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:

```
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:

<table>
<thead>
<tr>
<th># Matches old-style rule’s behaviour.</th>
<th># New behaviour not previously possible.</th>
</tr>
</thead>
<tbody>
<tr>
<td>- if:</td>
<td>- if:</td>
</tr>
<tr>
<td>capability: LHC_Higgs_LogLike</td>
<td># Look mum, no capability ☹</td>
</tr>
<tr>
<td>then:</td>
<td>module: ColliderBit</td>
</tr>
<tr>
<td>module: ColliderBit</td>
<td>function: calc_HS_LHC_LogLike</td>
</tr>
<tr>
<td>function: calc_HS_LHC_LogLike</td>
<td>then:</td>
</tr>
<tr>
<td>options:</td>
<td>options:</td>
</tr>
<tr>
<td>foo: &quot;bar&quot;</td>
<td>foo: &quot;bar&quot;</td>
</tr>
</tbody>
</table>

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:

```yaml
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 → 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.

<table>
<thead>
<tr>
<th>Matching field</th>
<th>Value Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>capability:</td>
<td>string</td>
<td>At least one of these is required.</td>
</tr>
<tr>
<td>type:</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>function:</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>module:</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>functionChain:</td>
<td>[string,string,...]</td>
<td>Optional</td>
</tr>
<tr>
<td>!match_all</td>
<td>N/A (Tag)</td>
<td>Optional</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifier field</th>
<th>Value Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>purpose:</td>
<td>string</td>
<td>Required</td>
</tr>
<tr>
<td>sub_capabilities:</td>
<td>YAML Node</td>
<td>Optional</td>
</tr>
<tr>
<td>printme:</td>
<td>boolean</td>
<td>Optional</td>
</tr>
<tr>
<td>dependencies:</td>
<td>Module rule(s)</td>
<td>Optional</td>
</tr>
<tr>
<td>backends:</td>
<td>Backend rule(s)</td>
<td>Optional</td>
</tr>
</tbody>
</table>
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*:

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

= *

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 → 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.

<table>
<thead>
<tr>
<th>Matching Field</th>
<th>Value Type</th>
<th>OK in <em>if</em> block?</th>
<th>OK in <em>then</em> block?</th>
<th>Implicit conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>capability:</td>
<td>string</td>
<td>Yes</td>
<td>Yes</td>
<td>if</td>
</tr>
<tr>
<td>type:</td>
<td>string</td>
<td>Yes</td>
<td>Yes</td>
<td>if</td>
</tr>
<tr>
<td>function:</td>
<td>string</td>
<td>Yes</td>
<td>Yes</td>
<td>then</td>
</tr>
<tr>
<td>module:</td>
<td>string</td>
<td>Yes</td>
<td>Yes</td>
<td>then</td>
</tr>
<tr>
<td>functionChain:</td>
<td>[string,string,...]</td>
<td>No</td>
<td>Yes</td>
<td>then</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifier Field</th>
<th>Value Type</th>
<th>OK in <em>if</em> block?</th>
<th>OK in <em>then</em> block?</th>
<th>Implicit conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>options:</td>
<td>YAML Node</td>
<td>No</td>
<td>Yes</td>
<td>then</td>
</tr>
<tr>
<td>dependencies:</td>
<td>Module rule(s)</td>
<td>No</td>
<td>Yes</td>
<td>then</td>
</tr>
<tr>
<td>backends:</td>
<td>Backend rule(s)</td>
<td>No</td>
<td>Yes</td>
<td>then</td>
</tr>
<tr>
<td>!weak</td>
<td>N/A (Tag)</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Compilation of \textit{Rules} $\rightarrow$ \textit{BackendRule}

\textbf{Table:} Fields permitted in backend rules built from \textit{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 \textit{if} and \textit{then} blocks. The implicit conversion of the \textit{capability} field depends on the presence of the \textit{group} field: if the \textit{group} field is present, \textit{capability} is implicitly converted to a member of the \textit{then} block; if \textit{group} is absent, \textit{capability} is implicitly converted to a member of the \textit{if} block. From the GAMBIT 2 paper draft.

<table>
<thead>
<tr>
<th>Matching Field</th>
<th>Value Type</th>
<th>OK in \textit{if} block?</th>
<th>OK in \textit{then} block?</th>
<th>Implicit conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>capability:</td>
<td>string</td>
<td>Yes</td>
<td>Yes</td>
<td>depends on \textit{group}</td>
</tr>
<tr>
<td>type:</td>
<td>string</td>
<td>Yes</td>
<td>Yes</td>
<td>if</td>
</tr>
<tr>
<td>function:</td>
<td>string</td>
<td>Yes</td>
<td>Yes</td>
<td>then</td>
</tr>
<tr>
<td>version:</td>
<td>string</td>
<td>Yes</td>
<td>Yes</td>
<td>then</td>
</tr>
<tr>
<td>backend:</td>
<td>string</td>
<td>Yes</td>
<td>Yes</td>
<td>then</td>
</tr>
<tr>
<td>group:</td>
<td>string</td>
<td>Yes</td>
<td>No</td>
<td>if</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifier Field</th>
<th>Value Type</th>
<th>OK in \textit{if} block?</th>
<th>OK in \textit{then} block?</th>
<th>Implicit conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{!weak}</td>
<td>N/A (Tag)</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
</tbody>
</table>
So what *doesn’t* work any longer?

- rules:
  - capability: A
  - function: B
  - backends:
    - backend: C

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

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

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

**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:

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

DarkBit_MSSM7.yaml:

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

DarkBit_ScalarSingletDM_Z2.yaml:

```yaml
# Options for Process Catalog setup
- function: TH_ProcessCatalog_ScalarSingletDM_Z2
  ProcessCatalog MinBranching: 0 # Minimum branching fraction of included processes

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

ScalarSingletDM_Z3.yaml:

```yaml
# Relic density settings for MicrOmega
- capability: RD_oh2_Xf_MicrOmega
  function: RD_oh2_Xf_MicrOmega
  options:
    fast: 1 # 0: standard (default), 1: fast
    Beps: le-5 # le-5: standard, 1: switches coann off
    backends:
      - backend: MicrOmega_ScalarSingletDM_Z3
```

---

**Note:** The YAML files and related configurations are listed, but the content is too technical and detailed to be summarized in a simple text format. The presence of specific terms like `CoannCharginosNeutralinos`, `CoannStemsions`, `CoannMaxMass`, `RD_spectrum_SUSY`, `RD_oh2_Xf_MicrOmega`, and the usage of SuperIso and FeynHiggs suggest a focus on particle physics and the analysis of dark matter and SUSY scenarios. The configuration parameters are designed to handle various physical processes, including coannihilation spectra and branching ratios, with options for fine-tuning the simulation settings.
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.