Static and Strong Typing for Extended *Mathematica*

Peter Fritzson

Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden
Email: petfr@ida.liu.se
Phone:+46 13 281484, Fax: +46 13 284499

Abstract

There are at least three reasons why a static type system is a useful extension to *Mathematica*: A type checker can find errors during software development in *Mathematica*; Object oriented typing is useful for handling complexity when building large applications or equation-based simulation models; Precise static type information is needed for efficient internal compilation of *Mathematica* as well as for automatic translation to efficient code in languages such as C++ or Fortran.

This paper presents a short overview of the syntax and semantics as well as rationale for a static type system designed to be well integrated into *Mathematica*. A static type system makes it possible to find type errors in the whole program before execution starts, in contrast to the standard dynamic type checking of *Mathematica* which only finds type errors in parts of the program during execution. The type system includes basic types, array types, record types, typed variables, typed functions and object oriented constructs. The syntax of the type extensions is standard *Mathematica*, and the implementation is entirely within *Mathematica*.

1 Introduction

This paper informally presents a static type system designed to be well integrated into *Mathematica*. Declarations are introduced to associate static types with *Mathematica* objects. The syntax is *Mathematica* compatible, which makes it possible to use the type extensions in ordinary *Mathematica* code. There are several reasons why a static type system is a useful extension to *Mathematica*:

- Precise static type information is needed for generation of efficient executable code.

- A type checker is useful for finding errors during software development in *Mathematica*.
- Object oriented typing is useful to handle complexity when building large applications and equation-based simulation models.

Additional requirements of a type system are:

- *Ease of use*. The type system should be easy to understand and use.
- *Readability and standardization*. The type notation should be readable, and conform to common program language standards, as well as conform to relevant *Mathematica* conventions.
- *Compatibility*. Typed *Mathematica* code should execute together with untyped code. Adding type information should be a pure extension—existing code should in general not need to be changed.

First we discuss the motivation behind a static type system in some more detail.

1.1 Types for code generation

Precise static type information is needed for translating *Mathematica* into efficient code in languages such as Fortran90 and C++. It is also needed for more efficient internal compilation of *Mathematica* to efficient code as evidenced by the type information parameters to the standard *Mathematica* `Compile` function. Experience from our earlier work on code generation to C++ and Fortran from the ObjectMath [1,2,4] extension to *Mathematica* made it clear that precise static type information could not be automatically deduced from dynamically typed *Mathematica* code in all cases, especially when arrays are involved. Therefore declaration of static type information was introduced. However, in many cases static type information can be derived automatically through type inference.

1.2 The need for type checking

Debugging *Mathematica* programs can be hard. Simple spelling errors and other mistakes may cause pattern matching to fail, which causes huge unevaluated expressions to be returned to the user. It is usually not so easy to realize where the source of the error is located. Partial dynamic type checking of function parameters can be turned on, but will only be able to catch errors for the particular test cases which are used during debugging and testing.

A static type checker can be of great help here, in finding simple mistakes such as spelling errors of variables or function names, wrong number of arguments to functions, mismatch of actual argument types and formal parameter types, etc. Concerning parameter types, most builtin *Mathematica* functions have numeric parameters which will be assigned the type `Real` in the static type system. The static type `Real` is of course an approximation, since there are several numeric

forms in *Mathematica* such as infinite precision numbers and fractions. However, the approximation to `Real` fits well with code generation to statically type languages such as Fortran90 or C++ as well as being a reasonable static type approximation for execution within *Mathematica*, since the exact numeric type may change dynamically during execution.

A relevant question is whether the static type system will be able to detect enough errors, since most functions are numeric anyway, and declaring the type `Real` for function parameters and results will not add that much information. There are however many functions which accept parameters that are vectors and arrays of different forms, and for which precise type information is quite useful for type checking. Additionally, checking the number of function parameters and whether a function or variable has been declared will catch many common mistakes by *Mathematica* programmers.

1.3 Types for object oriented simulation modeling

Simulation models are usually constructed to simulate some model of aspects of the external world. This is precisely where object orientation is most useful. For example, typical mechanical systems consist of a number of mechanical components which can be described by classes containing equations that describe motion, forces, material properties, etc. Simulation models of mechanical and other systems can be put together by connecting such objects from class libraries. For example, a car contains connected objects such as motor, transmission, wheels, etc. Inheritance provides reuse of equations and function definitions when inheriting from general classes to more application specific instances. Thus, object oriented type and class mechanisms provide structuring and reuse when building mathematical simulation models of physical systems.

2 Typed declarations

When introducing declarations and static typing as an extension to *Mathematica*, some keywords and names need to be reserved. The chosen keywords should preferably not be used for other purposes within *Mathematica*, be intuitive and easy to understand, and correspond to common practice in other programming languages.

The same requirements hold for the syntax of typed definitions. It should be readable, easy to use, be compatible with *Mathematica* syntax, and correspond to common programming language conventions.

2.1 Declarations in packages

The notion of declaration is already present in standard *Mathematica*, although it is not very pronounced. A *Mathematica* package can be regarded as a sequence of untyped variable and function declarations each of which may be ended by semicolon.

For example:

```
BeginPackage["ExamplePackage"]

varname1;
varname2;

func1[a_,b_] := ...;
func2[x_,y_] := ...;

EndPackage[]
```

The untyped "declarations" of variables `varname1` and `varname2` introduce these names into the name context of the package. The "declarations" of functions `func1` and `func2` define these functions.

Declaration separators and end-markers

Notice the use of semicolon in the above package example as an end-marker and separator between declarations, which is allowed for definitions at the package level in standard *Mathematica*. This is consistent with declaration syntax in common programming languages such as Java, C, and C++, etc.

Unfortunately, currently there are three mutually inconsistent syntax options for ending or separating declarations in *Mathematica*: At the package level declarations are ended by a semicolon or new line, whereas declarations of local variables are separated by comma. Such inconsistency easily gives rise to unnecessary syntax errors, especially when cutting and pasting declarations between different scope levels.

Therefore, regarding *typed* declarations we will chose semi-colon as a universal separator and end-marker for declarations at all levels, including declarations within classes and packages, local declarations, and field declarations within records. However, the other two syntax options are still supported as in standard *Mathematica* for reasons of compatibility.

2.2 Basic types

The basic type names `Real`, `Integer`, `Complex`, `Symbol`, and `String` etc. are already defined by *Mathematica* to be used in patterns and as head tags of basic objects. However, since the meaning of these words are essentially the same when used in a static type system, there should not be a problem to reuse these type names. To choose other names would be confusing for the user.

We introduce the type name `Boolean` as the type of values `True` or `False`, and `Null` to indicate the empty type or absence of type, e.g. for a procedure that does not return any data value.

The type name `AnyType` indicates that an object may have any type, which is useful to describe the type of certain objects, e.g. the element type of an array that

may contain a mixture of objects such as real numbers, integers, strings, etc.

2.3 Patterns versus types

Are *Mathematica patterns* the same as *static types*? One might be tempted to answer yes to this question since both notions describe sets of objects which fulfill a pattern or type constraint. There are however certain differences between patterns and types:

- A pattern language is designed to express structural properties to be dynamically tested during execution, whereas a static type system describes properties to be checked statically before execution starts. This tends to influence the pattern and type notation.
- Certain aspects of the static type system, e.g. user-defined type names, record types, arrays, classes, etc. do not fit well into the *Mathematica* pattern language.
- Static type information can be approximate when used for type checking. For example, our `Real` type is an annotation that tells the system that a certain object (e.g. a function call) has a potential numeric (non-integer, `Real` or `Rational`) value if all arguments to such an object are numeric and not symbolic.
- Precise static type information is needed for generation of efficient code in statically compiled languages. Sometimes this information must be provided by the user to obtain the precise intended meaning for generated code.

Since one of our basic requirements is that static type information should not change the behavior or degrade performance of interpreted *Mathematica* code, we have made the design choice to declare static type information for functions separated from the function parameter patterns.

2.4 Typed function declarations

Regard the following three variants of the same untyped function `f2` in *Mathematica*, for which the second and third rules are attempts to include some type information:

```
f2[x_] := x+2;

f2[x:_Integer] := x+2;

f2[x:(_ | _Integer)] := x+2;
```

The first definition of `f2` works for both symbolic and numeric arguments, which is often what the user intends, e.g. when producing symbolic expressions that will eventually be computed numerically. If an `_Integer` pattern is provided as a parameter "type" in the second definition, the function will unfortunately no longer work for symbolic arguments such as names of variables with potential integer val-

ues. The third definition can both handle symbolic arguments and provide some type information, but may collide with some uses of `Alternative ( | )` and still does not specify a function return type.

Therefore, for reasons mentioned in the previous section we provide the argument types and the function type separately in a function signature in front of the function head, similar to Java, C, C++. An arrow in the signature indicates mapping from input argument types to output result types. Some examples are shown below.

```
(Real->Real) sin2[x_] := Sin[x]+2.0;

(Real->Null) myprint[x_] := Print[x];

({}->Real)   myrandom[] := Random[];

({Integer,Real}->Real)  myfunc[x_,y_]  := x+y*y;

({Real,Real}->Integer
) sincos3[x_,y_] := Sin[x]+Cos[y]+myfunc[x,y];
```

Below is the `FullForm` of the first definition (`sin2`). As can be seen, the blank(s) between the signature and the function head give rise to a `Times[]` node, whereas the arrow from argument types to result type becomes a `Rule[]` node.

```
SetDelayed[
  Times[Rule[Real,Real], sin2[Pattern[x,Blank[]]] ],
  Plus[Sin[x],2.]
]
```

Since `Times[]` nodes are essentially never used as function names in normal *Mathematica* code, `SetDelayed` can for `Times[]` nodes be redefined to perform the special action of storing away type information in a symbol table, as well as defining an untyped `sin2` function as usual. This stored type information is then used for type checking and code generation.

Type arguments to the *Mathematica* Compile function

The example below illustrates the rather drastic changes that need to be done to a typical user-defined function such as `sincos3`, in order to use the standard *Mathematica* `Compile` function. This violates our requirements of compatibility and coexistence with interpreted *Mathematica* code and makes the function definition much less readable. Therefore this notation is not a viable option for typed *Mathematica* function definitions.

```
sincos3 = Compile[{{x, _Real}, {y, _Real}},
                  Sin[x]+Cos[y]+myfunc[x,y],
                  {{myfunc[__], _Integer}}]
```

2.5 Typed declarations

The types of global variables also need to be declared. We introduce the `Var[]` declarator for declaring variables, as in the example package below:

```
BeginPackage["TypedExamplePackage"]

Var[
 Real     varname1;
 Integer  varname2;
];

({Real,Real}->Real) myfunc[x_,y_]  := x+y*y;

({Real,Real}->Real) sincos[x_,y_]  :=
                                  Sin[x]+Cos[y]+myfunc[x,y];
EndPackage[]
```

User-defined type names are declared via the `Type[]` declarator:

```
Type[
 MyReal  = Real;
 MyInt   = Integer;
 MyArr10 = ArrayOf[Real,{10}];
];
```

2.6 Type constructors and array types

A type constructor is a constructor that can create types, and may have types or other entities as arguments.

For example, `ArrayOf` is a type constructor for the creation of array types. As an example, `ArrayOf[Real,{10}]` is a type for vectors of length 10 containing real numbers. The `ArrayOf` keyword was chosen to avoid collision with the already existing *Mathematica* `Array` function.

2.7 Record types

Record type declarations are introduced through the `RecordType` declarator:

```
RecordType[PersonType = Person[Real age; String name]];
```

A record type may contain several variants of records, where each variant is tagged by a specific data constructor. The vertical bar operator (|) separates the record variant alternatives. A record alternative may be without field specifiers, in case a constructor with no parameters is given.

```
RecordType[SimpleExpr = INT
                      | REAL
```

```
                        | XPLUS[Expr; Expr]
                        | XMINUS[Expr; Expr]
                        | XTIMES[Expr arg1; Expr arg2]
                        | XDIV[Expr arg1; Expr arg2]
];
```

This type declaration creates the type `SimpleExpr` and defines the data constructors `INT`, `REAL`, `XPLUS`, `XMINUS`, `XTIMES` and `XDIV`.

2.8 Object oriented constructs

The static type system also contains object oriented constructs such as `Class`, based on the ObjectMath[1,2,4] extension to *Mathematica*. A `Class` can contain functions (i.e methods), variables and other classes. This most recent version of the object oriented constructs has been designed to be compatible with the *Modelica*[5] modeling language, with some influence from Java. A class example is shown below.

```
Class[BiCycle[C,P];  Extends[TwoWheeler[C]];
  Class[frontwheel;  Extends[Wheel[P]]; ];
  Class[rearwheel;   Extends[Wheel[P]]; ];
  Class[frame;       Extends[Body];     ];
  ...
]; (* end BiCycle *)
```

Conclusion

We have presented a short overview of some aspects of the syntax, semantics as well as design rationale for a static type system extension to *Mathematica*, with applications in type checking, better code generation and simulation tools based on *Mathematica*.

References

[1] Peter Fritzson, Lars Viklund, Dag Fritzson, Johan Herber. High Level Mathematical Modeling and Programming in Scientific Computing. *IEEE Software*, pp. 77-87, July 1995.

[2] Lars Viklund, Peter Fritzson: ObjectMath – An Object-Oriented Language and Environment for Symbolic and Numerical Processing in Scientific Computing. *Scientific Programming*, Vol. 4, pp. 229-250, 1995.

[3] Stephen Wolfram. *The Mathematica Book*, Wolfram Media Inc., 1996.

[4] ObjectMath Home Page, `http://www.ida.liu.se/labs/pelab/omath`.

[5] Modelica Home Page, `http://www.Dynasim.se/Modelica`.