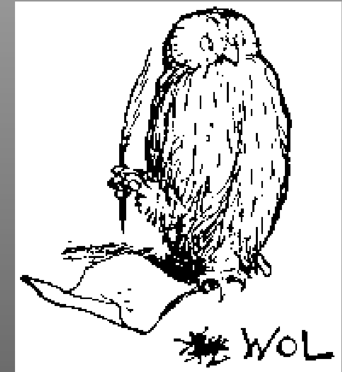
FeynArts and FormCalc

Thomas Hahn

Max-Planck-Institut für Physik
München

Contradiction



An Introduction is to introduce people, but FeynArts and Friends have already been introduced to you. So this is the opposite. When we asked Pooh what the opposite of an Introduction was, he said "The what of a what?" which didn't help us as much as we had hoped, but luckily Owl kept his head and told us that the Opposite of an Introduction, my dear Pooh, was a Contradiction.

. . . actually, a Distinction:

FeynArts/FormCalc	FeynHiggs
Generic answer to an arbitrary question	Concrete answer to a specific question
Paint & brush	Finished painting
Symbolic part (Mathematica)	Numerical program (hand-tuned Fortran)
Numerical part (generated Fortran)	Mathematica interface

Plan: Walk through the general setup of these programs and show some perhaps non-standard applications.

http://feynarts.de/examples.tar.gz

Automated Diagram Evaluation

FeynArts

Amplitudes

FormCalc

Fortran Code

LoopTools

Diagram Generation:

- Create the topologies
- Insert fields
- Apply the Feynman rules
- Paint the diagrams

Algebraic Simplification:

- Contract indices
- Calculate traces
- Reduce tensor integrals
- Introduce abbreviations

Numerical Evaluation:

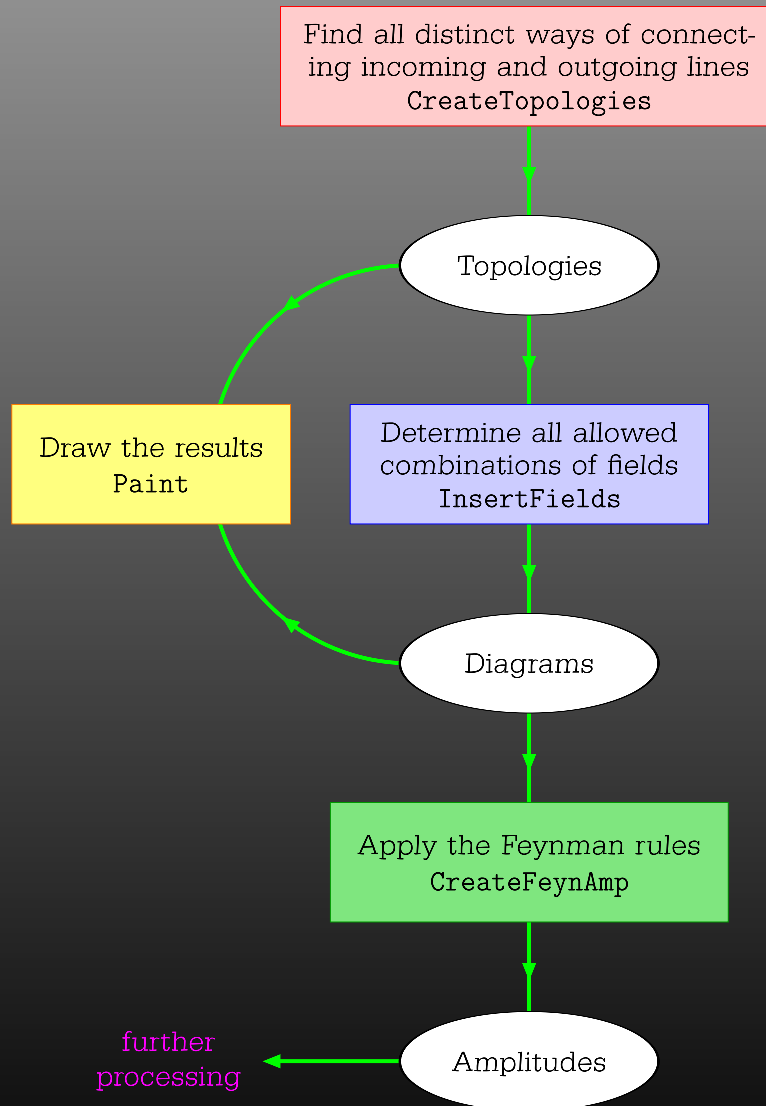
- Convert Mathematica output to Fortran code
- Supply a driver program
- Implementation of the integrals

Symbolic manipulation (Computer Algebra) for the structural and algebraic operations.

Compiled high-level language (Fortran) for the numerical evaluation.

$|\mathcal{M}|^2 \longrightarrow$ **Cross-sections, Decay rates, . . .**

FeynArts



EXAMPLE: generating the photon self-energy

```
top = CreateTopologies[ 1, 1 -> 1 ]
```

```
Paint[top]
```

```
ins = InsertFields[ top, V[1] -> V[1],
      Model -> SM ]
```

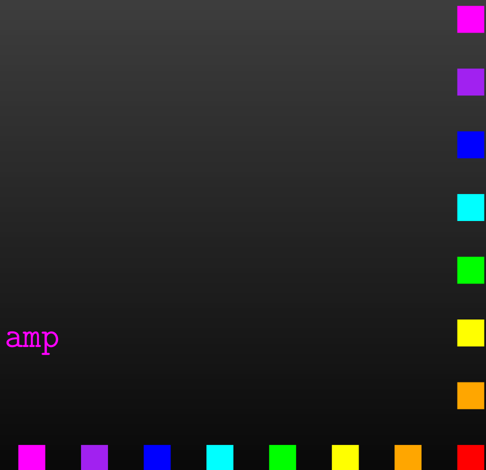
use the Standard Model

the name of the photon in the “SM” model file

```
Paint[ins]
```

```
amp = CreateFeynAmp[ins]
```

```
amp >> PhotonSelfEnergy.amp
```

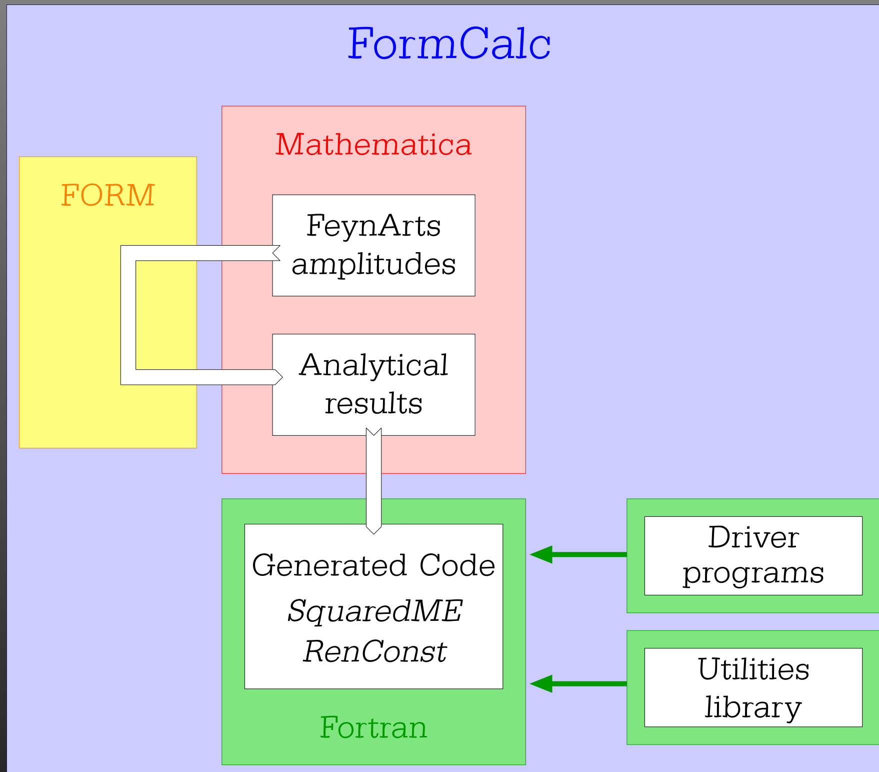


Algebraic Simplification

The amplitudes output by FeynArts so far are in no good shape for direct numerical evaluation. Some objects must/should be handled symbolically, e.g. tensorial objects, Dirac traces, dimension (D vs. 4).

- contract indices as far as possible,
- evaluate fermion traces,
- perform the tensor reduction,
- add local terms arising from D·(divergent integral),
- simplify open fermion chains,
- simplify and compute the square of SU(N) structures,
- "compactify" the results as much as possible.

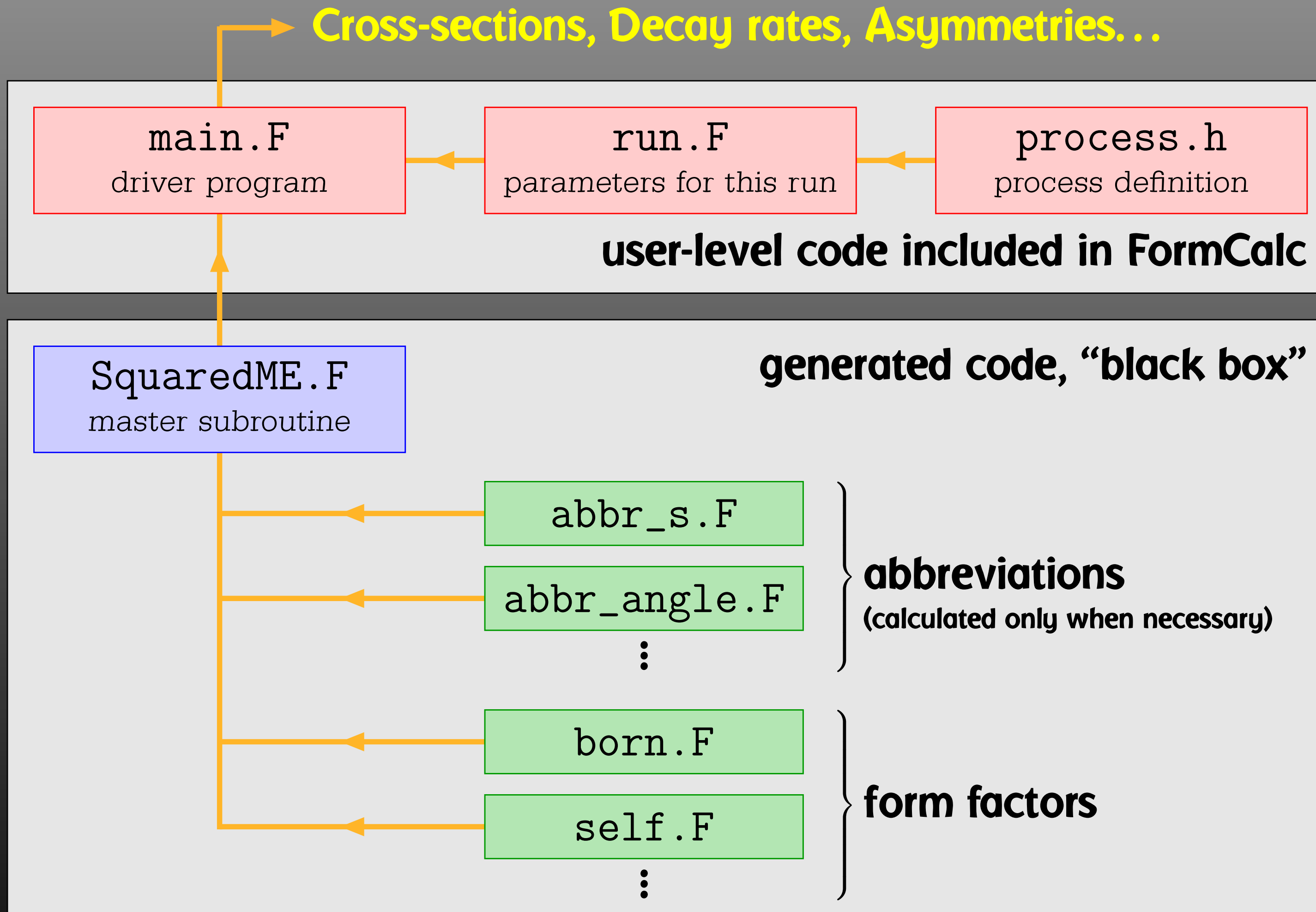
FormCalc



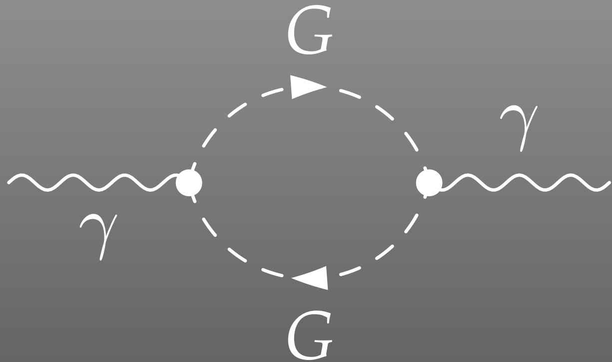
EXAMPLE: Calculating the photon self-energy

```
In[1]:= << FormCalc`

FormCalc 6.1
by Thomas Hahn
last revised 6 Jul 09

In[2]:= CalcFeynAmp[<< PhotonSelfEnergy.amp]

preparing FORM code in /tmp/m1.frm
> 2 amplitudes with insertions
> 5 amplitudes without insertions
running FORM...  ok

                        -3 Alfa Pair1 A0[MW2]
Out[2]= Amp[{0} -> {0}][--------------------- +
                               2 Pi

   3 Alfa Pair1 B00[0, MW2, MW2]
   -----------------------------,
                Pi

   (Alfa Pair1 A0[MLE2[Gen1]]
    ------------------------- +
               Pi

    Alfa Pair1 A0[MQD2[Gen1]]
    ------------------------- +
              3 Pi

    4 Alfa Pair1 A0[MQU2[Gen1]]
    --------------------------- -
               3 Pi

    2 Alfa Pair1 B00[0, MLE2[Gen1], MLE2[Gen1]]
    ------------------------------------------- -
                        Pi

    2 Alfa Pair1 B00[0, MQD2[Gen1], MQD2[Gen1]]
    ------------------------------------------- -
                       3 Pi

    8 Alfa Pair1 B00[0, MQU2[Gen1], MQU2[Gen1]]
    -------------------------------------------) *
                       3 Pi

    SumOver[Gen1, 3]]
```

Numerical Evaluation in Fortran 77

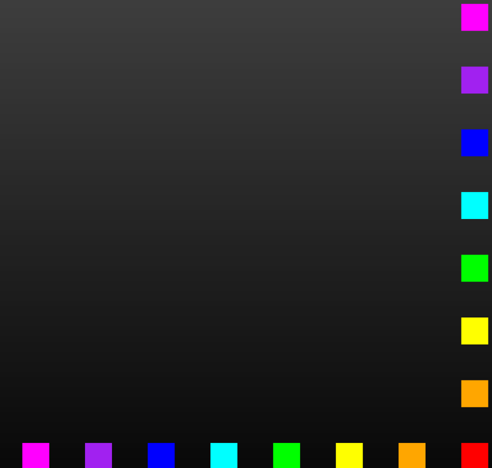


Sample CreateFeynAmp output

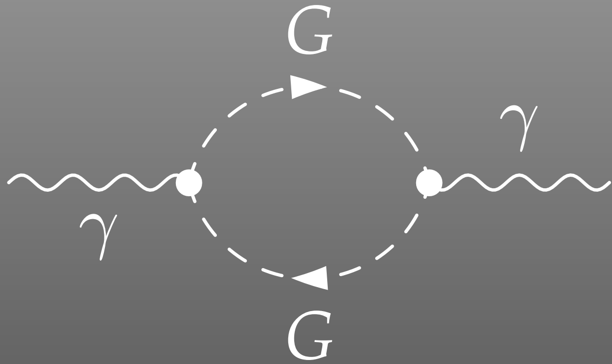


= FeynAmp[*identifier* ,
loop momenta ,
generic amplitude ,
insertions]

```
GraphID[Topology == 1, Generic == 1]
```

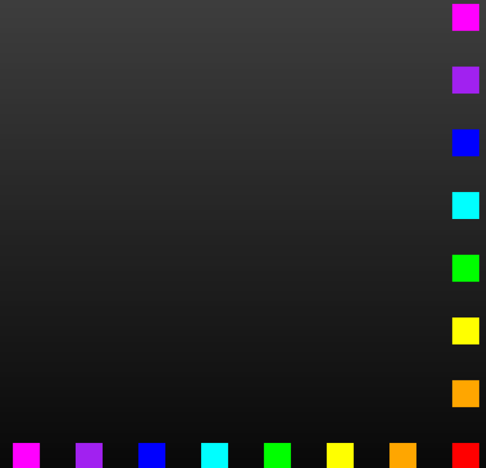


Sample CreateFeynAmp output

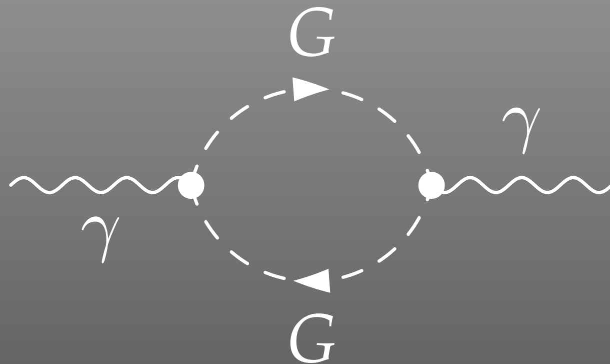


= FeynAmp[*identifier* ,
loop momenta ,
generic amplitude ,
insertions]

`Integral[q1]`



Sample CreateFeynAmp output



= FeynAmp[*identifier*,
loop momenta,
generic amplitude,
insertions]

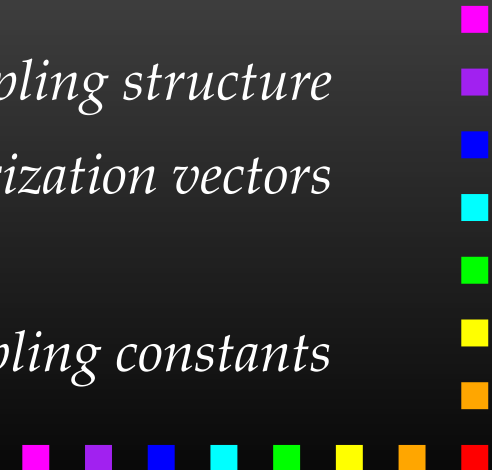
$\frac{\text{I}}{32\ \text{Pi}^4}$ RelativeCF .... *prefactor*

FeynAmpDenominator[$\frac{1}{\text{q1}^2 - \text{Mass[S[Gen3]]}^2}$,
$\frac{1}{(-\text{p1} + \text{q1})^2 - \text{Mass[S[Gen4]]}^2}$] .... *loop denominators*

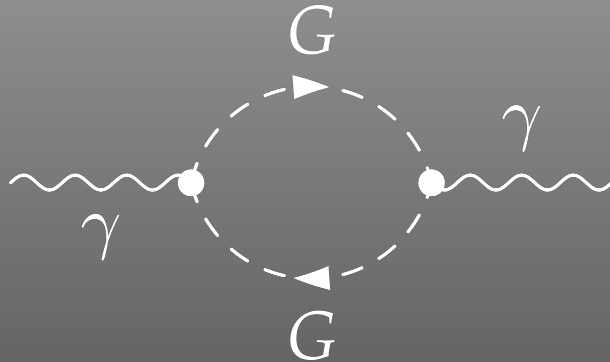
(p1 - 2 q1)[Lor1] (-p1 + 2 q1)[Lor2] .... *kin. coupling structure*

ep[V[1], p1, Lor1] ep*[V[1], k1, Lor2] .... *polarization vectors*

$\text{G}^{(0)}_{\text{SSV}}$[(Mom[1] - Mom[2])[KI1[3]]]
$\text{G}^{(0)}_{\text{SSV}}$[(Mom[1] - Mom[2])[KI1[3]]], ... *coupling constants*

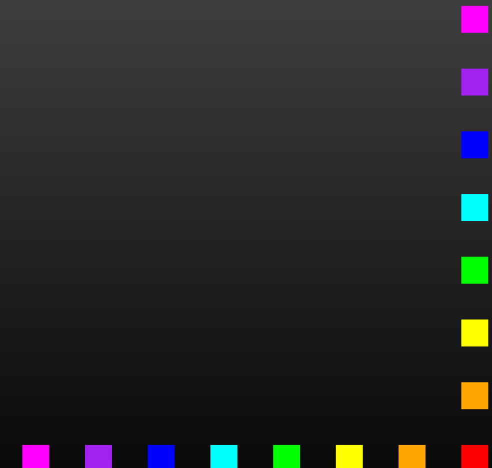


Sample CreateFeynAmp output



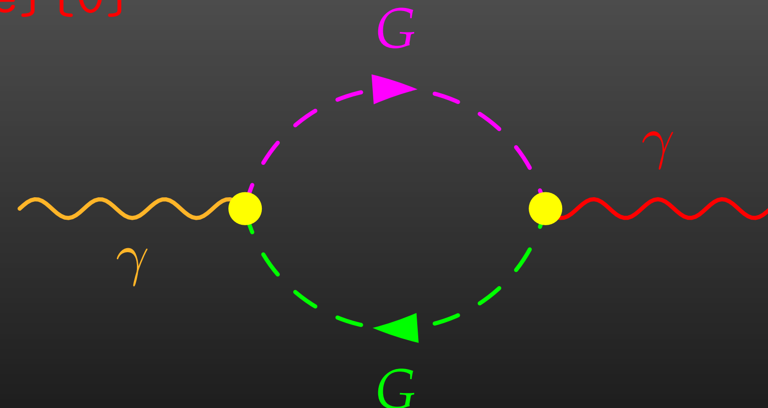
= FeynAmp[*identifier* ,
loop momenta ,
generic amplitude ,
insertions]

```
{ Mass[S[Gen3]],
  Mass[S[Gen4]],
  G(0)SSV[(Mom[1] - Mom[2])[KI1[3]]],
  G(0)SSV[(Mom[1] - Mom[2])[KI1[3]]],
  RelativeCF } ->
Insertions[Classes][{MW, MW, I EL, -I EL, 2}]
```



Sample Paint output

```
\begin{feynartspicture}(150,150)(1,1)
\FADiagram{}
\FAProp(6.,10.)(14.,10.)(0.8,){ScalarDash}{-1}
\FALabel(10.,5.73)[t]{$G$}
\FAProp(6.,10.)(14.,10.)(-0.8,){ScalarDash}{1}
\FALabel(10.,14.27)[b]{$G$}
\FAProp(0.,10.)(6.,10.)(0.,){Sine}{0}
\FALabel(3.,8.93)[t]{$\gamma$}
\FAProp(20.,10.)(14.,10.)(0.,){Sine}{0}
\FALabel(17.,11.07)[b]{$\gamma$}
\FAVert(6.,10.){0}
\FAVert(14.,10.){0}
\end{feynartspicture}
```



Technically: uses its own PostScript prologue.

FormCalc Output

A typical term in the output looks like

```
C0i[cc12, MW2, MW2, S, MW2, MZ2, MW2] *
  ( -4 Alfa2 MW2 CW2/SW2 S AbbSum16 +
    32 Alfa2 CW2/SW2 S^2 AbbSum28 +
    4 Alfa2 CW2/SW2 S^2 AbbSum30 -
    8 Alfa2 CW2/SW2 S^2 AbbSum7 +
    Alfa2 CW2/SW2 S (T - U) Abb1 +
    8 Alfa2 CW2/SW2 S (T - U) AbbSum29 )
```

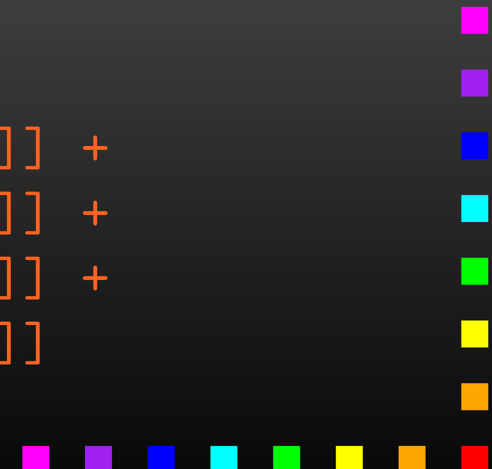
Abbreviations

Outright factorization is usually out of question.
Abbreviations are necessary to reduce size of expressions.

```
AbbSum29 = Abb2 + Abb22 + Abb23 + Abb3

Abb22 = Pair1 Pair3 Pair6

Pair3 = Pair[e[3],k[1]]
```

The full expression corresponding to `AbbSum29` **is**

```
Pair[e[1],e[2]] Pair[e[3],k[1]] Pair[e[4],k[1]] +
Pair[e[1],e[2]] Pair[e[3],k[2]] Pair[e[4],k[1]] +
Pair[e[1],e[2]] Pair[e[3],k[1]] Pair[e[4],k[2]] +
Pair[e[1],e[2]] Pair[e[3],k[2]] Pair[e[4],k[2]]
```



Categories of Abbreviations

- Abbreviations are **recursively defined** in several levels.
- When generating Fortran code, FormCalc introduces another set of abbreviations for the **loop integrals.**

In general, the **abbreviations are thus costly in CPU time.**
It is key to a decent performance that the abbreviations are separated into different **Categories:**

- **Abbreviations that depend on the helicities,**
- **Abbreviations that depend on angular variables,**
- **Abbreviations that depend only on $\sqrt{s}$.**

Correct execution of the categories guarantees that **almost no redundant evaluations** are made and makes the generated code essentially as fast as hand-tuned code.

External Fermion Lines

An amplitude containing **external fermions** has the form

$$\mathcal{M} = \sum_{i=1}^{n_F} c_i\, F_i \quad \textbf{where} \quad F_i = \textbf{(Product of)}\, \langle u|\, \Gamma_i\, |v\rangle\,.$$

$n_F =$ number of fermionic structures.

Textbook procedure: **Trace Technique**

$$|\mathcal{M}|^2 = \sum_{i,j=1}^{n_F} c_i^*\, c_j\, F_i^* F_j$$

where $\quad F_i^* F_j = \langle v|\, \bar\Gamma_i\, |u\rangle\, \langle u|\, \Gamma_j\, |v\rangle = \mathrm{Tr}\big(\bar\Gamma_i\ |u\rangle\langle u|\ \Gamma_j\ |v\rangle\langle v|\big)\,.$

Problems with the Trace Technique

PRO: Trace technique is independent of any representation.

CON: For n_F F_i's there are n_F^2 $F_i^* F_j$'s.

Things get worse the more vectors are in the game: multi-particle final states, polarization effects . . .

Essentially $n_F \sim$ (# of vectors)! because all combinations of vectors can appear in the Γ_i.

Solution: Use Weyl-van der Waerden spinor formalism to compute the F_i's directly.

Sigma Chains

Define Sigma matrices and 2-dim. Spinors as

$$\sigma_\mu = (\mathbb{1}, -\vec\sigma)\,, \qquad \langle u|_{4\mathrm{d}} \equiv \big(\langle u_+|_{2\mathrm{d}}\,, \langle u_-|_{2\mathrm{d}}\big)\,,$$

$$\overline\sigma_\mu = (\mathbb{1}, +\vec\sigma)\,, \qquad |v\rangle_{4\mathrm{d}} \equiv \begin{pmatrix} |v_-\rangle_{2\mathrm{d}} \\ |v_+\rangle_{2\mathrm{d}} \end{pmatrix}.$$

Using the chiral representation it is easy to show that every chiral 4-dim. Dirac chain can be converted to a *single* 2-dim. sigma chain:

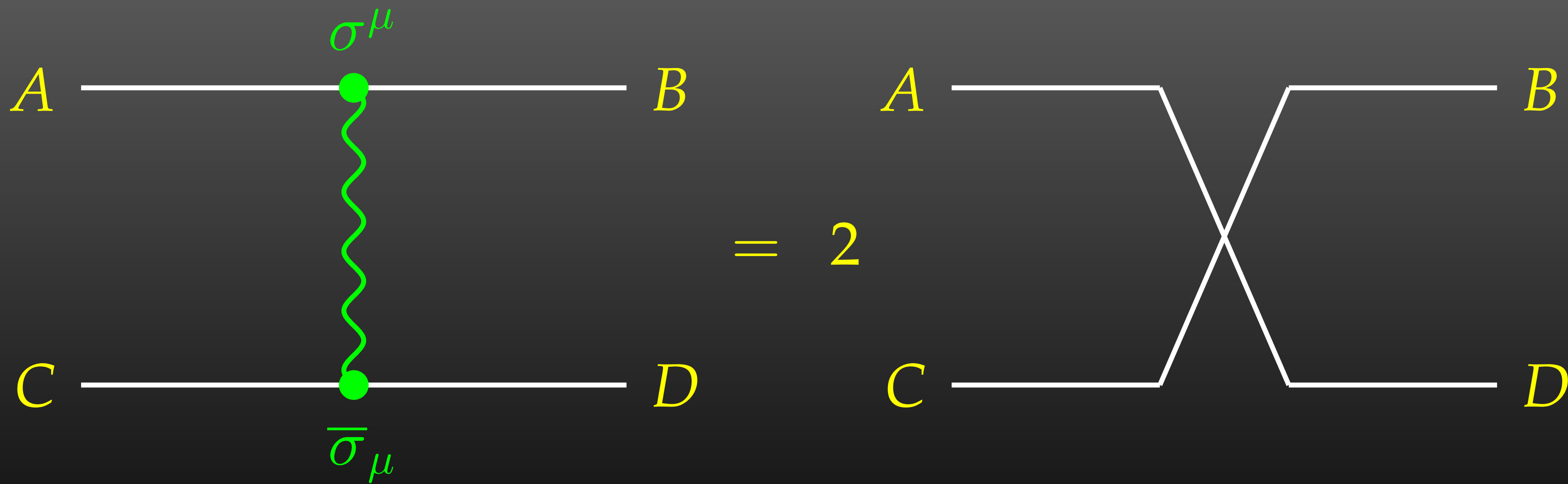
$$\langle u|\,\omega_-\gamma_\mu\gamma_\nu\cdots|v\rangle = \langle u_-|\,\overline\sigma_\mu\sigma_\nu\cdots|v_\pm\rangle\,,$$

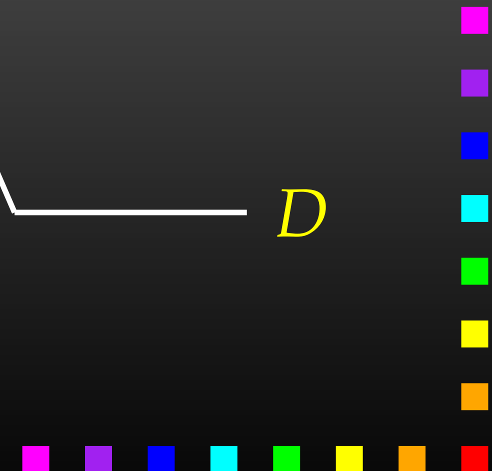
$$\langle u|\,\omega_+\gamma_\mu\gamma_\nu\cdots|v\rangle = \langle u_+|\,\sigma_\mu\overline\sigma_\nu\cdots|v_\mp\rangle\,.$$

Fierz Identities

With the Fierz identities for sigma matrices it is possible to **remove all Lorentz contractions** between sigma chains, e.g.

$$\langle A|\,\sigma_\mu\,|B\rangle\ \langle C|\,\overline{\sigma}^\mu\,|D\rangle \;=\; 2\,\langle A|D\rangle\ \langle C|B\rangle$$





Implementation

- **Objects (arrays):** $|u_\pm\rangle \sim \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}, \quad (\sigma \cdot k) \sim \begin{pmatrix} a & b \\ c & d \end{pmatrix}$
- **Operations (functions):**

$$\langle u | v \rangle \sim (u_1 \;\; u_2) \cdot \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \qquad \texttt{SxS}$$

$$(\overset{(-)}{\sigma} \cdot k) \, | v \rangle \sim \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \qquad \texttt{VxS}, \; \texttt{BxS}$$

Sufficient to compute any sigma chain:

$$\langle u | \, \sigma_\mu \overline{\sigma}_\nu \sigma_\rho \, | v \rangle \, k_1^\mu k_2^\nu k_3^\rho = \texttt{SxS}(\, u, \, \texttt{VxS}(\, k_1, \, \texttt{BxS}(\, k_2, \, \texttt{VxS}(\, k_3, \, v \,)\,)\,)\,)$$

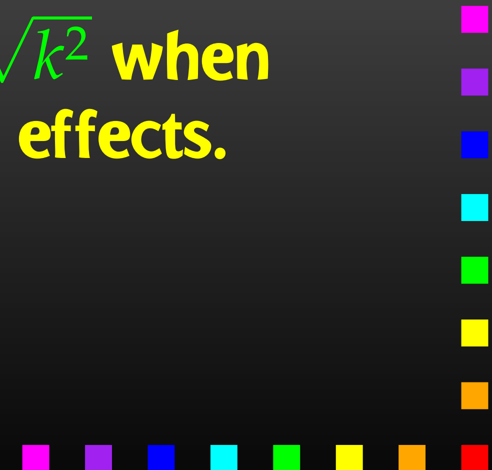
More Freebies

- **Polarization does not 'cost' extra:**
  = Get spin physics for free.
- **Better numerical stability because components of k^μ are arranged as 'small' and 'large' matrix entries, viz.**

$$\sigma_\mu k^\mu = \begin{pmatrix} k_0 + k_3 & k_1 - \mathrm{i}k_2 \\ k_1 + \mathrm{i}k_2 & k_0 - k_3 \end{pmatrix}$$

$$\downarrow$$

Large cancellations of the form $\sqrt{k^2 + m^2} - \sqrt{k^2}$ when $m \ll k$ are avoided: better precision for mass effects.

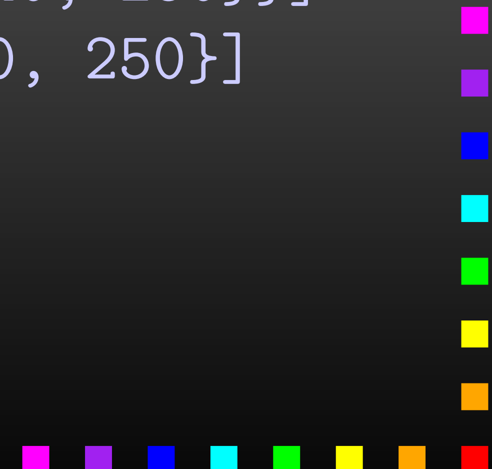


Mathematica Interface

The new Mathematica Interface turns the generated stand-alone Fortran code into a Mathematica function for evaluating the cross-section or decay rate as a function of user-selected model parameters.

The benefits of such a function are obvious, as the whole instrumentarium of Mathematica commands can be applied to them. Just think of

```
FindMinimum[sigma[TB, MA0], {{TB, 5}, {MA0, 250}}]
ContourPlot[sigma[TB, MA0], {TB, 5}, {MA0, 250}]
...
```



Mathematica Interface – Input

The changes to the code are minimal.

Example line in `run.F` for Stand-alone Fortran code:

```
#define LOOP1 do 1 TB = 5, 50, 5
```

Change for the Mathematica Interface:

```
#define LOOP1 call MmaGetReal(TB)
```

The variable `TB` is 'imported' from Mathematica now, i.e. the cross-section function in Mathematica becomes a function of `TB` hereby.

The user has full control over which variables are 'imported' from Mathematica and which are set in Fortran.

Mathematica Interface – Output

Similar to the MmaGetReal invocations, the Fortran program can also **‘export’ variables to Mathematica.**

For example, the line that prints a parameter in the stand-alone code is

```
#define PRINT1 SHOW "TB", TB
```

becomes

```
#define PRINT1 call MmaPutReal("TB", TB)
```

for the Mathematica Interface and transmits the value of TB to Mathematica.

Mathematica Interface – Usage

Once the changes to `run.F` are made, the program `run` is compiled as usual:

```
./configure
make
```

It is then loaded in Mathematica with

```
Install["run"]
```

Now a Mathematica function of the same name, `run`, is available. There are two ways of invoking it:

Compute a differential cross-section at $\sqrt{s} = \mathtt{sqrtS}$:

```
run[sqrtS, arg1, arg2, ...]
```

Compute a total cross-section for $\mathtt{sqrtSfrom} \leqslant \sqrt{s} \leqslant \mathtt{sqrtSto}$:

```
run[{sqrtSfrom, sqrtSto}, arg1, arg2, ...]
```

Mathematica Interface – Data Retrieval

The output of the function `run` is an integer which indicates how many records have been transferred. For example:

```
Para[1] = {TB -> 5., MA0 -> 250.}
Data[1] = {DataRow[{500.}, {0.0539684, 0.}, {2.30801 10^-21, 0.}],
           DataRow[{510.}, {0.0515943, 0.}, {4.50803 10^-22, 0.}]}
```

`Para` contains the parameters exported from the Fortran code.
`Data` contains:

- **the independent variables,**
  here e.g. $\{500.\} = \{\sqrt{s}\}$,
- **the cross-sections,**
  here e.g. $\{0.0539684,\ 0.\} = \{\sigma^{\text{tree}}_{\text{tot}}, \sigma^{\text{1-loop}}_{\text{tot}}\}$, and
- **the integration errors,**
  here e.g. $\{2.30801\ 10\hat{}{-21},\ 0.\} = \{\Delta\sigma^{\text{tree}}_{\text{tot}}, \Delta\sigma^{\text{1-loop}}_{\text{tot}}\}$.

Code-generation Functions

FormCalc's code-generation functions are now public and disentangled from the rest of the code. They can be used to **write out an arbitrary Mathematica expression as optimized Fortran code:**

- `handle = OpenFortran["file.F"]`
  opens *file.F* as a Fortran file for writing,
- `WriteExpr[handle, {var -> expr, ...}]`
  writes out Fortran code which calculates *expr* and stores the result in *var*,
- `Close[handle]`
  closes the file again.

Code generation

- **Expressions too large** for Fortran are split into parts, as in

```
var = part1
var = var + part2
...
```

- **High level of optimization,** e.g. common subexpressions are pulled out and computed in temporary variables.
- **Many ancillary functions,** e.g.

  `PrepareExpr`, `OnePassOrder`, `SplitSums`,
  `$Prefix`, `CommonDecl`, `SubroutineDecl`, etc.

  make code generation versatile and highly automatable, such that the resulting code needs few or no changes by hand.

Scripting Mathematica

Efficient batch processing with Mathematica:

Put everything into a script, using sh's Here documents:

```
#! /bin/sh ................... Shell Magic
math << \_EOF_ ............. start Here document (note the \)
  << FeynArts`
  << FormCalc`
  top = CreateTopologies[...];
  ...
_EOF_ ...................... end Here document
```

Everything between "<< *tag*" and "*tag*" goes to Mathematica as if it were typed from the keyboard.

Note the "\\" before *tag*, it makes the shell pass everything literally to Mathematica, without shell substitutions.

Scripting Mathematica

- Everything contained in **one compact shell script,** even if it involves several Mathematica sessions.
- Can combine with arbitrary shell programming, e.g. can use **command-line arguments** efficiently:

```
#! /bin/sh
math -run "arg1=$1" -run "arg2=$2" ... << \END
   ...
END
```

- Can easily be **run in the background,** or combined with utilities such as **make.**

Debugging hint: `-x` **flag** makes shell echo every statement,

```
#! /bin/sh -x
```

Three Levels of Fields

Generic level, e.g. `F, F, S`

$$C(F_1, F_2, S) = G_-\omega_- + G_+\omega_+$$

Kinematical structure completely fixed, most algebraic simplifications (e.g. tensor reduction) can be carried out.

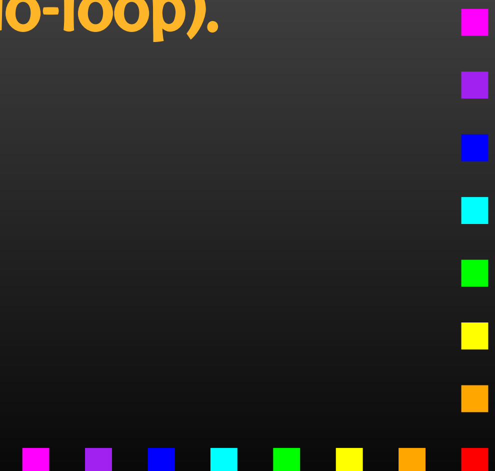
Classes level, e.g. `-F[2], F[1], S[3]`

$$\bar{\ell}_i \nu_j G: \quad G_- = -\frac{\mathrm{i}\, e\, m_{\ell,i}}{\sqrt{2}\sin\theta_w M_W}\delta_{ij}\,, \quad G_+ = 0$$

Coupling fixed except for i, j (can be summed in do-loop).

Particles level, e.g. `-F[2,{1}], F[1,{1}], S[3]`

insert fermion generation (1, 2, 3) for i and j



The Model Files

One has to set up, once and for all, a

- **Generic Model File** (seldomly changed) containing the generic part of the couplings,

Example: the FFS coupling

$$C(F, F, S) = G_- \omega_- + G_+ \omega_+ = \vec{G} \cdot \begin{pmatrix} \omega_- \\ \omega_+ \end{pmatrix}$$

```
AnalyticalCoupling[s1 F[j1, p1], s2 F[j2, p2], s3 S[j3, p3]]
== G[1][s1 F[j1], s2 F[j2], s3 S[j3]] .
     { NonCommutative[ ChiralityProjector[-1] ],
       NonCommutative[ ChiralityProjector[+1] ] }
```

The Model Files

One has to set up, once and for all, a

- **Classes Model File** (for each model) declaring the particles and the allowed couplings

Example: the $\bar\ell_i \nu_j G$ coupling in the Standard Model

$$\vec{G}(\bar\ell_i, \nu_j, G) = \begin{pmatrix} G_- \\ G_+ \end{pmatrix} = \begin{pmatrix} -\dfrac{\mathrm{i}\, e\, m_{\ell,i}}{\sqrt{2}\sin\theta_w M_W}\delta_{ij} \\ 0 \end{pmatrix}$$

```
C[ -F[2,{i}], F[1,{j}], S[3] ]
== { {-I EL Mass[F[2,{i}]]/(Sqrt[2] SW MW) IndexDelta[i, j]},
     {0} }
```

Tweaking Model Files

Or, How to efficiently make changes in an existing model file.

Bad: Copy the model file, modify the copy. – Why?

- It is typically not very transparent what has changed.
- If the original model file changes (e.g. bug fixes), these do not automatically propagate into the derivative model file.

Better: Create a new model file which reads the old one and modifies the particles and coupling tables.

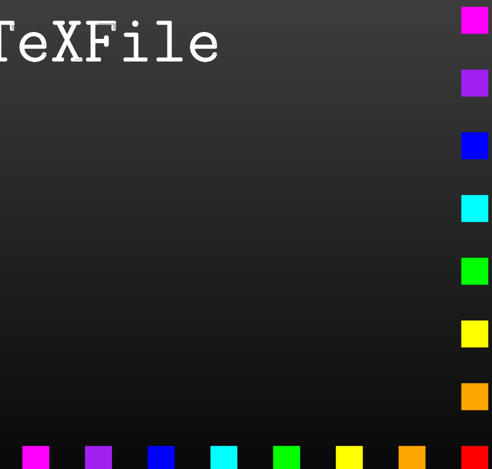
- `M$ClassesDescription` = list of particle definitions,
- `M$CouplingMatrices` = list of couplings.

Tweaking Model Files

Example: Introduce enhancement factors for the b-$\bar{b}$-h_0 and b-$\bar{b}$-H_0 Yukawa couplings in the MSSM.

```
LoadModel["MSSM"]

EnhCoup[ (lhs:C[F[4,{g_,_}], -F[4,_], S[h:1|2]]) == rhs_ ] :=
  lhs == Hff[h,g] rhs

EnhCoup[other_] = other

M$CouplingMatrices = EnhCoup/@ M$CouplingMatrices
```

To see the effect, make a printout with the `WriteTeXFile` **utility of FeynArts.**



Not the Cross-Section

Or, How to get things the Standard Setup won't give you.

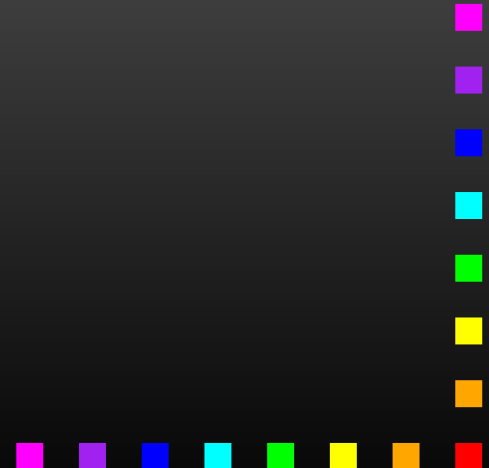
Example: extract the Wilson coefficients for $b \to s\gamma$.

```
tops = CreateTopologies[1, 1 -> 2]
ins = InsertFields[tops, F[4,{3}] -> {F[4,{2}], V[1]}]
vert = CalcFeynAmp[CreateFeynAmp[ins], FermionChains -> Chiral]

mat[p_Plus] := mat/@ p

mat[r_. DiracChain[s2_Spinor, om_, mu_, s1:Spinor[p1_, m1_, _]]] :=
  I/(2 m1) mat[r DiracChain[sigmunu[om]]] +
  2/m1 r Pair[mu, p1] DiracChain[s2, om, s1]

mat[r_. DiracChain[sigmunu[om_]], SUNT[Col1, Col2]] :=
  r O7[om]/(EL MB/(16 Pi^2))

mat[r_. DiracChain[sigmunu[om_]], SUNT[Glu1, Col2, Col1]] :=
  r O8[om]/(GS MB/(16 Pi^2))

coeff = Plus@@ vert //. abbr /. Mat -> mat

c7 = Coefficient[coeff, O7[6]]
c8 = Coefficient[coeff, O8[6]]
```

Not the Cross-Section

Using FormCalc's output functions it is also pretty straightforward to **generate your own Fortran code:**

```
file = OpenFortran["bsgamma.F"]

WriteString[file,
  SubroutineDecl["bsgamma(C7,C8)"] <>
  "\tdouble complex C7, C8\n" <>
  "#include \"looptools.h\"\n"]

WriteExpr[file, {C7 -> c7, C8 -> c8}]

WriteString[file, "\tend\n"]

Close[file]
```



Summary and Outlook

- Serious perturbative calculations these days can generally no longer be done by hand:
  - Required accuracy, Models with many particles, ...
- Hybrid programming techniques are necessary:
  - Computer algebra is an indispensable tool because many manipulations must be done symbolically.
  - Fast number crunching can only be achieved in a compiled language.
- Software engineering and further development of the existing packages is a must:
  - As we move on to ever more complex computations (more loops, more legs), the computer programs must become more "intelligent," i.e. must learn all possible tricks to still be able to handle the expressions.

Finally

Using FeynArts and FormCalc is a lot like driving a car:

- You have to decide where to go (this is often the hardest decision).
- You have to turn the ignition key, work gas and brakes, and steer.
- But you don't have to know, say, which valve has to open at which time to keep the motor running.
- On the other hand, you can only go where there are roads. You can't climb a mountain with your car.