Putting Types To Work In Perl 6

Jonathan Worthington

YAPC::Europe 2008

Types

What Are Types?

- There's More Than One Way To Define It (TMTOWTDI for theorists ;-))
- A common definition: a type classifies a value (e.g. `42` is an integer, `"monkey"` is a string...)
- Another definition: a type defines the representation of and set of operations that can be performed on a value

Perl and Types

- In Perl 5, we've not tended to think so much about types
- Of course, there are various container types: scalars, arrays, hashes…
- Perl 6 introduces a much richer type system
- You can use it to help you write safer and possibly faster code, or you can ignore it and it will stay out of the way

.WHAT

Values Know Their Types

- All values can be interrogated to find out what their type is (including those stored in variables) using the .WHAT macro
- Returns a proto-object (empty instance of the class), which stringifies to the short name of the type

```
say 42.WHAT;                    # Int
say "OH HAI".WHAT;              # Str
say (1,2,3).WHAT;               # List
say ("Beer" | "Vodka").WHAT;    # Junction
```

Values Know Their Types

- If you have defined your own class, then it returns the proto-object for that

```
class Place {
    has $.name;
    has $.population is rw;
}
my $joppa = Place.new(
    name => 'Joppa',
    population => 10000
);
say $joppa.WHAT;                # Place
say $joppa.name.WHAT;           # Str
say $joppa.population.WHAT; # Int
```

Type Annotations & Constraints

Type Constraints

- Perl 6 allows you to specify a type when declaring a variable

```
my Int $answer;
```

- The writing of types on variables is called a type annotation
- Results in constraining what may be stored in the variable

```
$answer = 42;        # Assignment succeeds
$answer = "Ruby";    # Type check exception
                     # is thrown
```

Type Constraint Enforcement

- Perl 6 promises that the type constraints will be enforced at runtime at latest
- However, in cases where compilers can determine that an operation will always fail due to type constraints at compile time, it may give an error then too
- There will probably be various pragmas (maybe not in 6.0.0) that let you trade in dynamism for more compile-time checks

Where To Use Annotations

- Type constraints can be applied to
  - Variables (as shown already)
  - Parameters on subs/methods (in the future, on the return type too)

```
sub add(Int $x, Int $y) { ... }
```

  - Attributes in a class

```
class Place {
    has Str $.name;
    has Int $.population is rw;
}
```

Things that can act as type constraints

Classes

- When you declare a class, you can use its name as a type constraint
- You can then only assign things in an "isa" relationship with that class to the variable
- That is, you can assign an instance of a subclass of the variable
- Safe because it can do at least the operations of the parent class

Classes – Example

- Declare some classes

```
class Weapon { }
class AntiAircraftGun is Weapon { }
class Cannon is Weapon { }
class Chimp { }
```

- Results of various assignments

```
my Weapon $x;
$x = AntiAircraftGun.new();   # ok
$x = Cannon.new();            # ok
my Cannon $y = Cannon.new(); # ok
$y = Weapon.new();            # exception
$y = Chimp.new();             # exception
```

Chimps Are Not Cannons

~~isa~~

A Common Idiom

- When you declare that a variable is of the type of a certain class, then an undefined value of it is the proto-object for that class
- The .= operator calls a method on the LHS and then assigns the return value of the method call
- Together, we can instantiate a Chimp:

```
my Chimp $charlie .= new();
```

Roles

- You can use roles as type constraints also
- In this case, assignments will only succeed if what is being assigned does the role

```
role Amuse { }
class Chimp does Amuse { }
class SoftDrinksShop { }
my Amuse $thingy = Chimp.new(); # ok
$thingy = new SoftDrinksShop(); # exception
```

A Chimp does Amuse…

…but a soft drinks shop does not…

A pint of ale, please.

We are not amused.

lolz, we only has the coke!

Object Orientation, The Perl 6 Way

…usually.

Refinement Types

- As well as classes and roles, you can also introduce refinement types using the **`subset`** keyword
- These take an existing type (possibly another refinement type) and add some additional constraints

```
subset EvenInt of Int where
    { $^n % 2 == 0 };
my EvenInt $n = 42;              # ok
$n = 13;                         # exception
```

Refinement Types

- Can declare anonymous refinement types too
- For example, we can make a sub take two lists of the same length

```
sub mix(List $a, List $b
        where { $a.elems == $b.elems }) {
    ...
}
mix([1,2,3], [4,5,6]); # ok
mix([1,2,3], [4,5]);   # exception
```

Multiple Dispatch

TMTOWT(Dispatch)I

- In Perl 6, you can introduce multiple routines (subs, methods) with the same name, but taking a different number or different types of parameters
- At dispatch time, the parameters you are calling the routine with are used to decide which is the best one to call
- Warning: implementation of this is currently a work in progress

Arity

- You use the **`multi`** keyword to specify that a sub or method does multiple dispatch (otherwise, you'd get a redefinition warning/error)

```
multi sub foo($a) { say "1 arg" }
multi sub foo($a, $b) { say "2 args" }
foo(42);      # 1 arg
foo(39, 3);   # 2 args
```

- You can drop the **`sub`** if you wish

```
multi foo($a, $b, $c) { say "3 args" }
```

Type

- As well as arity, you can distinguish multi variants by having parameters of different types

```
class Paper {}
class Scissors {}
class Stone {}
multi win(Paper $a, Stone $b)     { 1 }
multi win(Scissors $a, Paper $b)  { 1 }
multi win(Stone $a, Scissors $b)  { 1 }
multi win(Any $a, Any $b)         { 0 }
say win(Paper, Scissors); # 0
say win(Paper, Stone);    # 1
```

Candidate Ordering

- Candidates are examined in order of type narrowness (for class/role types)
- Build a DAG of the candidates, with an edge from A to B if A is narrower than B
  - Narrower if one parameter is narrower and the rest are narrower or tied
- Then produce a topological sort of it to get the dispatch order

Dispatch Algorithm

- Discard any candidates that could never possibly work
- Search what's left in order; if there is one unambiguous solution, call it
- Otherwise see if there are any subset constraints we can choose by, then if anything is marked as default
- If still ambiguous, call any proto; if there is none, then the dispatch fails

Multi-Methods

- Very similar to multi-subs

```
multi method(Int $answer) { say $answer }
```

- We dispatch on the invocant and then consider any multi variants
- A multi with identical types and arity located in a subclass hides anything in the superclass (for calls on instances of the subclass)

Generic Routines

Type Variables

- Just as you have have variables holding values, you can also have type variables which hold types
- They have the sigil `::`
- Can declare and scope them just as you can any other variable, and use them where you would write a type

```
my ::T = $condition ?? Foo !! Bar;
my $x = T.new()
```

Generic Routines

- You can capture the type of a parameter that is passed to a routine

```
sub foo(::T $x) { say "Got a " ~ T.WHAT; }
foo(4.2);       # Got a Num
foo("OH HAI"); # Got a Str
```

- You can use it as a type constraint later in the parameter list too (note only use `::T` to bind it, then use `T` later)

```
sub foo(::T $x, T $y) { }
foo("OH HAI", "KPLZTHNXBYE"); # ok
foo("OH HAI", 42");           # exception
```

Status

What's Implemented

- .WHAT to get the type of something
- Enforcement of type constraints (class, role and subset) on variables, attributes and parameters
- Declaration of refinement types with the subset keyword
- Type variables and generic routines
- Arity based multi-subs

Work In Progress

- Type based multi-subs (we have a very preliminary implementation, but not using the correct algorithm and it doesn't work with roles or refinement types)
- Truly supporting multi-methods (we don't handle overrides correctly yet)
- The `is default` trait
- Declaring `proto` subs

Things Hopefully For Later This Year

- Type-parametric roles
  - You can make roles take type and value parameters
  - Then can choose the correct role based upon what types/values you supply when doing it
  - Same dispatch algorithm as for multis

Things For The More Distant Future

- These are not required for 6.0.0 but will be good to have at some point
  - Some static type analysis, to catch obvious mistakes at compile time (things that couldn't work at runtime)
  - Optimisations based upon type annotations and/or inferences
  - Multi-dispatch on named parameters and/or desired return type

Object Orientation, The Perl 6 Way

Danke

Спасибо Gracias

Thank You

Dank je D'akujem

Merci Tak

Questions?