Optimising Code Generation with haggies

T. Reiter

Nikhef, Science Park 105, 1098 XG Amsterdam, The Netherlands

Abstract

This article describes haggies, a program for the generation of optimised programs for the efficient numerical evaluation of mathematical expressions. It uses a multivariate Horner-scheme and Common Subexpression Elimination to reduce the overall number of operations.

The package can serve as a back-end for virtually any general purpose computer algebra program. Built-in type inference that allows to deal with non-standard data types in strongly typed languages and a very flexible, pattern-based output specification ensure that haggies can produce code for a large variety of programming languages.

We currently use haggies as part of an automated package for the calculation of one-loop scattering amplitudes in quantum field theories. The examples in this articles, however, demonstrate that its use is not restricted to the field of high energy physics.

Key words: Computer algebra, Code generation, Automation of perturbative calculations

PACS: 02.70.WZ, 07.05.Bx, 12.38.Bx, 12.38.Cy

Program Summary

Manuscript Title: Efficient Programs from Expressions with haggies

Authors: T. Reiter

Program Title: haggies

Email address: thomasr@nikhef.nl (T. Reiter)

URL: http://www.nikhef.nl/~thomasr/ (T. Reiter)
Programming language: Java, JavaCC
Operating system: Any system that runs the Java Virtual Machine
RAM: determined by the size of the problem
Number of processors used: dynamical, up to the number of installed CPUs
Keywords: Computer algebra, Code generation, Automation of perturbative calculations
PACS: 02.70WZ, 07.05.Bx, 12.38.Bx, 12.38.Cy
Nature of problem: Generation of optimised programs for the evaluation of possibly large algebraic expressions
Solution method: Java implementation
1. Motivation

In physics problems with large numbers of parameters the appearance of large expressions is commonly observed. Exploiting the symmetries of the problem often simplifies the expression sufficiently to allow for a direct numerical implementation. However, these simplifications sometimes are not obvious due to the lack of an algorithmic description in which order to apply the underlying relations. In this case they are not suitable for an automated setup.

One representative class of problems are higher order corrections in perturbative quantum field theory. In order to meet the precision of modern collider experiments such as the Large Hadron Collider (LHC), the leading order approximation of scattering amplitudes is often not sufficient and at least one-loop precision is required for many processes with up to four particles in the final state.\[1\]. It is therefore not surprising that recently the automation of such calculations has become a very active field of research, leading to new tools and promising techniques\[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\]. The interested reader will find a more general overview over the current status of higher order corrections in particle physics for example in \[15\] or in \[1\].

The calculation of scattering amplitudes to one-loop precision can be divided into two subproblems. One part of the amplitude, the real corrections, consists of tree-level Feynman diagrams, describing the radiation of an extra, unobserved particle. The second half of the calculation describes the exchange of a virtual particle and leads to one-loop diagrams. Both parts of the calculation contain singularities which cancel after summing over both of the two contributions. In Quantum Chromodynamics (QCD), a commonly used technique for the regularisation of these singularities is the subtraction method by Catani and Seymour\[16\], which has led to several automated implementations \[17, 18, 19, 20, 21\]. Combined with existing tools for tree-level calculations the real corrections, these implementations provide a complete solution for the real emission corrections.

The calculation of the virtual corrections requires the computation of one-loop Feynman diagrams. The two basic strategies are a fully numerical approach, which is mainly followed in the implementation of unitarity-based methods \[2, 7, 12\], and a semi-numerical/algebraic approach which appears to be well-suited for calculations based on Feynman diagrams \[3, 6, 8, 9, 11\].
2. Background

Here, the package Golem is discussed, which uses the latter technique to produce a numerically stable representation of the virtual corrections of cross-sections. This package forms the original motivation for the development of haggies.

The matrix element for the virtual corrections to a process in QCD with $N$ particles in the final state can be written as

$$ M_{\text{virt}} = \sum_{|c\rangle \in \mathcal{B}} A_c(p_a, p_b; p_1, \ldots, p_N) \cdot |c\rangle, \quad (1) $$

where $\mathcal{B}$ denotes some basis for the colour tensor of the external particles. The right hand side can be decomposed further by projecting onto helicity states and by using the fact that the matrix element is defined as the sum over all contributing Feynman diagrams $G$,

$$ M_{\text{virt}}^{(\lambda)} = \sum_{G} \sum_{|c\rangle \in \mathcal{B}} G_c(p_1^\lambda, \ldots, p_N^\lambda) \cdot |c\rangle, \quad (2) $$

$$ |M|^2 = \sum_{\{\lambda\}} M_{\text{virt}}^{(\lambda)} (M_{\text{virt}}^{(\lambda)})^\dagger. \quad (3) $$

One of the tasks of the Golem implementation is to generate a Fortran 90 subroutine for each diagram $G_c(p_1^\lambda, \ldots, p_N^\lambda) \cdot |c\rangle$. Each of these diagrams is a linear combination of tensor integrals of the type

$$ I_{N}^{\mu_1, \ldots, \mu_r}(S) = \int \frac{dnk}{i\pi^{n/2}} \frac{k^{\mu_1} \ldots k^{\mu_r}}{\prod_{j=1}^{N} [(k + r_j)^2 - m_j^2]} $$

contracted with corresponding coefficients $c_{\mu_1 \ldots \mu_r}$. The vectors $r_j$ are linear combinations of the external momenta and $m_j$ are the propagator masses. The integrals are dimensionally regularised in $n = (4 - 2\varepsilon)$ dimensions and the result can be expressed as a Laurent series $A/\varepsilon^2 + B/\varepsilon + C + O(\varepsilon)$. The algorithm described in [3] is used to express the tensor integrals in terms of Feynman parameter integrals of the form

$$ I_{N}^{l_1, \ldots, l_p}(S) = (-1)^N \Gamma(N - d/2) \times $$

$$ \int \prod_{j=1}^{N} (dz_j \Theta(z_j)) \delta \left(1 - \sum_{i} z_i \right) \frac{z_{l_1} \ldots z_{l_p}}{\left(-\frac{1}{2} z^T S z - i\delta\right)^{N-d/2}} \quad (5) $$
with the matrix

\[ S_{ij} = (r_i - r_j)^2 - m_i^2 - m_j^2. \]  

(6)

The coefficients in front of these integrals are in general rational polynomi-
als in terms of spinor products of external vectors (\( \langle p_i p_j \rangle \) and \([p_i p_j] \)) and constants such as masses and coupling constants.

Figure 1: A Feynman diagram contributing to the process \( gg \to b \bar{b} b \bar{b} \) at next-to-leading order in [QCD].

The integrals in Equation (5) — or rather form factors that consist of com-
binations of these integrals — are evaluated through a Fortran 90 library
and translate directly to function calls. However, for complicated \( 2 \to 4 \) pro-
cesses the number of terms and the complexity of the coefficients in front of
the form factors can grow very large. For example, in the process \( gg \to b \bar{b} b \bar{b} \)
we observe for the textual representation of the expression for a single six-
point diagram (see Figure 1) a size of 9MB (43,918 terms). Expressions of
this complexity cannot be compiled efficiently by standard Fortran 90 com-
pilers; often a compilation is not possible at all, prohibited by the sheer size
of the expression. Some of the reasons for that are:

- typically, compilers implement algorithms that produce optimal code
  for relatively small subprograms at the expense of a non-linear time
  and/or resource consumption at compile time. The GNU compiler
collection\(^1\) for example, uses a graph colouring algorithm for register
  allocation which is known to have a runtime that scales quadratically

\(^1\)version 4.3
with the number of temporary variables. These algorithms fail for very large subprograms.

- most compilers make no or little assumptions about the algebraic properties of binary operators. Especially for overloaded operators, properties such as commutativity or associativity cannot be specified. Therefore, expressions very often have to be evaluated unoptimised.

- side-effects, aliasing and the possible dependence on global variables limit a compiler’s possibilities of reusing the outcome of a function call. Therefore, Common Subexpression Elimination (CSE) does typically not include function calls.

Although this list of reasons might look to the reader like a list of shortcomings of current compiler technology, there are good reasons not to touch the current behaviour of the compilers: for example, when overloading the operator ‘*’ with a matrix product, commutativity should not be assumed by the compiler, and the result of a function that returns random numbers or the current time should not be reused.

One alternative would be an extension of the target language by an appropriate set of keywords for marking functions and operators as symmetric or free of side-effects. Fortran 90, for example, has a keyword ‘pure’ which allows to specify a function as side-effect free. However, further restrictions would be necessary to allow for fully automated optimisation — on the other hand these restrictions (e.g. absence of pointers and global variables) can become prohibitive in other parts of the program.

Another alternative is to put these additional assumptions into a preprocessor that then presents the already optimised source code to an existing compiler. Within the Golem implementation, we have chosen this solution, which lead to the development of the program haggies. Although rooted in the field of high energy physics, the possible applications of haggies are much broader as will be shown in the demonstration programs.

The remainder of this article is structured as follows: Section 3 describes the algorithm used to transform the input expressions. The installation and the system requirements are briefly described in Section 4 then follows a number of examples in Section 5. In the appendix we give a complete reference over the language of the configuration and template files.

---

Overloaded operators have to be viewed as function calls, too.
3. Description of the Algorithm

3.1. Overview

The program consists of several consecutive parts each of which runs on the representation of the expression produced by the previous part. A schematic overview is given in Figure 2. Each of the steps is described below.

3.2. Parsing and AST Representation

The parsing of the textual representation of the input expressions has the only challenge that an efficient memory representation has to be used. For the parsing itself the Java parser generator JavaCC is used, which produces a recursive descent parser [22] from an Extended Backus-Naur Form (EBNF) description of the expression grammar. At this point the program offers two interfaces: the expression syntax used by Form [23, 24] which is fairly compatible with the expression syntax of most computer algebra systems,
and a reader for Mathematica expressions, which distinguishes itself from most other programs by the use of square brackets in function calls instead of round brackets.

Figure 3: [AST] representation of the expression $5/2xyf(x) - xf(x) \cdot (y - 1) + 3y^2$; here, the second term is already expanded. The dotted lines separate basis and exponents inside a product. A mixed tree/array representation is used to guarantee quick access to the elements. The character ‘@’ is used here to denote a pointer.

The Abstract Syntax Tree (AST) implements commutativity and distributivity to store a sorted, fully expanded representation of each expression. Both sums and products use arrays to store their terms (resp. factors). Figure 3 shows a pictogram of the memory representation of a simple expression.

During the construction of the AST the expression is brought into a canonical form by expanding all products and by sorting all factors (resp. terms) inside products (resp. sums). At this point we do not yet introduce a Directed Acyclic Graph (DAG) representation, as this is achieved later during the CSE.

3.3. Type Inference

Although for many problems it is sufficient to work with a single numerical data type throughout the whole calculation, it often adds more flexibility to

---

3 Automatic expansion can be suppressed by specifying the -E option at the command line.
mix different data types. As an example the user is referred to Section 5.1, which implements a Runge-Kutta integrator for the differential equation $\vec{y}' = f(x, \vec{y})$. This example requires the notion of at least two data types, a floating point type (e.g. `double`) for $x$ and a vector type (`double[]`) for $\vec{y}$ and $\vec{y}'$. Without this flexibility the only way out is to implement the algorithm for a fixed number of variables $y_1, \ldots, y_n$ and to regenerate the code whenever $n$ needs to be changed.

In the case of the Golem implementation, having multiple data types is a necessity rather than added flexibility. One has to deal with integer indices, Laurent expansions in $(1/\varepsilon)$, Taylor expansions in $\varepsilon$ and complex and real numbers at the same time (see Table 1). Between the objects of different

| L   | Laurent expansion, such as $A/\varepsilon^2 + B/\varepsilon + C + O(\varepsilon)$ |
| E   | Taylor expansion, such as $X + Y\varepsilon + Z\varepsilon^2 + O(\varepsilon^3)$ |
| I   | Integer numbers |
| R   | Real numbers |
| C   | Complex numbers |

Table 1: Basic types used for the objects in Golem expressions.

types there are valid operations such as $C\cdot C \rightarrow C$ or $L\cdot E \rightarrow L$. On the other hand, ill-defined expressions, such as $C/L$, indicate an error in an earlier step of the calculation and should be rejected. The type system of haggies allows also for function types and implicit coercions. Table 2 shows an extract of the list of definitions for the Golem type system.

The program haggies expects each symbol that appears on the right-hand side of a calculation to be defined with a type. The type of each subexpression is inferred starting at the leaves of the AST using the type information of the operations and functions and if necessary, by inserting coercions. If for any subexpression the type cannot be inferred by the defined rules the program reports an error and terminates. It is, however, not considered an error if there is more than one possibility to infer the type of a subexpression; the program will choose one possibility, relying on the user to define a sound type system in the configuration files.\footnote{This also means that there is no backtracking if the inference fails beyond a choice point.}
operations
L·E → L
L + L → L
I/I → R
...
coercions
I → R
R → C
C → L

Laurent expansion with $A = B = 0$.

definitions
\[ \varepsilon : \quad E \]
\[ m_T, m_W, e, g_s, \ldots : \quad R \] \quad masses and coupling constants
\[ \langle p_ip_j \rangle, [p_ip_j] : \quad C \] \quad spinor products
\[ I^3(l_1, l_2) : \quad I, I \rightarrow L \]
...

Table 2: Extract of the Golem type system.

The inferred types will become relevant in Section 3.5, where CSE is discussed: consider the expression \( f(xy) + g(xy) \) and the following, generated program fragment:

\[
t1 := x*y;
t2 := f(t1);
t3 := g(t1);
t3 := t2+t3;
\]

The newly introduced variable \( t1 \) can have a type different from that of \( t2 \) and \( t3 \). In a strongly typed language this type information has to be made available to prepend the above code segment by the necessary declarations, such as the following:

```
var t1: integer;
var t2: real;
var t3: real;
```

Another reason for attaching type information to all subexpression is the use of user-defined data types such as multiprecision numbers or to allow the
use of interval arithmetic\footnote{See also the examples in Section 5.1.} in languages without operator overloading these cases require to replace the operator expressions by function or method calls.

\subsection{3.4. Multivariate Horner Scheme}

The first transformation applied to the expressions is a multivariate Horner scheme. This step basically follows the method proposed in \cite{25}. For univariate polynomials the Horner scheme provides the fastest implementation of evaluating the polynomial at a given position, i.e. the evaluation can be performed with the least number of multiplications and additions \cite{26}. In the multivariate case, however, it is not clear a priori which evaluation scheme leads to the smallest number of arithmetic operations. Consider, for example the polynomial

\[ f(x_1, x_2, x_3) = x_3^2 x_2 + x_1^2 x_3 + x_1^2 x_2 x_3. \]  

\begin{equation}
\tag{7}
\end{equation}

Obviously, extracting \( x_2^2 \) from each term saves the largest number of operations, and we obtain

\[ f(x_1, x_2, x_3) = x_2^2 (x_1 x_2 + x_3 + x_2 x_3). \]  

\begin{equation}
\tag{8}
\end{equation}

The next decision already becomes less trivial, as we can either select \( x_2 \) or \( x_3 \), and one obtains

\[ f(x_1, x_2, x_3) = x_1^2 (x_2 (x_1 + x_3) + x_3) \quad \text{or} \quad \]

\[ f(x_1, x_2, x_3) = x_1^2 (x_1 x_2 + x_3 (1 + x_2)) \]  

\begin{equation}
\tag{9a}
\end{equation}

\begin{equation}
\tag{9b}
\end{equation}

respectively. The representation in Equation (9a) requires one multiplication less than Equation (9b). This simple example already shows that the order in which the variables are extracted will impact the efficiency of the evaluation.

In general, a multivariate Horner scheme can be generated by Algorithm \[ \square \]. At each step one or more variables \( x_1^{m_1} \cdots x_n^{m_n} (0 \leq m_i) \) are selected and the polynomial is split into terms that contain the selected product of variables and terms \( a_0 \) from which this product cannot be factored out. The algorithm still leaves the heuristics behind select_coeffs undefined. A global optimisation would try to minimise the number of multiplications of the final expression, since the number of additions remains constant. On the other hand it is clear that such a global strategy needs to consider far too many possibilities and hence cannot be implemented efficiently.
Algorithm 1 multivariate_horner(f)

Require: polynomial $f(x_1, \ldots, x_n)$
\[
\vec{m} \leftarrow \text{select_coeffs}(f)
\]
Find $a_1, a_0$ such that $f(x_1, \ldots, x_n) = x_1^{m_1} \cdots x_n^{m_n} a_1 + a_0$.
\[
b_1 \leftarrow \text{multivariate_horner}(a_1)
b_0 \leftarrow \text{multivariate_horner}(a_0)
\]
return $x_1^{m_1} \cdots x_n^{m_n} b_1 + b_0$

The simplest strategy that runs at each step of the algorithm in linear time is to select only one variable at a time. The authors of [25] suggest the selection of the variable which appears in the highest number of terms, which leads to the highest immediate decrease of arithmetic operations at a given step. Our implementation allows in a very straightforward manner to add new strategies but it proves difficult to come up with strategies which, in combination with CSE, outperform the original method.

3.5. Common Subexpression Elimination

Common Subexpression Elimination denotes a source code transformation in which temporary variables are introduced for each subexpression such that it is only calculated once and can be reused at a later point in the calculation without having to be calculated again. A great simplification for our program is the fact that we transform only expressions and therefore can neglect all complications arising from control structures such as loops and jumps.

We consider again the example from Figure 3. The Horner form for this expression as produced by our program is
\[
x \cdot \left(3 \left(\frac{2}{3} y f(x) + f(x)\right) + 3 y^2\right).
\]
If $f(x)$ encodes a computationally expensive procedure call a good programmer would have written
\[
t := f(x);
result := x*(3/2*y*t+t) + 3*y*y;
\]

\footnote{See, for example [22].}
\footnote{This program fragment assumes that the division operator denotes a floating point division, not integer division.}
CSE does precisely the same; the program produced from this example is shown below:

\[
\begin{align*}
$1 & := f(x); \\
$2 & := $1 \times y; \\
$3 & := 3/2 \times $2; \\
$4 & := $1 + $3; \\
$5 & := y \times y; \\
$6 & := 3 \times $5; \\
$7 & := $4 \times x; \\
$8 & := $6 + $7; \\
\text{result} & := $8;
\end{align*}
\]

Since one of the main goals is to minimise the number of multiplications the program takes special care about high powers of variables. For a single variable the most efficient way to calculate an integer power is the binary exponentiation which uses the binary representation of the exponent in order to compute the power with the least number of multiplications. For example \(x_3^5\) can be computed as

\[
x_3^5 = x_3^{2^0 + 2^2} = x_3^{1+2 \cdot 2} = x_3 \cdot (x_3^2 \cdot x_3^2),
\]

which requires 3 multiplications, since the value of \(x_3^2\) can be stored and reused. If we apply this trick to each variable of the term \(x_1^3 x_2^4 x_3^5\) we can compute the expression with nine multiplications. A more efficient way to compute the same result is multi-exponentiation: here, we consider the binary representation of a vector of exponents,

\[
\begin{pmatrix}
3 \\
4 \\
5
\end{pmatrix}
= \begin{pmatrix}
1 \\
0 \\
1
\end{pmatrix} + 2 \left( \begin{pmatrix}
1 \\
0 \\
0
\end{pmatrix} + 2 \begin{pmatrix}
0 \\
1 \\
1
\end{pmatrix} \right)
\]

The according factorisation of the product would be

\[
(x_1 x_3)(x_1 \cdot (x_2 x_3)^2)^2
\]

which can be calculated with only six multiplications. This can also be seen from the output produced by haggies:

\[
\begin{align*}
$1 & := x1 \times x3; \\
$2 & := x2 \times x3;
\end{align*}
\]
\$3 := \$2*\$2;
\$4 := \$3*x1;
\$5 := \$4*\$4;
\$6 := \$1*\$5;

3.6. Register Allocation

It would, of course, be very inefficient if the algorithm stopped at this point since a new variable is introduced for each operation. Many of the intermediate results need to be saved only for a short period and can be discarded after a small number of instructions. Hence in a further step haggies needs to determine the life span of each intermediate result and replace the virtual variables \$1, \$2, \ldots by a much smaller number of actual local variables. At this point, an actual compiler would try to match the local variables with physical processor registers and introduce so-called memory spills whenever the number of registers on the target machine is not sufficient. We can, however, assume an unlimited number of registers and try to minimise the number of actually used registers; the allocation of physical registers for these intermediates is left to the compiler.

Usually this step is implemented by a graph colouring algorithm [27]: the graph is constructed by connecting all virtual variables with an overlapping life range. This graph is then coloured by the number of colours corresponding to the number of available registers. If the graph is not colourable memory spills are introduced. This type of algorithm typically produces allocations close to the optimal solution but the price to pay is a worst-case performance which grows quadratically in the number of life ranges. For very large expressions this approach can render compilation impossible.

The program haggies implements a single pass strategy that runs in linear time [28]. This Linear Scan Register Allocation algorithm has been proposed with Just-In-Time compilation in mind, where compile time and run time performance are equally important. In our case its striking feature is the linear scaling behaviour.

We run this algorithm separately for each different data type as defined by the type inference. For the example from Figure 3 it suffices to introduce two variables. After register allocation the resulting program is as follows:

\begin{align*}
t1 & := f(x); \\
t2 & := t1 * y; \\
t2 & := 3/2 * t2;
\end{align*}
\[ t_1 := t_1 + t_2; \]
\[ t_2 := y \times y; \]
\[ t_2 := 3 \times t_2; \]
\[ t_1 := t_1 \times x; \]
\[ t_1 := t_1 + t_2; \]
\[ \text{result} := t_1; \]

3.7. Output and Syntax Transformations

As already mentioned, one important consideration in the design of the program haggies is its independence from the target language. For a large class of problems and languages only minor differences distinguish the output, such as different notations for the assignment operator (\(=\) as opposed to \(:=\)), the presence of a semicolon at the end of a statement or requirements with respect to indentation and line continuations in fixed-form and some scripting languages. These differences are very often solved by running a standard text processor, such as `sed` over the output produced by some computer algebra program. However, some output formats can not easily be achieved by simple text transformation. Some common problems are listed below:

- The (non-)existence of an exponentiation operator: In C this operator has to be emulated by a function call to the `pow` function, some languages use the operator `**` instead of `^`. In some languages the function name depends on the data type, e.g. `powf` and `powd` for single resp. double precision numbers.

- Languages such as Lisp require the expression in prefix notation and/or use a different notation for function calls (`(f x)` instead of `f(x)`).

- The division operator between two integer numbers in some languages is interpreted as an integer division. Hence, a naive translation of \(1/3\) yields 0 instead of the correct numerical value 0.333\ldots.

- The use of non-standard data types in languages without operator overloading needs to be replaced by function or method calls. In Java, the use of the class `BigDecimal` requires method calls such as `x.add(y)` rather than `x+y` similar problems arise in the context of multiprecision libraries in other languages.
We solve the above problems by allowing for the specification of patterns for each operation along with each declared data type. The declaration of a data type represented by the `BigDecimal` class in Java could be declared as follows:

```java
@type
R = "BigDecimal", "%s.add(%s)",
   "%s.subtract(%s)", "%s.negate()";
I = "long";
F = "double";

@operation
R * R → R =
   "%s.multiply(%s)", "%s.divide(%s)";
R * I → R = "%s.multiply(%s)", "%s.divide(%s)",
   "BigDecimal.valueOf(%2$s).divide(%1$s)";
```

Lines 2–3 define the type ‘R’ which in Java is represented by the class `BigDecimal`; the strings that follow the type name are, in this order, the pattern to be applied for an addition, subtraction and the unary minus. The patterns are defined by the syntax for the Java class `java.text.Formatter`. Similarly, line 4 defines a type ‘I’ represented by a long integer in Java. In this case we can leave out the patterns since the default operators can be used. Lines 7–10 define the multiplication and division of these two data types. Here, the first two patterns on the right-hand side define multiplication and division. An optional third pattern denotes the reverse division (I/R rather than R/I).

Similar transformations can be defined for symbols and functions. The examples below collect some commonly used patterns. A systematic description of the syntax can be found in Appendix A. The character "\" denotes a blank character.

```java
@define
sin , cos : F → F = "Math.%s(%s)";
mat : I , I → F = "m[%2$s][%3$s]";
vec . . . : I → F = "%s[%s]";
abs : R → R = "%2$s.abs()";
ARRAY3 : R, R, R = "{%2$s,%3$s,%4$s}";
```

The first line contains names which need to be qualified by a module or class name. The function ‘mat’ is a placeholder for the array ‘m’. The next line
contains an ellipse on the left hand side of the definition which applies to all symbols that start with the prefix ‘vec’. The function ‘abs’ is transformed into the according method call and the last example shows a function which is transformed into an array literal.

These examples also show a couple of shortcomings and pitfalls. First of all, there is no function overloading in haggies. It is, for example, not allowed to define the function ‘abs’ twice with a different type. Currently, there is also no notion of argument lists of variable length. Hence, we have defined ‘ARRAY3’ in the above example and would have to define different symbols for arrays of different size. The user should be aware that the pattern ”%1$s” holds the function name and therefore the arguments start at ”%2$s”.
4. Installation and Requirements

4.1. Requirements

haggies runs on a Java virtual machine that supports the Java API in version 1.5 or higher. For compilation of the sources from scratch also JavaCC version 5.0 or higher is needed. The examples have additional requirements which are described separately with each example. The compilation is based on Apache’s Ant tool.

4.2. Download

All files can be downloaded from the URL


where filename is one of the following:

haggies.jar containing the precompiled classes as Java Archive (JAR) file. This is the only file required to run haggies

haggies-src.jar containing the source code of the project including the demo programs. It should be noted, that JavaCC version 4.3 or newer and Apache Ant are required in order to compile the project.

haggies-demo.zip containing the demo files only.

4.3. Compilation

For those users who prefer to compile the project from the source files we describe here the installation procedure. This section can be skipped when working with the precompiled JAR file.

In the following we assume a Linux like system with the Java Development Kit (JDK) installed.

1. The sources need to be extracted into a new directory: copy or move the file haggies-src.jar to that directory and change the working path to the target directory. Run jar xf haggies-src.jar.

2. The directory now contains, amongst others, the file

   site-properties.template

---

8JavaCC is available from [https://javacc.dev.java.net](https://javacc.dev.java.net)
which needs to be copied to `site.properties`. Open the file in an 
editor and change the directory `javacchome` to the installation path of 
JavaCC.

3. Run `ant` to compile everything. If the compilation was successful the 
file `dist/haggies.jar` has been created.

4. With the GNU Java compiler installed there is the optional choice 
to create a binary executable from the same sources. To do this, in 
`site.properties` the option `gcj.installed` must be set to ‘true’; 
then one can run `ant bin`. The executable is written to `dist/haggies`.

5. The files in the subdirectory `build` are not needed anymore and can 
be removed with the command `ant clean`.

5. Examples

In order to use the example files one needs to download and unpack the 
file `haggies-demo.zip` The examples are located in the subdirectory `demo`.

5.1. Runge-Kutta Integrator

**Synopsis:** comparison of different Runge-Kutta methods 
applied to the Lorenz oscillator

**Objective:** basic usage of `haggies`, generating output for 
different target languages

**Requirements:** Form, Fortran 90 compiler

**Directory:** demo/rk

5.1.1. Overview

In this example, we construct implementations of the Runge-Kutta method 
for different programming languages from the according coefficients. Given 
the initial value problem

$$\frac{d}{dx} \vec{y}(x) = f(x, \vec{y}), \quad \vec{y}(x_0) = \vec{y}_0. \quad (14)$$

The series

$$\vec{y}_{n+1} = \vec{y}_n + h \sum_{i=1}^{s} b_i \vec{k}_i, \quad \sum_{i=1}^{s} b_i = 1 \quad (15)$$

---

9This file is not needed if `haggies-src.jar` has been downloaded already.
with
\[ k_i = f(x_n + c_i h, y_n + a_{i1} h k_1 + \ldots + a_{i,i-1} h k_{i-1}) \] (16)
defines an explicit Runge-Kutta method. The choice of the coefficients \( b_i, c_i \) and \( a_{ij} \) determines the order \( p \),
\[ \bar{y}(x_0 + nh) = y_n + O(h^p). \] (17)
In practice, one works with two different sets \( b_i \) and \( d_i \) which are chosen such that replacing \( b_i \) by \( d_i \) in the above equations results in a method of order \( (p - 1) \). The difference between the two results gives an estimate for the error and allows for an adaptive choice of \( h \).

As an application of the Runge-Kutta method we integrate the Lorenz oscillator
\[
\frac{d}{dt} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \sigma \cdot (y - x) \\ x \cdot (\rho - z) - y \\ x \cdot y - \beta \cdot z \end{pmatrix}
\] (18)
for \( \sigma = 10, \beta = 8/3, \rho = 28 \) and \( 0 \leq t \leq 20 \). This set of coupled, non-linear differential equations describes an unstable dynamics which is very sensitive to numerical inaccuracies and therefore provides a good benchmark for different integration methods.

5.1.2. C-Implementation

The file \( \text{rk.frm} \) encodes the coefficients for different methods [30, 31, 32, 33] as a Form program which generates the expressions for \( \bar{y}_{n+1} \) and \( \bar{z}_{n+1} = \bar{y}_{n+1}|_{b_i \rightarrow d_i} \). Although there are more direct methods for constructing an implementation of the integrator, this example shows how one can produce different programs from the same computer algebra source using \( \text{haggies} \) as a converter. For the simplest implemented integrator of order 2 the Form program generates the two expressions
\[
y_1 = y_0 + 1/2 \cdot f(x_0 + h, y_0 + f(x_0, y_0) \cdot h) \cdot h + 1/2 \cdot f(x_0, y_0) \cdot h; \quad (19a)
z_1 = y_0 + f(x_0, y_0) \cdot h; \quad (19b)
\]
For the higher order methods these expressions are substantially longer.

The next step is to generate a C-file from the above expressions using \( \text{haggies} \). The program requires a configuration file (Listing 1) and a template file (Listing 2) which together determine the transformations.
The configuration file defines all occurring symbols, the admitted operations between them and their representation in the output file. The template file, on the other hand, determines the structure of the output file and the format of the statements and declarations.

The \texttt{@language} statement in line 1 of Listing 1 defines the languages, both of the input and the output file.\footnote{See also Appendix A.1} In line 2–4 we define the two different data types that appear in the problem, a scalar numeric type \texttt{S} for $x_0$ and $h$ and a vectorial type \texttt{T} for $\vec{y}_0, \vec{y}_1, \ldots$. Here, we have assigned the \texttt{C} data type \texttt{float} to both formal data types which corresponds to the case where $\vec{y}$ is just one-dimensional.

In Lines 5–7 we define coercions, which are applied whenever necessary. The data types \texttt{@int} and \texttt{@int/@int} correspond to the data types of integer and rational literals. If the program encounters the expression $5 \vec{y}$ then first the coercion \texttt{@int→T} is applied after which the operation \texttt{S*T→T} can be matched. The optional right-hand side of a \texttt{@coerce} statement defines the textual transformation of the current expression, hence the literal ‘5’ will be
transformed into ‘5.0’. In the case of the data type @int/@int the two fields of the pattern correspond to numerator and denominator of the fraction.[11]

The @define statement (Lines 8–11) declares all symbols which are valid in the input expression and associates them with a type. In order to denote groups of symbols with a common prefix and/or suffix one can use the ellipsis; instead of ‘y0, y1: T’ we could also have written ‘y ... : T’. The data type of f is a functional type which maps the argument list of type ‘(S, T)’ to the return type ‘T’.[12]

So far we have not defined yet which operations are valid between the different data types. In the @operator section in Lines 12–14 we define the multiplication[13] between the data types ‘S’ and ‘T’. An implicit declaration of addition, subtraction and negation is always made together with the @type statement. Hence, also the operations ‘T+T→T’ and ‘S+S→S’ are permitted. A detailed reference on these two statements can be found in Appendices A.5 and A.2.

The last statement of the configuration file forms the @polynomial statement in Line [15]. Here one lists all symbols that should be considered as parts of the monomials when applying the Horner scheme to the expression. All remaining symbols form the coefficients of the monomials together with the numerical coefficients. The keyword @polynomial can also be abbreviated by @poly.[14]

The program haggies transforms the expressions from the input file into a sequence of assignments that calculates this expression numerically. In general, these statements need to be embedded syntactically in some kind of module, class or procedure. Since in our example the C-output is for demonstration purposes only the template file in Listing 2 has been kept as simple as possible. As we progress through the further examples we will encounter more complex template files. Template files can contain parts of a program in the target language, which will be emitted to the output file verbatim, plus markup tags which are enclosed by a pair of brackets [% ... %], which are highlighted in colour through-out this article. Inside a tag a single quote starts a comment that spans to the end of the tag.

In Line 7 we start a loop which iterates over all symbols that are intro-

---

[11] See also Appendix A.4
[12] See also Appendix A.3
[13] and therefore also division
[14] See also Appendix A.7
float integrate(
    float (*f)(float, float),
    float x0,
    float y0,
    float h,
    float& err)
{
    // Markup which is processed by haggies is marked in red.

    // Listing 2: Template file `c.out' to produce C output for the Runge-Kutta example.
    err = abs(y1 - z1);
    return y1;
}
duced by the CSE; these symbols have the names ‘$1’, ‘$2’ and so on. A loop has the general form

\[
\begin{array}{l}
\text{% @for name opt_1 [=}value_1\} opt_2 [=}value_2\} \ldots \\
\text{loop body[%} \\
\text{@end @for %]}
\end{array}
\]

The name of the iterator in our case is ‘symbols’; inside the loop it defines the name of each symbol [\$_.] together with its type name [%type.name%] (here ‘S’ and ‘T’) and its type representation [%type.repr%] (here ‘float’ in both cases). In general the type name is the left hand side of a type declaration and the type representation the right hand side; the type representation defaults to the type name if the right hand side was omitted in the configuration file. Most loops also define the two Boolean values [%is_first%] and [%is_last%] which can be used inside an [%@if%] statement to output some section only before the first or after the last element of a loop\textsuperscript{15}. The loop is terminated in Line\textsuperscript{10}. Note that all arguments of the [%@end @for%] statement are ignored and merely serve as comments which makes it easier to maintain nested loops.

The pair of options ‘match’ and ‘format’ is common to most tags and allows a pattern based transformation of the symbol concerned. The value supplied to the option ‘match’ is interpreted as a regular expression according to the syntax accepted by the Java class \texttt{java.util.regex.Pattern}; the value of ‘format’ is a \texttt{Java printf} format\textsuperscript{16}. The groups\textsuperscript{17} matched by the regular expression are taken as the fields of the format. In our example, we have the pair [%match=’\$(d+)’ format=’t%s’%]; if applied to the symbol ‘$123’ the matched group is the substring ‘123’ which is substituted for the field ‘%s’ in the format and hence transforms to the symbol ‘t123’.

The second loop of the program spans the Lines\textsuperscript{11}-\textsuperscript{19}. Here we iterate over the instructions that compute the expressions. In this loop, the symbol [%$._%] denotes the left hand side of the assignment. Furthermore, one can access the right hand side expression through the tag [%expression%]

\textsuperscript{15}The difference to placing the code just before or after the loop becomes important for empty iterators which trigger the loop zero times.

\textsuperscript{16}The format is defined by the methods of the class \texttt{java.util.Formatter}. Because of the similarity to the \texttt{printf} format in C we call it \texttt{Java printf} format.

\textsuperscript{17}A group denotes a subpattern delimited by a pair of parentheses. Groups are used in substitutions to use parts of the match in the replacement.
and its type through [% type.name %]. It should be mentioned that for the right hand side expression the match-format pair is not applied to the textual representation of the expression but to each single symbol that occurs inside the expression and is simply ignored for symbols that do not match the regular expression.

On the left hand side of the assignment we have put a select statement. This is necessary because the match-format transformation should only be applied to symbols starting with a ‘$’ but not to ‘y1’ and ‘z1’. The regular expression ’(.)*’ picks only the first letter of the symbol which is then compared to the values in the [% @case %] clauses. The general form of a select statement is

```%
@select name opt1[=value1] opt2[=value2] ...
@case value11 value12 ...
  %]branch 1 [%
@case value21 value22 ...
  %]branch 2 [%
  :
@else
  %]branch n [%
@end @select %
```

The else-branch can be omitted.

The C-file is generated by running haggies on the input using the above files as configuration and template files. In the simplest case the command looks as follows

```
java -jar haggies.jar -cc.in -tc.out -omethod_h.c method_h.txt
```

Here we assume that the Equations (19) are found in the file ‘method_h.txt’, which must define the symbols ‘y1’ and ‘z1’ by definitions of the form ⟨symbol⟩ ‘=’ ⟨expression⟩ ‘;’.

The option -c specifies the configuration file, -t specifies the template and -o the output file. If the option -o is omitted the output is written to the standard output; if no input file is given the input is read from the standard input. If we add the option -V2 we will find the following output on the screen:

```
18
```

A full list of options can be obtained if the program is invoked with the argument --help. See also Appendix C.
This statistics gives information about the number of operations in each of
the unoptimised expressions (here y1 and z1) and the cost of the program
in the output. From left to right the numbers denote the number of mul-
tiplications/divisions, the number of additions/subtractions, the number of
dot products and the number of function calls. As expected for the Heun
method, the program requires two function calls; since we start from very
simple input expressions there are no big savings in the number of multiplica-
tions. In more complicated examples we will find much larger ratios between
input and output in the first column. The generated C-code can be found in
Listing [3].

5.1.3. Fortran 90 Implementation

We want to implement the program of the previous section in Fortran 90
from the same computer algebra output such that all different methods can be
used simultaneously in one program. On top we need to distinguish between
the two types ‘S’ and ‘T’, since the latter denotes a three-vector in the case of
the Lorenz oscillator. We introduce a kind ‘ki’ for all floating point variables
in use:

integer, parameter :: ki = kind(1.0d0)

The first few lines of the configuration file are modified to accommodate
the differences between C and Fortran 90 (see Listing [4]).

One notices that we have not specified the type representation for ‘S’
and ‘T’. These representations are set in the template file ‘fortran.out’
(Listing [5]) by the [% @with %] statement that encloses the whole file.

The general structure of a [% @with %] statement is as follows.

[% @with name opt1[=value1] opt2[=value2] ...]
body[%
@end @with %]

Within its body, symbols are locally defined depending on the environment
which is chosen by name and the options.

The simplest case is the environment ‘env’ which takes its options literally
as assignments. Hence we define the symbols [%SType%] and [%TType%] to
float integrate(
    float (*f)(float, float),
    float x0,
    float y0,
    float h,
    float& err) {
    float t1;
    float t2;
    float t3;
    float t4;

    t4 = h + x0;
    t1 = f(x0, y0);
    t1 = h * t1;
    t2 = t1 + y0;
    t3 = f(t4, t2);
    t3 = h * t3;
    t3 = 1.0 / 2.0 * t3;
    t3 = t3 + y0;
    t1 = 1.0 / 2.0 * t1;
    t1 = t1 + t3;
    y1 = t1;
    z1 = t2;
    err = abs(y1 - z1);
    return y1;
}

Listing 3: The output produced from the template file ‘c.out’ when applied to the expressions of the Heun method
@language fortran90;
@type T; S;
@coerce
    @int → S = "%.0_ki";
    @int/@int → S = "%.0_ki/%.0_ki";
: :

Listing 4: The modifications made for the file rk/fortran.in The left out part of the file is identical to rk/c.in

[%@with env TType="real(ki), dimension(3)"
SType="real(ki)"
]%
:

function rk[%
    @if output.is.file %][%
        output.file.match=".*method_(*).f90"
        format="%s"][%
    @else %]noname[%
    @end @if %](f, x0, y0, h, err) result(y1)
:
[%@SType %], intent(in) :: x0, h
[%@TType %], intent(in) :: y0
[%@SType %], intent(out) :: err
[%@TType %] :: y1, z1[%
@for symbols match="\\$(\\d+)" format="t%s"][%
[%@select type.name
    @case T%][%@TType %][%
    @case S%][%@SType %][%
    @end @select %] :: [% $ %][%
    @end @for symbols %]
:
[%@end @with env%]

Listing 5: Parts of the template file ‘fortran.out’ to produce Fortran 90 output for the Runge-Kutta example.
denote the Fortran 90 representation of the according data type. Instead of specifying the parameters inside the template one can load the parameters from a separate file by using the option ‘file = ”properties−file”’ with the ‘env’ environment, where ‘properties−file’ must correspond to the name of an existing file.

The second major modification of the template file is in the definition of the symbols. Instead of the variable [% type.repr %] a [% @select %] statement is used to choose the appropriate variable according to the value of [% type.name %], which has either the value ‘T’ or ‘S’.

Finally, this template file also demonstrates how identifiers can be derived from the name of the output file. Since the output might go to the standard output channel rather than a file, depending on the parameters with which haggies was called, we need to place an [% @if %] statement around any block that is using the variable ‘output.file’. Here the if-statement tests if the variable ‘output.is.file’ is set to true, which is only the case if the output is written to a file. The pattern in the option ‘match’ selects the relevant part of the file name. The general syntax of the if-statement is as follows:

[ % @if name opt_1 [=value_1] opt_2 [=value_2] ... %] branch 1[ %
   @elif name_2 opt_21 [=value_21] opt_22 [=value_22] ... %] branch 2[ %
   ....[ %
   @else %] branch n[ %]
@end @if %]

The if-statement may contain any number of elif-branches and at most one else-clause.

The directory contains a make file which generates the file lorenz.exe along with the Fortran 90 files for all implemented Runge-Kutta methods. The program lorenz.f90 implements an adaptive integrator that uses the different Runge-Kutta methods for the time step. In Figure 4 the different methods are compared for a common precision goal.
5.2. Gaussian Quadrature

**Synopsis:** Obtaining weights and roots for Gaussian quadrature with non-standard integration kernels

**Objective:** output format for user-defined types in languages without operator overloading

**Requirements:** Form, Java compiler

**Directory:** demo/op

5.2.1. Overview

Gaussian quadrature describes the systematic approximation of an integral by a finite sum. In this example we restrict the formulae to the integration interval between zero and one as in

$$
\int_0^1 dx \, \omega(x)f(x) \approx \sum_{i=1}^N \omega_i f(x_i),
$$

(20)

where $\omega(x)$ denotes a distribution and $f(x)$ a function which is regular in the integration region. The extension to distributions is motivated by the regularisation of infrared singularities in gauge theories. The singularity
structure of scattering amplitudes at higher orders leads one to integrals of
the form
\[ I_1 = \int_0^1 dx \left( -\log(1-x) \right)_+ f(x), \tag{21} \]
\[ I_2 = \int_0^1 dx \left( \frac{1}{1-x} \right)_+ f(x) \tag{22} \]
and similar integrals, where the plus-distribution \( ()_+ \) denotes the prescription
\[ \int_0^1 dx (g(x))_+ f(x) \equiv \int_0^1 dx g(x) (f(x) - f(1)). \tag{23} \]

Since the function \( f(x) \) itself is related to a cross-section and needs to be eval-
uated through Monte-Carlo integration a typical strategy is the evaluation of
the \( x \)-integral through Monte-Carlo techniques which requires the approxima-
tion of the distributions \( \omega(x) \) by functions that are suitable for Monte-Carlo
integration. These technical difficulties could be avoided if the \( x \)-integration
is carried out by a quadrature rule that is adapted to the integration kernel.
A drawback, of course, is that the evaluation of the Monte-Carlo integrals
in \( f(x) \) has to be repeated a fixed number of times at positions \( x_i \); on the
other hand any problems connected to the approximation of the distributions
can be avoided in the case of a Gauss quadrature for the \( x \)-integral and one
expects a better convergence of the Monte-Carlo integrals inside \( f(x_i) \).

The determination of the weights \( \omega_i \) and the roots \( x_i \) in Equation (20)
requires the introduction of polynomials \( p_m(x) \) orthogonal with respect to
the inner product
\[ \langle p_m, p_n \rangle = \int_0^1 dx \omega(x)p_m(x)p_n(x) = \delta_{mn}. \tag{24} \]
The form of these polynomials can be derived using the determinant formula
Here we used the symbols

\[ \sigma_{ij} = \int_0^1 dx \omega_p(x)(1 - x)^{i+j}, \quad \forall i, j \geq 0 \]  

(26)

which for the above kernels are

\[ \sigma_{0,0}^{(1)} = 0 \quad \text{and} \quad \sigma_{i,j}^{(1)} = \frac{1}{(i+j)^2}, \quad i + j > 0 \]  

(27)

\[ \sigma_{0,0}^{(2)} = 0 \quad \text{and} \quad \sigma_{i,j}^{(2)} = \frac{1}{i+j}, \quad i + j > 0 \]  

(28)

The roots \( x_i \) in Equation (20) are the roots of the polynomial \( p_N \) and the weights \( \omega_i \) are defined as

\[ \omega_i = \int_0^1 dx \left( \omega(x) \right) + \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}. \]  

(29)

5.2.2. Implementation

We use the equations from the previous section to generate the expressions for \( p_N(x) \) for different values of \( N \) using a \[23, 24\] program that evaluates the determinant in Equation (25). The program \texttt{haggies} is used to produce a \texttt{Java} program which evaluates \( \omega_i \) and \( x_i \) to arbitrary precision using the class \texttt{BigDecimal} for the numerical operations.

At this point, one of the shortcomings of \texttt{Java} is the lack of operator overloading. Rather than simply writing \( x*y \) one has to use the form \( x\text{.multiply}(y) \) if \( x \) and \( y \) are declared as objects of the class \texttt{BigDecimal}, which in \texttt{Java} implements arbitrary precision numbers.

In \texttt{haggies} this problem is circumvented by using patterns to specify the form of arithmetic operations for each type. The configuration file of the integration example is shown in Listing 6

\[19\]The use of \( (1 - x) \) instead of \( x \) is motivated by the form of \( \omega(x) \).
Listing 6: Configuration file `op.in` used in the Gaussian quadrature example. The symbol 
\[1-x\] as used in `Form` needs to be escaped properly.

```
@language form → c;
@type F = "BigDecimal", "%s.add(%s, mc)", "%s.subtract(%s, mc)", "%s.negate(mc)";
@coerce
@int → F = "number("%s", mc)";
@int/@int → F = "number("%s", "%s", mc)";
@define \[1\−x\] : F = "xbar";
N, P : F;
@operator
F * F → F = "%s.multiply(%s, mc)", "%s.divide(%s, mc)";
@polynomial \[1\−x\];
```

The most obvious change to the previous examples are the additional arguments to the `@type` and `@operator` statements. For the type-statement the arguments from left to right have the meaning

1. type name in the target language; this value can be accessed through the field [% type.repr %] in the template file
2. pattern for binary `+` operator
3. pattern for binary `−` operator
4. pattern for unary `−` operator

The patterns use the same syntax as the `format` method in Java. This means that in order to change the order of the operands one could use the form "%2$s...%1$s" to select the second operand before the first one. Similarly, the arguments to the `@operator` statements are in this order

1. pattern for multiplication
2. pattern for division
3. pattern for inverse division

The third argument becomes relevant if two different types are involved. Consider, for example, the expression \[t/s\] with \[s : S\] and \[t : T\]. This would match the operation \[S*T→R\] although the operands are in reverse order. In
general, we cannot assume that both \( s/t \) and \( t/s \) can be represented by the same pattern in the target language. Hence, \texttt{haggies} provides two different patterns. However, one must take care to use the right order of the arguments in the patterns. The definition

\[
@\text{operator} \quad S \ast T \rightarrow R = "\text{mul}_S \cdot T(\%s, \%s)" , \\
"\text{div}_S \cdot T(\%s, \%s)" , "\text{div}_T \cdot S(\%2$s, \%1$s)" ;
\]

translates \( s/t \) into \( \text{div}_S \cdot T(s, t) \) and \( t/s \) into \( \text{div}_T \cdot S(t, s) \). Omission of the third argument would lead to \( \text{div}_S \cdot T(t, s) \) in the case of \( t/s \).

Another important issue about using the \texttt{BigDecimal} class is that numeric literals cannot be constructed at compile time. The number 4.0/7.0, for example is constructed by the expression

\[
\text{(new BigDecimal("4.0")}. \text{divide(new BigDecimal("7.0"))}
\]

To avoid unnecessary constructor calls, \texttt{haggies} provides the command line option \texttt{-n} which causes the \texttt{CSE} to treat numbers as symbols. Therefore, in the expression \( 2/3x + 2/3y \) with the option \texttt{-n} activated the number \( 2/3 \) is replaced by a symbol and reused in both terms.

As an example result Table 3 shows the roots and weights for \( I_1 \) and \( I_2 \) for \( n = 10 \).

<table>
<thead>
<tr>
<th>( i )</th>
<th>( x_i )</th>
<th>( \omega_i )</th>
<th>( I_1 )</th>
<th>( \omega_i )</th>
<th>( I_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.03426596210338113</td>
<td>0.004059350690970753</td>
<td>0.0144124964045317</td>
<td>0.03968369730743715</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.117456505243587</td>
<td>0.004059350690970753</td>
<td>0.0144124964045317</td>
<td>0.03968369730743715</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.2349228996712536</td>
<td>0.004059350690970753</td>
<td>0.0144124964045317</td>
<td>0.03968369730743715</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.36223839845799</td>
<td>0.004059350690970753</td>
<td>0.0144124964045317</td>
<td>0.03968369730743715</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.4317950092446798</td>
<td>0.004059350690970753</td>
<td>0.0144124964045317</td>
<td>0.03968369730743715</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.48277675979277461</td>
<td>0.004059350690970753</td>
<td>0.0144124964045317</td>
<td>0.03968369730743715</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.5027599682488169</td>
<td>0.004059350690970753</td>
<td>0.0144124964045317</td>
<td>0.03968369730743715</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.5027599682488169</td>
<td>0.004059350690970753</td>
<td>0.0144124964045317</td>
<td>0.03968369730743715</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.5027599682488169</td>
<td>0.004059350690970753</td>
<td>0.0144124964045317</td>
<td>0.03968369730743715</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.5027599682488169</td>
<td>0.004059350690970753</td>
<td>0.0144124964045317</td>
<td>0.03968369730743715</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: The roots \( x_i \) and the weights \( \omega_i \) for the integration kernels \( \omega(x) \) of the integrals \( I_1 \) and \( I_2 \).

\[\text{The long form is \texttt{--cse-on-numbers}}.\]
5.3. Colour Algebra

Synopsis: Calculating the colour correlation matrix for a QCD matrix element in a certain basis

Objective: Initialisation of a matrix

Requirements: Form

Directory: demo/color

5.3.1. Overview

The previous two examples already introduced all important language features present in haggies. This example is taken from the Golem project; it generates the colour correlation matrix for the six-gluon amplitude in QCD. If one assigns the adjoint SU(3) indices $A_1, \ldots, A_6$ to the gluons, a colour basis can be defined by the traces of the form

\[
\begin{align*}
[120] & \quad \text{tr}\{t^{A_{\sigma_1}} \cdots t^{A_{\sigma_6}}\}, \\
[90] & \quad \text{tr}\{t^{A_{\sigma_1}} t^{A_{\sigma_2}}\} \text{tr}\{t^{A_{\sigma_3}} \cdots t^{A_{\sigma_6}}\}, \\
[40] & \quad \text{tr}\{t^{A_{\sigma_1}} t^{A_{\sigma_2}} t^{A_{\sigma_3}}\} \text{tr}\{t^{A_{\sigma_4}} t^{A_{\sigma_5}} t^{A_{\sigma_6}}\}, \\
[15] & \quad \text{tr}\{t^{A_{\sigma_1}} t^{A_{\sigma_2}}\} \text{tr}\{t^{A_{\sigma_3}} t^{A_{\sigma_4}}\} \text{tr}\{t^{A_{\sigma_5}} t^{A_{\sigma_6}}\},
\end{align*}
\]

(30a) (30b) (30c) (30d)

where $\sigma$ in each case are the distinguishable permutations of the indices $\{1, \ldots, 6\}$. The numbers in square brackets denote how many of these permutations exist. Adding up these numbers yields the dimension of the colour space for this process, which is 265. If we denote these basis elements by $|i\rangle$, $i = 1, \ldots, 265$ the contraction of the adjoint indices between a basis element and a conjugate basis element defines an inner product $\langle ij \rangle$. The corresponding Gram matrix is called the colour correlation matrix and plays a role in squaring QCD amplitudes.

5.3.2. Implementation

This example contains a Form program which constructs all $\langle ij \rangle$ as functions in $N_C$, the number of colours, which is 3 in QCD. These entries are stored in variables CC$i$-$j$ in the file color.txt. The constant TR is the normalisation of the generators $\text{tr}\{t^{A} t^{B}\} = TR^A B$. haggies is used to transform this file into a Fortran 90 program to initialise an array $\text{CC}(i, j) = \langle ij \rangle$. Since many of the entries of $\langle ij \rangle$ are zero it would be very inefficient to generate an instruction of the form for each zero entry. A more efficient solution is to initialise the whole array to zero and to modify only the non-zero entries. This is achieved by the following fragment in the file color.out:
There are two points worth mentioning in this template file:

- The symbol ‘is_zero’ inside a ‘instructions’ loop evaluates to true if the right-hand side of the assignment is zero. Hence the corresponding if-statement in the example above only produces code for those assignments with non-zero right-hand side.

- The variables CC_{i,j} are transformed by the match-format pair

  \[
  \text{match} = \"CC_{(d+)}(d+)^{d+}\text{format} = \"CC(%s , %s)\text{end format}\n  \]

  which stores the current values of \(i\) and \(j\) in the two groups \((d+)\) and uses them to fill the two fields \(\%s\).

\[21\] The pattern \(d\) matches a decimal digit.
5.4. Amplitude Calculations

Synopsis: Preparing a Feynman-diagrammatic expression as input for the OPP method

Objective: Using brackets and exploiting possibilities for caching

Requirements: Form, Fortran 90 Compiler

Directory: demo/cut

5.4.1. Overview

In this example we consider one of the main original motivations behind haggies, the evaluation of one-loop scattering amplitudes. We outline a method different from the one implemented in Golem, which suits better as a short example.

A one-loop amplitude\(^{22}\) can be represented in a function basis consisting of scalar Feynman integrals

\[
\mathcal{M}_{\text{virt}} = \sum_{N=1}^{4} \sum_{\alpha} c_{N,\alpha} I_{N}^{\alpha}(S^\alpha) + \mathcal{R},
\]

where the functions \(I_{N}^{\alpha}(S)\) are defined in Equation (4) and the label \(\alpha\) runs over all possible deletions of propagators that lead to the according \(N\)-point topologies. The last term \(\mathcal{R}\) is the so-called rational term which is free of transcendental functions.

As mentioned in the introduction, in the Golem approach one uses recurrence relations to reduce the amplitude to a basis similar to the one in Equation (31)\(^{35}\). An alternative approach is to use the knowledge about the structure of the amplitude to determine the coefficients \(c_{N,\alpha}\), which the OPP method achieves by evaluating the integrand under the \(\int d^n k\) integral for specific values of \(k\)\(^{36}\). One of the drawbacks of this method is the fact that \(\mathcal{R}\) can only be reconstructed partially and another method is required for its full determination. Therefore — and for the sake of simplicity — the example program only deals with the cut constructable parts of the amplitude and leaves out the discussion of \(\mathcal{R}\).

\(^{22}\)For the ease of the argument we consider leg-ordered colour-subamplitudes.
5.4.2. Implementation

In order not to overload the example by physics details, the input expression has been chosen as the numerator of a box diagram from the reaction $u \bar{u} \rightarrow t \bar{t}$, contracted with the tree-level diagram. This contraction ensures that all spinor lines can be expressed as traces, which can be expanded efficiently in Form. The resulting expression is written in terms of Mandelstam variables $s_i = k_i^2$ and $s_{ij} = (k_i + k_j)^2$ and in terms of dot-products involving the integration momentum $q$, such as $q^2$ and $q \cdot k_i$.

The authors of [36] provide a Fortran 90 implementation of their algorithm (CutTools) which is described in [37]. As an input this program requires a function $\text{num}(k, \tilde{k}^2)$ which evaluates the numerator of the amplitude for a specific value of the integration momentum, a complex four-vector $k$, and the value of the $(n - 4)$ dimensional part $\tilde{k}^2$.

The Form program numerator.frm generates an expression for the numerator as described; then we use haggies to write the Fortran 90 routine num in a form where the $k$ dependence is separated from the dependence on $s_i$ and $s_{ij}$, such that large parts of the expression are evaluated only once per phase space point and not for each value of $k$. Since the set of Mandelstam variables depends on the number of external particles the configuration file numerator.in and the according definitions in Fortran 90 are written by the Form program, allowing for different processes to be calculated by just small modifications.

In the input file we focus on the two lines below and their implications for the rest of the program:

```
@brackets [1] q, q2;
@brackets [2] 3000;
```

The first line advises haggies to factor out all occurrences of the vector $q$, which in the implementation represents the integration momentum $k$, and of the symbol $q^2 = k^2$. This statement will also factor out all dot-products involving $q$.

The second line introduces a second level of brackets wherever a subexpression contains more than 3000 terms, which has been chosen arbitrarily and should be changed to much smaller values if the user wants to explore its effect for this simple example. It should be noted that, in order to avoid

---

23 All external momenta $k_i$ are incoming.
unnecessary function calls, **haggies** does not introduce this extra level of brackets if only one bracket would be generated.

In several places of the template file `numerator.out` we find a pair of nested loops as the one shown below

```latex
[% @do brackets prefix="outer." bracket="subex%03d" %]
[% @do brackets prefix="inner.
bracket="subex%03d_%03d" %]

! inner block
[
[% @end @do %]

! outer block
[
[% @end @do %]
```

The two different prefixes have been chosen since the variables of the outer block should also be visible inside the inner block. The option ‘bracket’ determines how the name of the bracket is displayed. For each (nested) bracket **haggies** keeps a path of numbers which uniquely determines the current bracket. The outermost loop corresponds to the command `@brackets[1]` in the configuration file, whereas the nested loop corresponds to the command `@brackets[2]`. The values of ‘outer.$.’ have the values `subex001, subex002` etc; the values of ‘inner.$.’ are `subex001_001, subex001_002` etc where ‘outer.$.’ is `subex001` and `subex002_001` etc for the second entry of the outer loop and so on. This means, the fields are filled left to right from the innermost to outermost bracket level. This behaviour makes it easier to access the innermost counter in languages where local functions can be used for nested brackets.

Inside each brackets-loop the variable ‘type_repr’ denotes the type of the current bracket and the user has also access to the iterators ‘symbols’ and ‘instructions’ which iterate over the local symbols (resp. instructions) of the subprogram which calculates the current bracket. The result of the bracket is stored in the symbol ‘&result’.

In order to propagate the results of the bracket from the inner to the outer loops the function [% expression %] has the argument ‘bracket’, which must be set appropriately if the current expression can refer to a bracket. The declaration part of the function `num` looks as follows in the template file:

```latex
[% @for symbols %][%
    type_repr %] ::= [% $_
    match="\\$\.(.*)" format="r%s" %]
[% @end @for symbols %][%
The first loop introduces local variables for all temporary variables. The second loop declares local, static variables \( b_1, b_2, \ldots \) for the brackets from the first bracket level.

In order to recompute the values of the variables \( b_i \) only when needed the flag \texttt{dirty_cache} is introduced, which is set whenever the kinematics of the process is updated. The following lines set up a conditional block for the recomputation of these symbols.

\[
\text{% @for brackets prefix="outer." bracket="b%d" %}
[% \texttt{outer.type.repr} %], \text{save :: [% \texttt{outer.$_} %] = \&}
& (0.d0, 0.d0)[%
@end @for brackets %]
\]

We use the ‘\texttt{is.first}’ and ‘\texttt{is.last}’ functions to put the if-block in the right place. Inside the loop we need to access the symbols \( b_1, b_2, \ldots \) at the same time as the symbols \texttt{subex001}, \texttt{subex002} etc. Since there is only one ‘\texttt{bracket}’ option to specify a pattern we construct the left-hand sides explicitly by accessing the function ‘\texttt{outer.index}’.

The last part of the subprogram is the loop over the instructions. The main difference to any of the previous examples is the additional option ‘\texttt{bracket}’ in the function ‘\texttt{expression}’ at the right-hand side of the assignments. This option is used to format the brackets in the generated expressions. It should be noted that here the pattern which is used is \texttt{"b%d"} which refers to the cached symbols rather than the functions to calculate the brackets.

\[
q2 = \text{dotproduct}(q, q)[%
\]

\footnotetext{24}{This also ensures that no empty if-statement is generated, if no brackets are present}
The subroutines for the brackets are structured in a similar way with one main difference: the caching is applied only at the outermost level since in this problem there are only two sets of symbols with respect to their update-frequency, the integration momentum $k$ which is different for each call and the Mandelstam variables which only change for each new external kinematics. However, for other problems it might well make sense to introduce multiple levels of caching, for example in multidimensional parameter scans.

5.4.3. Parallelisation

In the file `parallel.out` we introduced the necessary OpenMP [38] directives which are necessary to parallelise the computation of the brackets. The only lines which are different from the above code are inside the initialisation block for the variables $b_i$:

```plaintext
[@for brackets prefix="outer." bracket="subex%03d" %]
  [@if outer.is_first %] if (dirty_cache) then
    !$OMP PARALLEL SECTIONS[%]
    @end if %
  !$OMP SECTION
  b[% outer.index %] = [% outer.$_ %]() %
  @if outer.is_last %]
  !$OMP END PARALLEL SECTIONS
end if[%]
[@end @for brackets %]
```

The inserted directives create a parallel section for each of the function calls inside the if-statement. The actual performance gain, of course, depends very much on the problem. For simple cases almost always the overhead
of creating the threads is larger than the speedup from the parallel calculation. Depending on the number and the sizes of the brackets one might also consider other than the outermost level to introduce parallelism.

5.5. Interval Constraint Solver

Synopsis: Solving the Broyden-Banded function
Objective: Generating output for a scripting language, working with interval arithmetic
Requirements: Form, Python
Directory: demo/cs

5.5.1. Overview

This last example implements an interval constraint solver. The basic idea here is to determine the zeroes of a set of functions

\[ \vec{x} \text{ such that } f_i(\vec{x}) = 0, \quad \forall i = 1, \ldots, n \quad (32) \]

by a so-called branch and prune algorithm [39].

The program uses interval arithmetic to map a box \( B \subset \mathbb{R}^n \) onto a box \( f(B) \); if \( \vec{0} \in f(B) \) the box \( B \) is divided into halves along one of the dimensions, \( B_1 \cup B_2 = B \), and the algorithm is applied to both of the daughter boxes. If \( \vec{0} \not\in B \) the box is discarded. The algorithm stops if each of the boxes has a small enough volume and is considered a solution, which must be checked by another method, since from \( \vec{0} \in f(B) \) one cannot conclude \( \exists \vec{x} \in B : f(\vec{x}) = \vec{0} \) if \( f(B) \) is calculated by the rules of interval arithmetic.

We implement the method for the Broyden banded function, a typical benchmark for optimisation software [40]. This set of functions is defined as

\[ f_i(x_1, \ldots, x_n) = x_i(2 + 5x_i^2) + 1 - \sum_{j \in J_i} x_j(x_j + 1) \quad (33) \]

where \( J_i = \{ j : j \neq i, \max(1, i - 5) \leq j \leq \min(n, i + 1) \} \).

5.5.2. Implementation

A simple Form program generates the set of equation for a given \( n \). The output is read by haggies and transformed into a Python program. The interval arithmetic is implemented by the package mpmath by Fredrick Johansson [25]. As in the first example, we specify the option \(-n\) to haggies at

the command line to avoid multiple generation of interval objects of con-
tstants. In the template file the assignment to the variables \( f_i \) is replaced by

```python
if 0 not in [% expression ... %]:
    return False
```

This block returns from the function and yields \texttt{False} if any of the expressions can be proved not to contain zero.
6. Conclusion

In this article we have reported on haggies, an open source program for the generation of optimised code for the efficient numerical evaluation of mathematical expressions. We have put an emphasis on the generality of the code generator, restricting the set of possible output formats as little as possible. Part of this effort is the introduction of a type inference system which allows the user to combine different data types to represent various algebraical objects. The example programs demonstrate that the presence of such a type inference system provides the capability of handling situations which are difficult to cope with using traditional code generators. On the one hand, it allows to declare temporary variables with the proper type in statically typed languages without the use of implicit typing\textsuperscript{26}. On the other hand, the type system makes it easy to specify rules for translating the algebraic operators into function and method calls, which is required when working with non-standard types in languages without operator overloading, such as Java.

The program haggies transforms the input expressions by a multivariate Horner scheme reducing the number of multiplications and performs a Common Subexpression Elimination on the resulting expression. The careful choice of algorithms in the code generator guarantees that the time for the code transformation scales linearly with the size of the input expression. This enables us to transform and compile relatively large expressions: we currently use haggies successfully inside Golem, a package for generating code for one-loop corrections to scattering processes in quantum field theories. In this application haggies deals with expressions of up to several 10MB consisting of up to $O(10^5)$ terms.

We have implemented a notion of bracketing at an arbitrary number of levels to factor out certain symbols treating each bracket in a separate subprogram. We find this feature useful in three main applications: first of all these brackets allow to split an expression into smaller, independent subprograms if the resulting program is too large for compilation. A second advantage is the possibility of factoring out symbols that change more frequently than others. The (relatively) constant subexpressions can be stored in static variables and are only recomputed if needed. A third aspect is par-

\textsuperscript{26}The keyword implicit in Fortran is a well known source for undetected programming errors.
allelisation as it is very easy to parallelise the subprogram invocations for the different subexpressions.

The features such as the type inference system and the bracketing are mandatory for the use of haggies inside Golem and beneficial for many other applications. Through the separation of the template files from the computer algebra one can produce different programs from the same source without modifying the Computer Algebra System (CAS) code, which also facilitates the integration of haggies with existing projects.

Acknowledgements

The author thanks the organisers of the Les Houches workshop on Physics at TeV Colliders, where part of this work has been completed. The author wants to acknowledge Gudrun Heinrich for careful testing of the program and for stimulating discussion.

A. Input Description

A.1. The @language Statement

The configuration file must contain exactly one @language statement. The exact syntax is

@language input_language → output_language;

Currently, possible choices for ‘input_language’ are ‘mathematica’ and ‘form’. The latter option is suitable for most computer algebra formats. The Mathematica format implements a subset of the Mathematica language and allows for the most commonly used operators. In both cases the file of input expressions consists of a list of assignments with the equals sign (‘=’) as assignment operator and a semicolon terminating each expression.

Possible values for ‘output_language’ are

fortran90 produces output for Fortran 90 and later versions of Fortran.

c is recommended for C-like languages like C/C++, Java. It is also suitable for many scripting languages that do not require continuation characters for continuation lines.

python can be used for scripting languages which use the backslash as continuation character. The exponentiation operator is set to ‘**’ but can be changed in the configuration file.
lisp produces expressions in prefix notation for Lisp.

mathematica produces output suitable for Mathematica.

maple is recommended for Maple.

ada is recommended for Ada.

A.2. The @type Statement

The definition of a new type is introduced by the @type statement. In its simplest form it consists only of the keyword and the type name.

@type T;

Each type that one wants to use needs to be declared exactly once in a configuration file. Optional arguments can be added after an equals-sign to specify the representation of the type in the target language and the representation of the operations ‘a + b’, ‘a − b’ and ‘−a’. Therefore, the following declarations are valid.

@type T1 = "float";
@type T2 = "double []", "add_vectors(%s,%s)", "subtract_vec(%s,%s)", "negate_vec(%s)";

The rightmost arguments can be dropped; however, there are hardly any cases where some but not all of the operations need to be redefined. If no right-hand side is specified the type representation defaults to the type name on the left-hand side.

haggies has two built-in types, @int and @int/@int which are the types of integer and rational literals respectively.

A.3. The @define Statement

All symbols that appear in an expression need to be defined with a defined type. An optional which determines the representation in the target language can be added with an equals sign. On the left-hand side of the definition, a list of patterns is allowed.

@define
    x, y, z: T1;
    arr: I → T2 = "a[%2$s]";
    f, g, h: T1, T1 → T1 = "%s(%s,%s,dummy)";

The arrow \((\rightarrow = \to)\) denotes a function type. Currently, no overloaded or
generic functions can be specified and the number of arguments needs to be
fixed. When defining functions the first wildcard in the pattern contains the
function name and the arguments start with the second wildcard. As the
wildcard syntax follows Java’s syntax for the class \texttt{java.util.Formatter},
one can explicitly access the different arguments by their number (starting
from the index 2): in order to transform \(f(a, b, c)\) into \(a.f(c, b)\) one can use
the pattern "\%2$s.%1$s(%4$s,\%3$s)".

\subsection*{A.4. The \texttt{@coerce} Statement}

In mathematics one often makes use of canonical embeddings of smaller
domains into larger ones, such as \(4 = 4/1\) \((\texttt{Z} \to \texttt{Q})\) or \(1 = 1 + 0i\) \((\texttt{R} \to \texttt{C})\).
In type systems the according concept are implicit conversions or so-called
coercions. In \texttt{haggies} a very primitive implementation of coercions exists:
whenever a type different from the expected is found, \texttt{haggies} scans the list
of defined coercions for an exact match and applies the conversion in case of
a success. The coercions are simple in the sense that if coercions \(c_1 : T_1 \to T_2\)
and \(c_2 : T_2 \to T_3\) are defined, \texttt{haggies} will not automatically apply \(c_2 \circ c_1\)
where \(T_1\) is found and \(T_3\) is expected. Instead, the user has to define a
coercion \(T_1 \to T_3\) explicitly.

Coercions can have an optional pattern which specifies an appropriate
conversion function. Consider the examples below:

\begin{verbatim}
@coerce
    @int \to Int;
    @int \to Float = "%s.0";
    @int/@int \to Float = "%s/%s.0";
    Float \to Polynomial = "(new Polynomial(%s,0))";
\end{verbatim}

If a pattern is used the wildcard contains the value to be converted. A
special case is the type \texttt{@int/@int} where two wildcards are available, one
for the numerator and a second one for the denominator. In the last line
of the example we assume the existence of a class \texttt{Polynomial} with a con-
structor \texttt{Polynomial(float coeff, int power)} that constructs the term
\texttt{coeff} \cdot \texttt{x}^{\texttt{power}}.
A.5. The @operator Statement

Each multiplicative operator including dot-products and powers must be defined by an @operator statement. The @operator statement can have one or more right-hand sides, depending on the operator.

@operator \( T_1 \ast T_2 \rightarrow T_R = P(ab) \), \( P(a/b) \), \( P(b/a) \);

@operator \( T_1 \cdot T_2 \rightarrow T_R = P(u \cdot v) \);
@operator \( T_1 \hat{\ast} T_2 \rightarrow T_R = P(u^v) \);

Here, \( P(\ldots) \) denotes patterns for the according operations. In case of the multiplication they default to "\%s\ast\%s", "\%s/\%s" and "\%2$s/\%1$s" respectively for most languages. The last pattern is used only if \( T_1 \) and \( T_2 \) are different. Care has to be taken because in \( P(b/a) \) the order of the arguments is not reversed automatically.

The default for exponentiation operator is defined according to the target language as "\%s\ast\%s", "\%s\ast\%s" or "pow(\%s,\%s)". For languages without a default dot-product the default for the second line is "\_\_DOT\_\_((\%s,\%s))" to indicate the missing definition in the output.

A.6. The @nullary Statement

The configuration file may contain zero, one or more @nullary statements. This statement declares a list of functions that take zero arguments. In some languages no distinction is made between \( f \) and \( f() \). For optimisation purposes it might, however, be necessary to detect symbols which require a function call and to evaluate them only once. For example, some languages implement the number \( \pi \) as a nullary function call rather than a predefined constant. The declaration

@nullary \( \pi \);

would ensure that the function \( \pi \) is called only once and assigned to a symbol during CSE. The configuration file may contain zero, one or more occurrences of the @nullary statement.

---

27 Powers by integer exponents are implicit if the operator \( \ast T \rightarrow T \) is defined. Addition operators are implicitly defined by the @type statement.
A.7. The \texttt{@polynomial} Statement

The configuration file may contain zero, one or more \texttt{@polynomial} (or short: \texttt{@poly}) statements. The keyword must be followed by a list of symbols, which are factored out during the Horner scheme phase. All other symbols are considered part of the coefficient of the terms. No patterns are allowed in this statement.

The following statements together define the symbols $s_1$, $s_2$ and $s_3$ as symbols in the polynomial part of an expression.

\begin{verbatim}
@polynomial s1, s2;
@poly s3;
\end{verbatim}

A.8. The \texttt{@brackets} Statement

There are two forms of bracketing in \texttt{haggies}. The first form has the syntax

\begin{verbatim}
@brackets number;
\end{verbatim}

where \texttt{number} must be a positive integer literal. This statement ensures that no more than \texttt{number} terms are grouped together and calculated in a subprogram. In the template file the loop \texttt{[% @for brackets %]} can be used to enumerate all brackets generated by this statement.

In the second form of the \texttt{@brackets} keyword is followed by a list of patterns. Those patterns are factored out and the remaining factors are collected inside brackets and computed in a subprogram. If a pattern appears inside parenthesis it matches any factor that contains a symbol matching this pattern; in particular, factors are matched that contain this symbol inside function arguments or in denominators. Without parenthesis patterns are not recognised inside function arguments.

An example of a valid brackets statement is the following.

\begin{verbatim}
@brackets a, b . . .;
@brackets f, (x), y, ( . . . b);
\end{verbatim}

With this declaration, \texttt{haggies} factors out the symbols ‘a’ and ‘y’ and all symbols that start with the letter ‘b’. It also factors out all occurrences of the function \texttt{f}\footnote{assuming \texttt{f} was declared as a function} and all factors that contain either directly or as part of a subexpression the symbol \texttt{x} or symbols ending with \texttt{b}.
Brackets can be nested arbitrarily. The level is indicated by a number in square brackets following directly after the initial keyword. The following example defines three levels of brackets.

```plaintext
@brackets [1] s12, s23;
@brackets [10] 10000;
@brackets [5] (x1), (x2), Log;
@brackets [1] m...;
```

At the first level the variables s12, s23 and all symbols starting with an m are selected. The second level of brackets factors out all occurrences of x1 and x2 and all logarithms. The innermost level ensures that no more than 10,000 terms are computed in a subprogram.

It should be noted that neither the order of the @brackets statements nor the value of the bracket level is important but only the relative ordering of the levels (1 < 5 < 10). The two forms of bracketing (by number or by pattern) cannot be mixed at the same level of nesting. Nested brackets can be accessed through nested [% @for brackets %] loops in the template file.

As a word of warning we like to add that excessive bracketing can decrease the degree of optimisation as no optimisation can be applied across the different brackets.

B. Templates

B.1. Markup Structure

The structure of the template files is closely related to other markup languages such as XML. However, since ‘⟨’ and ‘⟩’ are already heavily used in most target languages their use for indication of markup tags would lead to a proliferation of their escaped representation. Since this is true for virtually any character we use the two letter combinations [% ... %].

The first word inside the brackets must be either a keyword preceded by an ‘@’ sign or the name of a predefined function. The remainder of the tag consists of either single words indicating flags or of options of the form key="value" or key=value (with or without quotes). Quoted strings can contain the usual backslash-escape sequences. In addition, the sequence ”\N” creates a line separator according to the operating system standard.

---

29 e.g. &lt; and &gt; in XML
30 In fact the Java property line.separator is used.
Inside a tag, a comment is indicated by a single quote (') and terminates with the end of the tag.

B.2. Control Structures

Control structures always start with a keyword; the corresponding end-tag has the form [% @end @keyword %]. Arguments in the end-tag are always ignored and can be used as commentary.

B.2.1. Conditional Branching

There are two control structures that can be used for the selection of conditional branches in a template file. The if–elif–else structure and the select–case–else block.

The select–case–else block requires at least one case-branch; the else-branch is optional. The first case-branch is inside the select-tag.

```
[% @select name opt1[=value1] opt2[=value2] ... 
  @case value11 value12 ... 
    %]branch 1[%
  @case value21 value22 ... 
    %]branch 2[%
  ;
  @else
    %]branch n[%
@end @select %]
```

Its semantics is as follows: The expression in the first tag is evaluated with its semantic as a function. The result is compared to the values in the first case-tag. If one of the values matches the result the according branch is evaluated and evaluation continues after the select–case–else block. Otherwise the following case-branches are tested until either one of the tests succeeds or the optional else-branch is reached. Unlike the switch-statement in C-like languages, the select–case–else block ensures that at most one branch is evaluated. If no else-branch is present and all tests fail the block evaluates to an empty string. It should be noted that the values in the case-branches are not evaluated further, i.e. are not interpreted as variable or function names but taken as literal values.

The if–elif–else structure contains at least one branch (here: branch 1). All elif-branches and the else-branch are optional.
The expression in the if-branch is evaluated and the result converted to a truth-value. If the result of the evaluation is true the according branch is evaluated; otherwise the elif-branches are processed in this way until the first elif-branch returns true. If no test in the elif-branches succeeded the optional else-branch is evaluated. If no else-branch is present and all tests fail this structure evaluates to the empty string.

B.2.2. Repetition

Repeated evaluation of a block can be achieved with the for-loop.

The name must be the name of a predefined iterator (See Section B.3). For each element of the iterator a set of variables (depending on the iterator) is set to the according values and made visible within the loop body. The loop body is evaluated and appended to the output for each iteration.

B.2.3. Scopes

The with-statement allows to retrieve a set of variables from a source and to make them visible within its scope.

The name must be the name of a predefined environment (See Section B.4). A set of variables is set to the according values and made visible within the scope of the with statement.
B.3. Predefined Iterators

B.3.1. repeat

Repeats the loop body a given number of times. The options for this iterator are:

from (default=1) start value

to end value

by (default=1) increment

shift (default=0) value added to the counter when assigned to ‘$.’

prefix (optional) a common prefix to all variable names declared by this iterator.

Inside the loop the following variables are defined:

is_first Boolean value which indicates if this is the first iteration of the loop

is_last Boolean value which indicates if this is the last iteration of the loop

$ the current value of the counter

B.3.2. brackets

Iterates over all brackets; can be nested to access nested levels of brackets. The options for this iterator are:

prefix (optional) a common prefix to all variables defined by this iterator.

only (optional) expects a type name as value. If present, only brackets of the given return type are enumerated.

bracket printf format which is used to format the bracket symbol. For each level of nesting an extra format field can be used. The last format field corresponds to the most deeply nested bracket. Hence for the third subbracket of the fifth bracket the string ”%2$d_%1$d” generates ‘5_3’.

result determines how the variable &result is formatted.

Inside the loop the following variables are defined:

is_first Boolean value which indicates if this is the first iteration of the loop
is_last  Boolean value which indicates if this is the last iteration of the loop

$_$  the current bracket symbol formatted by the ‘bracket’ option.

index  the index of the most deeply nested bracket

type.name  the name of the data type of this bracket

type.repr  the representation of the data type of this bracket

B.3.3. instructions
  Enumerates the instructions of the current subprogram in the correct order.
  The options for this iterator are:

prefix  (optional) a common prefix to all variables defined by this iterator.

Inside the loop the following variables are defined:

is_first  Boolean value which indicates if this is the first iteration of the loop

is_last  Boolean value which indicates if this is the last iteration of the loop

is_zero  Boolean value which indicates if the right-hand side is identically zero.

index  the index of this instruction

$_$  the current left-hand side symbol

type.name  the name of the data type of the left-hand side of this assignment

B.3.4. symbols
  Enumerates all temporary symbols of this subroutine.
  The options for this iterator are:

prefix  (optional) a common prefix to all variables defined by this iterator.

only  (optional) expects a type name as value. If present, only symbols of
  the given return type are enumerated.

Inside the loop the following variables are defined:
is_first  Boolean value which indicates if this is the first iteration of the loop

is_last  Boolean value which indicates if this is the last iteration of the loop

$_$  the current symbol

type.name  the name of the data type of this symbol

type.repr  the representation of the data type of this symbol

B.4. Predefined Environments

It should be noted, that wherever it is possible to define new variable names the user must not use ‘path’ and ‘IP’ as names as these are used internally.

B.4.1. env

If the option ‘file’ is specified it must point to an existing file, from which a set of properties is read and assigned to variables. Otherwise all options are interpreted as variable assignments.

B.4.2. args

Provides all command line definitions (-D options) as variables. The option ‘prefix’ can be used to put a common prefix in front of all variable names.

B.4.3. os

Provides all environment variables of the shell or the operating system as variables. The option ‘prefix’ can be used to put a common prefix in front of all variable names.

B.4.4. vm

Provides all system properties of the Java virtual machine as variables. The option ‘prefix’ can be used to put a common prefix in front of all variable names.

B.4.5. eval

Similar to the ‘env’ environment with the difference that the right-hand sides are evaluated as described in Section B.8.
B.5. Predefined Functions

B.5.1. time stamp
Returns the current time and date.

format (optional) specifies the output format according to the rules used in java.text.SimpleDateFormat.

locale (optional) specifies a language and/or a country for the formatting of the date, e.g. ‘en_GB’, ‘en_US’, ‘fr’, ‘de_CH’ ...

B.5.2. user name
Returns the name of the current user. This name is determined by the Java property user.name.

B.5.3. eval
Evaluates an expression according to the expression syntax described in Section B.8. The result can be transformed with the same rules as described in Section B.7.

expression the expression to be evaluated

B.5.4. LINE
Returns the current line-number in the output file. This can be used to generate error messages in languages that do not automatically support exception handling.

B.6. Other Commands
The following commands can be used as functions, with the only difference that they cannot appear in the tags of an if- or select-statement.

B.6.1. include
Includes a file specified by its file name. The file is then interpreted as if it was part of the template file. In particular, all tags in the included file are interpreted and all variables defined at the point of the include statement are visible inside the included file.

file name of the file to be included
B.6.2. tab
The tag [% tab %] generates a horizontal tabulator character[^31].

rep (optional) specifies how many tab-characters should be printed.

B.6.3. nl
The tag [% nl %] generates a new line character. Which character(s) are
printed is determined by the Java property line.separator.

rep (optional) specifies how many new-line sequences should be printed.

B.6.4. expression
This function is defined only inside the ‘instruction’ iterator. It generates
a suitable representation for the right-hand side of the current assignment.

match Together with the ‘format’ option this is used to transform the names
of the temporary variables (‘$1’, ‘$2’, ...) The pattern provided in
the option ‘match’ is applied to each symbol in the expression. If the
match fails, the symbol is returned unchanged. If it succeeds the option
‘format’ is applied to the groups of the match.

format a printf format. See option ‘match’ for details.

bracket a printf format used to format bracket names. For more details
see the documentation on the ‘brackets’ iterator.

B.7. Variable Evaluation
If no options are specified, variables evaluate to its current value as it is.
However, there are a couple of transformations that can be applied to any
variable and some of the functions by providing a set of options.

match (optional) a Java regular expression that must match the value of
the variable.

format (optional) a printf format string. If the above regular expression is
present and contains groups[^32] the wildcards are filled with the contents
of the groups. Otherwise the value itself is used.

[^31]: i.e. ‘\t’ = ASCII 0x09
[^32]: Groups are regular expressions in parenthesis.
convert (optional) if present, a conversion is applied to the value before it is used in the format string. Possible values are

**upper** convert to upper case letters

**lower** convert to lower case letters

**bool** interprets value as logical value. If the options ‘true’ and ‘false’ are specified their values are used instead; example:

```
[% iszero convert=boolean
    true=ZERO false=NON_ZERO%
```

**number** interprets the value as a number. If the option ‘radix’ is specified the value is assumed to be in the given radix.

**locale** (optional) a description of a country and/or language separated by underscores. This string is specified by the ISO two-letter combinations for countries and languages (e.g. ‘en_US’, ‘de_DE’, ‘fr_CA’ or language only ‘en’, ‘de’, ‘fr’).

**default** (optional) value to be used if a variable is not defined

Since internally all variables are stored as strings, conversion to numbers is required if one wants to apply number format descriptors in the format. An example would be [% helicity format=”%7d” convert=number %].

**B.8. The Built-in Calculator**

The predefined environment and function ‘eval’ uses the following set of operators and functions. Function names must be quoted in single quotes to avoid their interpretation as variable names unless the function name should be retrieved from a variable.

Table 4 collects all implemented operators in their precedence from strong to weak operator binding. Table 5 contains a list of all implemented functions. The term regex refers to a string constant that uses Java regular expression format.

```
Table 4: Operator precedence of the ‘eval’ function
```

---

*Table is continued on the next page.*
Table 4: Operator precedence of the ‘eval’ function (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s[m:n]</code></td>
<td>extract substring (zero-based)</td>
</tr>
<tr>
<td><code>x (...)</code></td>
<td>function call (see extra table)</td>
</tr>
<tr>
<td><code>x!</code></td>
<td>factorial</td>
</tr>
<tr>
<td><code>x!!</code></td>
<td>double factorial</td>
</tr>
<tr>
<td><code>=x</code></td>
<td>test: is not null</td>
</tr>
<tr>
<td><code>#x</code></td>
<td>string length</td>
</tr>
<tr>
<td><code>*x</code></td>
<td>dereference: interpret as variable name</td>
</tr>
<tr>
<td><code>\</code>x`</td>
<td>convert integer to float</td>
</tr>
<tr>
<td><code>!x</code></td>
<td>logical not</td>
</tr>
<tr>
<td>`</td>
<td>x`</td>
</tr>
<tr>
<td><code>&lt;&gt;</code>x`</td>
<td>trim: remove leading and trailing blanks</td>
</tr>
<tr>
<td><code>&lt;&lt;x</code></td>
<td>left trim: remove leading blanks</td>
</tr>
<tr>
<td><code>&gt;&gt;x</code></td>
<td>right trim: remove trailing blanks</td>
</tr>
<tr>
<td><code>x^y</code> or <code>x**y</code></td>
<td>power</td>
</tr>
<tr>
<td><code>s@n</code></td>
<td>remove leftmost n − 1 characters from string</td>
</tr>
<tr>
<td>`s</td>
<td>n`</td>
</tr>
<tr>
<td><code>s#t</code></td>
<td>concatenate strings</td>
</tr>
<tr>
<td><code>s</code>`n`</td>
<td>repeat string n times</td>
</tr>
<tr>
<td><code>+x</code></td>
<td>unary plus</td>
</tr>
<tr>
<td><code>−x</code></td>
<td>unary minus</td>
</tr>
<tr>
<td><code>x*y</code></td>
<td>arithmetic multiplication</td>
</tr>
<tr>
<td><code>x/y</code></td>
<td>arithmetic (true) division</td>
</tr>
<tr>
<td><code>x//y</code></td>
<td>integer division</td>
</tr>
<tr>
<td><code>x%y</code></td>
<td>remainder of integer division</td>
</tr>
<tr>
<td><code>x+y</code></td>
<td>arithmetic sum</td>
</tr>
<tr>
<td><code>x−y</code></td>
<td>arithmetic difference</td>
</tr>
<tr>
<td><code>s&lt;&lt;n</code></td>
<td>align left: append up to n blanks</td>
</tr>
<tr>
<td><code>s&gt;&gt;n</code></td>
<td>align right: prepend up to n blanks</td>
</tr>
<tr>
<td><code>s&lt;&gt;n</code></td>
<td>centre using a n as width</td>
</tr>
<tr>
<td><code>x:y</code></td>
<td>minimum: lesser of two numbers</td>
</tr>
<tr>
<td><code>x:y</code></td>
<td>maximum: greater of two numbers</td>
</tr>
<tr>
<td><code>s=t</code></td>
<td>equality: compare as strings</td>
</tr>
<tr>
<td><code>x==y</code></td>
<td>equality: compare as numbers</td>
</tr>
<tr>
<td><code>x=y!</code></td>
<td>inequality: compare as numbers</td>
</tr>
<tr>
<td><code>x&gt;=y</code></td>
<td>comparison: compare as numbers</td>
</tr>
</tbody>
</table>

Table is continued on the next page.
Table 4: Operator precedence of the ‘eval’ function (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x&amp;&amp;y</td>
<td>logical and</td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>x&gt;&lt;&lt;y</td>
<td>logical exclusive or</td>
</tr>
<tr>
<td>x?s:t</td>
<td>ternary if</td>
</tr>
</tbody>
</table>

Table 5: Functions available in the ‘eval’ function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘cos’(x)</td>
<td>trigonometric function</td>
</tr>
<tr>
<td>‘sin’(x)</td>
<td>trigonometric function</td>
</tr>
<tr>
<td>‘tan’(x)</td>
<td>trigonometric function</td>
</tr>
<tr>
<td>‘acos’(x)</td>
<td>inverse trigonometric function</td>
</tr>
<tr>
<td>‘asin’(x)</td>
<td>inverse trigonometric function</td>
</tr>
<tr>
<td>‘atan’(x)</td>
<td>inverse trigonometric function</td>
</tr>
<tr>
<td>‘atan2’(x,y)</td>
<td>angle (in radians) between point (x, y) and x-axis</td>
</tr>
<tr>
<td>‘exp’(x)</td>
<td>$e^x$</td>
</tr>
<tr>
<td>‘pow’(a,x)</td>
<td>$a^x$</td>
</tr>
<tr>
<td>‘log’(x)</td>
<td>$\log x$</td>
</tr>
<tr>
<td>‘log’(x, a)</td>
<td>$\log_a x$</td>
</tr>
<tr>
<td>‘sqrt’(x)</td>
<td>$\sqrt x$</td>
</tr>
<tr>
<td>‘ceil’(x)</td>
<td>round towards $+\infty$</td>
</tr>
<tr>
<td>‘floor’(x)</td>
<td>round towards $-\infty$</td>
</tr>
<tr>
<td>‘round’(x)</td>
<td>round towards nearest neighbour = $\text{floor}(x+1/2)$</td>
</tr>
<tr>
<td>‘sign’(x)</td>
<td>sign of $x$</td>
</tr>
<tr>
<td>‘random’(x,y)</td>
<td>uniformly distributed pseudo-random number between $x$ and $y$</td>
</tr>
<tr>
<td>‘choose’(a,b,c,...)</td>
<td>pseudo-random selection of one argument</td>
</tr>
<tr>
<td>‘subst’(p,r,s)</td>
<td>substitute all occurrences of regex $p$ in $s$ by $r$</td>
</tr>
<tr>
<td>‘matches’(p,s)</td>
<td>test if $s$ matches regular expression $p$</td>
</tr>
<tr>
<td>‘startswith’(s,t)</td>
<td>test if $s$ starts with prefix $t$</td>
</tr>
<tr>
<td>‘endswith’(s,t)</td>
<td>test if $s$ ends with suffix $t$</td>
</tr>
<tr>
<td>‘contains’(s,t)</td>
<td>test if $s$ contains $t$</td>
</tr>
<tr>
<td>‘pos’(t,s)</td>
<td>first position of $t$ in $s$ (0-based) or $-1$</td>
</tr>
<tr>
<td>‘rpos’(t,s)</td>
<td>last position of $t$ in $s$ (0-based) or $-1$</td>
</tr>
<tr>
<td>‘lc’(s)</td>
<td>convert to lower case</td>
</tr>
<tr>
<td>‘uc’(s)</td>
<td>convert to upper case</td>
</tr>
<tr>
<td>‘words’(s)</td>
<td>count words separated by white-space</td>
</tr>
</tbody>
</table>

*Table is continued on the next page.*
Table 5: Functions available in the ‘eval’ function (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'words'(s,i)</td>
<td>return i-th word of s (first word: i = 1)</td>
</tr>
<tr>
<td>'split'(s,d)</td>
<td>count tokens separated by regex d</td>
</tr>
<tr>
<td>'split'(s,d,i)</td>
<td>return i-th token of s</td>
</tr>
<tr>
<td>'format'(f,x,y,...)</td>
<td>apply printf format sequences in f to x, y, ...</td>
</tr>
<tr>
<td>'escape'(s,f)</td>
<td>escape special characters in s</td>
</tr>
<tr>
<td>'e' ∈ f</td>
<td>use XML/HTML entities</td>
</tr>
<tr>
<td>'d' ∈ f</td>
<td>escape double quotes</td>
</tr>
<tr>
<td>'s' ∈ f</td>
<td>escape single quotes (by doubling ')</td>
</tr>
<tr>
<td>'b' ∈ f</td>
<td>use backslash sequences</td>
</tr>
<tr>
<td>'x' ∈ f</td>
<td>allow ‘\x..’ hexadecimal sequences</td>
</tr>
<tr>
<td>'o' ∈ f</td>
<td>allow octal backslash sequences</td>
</tr>
<tr>
<td>'u' ∈ f</td>
<td>use URL encoding</td>
</tr>
</tbody>
</table>

B.9. Emulating Procedure Calls

The combination of environments with include files allows one the efficient emulation of procedures, i.e. of repeatedly used, parametrised text blocks. Consider the following example: first we write an include file that defines a record in a Pascal-like language.

```
"foo.prc"
(* [%f%] is a record with [%n%] entries *)
type [%f%] = record [% @for repeat to=n %]
  field [%$_.%] : integer ; [%
  @end  @for%]
end ;
```

The second fragment shows its usage.

```
[% @with env f="bar" n="3" %][%
  include file="foo.prc"%][%
  @end with%]
[% @with env f="baz" n="10" %][%
  include file="foo.prc"%][%
  @end with%]
```

B.10. Escaping [%

In the unlikely case that one needs to write [% verbatim in the output we recommend the use of the following structure.

61
The same technique can be used to generate other symbols that are not or not easy to generate in the input file, e.g.

\[
\text{% with env uuml=}"\u00fc" \text{ c t r l L=}"\x0C" \%
\]

would define the Unicode character uuml=‘ü’ and the ASCII code Ctrl+L.

C. Command Line Options

C.1. --help or -h
   Prints the list of available options and exits.

C.2. --version or -v
   Prints the version of the program and exits.

C.3. --verbose[=<value>] or -V [<value>]
   Filters which messages are written to the screen. If the value is ommitted all status messages with timings are printed. The option -V2 restricts the output to the operation counts.

C.4. --stdin or -f
   Adds the standard input to the list of input files. If no input files are given at all, this option is implied.

C.5. --template=<value> or -t <value>
   Selects the template file. This option must not be ommitted.

C.6. --config=<value> or -c <value>
   Selects the configuration file. This option must not be ommitted.

C.7. --output=<value> or -o <value>
   Sets the output file. If this option is not specified the output file is written to the standard output.

C.8. --cse-on-numbers or -n
   Treat numbers as symbols in CSE. This option is recommended if the data type for numeric constants cannot be represented by literals but requires a function or constructor call. If the -n option is specified the program introduces variables for all numeric constants to save constructor calls.
C.9. `--horner=<value> or -H <value>`

Selects a strategy for the horner scheme. The default is the original greedy strategy, which is implemented in the class `SingleCount`. The value must correspond to a class name in the package `haggies.analyser.strategies`. Optionally, one can add arguments for the constructor separated by colons. Please, consult the API documentation for more details. If the class name is `null` no Horner scheme is carried out.

C.10. `--allocator=<value> or -A <value>`

Selects an algorithm for the variable allocation. Possible values are

'C' selects the graph colouring algorithm for variable allocation. This algorithm typically fails for large expressions.

'E' selects the Extended Linear Search strategy. This option is the default.

'S' an alternative implementation of the Extended Linear Search algorithm with reduced memory usage but slightly slower.

C.11. `--no-expand` or `-E`

If this option is set the expansion of terms in parentheses is suppressed. This option should be used if the input expression is provided in a (partially) factorised form which should be kept in the output.

C.12. `--expand` or `-e`

Expansion of parentheses is enforced. This is the default.

D. Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>CAS</td>
<td>Computer Algebra System</td>
</tr>
<tr>
<td>CSE</td>
<td>Common Subexpression Elimination</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>EBNF</td>
<td>Extended Backus-Naur Form</td>
</tr>
<tr>
<td>JAR</td>
<td>Java Archive</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Kit</td>
</tr>
</tbody>
</table>
LHC  Large Hadron Collider

QCD  Quantum Chromodynamics

References


[27] Chaitin, G. J. et al., Computer Languages 6 (1981) 47.


[34] http://java.sun.com/j2se/1.5.0/docs/api/index.html?java/util/regex/Pattern.html


