Ännu krämigare, underbart gott.
Kavli
ORIGINALET SEDAN 1924
Räk & KosmoOst
KRÄMIG MJUKOST GJORD PÅ UTVALDA KLASSISKA OSTAR, ISHAVSRÄKOR OCH HANDSKALADE KRÄFTOR
275g
16% FETT

New School YAML Rules (& Observables)

The GAMBIT YAML Rules you've needed in a tube for years but didn't know.

Pat Scott GAMBIT XV, KICC, Cambridge, July 12 2023

Q. What has **6 years of use** shown is **wrong** with the current system?

Q. What has **6 years of use** shown is **wrong** with the current system?

1. You must write a separate `ObsLikes` entry for every module function that should be targeted by a scan.

Q. What has **6 years of use** shown is **wrong** with the current system?

1. You must write a separate `ObsLikes` entry for every module function that should be targeted by a scan.
2. Ill-formed fields in `Rules` and `ObsLikes` are usually silently ignored.

Q. What has **6 years of use** shown is **wrong** with the current system?

1. You must write a separate `ObsLikes` entry for every module function that should be targeted by a scan.
2. Ill-formed fields in `Rules` and `ObsLikes` are usually silently ignored.
3. Unused `Rules` are not reliably detected (some are, some aren't).

Q. What has **6 years of use** shown is **wrong** with the current system?

1. You must write a separate `ObsLikes` entry for every module function that should be targeted by a scan.
2. Ill-formed fields in `Rules` and `ObsLikes` are usually silently ignored.
3. Unused `Rules` are not reliably detected (some are, some aren't).
4. The logic of what is meant by a `Rule` is a bit ad hoc, and open to misinterpretation:

Q. What has **6 years of use** shown is **wrong** with the current system?

1. You must write a separate `ObsLikes` entry for every module function that should be targeted by a scan.
2. Ill-formed fields in `Rules` and `ObsLikes` are usually silently ignored.
3. Unused `Rules` are not reliably detected (some are, some aren't).
4. The logic of what is meant by a `Rule` is a bit ad hoc, and open to misinterpretation:

```yaml
ObsLikes:
  # Quiz: does this tell the dep resolver to use ColliderBit::calc_HS_LHC_LogLike when
  # capability LHC_Higgs_LogLike is required, or does it just specify an option to pass
  # to ColliderBit::calc_HS_LHC_LogLike *if* it is used in a given scan?
  - capability: LHC_Higgs_LogLike
    module: ColliderBit
    function: calc_HS_LHC_LogLike
    options:
      foo: "bar"
```

Q. What has **6 years of use** shown is **wrong** with the current system?

5. They are not very flexible, so some things that you might expect to work just don't:

```
ObsLikes:
  # Nope. Always need to specify "capability".
  - function: calc_HS_LHC_LogLike
    purpose: LogLike

Rules:
  # Nope. Need a "capability" for that "backends" entry too
  - capability: my_capability
    function: my_function
    backends:
      - {backend: my_backend, version: 0.0}

  # Nopity nope. Sorry, no way.
  - backend: DDCalc
    version: 2.3.0
```

Using !match_all fixes issue 1

You can now select multiple module functions to include in a scan with one `ObsLikes` entry.

From `yaml_files/QCDAxions.yaml`:

```yaml
ObsLikes:
  - !match_all
    capability: lnL_CAST.*
    purpose: LogLike
```

This matches both capability `lnL_CAST2007` and capability `lnL_CAST2017`. One function matches each capability, so two module functions get included in the likelihood function.

Note that regex is allowed in all `ObsLikes` and `Rules` now! It can help when using `!match_all`, but it isn't required.

Explicit rules

New-style rules come with explicit `if` and `then` clauses:

```yaml
# Matches old-style rule's behaviour.
- if:
    capability: LHC_Higgs_LogLike
  then:
    module: ColliderBit
    function: calc_HS_LHC_LogLike
    options:
      foo: "bar"
```

```yaml
# New behaviour not previously possible.
- if:
    # Look mum, no capability
    module: ColliderBit
    function: calc_HS_LHC_LogLike
  then:
    options:
      foo: "bar"
```

Fixes 4: The logic of what is meant by a `Rule` is a bit ad hoc, and open to misinterpretation.
5: They are not very flexible, so some things that you might expect to work just don't.

Explicit rules

New-style rules come with explicit `if` and `then` clauses:

```
Rules:


  # Oooh yeah. 😎
  - if:
      backend: DDCalc
    then:
      version: 2.3.0
```

Fixes 5: They are not very flexible, so some things that you might expect to work just don't.

Compilation of `Rules` and `ObsLikes`

GAMBIT Core now compiles all `Rules` and `ObsLikes` from YAML into instances of new C++ classes.

- `Observable`, `ModuleRule` or `BackendRule`.
- Checks *every* field of *every* entry in `Obslikes` and `Rules` section for validity.
- The `dependencies` field now contains nested `ModuleRule` instances.
- The `backends` field now contains nested `BackendRule` instances.
- Rules log which functions matched them at dep resolution time
  $\longrightarrow$ foolproof checking that all `Rules` are used.

Fixes 2: Ill-formed fields in `Rules` and `ObsLikes` are usually silently ignored.
3: Unused `Rules` are not reliably detected (some are, some aren't).

Compilation of `ObsLikes`

Table: Fields permitted in `ObsLikes` entries of a GAMBIT YAML file. All strings may contain regular expressions (regex). From the GAMBIT 2 paper draft.

Matching field	Value Type	Required?
`capability:`	`string`	At least one of these is required.
`type:`	`string`	
`function:`	`string`	
`module:`	`string`	
`functionChain:`	`[string,string,...]`	Optional
`!match_all`	N/A (Tag)	Optional
Modifier field	**Value Type**	**Required?**
`purpose:`	`string`	Required
`sub_capabilities:`	YAML Node	Optional
`printme:`	`boolean`	Optional
`dependencies:`	Module rule(s)	Optional
`backends:`	Backend rule(s)	Optional

Backwards compatibility: implicit conversions to new Rules

All *compiled* rules now have `if` and `then` clauses.

But they will be implicitly constructed from old-style rules without `if` and `then`:

```yaml
Rules:
  - capability: A
    type: B
    function: Cfunc
    module: ExampleBit
```

=

```yaml
Rules:
  - if:
      capability: A
      type: B
    then:
      function: Cfunc
      module: ExampleBit
```

- Means most of your existing YAML files will work fine
- But that's not an excuse to be lazy – write explicit rules in future, they're much clearer, safer and more powerful!

Compilation of `Rules` $\longrightarrow$ `ModuleRule`

Table: Fields permitted in module rules built from `Rules` entries of a GAMBIT YAML file. All strings may contain regular expressions (regex). All fields are optional, but at least one field is required in each of the `if` and `then` blocks. From the GAMBIT 2 paper draft.

Matching Field	Value Type	OK in `if` block?	OK in `then` block?	Implicit conversion
`capability:`	`string`	Yes	Yes	`if`
`type:`	`string`	Yes	Yes	`if`
`function:`	`string`	Yes	Yes	`then`
`module:`	`string`	Yes	Yes	`then`
`functionChain:`	`[string,string,...]`	No	Yes	`then`
Modifier Field	**Value Type**	**OK in `if` block?**	**OK in `then` block?**	**Implicit conversion**
`options:`	YAML Node	No	Yes	`then`
`dependencies:`	Module rule(s)	No	Yes	`then`
`backends:`	Backend rule(s)	No	Yes	`then`
`!weak`	N/A (Tag)	No	No	N/A

Compilation of `Rules` $\longrightarrow$ `BackendRule`

Table: Fields permitted in backend rules built from `Rules` entries of a GAMBIT YAML file. All strings may contain regular expressions (regex). All fields are optional, but at least one field is required in each of the `if` and `then` blocks. The implicit conversion of the `capability` field depends on the presence of the `group` field: if the `group` field is present, `capability` is implicitly converted to a member of the `then` block; if `group` is absent, `capability` is implicitly converted to a member of the `if` block. From the GAMBIT 2 paper draft.

Matching Field	Value Type	OK in `if` block?	OK in `then` block?	Implicit conversion
`capability:`	**`string`**	Yes	Yes	depends on `group`
`type:`	**`string`**	Yes	Yes	`if`
`function:`	**`string`**	Yes	Yes	`then`
`version:`	**`string`**	Yes	Yes	`then`
`backend:`	**`string`**	Yes	Yes	`then`
`group:`	**`string`**	Yes	No	`if`
Modifier Field	Value Type	OK in `if` block?	OK in `then` block?	Implicit conversion
`!weak`	N/A (Tag)	No	No	N/A

So what *doesn't* work any longer?

```
Rules:
  - capability: A
    function: B
    backends:
      {backend: C }
```

$\longrightarrow$

```
Rules:
  - capability: A
    function: B
    backends:
      {backend: C, group: D }
```

or

```
Rules:
  - capability: A
    function: B
    backends:
      - if:
          group: D
        then:
          backend: C
```

So what *doesn't* work any longer?

```
Rules:
  - options:
      option1: A
      option2: B
```

$\longrightarrow$

```
Rules:
  - if:
      function: any
    then:
      options:
        option1: A
        option2: B
```

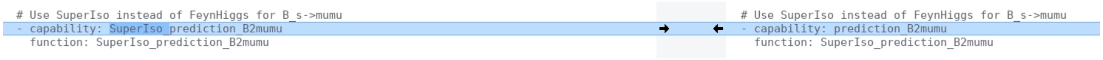
```
Rules:
  - module: ExampleBit
    options:
      option1: A
      option2: B
```

$\longrightarrow$

```
Rules:
  - if:
      module: ExampleBit
    then:
      options:
        option1: A
        option2: B
```

And what ***bugs*** did the new system find in existing YAML files?

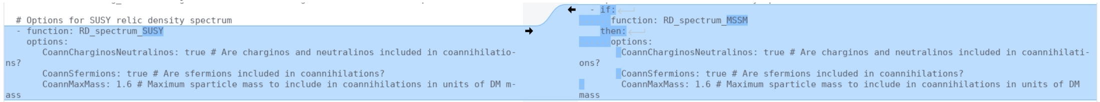
`CMSSM.yaml`, `FlavBit_CMSSM.yaml`, `MSSM7.yaml`, `MSSM9.yaml`, `NUHM1.yaml`, `NUHM2.yaml`:



```
  # Use SuperIso instead of FeynHiggs for B_s->mumu
  - capability: SuperIso_prediction_B2mumu
    function: SuperIso_prediction_B2mumu
```

```
  # Use SuperIso instead of FeynHiggs for B_s->mumu
  - capability: prediction_B2mumu
    function: SuperIso_prediction_B2mumu
```

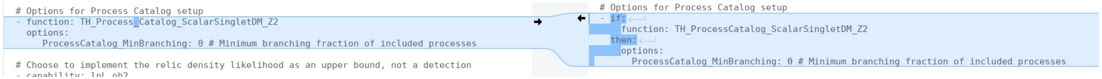
`DarkBit_MSSM7.yaml`:



```
  # Options for SUSY relic density spectrum
  - function: RD_spectrum_SUSY
    options:
       CoannCharginosNeutralinos: true # Are charginos and neutralinos included in coannihilatio-
ns?
       CoannSfermions: true # Are sfermions included in coannihilations?
       CoannMaxMass: 1.6 # Maximum sparticle mass to include in coannihilations in units of DM m-
ass
```

```
  - if:
      function: RD_spectrum_MSSM
    then:
      options:
        CoannCharginosNeutralinos: true # Are charginos and neutralinos included in coannihilati-
ons?
        CoannSfermions: true # Are sfermions included in coannihilations?
        CoannMaxMass: 1.6 # Maximum sparticle mass to include in coannihilations in units of DM
mass
```

`DarkBit_ScalarSingletDM_Z2.yaml`:

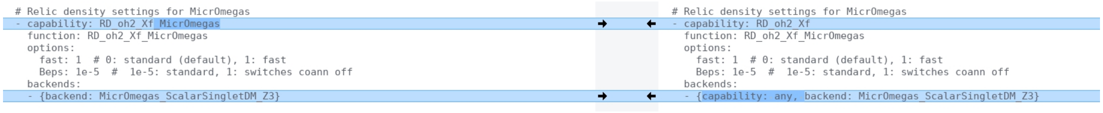


```
  # Options for Process Catalog setup
  - function: TH_Process_Catalog_ScalarSingletDM_Z2
    options:
       ProcessCatalog_MinBranching: 0 # Minimum branching fraction of included processes

  # Choose to implement the relic density likelihood as an upper bound, not a detection
  - capability: lnL_oh2
```

```
  # Options for Process Catalog setup
  - if:
      function: TH_ProcessCatalog_ScalarSingletDM_Z2
    then:
      options:
        ProcessCatalog_MinBranching: 0 # Minimum branching fraction of included processes
```

`ScalarSingletDM_Z3.yaml`:



```
  # Relic density settings for MicrOmegas
  - capability: RD_oh2_Xf_MicrOmegas
    function: RD_oh2_Xf_MicrOmegas
    options:
      fast: 1  # 0: standard (default), 1: fast
      Beps: 1e-5  #  1e-5: standard, 1: switches coann off
    backends:
    - {backend: MicrOmegas_ScalarSingletDM_Z3}
```

```
  # Relic density settings for MicrOmegas
  - capability: RD_oh2_Xf
    function: RD_oh2_Xf_MicrOmegas
    options:
      fast: 1  # 0: standard (default), 1: fast
      Beps: 1e-5  #  1e-5: standard, 1: switches coann off
    backends:
    - {capability: any, backend: MicrOmegas_ScalarSingletDM_Z3}
```

Where to go for more info

- This is in the master as of Monday morning.
- It's written up in full in the GAMBIT 2 paper draft at `gambit_community/Papers/R3/GAMBIT_2_0` if you want some reference material.
- Pull request #410 makes for fun reading if you really want gory details about why each aspect of the new design is the way it is. Thanks Tomás!
- I am more than happy to answer any and all questions about it, and to help resolve any issues transitioning to the new rules – whether during the meeting or after.

☕ ☕ ☕ ☕ ☕ ☕ ☕ ☕ ☕ ☕ ☕ ☕ ☕ ☕