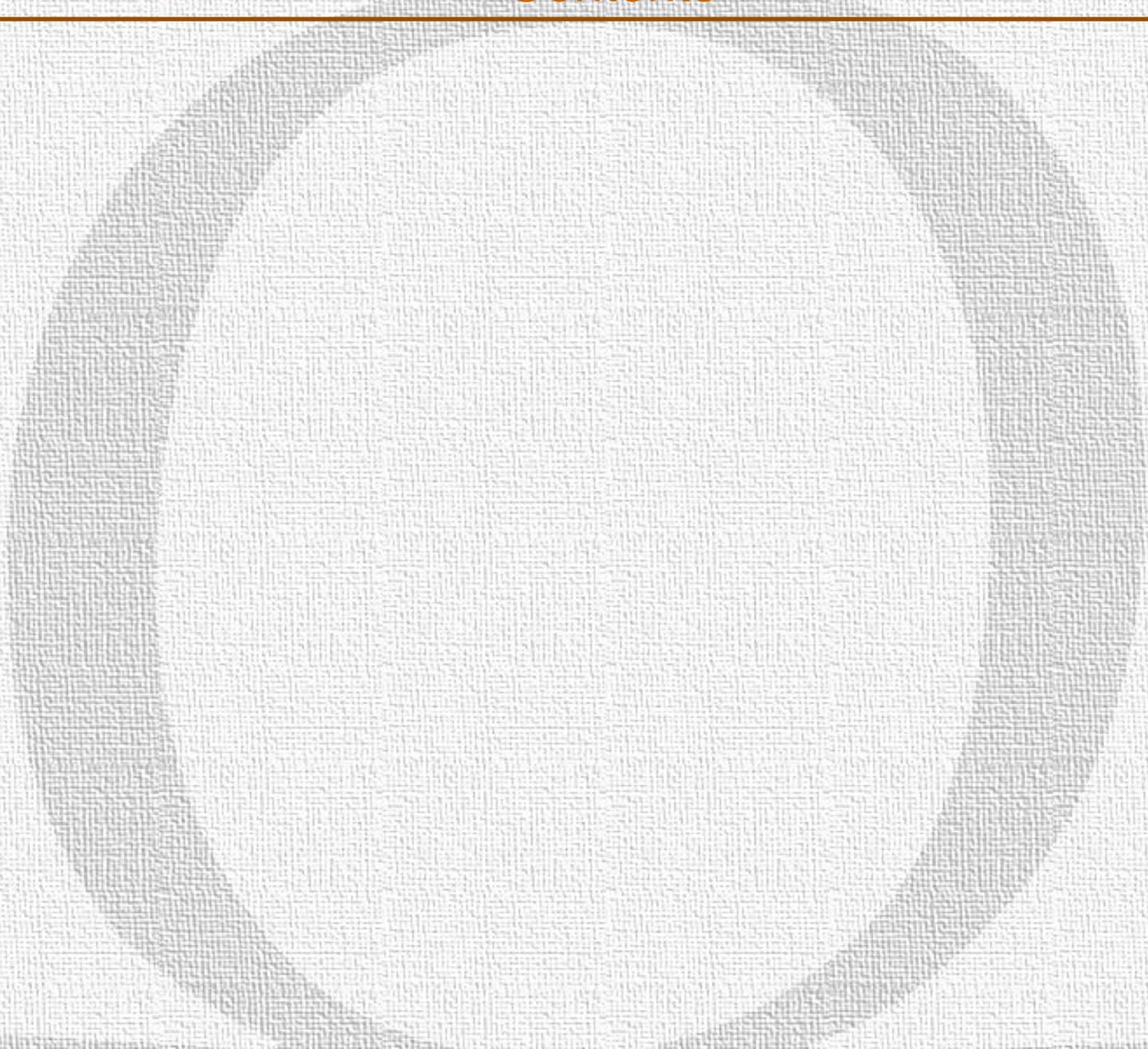




# O'Mega: An Optimizing Matrix Element Generator

Thorsten Ohl  
— TU Darmstadt —  
<ohl@hep.tu-darmstadt.de>

ACAT2000, Fermilab, October 2000





## 1 Feynman Diagrams, DAGs, and Keystones . . . . 2

- Perturbative Complexity ○ One Particle Off Shell Wave Functions
- Keystones ○ Symmetric Keystones ○ Directed Acyclical Graphs
- Algorithm ○ Ward Identities



**1 Feynman Diagrams, DAGs, and Keystones . . . . . 2**

- Perturbative Complexity ○ One Particle Off Shell Wave Functions
- Keystones ○ Symmetric Keystones ○ Directed Acyclical Graphs
- Algorithm ○ Ward Identities

**2 Implementation . . . . . 11**

- Functional Programming ○ The DAG Functor ○ Objective Caml  
(a. k. a. O’Caml) ○ O’Mega ○ Architecture



**1 Feynman Diagrams, DAGs, and Keystones . . . . . 2**

- Perturbative Complexity ◦ One Particle Off Shell Wave Functions
- Keystones ◦ Symmetric Keystones ◦ Directed Acyclical Graphs
- Algorithm ◦ Ward Identities

**2 Implementation . . . . . 11**

- Functional Programming ◦ The DAG Functor ◦ Objective Caml  
(a. k. a. O’Caml) ◦ O’Mega ◦ Architecture

**3 Conclusions . . . . . 18**

- First Results ◦ Outlook



1	Feynman Diagrams, DAGs, and Keystones . . . . .	2
	Perturbative Complexity . . . . .	2
	One Particle Off Shell Wave Functions . . . . .	3
	Keystones . . . . .	4
	Symmetric Keystones . . . . .	5
	Directed Acyclical Graphs . . . . .	7
	Algorithm . . . . .	8
	Ward Identities . . . . .	9
2	Implementation . . . . .	11
3	Conclusions . . . . .	18



The number of tree Feynman diagrams w/  $n$  legs in vanilla  $\phi^3$ -theory is

$$F(n) = (2n - 5)!! = (2n - 5) \cdot (2n - 7) \cdot \dots \cdot 3 \cdot 1$$

$n$	
4	
5	
6	
7	
8	
9	
10	
11	
12	



The number of tree Feynman diagrams w/  $n$  legs in vanilla  $\phi^3$ -theory is

$$F(n) = (2n - 5)!! = (2n - 5) \cdot (2n - 7) \cdot \dots \cdot 3 \cdot 1$$

$n$	$F(n)$
4	3
5	15
6	105
7	945
8	10395
9	135135
10	2027025
11	34459425
12	654729075



The number of tree Feynman diagrams w/  $n$  legs in vanilla  $\phi^3$ -theory is

$$F(n) = (2n - 5)!! = (2n - 5) \cdot (2n - 7) \cdot \dots \cdot 3 \cdot 1$$

$n$	$F(n)$
4	3
5	15
6	105
7	945
8	10395
9	135135
10	2027025
11	34459425
12	654729075

 computational costs grow beyond all reasonable limits



The number of tree Feynman diagrams w/  $n$  legs in vanilla  $\phi^3$ -theory is

$$F(n) = (2n - 5)!! = (2n - 5) \cdot (2n - 7) \cdot \dots \cdot 3 \cdot 1$$

$n$	$F(n)$
4	3
5	15
6	105
7	945
8	10395
9	135135
10	2027025
11	34459425
12	654729075

 computational costs grow beyond all reasonable limits

 gauge theory cancellations cause loss of precision



The number of tree Feynman diagrams w/  $n$  legs in vanilla  $\phi^3$ -theory is

$$F(n) = (2n - 5)!! = (2n - 5) \cdot (2n - 7) \cdot \dots \cdot 3 \cdot 1$$

$n$	$F(n)$
4	3
5	15
6	105
7	945
8	10395
9	135135
10	2027025
11	34459425
12	654729075

☹️ computational costs grow beyond all reasonable limits

☹️ gauge theory cancellations cause loss of precision

Number of independent momenta

$$P(n) = \frac{2^n - 2}{2} - n = 2^{n-1} - n - 1$$



The number of tree Feynman diagrams w/  $n$  legs in vanilla  $\phi^3$ -theory is

$$F(n) = (2n - 5)!! = (2n - 5) \cdot (2n - 7) \cdot \dots \cdot 3 \cdot 1$$

$n$	$F(n)$	$P(n)$
4	3	3
5	15	10
6	105	25
7	945	56
8	10395	119
9	135135	246
10	2027025	501
11	34459425	1012
12	654729075	2035

☹ computational costs grow beyond all reasonable limits

☹ gauge theory cancellations cause loss of precision

Number of independent momenta

$$P(n) = \frac{2^n - 2}{2} - n = 2^{n-1} - n - 1$$



The number of tree Feynman diagrams w/  $n$  legs in vanilla  $\phi^3$ -theory is

$$F(n) = (2n - 5)!! = (2n - 5) \cdot (2n - 7) \cdot \dots \cdot 3 \cdot 1$$

$n$	$F(n)$	$P(n)$
4	3	3
5	15	10
6	105	25
7	945	56
8	10395	119
9	135135	246
10	2027025	501
11	34459425	1012
12	654729075	2035

☹️ computational costs grow beyond all reasonable limits

☹️ gauge theory cancellations cause loss of precision

Number of independent momenta

$$P(n) = \frac{2^n - 2}{2} - n = 2^{n-1} - n - 1$$

$\therefore$  Feynman diagrams **extremely redundant** for many particles in the final state!



The number of tree Feynman diagrams w/ n legs in vanilla  $\phi^3$ -theory is

$$F(n) = (2n - 5)!! = (2n - 5) \cdot (2n - 7) \cdot \dots \cdot 3 \cdot 1$$

n	F(n)	P(n)
4	3	3
5	15	10
6	105	25
7	945	56
8	10395	119
9	135135	246
10	2027025	501
11	34459425	1012
12	654729075	2035

☹️ computational costs grow beyond all reasonable limits

☹️ gauge theory cancellations cause loss of precision

Number of independent momenta

$$P(n) = \frac{2^n - 2}{2} - n = 2^{n-1} - n - 1$$

∴ Feynman diagrams **extremely redundant** for many particles in the final state!

☹️ terms much too large to expect any help from **common subexpression elimination** by optimizing compilers that don't understand any **physics!**



One particle off-shell wave functions (1POWs) are obtained from Greensfunctions by applying the LSZ reduction formula to all but one line:

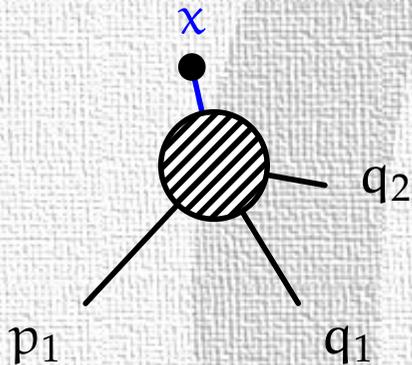
$$W(\mathbf{x}; p_1, \dots, p_n; q_1, \dots, q_m) = \langle \phi(q_1), \dots, \phi(q_m); \text{out} | \Phi(\mathbf{x}) | \phi(p_1), \dots, \phi(p_n); \text{in} \rangle .$$



One particle off-shell wave functions (**1POWs**) are obtained from Greensfunctions by applying the LSZ reduction formula to all but one line:

$$W(\mathbf{x}; p_1, \dots, p_n; q_1, \dots, q_m) = \langle \phi(q_1), \dots, \phi(q_m); \text{out} | \Phi(\mathbf{x}) | \phi(p_1), \dots, \phi(p_n); \text{in} \rangle .$$

E. g.  $\langle \phi(q_1), \phi(q_2); \text{out} | \Phi(\mathbf{x}) | \phi(p_1); \text{in} \rangle$  in unflavored scalar  $\phi^3$ -theory at tree level

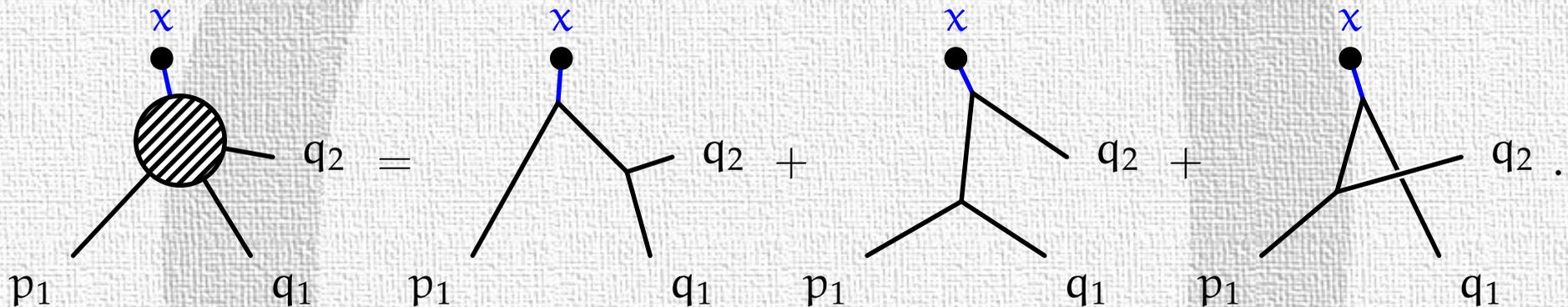




One particle off-shell wave functions (**1POWs**) are obtained from Greensfunctions by applying the LSZ reduction formula to all but one line:

$$W(x; p_1, \dots, p_n; q_1, \dots, q_m) = \langle \phi(q_1), \dots, \phi(q_m); \text{out} | \Phi(x) | \phi(p_1), \dots, \phi(p_n); \text{in} \rangle .$$

E. g.  $\langle \phi(q_1), \phi(q_2); \text{out} | \Phi(x) | \phi(p_1); \text{in} \rangle$  in unflavored scalar  $\phi^3$ -theory at tree level

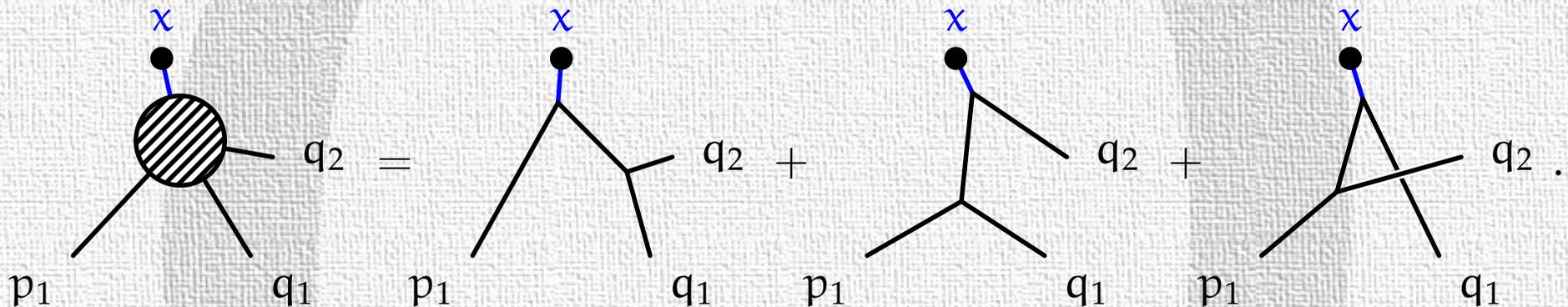




One particle off-shell wave functions (**1POWs**) are obtained from Greensfunctions by applying the LSZ reduction formula to all but one line:

$$W(x; p_1, \dots, p_n; q_1, \dots, q_m) = \langle \phi(q_1), \dots, \phi(q_m); \text{out} | \Phi(x) | \phi(p_1), \dots, \phi(p_n); \text{in} \rangle .$$

E. g.  $\langle \phi(q_1), \phi(q_2); \text{out} | \Phi(x) | \phi(p_1); \text{in} \rangle$  in unflavored scalar  $\phi^3$ -theory at tree level



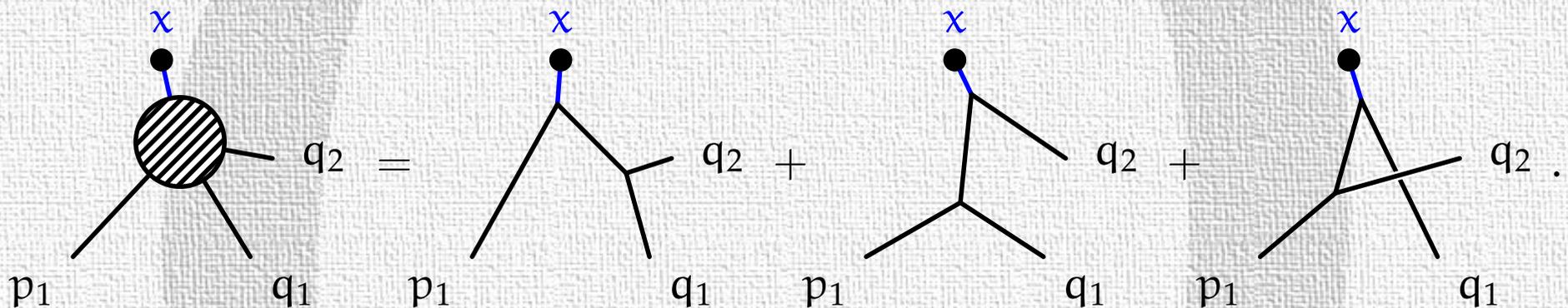
- the set of **all** 1POWs at tree level grows **exponentially** and can be constructed recursively from other 1POWs at tree level (the recursion terminates at the external wave functions).



One particle off-shell wave functions (**1POWs**) are obtained from Greensfunctions by applying the LSZ reduction formula to all but one line:

$$W(x; p_1, \dots, p_n; q_1, \dots, q_m) = \langle \phi(q_1), \dots, \phi(q_m); \text{out} | \Phi(x) | \phi(p_1), \dots, \phi(p_n); \text{in} \rangle .$$

E. g.  $\langle \phi(q_1), \phi(q_2); \text{out} | \Phi(x) | \phi(p_1); \text{in} \rangle$  in unflavored scalar  $\phi^3$ -theory at tree level



- the set of **all** 1POWs at tree level grows **exponentially** and can be constructed recursively from other 1POWs at tree level (the recursion terminates at the external wave functions).

(rhetorical) question: can we find **keystones**  $K$  with

$$T = \sum_{i=1}^{F(n)} D_i = \sum_{k,l,m=1}^{P(n)} K_{f_k f_l f_m}^3(p_k, p_l, p_m) W_{f_k}(p_k) W_{f_l}(p_l) W_{f_m}(p_m) + \text{quartic} + \dots$$



There are two extreme solutions:

1. one maximally **symmetric**
2.  $n$  maximally **asymmetric**



There are two extreme solutions:

1. one maximally **symmetric**
2.  $n$  maximally **asymmetric**

The maximally **asymmetric** solutions are simple: pick any external line  $j$  and make the adjacent vertex a keystone. Equivalently, calculate the 1POW for particle  $j$  and put the amputated line on-shell:

$$T = W_{f_j}(p_j) \Big|_{\text{amp.,on-shell}}$$



There are two extreme solutions:

1. one maximally **symmetric**
2.  $n$  maximally **asymmetric**

The maximally **asymmetric** solutions are simple: pick any external line  $j$  and make the adjacent vertex a keystone. Equivalently, calculate the 1POW for particle  $j$  and put the amputated line on-shell:

$$T = W_{f_j}(p_j) \Big|_{\text{amp.,on-shell}}$$

- symbolic equivalent of the numerical Schwinger-Dyson equations of **HELAC** (cf. talk by **Costas Papadopoulos**)



There are two extreme solutions:

1. one maximally **symmetric**
2.  $n$  maximally **asymmetric**

The maximally **asymmetric** solutions are simple: pick any external line  $j$  and make the adjacent vertex a keystone. Equivalently, calculate the 1POW for particle  $j$  and put the amputated line on-shell:

$$T = W_{f_j}(p_j) \Big|_{\text{amp.,on-shell}}$$

- symbolic equivalent of the numerical Schwinger-Dyson equations of **HELAC** (cf. talk by **Costas Papadopoulos**)

 empirically, a correct implementation will require a **few percent less multiplications** in the amplitude compared to the symmetric solution below



There are two extreme solutions:

1. one maximally **symmetric**
2.  $n$  maximally **asymmetric**

The maximally **asymmetric** solutions are simple: pick any external line  $j$  and make the adjacent vertex a keystone. Equivalently, calculate the 1POW for particle  $j$  and put the amputated line on-shell:

$$T = W_{f_j}(p_j) \Big|_{\text{amp.,on-shell}}$$

- symbolic equivalent of the numerical Schwinger-Dyson equations of **HELAC** (cf. talk by **Costas Papadopoulos**)
- 😊 empirically, a correct implementation will require a **few percent less multiplications** in the amplitude compared to the symmetric solution below
- 😞 numerical problems have a **few more steps to accumulate** compared to the symmetric solution below



There are two extreme solutions:

1. one maximally **symmetric**
2.  $n$  maximally **asymmetric**

The maximally **asymmetric** solutions are simple: pick any external line  $j$  and make the adjacent vertex a keystone. Equivalently, calculate the 1POW for particle  $j$  and put the amputated line on-shell:

$$T = W_{f_j}(p_j) \Big|_{\text{amp.,on-shell}}$$

- symbolic equivalent of the numerical Schwinger-Dyson equations of **HELAC** (cf. talk by **Costas Papadopoulos**)
  - 😊 empirically, a correct implementation will require a **few percent less multiplications** in the amplitude compared to the symmetric solution below
  - 😞 numerical problems have a **few more steps to accumulate** compared to the symmetric solution below
- ∴ more testing with sensitive gauge theory amplitudes required



The maximally **symmetric** solutions (corresponding to the numerical approach of **ALPHA**) are built from balanced **inequivalent partitions** of external momenta:

$n$	$\Sigma$	$\Sigma$
4	4	$1 \cdot (1, 1, 1, 1) + 3 \cdot (1, 1, 2)$
5	26	$1 \cdot (1, 1, 1, 1, 1) + 10 \cdot (1, 1, 1, 2) + 15 \cdot (1, 2, 2)$
6	236	$1 \cdot (1, 1, 1, 1, 1, 1) + 15 \cdot (1, 1, 1, 1, 2) + 40 \cdot (1, 1, 1, 3)$ $+ 45 \cdot (1, 1, 2, 2) + 120 \cdot (1, 2, 3) + 15 \cdot (2, 2, 2)$



The maximally **symmetric** solutions (corresponding to the numerical approach of **ALPHA**) are built from balanced **inequivalent partitions** of external momenta:

$n$	$\Sigma$	$\Sigma$
4	4	$1 \cdot (1, 1, 1, 1) + 3 \cdot (1, 1, 2)$
5	26	$1 \cdot (1, 1, 1, 1, 1) + 10 \cdot (1, 1, 1, 2) + 15 \cdot (1, 2, 2)$
6	236	$1 \cdot (1, 1, 1, 1, 1, 1) + 15 \cdot (1, 1, 1, 1, 2) + 40 \cdot (1, 1, 1, 3)$ $+ 45 \cdot (1, 1, 2, 2) + 120 \cdot (1, 2, 3) + 15 \cdot (2, 2, 2)$

**Subtlety:** partitions for an **even** number of external lines of the type

$$(n_1, n_2, \dots, n_{d-1}, n_1 + n_2 + \dots + \dots + n_{d-1})$$

are two-fold degenerate



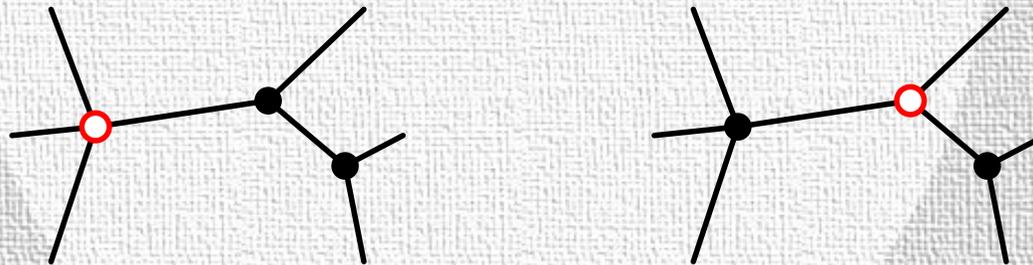
The maximally **symmetric** solutions (corresponding to the numerical approach of **ALPHA**) are built from balanced **inequivalent partitions** of external momenta:

$n$	$\Sigma$	$\Sigma$
4	4	$1 \cdot (1, 1, 1, 1) + 3 \cdot (1, 1, 2)$
5	26	$1 \cdot (1, 1, 1, 1, 1) + 10 \cdot (1, 1, 1, 2) + 15 \cdot (1, 2, 2)$
6	236	$1 \cdot (1, 1, 1, 1, 1, 1) + 15 \cdot (1, 1, 1, 1, 2) + 40 \cdot (1, 1, 1, 3) + 45 \cdot (1, 1, 2, 2) + 120 \cdot (1, 2, 3) + 15 \cdot (2, 2, 2)$

**Subtlety:** partitions for an **even** number of external lines of the type

$$(n_1, n_2, \dots, n_{d-1}, n_1 + n_2 + \dots + \dots + n_{d-1})$$

are two-fold degenerate, e. g.  $(1, 1, 1, 3)$  and  $(1, 2, 3)$  contain the **same** diagram



$\therefore$  choose representatives **consistently** (with a lot of help from **Mauro Moretti**).



$F(d_{\max}, n)$  = # of Feynman diagrams with  $n$  external legs

in unflavored

$$\mathcal{L} = \frac{1}{2} \partial_\mu \phi \partial^\mu \phi - \frac{m^2}{2} \phi^2 + \sum_{d=3}^{d_{\max}} \frac{\lambda_d}{d!} \phi^d$$

theory.



$F(d_{\max}, \mathbf{n})$  = # of Feynman diagrams with  $\mathbf{n}$  external legs

in unflavored

$$\mathcal{L} = \frac{1}{2} \partial_\mu \phi \partial^\mu \phi - \frac{m^2}{2} \phi^2 + \sum_{d=3}^{d_{\max}} \frac{\lambda_d}{d!} \phi^d$$

theory. In a partition  $\mathbf{N}_{d,n} = \{n_1, n_2, \dots, n_d\}$  with  $n = n_1 + n_2 + \dots + n_d$ , there are

$$\tilde{F}(d_{\max}, \mathbf{N}_{d,n}) = \frac{1}{(1 + \delta_{n_d, n_1 + n_2 + \dots + n_{d-1}})} \frac{n!}{|\mathcal{S}(\mathbf{N}_{d,n})|} \prod_{i=1}^d \frac{F(d_{\max}, n_i + 1)}{n_i!}$$

Feynman diagrams ( $|\mathcal{S}(\mathbf{N})|$  the size of the symmetric group of  $\mathbf{N}$ ).



$F(d_{\max}, n)$  = # of Feynman diagrams with  $n$  external legs

in unflavored

$$\mathcal{L} = \frac{1}{2} \partial_\mu \phi \partial^\mu \phi - \frac{m^2}{2} \phi^2 + \sum_{d=3}^{d_{\max}} \frac{\lambda_d}{d!} \phi^d$$

theory. In a partition  $N_{d,n} = \{n_1, n_2, \dots, n_d\}$  with  $n = n_1 + n_2 + \dots + n_d$ , there are

$$\tilde{F}(d_{\max}, N_{d,n}) = \frac{1}{(1 + \delta_{n_d, n_1 + n_2 + \dots + n_{d-1}})} \frac{n!}{|\mathcal{S}(N_{d,n})|} \prod_{i=1}^d \frac{F(d_{\max}, n_i + 1)}{n_i!}$$

Feynman diagrams ( $|\mathcal{S}(N)|$  the size of the symmetric group of  $N$ ). Non trivial cross-check

$$F(d_{\max}, n) = \sum_{d=3}^{d_{\max}} \sum_{\substack{N = \{n_1, n_2, \dots, n_d\} \\ n_1 + n_2 + \dots + n_d = n \\ 1 \leq n_1 \leq n_2 \leq \dots \leq n_d \leq \lfloor n/2 \rfloor}} \tilde{F}(d_{\max}, N)$$



$F(d_{\max}, n)$  = # of Feynman diagrams with  $n$  external legs

in unflavored

$$\mathcal{L} = \frac{1}{2} \partial_\mu \phi \partial^\mu \phi - \frac{m^2}{2} \phi^2 + \sum_{d=3}^{d_{\max}} \frac{\lambda_d}{d!} \phi^d$$

theory. In a partition  $N_{d,n} = \{n_1, n_2, \dots, n_d\}$  with  $n = n_1 + n_2 + \dots + n_d$ , there are

$$\tilde{F}(d_{\max}, N_{d,n}) = \frac{1}{(1 + \delta_{n_d, n_1 + n_2 + \dots + n_{d-1}})} \frac{n!}{|\mathcal{S}(N_{d,n})|} \prod_{i=1}^d \frac{F(d_{\max}, n_i + 1)}{n_i!}$$

Feynman diagrams ( $|\mathcal{S}(N)|$  the size of the symmetric group of  $N$ ). Non trivial cross-check

$$F(d_{\max}, n) = \sum_{d=3}^{d_{\max}} \sum_{\substack{N = \{n_1, n_2, \dots, n_d\} \\ n_1 + n_2 + \dots + n_d = n \\ 1 \leq n_1 \leq n_2 \leq \dots \leq n_d \leq \lfloor n/2 \rfloor}} \tilde{F}(d_{\max}, N)$$

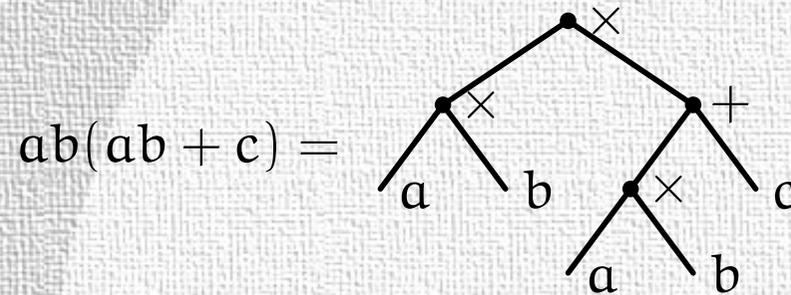
☺ has been checked numerically up to  $n = \mathcal{O}(100)$ .



Directed Acyclical Graphs (DAGs) are a more efficient representation for arithmetical expressions than the equivalent trees.

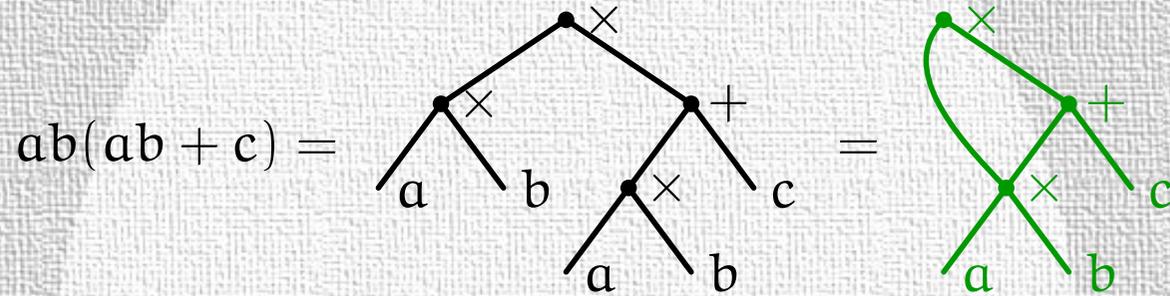


Directed Acyclical Graphs (DAGs) are a more efficient representation for arithmetical expressions than the equivalent trees. E. g.:



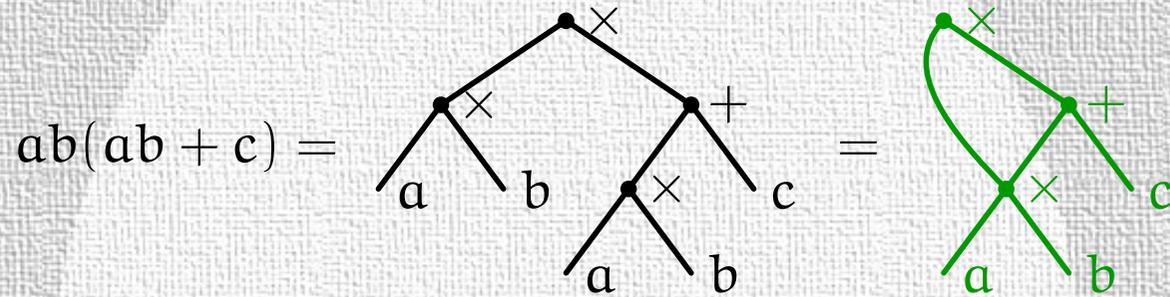


Directed Acyclical Graphs (DAGs) are a more efficient representation for arithmetical expressions than the equivalent trees. E. g.:





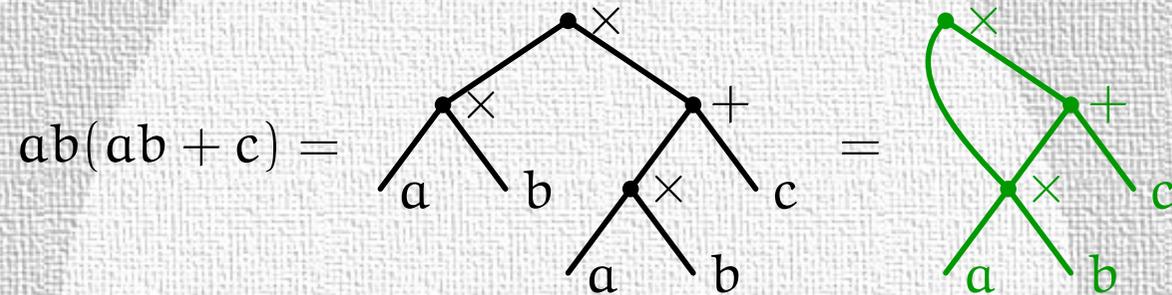
Directed Acyclical Graphs (DAGs) are a more efficient representation for arithmetical expressions than the equivalent trees. E. g.:



- partial order “*depends on*” prohibits cycles



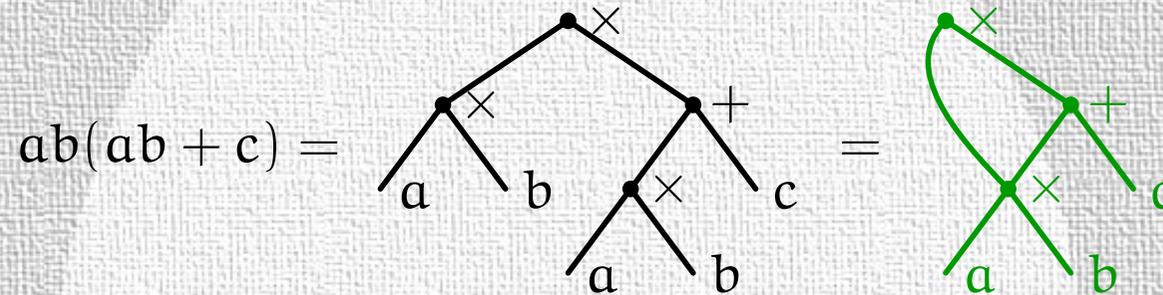
Directed Acyclical Graphs (DAGs) are a more efficient representation for arithmetical expressions than the equivalent trees. E. g.:



- partial order “*depends on*” prohibits cycles
- any tree is equivalent to **at least one** DAG (itself)
- any DAG is equivalent to **only one** tree



Directed Acyclical Graphs (DAGs) are a more efficient representation for arithmetical expressions than the equivalent trees. E. g.:

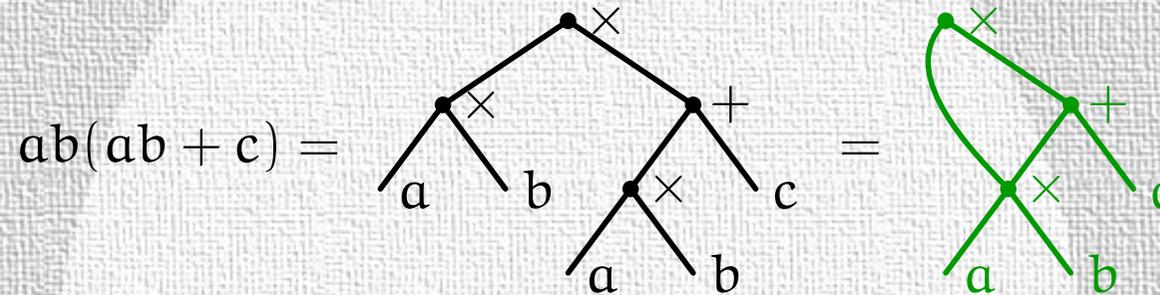


- partial order “*depends on*” prohibits cycles
- any tree is equivalent to **at least one** DAG (itself)
- any DAG is equivalent to **only one** tree

Sets of trees and sets of DAGs can be characterized by the set of possible **leaves** and the possible **branches**



Directed Acyclical Graphs (DAGs) are a more efficient representation for arithmetical expressions than the equivalent trees. E. g.:



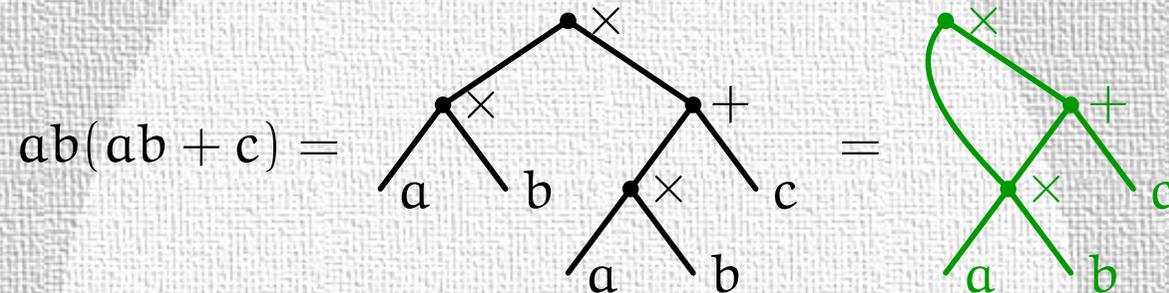
- partial order “depends on” prohibits cycles
- any tree is equivalent to **at least one** DAG (itself)
- any DAG is equivalent to **only one** tree

Sets of trees and sets of DAGs can be characterized by the set of possible **leaves** and the possible **branches**

∴ there is a **functor** from the category of trees to the category of DAGs.



Directed Acyclical Graphs (DAGs) are a more efficient representation for arithmetical expressions than the equivalent trees. E. g.:



- partial order “depends on” prohibits cycles
- any tree is equivalent to **at least one** DAG (itself)
- any DAG is equivalent to **only one** tree

Sets of trees and sets of DAGs can be characterized by the set of possible **leaves** and the possible **branches**

∴ there is a **functor** from the category of trees to the category of DAGs.

😊 modern programming languages allow to make such algebraic relationships explicit and quickly lead to correct implementations.



By virtue of their recursive construction, **tree level 1POWs form a DAG**



By virtue of their recursive construction, **tree level 1POWs form a DAG**:

- ∴ find the smallest DAG that corresponds to a given tree (i. e. a sum of Feynman diagrams)



By virtue of their recursive construction, **tree level 1POWs form a DAG**:

$\therefore$  find the smallest DAG that corresponds to a given tree (i. e. a sum of Feynman diagrams).

Systematic procedure:

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



By virtue of their recursive construction, **tree level 1POWs form a DAG**:

$\therefore$  find the smallest DAG that corresponds to a given tree (i. e. a sum of Feynman diagrams).

Systematic procedure:

**Grow**: starting from the external particles, build the tower of **all** 1POWs up to a given height (the height is always less than the number of external lines) 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.



By virtue of their recursive construction, **tree level 1POWs form a DAG**:

$\therefore$  find the smallest DAG that corresponds to a given tree (i. e. a sum of Feynman diagrams).

Systematic procedure:

**Grow**: starting from the external particles, build the tower of **all** 1POWs up to a given height (the height is always less than the number of external lines) 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.



By virtue of their recursive construction, **tree level 1POWs form a DAG**:

∴ find the smallest DAG that corresponds to a given tree (i. e. a sum of Feynman diagrams).

Systematic procedure:

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

**Select**: from  $D$ , determine **all** possible **flavored kestones** 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 kestones.

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



By virtue of their recursive construction, **tree level 1POWs form a DAG**:

∴ find the smallest DAG that corresponds to a given tree (i. e. a sum of Feynman diagrams).

Systematic procedure:

**Grow**: starting from the external particles, build the tower of **all** 1POWs up to a given height (the height is always less than the number of external lines) 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.



the resulting expression contains **no** more redundancies!



Even for vector particles, the 1POWs are ‘almost’ physical objects and satisfy simple Ward Identities in unbroken gauge theories

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



Even for vector particles, the 1POWs are ‘almost’ physical objects and satisfy simple Ward Identities in unbroken gauge theories

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

and in spontaneously gauge theories in  $R_\xi$ -gauge

$$\frac{\partial}{\partial x_\mu} \langle \text{out} | W_\mu(x) | \text{in} \rangle_{\text{amp.}} = \xi_W m_W \langle \text{out} | \phi_W(x) | \text{in} \rangle_{\text{amp.}} \cdot$$



Even for vector particles, the 1POWs are ‘almost’ physical objects and satisfy simple Ward Identities in unbroken gauge theories

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

and in spontaneously gauge theories in  $R_\xi$ -gauge

$$\frac{\partial}{\partial x_\mu} \langle \text{out} | W_\mu(x) | \text{in} \rangle_{\text{amp.}} = \xi_W m_W \langle \text{out} | \phi_W(x) | \text{in} \rangle_{\text{amp.}} \cdot$$

 code for matrix elements can optionally be instrumented to check these Ward identities.



---

1	Feynman Diagrams, DAGs, and Keystones . . . . .	2
2	Implementation . . . . .	11
	Functional Programming . . . . .	11
	The DAG Functor . . . . .	12
	Objective Caml (a. k. a. O'Caml) . . . . .	14
	O'Mega . . . . .	15
	Architecture . . . . .	16
3	Conclusions . . . . .	18



- Hindley-Milner type system



- Hindley-Milner type system
  - no holes (i. e. no casts): programs never crash



- Hindley-Milner type system
  - no holes (i. e. no casts): programs never crash
  - automatic type inference combines type safety with concise notation



- Hindley-Milner type system
  - no holes (i. e. no casts): programs never crash
  - automatic type inference combines type safety with concise notation
  - parametric polymorphism facilitates code reuse by generic programming



- Hindley-Milner type system
  - no holes (i. e. no casts): programs never crash
  - automatic type inference combines type safety with concise notation
  - parametric polymorphism facilitates code reuse by generic programming
- persistent data structures



- Hindley-Milner type system
  - no holes (i. e. no casts): programs never crash
  - automatic type inference combines type safety with concise notation
  - parametric polymorphism facilitates code reuse by generic programming
- persistent data structures
  - all accessible values remain valid forever



- Hindley-Milner type system
    - no holes (i. e. no casts): programs never crash
    - automatic type inference combines type safety with concise notation
    - parametric polymorphism facilitates code reuse by generic programming
  - persistent data structures
    - all accessible values remain valid forever
- ∴ manipulation of recursive data structures (trees, DAGs, etc.) straightforward, no user memory management required



- Hindley-Milner type system
  - no holes (i. e. no casts): programs never crash
  - automatic type inference combines type safety with concise notation
  - parametric polymorphism facilitates code reuse by generic programming
- persistent data structures
  - all accessible values remain valid forever
  - ∴ manipulation of recursive data structures (trees, DAGs, etc.) straightforward, no user memory management required
- first class functions



- **Hindley-Milner** type system
  - **no holes** (i. e. no casts): programs **never** crash
  - **automatic type inference** combines type safety with concise notation
  - **parametric polymorphism** facilitates code reuse by generic programming
- **persistent** data structures
  - all accessible values remain valid forever
  - ∴ manipulation of recursive data structures (trees, DAGs, etc.) straightforward, **no** user memory management required
- **first class** functions, e. g. composition

```
# let compose f g x = f (g x) ; ;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```
- **functors** mapping **abstract data types** (cf. **categories**)



- **Hindley-Milner** type system
  - **no holes** (i. e. no casts): programs **never** crash
  - **automatic type inference** combines type safety with concise notation
  - **parametric polymorphism** facilitates code reuse by generic programming
- **persistent** data structures
  - all accessible values remain valid forever
  - ∴ manipulation of recursive data structures (trees, DAGs, etc.) straightforward, **no** user memory management required
- **first class** functions, e. g. composition

```
# let compose f g x = f (g x) ; ;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```
- **functors** mapping **abstract data types** (cf. **categories**), e. g.
  - **one code** creates a set from **any** ordered type (numbers, flavors, ...)



- **Hindley-Milner** type system
  - **no holes** (i. e. no casts): programs **never** crash
  - **automatic type inference** combines type safety with concise notation
  - **parametric polymorphism** facilitates code reuse by generic programming
- **persistent** data structures
  - all accessible values remain valid forever
  - ∴ manipulation of recursive data structures (trees, DAGs, etc.) straightforward, **no** user memory management required
- **first class** functions, e. g. composition

```
# let compose f g x = f (g x) ; ;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```
- **functors** mapping **abstract data types** (cf. **categories**), e. g.
  - **one code** creates a set from **any** ordered type (numbers, flavors, ...), **another code** creates a DAG from **any** forest (binary, n-ary, ...)



An **ordered** type:

```
module type Ord = sig type t val compare : t -> t -> int end
```

**Forests** are essentially **parent/offspring** relations:

```
module type Forest =  
  sig  
    module Nodes : Ord  
    type node = Nodes.t  
    type edge and children and t = edge * children  
    val compare : t -> t -> int  
    val for_all : (node -> bool) -> t -> bool  
    val fold : (node -> 'a -> 'a) -> t -> 'a -> 'a  
  end
```

The DAG **functor** creates the DAGs corresponding to a given forest:

```
module MakeDAG (F : Forest) :  
  DAG with type node = F.node and type edge = F.edge  
  and type children = F.children
```



DAGs are essentially condensed **parent/offspring** relations:

```
module type DAG =
  sig
    type node and edge and children and t
    val empty : t
    val add_node : node -> t -> t
    val add_offspring : node -> edge * children -> t -> t
    exception Cycle
    val is_node : node -> t -> bool
    val is_sterile : node -> t -> bool
    val is_offspring : node -> edge * children -> t -> bool
    val fold_nodes : (node -> 'a -> 'a) -> t -> 'a -> 'a
    val fold : (node -> edge * children -> 'a -> 'a) ->
      t -> 'a -> 'a
    val harvest : t -> node -> t -> t
    val lists : t -> (node * (edge * children) list) list
  end
```

**harvest** selects the **minimal** DAG, **list** yields a sequence of **assignments**.



- interactive toplevel and fast bytecode compiler for development and prototyping
- lean & mean implementation . . .



- interactive toplevel and fast bytecode compiler for development and prototyping
- lean & mean implementation . . .
  - 😊 optimizing compiler can reach 50% of the speed of raw C for the same algorithm



- interactive toplevel and fast bytecode compiler for development and prototyping
- lean & mean implementation ...
  - 😊 optimizing compiler can reach 50% of the speed of raw C for the same algorithm
  - ∴ can beat C and C++ with recursive algorithms



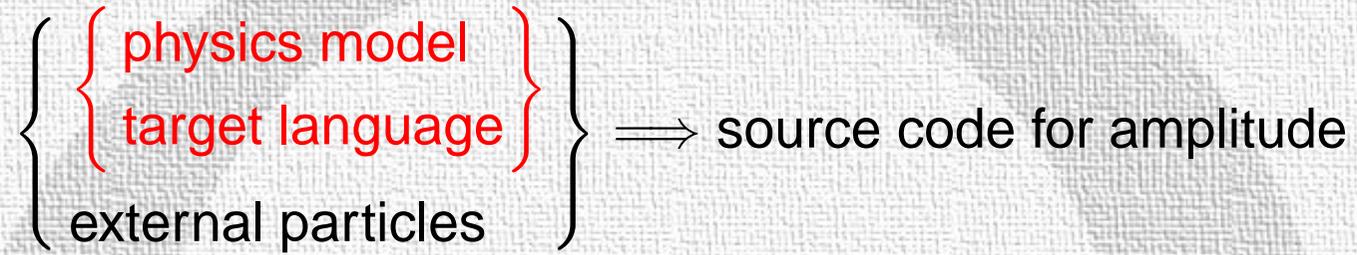
- interactive toplevel and fast bytecode compiler for development and prototyping
- lean & mean implementation ...
  - 😊 optimizing compiler can reach 50% of the speed of raw C for the same algorithm
    - ∴ can beat C and C++ with recursive algorithms
  - 😊 O'Caml's transparent applicative functors more suitable for my applications than Standard-ML's generative functors



- interactive toplevel and fast bytecode compiler for development and prototyping
- lean & mean implementation ...
  - 😊 optimizing compiler can reach 50% of the speed of raw C for the same algorithm
    - ∴ can beat C and C++ with recursive algorithms
  - 😊 O'Caml's transparent applicative functors more suitable for my applications than Standard-ML's generative functors
  - 😊 available as **free** and **open source** software



- interactive toplevel and fast bytecode compiler for development and prototyping
- lean & mean implementation ...
  - 😊 optimizing compiler can reach 50% of the speed of raw C for the same algorithm
    - ∴ can beat C and C++ with recursive algorithms
  - 😊 O'Caml's transparent applicative functors more suitable for my applications than Standard-ML's generative functors
  - 😊 available as **free** and **open source** software
  - 😊 the bytecode variant works on **all systems** with an ANSI-C compiler, the native compiler is available for all relevant systems
  - 😊 can be bootstrapped in  $\mathcal{O}(10)$  minutes





$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{physics model} \\ \text{target language} \end{array} \right\} \\ \text{external particles} \end{array} \right\} \implies \text{source code for amplitude}$$

- the **physics model** is currently specified in **O'CamL**, e. g.:

```
let charged_currents n =  
  [ ((L (-n), Wm, N n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((N (-n), Wp, L n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((D (-n), Wm, U n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((U (-n), Wp, D n), FBF (1, Psibar, VL, Psi), G_CC) ]
```

there will soon be a parser for **CompHEP** model files (probably also for **GRACE**) and later for O'Mega's own input language.



$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{physics model} \\ \text{target language} \end{array} \right\} \\ \text{external particles} \end{array} \right\} \implies \text{source code for amplitude}$$

- the **physics model** is currently specified in **O'Caml**, e. g.:

```
let charged_currents n =  
  [ ((L (-n), Wm, N n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((N (-n), Wp, L n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((D (-n), Wm, U n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((U (-n), Wp, D n), FBF (1, Psibar, VL, Psi), G_CC) ]
```

there will soon be a parser for **CompHEP** model files (probably also for **GRACE**) and later for O'Mega's own input language.

- the **target language** is specified as a O'Caml module.



$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{physics model} \\ \text{target language} \end{array} \right\} \\ \text{external particles} \end{array} \right\} \implies \text{source code for amplitude}$$

- the **physics model** is currently specified in **O'Cam1**, e. g.:

```
let charged_currents n =  
  [ ((L (-n), Wm, N n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((N (-n), Wp, L n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((D (-n), Wm, U n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((U (-n), Wp, D n), FBF (1, Psibar, VL, Psi), G_CC) ]
```

there will soon be a parser for **CompHEP** model files (probably also for **GRACE**) and later for O'Mega's own input language.

- the **target language** is specified as a O'Cam1 module. Inventing a specification language will not be economical in this case.



$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{physics model} \\ \text{target language} \end{array} \right\} \\ \text{external particles} \end{array} \right\} \implies \text{source code for amplitude}$$

- the **physics model** is currently specified in **O'Cam1**, e. g.:

```
let charged_currents n =  
  [ ((L (-n), Wm, N n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((N (-n), Wp, L n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((D (-n), Wm, U n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((U (-n), Wp, D n), FBF (1, Psibar, VL, Psi), G_CC) ]
```

there will soon be a parser for **CompHEP** model files (probably also for **GRACE**) and later for O'Mega's own input language.

- the **target language** is specified as a O'Cam1 module. Inventing a specification language will not be economical in this case.
- the external particles are given on the command line



$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{physics model} \\ \text{target language} \end{array} \right\} \\ \text{external particles} \end{array} \right\} \implies \text{source code for amplitude}$$

- the **physics model** is currently specified in **O'Cam1**, e. g.:

```
let charged_currents n =  
  [ ((L (-n), Wm, N n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((N (-n), Wp, L n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((D (-n), Wm, U n), FBF (1, Psibar, VL, Psi), G_CC);  
    ((U (-n), Wp, D n), FBF (1, Psibar, VL, Psi), G_CC) ]
```

there will soon be a parser for **CompHEP** model files (probably also for **GRACE**) and later for O'Mega's own input language.

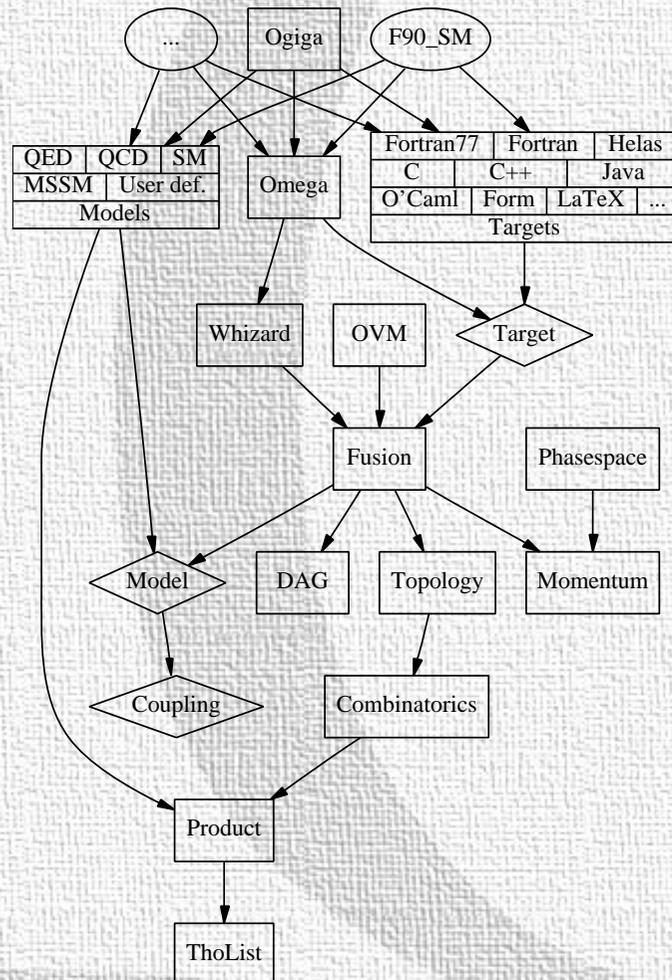
- the **target language** is specified as a O'Cam1 module. Inventing a specification language will not be economical in this case.
- the external particles are given on the command line (GUI later)



O'Caml's powerful module system supports a very flexible architecture w/ **functors** building applications from independent modules



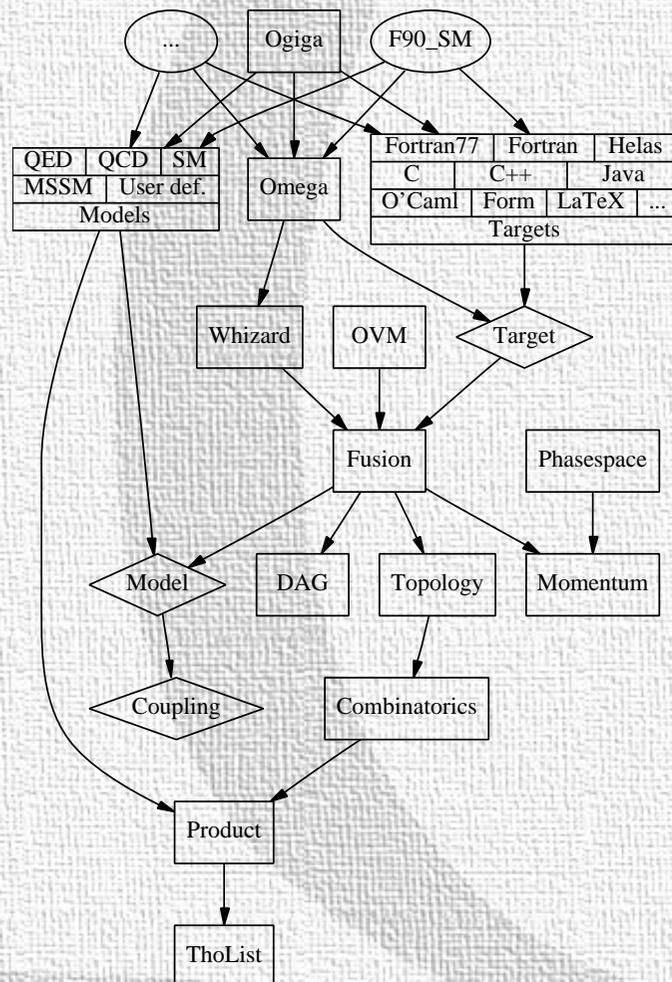
O'Caml's powerful module system supports a very flexible architecture w/ **functors** building applications from independent modules





O'Caml's powerful module system supports a very flexible architecture w/ **functors** building applications from independent modules

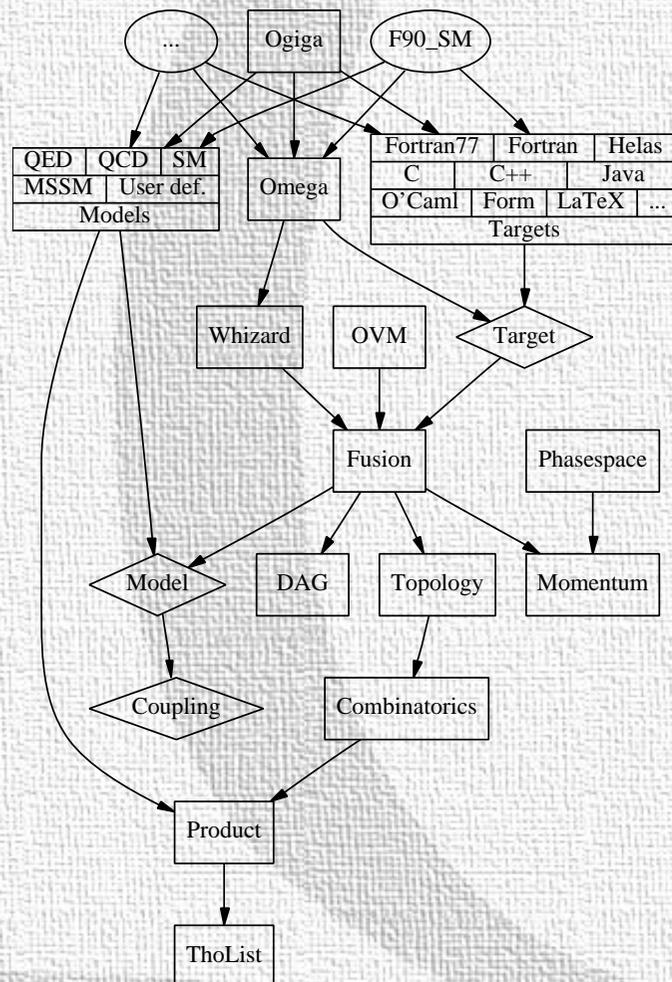
The module `Targets` contains implementations of the signature `Target` for each target language





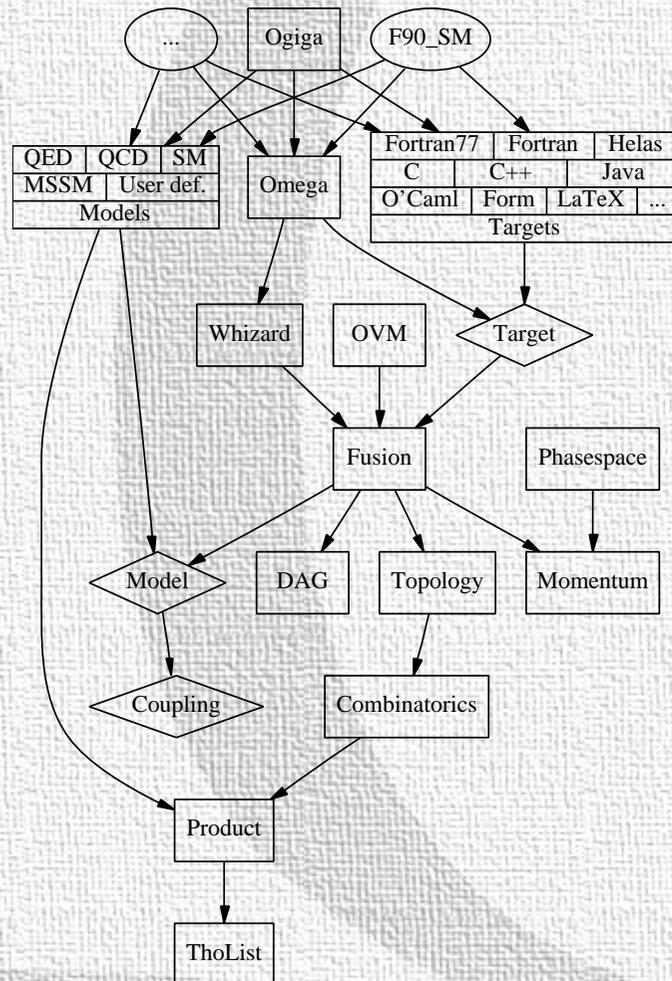
O'Caml's powerful module system supports a very flexible architecture w/ **functors** building applications from independent modules

The module `Targets` contains implementations of the signature `Target` for each target language & the module `Models` contains implementations of `Model` for each (class of) physics models.





O’Caml’s powerful module system supports a very flexible architecture w/ **functors** building applications from independent modules



The module `Targets` contains implementations of the signature `Target` for each target language & the module `Models` contains implementations of `Model` for each (class of) physics models. E.g. the application writing Fortran95 for the standard model is

```
module O = Omega.Make
    (Fusion.Binary)
    (Targets.Fortran)
    (Models.SM)

let _ = O.main ()
```



---

1 Feynman Diagrams, DAGs, and Keystones . . . . .	2
2 Implementation . . . . .	11
3 Conclusions . . . . .	18
First Results . . . . .	18
Outlook . . . . .	19



Radiative corrections to four fermion production, standard model, unitarity gauge:

process	Diagrams		
$e^+e^- \rightarrow$	#		
$e^+\bar{\nu}_e d\bar{u}$			
$e^+\bar{\nu}_e d\bar{u}\gamma$			
$e^+\bar{\nu}_e d\bar{u}\gamma\gamma$			
$e^+\bar{\nu}_e d\bar{u}\gamma\gamma\gamma$			
$e^+\bar{\nu}_e d\bar{u}\gamma\gamma\gamma\gamma$			



Radiative corrections to four fermion production, standard model, unitarity gauge:

process	Diagrams		
$e^+e^- \rightarrow$	#		
$e^+\bar{\nu}_e d\bar{u}$	20		
$e^+\bar{\nu}_e d\bar{u}\gamma$	146		
$e^+\bar{\nu}_e d\bar{u}\gamma\gamma$	1112		
$e^+\bar{\nu}_e d\bar{u}\gamma\gamma\gamma$	12420		
$e^+\bar{\nu}_e d\bar{u}\gamma\gamma\gamma\gamma$	138816		



Radiative corrections to four fermion production, standard model, unitarity gauge:

process $e^+e^- \rightarrow$	Diagrams		O'Mega	
	#		#prop.	
$e^+\bar{\nu}_e d\bar{u}$	20		14	
$e^+\bar{\nu}_e d\bar{u}\gamma$	146		36	
$e^+\bar{\nu}_e d\bar{u}\gamma\gamma$	1112		94	
$e^+\bar{\nu}_e d\bar{u}\gamma\gamma\gamma$	12420		168	
$e^+\bar{\nu}_e d\bar{u}\gamma\gamma\gamma\gamma$	138816		344	



Radiative corrections to four fermion production, standard model, unitarity gauge:

process	Diagrams		O'Mega	
	#	vertices	#prop.	vertices
$e^+ e^- \rightarrow e^+ \bar{\nu}_e d \bar{u}$	20	80	14	44
$e^+ e^- \rightarrow e^+ \bar{\nu}_e d \bar{u} \gamma$	146	730	36	151
$e^+ e^- \rightarrow e^+ \bar{\nu}_e d \bar{u} \gamma \gamma$	1112	6672	94	468
$e^+ e^- \rightarrow e^+ \bar{\nu}_e d \bar{u} \gamma \gamma \gamma$	12420	86940	168	1246
$e^+ e^- \rightarrow e^+ \bar{\nu}_e d \bar{u} \gamma \gamma \gamma \gamma$	138816	1110528	344	3746



Radiative corrections to four fermion production, standard model, unitarity gauge:

process	Diagrams		O'Mega	
	#	vertices	#prop.	vertices
$e^+ e^- \rightarrow$				
$e^+ \bar{\nu}_e d \bar{u}$	20	80	14	44
$e^+ \bar{\nu}_e d \bar{u} \gamma$	146	730	36	151
$e^+ \bar{\nu}_e d \bar{u} \gamma \gamma$	1112	6672	94	468
$e^+ \bar{\nu}_e d \bar{u} \gamma \gamma \gamma$	12420	86940	168	1246
$e^+ \bar{\nu}_e d \bar{u} \gamma \gamma \gamma \gamma$	138816	1110528	344	3746

- O'Mega amplitudes for up to 7 particles ("2  $\rightarrow$  5") tested against MADGRAPH



Radiative corrections to four fermion production, standard model, unitarity gauge:

process	Diagrams		O'Mega	
	#	vertices	#prop.	vertices
$e^+ e^- \rightarrow$				
$e^+ \bar{\nu}_e d \bar{u}$	20	80	14	44
$e^+ \bar{\nu}_e d \bar{u} \gamma$	146	730	36	151
$e^+ \bar{\nu}_e d \bar{u} \gamma \gamma$	1112	6672	94	468
$e^+ \bar{\nu}_e d \bar{u} \gamma \gamma \gamma$	12420	86940	168	1246
$e^+ \bar{\nu}_e d \bar{u} \gamma \gamma \gamma \gamma$	138816	1110528	344	3746

- O'Mega amplitudes for up to 7 particles ("2  $\rightarrow$  5") tested against MADGRAPH

😊 agreement for random momenta **always** better than  $10^{-11}$



Radiative corrections to four fermion production, standard model, unitarity gauge:

process	Diagrams		O'Mega	
	#	vertices	#prop.	vertices
$e^+e^- \rightarrow e^+\bar{\nu}_e d\bar{u}$	20	80	14	44
$e^+e^- \rightarrow e^+\bar{\nu}_e d\bar{u}\gamma$	146	730	36	151
$e^+e^- \rightarrow e^+\bar{\nu}_e d\bar{u}\gamma\gamma$	1112	6672	94	468
$e^+e^- \rightarrow e^+\bar{\nu}_e d\bar{u}\gamma\gamma\gamma$	12420	86940	168	1246
$e^+e^- \rightarrow e^+\bar{\nu}_e d\bar{u}\gamma\gamma\gamma\gamma$	138816	1110528	344	3746

- O'Mega amplitudes for up to 7 particles ("2  $\rightarrow$  5") tested against MADGRAPH

😊 agreement for random momenta **always** better than  $10^{-11}$

First **realistic application**

- simulation of **six fermion final states** in  $W^+W^-$  scattering for the **TESLA Technical Design Report**, using **WHIZARD** by **Wolfgang Kilian** as unweighted event generator.



Radiative corrections to four fermion production, standard model, unitarity gauge:

process	Diagrams		O'Mega	
	#	vertices	#prop.	vertices
$e^+e^- \rightarrow e^+\bar{\nu}_e d\bar{u}$	20	80	14	44
$e^+e^- \rightarrow e^+\bar{\nu}_e d\bar{u}\gamma$	146	730	36	151
$e^+e^- \rightarrow e^+\bar{\nu}_e d\bar{u}\gamma\gamma$	1112	6672	94	468
$e^+e^- \rightarrow e^+\bar{\nu}_e d\bar{u}\gamma\gamma\gamma$	12420	86940	168	1246
$e^+e^- \rightarrow e^+\bar{\nu}_e d\bar{u}\gamma\gamma\gamma\gamma$	138816	1110528	344	3746

- O'Mega amplitudes for up to 7 particles ("2  $\rightarrow$  5") tested against MADGRAPH

😊 agreement for random momenta **always** better than  $10^{-11}$

First **realistic application**

- simulation of **six fermion final states** in  $W^+W^-$  scattering for the **TESLA Technical Design Report**, using **WHIZARD** by **Wolfgang Kilian** as unweighted event generator.

Get it from <http://www.ikp.physik.tu-darmstadt.de/~ohl/omega/>.



- QCD



- QCD
  - up to two colored particles are already handled



- QCD
  - up to two colored particles are already handled
  - factorization for many-jet final states?



- QCD
  - up to two colored particles are already handled
  - factorization for many-jet final states?
- Supersymmetry and MSSM



- QCD
  - up to two colored particles are already handled
  - factorization for many-jet final states?
- Supersymmetry and MSSM
  - 😊 Jürgen Reuter (Darmstadt) has added unified support for Dirac and Majorana fermions using the Feynman rules of Ansgar Denner et al.



- QCD
  - up to two colored particles are already handled
  - factorization for many-jet final states?
- Supersymmetry and MSSM
  - 😊 Jürgen Reuter (Darmstadt) has added unified support for Dirac and Majorana fermions using the Feynman rules of Ansgar Denner et al.
- O'Mega Virtual Machine



- **QCD**
  - up to two colored particles are already handled
  - factorization for many-jet final states?
- **Supersymmetry** and **MSSM**
  - 😊 **Jürgen Reuter (Darmstadt)** has added unified support for Dirac and Majorana fermions using the Feynman rules of **Ansgar Denner et al.**
- **O'Mega Virtual Machine**
  - ∴ most time is spent in non-trivial vertex evaluations for vectors and spinors, that take  $O(10)$  complex multiplications



- **QCD**
  - up to two colored particles are already handled
  - factorization for many-jet final states?
- **Supersymmetry** and **MSSM**
  - 😊 **Jürgen Reuter (Darmstadt)** has added unified support for Dirac and Majorana fermions using the Feynman rules of **Ansgar Denner et al.**
- **O'Mega Virtual Machine**
  - ∴ most time is spent in non-trivial vertex evaluations for vectors and spinors, that take  $O(10)$  complex multiplications
  - ∴ virtual vertex evaluation machines can challenge native code and avoid compilations



- **QCD**
  - up to two colored particles are already handled
  - factorization for many-jet final states?
- **Supersymmetry** and **MSSM**
  - 😊 **Jürgen Reuter (Darmstadt)** has added unified support for Dirac and Majorana fermions using the Feynman rules of **Ansgar Denner et al.**
- **O'Mega Virtual Machine**
  - ∴ most time is spent in non-trivial vertex evaluations for vectors and spinors, that take  $O(10)$  complex multiplications
  - ∴ virtual vertex evaluation machines can challenge native code and avoid compilations
- **O'Giga: O'Mega Graphical Interface for Generation and Analysis**



- **QCD**
  - up to two colored particles are already handled
  - factorization for many-jet final states?
- **Supersymmetry** and **MSSM**
  - 😊 **Jürgen Reuter (Darmstadt)** has added unified support for Dirac and Majorana fermions using the Feynman rules of **Ansgar Denner et al.**
- **O'Mega Virtual Machine**
  - ∴ most time is spent in non-trivial vertex evaluations for vectors and spinors, that take  $O(10)$  complex multiplications
  - ∴ virtual vertex evaluation machines can challenge native code and avoid compilations
- **O'Giga**: **O**'Mega **G**raphical **I**nterface for **G**eneration and **A**nalysis
- **O'Tera**: **O**'Mega **T**ool for **E**valuating **R**enormalized **A**mplitudes



- **QCD**
  - up to two colored particles are already handled
  - factorization for many-jet final states?
- **Supersymmetry** and **MSSM**
  - 😊 **Jürgen Reuter (Darmstadt)** has added unified support for Dirac and Majorana fermions using the Feynman rules of **Ansgar Denner et al.**
- **O'Mega Virtual Machine**
  - ∴ most time is spent in non-trivial vertex evaluations for vectors and spinors, that take  $O(10)$  complex multiplications
  - ∴ virtual vertex evaluation machines can challenge native code and avoid compilations
- **O'Giga: O'Mega Graphical Interface for Generation and Analysis**
- **O'Tera: O'Mega Tool for Evaluating Renormalized Amplitudes**