New Features in FeynArts & Friends, and how they got used in FeynHiggs

Thomas Hahn

Max-Planck-Institut für Physik
München

Package Types

‘Production’	‘Exploration’	‘Specific’
		
MG5_aMC@NLO GoSam OpenLoops	FormCalc FeynCalc Package-X	FeynHiggs DarkSUSY Prospino

One-loop since mid-1990s

Automated NLO computations is an industry today, with many packages becoming available in the last decade.

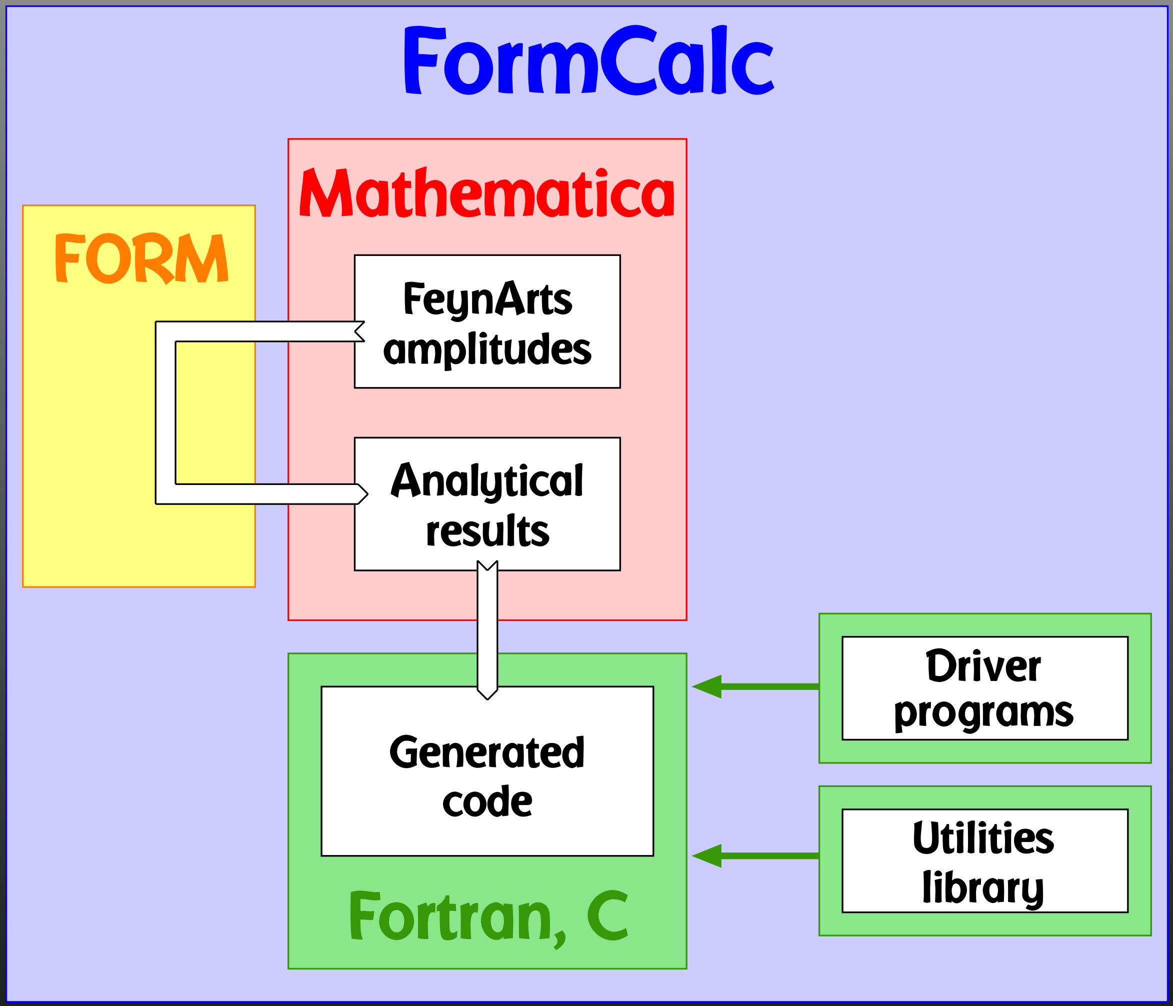
Here: FeynArts (1991) + FormCalc (1995)

FormCalc was doing largely the same as FeynCalc (1992) but used FORM for the time-consuming tasks, hence the name FormCalc.

- Feynman-diagrammatic method,
- Analytic calculation as far as possible ('any' model),
- Generation of code for the numerical evaluation of the squared matrix element.

FeynArts + FormCalc also used as 'engine' in SARAH, SloopS.

FormCalc Evaluation Scheme



FeynHiggs est omnis divisa in partes tres

- **Code hand-written for FeynHiggs**
  The 'back bone': structural code, utility functions, contributions taken from literature
- **Code generated from external expressions**
  (Large) Mathematica expressions from various sources: 2L Higgs SEs, EFT ingredients, $g_\mu - 2$, 2L parts of Δr, etc.
- **Code generated from calculations done for FeynHiggs**
  Everything in the 'gen' subdirectory of FeynHiggs.
  Full control over model content, particle selection, resummations/K-factors, renormalization prescription, etc.

Improvements in Code Generation

- Before 2.14.3: entire renormalization hard-coded.
  Now: counter-terms + ren. const. taken from model file.
- 1L SEs automatically split into parts: $t/\tilde{t}$; $+b/\tilde{b}$; $f/\tilde{f}$; all.
  Sectors of the MSSM can be looked at even in the presence of a generated renormalization.
- Generator to a certain extent model-aware, e.g.
  ▷ knows relevant flags, e.g. `$MHpInput` (H input mass),
  ▷ knows how to simplify 2×2 Sf mixing.
- Not a 'generator generator' approach, i.e. even if scripts ran (or were modified to run) with 'arbitrary' model file, the produced code would still need to be embedded in and called from the main program.

Declarations + Code in One File

`DeclIf` → `"`*var*`"` **(option of `WriteExpr`)**

Inserts preprocessor statements of the form

```
#ifndef var
#define var
...declarations...
#else
...code...
#endif
```

Usage: include resulting file twice.

Solves problem of declaration order, e.g. when including several generated files or with inline function definitions.

`File/SubroutineIncludes` **correctly handled for Fortran, C.**

Temporary Variables

`ToVars[`*patt*`,`*name*`][`*expr*`]` **introduces variables '*nameNNN*' for subexpressions matching** *patt*

`MakeTmp` $\to$ `ToVars[`*patt*`,`*name*`]` **(option of `PrepareExpr`)**

Introduces variables for specific objects for

- **better performance (variable hoisting) and/or**
- **easier debugging (combine with `DebugLines`, `$DebugCmd`).**

Example:

`WriteExpr[`*expr*`, MakeTmp` $\to$ `ToVars[LoopIntegrals,Head]]`

Improved Abbreviations

`Abbreviate[`*expr*`,` *level*`]` - unchanged
`Abbreviate[`*expr*`,` *func*`]` - *func*`[`x`]` = `True,False` (old)
= subexpr of x (new)

`Abbr[]`, `Subexpr[]` - now with patterns on l.h.s.
so that they can be used in Mathematica

old:

```
Sub333[Gen5] → A0[Mf2[2,Gen5]] - ...
```

new:

```
Sub333[Gen5_] → A0[Mf2[2,Gen5]] - ...
```

Finding Dependencies

`FindDeps[`*list*, *patt*`]` **finds all variables in** *list* **whose r.h.s. directly or indirectly depends on** *patt*.

Example:

```
list = {a → x,
        b → 2,
        c → 3 + a,
        d → b + c}

FindDeps[list, x] → {a, c, d}
```

Named Array Indices

`Enum[`*ind*`]` **associates indices *ind* with integers.**

`ClearEnum[]` **clears all `Enum` associations.**

Named array indices enhance readability, `Enum` needed to correctly determine array dimensions.

Example:

```
Enum["h0h0", "HHHH", "A0A0", "HmHp",
     "h0HH", "h0A0", "HHA0"]
```

Persistent Names for Generic Objects

Generic amplitude contains objects not acceptable to FORM, e.g. `G[`*sym*`][`*cto*`][`*fi*`][`*kin*`]` (generic coupling).

`CalcFeynAmp` must substitute generic objects by symbols.

So far: ad hoc introduction of numbered symbols, e.g. "`Coupling5`," not consistent outside one FormCalc session.

Now: portable name-mangling allows to generate generic 'building blocks' for applications, but produces names like "`GV1VbtVbbg12Kp3g23Pq1g13kQ2`."

Propagator-Dependent Masses and Vertices

FeynArts allows masses and couplings that depend on the propagator type, usually to distinguish loop from non-loop particles.

Example:
a) particle description:

```
S[1] == { ..., Mass → Mh0tree,
               Mass[Loop] → Mh0, ...}
```

b) coupling definition:

C[S[1,type1], S[2,type2], S[2,type3]] == *coupling*

Caveat: `type1,2,3` **placeholders, not literals.**

Changes for Mixing Fields

Mixing Fields propagate as themselves but couple as their left and right partners. Example: G^0-Z, $G^\pm$-$W^\pm$ mixing in the SM in a non-Feynman gauge.

So far: representation at

- Generic level: `Mix[`g`,`g'`]` forward, `Rev[`g`,`g'`]` backward,
- Classes level: `Mix[`g`,`g'`]` forward, `2 Mix[`g`,`g'`]` backward.

This lead to inconsistencies (too many/few diagrams) so that now the reversed field is represented by `Rev[`g`,`g'`]` also at Classes level.

Need to review/adapt model files which contain mixing fields.

Mixed Precision in One Code

Numerical stability of FeynHiggs generally satisfactory but e.g. non-degenerate 2L EFT threshold corrections exhibit numerical artifacts even in not-too-extreme scenarios.

Available for long: `./configure --quad`
all-out quad precision simple to realize (compiler flags) but vastly slower, plus the API changes.

Want to use higher precision only for neuralgic parts.

Need to address:

- Types for real, complex.
- Number literals.
- Name mangling.

Mixed Precision in One Code

Currently in FeynHiggs + being implemented in LoopTools:
"Poor man's template programming"

```
#if REALSIZE == 16
#  define RealSize 16
#  define ComplexSize 32
#  define RealSuffix Q
#elif REALSIZE == 10
#  define RealSize 10
#  define ComplexSize 20
#  define RealSuffix T
#else
#  define RealSize 8
#  define ComplexSize 16
#  define RealSuffix D
#endif
```

Mixed Precision in One Code

Types and Conversion Functions:

```
#define RealType real*RealSize
#define ComplexType complex*ComplexSize

#define Re(c) real(c,kind=RealSize)
#define Im(c) imag(c)
#define Conjugate(c) conjg(c)
#define ToComplex(c) cmplx(c,kind=RealSize)
#define ToComplex2(r,i) cmplx(r,i,kind=RealSize)
```

Mixed Precision in One Code

Name mangling (tested on gfortran, ifort, pgf):

```
#define _id(s) s
#define ComplexSuffix _id(C)RealSuffix
#define _R(s) _id(s)RealSuffix
#define _C(s) _id(s)ComplexSuffix
#define N(n) _id(n)_id(_)RealSize
#define Frac(n,d) (real(n,kind=RealSize)/(d))
```

Identical source compiles into different-precision versions at the switch of a preprocessor flag.

Mixed Precision in One Code

Can likewise unify code if only arg type (real/complex) differs:

```
#if COMPLEXARGS
#  define ArgType ComplexType
#  define ArgQuad ComplexQuad
#  define ArgSuffix ComplexSuffix
#  define ArgLen 2
#else
#  define ArgType RealType
#  define ArgQuad RealQuad
#  define ArgSuffix RealSuffix
#  define ArgLen 1
#endif

#define _A(s) _id(s)ArgSuffix
```

Mixed Precision in One Code

Examples of actual code:

```
subroutine _A(Bcoeff)(B, args)
implicit none
ComplexType B(*)
ArgType args(*)
...
```

Example of literals with N (Bernoulli numbers):

```
  data bf /
&   N(-.25),
&   N(.0277777777777777777777777777777777778),
&   N(-.0277777777777777777777777777777777778e-2),
&   N(4.7241118669690098261526832955404384e-6),
&   N(-9.1857730746619635508524397413286302 2e-8), ... /
```

Example of Frac:

```
  B(bb001) = Frac(1,8)*( 2*m1*B(bb1) - a0(2) +
&   (p + dm)*(B(bb11) + Frac(1,6)) - Frac(1,2)*(m1 + m2) )
```

Summary

Many small functions/additions to FeynArts, FormCalc, & LoopTools, mostly triggered by FeynHiggs development.

Together significant improvements, in particular in code generation:

- **Convenience of Code Generation:**
  `DeclIf`, `Enum`, `ClearEnum`
- **Variable/Abbreviation handling:**
  `ToVars`, `MakeTmp`, `Abbreviate`
- **Generic Amplitudes:**
  persistent names, propagator-type-dependent particle properties, mixing fields
- **Mixing precision** within the same code