Refinement types and other werid language features for physics

Jim Pivarski

Princeton University – IRIS-HEP

May 15, 2019

Quite a few groups have been thinking about physics event processing languages, explicitly or implicitly.

- LHADA/ADL: Sezen Sekmen, Harry Prosper, Philippe Gras
- CutLang: Gokhan Unel
- IRIS-HEP Analysis Systems: Gordon Watts, Mason Proffitt, Emma Torro
- FAST-Carpenter (YAML): Benjamin Krikler
- NAIL: Andrea Rizzi
- RDataFrame: Enrico Guiraud, Danilo Piparo, and the ROOT Team
- AEACuS & RHADAManTHUS: Joel Walker (phenomenology)
- Femtocode: me, though not for several years...

see Analysis Description Languages Workshop (May 6–8)

Originally, I was going to talk about refinement types
(a core feature of Femtocode), but this talk has grown.

Refinement type

From Wikipedia, the free encyclopedia

In type theory, a **refinement type**[1][2][3] is a type endowed with a predicate which is assumed to hold for any element of the refined type. Refinement types can express preconditions when used as function arguments or postconditions when used as return types: for instance, the type of a function which accepts natural numbers and returns natural numbers greater than 5 may be written as $f : \mathbb{N} \rightarrow \{n : \mathbb{N} \mid n > 5\}$. Refinement types are thus related to behavioral subtyping.

History [edit]

The concept of refinement types was first introduced in Freeman and Pfenning's 1991 *Refinement types for ML* [1], which describes a presents a type system for a subset of Standard ML. The type system "preserves the decidability of ML's type inference" whilst still "allowing more errors to be detected at compile-time". In more recent times, refinement type systems have been developed for languages such as Haskell[4], TypeScript[5] and Scala.

Example in practice: prevent NaN at compile-time

```
x / y
```

```
femtocode.parser.FemtocodeError: Function "/" does not accept
arguments with the given types:

    /(real,
      real)

    Indeterminate form (0 / 0) is possible; constrain with if-else.

Check line:col 1:0 (pos 0):

    x / y
----^
```

Example in practice: prevent NaN at compile-time

```
if y != 0: x / y else: None
```

```
----> union(null, real)
```

Example in practice: identify impossible situations

```
x == 5 and y == 6 and x == y
```

```
femtocode.parser.FemtocodeError: Function "==" does not accept
arguments with the given types:

    ==(integer(min=5, max=5),
       integer(min=6, max=6))

    The argument types have no overlap (values can never be equal).

Check line:col 1:27 (pos 27):

    x == 5 and y == 6 and x == y
-------------------------------^
```

Example in practice: identify impossible situations

```
x == y and x == 5 and y == 6
```

```
femtocode.parser.FemtocodeError: Function "==" does not accept
arguments with the given types:

    ==(integer(min=5, max=5),
       integer(min=6, max=6))

    The argument types have no overlap (values can never be equal).

Check line:col 1:5 (pos 5):

    x == y and x == 5 and y == 6
---------^
```

It's possible to turn *all* runtime errors into compile errors

Total functional programming

From Wikipedia, the free encyclopedia

Total functional programming (also known as **strong functional programming**,[1] to be contrasted with ordinary, or *weak* functional programming) is a programming paradigm that restricts the range of programs to those that are provably terminating.[2]

Restrictions [edit]

Termination is guaranteed by the following restrictions:

1. A restricted form of recursion, which operates only upon 'reduced' forms of its arguments, such as Walther recursion, substructural recursion, or "strongly normalizing" as proven by abstract interpretation of code.[3]
2. Every function must be a total (as opposed to partial) function. That is, it must have a definition for everything inside its domain.
   - There are several possible ways to extend commonly used partial functions such as division to be total: choosing an arbitrary result for inputs on which the function is normally undefined (such as $\forall x \in \mathbb{N}. x \div 0 = 0$ for division); adding another argument to specify the result for those inputs; or excluding them by use of type system features such as refinement types.[2]

Including value sets in the type definitions lets us identify runtime conditions in the type-check.

The hard parts are recursion (structural recursion) and infinite datasets (codata), both of which can be safely excluded from physics event processing.

Even without *total* functions, it can be useful for array lengths

Awkward-Array users sometimes make this mistake:

```
(nMuons > 0) & (Muons_pt[:, 0] > 30)    # intersection of masks
```

The first mask requires events with at least one muon and the second requires the first muon to have 30 GeV, *but the selections are independent,* so the second fails at runtime when some events have no muons.

Even without *total* functions, it can be useful for array lengths

Awkward-Array users sometimes make this mistake:

```
(nMuons > 0) & (Muons_pt[:, 0] > 30)     # intersection of masks
```

The first mask requires events with at least one muon and the second requires the first muon to have 30 GeV, *but the selections are independent*, so the second fails at runtime when some events have no muons.

The right way to do it is with a single selection:

```
Muons_pt[(nMuons > 0), 0] > 30           # mask first dim, pick 0
```

Even without *total* functions, it can be useful for array lengths

Awkward-Array users sometimes make this mistake:

```
(nMuons > 0) & (Muons_pt[:, 0] > 30)     # intersection of masks
```

The first mask requires events with at least one muon and the second requires the first muon to have 30 GeV, *but the selections are independent*, so the second fails at runtime when some events have no muons.

The right way to do it is with a single selection:

```
Muons_pt[(nMuons > 0), 0] > 30           # mask first dim, pick 0
```

This error would be safer and more informative as a type error.

Minimal application of refinement types for particle collections

It would be useful for an array's type description to include bounds on its length—minimally, whether it could be empty or not.

- Some languages (e.g. Numba) already include an array's dimension in its type.
- Some functions, like integer `min`/`max` or `argmin`/`argmax`, don't have a good runtime solution for empty arrays.
- Some arrays, such as those coming from a group-by operations, can be guaranteed non-empty.
- Functional operations, like `map`, `filter`, and `joins`, transform array lengths in semi-predictable ways.

Pattern matching: like regular expressions for data structures.

Scala:

```
def pz(particle: Particle) = match particle {
    case Neutral(pt, eta, _)     => pt * sinh(eta)
    case Charged(pt, eta, _, q) => pt * sinh(eta)
}
```

Haskell:

```
pz :: (Particle particle) => particle -> Float
pz (Neutral pt eta _)   = pt * sinh(eta)
pz (Charged pt eta _ q) = pt * sinh(eta)
```

Could we use pattern matching to associate particle candidates to a given decay chain?

Something like this?

```
Higgs {
    Z1 {
        lep1, lep2 in electrons or lep1, lep2 in muons
        requiring lep1.charge != lep2.charge
    }
    Z2 {
        lep3, lep4 in electrons or lep3, lep4 in muons
        requiring lep3.charge != lep4.charge
    }
    minimizing (Z1.mass - 91)**2 + (Z2.mass - 91)**2
}
```

where the match ensures that leptons aren't double-counted in both Z's?

Or, what about this?

Behold Racket (Scheme)'s two-dimensional syntax!

```
(define (subtype? a b)
  #2dmatch
  +-----------+-----------+--------+-----------+
  |   a   b   | 'Integer  | 'Real  | 'Complex  |
  +-----------+-----------+--------+-----------+
  | 'Integer  |              #t                |
  +-----------+-----------+                    |
  | 'Real     |           |                    |
  +-----------+           +--------+           |
  | 'Complex  |        #f          |           |
  +-----------+--------------------+-----------+)
```

see Racket documentation

So maybe something like this?

The ASCII art of the decay is literally the code used to match it.

```
Higgs : fit (Z1.mass - 91)**2 + (Z2.mass - 91)**2
    |
    +--> Z1 : cut lep1.charge != lep2.charge
    |      |
    |      +--> lep1, lep2 in electrons or lep1, lep2 in muons
    |
    +--> Z2 : cut lep3.charge != lep4.charge
           |
           +--> lep3, lep4 in electrons or lep3, lep4 in muons
```

Maybe the arrows are unnecessary; maybe an indentation rule like Python's. . .

I tried out some ideas with a simple parser and interpreter

https://github.com/diana-hep/rejig/blob/master/pattern-match/define-and-run.py

Syntax:

```
pattern:    derivation* constraint* "for" multiple
derivation: CNAME "=" expression
constraint: "if" expression -> if
          | "best" expression -> best
          | "sort" expression -> sort
multiple:   single ("," single)*
single:     CNAME ("," CNAME)* "in" expression
```

and normal expression syntax (math operations, comparisons, etc.).

I tried out some ideas with a simple parser and interpreter

https://github.com/diana-hep/rejig/blob/master/pattern-match/define-and-run.py

Example:

```
higgs2e2mu = match {
    z1 = lep1 + lep2
    z2 = lep3 + lep4
    hmass = mass(z1 + z2)
    if lep1.charge != lep2.charge
    if lep3.charge != lep4.charge
    for lep1, lep2 in electrons, lep3, lep4 in muons
}
```

Complete example: same flavor and opposite flavor dileptons

```
mass(particle) = sqrt(particle.E**2 - particle.px**2 - particle.py**2 - particle.pz**2)

same_flavor(collection) = match {
    z1 = lep1 + lep2; z2 = lep3 + lep4
    hmass = mass(z1 + z2)
    if lep1.charge != lep2.charge
    if lep3.charge != lep4.charge
    sort (mass(z1) - 91)**2 + (mass(z2) - 91)**2
    for lep1, lep2, lep3, lep4 in collection
}[:1]   # at most one

higgs4e  = same_flavor(electrons)
higgs4mu = same_flavor(muons)

higgs2e2mu = match {
    z1 = lep1 + lep2; z2 = lep3 + lep4                    # hmmm... repetitive
    hmass = mass(z1 + z2)
    if lep1.charge != lep2.charge
    if lep3.charge != lep4.charge
    sort (mass(z1) - 91)**2 + (mass(z2) - 91)**2
    for lep1, lep2 in electrons, lep3, lep4 in muons     # only line that's different
}[:1]   # at most one
```

A good language construct can be used for other purposes

How about matching generator-level and reconstructed particles?

(Or matching jets from different algorithms, or computing isolation variables, etc.)

```
genreco = match {
    if delta_R(gen, reco) < 0.5
    for gen in generator, reco in reconstructed
}
```

A good language construct can be used for other purposes

How about matching generator-level and reconstructed particles?

(Or matching jets from different algorithms, or computing isolation variables, etc.)

```
genreco = match {
    if delta_R(gen, reco) < 0.5
    for gen in generator, reco in reconstructed
}
```

It's a start, but now we need to group by either `gen` or `reco` and find the best (minimum `delta_R`) in each group.

Should grouping be part of the `match` syntax or separate?

(Self) Criticisms

- This is not so much a pattern-matching syntax as a declarative looping construct.
- It's starting to look wordy, like SQL (or COBOL).
- Like SQL and COBOL, it's using special syntax where a chain of functionals would work as well.