      - UserDefinedVariable* - a no arguments UserDefinedFunction;
    - UserDefinedFunction* - a function defined by the user (vs. the Java defined functions) - f(x,y)=x*sin(y)+1;
      - UserDefinedVariable* - a no arguments UserDefinedFunction (a=5;) ;
  - Boolean - a Complex could be interpreted as a true/false value. 0+0i is false, everything else is true.

3. Function types

A function is uniquely identified by its name and the number of its arguments. For FrAid the type of the arguments is insignificant and if there are any restrictions placed upon them by the function itself FrAid remains unaware.
Example:

```

```

```
                //although these functions both take only a particular type
of argument
                //FrAid won't see any difference and will just override the
first one
                //(the Java defined functions can not be overriden)
                f(x)= if isString(x) then "OK" else "ERROR";
                f(x)= if isComplex(x) then "OK" else "ERROR";

                //this is OK, different name
                g(x)= if isComplex(x) then "OK" else "ERROR";

                //or this, different number of arguments
                f(x,y)= if isComplex(x) then "OK" else "ERROR";
```

3.1. Defined Functions

The user defined FrAid functions can be overriden at any moment of the FrAid execution. Example:

```
                    f(x)=x^2;
                    plot('f(x));
                    f(x)=sin(x); //see what happens to the plot when you
change the definition of f(x)
```

The user defined FrAid functions can only have fixed number of arguments.

The user defined FrAid functions can only return Complex, String and Boolean (see Checked Types).

3.2. Library Functions

The Library Functions can not be overriden by user defined functions

```
sin(x)=x*x; //will return Trying to override Java defined function error
```

The Library Functions can take:

- Fixed number of arguments - `add(x,y)`
- Variable number of arguments - `sum(x,y,z)` vs. `sum(x)`.

The Library functions can return Complex, String and Boolean (see Checked Types). Although not enforsed in the grammar the preffered way for a library function to return/define a function is to take a String argument for the function name and register the function (still returning Complex, String or Boolean). The rest of the code then can use the newly registered function addressing it by name. For examples see rk(), rk1() and fourier().

3.3. Library Functions with variable number of arguments

A Library Function with variable number of arguments can coexist with any other function witht the same name but fixed number of arguments.

A Library Function with variable number of args. can place restriction on the number of its arguments. `sampledF()` for instance won't work unless an odd, greater than seven number of arguments is passed (the first of which is a String, see type checks and examples).

If two functions with the same name exist, one with fixed number the other with variable number of args. and a call is made for a function with this name and the number of args of the fixed number function, the fixed number function takes precedence.

Java Note

It is very easy to implement your own Libraray Functions, just implement the ComplexFunction interface and place it in a place where the symbol table can find it. Using this technique FrAid can be made to do completely different things, even the standard operators' +, -, *, ... behaviour could be changed (well, you need to know what you are doing here, but the point is that it is quite flexible). FrAid comes with a simple code generator which can generate library functions and plugins - see the Library Function Generator and the Plugin Generator

3.4. Sampled Functions

A Sampled Function is the equivalent of a Vector of a certain length in other languages with the difference that in FrAid they have a start point attached to the first element and a (fixed!) step between its elements:

```
                    v(x)=samples(1,1,2,2,2); //create a sampled function
which starts at 1,
                                             //has a step=1 and three
elements equal to 2;
                    v(x)=vector(2,2,2);      //equivalent to the one above
```

This way the sampled functions can be used anywhere non-sampled functions can be used:

```
                    plot({v+sin});           //add a sampled and
non-sampled function
```

Once created the start point and step of a Sampled Function can be checked (from the example above):

```
                    startS(v); // --> 1
                    stepS(v);  // --> 1
```

or changed:

```
                    startS(v,.5);
```

```
                    stepS(v,.3);
                    startS(v); // --> .5
                    stepS(v);  // --> .3
```

The length of a Sampled Function could be checked with `lengthS(v); // --> 3` but can not be changed without redefining the whole function.

The individual elements can be checked or changed using the elemS() function.

Sampled Functions can be concatenated, truncated, padded, shift-rotated, etc.

As you may have noticed many functions which deal with Sampled Functions have the 'S' postfix in their names (printS, stepS, startS, ...)

Every non-sampled function could be sampled:

```
                    f(x)=sin(x)+cos(x);             //non-sampled function
                    fs(x)=sampleL(f, 0, 1, 1000);   //fs(x) is the sampled
equivalent of f(x) in the interval [0,1]
```

Note:

Like everything else FrAid keeps track of the changes in Sampled Functions and the functions they depend on. For the example above see what happens (`plot(fs);`) when you change the definition of f: `f(x)=sin(x)-cos(x); - fs(x) will be resampled!`

The main purpose of the Sampled Functions is to allow DSP and processing of sampled data (like signals from the sound card). For consistency with the non-sampled functions outside of the interval where the samples are defined the function evaluates to zero.

Note:

In versions prior to 1.5 the concept of Sampled Functions existed but only through the sampledF() function (now deprecated).

3.5. Generator Functions

Generator Functions can be used in two contexts:

- Function definitions: `fs(x)=sampleL(f, 0, 1, 1000);`
- Where a function is expected as an argument: `plot(sampleL(f, 0, 2*Pi, 1000));` in which case a hidden function is registered (in this respect they are very similar to the meta expressions)

The Generator functions can create both sampled or non-sampled functions. For example `sampleL(), truncateS` create Sampled Functions while `firResp(), icft()` create regular "analog" functions.

3.6. Functions generated by Meta Expressions

The FrAid meta expressions (`{xxx}`) generate hidden functions which are passed where a function is expected as an argument: `plot({sin+cos});`. As already noted, the meta expressions are just a short hand and **everything in FrAid could be done without their use**.

> **Note:**
> Both the meta expressions and Generator Functions (when used outside of function definitions) create hidden functions which have names prefixed by "_hiddenF". They can be accesed by name just like any other FrAid function.

""

""