There's More Than One Way To Dispatch It

Джонатан Вортингтон

Український воркшоп «Перл мова» 2009

Topological sorting of a directed acyclic graph constructed from a type narrowness analysis.

WTF?!

Single Dispatch

- We're used to writing subroutines with a name…

```
# Perl 6 sub taking one parameter $name
sub greet($name) {
    say "Ahoj, $name!";
}
```

- And calling it by its name, passing any parameters

```
greet('Anna');   # Ahoj, Anna!
```

Single Dispatch

- It's easy

- Of course, sometimes we want to write things that are a bit more flexible in what parameters they need

- For example, optional parameters

```
sub greet($name, $greeting = 'Ahoj') {
    say "$greeting, $name!";
}
greet('Anna');                    # Ahoj Anna
greet('Лена', 'Привет '); # Привет, Лена"
```

Multiple Dispatch

- Takes the idea of determining the behaviour by the arguments that are passed a step further

- We write multiple routines with the same name, but different signatures

- We let the runtime engine analyse the parameters that we are passing and call the best routine (known as the best <u>candidate</u>).

Multiple Dispatch – New In Perl 6!

- Multiple dispatch is one of the new features built in to Perl 6

- Not just an obscure feature, but actually right at the heart of the language

  - Operator overloading in Perl 6 will be done by multi-dispatch routines

  - (In fact, all of the built-in operators are invoked by a multi-dispatch.)

Arity

Dispatch By Arity

- Arity = number of arguments that a routine takes

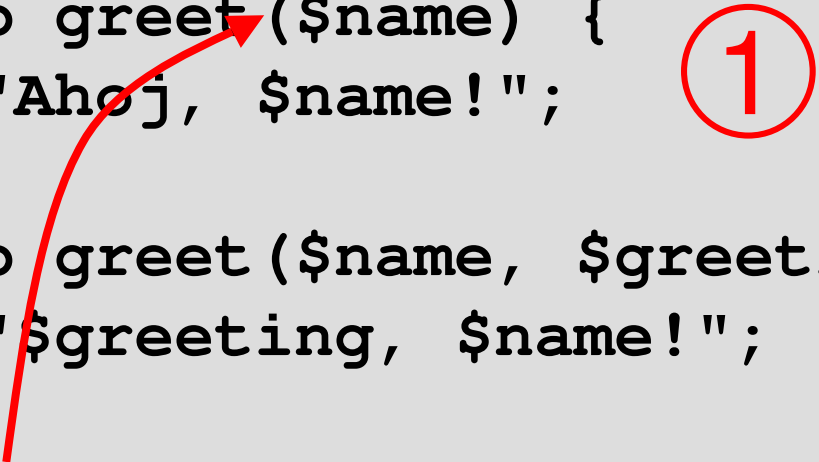- Could do the previous example as:

```
multi sub greet($name) {
    say "Ahoj, $name!";
}
multi sub greet($name, $greeting) {
    say "$greeting, $name!";
}
greet('Anna');                  # Ahoj Anna
greet('Лена', 'Привет ');       # Привет, Лена"
```

Dispatch By Arity

- Arity = number of arguments that a routine takes

- Could do the previous example as:

```
multi sub greet($name) {
    say "Ahoj, $name!";          ①
}
multi sub greet($name, $greeting) {
    say "$greeting, $name!";
}
greet('Anna');                   # Ahoj Anna
greet('Лена', 'Привет ');        # Привет, Лена"
```

Dispatch By Arity

- Arity = number of arguments that a routine takes

- Could do the previous example as:

```
multi sub greet($name) {
    say "Ahoj, $name!";
}
multi sub greet($name, $greeting) {     ②
    say "$greeting, $name!";
}
greet('Anna');                  # Ahoj Anna
greet('Лена', 'Привет ');  # Привет, Лена"
```

Type-Based Dispatch

A Bit About Types

- In Perl 6, values know what kind of thing they are

```
say 42.WHAT;                     # Int
say "пива".WHAT;                 # Str
sub answer { return 42 }
say &answer.WHAT;                # Sub
```

- Including your own classes

```
class Dog { … }
my $fido = Dog.new();
say $fido.WHAT;                  # Dog
```

A Bit About Types

- We can refer to types in our code by name

- For example we can declare a variable can only hold certain types of thing

```
my Int $x = 42;            # OK, 42 isa Int
$x = 100;                  # OK, 100 isa Int
$x = "CHEEZBURGER";        # Error
```

- Again, this works with types you have defined in your own code too

Type-Based Dispatch

- We can write types in a signature
- They are used to help decide which candidate to call

```
multi sub double(Num $x) {
    return 2 * $x;
}
multi sub double(Str $x) {
    return "$x $x";
}
say double(21);      # 42
say double("hej");   # hej hej
```

Type-Based Dispatch

- Paper/Scissor/Stone is easy now

```
class Paper    { }
class Scissor { }
class Stone    { }
multi win(Paper $a,    Stone $b)     { 1 }
multi win(Scissor $a, Paper $b)     { 1 }
multi win(Stone $a,    Scissor $b) { 1 }
multi win(Any $a,      Any $b)       { 0 }

say win(Paper.new, Scissor.new);   # 0
say win(Stone.new, Stone.new);     # 0
say win(Paper.new, Stone.new);     # 1
```
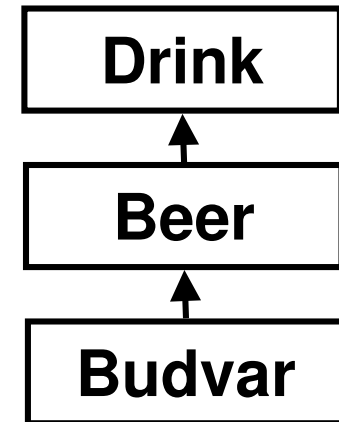
Type Hierarchies in Multi Dispatch

- It's quite clear to see what will happen in the previous examples

- When we have a more complex type hierarchy, things are less simple…

- …especially when we may have different parameters belonging to different or related type hierarchies…

- …got a headache yet?

Type Hierarchies in Multi Dispatch

- It's all based upon the idea of type <u>narrowness</u>

- Consider classes in an inheritance relationship

- Here, we say that Beer is a narrower type than Drink, and Budvar is a narrower type than Beer

```
┌──────────┐
│  Drink   │
└──────────┘
     ↑
┌──────────┐
│  Beer    │
└──────────┘
     ↑
┌──────────┐
│ Budvar   │
└──────────┘
```

Type Hierarchies in Multi Dispatch

- This works for one parameter, but what about candidates overall?

- We say that one candidate is narrower than another when:

  - At least one parameter is narrower

  - The rest of the parameters are either narrower or tied (that is, the same type or not related types)

Type Hierarchies in Multi Dispatch

- Some one-parameter examples

```
multi drink(Budvar $glass) { … }
```
~ is narrower than ~
```
multi drink(Beer $glass) { … }
```

```
multi drink(Beer $glass) { … }
```
~ is tied with (same type) ~
```
multi drink(Beer $glass) { … }
```

```
multi drink(Milk $glass) { … }
```
~ is tied with (unrelated type) ~
```
multi drink(Budvar $glass) { … }
```

Type Hierarchies in Multi Dispatch

- Some trickier examples

```
multi drink(Budvar $a, Beer $b)   { … }
```
~ is narrower than ~
```
multi drink(Beer $a,   Beer $b)   { … }
```

```
multi drink(Budvar $a, Beer $b)   { … }
```
~ is narrower than ~
```
multi drink(Beer $a,   Milk $b)   { … }
```

```
multi drink(Budbar $a, Beer $b)   { … }
```
~ is tied with ~
```
multi drink(Beer $a,   Budvar $b) { … }
```

Type Hierarchies in Multi Dispatch

- We use narrowness to produce a candidate ordering:

  - Compare every candidate for narrowness with every other candidate

  - Build a graph with arrows from A to B when A is narrower than B

  - Do a topological sort

Type Hierarchies in Multi Dispatch

- Things to notice about this algorithm that may not be immediately obvious

  - We do the candidate sorting once, not per call (so we don't have to compute the ordering per call, which would really hurt performance)

  - It is completely independent of parameter ordering (the first and last parameters have equal importance)

When Dispatch Fails

Dispatch Failures

- Multiple dispatch can fail in a couple of ways
  - When all candidates have been considered, and none of them accept the parameters we have passed
  - When we have two or more candidates that accept the parameters and have no way to decide which one is better

No Applicable Candidates

- The following program will give an error saying that there are no applicable candidates

```
multi sub double(Num $x) {
    return 2 * $x;
}
multi sub double(Str $x) {
    return "$x $x";
}
double(1..10); # 1..10 is a Range object
```

Ambiguous Candidates

- This one fails due to ambiguity

```
multi sub say_sum(Num $x, Int $y) {
    say $x + $y;
}
multi sub say_sum(Int $x, Num $y) {
    say $x + $y;
}
say_sum(15, 27);
```

- But helpfully tells you what conflicted

```
Ambiguous dispatch to multi 'say_sum'.
Ambiguous candidates had signatures:
:(Num $x, Int $y)
:(Int $x, Num $y)
```

Tie-Breaking With Subtypes

Introducing Subtypes

- In Perl 6, you can take an existing type and "refine" it

```
subset PositveInt of Int where { $_ > 0 }
```

- You can also write an anonymous refinement on a sub parameter

```
sub divide(Num $a,
           Num $b where { $^n != 0 }) {
    return $a / $b;
}
say divide(126, 3); # 42
say divide(100, 0); # Type check failure
```

Subtypes In Multiple Dispatch

- In multiple dispatch, subtypes act as "tie-breakers"
    - First, we narrow down the possible candidates based upon the role or class they expect the parameter to inherit from or do
    - Then, if we have multiple candidates left, we use the subtypes to try and pick a winner

Subtypes In Multiple Dispatch

- Here is an example of using subtypes to distinguish between two candidates

```
multi say_short(Str $x) {
    say $x;
}
multi say_short(Str $x
                where { .chars >= 12 }) {
    say substr($x, 0, 10) ~ '...';
}
say_short("Beer!");          # Beer!
say_short("BeerBeerBeer!"); # BeerBeerBe...
```

If all else fails...

The `is default` Trait

- If you are left with multiple ambiguous candidates, you may also use the **is default** trait to disambiguate them

```
multi foo(Int $x) { 1 }
multi foo(Int $x) is default { 2 }
say foo(1); # 2
```

- This should probably be seen as something of a last resort, and only holds up as long as someone else doesn't write a default of their own!

Writing a proto

- You can also write a fallback that is called if there is an ambiguous dispatch or one that no candidates match

- This is called a proto; we call the one most immediately in scope at the time of the call

```
proto say_short(Any $x) {
    # Stringify and re-dispatch.
    say_short(~$x);
}
```

Дякую!

Questions?