FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Interfacing Computer Algebra Systems with ExaStencils

Mehdi Rezaiepour

Master Thesis

Interfacing Computer Algebra Systems with ExaStencils

Mehdi Rezaiepour

Master Thesis

Aufgabensteller:	Prof. H. Köstler
Betreuer:	J. Hoenig, S. Kuckuk, M. Heisig
Bearbeitungszeitraum:	22.11.2020

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Master Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 22. November 2020 .....

Abstract

In this work, we developed interfaces for two Computer Algebra Systems (CAS) to be used in conjunction with ExaStencils compiler. The purpose of these interfaces are the utilization of CAS in expression simplifications. The chosen CAS packages were SymPy (in Python) and Maxima (in Lisp). Considering these packages are not developed in the same language as ExaStencils (written in Scala), developing their interfaces faced further complication. To alleviate this obstacle, ScalaPy package was used for interface with SymPy, and ABCL package was utilized to develop the interface with Maxima. Since ExaStencils have 5 layers (L1, L2, L3, L4, and IR), and use of CAS was desired in L2, L3, and IR, for each of these layers a separate interface was developed for every CAS. Although similar in essence, due to the differences in data structures used in each layer of ExaStencils, 6 interfaces were developed. For simplicity, in this report, different interfaces point to the two groups of interfaces developed for two CAS packages. The design and development of both interfaces and their integration were performed in a way that enables the user (developers of ExaStencils) to choose which CAS would be used in each layer of the ExaStencils compiler, if any.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

One of the most effective ways to shorten the run-time of simulation is simply by not performing some of the calculations! Therefore, one could gain more performance by *simplifying* the calculations to reduce the number of arithmetic instructions needed to be performed likewise to replace them with a cheaper one. To pursue these goals, one could either use their mathematics knowledge or use a software package to facilitate the task. The group of software packages that could aid the manipulation of mathematical expressions is generally called "Computer Algebra System (CAS)".

In this thesis, we provide the interface required for utilizing two CAS package, in the ExaStencils compiler. The goal of the ExaStencils compiler is the generation of highly optimized C++ and CUDA code; hence, it has to eliminate unnecessary computations where possible. Since the development of a simplification mechanism that rivals any of the freely available CAS packages requires years of research and development, we decided to use well developed and supported packages instead of developing a new one. SymPy is one of the most popular packages which could be used as a library in Python programs. Another, much older and sophisticated CAS software is Maxima, developed in Common Lisp. Since ExaStencils is written in Scala and non of these packages are available in Scala, an interface is required to use them in ExaStencils.

To further introduce the reader to the ExaStencils project and illustrate the importance and urgency of having a CAS package coupled with the current code generator, we take a look at the ExaStencils. At first, we briefly introduce a few concepts. Then, we discuss the steps taken in generating the C++ or CUDA code. Finally, the existing mechanism for simplification of expression in the ExaStencils compiler is presented.

1.1 Domain-specific language

Computer programming languages could be divided into two groups; general-purpose languages and domain-specific languages (DSL). General-purpose languages are the group of programming languages that are applicable over a wide variety of domains. Unlike the general-purpose languages, domain-specific languages are designed to be used only in one special domain. Developing a programming language according to the requirements of a special domain enables the developers to have a more expressive and concise code while using all the possible optimizations to have the best performance. Moreover, it eliminates the need for the end user to have a comprehensive knowledge of high performance computing techniques. It also speeds up the development of new solutions and makes it possible to generate the code for several different hardware architectures.

1.2 ExaStencils

The framework used in the code generation is called ExaStencils. It provides an infrastructure for the development of geometric multigrid solvers using ExaSlang, a DSL designed for this purpose. ExaStencils' code generator first parses the DSL input code. Then, after error checking and few transformations, it generates the equivalent intermediate representation, IR. Afterward, it performs several optimizations on the abstract syntax tree (AST), before generating the target code (pretty-printing C++, or CUDA) [9]. Figure 1 shows the general workflow of ExaStencils. Dead code elimination, address pre-calculation, loop unrolling, vectorization, and arithmetic simplifications are a few of the automatic optimizations used in ExaStencils [6].

1.3 ExaSlang

ExaSlang is an external DSL developed to facilitate the generation of massively parallel geometric multigrid solvers [9]. The language has a Multi-layered structure as is shown in figure 2. In fact, it is comprised of four different layers of DSLs, L1, L2, L3, and L4. Each layer represents a level of abstraction.

The first layer, named Layer 1, describes the problem in the form of a partial differential equation to be solved on a continuous domain. The computation domain and boundaries are also specified in

this layer. Generally, this layer is introduced to be facilitated by users with insufficient knowledge of programming for the description of the case as a continuous problem. As we move to the higher levels, the abstractness of the descriptions decreases. In Layer 2, the user has the opportunity to describe the problem as a discretized formula. Therefore, it suits the users with a higher level of knowledge. For the first time in Layer 3, the user is able to set the parameter values, and settings, as well as algorithmic factors, and begin to be involved with the multigrid method. Here multigrid cycle could be customized, also smoothing operators could be determined. This layer is targeted toward the users with very high knowledge of programming and mathematics, e.g., computer scientists and mathematicians, due to the access to details of algorithm and discretization. The next layer, Layer 4, is considered solely the domain of computer scientists since the user could access details, such as data structures for data exchange, communication patterns, and other parts of the parallelization factors. The four mentioned layers are reachable by the user. The coupling of these layers of the user-inputs and the output of the compiler is carried out by a fifth layer, called IR (intermediate representation). Objects of this layer only exist in the compiler memory while compiling the inputs to generate output C++ or CUDA code. This layer is not part of ExaSlang, but an internal part of the ExaStencils compiler [9].

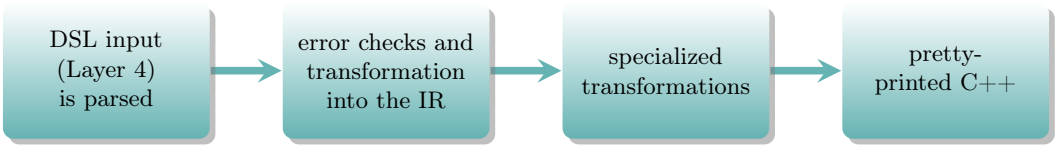

Figure 1: General workflow of ExaStencils

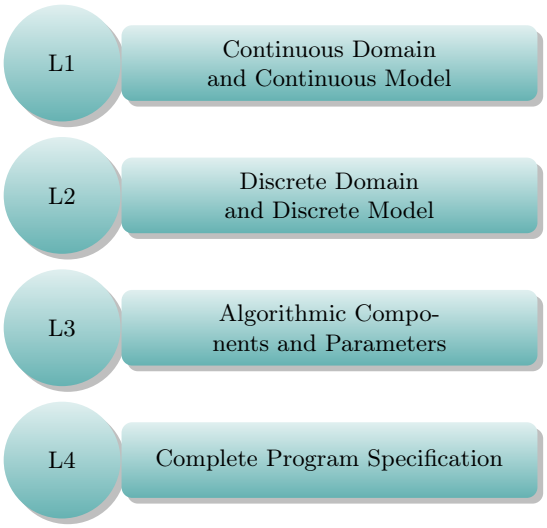

Figure 2: Different layers in ExaSlang

1.4 ExaStencils compiler

The compiler identifies the input files based on the setting read from a `.settings` file. Any of the layers L1, L2, L3, or L4 could be used as the starting layer. After parsing the input file of each layer to its representation (data structures), a series of transformations could be performed on the program. Each layer of the ExaStencils has its own set of data structures. As the transformations in each layer end, the resulted tree will be mapped to a tree in the next layer. This tree consists of nodes with types equivalent to the previous node. And the process continues until reaching the IR Layer. When the IR layer finishes handling its tasks, the program could be pretty-printed to the C++ code.

1.4.1 Transformations

In each layer of ExaStencils, the AST could be modified to transform its state into another one. These modifications are called transformation. All transformations are applied to the program in depth-first order. It is possible to apply transformation to a limited part of the program. Transformations include altering, appending, discarding, and replacing components of the program's tree. Every transformation has two parts, a pattern to search for, and the element replacing the pattern. In defining transformations, the `match` mechanism of Scala comes handy. This mechanism is called "pattern matching" and is available for checking a value against a pattern. It could be utilized using `match` expression. When a match found, it is possible to dismantle a value into its composing elements. `match` is in practice a more usable `switch` statement. For an object to be used in a match expression, it has to be of the type `case class` or `case object`. Listing 1 shows an example of transformation using pattern matching.

```
var strategy = DefaultStrategy("Simplify expressions")

strategy += Transformation( "Simplify addition", {
    case IR_Addition(ListBuffer(IR_IntegerConstant(x),
                                IR_IntegerConstant(0)))
    =>
        IR_IntegerConstant(x)} )
```

Source Code 1: Example of a transformation

Here, when an instance of `IR_Addition` is encountered, if its two operands are an integer constant `IR_IntegerConstant` and the integer value zero, it would be replaced by the integer constant (e.g., `3 + 0 => 3`).

```
object IR_GeneralSimplify extends DefaultStrategy(
                            "Simplify general expressions") {

    this += new Transformation( "Simplify zeros", {
        case IR_Addition(ListBuffer(IR_IntegerConstant(x),
                                    IR_IntegerConstant(0)))
            =>
            IR_IntegerConstant(x)

        case IR_Subtraction(ListBuffer(IR_IntegerConstant(x),
                                       IR_IntegerConstant(0)))
            =>
            IR_IntegerConstant(x)
    } )
}
```

Source Code 2: Example of a strategy

1.4.2 Strategies

Strategies are set of transformations grouped together, and are applied to the program state as the `handle()` method of each layer is executed. Transformations of a strategies of type `DefaultStrategy` would be applied in the same order as they were added. Listing 2 shows an example of this data structure.

1.4.3 Transactions

Before applying the strategy, a copy of the program state is made by opening a transaction with the `StateManager` object. If the strategy succeeds, the changes would be applied to the program; otherwise, it would be discarded and the last state of the program would be recovered.

1.4.4 StateManager

StateManager is the only object in ExaStencils that could access the program state directly. This object prevents the program state from being corrupted by simultaneous modifications performed by multiple strategies. `StateManager` provides the transaction interface and creates checkpoints. A checkpoint is a copy of the program state, that could be used in the restoration of the state when a transformation fails [9].

1.4.5 Optimizations

One of the goals of the ExaStencils project is to free the user from manually implementing the optimization required for having a high performance multigrid solver. These optimizations could be necessary each time the machine on which the program is running changes. To eliminate the need for such a time consuming and potentially reoccurring optimizations, the infrastructure developed for compiling ExaSlang into C++ and CUDA code performs several optimizations on the program before emitting the final code. As a result, the generated code is highly optimized both according to the problem in solving and the platform in use for running the solver. This approach results in a platform that allows the user to focus on the problem at hand instead of engaging in multidisciplinary operations required for developing solvers with adequate performance. To name a few, polyhedral optimizations, address pre-calculation, arithmetic simplifications, vectorization, and unrolling are among the optimizations performed by the ExaStencils compiler. These optimizations are discussed in detail in [6] and here we only take a brief look at them and refer the readers to [5].

Polyhedral optimizations are the set of nested loop transformations selected using the polyhedral model. The polyhedral model facilitates an automatic and unbiased search for the optimum loop transformation. This optimization is performed on the domain of all possible loop transformations: loop permutation, skewing, fusion, and distribution [6]. Address pre-calculation is pre-computing the expression used in indices of array accesses. It is routinely implemented as a part of commonly available compilers [1, 6]. Vectorization is the utilization of available SIMD (Single instruction, multiple data) registers of the processor. In essence, SIMD is a type of parallel computing where the execution of only a single instruction results in the calculation of several entities of an array. Availability and type of this set of instructions mainly depend on the hardware in use [4]. Unrolling is a loop transformation technique used for both increasing the chance of using the same memory region for several iterations (loop count) and to reduce the ratio of the instructions needed to be executed as a part of the control mechanism of the loop to the useful work that is performed.

Arithmetic simplifications are the focus of this work. A strategy for this purpose was already in place, as we started working on the development of an interface between CAS packages and the ExaStencils compiler. In the next chapters, we discuss these interfaces in depth.

2 SymPy Interface

In this chapter, we take a look at the developed infrastructure and the SymPy interface. We walk through the mappings required to and from SymPy. Then, we take a look at the steps for adding a new type to the supported types in the interface. In the end, we present a few examples of the expressions simplified by the SymPy interface and the steps of their mapping.

2.1 Introduction to SymPy

SymPy is an open-source full-featured general-purpose computer algebra system library written in Python. Its development started in 2005. The current version includes features such as symbolic arithmetic, polynomials, simplification, algebra, calculus, solvers, discrete mathematics, quantum mechanics, and matrices. Besides being implemented in Python, SymPy uses Python for interaction with the user. Therefore, it is considered to be an embedded (internal) domain-specific language. Using Python as the sole language in use, made SymPy a more approachable CAS package compared to the systems that use a new DSL language [7]. In this thesis, `SymPy` version 1.5.1 was used.

2.1.1 Simplification feature

We are mainly interested in the algebraic simplification capabilities of this library. It implements a variety of functions for manipulation and simplification of expressions, including trigonometric and hypergeometric expressions, rational functions, combinatorial functions, square roots, and expressions with common sub-expression [7].

To simplify an expression, `simplify` function is called. It applies several simplification methods, but the result is not guaranteed to be the simplest possible form of the expression [7]. Regardless, in this work, only the mentioned function was used, as calling other simplification functions in conjunction with `simplify` needs extra insight into the proper selection of the simplifying function based on the expression in interest, which is out of our specified domain.

2.2 Facility for interaction with SymPy

To map an ExaStencils expression to SymPy, the expression has to be inspected. Based on its data structure, the equivalent SymPy data-structure is chosen and a SymPy object would be created. To simplify the SymPy expression, `simplify` function is invoked on the object. Then, the result of simplification is mapped to ExaStencils. For mapping from SymPy to ExaStencils, the object has to be inspected and the equivalent ExaStencils data-structure has to be chosen and at the end, an ExaStencils object would be built. In this section, we discuss these steps in detail. However, we start by introducing the developed infrastructure.

2.2.1 ExaSimplifier objects

The outer-most layers which reside between ExaStencils and SymPy are the objects `L2_ExaSimplifier`, `L3_ExaSimplifier`, and `IR_ExaSimplifier`. By calling their `apply` method on an ExaStencils expression, as the first step, all of the caching objects of the SymPy interface are emptied. Then, the `applyImplementation` method is called and `applyImplementation`'s return value is returned from the apply method. Listing 3 shows the `apply` method of the object `L2_ExaSimplifier`. The `apply` method in these objects takes two arguments, an ExaStencils expression and a boolean value. If the boolean value is `true`, the result of the division of an integer by another integer could be promoted to a floating-point; otherwise, it remains integer even if it results in the loss of precision. In `applyImplementation` all the mentioned simplification steps (subsection 2.2) are performed and the resulted expression (ExaStencils expression simplified by the SymPy interface) will be returned to ExaStencils. Note that if at any of the steps an exception arises, the `applyImplementation` method would return its input expression without any simplification. `L2_VariableAccessCache` and `L2_NameNodeCache` will be discussed in the next sections.

```
def apply(expr : L2_Expression, canPromoteFromIntToFloat : Boolean)
: L2_Expression = {
    canPromoteIntToFloat = canPromoteFromIntToFloat
    L2_NameNodeCache.emptyCache()
    L2_VariableAccessCache.emptyCache()

    apply_implementation(expr)
}
```

Source Code 3: The apply method of the object L2_ExaSimplifier

2.2.2 ScalaPy

The use of the `ScalaPy` package enabled us to utilize Python libraries in scala. Since ScalaPy provides a dynamically-typed API and it performs no checking on the calls to the Python functions, all the error checking is performed in the Python interpreter. As a result, debugging the scala code would not be easy, if the problem lies in the Python code. To make working with SymPy using ScalaPy less error-prone, a wrapper class was developed that acts as an interface for the SymPy commands.

2.2.3 SymPyInterface

The class wrapping ScalaPy and exposing the features of the SymPy package that we are interested in is called `SymPyInterface`. Generally, it has two groups of members: functions that expose a SymPy (or Python) functionality, and Python-type aliases to be used in the inspection of the result of simplification. All the function names are chosen as close as possible to their SymPy counterpart. All of the Python-type alias names end with a `_t`. Note that to inspect a Python object in Scala, we have to use the Python function `isInstance`. Listing 4 shows an example of such a type-check used in `IR_MakeExa`. The columns SymPyInterface and SymPy in tables 6, 7, 8, 9, 10, and 11 show the names of all the member functions of SymPyInterface and the SymPy functionality that is exposed by them.

```
val symPy : SymPyInterface = new SymPyInterface
if (symPy.isInstance(root, symPy.string_t)) {
    return IR_StringConstant(root.toString())
}
```

Source Code 4: An example of the type-checking used in the IR_MakeExa

2.3 Mapping from ExaStencils to SymPy

To map an ExaStencils expression to a SymPy expression, `L2_MakeSymPy`, `L3_MakeSymPy`, and `IR_MakeSymPy` objects are used. These objects are explained in section 2.3.1.

2.3.1 Inspecting the input ExaStencils expression

To enable pattern matching using `match`, all the types used in the building of the AST of expressions in ExaStencils are defined as `case class` or `case object`. For inspection of ExaStencils objects from the layers `L2`, `L3`, and `IR`, three inspector objects were implemented. These objects are called `L2_MakeSymPy`, `L3_MakeSymPy`, and `IR_MakeSymPy`. The `apply` method of each of the inspector objects accepts an object of generic type `T` and returns an instance of type `ScalapyAny` (alias for the type `me.shadaj.scalapy.py.Any`). Then, using pattern matching, the type of the input object was identified. In cases where the expression contains sub-expressions, the sub-expression will be passed to the

`apply` method for inspection. After collecting all the parts of an expression, a SymPy expression using the resulted parts is built and returned from the `apply` method of the inspector object.

2.3.2 Choosing the equivalent SymPy data-structure and building a SymPy object

If the `match` expression in the `apply` method finds the type of its input expression, an equivalent SymPy object would be returned. Tables 6, 8, and 10 show the mapping of ExaStencils expressions to SymPy. The first column (ExaStencils) shows the ExaStencils expression type, the second one (SymPyInterface) is the name of SymPyInterface method called for building the SymPy expression. The last column (SymPy) shows the type of the resulted SymPy expression. Note that in some cases the result is not a SymPy expression, but a Python built-in type.

Since SymPy does not provide the `Subtraction` class but implements the subtraction by adding left-hand operand with the negated right-hand operand, the same method was used in the mapping. The same situation is true for the division. SymPy does not provide `Division` class. Instead, it uses the multiplication of left-hand operand by the right-hand operator to the power of -1. In the case of element-wise matrix operations, similar treatment is needed. Here, in place of `Add` and `Mult`, their element-wise (Hadamard) equivalents `hadamard_product` and `hadamard_power` are used.

In addition, no direct mapping from ExaStencils to SymPy for `IR_VariableAccess`, `L2_PlainVariableAccess`, `L2_LeveledVariableAccess`, `L3_PlainVariableAccess`, and `L3_LeveledVariableAccess`, is available. As a result, they are cached, and a `Symbol` is used in their place. To make the `Symbol`, an auto-generated name is used. Listing 5 shows how caching is done for variable access in L2.

```
val name = L2_VariableAccessCache.cacheIt(varAccess)
symPy.symbol(name)
```

Source Code 5: Caching performed for variable accesses in L2

By calling the `cacheIt` method, a copy of the `varAccess` is saved in `L2_VariableAccessCache` and a name is returned to be used for SymPy `Symbol`. The name is deterministic, meaning that for variable access objects with matching ExaStencils-class, similar variable names, and data-type, the same name would be generated by `L2_VariableAccessCache`, `L3_VariableAccessCache`, and `IR_VariableAccessCache`. This deterministic naming enables SymPy to recognize the similar variable accesses as the same `Symbol` and to performs simplifications on them (e.g., canceling and factoring variables).

```
functionCall : L2_FunctionCall
    =>
    val functionName = functionCall.function.name
    val args = functionCall.arguments
    functionName match {
        case "sin" => symPy.Sine(L2_MakeSymPy(args.head))
        case "cos" => symPy.Cosine(L2_MakeSymPy(args.head))
    }
```

Source Code 6: The implementation for sin, and cos functions in L2

Special treatment was required for function call expressions: `L2_FunctionCall`, `L3_FunctionCall`, and `IR_FunctionCall`. In these cases, based on the name of the function, the respective SymPy function call was used. Listing 6 shows the implementation of `sin` and `cos` functions in L2.

As it is shown, the name of the function to be called is obtained using `functionCall.function.name`. Then, using `match` expression, the appropriate SymPy function call is returned. Tables 7, 9, and 11 show the mapping from ExaStencils to SymPy. Note that if a function call was encountered, but the function name had no match, an exception would be thrown.

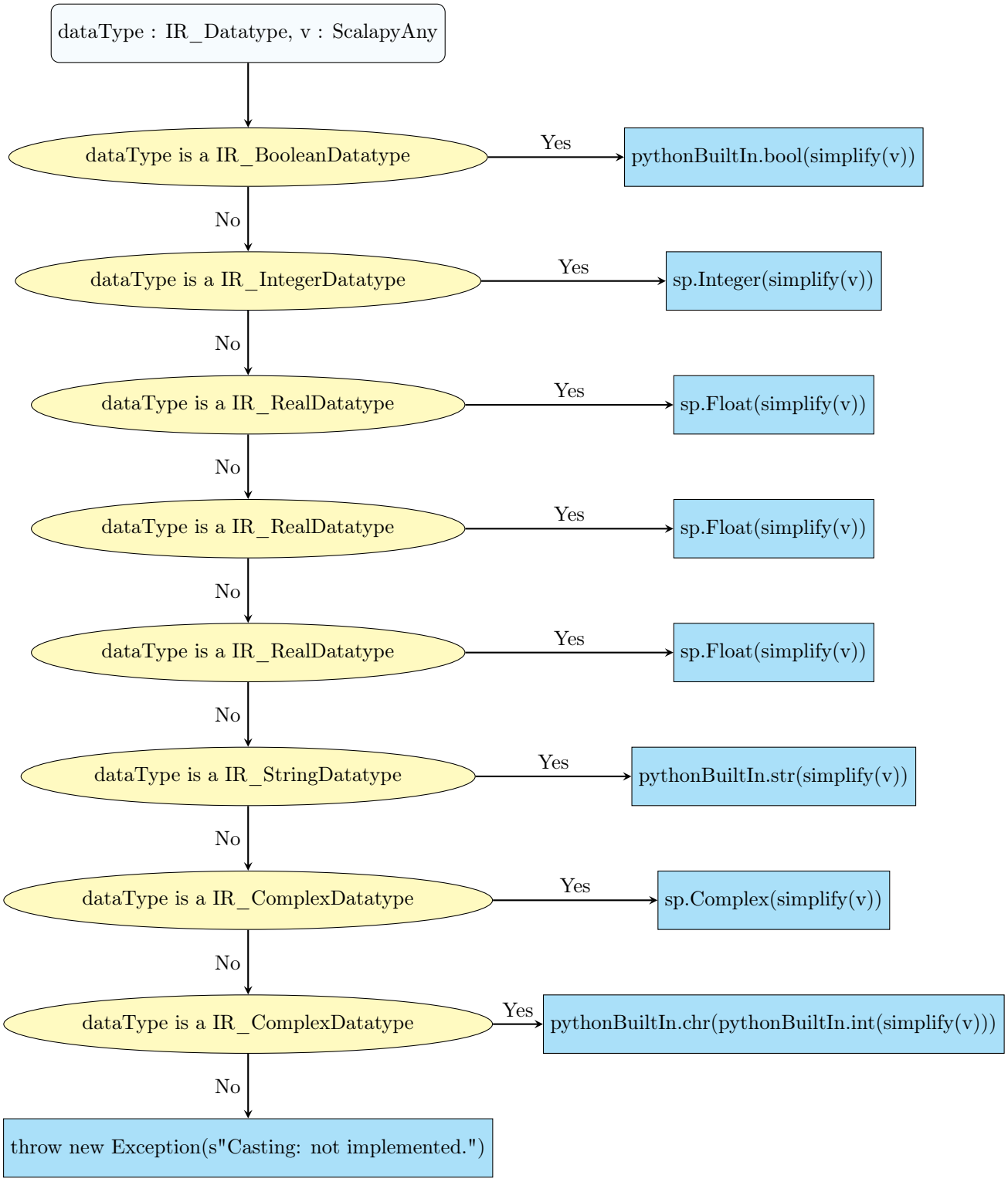

Figure 3: The decision tree used in the implementation of IR_Cast in the SymPyInterface class

Another special case is `IR_Cast` expressions. Casting exists only in the IR layer. In this case, at first, the expression to be cast would be mapped to SymPy. Then, based on the targeted data-type, the casting of SymPy expression is performed. Table 12 shows the ExaStencils target data-type and the SymPy casting used for each case. The SymPy expression would be checked for its type only in casting to `IR_CharDatatype`. If the input is of a non-integral type, it would be cast to a Python `int`, then the integer would be cast to a Python character using `chr`. Figure 3 shows the decision tree used in the implementation of `IR_Cast` in the SymPy Interface class. In cases, where the `match` expression in the `apply` method does not find the type of its input expression, a SymPy `Symbol` with an auto-generated name would be returned and the input object would be cached using name-node cache objects. Name-node cache objects are `L2_NameNodeCache`, `L3_NameNodeCache`, and `IR_NameNodeCache`. They provide a caching mechanism for the objects that have no equivalent in SymPy as illustrated for the L2 layer in listing 7. When calling the `cacheIt` method with the object passed as an argument, `L2_NameNodeCache` saves a copy of the object and returns an auto-generated unique name, to be used in the definition of a SymPy `Symbol`. This symbol would be used in place of the unmappable expression. The cached object could be retrieved from the name-node cache, using `getCachedObject` method of the caching object (in this example, `L2_NameNodeCache.getCachedObject`).

```
val name = L2_NameNodeCache.cacheIt(otherTypes)
symPy.symbol(name)
```

Source Code 7: Treatments for objects of L2 layer without any equivalent in SymPy

2.3.3 Invoking simplify function on the SymPy expression

For calling SymPy `Simplify`, `simplifyDoIt` method is provided in `SymPyInterface`. Calling it will run the SymPy `simplify` and then `doit` with the argument `deep=true`. `doit` will evaluate all the objects that have not been evaluated yet [11]. If an exception arises when performing the simplification, `simplify` without `doit` will be used. Listing 8 shows the implementation of `simplifyDoIt`.

```
try {
    sp.simplify(sp.simplify(expr).doit(deep = true))
} catch {
    case ex : Throwable =>  sp.simplify(expr)
}
```

Source Code 8: Implementation of simplifyDoIt

2.4 Mapping from SymPy to ExaStencils

To map the simplified SymPy expression to ExaStencils, `L2_MakeExa`, `L3_MakeExa`, and `IR_MakeExa` objects are developed. The `apply` method in these objects takes two arguments, an instance of `me.shadaj.scalapy.py.Any` and a boolean value. If the boolean value is `true`, the result of the division of an integer by another integer could be promoted to a floating-point; otherwise, it remains integer even if that means loss of precision. The `apply` method in turn calls the `applyImpl` method. It is in `applyImpl` that the object would be inspected and its ExaStencils equivalent would be created.

2.4.1 Inspecting the input SymPy expression

As it was already discussed in 2.2.3, the only way to determine the type of a ScalaPy object in Scala is by using `isInstance` function of Python. The comparison is performed against the type-alias members of `SymPyInterface`. Table 13 shows the aliases and the actual SymPy expression type they represent. When a match is found, the object would be further inspected, and ExaStencils expression would be constructed. Listing 9 shows the operations performed for an instance of SymPy `Mult`. Here, arguments of `Mult` have to be unwrapped, since `args` is not of type Scala-List, but the `Dynamic` type. By unwrapping it, a list of `Dynamic` instances is made. After getting all the factors of the multiplication object, factors are mapped to a list of ExaStencils expressions by calling `L2_MakeExa.applyListBuffer` on them. Afterward, the list is used for constructing an instance of `L2_Multiplication`. Note that in the for loop we have to cast the return value of `len` function (Python built-in) to `String` and then to `Int`. Since the return value of `len` is an instance of `Dynamic`, and not an integer, we have to unwrap it first (listing 9).

```
if (symPy.isInstance(root, symPy.mul_t))
{
    val tempRoot = symPy.simplify(root)
    val tempArgs = symPy.simplify(tempRoot.args)
    var listArgsDynamic = new ListBuffer[Dynamic]
    for (i <- 0 until
              symPy.pythonBuiltIn.len(tempArgs).toString().toInt)
    {
        listArgsDynamic += tempArgs.arrayAccess(i)
    }
    val listBufferArgs = L2_MakeExa.applyListBuffer(listArgsDynamic).
                                    to[ListBuffer]

    return new L2_Multiplication(listBufferArgs)
}
```

Source Code 9: Mapping of SymPy multiplication to ExaStencils

2.4.2 Choosing the equivalent ExaStencils data-structure and building an ExaStencils object

To choose the appropriate ExaStencils data type, the reverse of what already were discussed in 2.3.2 has to be performed. Depending on the layer, one of `L2_MakeExa`, `L3_MakeExa`, or `IR_MakeExa`, is used for mapping a SymPy expression to its equivalent expressions in ExaStencils.

```
val symbolName = root.toString()
val result =
        if (L2_NameNodeCache.hasIt(symbolName)) {
                L2_NameNodeCache.getCachedObject(symbolName).get
        } else if (L2_VariableAccessCache.hasIt(symbolName)) {
                L2_VariableAccessCache.getCachedObject(symbolName).get
        } else {
                throw new Exception("COULD NOT FIND ANY RELEVANT CACHED SYMBOL!")
        }
return result
```

Source Code 10: Mapping a SymPy symbol to ExaStencils

To do the mapping, the SymPy expression is passed to the `apply` method of the MakeExa object along with a boolean value that flags whether the promotion of the result of integer values divisions to floating-point is allowed or not. The input ExaStencils expression is checked against type-aliases available in the `SymPyInterface` and when a match is found, the mapping starts. If no match were found, an exception would be thrown. The mapping is straight forward, except for the `Symbols`. In this case, in the beginning, the `NameNodeCache_` objects of the layer would be checked. If it has the name, the object cached with the name would be retrieved using `getCachedObject`. When calling `getCachedObject`, the name-object pair will be removed from `NameNodeCache_`. If the name was not in the `NameNodeCache_`, `VariableAccessCache_` would be searched for it. If it contains the name, the object cached with it will be retrieved, but the name-object pair will not be removed from `VariableAccessCache_`, since the same variable-access might appear several times in an expression. In cases, when non of the caches has the name, an exception would be raised. Listing 10 shows the case for L2 layer.

2.5 Expanding the SymPy interface for new types

To expand the existing interface, such that it supports a new type, in the beginning, the relevant builder method has to be added to the `SymPyInterface` class. This method basically makes a call to the constructor of the relevant SymPy type. Since the ScalaPy exposes all the types from the SymPy package, the type names would not change. Then, the type alias is added to the `SymPyInterface` class. As it is shown in listing 11,, the name of the type to be used is similar to the name used in the builder method. This type alias is later used in the inspection of the result of simplification to map them to ExaStencils. The naming scheme used for the type aliases is `type-name + _t`. Here, we check the case of `integer` (listing 11).

```
def integer(v : Float) : ScalapyAny = sp.Integer(v)
val integer_t : ScalapyAny = sp.Integer
```

Source Code 11: Making an instance of SymPy's Integer

The function `integer` makes an instance of `Integer` (a SymPy class) and returns it. The `integer_t` acts as an alias to the `Integer` type. The next step is adding the mapping from ExaStencils to SymPy. To do so, adding a `case` to the `match` statement in the `apply` method of the MakeSymPy object is all needed for the mapping. For the primitive types of ExaStencils, only a call to the method added on the previous step and passing the value to it is required. For more complicated types, for example, `L2_Addition`, a call to the `apply` method of the MakeSymPy object is necessary to make sure that all the arguments are already mapped to SymPy (listing 12).

```
case operation : L2_Addition
    =>
    symPy.add(L2_MakeSymPy(operation.summands))
```

Source Code 12: Mapping L2_Addition to SymPy

To extend the mapping from SymPy to ExaStencils, `def applyImpl(root : ScalaPyAny)` has to be modified. The decision-making mechanism here is a series of `if ... else` statements. To determine the type of the expression's elements, the function `isInstance` from Python is used. The comparison is performed against the type alias we added before to the `SymPyInterface` object. Note that to access the data members of a SymPy expression in Scala code, the access procedure similar to what we could do in a Python code using the SymPy package, would serve the cause since ScalaPy exposes all the elements of Python objects.

As listing 13 shows, after finding a match using `isInstance`, mapping the elements of SymPy expression to ExaStencils expression has to be done by calling MakeExa's apply or in case of a list, `applyListBuffer`. The steps mentioned in this section show the most common cases and additional steps might be needed for exceptionally more complex data-types.

```
if (symPy.isInstance(root, symPy.add_t)) {
    val tempRoot = symPy.simplify(root)
    val tempArgs = symPy.simplify(tempRoot.args)
    var listArgsDynamic = new ListBuffer[Dynamic]
    for (i <- 0 until symPy.pythonBuiltIn.len(tempArgs).
                                        toString().toInt)
    {
        listArgsDynamic += tempArgs.arrayAccess(i)
    }
    val listBufferArgs =
        L2_MakeExa.applyListBuffer(listArgsDynamic).
                                                to[ListBuffer]
    return new L2_Addition(listBufferArgs)
}
```

Source Code 13: Mapping the emelents of SymPy expression to ExaStencils

2.6 Sample results

In this section, the simplification of two expressions and the results of each step are presented. The examination of the quality and performance of the SymPy interface is not a part of this section and is discussed in chapter 4.

Trigonometric expression

Figure 4 shows the tree representing the expression $(sin(i0) + cos(i0))^{2.0}$ in ExaStencils.

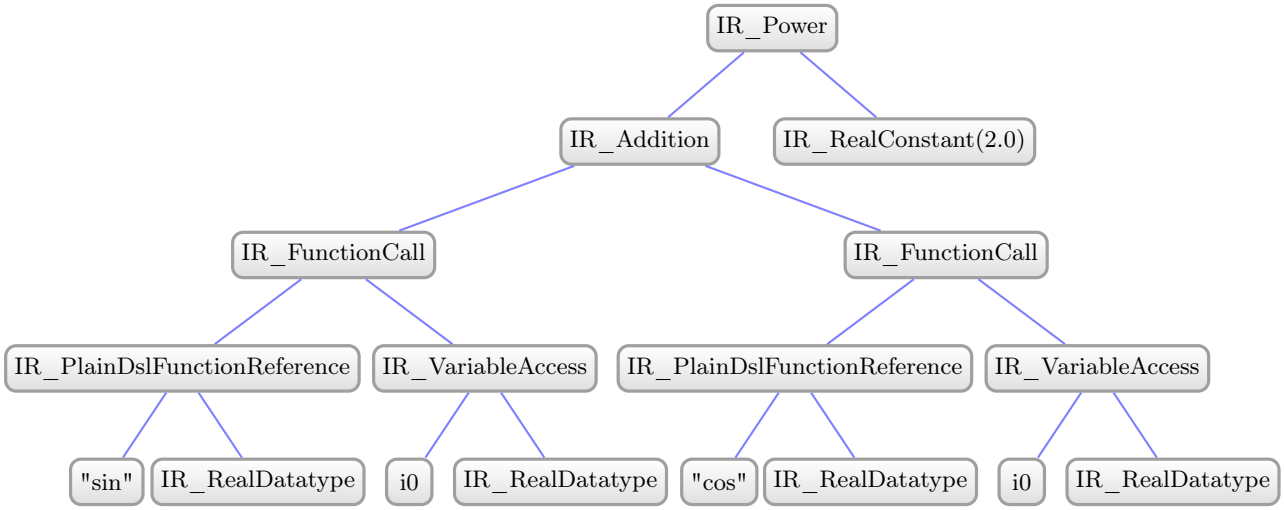

Figure 4: ExaStencils expression used as input for the SymPy interface (trigonometric expression)

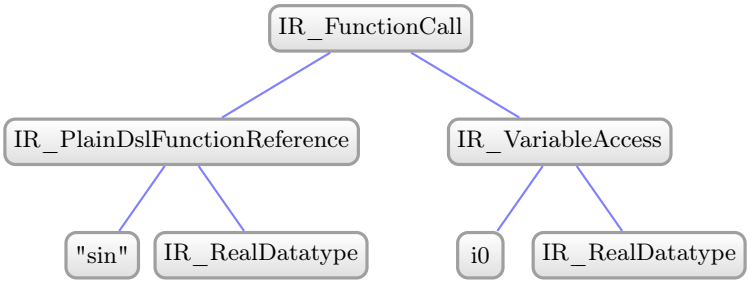

Figure 5: Actual structure of a function call expression in ExaStencils representing $sin(i0)$

Since the details of the data structure used in `IR_FunctionCall` do not change in this text (shown in figure 5), a simplified version of the structure (shown in figure 6) is used in the future trees.

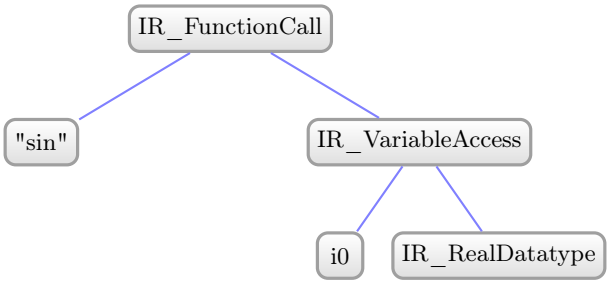

Figure 6: Simplified structure for function call expression used in this text (representing the expression $sin(i0)$)

After mapping the expression to SymPy we get the expression $(sin(i0) + cos(i0))^{2.0}$. For simplicity, the mangled name `_AUTO_GEN_NAME_IR_VA__i0__________________IR_RealDatatype_` is replaced by the original variable name `i0`. Then, the SymPy expression is simplified by passing it to the `simplifyDoIt` method of the class `SymPyInterface`'s instance. The result of this call is $2.0 * sin(i0 + pi/4) * *2.0$. After simplification, the resulted SymPy expression was mapped back to the ExaStencils, as shown in figure 7.

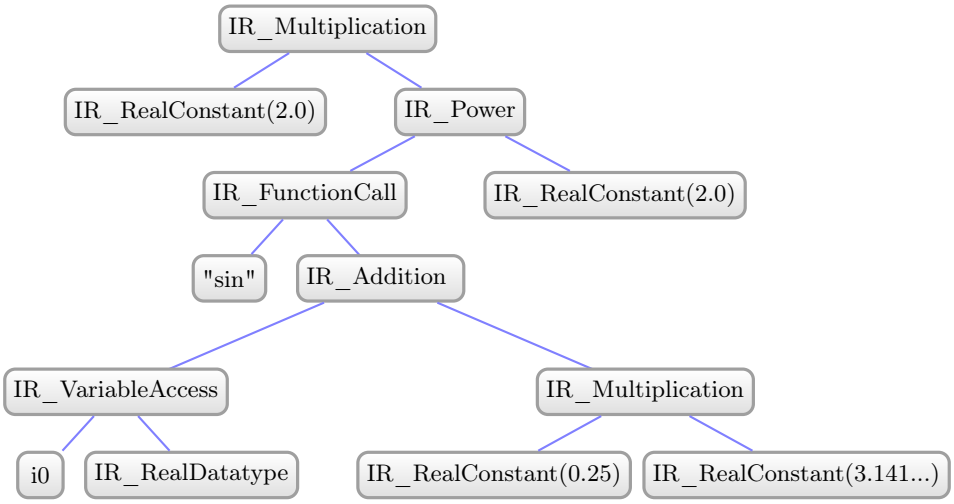

Figure 7: The result of mapping the simplified expression to ExaStencils (trigonometric expression)

Polynomial expression

Next example shows the simplification of the $(i1 + 2.0)(i1 + 2.0) * i1$ expression. Figure 8 shows the ExaStencils representation of this expression.

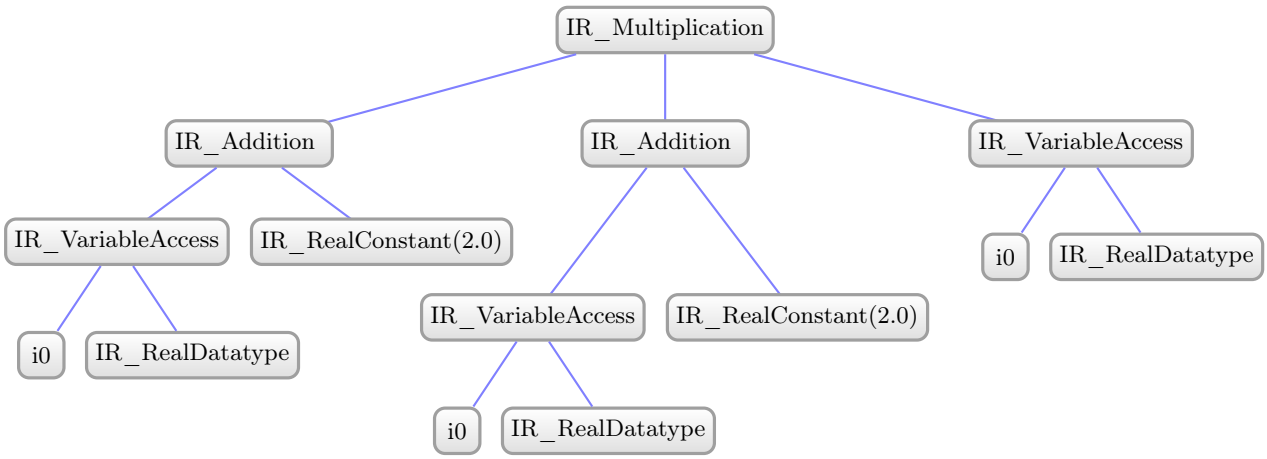

Figure 8: ExaStencils expression used as input for the SymPy interface (polynomial expression)

The result of mapping to SymPy is $(i1+2.0)*(i1+2.0)*i1$. After simplification, $4.0*i1*(0.5*i1+1)**2$ is returned. The simplified expression was mapped back to the ExaStencils and is shown in figure 9. The mentioned steps are performed for every expression simplification by the SymPy interface. Here, the analysis of performance and quality are skipped. This analysis as well as comparison between the two interfaces are presented in chapter 4.

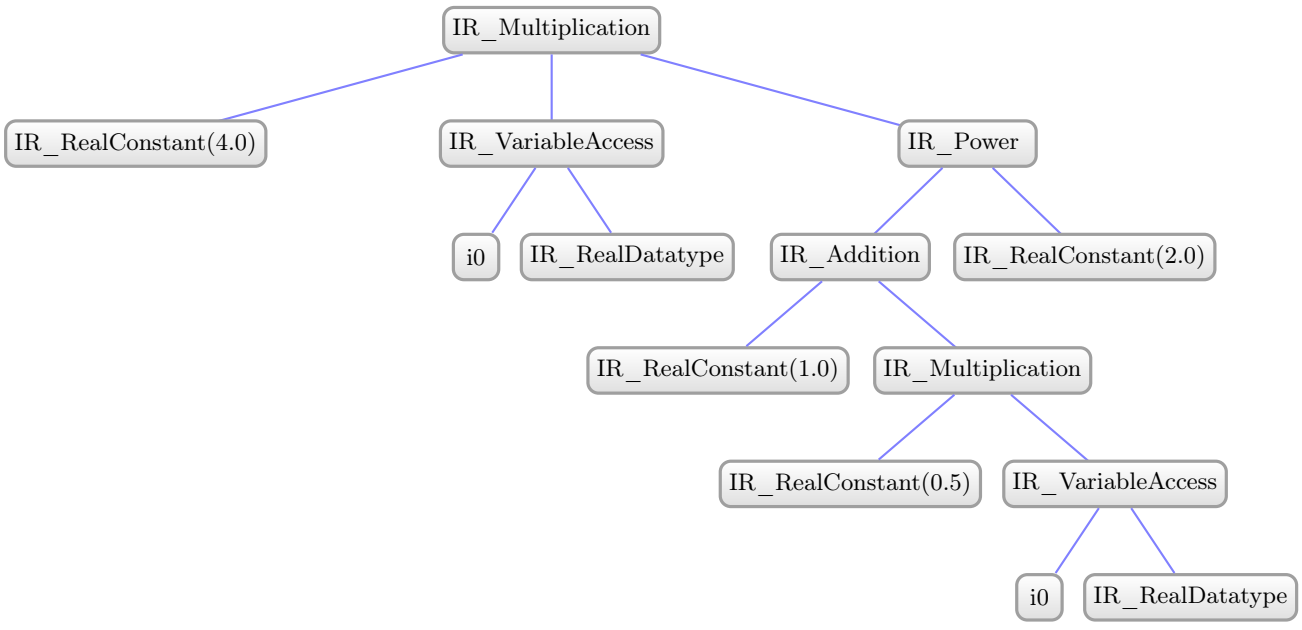

Figure 9: The result of mapping the simplified expression to ExaStencils (polynomial expression)

3 Maxima Interface

This chapter is dedicated to the Maxima interface, its design and implementation. At first, we explain the mapping from ExaStencils to and from Maxima. Then, we shortly explain the steps for adding the support for a new type to the interface. After that, we present two expressions as examples of the mappings and simplifications. To interface ExaStencils with Maxima the following steps are necessary; mapping from ExaStencils to Maxima, and mapping from Maxima to ExaStencils. Details of each step are discussed in the next sections, but first we take a look at Maxima, and the packages required for interacting with it in Scala.

3.1 Introduction to Maxima

The development of Maxima started at the mid-1960s at MIT under the name Macsyma, MAC's SYmbolic MAnipulator. The language used in the development was Lisp. In the early 1980s, the United States Department of Energy as one of the sponsors of the consortium developing Macsyma software made a free version of it, called Maxima, available for everyone. Maxima is currently available under GNU General Public License (GPL). After the release of Maxima, further developments and improvements were done on the Macsyma which is not available in the free version [8].

3.2 Facility for interaction with Maxima

3.2.1 ExaSimplifierMaxima objects

The ExaSimplifierMaxima objects are on the forefront of interfacing ExaStencils with Maxima, `L2_ExaSimplifierMaxima`, `L3_ExaSimplifierMaxima`, and `IR_ExaSimplifierMaxima`. By passing an ExaStencils expression to the `apply` method of these objects along with a boolean value, the simplification of expressions using Maxima is achievable. The boolean value signifies the possibility of promoting integers to float in the division of two integers. The ExaSimplifierMaxima uses Maxima to simplify the expression and returns the resulted ExaStenctils expression.

Analogous to the SymPy interface, in the Maxima interface caching is used for variable-access objects and for the cases where a direct mapping is not possible. The `apply` method clears the contents of both cache objects. Listing 14 shows the implementation of `apply` method. As it is shown, `apply` calls `applyImplementation` after clearing cache objects. `applyImplementation` performs the simplification steps. In case of successful execution of simplification, the resulted ExaStencils expression is returned. Similar to SymPyInterface, if an exception arises in any of the steps, the `applyImplementation` method would return its input expression without any simplification. This policy enables the interface to avoid crashes.

```
def apply(expr : L2_Expression, canPromoteFromIntToFloat : Boolean)
: L2_Expression = {
    canPromoteIntToFloat = canPromoteFromIntToFloat
    L2_NameNodeCache.emptyCache()
    L2_VariableAccessCache.emptyCache()
    val result = applyImplementation(expr)
    result
}
```

Source Code 14: Implementation of apply method

3.2.2 ABCL package

Armed Bear Common Lisp, commonly known as ABCL, is a package featuring a full implementation of the Common Lisp language. It provides a compiler and an interpreter and runs in the JVM [3].

The part we are interested in for the current work is the Java to Lisp integration APIs that it provides. This package is distributed as a single jar file. To use this package in the Scala code, the `libraryDependencies += "org.abcl" % "abcl" % "1.6.1"` could be added to the `build.sbt` file.

To access the ABCL, `org.armedbear.lisp` package has to be imported and a reference to an instance of interpreter could be attained by calling [3] as is shown in listing 15.

```
Interpreter interpreter = Interpreter.createInstance();
```

Source Code 15: Getting an instance of lisp interpreter (ABCL)

`createInstance()` is an static method. Note that ABCL does not allow multiple instances of `Interpreter` to exist in each instance of JVM. Therefore, only the first call to the `createInstance` would return a valid reference and the next calls would return `null`. As a result, in the `MaximaInterface` class a static member is defined to keep a reference to the `Interpreter` (listing 16). If it is `null`, first we try to get a reference to the interpreter by calling `Interpreter.getInstance()`. In case it returned `null` a call to the `interpreter.createInstance` method would be made. The interpreter object exposes the `eval` method, which could be used for the evaluation of an arbitrary string containing lisp code. In the `MaximaInterface` we used this method for loading `asdf`, and `maxima` packages. `asdf` is a build system that is used to define the way a Common Lisp program is constructed from its components [2].

Another class from the ABCL package that is used in this work is `Package`. An instance of `Package` exposes `findPackage`, and `findAccessibleSymbol` methods. Given the name of the package as a string, the `findPackage` method looks for it in the current lisp thread and returns a reference to the package object if it finds it. `findAccessibleSymbol` method searches the package for a `Symbol` with the given name. If it succeeds, a reference to the `Symbol` would be returned. Otherwise, the result would be `null`. It should be noted that `Symbol` could be a variable, flag, or a function. If it is a function, calling the `getSymbolFunction` method would return an instance of `LispObject` that could be cast into a `Function`. `Function` allows users to make a call using the `execute` method. The arguments of `execute` have to be of type `LispObject`. If a `Symbol` is in fact a variable, its value could be set using its `setSymbolValue` method or retrieved using `getSymbolValue` method.

3.2.3 MaximaInterface

The class `MaximaInterface` is implemented in Java since the ABCL is a Java package. To lower the cost of construction for instances, three static members were defined as is shown in listing 16. Only calling the constructor of `MaximaInterface` for the first time would initiate them. In case of failure to initialize any of these variables, an exception will be thrown.

Listing 16 shows the constructor of `MaximaInterface` class. The main infrastructure of `MaximaInterface` consists of several methods, which are developed to take the name of the desired Maxima-expression to be built and a list of arguments to be passed for the construction of the expression.

```
private static Interpreter  interpreter         = null;
private static Package      maxima_package      = null;
private static Package      common_lisp_package = null;

MaximaInterface() {
    if (interpreter == null) {
        interpreter = Interpreter.getInstance();
    }
    if (interpreter == null) {
        interpreter = Interpreter.createInstance();
        assert (interpreter != null);
    }
    if (maximaPackage == null) {
        interpreter.eval("(require 'asdf)");
        interpreter.eval("(asdf:operate 'asdf:load-op :maxima)");
        maximaPackage = Packages.findPackage("MAXIMA");
        assert (maximaPackage != null);
        // setting the flags for symplifications
        maximaPackage.findAccessibleSymbol("$SIMPSUM").
        setSymbolValue(JavaObject.getInstance((Boolean) true, true));
        maximaPackage.findAccessibleSymbol("$SIMP").
        setSymbolValue(JavaObject.getInstance((Boolean) true, true));
    }
    if (commonLispPackage == null) {
        commonLispPackage = Packages.findPackage("CL-USER");
    }
}
```

Source Code 16: The constructor of class MaximaInterface and the static members of this class

Listing 17, shows the implementation of the simplest of these methods, `callFunction1ArgImplementation`, which could be used for constructing a Maxima expression, in this case, a function call with only one argument. This method returns a `LispObject`. As it is shown, to build a Maxima expression, a call to a function-symbol is done. In the first step, accessible symbols of Maxima package are searched for the name of the function, here stored in `functionName`. In case of failure in finding it, symbols of common-lisp would be searched. Then, the function related to the symbol is acquired by calling `getSymbolFunction` method of the symbol. After building the equivalent lisp expression of the argument of function by calling `makeLispEquivalent` on `arg`, the expression is created by passing `argInLisp` to the `execute` method of the function object, `symbolFunction`.

To reduce the chance of making mistakes, for each Maxima expression type that is used in the interface, a method was added to the `MaximaInterface` class. This set of methods mostly act as a wrapper that makes a call to functions similar to `callFunction1ArgImplementation` and pass the arguments as well as the correct function name as a string to them. Listing 17 shows the implementation of `string` method as an example of this group of methods.

```
<T> LispObject callFunction1ArgImplementation(T arg, String functionName)
throws Exception
{
    // find the type (return it as a symbol) among symbols of the
    // maxima package
    Symbol expressionTypeSymbol =
        maximaPackage.findAccessibleSymbol(functionName);
    // if not found, look in the common-lisp
    if (expressionTypeSymbol == null) {
        expressionTypeSymbol =
            commonLispPackage.findAccessibleSymbol(functionName);
    }
    // throw if not found!
    if (expressionTypeSymbol == null) {
        throw new Exception("expressionTypeSymbol == null");
    }
    // convert the lisp equivalent of the input argument
    LispObject argInLisp = makeLispEquivalent(arg);
    // get the function corresponding to the symbol
    Function symbolFunction =
        (Function) expressionTypeSymbol.getSymbolFunction();
    // throw if not found
    if (symbolFunction == null) {
        throw new Exception("symbolFunction == null");
    }
    // execute the function, while passing the argument to it
    LispObject result = symbolFunction.execute(argInLisp);
    // return the result of the execution (a lisp/maxima expression)
    return result;
}

<T> LispObject string(T v) throws Exception {
    return callFunction1ArgImplementation(v, "STRING");
}
```

Source Code 17: The method callFunction1ArgImplementation and its use case

3.3 Mapping from ExaStencils to Maxima

Although the names are not always similar, in most cases a direct mapping from ExaStencils to Maxima is possible. And in cases with different structures, the mappings are done using the closest data structure available in Maxima. Tables 14, 15, and 16 show ExaStencils structures and their equivalent Maxima types used in the mapping.

3.3.1 Inspecting the input ExaStencils expression

The mechanism of inspecting the ExaStencils expression for the purpose of making a Maxima expression is similar to what we used for SymPy interface. By passing an ExaStencils expression to the `apply` method of MakeMaxima objects (`L2_MakeMaxima`, `L3_MakeMaxima`, and `IR_MakeMaxima`), the expression would be checked against all the ExaStencils types, mappable to Maxima. As a match was found, the equivalent Maxima expression would be generated by calling the relevant method of the `MaximaInterface` class instance. Special treatments are needed for variable-access objects and function call instances, analogous to the system used in MakeSymPy objects.

3.3.2 Choosing the equivalent Maxima data-structure and building a Maxima object

As a match in data-type has been found in the MakeMaxima objects, a call to `MaximaInterface`'s methods would be made. Tables 14, 15, 16, show ExaStencils expression and the method to be called. For the function call objects based on the name of the function, accessible from `functionCall.function.name` field of the object, the proper method of `MaximaInterface` would be chosen. Tables 14, 15, and 16 show the mapping from ExaStencils expression data-types to Maxima.

3.3.3 Invoking simplify function on the Maxima expression

As the Maxima object gets completed and returned to the `applyImplementation` method of ExaSimplifierMaxima object, the `DO_SIMPLIFICATION` method of `MaximaInterface` is called on the Maxima expression. The resulted simplified expression would be passed to the `Make_MIR` object for inspection and an intermediate representation would be constructed.

3.4 Mapping from Maxima to ExaStencils

The inspection of Maxima (ABCL) objects is done using an instance of the `Make_MIR` class. Then, the resulted MIR object would be further inspected and used for making the final ExaStencils expression.

3.4.1 Inspecting the input Maxima expression and building the Maxima intermediate representation (MIR)

Compared to the SymPy interface, this extra layer of abstraction was added to the Maxima interface. In SymPy, the resulted Python object was inspected and its corresponding ExaStencils expression was constructed immediately. However, in the Maxima interface, at first, a MIR object is constructed. Then, the correlative ExaStencils expression is built by inspecting the MIR object. This discrepancy is due to differences between the structure of expressions in SymPy and Maxima. Unlike SymPy, in Maxima an expression's operator or function could be wrapped along with hints for simplification and other controlling flags. e.g., `3 + 7` could be expressed as `((MAXIMA::MPLUS SIMP) 3 7)`.

The `Make_MIR` class is developed in Java to inspect Maxima expressions and then build a MIR object. The method `makeMIR` of this class accepts an instance of `ListObject` as an argument. The inspection starts with getting the name of the data-type as a string by calling the `getClass().getSimpleName()` method of the object under inspection. Listing 18 shows the implementation used for getting data-type as a string.

```
public String getClassName(LispObject obj) {
    return obj.getClass().getSimpleName();
}
```

Source Code 18: Getting data-type as a string

If the returned string equates to `"Fixnum"`, `"Bignum"`, `"DoubleFloat"`, or `"SimpleString"`, a MIR object would be constructed and returned. However, for `"SimpleVector"`, `"Simple-Vector"`, `"Symbol"`, `"simple-vector"`, and `"Cons"`, a more elaborate inspection and a more complicated mechanism are required. When non of the strings matched, an instance of the `MIR_Base` class with the name `"!!! - NOT HANDLED - !!!"` would be returned. This object would later causes an exception to be raised in the MakeExa objects. Therefore, the simplification would be aborted and the original ExaStencils expression would be returned as the result. For types resembling of a vector all the elements would be passed to `makeMIR` and the results would be collected. Then, an instance of `MIR_SimpleVector` would be constructed and returned. If the string value is `"Symbol"`, the `getName` method would extract its name and based on its value one of the following would happen. For "T, "COMMON-LISP:T", and "true" an instance of `MIR_Boolean(true)` and for "F", "false", "NIL", and "COMMON-LISP:NIL" a `MIR_Boolean(false)` would be returned. If the `Symbol`'s name is non of the above, an instance of

MIR_Symbol would be returned. It is important to note that an extra check is required to determine if a Symbol is in fact an internal Maxima symbol, Lisp symbol, or a user-defined one. This checking mechanism is shown in listing 19. In this listing knownMaximaSymbols is a list of all the known internal symbols.

```
final String[] knownMaximaSymbols =
{   "MAXIMA:SIMP", "MAXIMA:RATSIMP", "MAXIMA:FACTORED", "MAXIMA:IRREDUCIBLE",
    //
    "MAXIMA:MTIMES", "MAXIMA:MPLUS", "MAXIMA:RAT",
    //
    "Symbol", "Cons",
    // POWERS, ROOTS, AND LOGS
    "MAXIMA:MEXPT", "MAXIMA:%SQRT", "MAXIMA:%LOG10", "MAXIMA:%LOG",
    // Trig. Func.
    "MAXIMA:%SIN", "MAXIMA:%COS", "MAXIMA:%TAN", "MAXIMA:%COT",
    "MAXIMA:%SEC", "MAXIMA:%CSC", "MAXIMA:%ASIN", "MAXIMA:%ACOS",
    "MAXIMA:%ATAN", "MAXIMA:%ACOT", "MAXIMA:%ASEC",
    "MAXIMA:%ACSC", "MAXIMA:%SINH", "MAXIMA:%COSH", "MAXIMA:%TANH",
    "MAXIMA:%COTH", "MAXIMA:%SECH", "MAXIMA:%CSCH", "MAXIMA:%ASINH",
    "MAXIMA:%ACOSH", "MAXIMA:%ATANH", "MAXIMA:%ACOTH",
    "MAXIMA:%ASECH", "MAXIMA:%ACSCH",
    //
    "MAXIMA:MABS",
    //

    "COMMON-LISP:T", "T", "true"
};

boolean isMaximaSymbol(LispObject obj) {
        return Arrays.asList(knownMaximaSymbols).contains(obj.toString());
}

public MIR_Base makeMIR(LispObject obj) {

    /* ... some code here! ... */

    return new MIR_Symbol(((Symbol) obj).getName(), isMaximaSymbol(obj));
}
```

Source Code 19: Determining if a Symbol is in fact an internal Maxima or Lisp symbol, or a user defind one

MIR classes are used by reason of the complexity in expressions and mainly the possibility of multi-layer wrapping of operators in Maxima. Since the Maxima package is implemented in Lisp language, it inherited one of the key features of Lisp, expressions are in fact a list. Here an expression like `80 * i1` (represented with `MAXIMA::MTIMES`) could have the form shown in listing 20. In this expression, `MAXIMA::SIMP` is a `Simplification flag` (internally is implemented as a boolean `Symbol`), showing possibility of performing simplifications on the expression [10]. Setting the `MAXIMA::SIMP` to `false`, would disable the simplifications of expressions. To separate the complexity of removing these types of flags from expressions (list of operator and operand) and unwrapping the operators, the Maxima intermediate representation (MIR) is used.

```
((MAXIMA::MTIMES MAXIMA::SIMP MAXIMA::RATSIMP) 80 MAXIMA::|i1|)
```

Source Code 20: Possible format of an expression in Lisp

3.4.2 Choosing the equivalent ExaStencils data-structure and building an ExaStencils object based on MIR tree

The inspection of MIR objects is done in the MakeExa objects (`L2_MakeExa`, `L3_MakeExa`, `IR_MakeExa`). Since all the MIR classes are `case class`, use of the `match` mechanism is possible. As it is shown in fugure 10 (a shortened form the decision tree used in the implementation of `L2_MakeExa`'s `apply` method), for an input MIR object four different scenarios are possible.

If the object is an instance of a primitive type (`MIR_FixnumInteger`, `MIR_Bignum`, `MIR_String`, `MIR_Boolean`, and `MIR_DoubleFloat`), an instance of its equivalent L2 type is made without performing any other operation. In the case of `MIR_Symbol`, similar to the SymPy interface, first the caches are searched for its name. When any of the `_NameNodeCache` or `_VariableAccessCache` contains the name, the object associated with the name is retrieved. In case of failure in finding the name, an exception is thrown. For instances of `MIR_OperatorFunctionCall`, first, all the operands will be mapped to ExaStencils. Then, based on the name of the operator (stored in the `operator` field of `MIR_OperatorFunctionCall` instance) the equivalent ExaStencils expression is constructed. In figure 10 add (`+`) and multiply (`*`) are shown as examples.

It is worth noting that for function calls (like `sin`) first, an instance of `L2_PlainInternalFunctionReference` given name of the function and its return type is made. Then, using this object an instance of `L2_FunctionCall` would be constructed. If the object matches none of the above, an exception would be raised. Tables 18, and 19 show the mapping used for all the types, from MIR to ExaStencils.

3.5 Expanding the Maxima interface for new types

In this section we discuss the required steps for extending the supported types in the Maxima interface.

3.5.1 Adding a function to the MaximaInterface class for making the Maxima type

To clarify this step listing 21 is used. It shows the implementation of `cos` function. It is important to note that name of the function could be different from what one might expect or from what is mentioned on the existing Maxima documentation.

```
<T> LispObject Cosine(T arg) throws Exception {
        return callFunction1ArgImplementation(arg, "$COS");
}
```

Source Code 21: Implementation of cos function

In practice, the most time-consuming part of extending the Maxima interface is finding the right string. As examples, `cos` function and `+` operators are discussed here. The string `"$COS"` is used for invoking `cos` function.

In case of `cos` function, only a dollar-sign is added to the beginning of the function's name. But in cases like `"+"` operation, a completely different string has to be used as the name of the function.

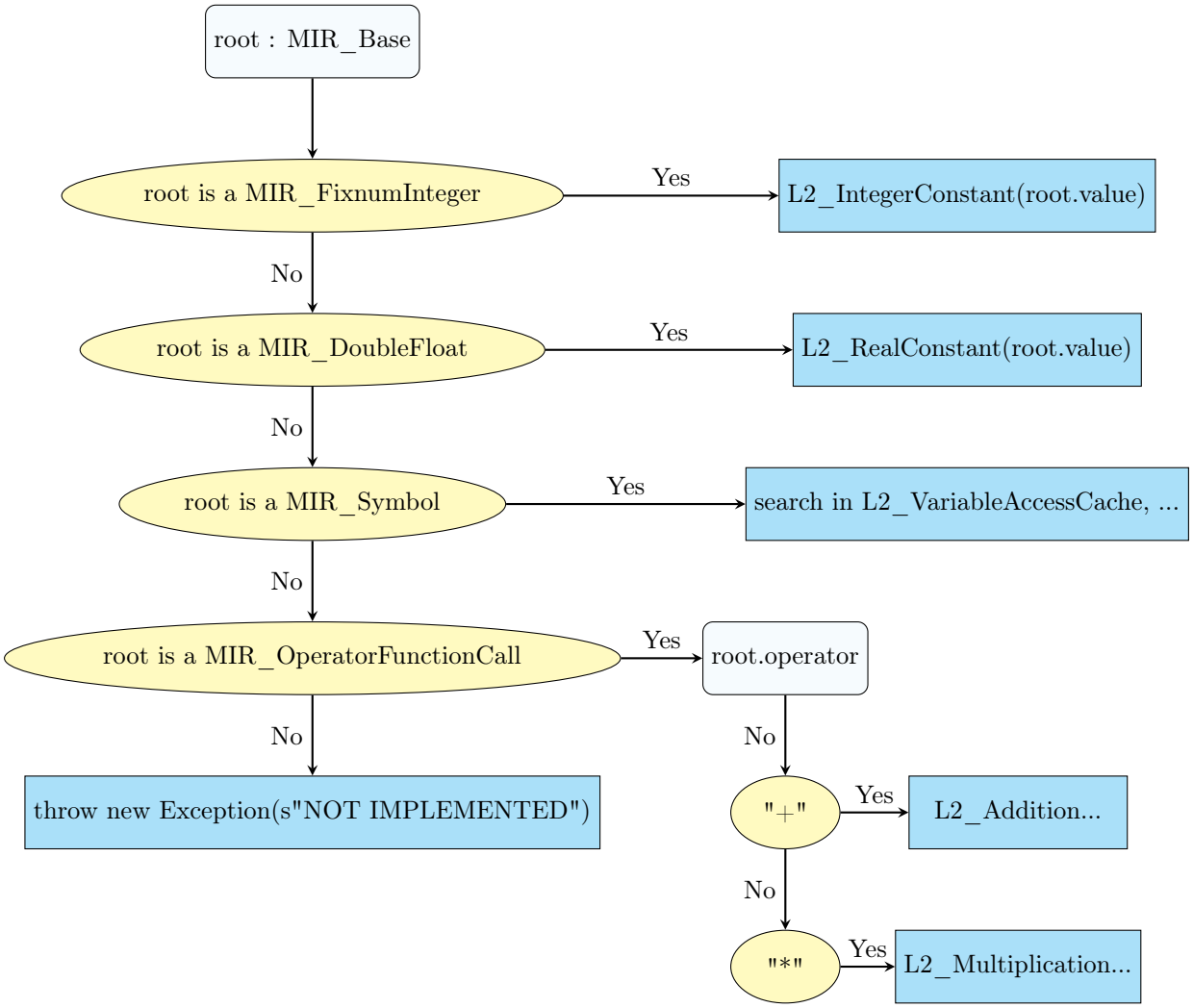

Figure 10: A briefed version of the decision tree used in the implementation of L2_MakeExa's apply method

Implementation of the `ADD` function is shown in listing 22. As it is shown, the string used for addition is `"ADD"`. In the next steps we will see the stark difference between this string and the one we get as we inspect the resulted Maxima object (`"MAXIMA:MPLUS"`).

```
LispObject ADD(scala.collection.immutable.Seq<LispObject> args) throws
Exception {
    LispObject[] lispArgs = new LispObject[args.size()];
    for (int i = 0; i < args.size(); ++i) {
        lispArgs[i] = makeLispEquivalent(args.apply(i));
    }
    return callFunctionNArgsImplementation("ADD", lispArgs);
}
```

Source Code 22: Implementation of ADD function

3.5.2 Adding the mechanism for mapping from ExaStencils to Maxima

In the `MakeMaxima` object, a new case has to be added to the `match` mechanism of the `apply` method. This case will map the new ExaStencils type to a call to the method added to the `MaximaInterface` for the new type. It also has to prepare the arguments to this method. Listing 23 shows such an implementation for mapping `L2_Addition` to Maxima.

```
case operation : L2_Addition
    =>
    val summandsMaxima = L2_MakeMaxima.apply(operation.summands)
    val result = maxima.ADD(summandsMaxima)
    result
```

Source Code 23: Mapping L2_Addition to Maxima

3.5.3 Adding the operation settings to the switch statement

Add the operation settings to the switch statement in the `makeMIR` method for mapping from Maxima to MIR. If `getClassName`, called on the Maxima object returns a string which is not equal to `"Cons"`, add the return string to the switch statement's cases, and the appropriate tratement for that specific type. As an example, the type `BigNum` in listing 24 is shown.

```
case "Bignum": {
        return new MIR_Bignum(( (Bignum) obj).value.longValueExact() );
}
```

Source Code 24: Adding BigNum to the switch statement of the makeMIR method

In such a situation (`"Bignum"`), a `case class` which inherits `MIR_Base` class is added to the `Compiler/src/exastencils/casInterfaces/maximaInterface/MIR_Types.scala` file and instantiated using the Maxima object values. This `case class` would be used later while mapping MIR objects to ExaStencils. When the return value of `getClassName` method called on the Maxima object is `"Cons"`, add the string representing the type to the `knownMaximaSymbols` array, which is a field of the `MIR_Maker` class. This array contains names of all the types of `Symbol` that should be treated as if they are an operator or a function.

One should note that when any of these Symbols are converted to string by calling `toString()`, despite what we see in the Maxima expression (e.g., `(MAXIMA::MPLUS  100 12)`), only one colon exists in the resulted string (e.g., `MAXIMA:MPLUS`).

As it was already discussed, in Lisp, all the expressions are in fact a list (`cons`). The first member is either a Maxima `Symbol` or a `cons`. In case of having a `Symbol` we look in the `knownMaximaSymbols` array, by calling `isMaximaSymbol`. If it returns `true`, then we construct an object of type `MIR_OperatorFunctionCall`. The first element in this `cons`, will determine the type of operation or function to be constructed. The next elements have to be inspected to determine if the `cons` is a wrapped function, operator, or an expression. If the next elements are members of `maximaSimplificationDirectives` array (shown in listing 25), then the `cons` is a wrapped one. As a result, an instance of `MIR_OperatorFunctionCall` with no arguments and its `args_in_higher_node` set to `true` would be returned. Otherwise, the `cons` is an expression. In this case, the next elements are considered to be operands or arguments of the operation or function call.

```
final String[] maximaSimplificationDirectives =
        {"MAXIMA:SIMP", "MAXIMA:RATSIMP",
        "MAXIMA:FACTORED", "MAXIMA:IRREDUCIBLE"};
```

Source Code 25: Maxima's simplification directives

Note that members of `maximaSimplificationDirectives` in the `MIR_Maker` class are the string representations of simplification flags. These flags are used for configuration of the simplification in Maxima. For example, in listing 26, `MAXIMA::SIMP` and `MAXIMA::RATSIMP` are two of such flags.

```
((MAXIMA::MPLUS MAXIMA::SIMP MAXIMA::RATSIMP) 80 MAXIMA::|i1|)
```

Source Code 26: An operation wrapped along with several simplification flags

Similar to the members of `knownMaximaSymbols`, when the string representation of these flags are obtained by calling the `toString()` method, only one colon would be found in the result. Therefore, the string to be added for them are `"MAXIMA:SIMP"` and `"MAXIMA:RATSIMP"`. To be clear, it is expected that in case of encountering a new flag that has to be ignored during the mapping, it should be added to the `maximaSimplificationDirectives`. Next, the name of operation or function is added to the cases of the switch statement, to set the `operatorfunction` variable. This string would be used in the mapping MIR object to ExaStencils. Therefore, it has to be the same in both the `MIR_Maker` class and `MakeExa` objects. For example, as it is shown in listing 27, for the `"MAXIMA:MPLUS"` the string `"+"` is used. Note that this string could be any desired one (here, "+", "summation", or even "Covfefe"), as long as the same string is used in mapping from MIR to ExaStencils. `operatorFunction` is later used for constructing an instance of `MIR_OperatorFunctionCall` class.

```
case "MAXIMA:MPLUS":
    operatorFunction = "+";  break;
```

Source Code 27: Setting the field operatorFunction for the case "MAXIMA:MPLUS"

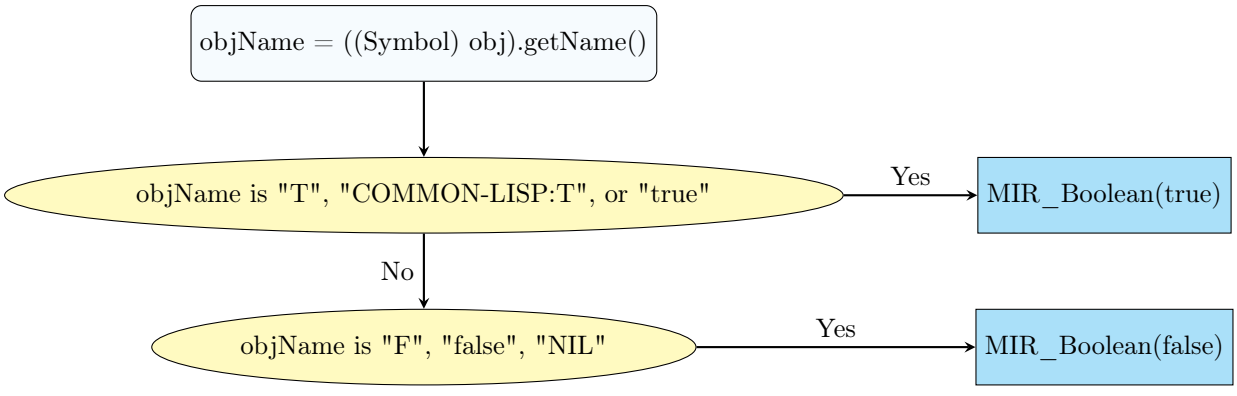

Figure 11: The decision tree used in the "Symbol" case and its subsequent switch statement

Add the implementation under `case "Symbol"` for `Symbols`. In this case, name of the Maxima object is the decisive factor. See figure 11 for an example of the decision tree used in the implementation.

3.5.4 Adding the mechanism for mapping from MIR to ExaStencils

When a new type is added to the supported types of the interface, the appropriate mapping from MIR to ExaStencils has to be implemented. This step is fairly straight forward since the MIR classed are `case class` and `match` could be utilized. For simple data types, building an instance of their equivalent in ExaStencils is done with no further steps. As an example, for `MIR_Bignum` in L2 (figure 10), an instance of `L2_IntegerConstant` is built.

A similar implementation could be added for a new type. If the new type is an operation or function, the implementation has to be added under the `operatorCall.operator match {...}`. Figure 10 shows the case for addition operator.

3.6 Sample results

To show the functionality of the simplification pipeline, two expression were chosen, $(sin(i0)+cos(i0))^{2.0}$ and $(i1+2.0)*(i1+2.0)*i1$. These expressions were simplified and the results of simplification steps are presented and discussed when needed. Note that the goal of this section is the illustration of all the steps in the simplification of an expression using the Maxima interface and their intermediate results. Therefore, the discussion on the performance and quality of the simplification is presented in the chapter 4.

Trigonometric expression

Figure 12 shows the input ExaStencils expression representing $(sin(i0)+cos(i0))^{2.0}$.

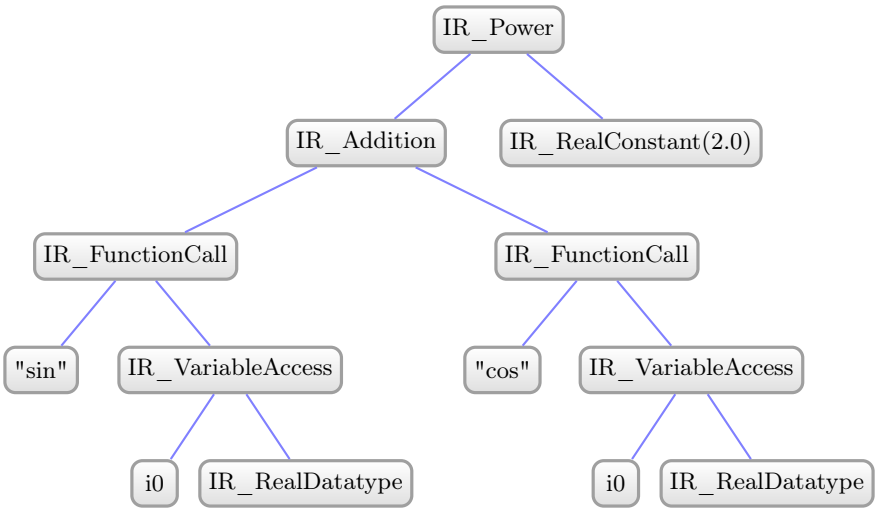

Figure 12: ExaStencils expression used as input to the ExaSimplifierMaxima (trigonometric expression)

This expression was mapped to Maxima as it is shown in figure 13. The variable name which was used in the Maxima was `AUTO_GEN_NAME_IR_VA__i0___________________IR_RealDatatype_` which was replaced with `i0` for simplicity.

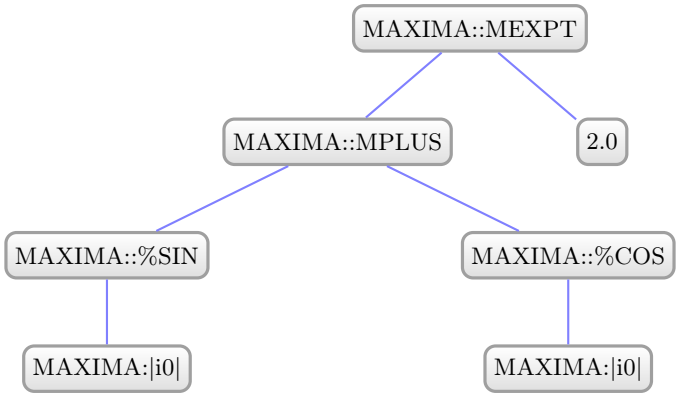

Figure 13: The equivalent of ExaStencils expression in Maxima (trigonometric expression)

Then, the simplification were performed by calling the `DO_SIMPLIFICATION` method of an instance of `class MaximaInterface` on the Maxima expression. The result of this call is shown in figure 14.

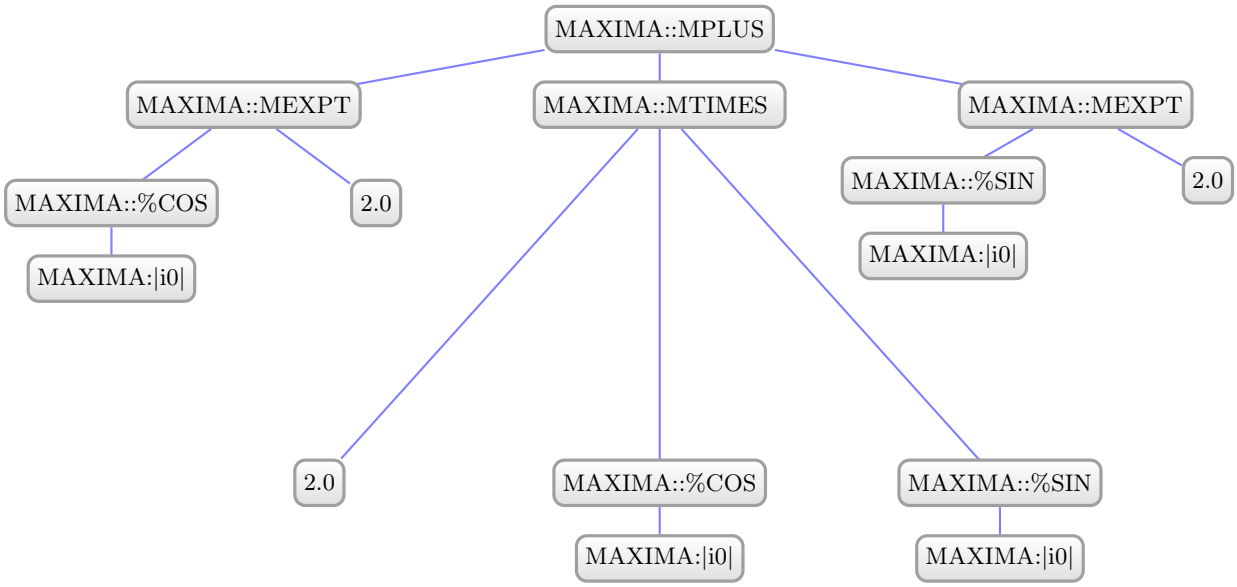

Figure 14: The simplified expression in Maxima (trigonometric expression)

After simplification, the resulted Maxima expression was mapped to ExaStencils. Figure 15 shows the result of mapping from Maxima to ExaStencils.

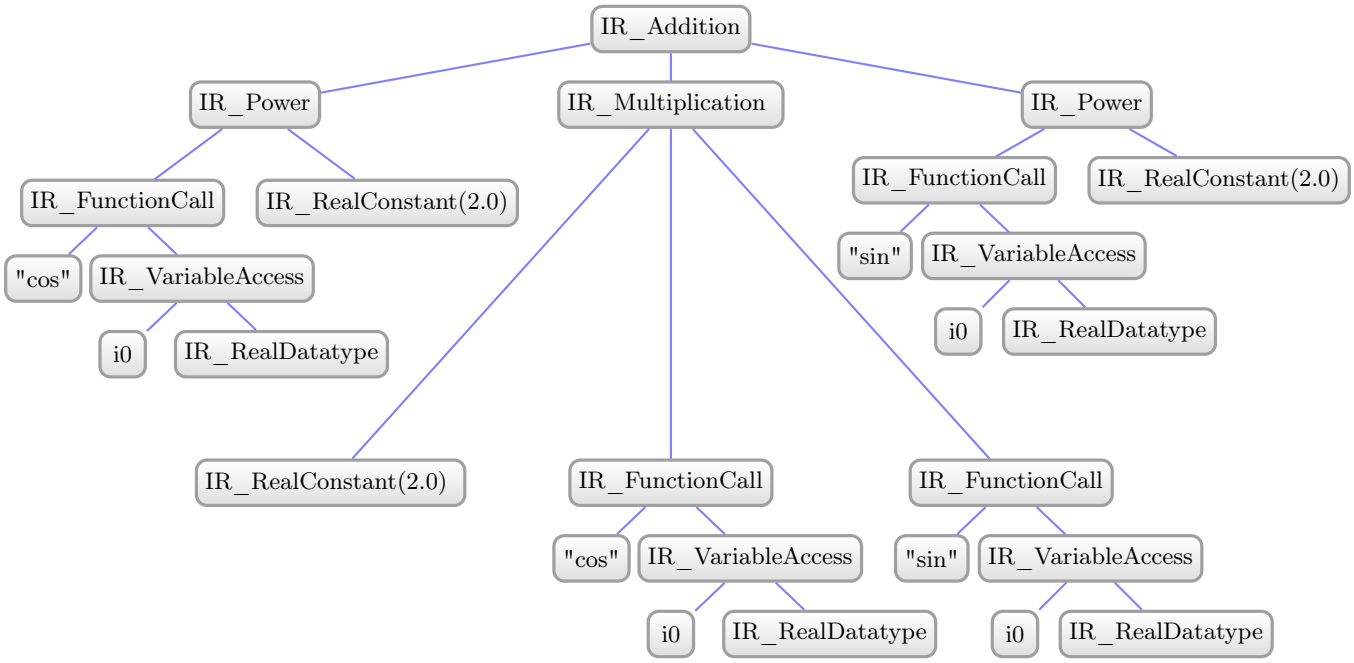

Figure 15: The result of simplification were mapped backed to ExaStencils (trigonometric expression)

Polynomial expression

Next example simplifies expression , $(i1 + 2.0) * (i1 + 2.0) * i1$. Figure 16 shows the ExaStencils representation of this polynomial expression.

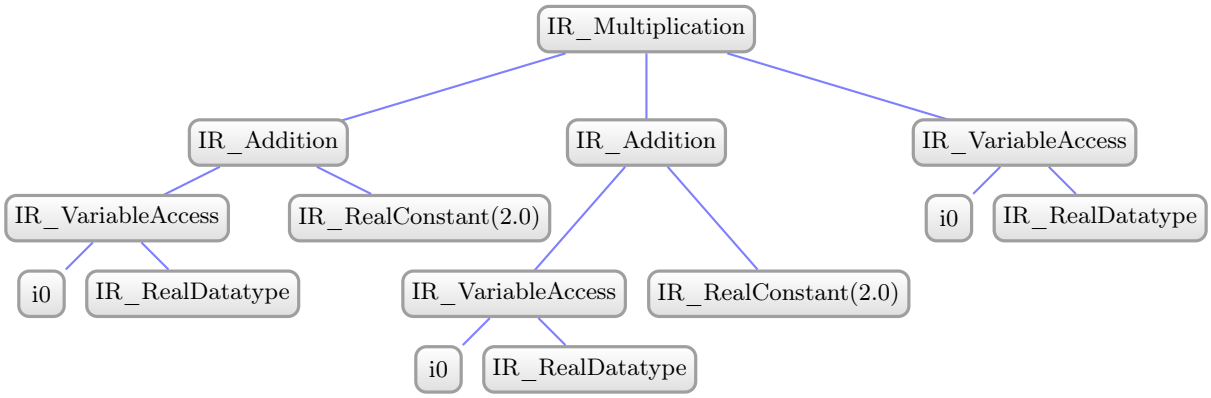

Figure 16: ExaStencils expression used as input to the ExaSimplifier (Polynomial expression)

This expression was mapped to Maxima (figure 17). Similar to the previous example, the variable name which was used in the Maxima (`AUTO_GEN_NAME_IR_VA__i1__________________IR_RealDatatype_`) was replaced with `i1` for simplicity.

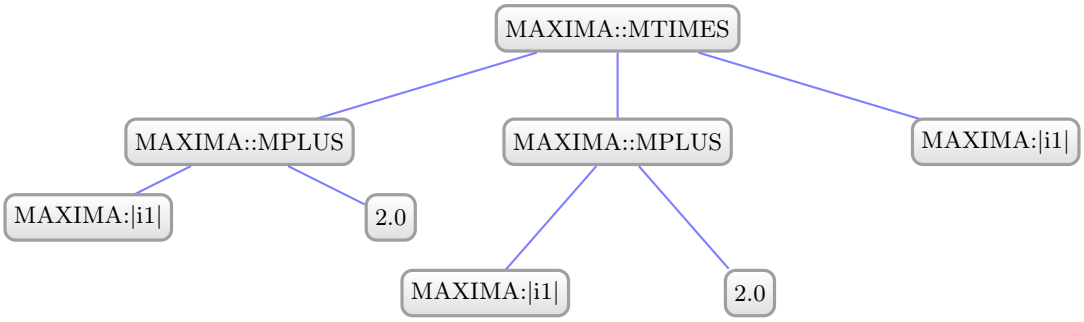

Figure 17: The equivalent of ExaStencils expression in Maxima (Polynomial expression)

Then, the simplification were performed by calling the `DO_SIMPLIFICATION` method of an instance of `class MaximaInterface` on the Maxima expression. The result of this call is shown in figure 18.

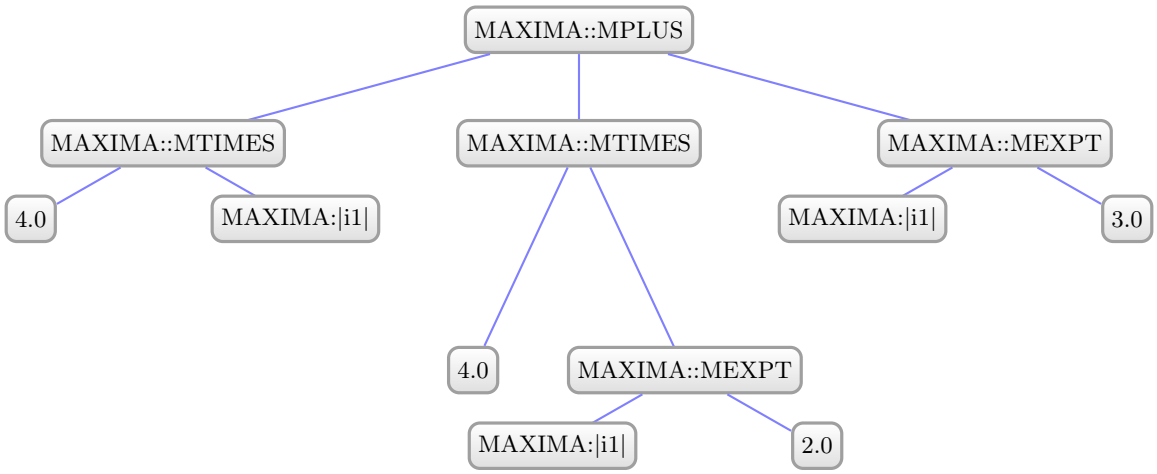

Figure 18: The simplified expression in Maxima (Polynomial expression)

After the simplification, the resulted Maxima expression was mapped to ExaStencils. Figure 19 shows the result of mapping from Maxima to ExaStencils.

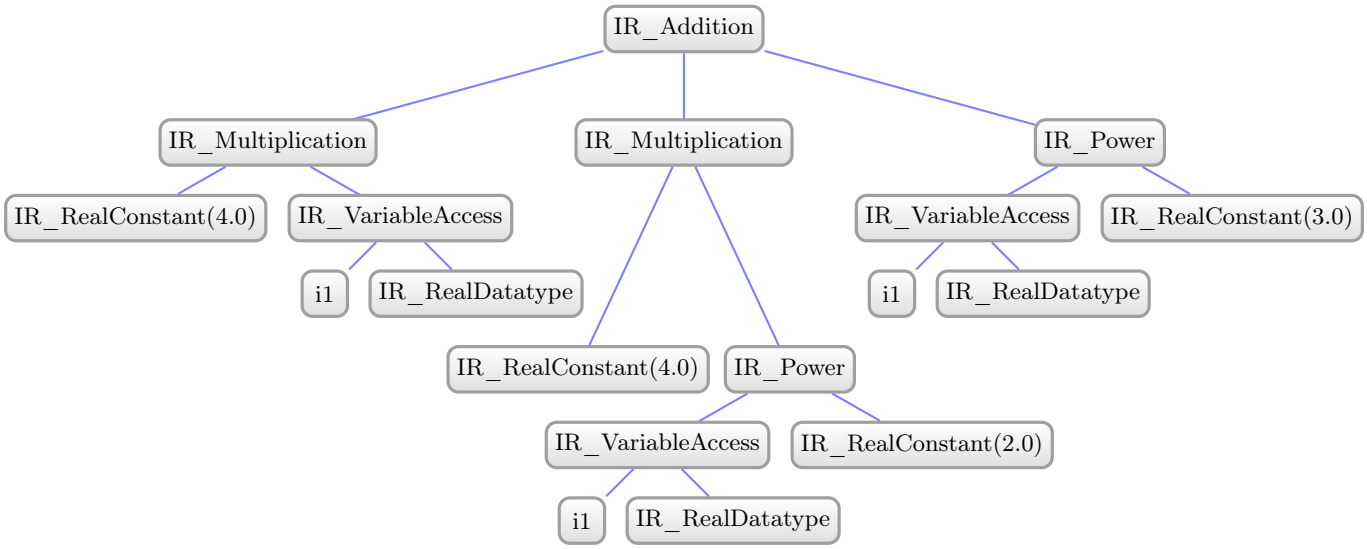

Figure 19: The result of simplification were mapped backed to ExaStencils (Polynomial expression)

This concludes all the steps in simplification using the Maxima interface. The quality and performance of the Maxima interface along with a comparison between the two interfaces are discussed in detail in chapter 4.

4 Comparison between the SymPy interface and the Maxima interface

To compare the performance of each interface we chose several expressions; the general solution of a fourth-degree polynomial, expanded form of the symbolic third-degree polynomial, a rational algebraic fraction, and a trigonometric expression.

Since the goal of CAS interfaces is the simplification of expression resulted from the discretization of equations for solving differential equations using the multigrid method, we chose a few of the most common forms of expression encountered in the discretization: polynomials, rational algebraic fractions, and trigonometric expression. Note that these examples are by no means complete representatives of the types of expression that might be encountered, but they are only a few examples to show the strong points of each interface and their shortcomings. Also, we like to emphasize that our focus is on the interfaces and not the CAS package itself. Therefore, the generalization of the finding as a metric of the performance of each CAS package is highly discouraged.

4.1 Methodology

Both interfaces were used to simplify the expressions. The results of the simplification of each expression by both interfaces were compared in terms of time requirements and the quality of simplification (relative, and only for that expression).

4.1.1 Timing

To make the timing more accurate each simplification was repeated 1000 times and the average of its running time was used in the comparisons. All of the simplifications were performed without any parallelization for both interfaces. We discuss the initial time required for each interface to be set up as a separate entity since it only has to be paid once. Including this time in the total measurement would cause dependency of the performance on the number of expressions to be simplified. Therefore, the timing reported for the simplification of expression could be considered as a metric relative to the throughput of each interface and naturally lower time is more desirable.

```
CPU:
    Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz
    clock fixed at: 3000MHz
    L1 Cache : 32KiB
    L2 Cache : 256KiB
    L3 Cache : 3MiB
Memory:
    size: 8GiB
    two banks of 4GiB:
    SODIMM DDR3 Synchronous 1600 MHz
    width: 64 bits
    clock: 1600MHz
```

Source Code 28: Specifications of the machine used in simplifications

4.1.2 Quality of simplification

For the quality of the simplification, we only counted the number of each operation in the input expression and the resulted expressions and compared them. Lower number of operations means lower cost. Since the cost of each operation may differ based on the architecture of the machine, we avoid comparing the mix of different operations with each other.

4.1.3 Simplifier functions

The simplification in the SymPy interface is achieved by invoking the function `simplifyDoIt` of the `SymPyInterface`. This function in turn calls the `simplify` function from the SymPy package, passing the expression. The resulted SymPy expression's `doit` method is then called using the argument `deep = true`. Note that `doit` causes recursively evaluation of all objects to be performed.

To simplify an expression using the Maxima interface the method `DO_SIMPLIFICATION` from `MaximaInterface` is called on the expression. In the case of Maxima since there is not a single general function, similar to the `simplify` function in SymPy that performs all the possible simplifications, we used all of the simplification function, *accessible* via the interface.

`RATSIMP_simplify(FACTOR_simplify(RADCAN_simplify(obj)));` is the implementation of these calls (`DO_SIMPLIFICATION` method) which uses `RATSIMP`, `FACTOR`, and `RADCAN` for simplification of the expression `obj`. Also, `SIMPSUM`, `SIMP`, and `RATSIMPEXPONS` flags were set to `true` [10]. It is important to note that since the selection of these simplifying methods is out of the scope of this work, we did not alter the order of calls or number of them for different expressions.

In the next sections, we take a look at the results of the simplification of each expression, using CAS interfaces, and compare them based on the factors discussed earlier.

4.2 Results of timing (performance)

We timed the simplification of each expression for 1000 times and calculated the average time for the simplification of one expression. Only in the case of Maxima, a considerable time (about 27 seconds) was spent on the start-up of the interface (loading ABCL and Maxima package). This time was not considered when measuring the performance, as we already discussed in section 4.1.1. As it is shown in table 1, the Maxima interface is considerably faster than the SymPy interface for all of the expressions. However, it should be taken into account that the reported time is the time that takes for each interface to map the expression to their respective CAS, simplify it, and map the result to the ExaStencils. Therefore, the reported performances are not a metric for the performance of the CAS alone, but for the combination of interface and CAS.

expression	Maxima time (ms)	SymPy time (ms)
The general solution of a fourth-degree polynomial	42.420	1841.99
Expanded form of the symbolic third-degree polynomial	10.012	4913.35
A rational algebraic fraction	8.932	1985.66
trigonometric expression	9.081	4383.58

Table 1: The average time for one expression to be simplified by each interface

4.3 Quality of the simplification

In this section, we investigate the quality of simplification performed by each interface and compare the results. The metric used here is already discussed in section 4.1.2. To avoid the complications related to the internal representation of each expression, we only used the pretty-printing of the expressions.

4.3.1 The general solution of a fourth-degree polynomial

The first expression is $12.343 * (x - 1020.0) * (x - 2.33) * (x + 26.0345) * (x + 1001.0)$. This expression is chosen to test interfaces with respect to the simplification of polynomials.

```
5.0E-9 * (  2468600000  * x^4 + 11613528700 * x^3 + -2521751539871711 * x^2
         + -59744109362972191 * x + 152893892090159220 )
```

Source Code 29: The result of simplification using the Maxima interface (The general solution of a fourth-degree polynomial)

```
12.343 * (1001.0 + x) * (26.0345 + x) * (-1020.0 + x) * (-2.33 + x)
```

Source Code 30: Result of simplification using the SymPy interface (The general solution of a fourth-degree polynomial)

In the next step, the same expression was passed to the SymPy interface. The result of the simplification using the SymPy interface is shown in listing 30. This expression is exactly the same as the input expression.

operation	**Input expression**	**The result of the Maxima interface**	**The SymPy interface**
*	4	5	4
+ and -	4	4	4
power	0	3	0

Table 2: Number of operations in each expression; input, the result of the Maxima interface, and the result of the SymPy interface (The general solution of a fourth-degree polynomial)

By comparing the number of operations in each expression as shown in table 2, it is evident that the Maxima interface only increased the cost of computation of this expression. The number of multiplications in the Maxima interface's result is 5, compared to 4 for the input expression and the result of SymPy simplification. Besides, we had no power in the input expression, but the Maxima interface has introduced it to its result (3 powers). Here we report the number of additions and subtraction as one entity. Note that only the SymPy interface kept the form of the expression, as it was already the general solution of the polynomial, $12.343*(x-1020.)*(x-2.33)*(x+26.0345)*(x+1001.)$. Therefore, the result of simplifications by the SymPy interface was exactly as the input expression. Since only the Maxima interface altered the form of expression, we investigated another polynomial to have a better comparison between two CAS interfaces in terms of the quality of simplification.

4.3.2 The expanded form of the symbolic third-degree polynomial

To further investigate the ability of each interface, we used the expanded form of $(a+b*x)(c+d*x)(e+f*x)$ polynomial (shown in listing 31) for testing.

```
a*c*e  +  a*c*f * x  +  a*d*e * x  +  a*d*f * x^2  +  b*c*e * x  +  b*c*f * x^2
+ b*d*e * x^2  +  b*d*f * x^3
```

Source Code 31: Input expression (expanded form of the symbolic third-degree polynomial)

The result of the simplifications performed by the Maxima interface is shown in listing 32.

```
b*d*f * x^3  +  (((a*d + b*c)*f) + b*d*e) * x^2  +  (a*c*f + (a*d + b*c)*e) * x
+ a*c*e
```

Source Code 32: Maxima's result (expanded form of the symbolic third-degree polynomial)

The same expression was passed to the SymPy interface for simplification. The result is shown in listing 33.

```
x * ((b*(c*e + (x * (c*f + (d*(e + f*x))))))  +  a*c*f + a*d*e + a*d*f * x )
+ a*c*e
```

Source Code 33: SymPy's result (expanded form of the symbolic third-degree polynomial)

operation	**Input expression**	**The result of the Maxima interface**	**The result of the SymPy interface**
*	23	17	16
+ and -	7	7	7
power	4	2	0

Table 3: Number of operations in each expression; input, the result of Maxima, and the result of SymPy (expanded form of the symbolic third-degree polynomial)

This time, both Maxima and SymPy interfaces altered the expression. While both CAS interfaces reduced the cost of computation, the SymPy interface had a much better performance, as is shown in table 3. Maxima interface reduced the number of multiplications from 23 to 17 and the SymPy interface resulted in only 16 multiplications. In addition, the result of simplifications in the SymPy interface contained 7 additions/subtractions, which is the same as the result of the Maxima interface and the input expression. The number of power calculations was lowered by both interfaces, but the SymPy interface had a better result with no power compared to 2 in the Maxima interface.

4.3.3 A rational algebraic fraction

```
( sqrt(x - 1) * x^4 + -1*sqrt(x - 1) * x^3 + 15*sqrt(x - 1) * x - 15*sqrt(x - 1) )
 / (x - 1)
```

Source Code 34: Input expression (rational algebraic fraction)

The next case is a rational algebraic fraction which is shown in listing 34. This expression could be simplified to $((x^3+15)(x-1)^{(1/2)})$. Maxima interface simplified this expression to $(-1+x)^{0.5}*(15+x^3)$. SymPy interface did not perform as well as the Maxima interface and resulted in $(-15+x^4+15*x-x^3)*(-1+x)^{-0.5}$. In this case, SymPy interface resulted in a more complex expression. As for the cost of each expression, both methods reduced it, but the Maxima interface performed much better. Maxima interface reduced the number of multiplications from 6 to 1, the same as the SymPy interface. But at the same time it resulted in a lower number of additions/subtractions and powers; 2 powers in the Maxima interface vs 3 in the SymPy interface and 2 additions in the Maxima interface's result vs 4 in the SymPy interface's.

operation	Input expression	The result of the Maxima interface	The result of the SymPy interface
*	6	1	1
+ and -	8	2	4
power	2	2	3
sqrt	4	0	0

Table 4: Number of operations in each expression; input, the result of Maxima, and the result of SymPy (rational algebraic fraction)

4.3.4 A trigonometric expression

Finally, we compared the performance of CAS interfaces in the simplification of trigonometric expressions. Here, we chose the expression of listing 35 as input.

```
-9 * x^2 * cot^2(x) + 9 * x^2 * csc^2(x) - 30 * x * cot^2(x) - 25 * cot^2(x)
+ 30 * x * csc^2(x) + 25 * csc^2(x)
```

Source Code 35: Input expression (trigonometric expression)

The input expression could be simplified to $(csc^2(x) - cot^2(x)) * (3 * x + 5)^2$. In this case, the Maxima interface simplified the expression as is shown in listing 36 while the SymPy interface did better, and resulted in the expression shown in listing 37.

```
(-25 - 30 * x - 9 * x^2) * cot^2(x) + ( 25 + 30 * x + 9 * x^2) * csc^2(x)
```

Source Code 36: Maxima's result (trigonometric expression)

```
25 * (1 + 0.6 * x)^2 * ((1/sin(x))^2 - (1/tan(x))^2)
```

Source Code 37: SymPy's result (trigonometric expression)

operation	Input expression	The result of the Maxima interface	The result of the SymPy interface
*	10	6	3
+ and -	5	5	2
power	8	4	3
trigonometric function	6	2	2
/	0	0	2

Table 5: Number of operations in each expression; input, the result of the Maxima interface, and the result of the SymPy interface (trigonometric expression)

Comparing the costs of each result (table 5) shows that although both had a reduction in cost, the SymPy interface clearly reduced the counts of more operation types (multiplications, additions and subtractions, and powers) compared to the Maxima interface (only division). It succeeded to reduce the number of multiplication from 10 to 3, compared to 6 in the Maxima interface. In addition, it reduced the number of additions/subtractions while the Maxima interface did not. Similar to the Maxima in-

terface, the SymPy interface resulted in an expression containing only 2 trigonometric function calls, compared to 6 in the input expression. However, the Maxima interface had a better result in the number of divisions which was zero where SymPy introduced two divisions to the expressions. In this example, it should be noted that the Maxima package uses the function `trigsimp` in simplification of trigonometric expressions, which was not accessible trough ABCL, and therefore, it is not used in the Maxima interface's simplifications.

As the investigated cases of this chapter suggest, it can be deduced that the Maxima interface outperforms the SymPy interface in simplification-time, especially as the number of expressions to be simplified increases. But for the quality of simplification, there is no absolute winner. In some cases, the results of the Maxima interface are better compared to the SymPy interface and in others the opposite hold. Once more we emphasize the fact that this comparison was between the interfaces and not the CAS package. Also, note that the examples in this chapter were solely for illustration of the fact that there is no guarantee that the result of simplification of an expression using these interfaces would be computationally cheaper, or equivalent to the input expression. As a result of the lack of this insurance, either a much more in-depth study of both CASs is required to enhance the decision making on "the interface to use for a given expression", or a mechanism should be added for estimation of the computational cost of the expressions. This mechanism could be used for comparing the result of each interface, and the input expression. Based on such a comparison, the best expression could be used as a result of simplification. By using the information on the throughput of each operation on the target hardware, the expression simplification could be fine-tuned for each hardware.

5 Conclusion

In this project, two interfaces were developed for the utilization of CAS packages in ExaStencils. SymPy (originally developed in Python) and Maxima (developed in Lisp) were the CAS chosen for this purpose. Both of them are free, open-source, and maintained by the community. Using these packages could ensure that some of the calculations are performed once and only at the time of generating the stencils code, rather than being repeated each time the resulted code would be executed. This approach would drastically ease the development of highly optimized code, while there would not be a need for maintaining a CAS package by the ExaStencils developers.

After developing and testing both interfaces it became clear that Maxima has a better performance in terms of speed of simplifications. However, no clear advantage was observed for any of the interfaces, in terms of the quality of simplification. In some cases, SymPy resulted in a better-simplified expression, in terms of the cost of calculating the resulted expression. And in other cases, Maxima had better results. In addition, there were cases that the resulted expression was more expensive to calculate compared to the input.

As it was already stated in the previous chapters, the goal of this work was to provide the interfaces for the CAS packages and not to find the best form of an expression to be used in the final generated code. Considering the fact that there is no guarantee that the resulting expression (using these interfaces) would be computationally cheaper than the input expression, the need for a decision-making mechanism arises. A mechanism that ensures only the cheapest available form of expression would be used in the final generated code.

5.1 Future work

As the extension of this work, we recommend performing a cost estimation on the result of each interface and the input expression, before making a decision on whether or not to use the simplified expression. We suggest the development of such a mechanism for estimation of the computational cost of the expressions. This mechanism could be used for comparing the result of each interface, and the input expression. Based on such a comparison, the best expression could be used as the result of simplification. Furthermore, by using the information on the throughput of each operation on the target hardware, the expression simplification could be fine-tuned for each hardware.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006. ISBN 0321486811. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321486811.

[2] Daniel Barlow and contributors. *ASDF: Another System Definition Facility*, 2019. URL https://common-lisp.net/project/asdf/asdf.html.

[3] Mark Evenson, Erik Hülsmann, Rudolf Schlatte, Alessio Stalla, and Ville Voutilainen. *Armed Bear Common Lisp User Manual*, April 2020.

[4] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman and Hall / CRC computational science series. CRC Press, 2011. ISBN 978-1-439-81192-4.

[5] Stefan Kronawitter. *Automatic Performance Optimization of Stencil Codes*. PhD thesis, Universität Passau, 2020.

[6] Stefan Kronawitter and Christian Lengauer. Optimizations applied by the exastencils code generator. *Faculty Inform. Math., Univ. Passau, Passau, Germany, Tech. Rep. MIP-1502*, pages 1–10, 2015.

[7] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

[8] Joel Moses. Macsyma: A personal history. *J. Symb. Comput.*, 47:123–130, 2012.

[9] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. Exaslang: A domain-specific language for highly scalable multigrid solvers. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 42–51. IEEE, 2014.

[10] Maxima Development Team. *Maxima Documentation*. Macsyma group at Project MAC and volunteer contributors. URL http://maxima.sourceforge.net/docs/manual/maxima.pdf.

[11] SymPy Development Team. *SymPy Documentation*. SymPy Development Team, 2019. URL https://docs.sympy.org.

6 Appendix

6.1 Tables related to the SymPy interface

Exastencils	**SymPyInterface**	**SymPy**
L2_IntegerConstant	integer	Integer
L2_BooleanConstant	boolean	(Python-built-in) bool
L2_RealConstant	float	Float
L2_StringConstant	string	(Python-built-in) str
L2_StringLiteral	string	(Python-built-in) str
L2_Addition	add	Add
L2_Multiplication	mul	Mul
L2_Subtraction	subtract	Add(l, Mul(Integer(-1), r))
L2_Division	divide	Mul(l, Pow(r, -1))
L2_Modulo	mod	Mod(l, r)
L2_Power	pow	Pow
L2_ElementwiseAddition	matAddElementwise	MatAdd
L2_ElementwiseSubtraction	matSubtractElementwise	MatAdd(l, MatMul(Integer(-1), r))
L2_ElementwiseMultiplication	matMulElementwise	hadamard_product(l, r)
L2_ElementwiseDivision	matDivideElementwise	MatMul(l, hadamard_power(r, Integer(-1)))
L2_ElementwiseModulo	matModElementwise	hadamard_product(l, hadamard_power(r, Integer(-1)))
L2_ElementwisePower	matPowElementwise	hadamard_power(l, r)
L2_EqEq	eqEq	__eq__
L2_Neq	nEq	__neq__
L2_Lower	lower	__lt__
L2_Greater	greater	__gt__
L2_LowerEqual	lowerEqual	__le__
L2_GreaterEqual	greaterEqual	__ge__
L2_GreaterEqual	greaterEqual	__ge__
L2_AndAnd	andAnd	And
L2_OrOr	orOr	Or
L2_PlainVariableAccess	symbol(name)	Symbol
L2_LeveledVariableAccess	symbol(name)	Symbol
L2_Negative(v)	Negative	__neg__
L2_Negation(v)	Not	(Python-built-in) not
L2_Equation(lhs, rhs)	equality	Equality
L2_LowerEqual(lhs, rhs)	lessThan	LessThan
L2_Lower(lhs, rhs)	strictLessThan	StrictLessThan
L2_GreaterEqual(lhs, rhs)	greaterThan	GreaterThan
L2_Greater(lhs, rhs)	strictGreaterThan	StrictGreaterThan
L2_NullExpression	-	(Python-built-in) None
L2_FunctionCall	See table 7	

Table 6: L2 expression mapping, r: right-hand operand, l: left-hand operand.

functionCall.function.name	SymPyInterface	SymPy
"sin"	Sine	sin
"cos"	Cosine	cos
"tan"	Tangent	tan
"cot"	Cotangent	cot
"sec"	Secant	sec
"csc"	Cosecant	csc
"asin"	Arcsin	asin
"acos"	Arccos	acos
"atan"	Arctan	atan
"acot"	Arccot	acot
"asec"	Arcsec	asec
"acsc"	Arccsc	acsc
"sinh"	Sinh	sinh
"cosh"	Cosh	cosh
"tanh"	Tanh	tanh
"coth"	Coth	coth
"sech"	Sech	sech
"csch"	Csch	csch
"asinh"	Arcsinh	asinh
"acosh"	Arccosh	acosh
"atanh"	Arctanh	atanh
"acoth"	Arccoth	acoth
"asech"	Arcsech	asech
"acsch"	Arccsch	acsch
"abs"	Abs	Abs
"fabs"	Abs	Abs
"pow"	pow	Pow
"ldexp"	mul(L2_MakeSymPy(args.head), pow(2, L2_MakeSymPy(args(1))))	Mul
"exp2"	pow(2, L2_MakeSymPy(args.head))	Pow
"exp10"	pow(10, L2_MakeSymPy(args.head))	Pow
"sqrt"	Sqrt	sqrt
"log10"	Log(L2_MakeSymPy(args.head), 10)	log(v, 10)
"log" with one argument	LogNat	log(v)
"log" with two argument	Log(L2_MakeSymPy(args.head), L2_MakeSymPy(args(1)))	log(v, b)
non-of-above	Exception	-

Table 7: L2_FunctionCall mapping

Exastencils	**SymPyInterface**	**SymPy**
L3_IntegerConstant	integer	Integer
L3_BooleanConstant	boolean	(Python-built-in) bool
L3_RealConstant	float	Float
L3_StringConstant	string	(Python-built-in) str
L3_StringLiteral	string	(Python-built-in) str
L3_Addition	add	Add
L3_Multiplication	mul	Mul
L3_Subtraction	subtract	Add(l, Mul(Integer(-1), r))
L3_Division	divide	Mul(l, Pow(r, -1))
L3_Modulo	mod	Mod(l, r)
L3_Power	pow	Pow
L3_ElementwiseAddition	matAddElementwise	MatAdd
L3_ElementwiseSubtraction	matSubtractElementwise	MatAdd(l, MatMul(Integer(-1), r))
L3_ElementwiseMultiplication	matMulElementwise	hadamard_product(l, r)
L3_ElementwiseDivision	matDivideElementwise	MatMul(l, hadamard_power(r, Integer(-1)))
L3_ElementwiseModulo	matModElementwise	hadamard_product(l, hadamard_power(r, Integer(-1)))
L3_ElementwisePower	matPowElementwise	hadamard_power(l, r)
L3_EqEq	eqEq	__eq__
L3_Neq	nEq	__neq__
L3_Lower	lower	__lt__
L3_Greater	greater	__gt__
L3_LowerEqual	lowerEqual	__le__
L3_GreaterEqual	greaterEqual	__ge__
L3_GreaterEqual	greaterEqual	__ge__
L3_AndAnd	andAnd	And
L3_OrOr	orOr	Or
L3_PlainVariableAccess	symbol(name)	Symbol
L3_LeveledVariableAccess	symbol(name)	Symbol
L3_Negative(v)	Negative	__neg__
L3_Negation(v)	Not	(Python-built-in) not
L3_Equation(lhs, rhs)	equality	Equality
L3_LowerEqual(lhs, rhs)	lessThan	LessThan
L3_Lower(lhs, rhs)	strictLessThan	StrictLessThan
L3_GreaterEqual(lhs, rhs)	greaterThan	GreaterThan
L3_Greater(lhs, rhs)	strictGreaterThan	StrictGreaterThan
L3_NullExpression	-	(Python-built-in) None
L3_FunctionCall	See table 9	-

Table 8: L3 expression mapping, r: right-hand operand, l: left-hand operand

functionCall.function.name	**SymPyInterface**	**SymPy**
"sin"	Sine	sin
"cos"	Cosine	cos
"tan"	Tangent	tan
"cot"	Cotangent	cot
"sec"	Secant	sec
"csc"	Cosecant	csc
"asin"	Arcsin	asin
"acos"	Arccos	acos
"atan"	Arctan	atan
"acot"	Arccot	acot
"asec"	Arcsec	asec
"acsc"	Arccsc	acsc
"sinh"	Sinh	sinh
"cosh"	Cosh	cosh
"tanh"	Tanh	tanh
"coth"	Coth	coth
"sech"	Sech	sech
"csch"	Csch	csch
"asinh"	Arcsinh	asinh
"acosh"	Arccosh	acosh
"atanh"	Arctanh	atanh
"acoth"	Arccoth	acoth
"asech"	Arcsech	asech
"acsch"	Arccsch	acsch
"abs"	Abs	Abs
"fabs"	Abs	Abs
"pow"	pow	Pow
"ldexp"	mul(L3_MakeSymPy(args.head), pow(2, L3_MakeSymPy(args(1))))	Mul
"exp2"	pow(2, L3_MakeSymPy(args.head))	Pow
"exp10"	pow(10, L3_MakeSymPy(args.head))	Pow
"sqrt"	Sqrt	sqrt
"log10"	Log(L3_MakeSymPy(args.head), 10)	log(v, 10)
"log" with one argument	LogNat	log(v)
"log" with two argument	Log(L3_MakeSymPy(args.head), L3_MakeSymPy(args(1)))	log(v, b)
non-of-above	Exception	-

Table 9: L3_FunctionCall mapping

Exastencils	**SymPyInterface**	**SymPy**
IR_IntegerConstant	integer	Integer
IR_BooleanConstant	boolean	(Python-built-in) bool
IR_RealConstant	float	Float
IR_DoubleConstant	float	Float
IR_RealConstant	float	Float
IR_StringConstant	string	(Python-built-in) str
IR_StringLiteral	string	(Python-built-in) str
IR_Addition	add	Add
IR_Multiplication	mul	Mul
IR_Subtraction	subtract	Add(l, Mul(Integer(-1), r))
IR_Division	divide	Mul(l, Pow(r, -1))
IR_Modulo	mod	Mod(l, r)
IR_Power	pow	Pow
IR_ElementwiseAddition	matAddElementwise	MatAdd
IR_ElementwiseSubtraction	matSubtractElementwise	MatAdd(l, MatMul(Integer(-1), r))
IR_ElementwiseMultiplication	matMulElementwise	hadamard_product(l, r)
IR_ElementwiseDivision	matDivideElementwise	MatMul(l, hadamard_power(r, Integer(-1)))
IR_ElementwiseModulo	matModElementwise	hadamard_product(l, hadamard_power(r, Integer(-1)))
IR_ElementwisePower	matPowElementwise	hadamard_power(l, r)
IR_EqEq	eqEq	__eq__
IR_Neq	nEq	__neq__
IR_Lower	lower	__lt__
IR_Greater	greater	__gt__
IR_LowerEqual	lowerEqual	__le__
IR_GreaterEqual	greaterEqual	__ge__
IR_GreaterEqual	greaterEqual	__ge__
IR_AndAnd	andAnd	And
IR_OrOr	orOr	Or
IR_BitwiseAnd	bitwiseAnd	__and__
IR_LeftShift	leftShift	__lshift__
IR_PlainVariableAccess	symbol(name)	Symbol
IR_LeveledVariableAccess	symbol(name)	Symbol
IR_Negative(v)	Negative	__neg__
IR_Negation(v)	Not	(Python-built-in) not
IR_Equation(lhs, rhs)	equality	Equality
IR_LowerEqual(lhs, rhs)	lessThan	LessThan
IR_Lower(lhs, rhs)	strictLessThan	StrictLessThan
IR_GreaterEqual(lhs, rhs)	greaterThan	GreaterThan
IR_Greater(lhs, rhs)	strictGreaterThan	StrictGreaterThan
IR_NullExpression	-	(Python-built-in) None
IR_AddressOf(v)	AddressOf	id
IR_ToInt(expr)	toInt	int
IR_MatrixExpression	matrix	Matrix
String	string	(Python-built-in) str
java.lang.String	string	(Python-built-in) str
IR_FunctionCall	See table 11	-
IR_Cast	IR_CastTo	See table 12

Table 10: IR expression mapping, r: right-hand operand, l: left-hand operand.

functionCall.function.name	SymPyInterface	SymPy
"sin"	Sine	sin
"cos"	Cosine	cos
"tan"	Tangent	tan
"cot"	Cotangent	cot
"sec"	Secant	sec
"csc"	Cosecant	csc
"asin"	Arcsin	asin
"acos"	Arccos	acos
"atan"	Arctan	atan
"acot"	Arccot	acot
"asec"	Arcsec	asec
"acsc"	Arccsc	acsc
"sinh"	Sinh	sinh
"cosh"	Cosh	cosh
"tanh"	Tanh	tanh
"coth"	Coth	coth
"sech"	Sech	sech
"csch"	Csch	csch
"asinh"	Arcsinh	asinh
"acosh"	Arccosh	acosh
"atanh"	Arctanh	atanh
"acoth"	Arccoth	acoth
"asech"	Arcsech	asech
"acsch"	Arccsch	acsch
"abs"	Abs	Abs
"fabs"	Abs	Abs
"pow"	pow	Pow
"ldexp"	mul(IR_MakeSymPy(args.head), pow(2, IR_MakeSymPy(args(1))))	Mul
"exp2"	pow(2, IR_MakeSymPy(args.head))	Pow
"exp10"	pow(10, IR_MakeSymPy(args.head))	Pow
"sqrt"	Sqrt	sqrt
"log10"	Log(IR_MakeSymPy(args.head), 10)	log(v, 10)
"log" with one argument	LogNat	log(v)
"log" with two argument	Log(IR_MakeSymPy(args.head), IR_MakeSymPy(args(1)))	log(v, b)
non-of-above	Exception	-

Table 11: IR_FunctionCall mapping

ExaStencils	**getTypeAsString(v)**	**SymPy**
IR_BooleanDatatype		(Python-built-in) bool
IR_IntegerDatatype		Integer
IR_RealDatatype		Float
IR_FloatDatatype		Float
IR_DoubleDatatype		Float
IR_CharDatatype	"<class 'int'>" "<class 'sympy.core.numbers.Integer'>"	(Python-built-in) chr
	"<class 'float'>" "<class 'sympy.core.numbers.Float'>" "<class 'sympy.core.numbers.Rational'>"	chr
other-strings		v
IR_StringDatatype		(Python-built-in) str
IR_ComplexDatatype(_)		Complex
other-types		throw new Exception(s"Casting: not implemented.")

Table 12: IR_Cast special treatment

SymPyInterface alias	SymPy/Python type
symbol_t	Symbol
float_t	Float
float_builtin_t	(Python built-in) float
rational_t	Rational
integer_t	Integer
integer_builtin_t	(Python built-in) int
boolean_t	(Python built-in) bool
string_t	(Python built-in) str
matrix_t	MutableDenseMatrix
matrixMutableDense_t	MutableDenseMatrix
matrixMutable_t	MutableMatrix
matrixImmutableDense_t	ImmutableDenseMatrix
matrixImmutable_t	ImmutableMatrix
array_t	DenseNDimArray
mod_t	Mod
mul_t	Mul
matMul_t	MatMul
matMulElementwise_t	HadamardProduct
add_t	Add
matAdd_t	MatAdd
matAddElementwise_t	MatAdd
abs_t	Abs
pow_t	Pow
matPow_t	MatPow
matPowElementwise_t	HadamardPower
exp_t	exp
equality_t	Equality
lessThan_t	LessThan
strictLessThan_t	StrictLessThan
greaterThan_t	GreaterThan
strictGreaterThan_t	StrictGreaterThan
and_t	And
or_t	Or
maximum_t	Max
tuple_t	Tuple
sine_t	sin
cosine_t	cos
tangent_t	tan
cotangent_t	cot
secant_t	sec
cosecant_t	csc
sinc_t	sinc
arcsin_t	asin
arccos_t	acos
arctan_t	atan
arccot_t	acot
arcsec_t	asec
arccsc_t	acsc
sinh_t	sinh
cosh_t	cosh
tanh_t	tanh
coth_t	coth

sech_t	sech
csch_t	csch
arcsinh_t	asinh
arccosh_t	acosh
arctanh_t	atanh
arccoth_t	acoth
arcsech_t	asech
arccsch_t	acsch
sqrt_t	sqrt
lognat_t	log
log_t	log
factorial_t	factorial
fibonacci_t	fibonacci

Table 13: Type alias used in SymPyInterface

6.2 Tables related to the Maxima interface

ExaStencils	**MaximaInterface**	**Maxima/Lisp/ABCL string used in calling callFunction 1/2/N ArgImplementation**
L2_IntegerConstant	integer	ABCL.Fixnum
L2_BooleanConstant	Bool	Lisp Symbol "T", or "NIL"
L2_RealConstant	floatingPoint	ABCL.DoubleFloat
L2_StringConstant	string	"STRING"
L2_StringLiteral	string	"STRING"
L2_Addition	ADD	"ADD"
L2_Subtraction	SUBTRACT	ADD(lhs, MULT(integer(-1), rhs))
L2_Multiplication	MULT	"MULT"
L2_Division	DIVIDE	"DIV"
L2_Modulo	MODULO	"MOD"
L2_Power	POWER	"POWER"
L2_EqEq	EQUAL	"EQUAL"
L2_Neq	NOT_EQUAL	NOT(EQUAL(arg))
L2_AndAnd	AND	NOT(OR(NOT(arg1), NOT(arg2))) /*FIXME*/
L2_OrOr	OR	"MORP"
L2_Negative	Negative	"NEG"
L2_Negation	NEGATION	"NEGATION"
L2_Equation	EQUAL	"EQUAL"
L2_LowerEqual	LESS_OR_EQUAL	OR(LESSTHAN(arg1, arg2), EQUAL(arg1, arg2))
L2_Lower	LESSTHAN	"LESSTHAN"
L2_GreaterEqual	GREATER_EQUAL	NOT(LESSTHAN(arg1, arg2))
L2_Greater	GREATER	NOT(LESS_OR_EQUAL(arg1, arg2))
L2_LeveledVariableAccess	cache in L2_VariableAccessCache and return a Symbol	maximaPackage addInternalSymbol(symbolName)
L2_PlainVariableAccess	cache in L2_VariableAccessCache and return a Symbol	maximaPackage addInternalSymbol(symbolName)
L2_FunctionCall	See table 15	
other types (L2_Node)	cache in L2_NameNodeCache and return a Symbol	maximaPackage .addInternalSymbol(symbolName)

Table 14: L2 ExaStencils to Maxima mapping

ExaStencils	**MaximaInterface**	**Maxima/Lisp/ABCL string used in calling callFunction 1/2/N ArgImplementation**
L3_IntegerConstant	integer	ABCL.Fixnum
L3_BooleanConstant	Bool	Lisp Symbol "T", or "NIL"
L3_RealConstant	floatingPoint	ABCL.DoubleFloat
L3_StringConstant	string	"STRING"
L3_StringLiteral	string	"STRING"
L3_Addition	ADD	"ADD"
L3_Subtraction	SUBTRACT	ADD(lhs, MULT(integer(-1), rhs))
L3_Multiplication	MULT	"MULT"
L3_Division	DIVIDE	"DIV"
L3_Modulo	MODULO	"MOD"
L3_Power	POWER	"POWER"
L3_EqEq	EQUAL	"EQUAL"
L3_Neq	NOT_EQUAL	NOT(EQUAL(arg))
L3_AndAnd	AND	NOT(OR(NOT(arg1), NOT(arg2))) /*FIXME*/
L3_OrOr	OR	"MORP"
L3_Negative	Negative	"NEG"
L3_Negation	NEGATION	"NEGATION"
L3_Equation	EQUAL	"EQUAL"
L3_LowerEqual	LESS_OR_EQUAL	OR(LESSTHAN(arg1, arg2), EQUAL(arg1, arg2))
L3_Lower	LESSTHAN	"LESSTHAN"
L3_GreaterEqual	GREATER_EQUAL	NOT(LESSTHAN(arg1, arg2))
L3_Greater	GREATER	NOT(LESS_OR_EQUAL(arg1, arg2))
L3_LeveledVariableAccess	cache in L3_VariableAccessCache and return a Symbol	maximaPackage addInternalSymbol(symbolName)
L3_PlainVariableAccess	cache in L3_VariableAccessCache and return a Symbol	maximaPackage. addInternalSymbol(symbolName)
L3_FunctionCall	See table 15	
other types (L3_Node)	cache in L3_NameNodeCache and return a Symbol	maximaPackage .addInternalSymbol(symbolName)

Table 15: L3 ExaStencils to Maxima mapping

ExaStencils	**MaximaInterface**	**Maxima/Lisp/ABCL string used in calling callFunction 1/2/N ArgImplementation**
IR_IntegerConstant	integer	ABCL.Fixnum
IR_BooleanConstant	Bool	Lisp Symbol "T", or "NIL"
IR_FloatConstant	floatingPoint	ABCL.DoubleFloat
IR_DoubleConstant	floatingPoint	ABCL.DoubleFloat
IR_RealConstant	floatingPoint	ABCL.DoubleFloat
IR_StringConstant	string	"STRING"
IR_StringLiteral	string	"STRING"
IR_Addition	ADD	"ADD"
IR_Subtraction	SUBTRACT	ADD(lhs, MULT(integer(-1), rhs))
IR_Multiplication	MULT	"MULT"
IR_Division	DIVIDE	"DIV"
IR_Modulo	MODULO	"MOD"
IR_Power	POWER	"POWER"
IR_EqEq	EQUAL	"EQUAL"
IR_Neq	NOT_EQUAL	NOT(EQUAL(arg))
IR_AndAnd	AND	NOT(OR(NOT(arg1), NOT(arg2))) /*FIXME*/
IR_OrOr	OR	"MORP"
IR_Negative	Negative	"NEG"
IR_Negation	NEGATION	"NEGATION"
IR_Equation	EQUAL	"EQUAL"
IR_LowerEqual	LESS_OR_EQUAL	OR(LESSTHAN(arg1, arg2), EQUAL(arg1, arg2))
IR_Lower	LESSTHAN	"LESSTHAN"
IR_GreaterEqual	GREATER_EQUAL	NOT(LESSTHAN(arg1, arg2))
IR_Greater	GREATER	NOT(LESS_OR_EQUAL(arg1, arg2))
IR_VariableAccess	cache in IR_VariableAccessCache and return a Symbol	maximaPackage addInternalSymbol(symbolName)
other types (IR_Node)	cache in IR_NameNodeCache and return a Symbol	maximaPackage .addInternalSymbol(symbolName)
IR_FunctionCall	See table 17	

Table 16: IR ExaStencils to Maxima mapping

FunctionCall string	**MaximaInterface**	**Maxima/Lisp/ABCL string used in calling callFunction 1/2/N ArgImplementation**
"sin"	Sine	"$SIN"
"cos"	Cosine	"$COS"
"tan"	Tangent	"$TAN"
"cot"	Cotangent	"$COT"
"sec"	Secant	"$SEC"
"csc"	Cosecant	"$CSC"
"asin"	Arcsin	"$ASIN"
"acos"	Arccos	"$ACOS"
"atan"	Arctan	"$ATAN"
"acot"	Arccot	"$ACOT"
"asec"	Arcsec	"$ASEC"
"acsc"	Arccsc	"$ACSC"
"sinh"	Sinh	"$SINH"
"cosh"	Cosh	"$COSH"
"tanh"	Tanh	"$TANH"
"coth"	Coth	"$COTH"
"sech"	Sech	"$SECH"
"csch"	Csch	"$CSCH"
"asinh"	Arcsinh	"$ASINH"
"acosh"	Arccosh	"$ACOSH"
"atanh"	Arctanh	"$ATANH"
"acoth"	Arccoth	"$ACOTH"
"asech"	Arcsech	"$ASECH"
"acsch"	Arccsch	"$ACSCH"
"abs"	Abs	"$ABS"
"fabs"	Abs	"$ABS"
"pow"	POWER	"POWER"
"ldexp"	MULT(args.head, POWER(integer(2), args(1)))	-
"exp2"	POWER(2, args.head)	-
"exp10"	POWER(10, args.head)	-
"sqrt"	SQRT	"$SQRT"
"log10"	DIVIDE(LogNat, LogNat(10))	-
"log"	DIVIDE(LogNat, LogNat(args(1)))	-
if functionName starts with "gen_"	Warning: starts with "gen_" it might be a generated function	-
else	throw Exception: not implemented	-

Table 17: FunctionCall to Maxima mapping

MIR type	**ExaStencils**
MIR_Symbol	IR_NameNodeCache, IR_VariableAccessCache
MIR_FixnumInteger	IR_IntegerConstant
MIR_Bignum	IR_IntegerConstant
MIR_String	IR_StringConstant
MIR_Boolean	IR_BooleanConstant
MIR_DoubleFloat	IR_RealConstant
MIR_Cons	IR_StringConstant(ERROR MESSAGE)
MIR_Nil	IR_NullExpression
MIR_OperatorFunctionCall	See table 19
otherTypes	Exception

Table 18: Mapping MIR type to IR ExaStencils

MIR type	ExaStencils
"+"	IR_Addition
"*"	IR_Multiplication
"-"	IR_Subtraction
"RATIONAL-VAL"	IR_Division
"/" if canPromoteFromIntToFloat==true and both operands are integer	IR_DoubleConstant(lhs / rhs)
"/" else	IR_Division
"^"	IR_Power
"Sqrt"	IR_FunctionCall, "sqrt"
"Log10"	IR_FunctionCall, "log10"
"LogNat"	IR_FunctionCall, "log"
"Sine"	IR_FunctionCall, "sin"
"Cosine"	IR_FunctionCall, "cos"
"Tan"	IR_FunctionCall, "tan"
"Cot"	IR_FunctionCall, "cot"
"Sec"	IR_FunctionCall, "sec"
"Csc"	IR_FunctionCall, "csc"
"ASin"	IR_FunctionCall, "atan"
"ACos"	IR_FunctionCall, "acos"
"ATan"	IR_FunctionCall, "atan"
"ACot"	IR_FunctionCall, "acot"
"ASec"	IR_FunctionCall, "asec"
"ACsc"	IR_FunctionCall, "acsc"
"Sinh"	IR_FunctionCall, "sinh"
"Cosh"	IR_FunctionCall, "cosh"
"Tanh"	IR_FunctionCall, "tanh"
"Coth"	IR_FunctionCall, "coth"
"Sech"	IR_FunctionCall, "sech"
"Csch"	IR_FunctionCall, "csch"
"ASinh"	IR_FunctionCall, "asinh"
"ACosh"	IR_FunctionCall, "acosh"
"ATanh"	IR_FunctionCall, "atanh"
"ACoth"	IR_FunctionCall, "acoth"
"ASech"	IR_FunctionCall, "asech"
"ACsch"	IR_FunctionCall, "acsch"
"Abs"	IR_FunctionCall, "abs"
None of the above	IR_StringConstant NOT IMPLEMENTED

Table 19: Mapping MIR_OperatorFunctionCall operand string to IR ExaStencils