O'Mega:
An Optimizing Matrix Element Generator

Mauro Moretti*

Dipartimento di Fisica, Università di Ferrara
and INFN, Sezione di Ferrara, Ferrara, Italy

Thorsten Ohl†
Jürgen Reuter‡

Darmstadt University of Technology
Schloßgartenstr. 9, D-64289 Darmstadt, Germany

IKDA 2001/06 LC-TOOL-2001-040 hep-ph/0102195

Abstract

We sketch the architecture of *O'Mega*, a new optimizing compiler for tree amplitudes in quantum field theory, and briefly describe its usage. O'Mega generates the most efficient code currently available for scattering amplitudes for many polarized particles in the Standard Model and its extensions.

*e-mail: `moretti@fe.infn.it`
†e-mail: `ohl@hep.tu-darmstadt.de`
‡e-mail: `reuter@hep.tu-darmstadt.de`

1 Introduction

Current and planned experiments in high energy physics can probe physics in processes with polarized beams and many tagged particles in the final state. The combinatorial explosion of the number of Feynman diagrams contributing to scattering amplitudes for many external particles calls for the development of more compact representations that translate well to efficient and reliable numerical code. In gauge theories, the contributions from individual Feynman diagrams are gauge dependent. Strong numerical cancellations in a redundant representation built from individual Feynman diagrams lead to a loss of numerical precision, stressing further the need for eliminating redundancies.

Due to the large number of processes that have to be studied in order to unleash the potential of modern experiments, the construction of nearly optimal representations must be possible algorithmically on a computer and should not require human ingenuity for each new application.

O'Mega [1, 2, 3] is a compiler for tree-level scattering amplitudes that satisfies these requirements. O'Mega is independent of the target language and can therefore create code in any programming language for which a simple output module has been written. To support a physics model, O'Mega requires as input only the Feynman rules and the relations among coupling constants.

Similar to the earlier numerical approaches [4] and [5], O'Mega reduces the growth in calculational effort from a factorial of the number of particles to an exponential. The symbolic nature of O'Mega, however, increases its flexibility. Indeed, O'Mega can emulate both [4] and [5] and produces code that is empirically at least twice as fast. The detailed description of all algorithms is contained in the extensively commented source code of O'Mega [1].

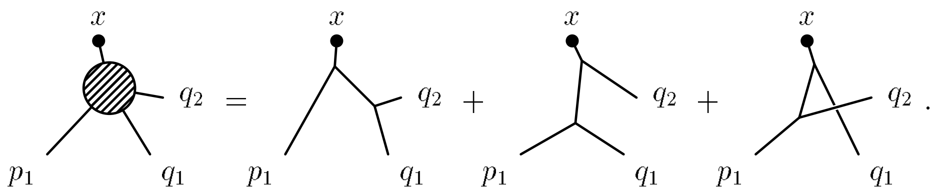
In this note, we sketch the architecture of O'Mega and describe the usage of the first version. The building blocks of the representation of scattering amplitudes generated by O'Mega are described in section 2 and directed acyclical graphs are introduced in section 3. The algorithm for constructing the directed acyclical graph is presented in section 4 and its implementation is described in section 5. We conclude with a few results and examples in section 6. Practical information is presented in the appendices: installation of the O'Mega software in appendix A, running of the O'Mega compiler in appendix B and using O'Mega's output in appendix C. Finally, appendix D briefly discusses mechanisms for extending O'Mega.

2 One Particle Off Shell Wave Functions

One Particle Off-Shell Wave Functions (1POWs) are obtained from connected Greensfunctions by applying the LSZ reduction formula to all but one external line while the remaining line is kept off the mass shell

$$W(x; p_1, \ldots, p_n; q_1, \ldots, q_m) = \\ \langle \phi(q_1), \ldots, \phi(q_m); \mathrm{out} | \Phi(x) | \phi(p_1), \ldots, \phi(p_n); \mathrm{in} \rangle \,. \quad (1)$$

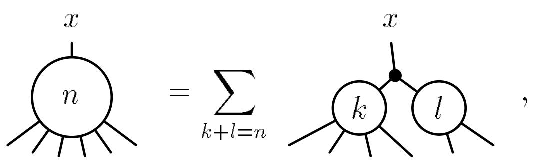
Depending on the context, the off shell line will either be understood as amputated or not. For example, $\langle \phi(q_1), \phi(q_2); \mathrm{out} | \Phi(x) | \phi(p_1); \mathrm{in} \rangle$ in unflavored scalar ϕ^3-theory is given at tree level by



(2)

The number of distinct momenta that can be formed from n external momenta is $P(n) = 2^{n-1} - 1$. Therefore, the number of tree 1POWs grows exponentially with the number of external particles and not with a factorial, as the number of Feynman diagrams, e. g. $F(n) = (2n-5)!! = (2n-5) \cdot \ldots 5 \cdot 3 \cdot 1$ in unflavored ϕ^3-theory.

At tree-level, the set of all 1POWs for a given set of external momenta can be constructed recursively



(3)

where the sum extends over all partitions of the set of n momenta. This recursion will terminate at the external wave functions.

For all quantum field theories, there are—well defined, but not unique—sets of *Keystones* K [1] such that the sum of tree Feynman diagrams for a given process can be expressed as a sparse sum of products of 1POWs without double counting. In a theory with only cubic couplings this is expressed as

$$T = \sum_{i=1}^{F(n)} D_i = \sum_{k,l,m=1}^{P(n)} K^3_{f_k f_l f_m}(p_k, p_l, p_m) W_{f_k}(p_k) W_{f_l}(p_l) W_{f_m}(p_m) \,, \quad (4)$$

with obvious generalizations. The non-trivial problem is to avoide the double counting of diagrams like

 ,

where the circle denotes the keystone. The problem has been solved explicitely for general theories with vertices of arbitrary degrees [1]. The solution is inspired by arguments [4] based on the equations of motion (EOM) of the theory in the presence of sources. The iterative solution of the EOM leads to the construcion of the 1POWs and the constraints imposed on the 1POWs by the EOM suggest the correct set [4] of partitions $\{(p_k, p_l, p_m)\}$ in equation (4).

The maximally symmetric solution selects among equivalent diagrams the keystone closest to the center of a diagram. This corresponds to the numerical expressions of [4]. The absence of double counting can be demonstrated by counting the number $F(d_{\max}, n)$ of unflavored Feynman tree diagrams with n external legs and vertices of maximum degree $d_{\max}$ in to different ways: once directly and then as a sum over keystones. The number $\tilde F(d_{\max}, N_{d,n})$ of unflavored Feynman tree diagrams for one keystone $N_{d,n} = \{n_1, n_2, \ldots, n_d\}$, with $n = n_1 + n_2 + \cdots + n_d$, is given by the product of the number of subtrees and symmetry factors

$$\tilde F(d_{\max}, N_{d,n}) = \frac{n!}{|\mathcal{S}(N_{d,n})|\sigma(n_d, n)} \prod_{i=1}^{d} \frac{F(d_{\max}, n_i + 1)}{n_i!} \tag{5a}$$

where $|\mathcal{S}(N)|$ is the size of the symmetric group of N, $\sigma(n, 2n) = 2$ and $\sigma(n, m) = 1$ otherwise. Indeed, it can be verified that the sum over all keystones reproduces the number

$$F(d_{\max}, n) = \sum_{d=3}^{d_{\max}} \sum_{\substack{N=\{n_1,n_2,\ldots,n_d\} \\ n_1+n_2+\cdots+n_d=n \\ 1\le n_1\le n_2\le\cdots\le n_d\le\lfloor n/2\rfloor}} \tilde F(d_{\max}, N) \tag{5b}$$

of *all* unflavored Feynman tree diagrams.

A second consistent prescription for the construction of keystones is maximally asymmetric and selects the keystone adjacent to a chosen external line. This prescription reproduces the approach in [5] where the tree-level Schwinger-Dyson equations are used as a special case of the EOM.

Recursive algorithms for gauge theory amplitudes have been pioneered in [6]. The use of 1POWs as basic building blocks for the calculation of scattering amplitudes in tree approximation has been advocated in [7] and a heuristic procedure, without reference to keystones, for minimizing the number of arithmetical operations has been suggested. This approach is used by MADGRAPH [8] for fully automated calculations. The heuristic optimizations are quite efficient for $2 \to 4$ processes, but the number of operations remains bounded from below by the number of Feynman diagrams.

2.1 Ward Identities

A particularly convenient property of the 1POWs in gauge theories is that, even for vector particles, the 1POWs are 'almost' physical objects and satisfy simple Ward Identities

$$\frac{\partial}{\partial x_\mu} \langle \mathrm{out} | A_\mu(x) | \mathrm{in} \rangle_{\mathrm{amp.}} = 0 \tag{6a}$$

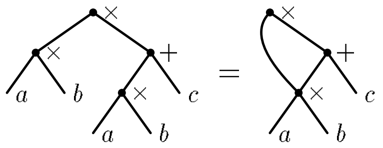
for unbroken gauge theories and

$$\frac{\partial}{\partial x_\mu} \langle \mathrm{out} | W_\mu(x) | \mathrm{in} \rangle_{\mathrm{amp.}} = -m_W \langle \mathrm{out} | \phi_W(x) | \mathrm{in} \rangle_{\mathrm{amp.}} \tag{6b}$$

for spontaneously broken gauge theories in R_ξ-gauge for all physical external states $|in\rangle$ and $|out\rangle$. Thus the identities (6) can serve as powerful numerical checks both for the consistency of a set of Feynman rules and for the numerical stability of the generated code. The code for matrix elements can optionally be instrumented by O'Mega with numerical checks of these Ward identities for intermediate lines.

3 Directed Acyclical Graphs

The algebraic expression for the tree-level scattering amplitude in terms of Feynman diagrams is itself a tree. The much slower growth of the set of 1POWs compared to the set of Feynman diagrams shows that this representation is extremely redundant. In this case, *Directed Acyclical Graphs* (DAGs) provide a more efficient representation, as illustrated by a trivial example

$ab(ab+c) =$  (7)

where one multiplication is saved. The replacement of expression trees by equivalent DAGs is part of the repertoire of optimizing compilers, known as *common subexpression elimination*. Unfortunately, this approach fails in practice for all interesting expressions appearing in quantum field theory, because of the combinatorial growth of space and time required to find an almost optimal factorization.

However, the recursive definition in equation (3) allows to construct the DAG of the 1POWs in equation (4) *directly* [1], without having to construct and factorize the Feynman diagrams explicitely.

As mentioned above, there is more than one consistent prescription for constructing the set of keystones [1]. The symbolic expressions constructed by O'Mega contain the symbolic equivalents of the numerical expressions computed by [4] (maximally symmetric keystones) and [5] (maximally asymmetric keystones) as special cases.

4 Algorithm

By virtue of their recursive construction in Eqs. (3), tree-level 1POWs form a DAG and the problem is to find the smallest DAG that corresponds to a given tree, (i.e. a given sum of Feynman diagrams). O'Mega's algorithm proceeds in four steps

Grow: starting from the external particles, build the tower of *all* 1POWs up to a given height (the height is less than the number of external lines for asymmetric keystones and less than half of that for symmetric keystones) and translate it to the equivalent DAG D.

Select: from D, determine *all* possible *flavored keystones* for the process under consideration and the 1POWs appearing in them.

Harvest: construct a sub-DAG $D^* \subseteq D$ consisting *only* of nodes that contribute to the 1POWs appearing in the flavored keystones.

Calculate: multiply the 1POWs as specified by the keystones and sum the keystones.

By construction, the resulting expression contains no more redundancies and can be translated to a numerical expression. In general, asymmetric keystones create an expression that is smaller by a few percent than the result from symmetric keystones, but it is not yet clear which approach produces the numerically more robust results.

The details of this algorithm as implemented in O'Mega are described in the source code [1]. The persistent data structures [9] used for the determination of D^* are very efficient so that the generation of, e. g. Fortran code for amplitudes in the Standard Model is always much faster than the subsequent compilation.

5 Implementation

The O'Mega compiler is implemented in O'Caml [10], a functional programming language of the ML family with a very efficient, portable and freely available implementation, that can be bootstrapped on all modern computers in a few minutes. The library modules built on experience from [11, 12].

The powerful module system of O'Caml allows an efficient and concise implementation of the DAGs for a specific physics model as a functor application [1]. This functor maps from the category of trees to the category of DAGs and is applied to the set of trees defined by the Feynman rules of any model under consideration.

The module system of O'Caml has been used to make the combinatorial core of O'Mega demonstrably independent from the specifics of both the physics model and the target language [1], as shown in Figure 1. A Fortran90/95 backend has been realized first, backends for C++ and Java will follow. The complete electroweak Standard Model has been implemented together with anomalous gauge boson couplings. The implementation of interfering color amplitudes is currently being completed.

Many extensions of the Standard Model, most prominently the Minimal Supersymmetric Standard Model (MSSM), contain Majorana fermions. In this case, fermion lines have no canonical orientation and the determination of the relative signs of interfering amplitudes is not trivial. However, the Feynman rules for Majorana fermions and fermion number violating interactions proposed in [13] have been implemented in O'Mega in analogy to the naive Feynman rules for Dirac fermions and both methods are available. Numerical comparisons of amplitudes for Dirac fermions calculated both ways show agreement at a small multiple of the machine precision. Thus, all ingredients for the MSSM are available in O'Mega and the implementation of the complete MSSM Lagrangian is currently under way. Non-minimal gauge models, including left-right symmetric models, can be implemented easily.

As mentioned above, the compilers for the target programming language are the slowest step in the generation of executable code. On the other hand, the execution speed of the code is limited by non-trivial vertex evaluations for vectors and spinors, which need $O(10)$ complex multiplications.

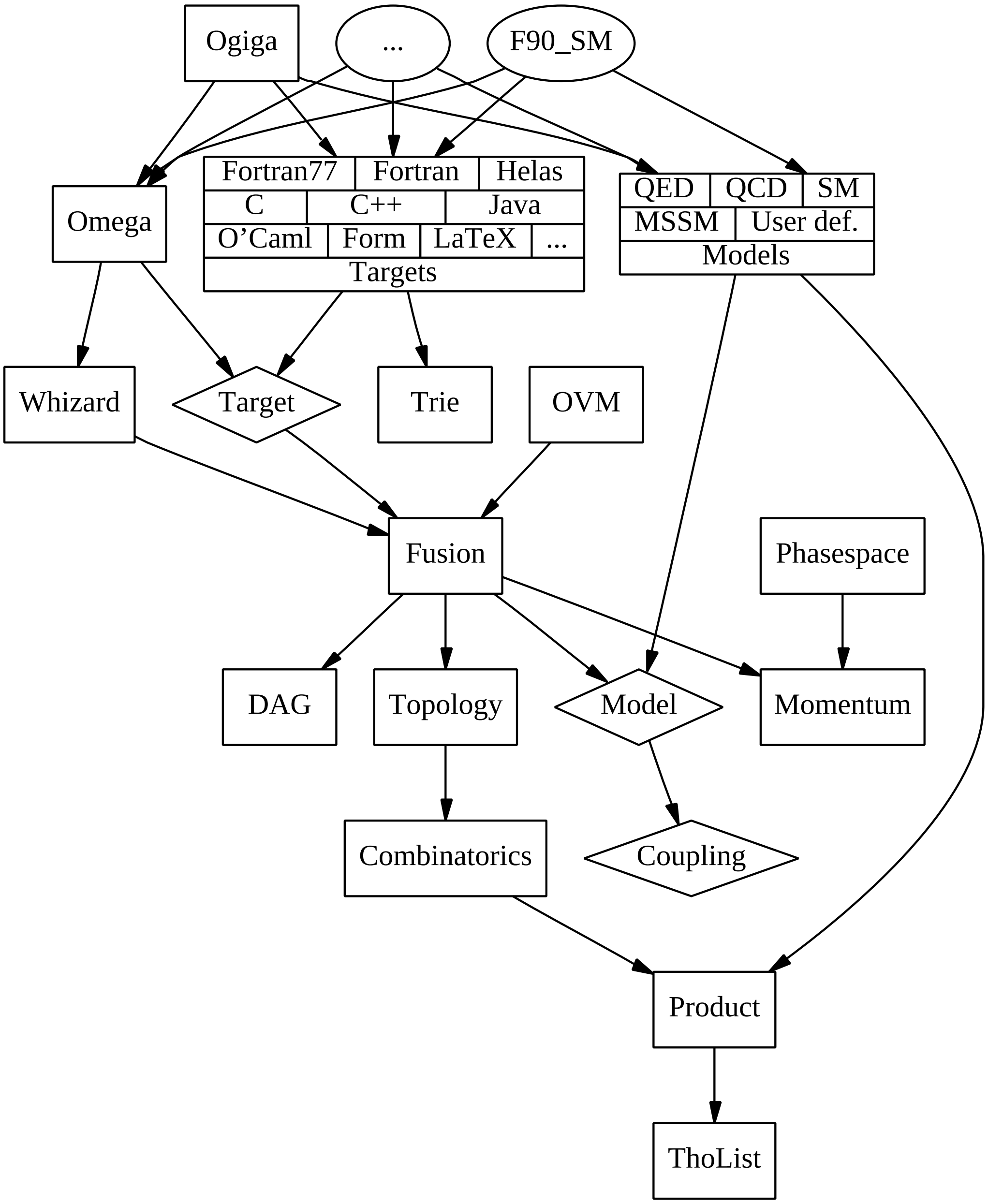


Figure 1: Module dependencies in O'Mega. The diamond shaped nodes denote abstract signatures defining functor domains and co-domains. The rectangular boxes denote modules and functors, while oval boxes stand for example applications.

process	Diagrams		O'Mega	
	#	vertices	#prop.	vertices
$e^+e^- \to e^+\bar\nu_e d\bar u$	20	80	14	45
$e^+e^- \to e^+\bar\nu_e d\bar u\gamma$	146	730	36	157
$e^+e^- \to e^+\bar\nu_e d\bar u\gamma\gamma$	1256	7536	80	462
$e^+e^- \to e^+\bar\nu_e d\bar u\gamma\gamma\gamma$	12420	86940	168	1343
$e^+e^- \to e^+\bar\nu_e d\bar u\gamma\gamma\gamma\gamma$	138816	1110528	344	3933

Table 1: Radiative corrections to four fermion production: comparison of the computational complexity of scattering amplitudes obtained from Feynman diagrams and from O'Mega. (The counts correspond to the full Standard Model—sans light fermion Yukawa couplings—in unitarity gauge with quartic couplings emulated by cubic couplings of non-propagating auxiliary fields.)

process	Diagrams		O'Mega	
	#	vertices	#prop.	vertices
$e^+e^- \to e^+\bar\nu_e d\bar u b\bar b$	472	2832	49	232
$e^+e^- \to e^+\bar\nu_e d\bar u b\bar b\gamma$	4956	34692	108	722
$e^+e^- \to e^+\bar\nu_e d\bar u b\bar b\gamma\gamma$	58340	466720	226	2212

Table 2: Radiative corrections to six fermion production: comparison of the computational complexity of scattering amplitudes obtained from Feynman diagrams and from O'Mega. (The counts correspond to the full Standard Model—sans light fermion Yukawa couplings—in unitarity gauge with quartic couplings emulated by cubic couplings of non-propagating auxiliary fields.)

Therefore, an *O'Mega Virtual Machine* can challenge native code and avoid compilations.

6 Results

6.1 Examples

Tables 1 and 2 show the reduction in computational complexity for some important processes at a e^+e^--linear collider including radiative corrections.

Using the asymmetric keystones can reduce the number of vertices by some 10 to 20 percent relativ to the quoted numbers for symmetric keystones.

6.2 Comparisons

HELAC's [5] diagnostics report more vertices than O'Mega for identical amplitudes. This ranges from comparable numbers for Standard Model processes with many different flavors to an increase by 50 percent for processes with many identical flavors. Empirically, O'Mega's straight line code is twice as fast as HELAC's DO-loops for identical optimizing Fortran95 compilers (not counting HELAC's initialization phase). Together this results in an improved performance by a factor of two to three.

The numerical efficiency of O'Mega's Fortran95 runtime library is empirically identical to HELAS [7]. Therefore, O'Mega's performance can directly be compared to MADGRAPH's [8] by comparing the number of vertices. For $2 \to 5$-processes in the Standard Model, O'Mega's advantage in performance is about a factor of two and grows from there.

The results have been compared with MADGRAPH [8] for many Standard Model processes and numerical agreement at the level of 10^{-11} has been found with double precision floating point arithmetic.

6.3 Applications

O'Mega generated amplitudes are used in the omnipurpose event generator generator WHIZARD [14]. The first complete experimental study of vector boson scattering in six fermion production for linear collider physics [15] has been facilitated by O'Mega and WHIZARD.

Acknowledgements

We thank to Wolfgang Kilian for providing the WHIZARD environment that turns our numbers into real events with unit weight. Thanks to the ECFA/DESY workshops and their participants for providing a showcase. This research is supported by Bundesministerium für Bildung und Forschung, Germany, (05 HT9RDA) and Deutsche Forschungsgemeinschaft (MA 676/6-1).

Finally, thanks to the Caml and Objective Caml teams at INRIA for the lean and mean implementation of a programming language that does not insult the programmer's intelligence.

References

[1] M. Moretti, T. Ohl, J. Reuter, C. Schwinn, *O'Mega, Version 1.0: An Optimizing Matrix Element Generator*, Long Write Up and User's Manual (in progress), `http://heplix.ikp.physik.tu-darmstadt.de/~ohl/omega/`.

[2] T. Ohl, *O'Mega: An Optimizing Matrix Element Generator*, Proceedings of the *Workshop on Advanced Computing and Analysis Technics in Physics Research,* Fermilab, October 2000, IKDA 2000/30, hep-ph/0011243.

[3] T. Ohl, *O'Mega & WHIZARD: Monte Carlo Event Generator Generation For Future Colliders*, Proceedings of the *Workshop on Physics and Experimentation with Future Linear e^+e^--Colliders (LCWS2000),* Fermilab, October 2000, IKDA 2000/31, hep-ph/0011287.

[4] F. Caravaglios and M. Moretti, Phys. Lett. **B358** (1995) 332 [hep-ph/9507237]. F. Caravaglios, M. Moretti, Z. Phys. **C74** (1997) 291.

[5] A. Kanaki, C. Papadopoulos, DEMO-HEP-2000/01, hep-ph/0002082, February 2000.

[6] F. A. Berends and W. T. Giele, Nucl. Phys. **B306** (1988) 759.

[7] H. Murayama, I. Watanabe, K. Hagiwara, KEK Report 91-11, January 1992.

[8] T. Stelzer, W.F. Long, Comput. Phys. Commun. **81** (1994) 357.

[9] Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1998.

[10] Xavier Leroy, *The Objective Caml System, Release 3.00, Documentation and User's Guide*, Technical Report, INRIA, 2000, `http://pauillac.inria.fr/ocaml/`.

[11] T. Ohl, *Lord of the Rings*, (Computer algebra library for O'Caml, unpublished).

[12] T. Ohl, *Bocages*, (Feynman diagram library for O'Caml, unpublished).

[13] A. Denner, H. Eck, O. Hahn and J. Küblbeck, Phys. Lett. **B291** (1992) 278; Nucl. Phys. **B387** (1992) 467.

[14] W. Kilian, *WHIZARD 1.0: A generic Monte-Carlo integration and event generation package for multi-particle processes*, `http://www-ttp.physik.uni-karlsruhe.de/Progdata/whizard/`, LC-TOOL-2001-039.

[15] R. Chierici, S. Rosati, and M. Kobel, *Strong Electroweak Symmetry Breaking Signals in* WW *Scattering at TESLA*, LC-PHSM-2001-038.

[16] E. E. Boos et al, *CompHEP - a package for evaluation of Feynman diagrams and integration over multi-particle phase space,* hep-ph/9908288.

A Installing O'Mega

A.1 Sources

O'Mega is Free Software and the sources can be obtained from `http://heplix.ikp.physik.tu-darmstadt.de/~ohl/omega/` or from `ftp://heplix.ikp.physik.tu-darmstadt.de/pub/ohl/omega`. The command

```
ohl@thopad:~mc$ zcat omega-current.tar.gz | tar xf -
```

will unpack the sources to the directory `omega`. The subdirectories of `omega` are

`bin` contains executable instances of O'Mega: `f90_SM.bin` (`f90_SM.opt` if the sytem is supported by O'Caml's native code compiler), `f90_QED.bin`, etc.

`doc` contains LaTeX sources of user documentation.

`examples` contains currently no supported examples.

`lib` contains library support for targets (Fortran90/95 modules, etc.).

`src` contains the unabridged and uncensored sources of O'Mega, including comments.

`tests` contains a battery of regression tests. Most tests require Madgraph [8].

`web` contains the 'woven' sources, i. e. a pretty printed version of the source including LaTeX documentation. Weaving the sources requires programs, `ocamlweb` and `noweb`. A complete PostScript file is available from the same place as the O'Mega sources. (It is not required for the end user to read this.)

A.2 Prerequisites

A.2.1 Objective Caml (a. k. a. O'Caml)

You need version 3.00 or higher[1]. You can get it from `http://pauillac.inria.fr/ocaml/`. There are precompiled binaries for some popular systems and complete sources. Building from source is straightforward (just follow the instructions in the file `INSTALL` in the toplevel directory, the defaults are almost always sufficient) and takes $\mathcal{O}(10)$ minutes on a modern desktop system. If available for your system (cf. the file `README` in the toplevel directory), you should build the native code compiler.

A.2.2 GNU `make`

This should be available for any system of practical importance and it makes no sense to waste physicist's time on supporting all incompatible flavors of `make` in existence. GNU `make` is the default on Linux systems and is often available as `gmake` on commercial Unices.

A.2.3 Fortran90/95 Compiler

Not required for compiling or running O'Mega, but Fortran90/95 is currently the only fully supported target.

A.3 Configuration

Before the next step, O'Caml must have been installed. Configuration is performed automatically by testing some system features with the command

```
$ ./configure
```

See

```
$ ./configure --help
```

for additional options. NB: The use of the options `--enable-gui` and `--enable-unsupported` is strongly discouraged. The resulting programs require additional prerequisites and even if you can get them to compile, the results are unpredictable and we will not answer any questions about them. NB: `configure` keeps it's state in `config.cache`. If you want to reconfigure after adding new libraries to your system, you should remove `config.cache` before running `configure`.

[1] O'Mega can probably be made to work with versions 2.0x. Given the simplicity of building O'Caml version 3.00, this is not worth the effort.

A.4 Compilation

The command

```
$ make bin
```

will build the byte code executables. For each pairing of physics model and target language, there will be one executable.

```
$ make opt
```

will build the native code executables if the sytem is supported by O'Caml's native code compiler and it is installed. The command

```
$ make f95
```

will build the Fortran90/95 library and requires, obviously, a Fortran90/95 compiler.

B Running O'Mega

O'Mega is a simple application that takes parameters from the commandline and writes results to the standard output device[2] (diagnostics go to the standard error device). E. g., the UNIX commandline

```
$ ./bin/f90_SM.opt e+ e- e+ nue ubar d > cc20_amplitude.f95
```

will cause O'Mega to write a Fortran95 module containing the Standard Model tree level scattering amplitude for $e^+e^- \to e^+\nu_e\bar{u}d$ to the file `cc20_amplitude.f95`. Particles can be combined with colons. E. g.,

```
$ ./bin/f90_SM.opt ubar:u:dbar:d ubar:u:dbar:d e+:mu+ e-:mu- > dy.f95
```

will cause O'Mega to write a Fortran95 module containing the Standard Model tree level parton scattering amplitudes for all Drell-Yan processes to the file `dy.f95`.

A synopsis of the available options, in particular the particle names, can be requested by giving an illegal option, e. g.:

[2] In the future, other targets than Fortran90/95 might require more than one output file (e. g. source files and header files for `C/C++`). In this case the filenames will be specified by commandline parameters.

```
$ ./bin/f90_SM.opt -?
./bin/f90_SM.opt: unknown option `-?'.
usage: ./bin/f90_SM.opt [options] [e-|nue|u|d|e+|nuebar|ubar|dbar\
   |mu-|numu|c|s|mu+|numubar|cbar|sbar|tau-|nutau|t|b\
   |tau+|nutaubar|tbar|bbar|A|Z|W+|W-|g|H|phi+|phi-|phi0]
  -target:function function name
  -target:90 don't use Fortran95 features that are not in Fortran90
  -target:kind real and complex kind (default: omega_prec)
  -target:width approx. line length
  -target:module module name
  -target:use use module
  -target:whizard include WHIZARD interface
  -model:constant_width use constant width (also in t-channel)
  -model:fudged_width use fudge factor for charge particle width
  -model:custom_width use custom width
  -model:cancel_widths use vanishing width
  -warning: check arguments and print warning on error
  -error: check arguments and terminate on error
  -warning:a check # of input arguments and print warning on error
  -error:a check # of input arguments and terminate on error
  -warning:h check input helicities and print warning on error
  -error:h check input helicities and terminate on error
  -warning:m check input momenta and print warning on error
  -error:m check input momenta and terminate on error
  -warning:g check internal Ward identities and print warning on error
  -error:g check internal Ward identities and terminate on error
  -forest ???
  -revision print revision control information
  -quiet don't print a summary
  -summary print only a summary
  -params print the model parameters
  -poles print the Monte Carlo poles
  -dag print minimal DAG
  -full_dag print complete DAG
  -file read commands from file
```

B.1 General Options

`-warning`: include code that checks the supplied arguments and prints a warning in case of an error.

-warning:a check the number of input arguments (momenta and spins) and print a warning in case of an error.

-warning:h check the values of the input helicities and print a warning in case of an error.

-warning:m check the values of the input momenta and print a warning in case of an error.

-warning:g check internal Ward identities and print a warning in case of an error (not supported yet!).

-error: like **-warning:** but terminates on error.

-error:a like **-warning:a** but terminates on error.

-error:h like **-warning:h** but terminates on error.

-error:m like **-warning:m** but terminates on error.

-error:g like **-warning:g** but terminates on error.

-revision print revision control information

-quiet don't print a summary

-summary print only a summary

-params print the model parameters

-poles print the Monte Carlo poles in a format understood by the WHIZARD program [14].

-dag print the reduced DAG in a format understood by the `dot` program.

-full_dag print the complete DAG in a format understood by the `dot` program.

-file read commands from file

B.2 Model Options

B.2.1 Standard Model

`-model:constant_width` use constant width (also in t-channel)

`-model:fudged_width` use fudge factor for charge particle width

`-model:custom_width` use custom width

`-model:cancel_widths` use vanishing width

B.3 Target Options

B.3.1 Fortran90/95

`-target:function` function name

`-target:90` don't use Fortran95 features that are not in Fortran90

`-target:kind` real and complex kind (default: `omega_prec`)

`-target:width` approx. line length

`-target:module` module name

`-target:use` use module

`-target:whizard` include WHIZARD interface

C Using O'Mega's Output

The structure of the outputfile, the calling convention and the required libraries depends on the target language, of course.

C.1 Fortran90/95

The Fortran95 module written by O'Mega has the following signature

```
module omega_amplitude
```

C.1.1 Libraries

The imported Fortran modules are

omega_kinds defines omega_prec, which can be whatever the Fortran compiler supports. NB: the support libraries have not yet been tuned to give reliable answers for amplitudes with gauge cancellations in single precision.

omega95 defines the vertices for Dirac spinors in the chiral representation and vectors.

omega95_bispinors is an alternative that defines the vertices for Dirac and Majorana spinors in the chiral representation and vectors using the Feynman rules of [13].

omega_parameters defines the coupling constants

```
use omega_kinds
use omega95
use omega_parameters
implicit none
private
```

C.1.2 Summary of Exported Functions

The functions and subroutines experted by the Fortran95 module are

- the scattering amplitude in different flavor bases (arrays of PDG codes or internal numbering):

```
public :: amplitude, amplitude_f, amplitude_1, amplitude_2
```

- the scattering amplitude with heuristics supressing vanishing helicity combinations:

```
public :: amplitude_nonzero, amplitude_f_nonzero, &
  amplitude_1_nonzero, amplitude_2_nonzero
```

- the squared scattering amplitude summed over helicity states

```
public :: spin_sum_sqme, spin_sum_sqme_1, sum_sqme
public :: spin_sum_sqme_nonzero, spin_sum_sqme_1_nonzero, &
  sum_sqme_nonzero
```

- "scattering" a general density matrix

```
public :: scatter, scatter_nonzero
```

- "scattering" a diagonal density matrix

```
public :: scatter_diagonal, scatter_diagonal_nonzero
```

- inquiry and maintenance functions

```
public :: allocate_zero
public :: multiplicities, multiplicities_in, multiplicities_out
public :: number_particles, &
  number_particles_in, number_particles_out
public :: number_spin_states, &
  number_spin_states_in, number_spin_states_out, &
  spin_states, spin_states_in, spin_states_out
public :: number_flavor_states, &
  number_flavor_states_in, number_flavor_states_out, &
  flavor_states, flavor_states_in, flavor_states_out
public :: number_flavor_zeros, &
  number_flavor_zeros_in, number_flavor_zeros_out, &
  flavor_zeros, flavor_zeros_in, flavor_zeros_out
public :: create, reset, destroy
```

C.1.3 Maintenance Functions

They currently do nothing, but are here for WHIZARD's [14] convenience

`create` is called only once at the very beginning.

`reset` is called whenever parameters are changed.

`destroy` is called at most once at the very end.

```
subroutine create ()
end subroutine create
subroutine reset ()
end subroutine reset
subroutine destroy ()
end subroutine destroy
```

Allocate an array of the size used by the heuristic that suppresses vanishing helicity combinations

```
interface allocate_zero
   module procedure allocate_zero_1, allocate_zero_2
end interface
```

for join numbering of in and out states

```
subroutine allocate_zero_1 (zero)
  integer, dimension(:,:), pointer :: zero
end subroutine allocate_zero_index
```

and for separate numbering of in and out states

```
subroutine allocate_zero_2 (zero)
  integer, dimension(:,:,:,:), pointer :: zero
end subroutine allocate_zero_index_inout
```

C.1.4 Inquiry Functions

The total number of particles, the number of incoming particles and the number of outgoing particles:

```
pure function number_particles () result (n)
  integer :: n
end function number_particles
pure function number_particles_in () result (n)
  integer :: n
end function number_particles_in
pure function number_particles_out () result (n)
  integer :: n
end function number_particles_out
```

The spin states of all particles that can give non-zero results and their number. The tables are interpreted as `s(1:,i)` contains the helicities for each particle for the `i`th helicity combination.

```
pure function number_spin_states () result (n)
  integer :: n
end function number_spin_states
pure subroutine spin_states (s)
  integer, dimension(:,:), intent(inout) :: s
end subroutine spin_states
```

The spin states of the incoming particles that can give non-zero results and their number:

```
pure function number_spin_states_in () result (n)
  integer :: n
end function number_spin_states_in
pure subroutine spin_states_in (s)
  integer, dimension(:,:), intent(inout) :: s
end subroutine spin_states_in
```

The spin states of the outgoing particles that can give non-zero results and their number:

```
pure function number_spin_states_out () result (n)
  integer :: n
end function number_spin_states_out
pure subroutine spin_states_out (s)
  integer, dimension(:,:), intent(inout) :: s
end subroutine spin_states_out
```

The flavor combinations of all particles that can give non-zero results and their number. The tables are interpreted as

`f(1:,i)` contains the PDG particle code for each particle for the `i`th helicity combination.

```
pure function number_flavor_states () result (n)
  integer :: n
end function number_flavor_states
pure subroutine flavor_states (f)
  integer, dimension(:,:), intent(inout) :: f
end subroutine flavor_states
```

The flavor combinations of the incoming particles that can give non-zero results and their number.

```
pure function number_flavor_states_in () result (n)
  integer :: n
end function number_flavor_states_in
pure subroutine flavor_states_in (f)
  integer, dimension(:,:), intent(inout) :: f
end subroutine flavor_states_in
```

The flavor combinations of the outgoing particles that can give non-zero results and their number.

```
pure function number_flavor_states_out () result (n)
  integer :: n
end function number_flavor_states_out
pure subroutine flavor_states_out (f)
  integer, dimension(:,:), intent(inout) :: f
end subroutine flavor_states_out
```

The flavor combinations of all particles that always can give a zero result and their number:

```
pure function number_flavor_zeros () result (n)
  integer :: n
end function number_flavor_zeros
pure subroutine flavor_zeros (f)
  integer, dimension(:,:), intent(inout) :: f
end subroutine flavor_zeros
```

The flavor combinations of the incoming particles that always can give a zero result and their number:

```
pure function number_flavor_zeros_in () result (n)
  integer :: n
end function number_flavor_zeros_in
pure subroutine flavor_zeros_in (f)
  integer, dimension(:,:), intent(inout) :: f
end subroutine flavor_zeros_in
```

The flavor combinations of the outgoing particles that always can give a zero result and their number:

```
pure function number_flavor_zeros_out () result (n)
  integer :: n
end function number_flavor_zeros_out
pure subroutine flavor_zeros_out (f)
  integer, dimension(:,:), intent(inout) :: f
end subroutine flavor_zeros_out
```

The same initial and final state can appear more than once in the tensor product and we must avoid double counting.

```
pure subroutine multiplicities (a)
  integer, dimension(:), intent(inout) :: a
end subroutine multiplicities

pure subroutine multiplicities_in (a)
  integer, dimension(:), intent(inout) :: a
end subroutine multiplicities_in

pure subroutine multiplicities_out (a)
  integer, dimension(:), intent(inout) :: a
end subroutine multiplicities_out
```

C.1.5 Amplitude

The function arguments of of the amplitude are

`k(0:3,1:)` are the particle momenta: `k(0:3,1)` and `k(0:3,2)` are the incoming momenta, `k(0:3,3:)` are the outgoing momenta. *All* momenta are the physical momenta, i.e. forward time-like or light-like. The signs of the incoming momenta are flipped *internally*. Unless asked by a commandline parameter, O'Mega will not check the validity of the momenta.

`s(1:)` are the helicities in the same order as the momenta. $s = \pm 1$ signify $s = \pm 1/2$ for fermions. $s = 0$ makes no sense for fermions and massless vector bosons $s = 4$ signifies an unphysical polarization for vector boson that the users are *not* supposed to use. Unless asked by a commandline parameter, O'Mega will not check the validity of the helicities.

`f(1:)` are the PDG particle codes in the same order as the momenta.

```
pure function amplitude (k, s, f) result (amp)
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, dimension(:), intent(in) :: s, f
  complex(kind=omega_prec) :: amp
end function amplitude
```

Identical to `amplitude (k, s, flavors(:,f))`, where `flavors` has been filled by `flavor_states`:

```
pure function amplitude_f (k, s, f) result (amp)
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, dimension(:), intent(in) :: s
  integer, intent(in) :: f
  complex(kind=omega_prec) :: amp
end function amplitude_f
```

Identical to `amplitude (k, spins(:,s), flavors(:,f))`, where `spins` has been filled by `spin_states` and `flavors` has been filled by `flavor_states`:

```
pure function amplitude_1 (k, s, f) result (amp)
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, intent(in) :: s, f
  complex(kind=omega_prec) :: amp
end function amplitude_1
```

Similar to `amplitude_1`, but with separate incoming and outgoing particles:

```
pure function amplitude_2 &
     (k, s_in, f_in, s_out, f_out) result (amp)
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, intent(in) :: s_in, f_in, s_out, f_out
  complex(kind=omega_prec) :: amp
end function amplitude_2
```

The following are subroutines and not functions, since Fortran95 restricts arguments of pure functions to `intent(in)`, but we need to update the counter for vanishing amplitudes.

`zero(1:,1:)` an array containing the number of times a combination of spin index and flavor index yielded a vanishing amplitude. After a certain threshold, these combinations will be skipped. `allocate_zero` will allocate the correct size.

`n` the current event count

```
pure subroutine amplitude_nonzero (amp, k, s, f, zero, n)
  complex(kind=omega_prec), intent(out) :: amp
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, dimension(:), intent(in) :: s, f
  integer, dimension(:,:), intent(inout) :: zero
  integer, intent(in) :: n
end subroutine amplitude_nonzero
```

```
pure subroutine amplitude_1_nonzero (amp, k, s, f, zero, n)
  complex(kind=omega_prec), intent(out) :: amp
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, intent(in) :: s, f
  integer, dimension(:,:), intent(inout) :: zero
  integer, intent(in) :: n
end subroutine amplitude_1_nonzero
```

```
pure subroutine amplitude_f_nonzero &
       (amp, k, s, f, zero, n)
  complex(kind=omega_prec), intent(out) :: amp
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, dimension(:), intent(in) :: s
  integer, intent(in) :: f
  integer, dimension(:,:), intent(inout) :: zero
  integer, intent(in) :: n
end subroutine amplitude_f_nonzero
```

`zero(1:,1:,1:,1:)` an array containing the number of times a combination of incoming and outgoing spin indices and flavor indices yielded a vanishing amplitude. `allocate_zero` will allocate the correct size.

```
pure subroutine amplitude_2_nonzero &
     (amp, k, s_in, f_in, s_out, f_out, zero, n)
  complex(kind=omega_prec), intent(out) :: amp
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, intent(in) :: s_in, f_in, s_out, f_out
  integer, dimension(:,:,:,:), intent(inout) :: zero
  integer, intent(in) :: n
end subroutine amplitude_2_nonzero
```

C.1.6 Summation

The the sums of squared matrix elements, the optional mask smask can be used to sum only a subset of helicities or flavors.

```
pure function spin_sum_sqme (k, f, smask) result (amp2)
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, dimension(:), intent(in) :: f
  logical, dimension(:), intent(in), optional :: smask
  real(kind=omega_prec) :: amp2
end function spin_sum_sqme

pure function spin_sum_sqme_1 (k, f, smask) result (amp2)
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, intent(in) :: f
  logical, dimension(:), intent(in), optional :: smask
  real(kind=omega_prec) :: amp2
end function spin_sum_sqme_1

pure function sum_sqme (k, smask, fmask) result (amp2)
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  logical, dimension(:), intent(in), optional :: smask, fmask
  real(kind=omega_prec) :: amp2
end function sum_sqme

pure subroutine spin_sum_sqme_nonzero (amp2, k, f, zero, n, smask)
  real(kind=omega_prec), intent(out) :: amp2
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, dimension(:), intent(in) :: f
  integer, dimension(:,:), intent(inout) :: zero
  integer, intent(in) :: n
  logical, dimension(:), intent(in), optional :: smask
end subroutine spin_sum_sqme_nonzero

pure subroutine spin_sum_sqme_1_nonzero (amp2, k, f, zero, n, smask)
  real(kind=omega_prec), intent(out) :: amp2
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, intent(in) :: f
  integer, dimension(:,:), intent(inout) :: zero
  integer, intent(in) :: n
  logical, dimension(:), intent(in), optional :: smask
end subroutine spin_sum_sqme_1_nonzero
```

```
pure subroutine sum_sqme_nonzero (amp2, k, zero, n, smask, fmask)
  real(kind=omega_prec), intent(out) :: amp2
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  integer, dimension(:,:), intent(inout) :: zero
  integer, intent(in) :: n
  logical, dimension(:), intent(in), optional :: smask, fmask
end subroutine sum_sqme_masked_nonzero
```

C.1.7 Density Matrix Transforms

There are also utility functions that implement the transformation of density matrices directly

$$\rho \to \rho' = T\rho T^\dagger \tag{8}$$

i. e.

$$\rho'_{ff'} = \sum_{ii'} T_{fi}\rho_{ii'}T^*_{f'i'} \tag{9}$$

and avoid double counting

```
pure subroutine scatter_correlated (k, rho_in, rho_out)
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  complex(kind=omega_prec), dimension(:,:,:,:), &
    intent(in) :: rho_in
  complex(kind=omega_prec), dimension(:,:,:,:), &
    intent(inout) :: rho_out
end subroutine scatter_correlated
```

```
pure subroutine scatter_correlated_nonzero &
       (k, rho_in, rho_out, zero, n)
  real(kind=omega_prec), dimension(0:,:), intent(in) :: k
  complex(kind=omega_prec), dimension(:,:,:,:), &
    intent(in) :: rho_in
  complex(kind=omega_prec), dimension(:,:,:,:), &
    intent(inout) :: rho_out
  integer, dimension(:,:,:,:), intent(inout) :: zero
  integer, intent(in) :: n
end subroutine scatter_correlated_nonzero
```

In no off-diagonal density matrix elements of the initial state are known, the computation can be performed more efficiently:

$$\rho'_f = \sum_i T_{fi}\rho_i T^*_{fi} = \sum_i |T_{fi}|^2 \rho_i \tag{10}$$

```
  pure subroutine scatter_diagonal (k, rho_in, rho_out)
    real(kind=omega_prec), dimension(0:,:), intent(in) :: k
    real(kind=omega_prec), dimension(:,:), intent(in) :: rho_in
    real(kind=omega_prec), dimension(:,:), intent(inout) :: rho_out
  end subroutine scatter_diagonal

  pure subroutine scatter_diagonal_nonzero &
         (k, rho_in, rho_out, zero, n)
    real(kind=omega_prec), dimension(0:,:), intent(in) :: k
    real(kind=omega_prec), dimension(:,:), intent(in) :: rho_in
    real(kind=omega_prec), dimension(:,:), intent(inout) :: rho_out
    integer, dimension(:,:,:,:), intent(inout) :: zero
    integer, intent(in) :: n
  end subroutine scatter_diagonal_nonzero
```

Finis.

```
end module omega_amplitude
```

NB: the name of the module can be changed by a commandline parameter and Fortran95 features like `pure` can be disabled as well.

C.2 FORTRAN77

The preparation of a FORTRAN77 target is straightforward, but tedious and will only be considered if there is sufficient demand and support.

C.3 HELAS

This target for the HELAS library [7] is incomplete and no longer maintained. It was used as an early benchmark for the Fortran90/95 library. No vector boson selfcouplings are supported.

C.4 `C`, `C++` & Java

These targets does not exist yet and we solicit suggestions from `C++` and Java experts on useful calling conventions and suppport libraries that blend well with the HEP environments based on these languages. At least one of the authors believes that Java would be a better choice, but the political momentum behind `C++` might cause an early support for `C++` anyway.

D Extending O'Mega

D.1 Adding A New Physics Model

Currently, this still requires to write O'Caml code. This is not as hard as it might sound, because an inspection of `bin/models.ml` shows that all that is required are some tables of Feynman rules that can easily be written by copying and modifyng an existing example, after consulting with `src/couplings.mli` or the corresponding chapter in the woven source. In fact, having the full power of O'Caml at one's disposal is very helpful for avoiding needless repetition.

Nevertheless, in the near future, there will be some special models that can read model specifications from external files. The first one of its kind will read CompHEP [16] model files. Later there will be a native O'Mega model file format, but it will probably go through some iterations.

D.2 Adding A New Target Language

This will always require to write O'Caml code, which is again not too hard. In addition a library for vertices will be required, unless the target performs complete inlining. NB: an early experiment with inlining Fortran proved to be an almost complete failure on Linux/Intel PCs. The inlined code was huge, absolutely unreadable and only marginally faster. The bulk of the computational cost is always in the vertex evaluations and function calls create in comparison negligible costs. This observation is system dependent, of course, and inlining might be beneficial for other architectures with better floating point performance, after all.