Advanced Polymorphism in Perl 6

Features of a second-generation type system

Author: John M. Długosz

dhcgnd702@sneakemail.com

http://www.dlugosz.com/Perl6/

Version 1 – April 27, 2008

Version 1.3 – April 30, 2008 (see *Changes/Progress* on page 20)

This is a first look at a working design for the Perl 6 type system. It has not been accepted or endorsed by anyone, but is being shown for first impressions and feedback.

Table of Contents

Introduction

If you are familiar with classes and object-oriented programming in languages like C++, Java, and C#, you have a notion that derived classes can be used anywhere

that code was written to use the base class. This is commonly called an "isa" relationship, and forms the bases of polymorphism in these languages.

However, the rules for substitutability demand that overridden functions have to match the signatures of the original. C++ etc. won't let you do otherwise.

Languages like Smalltalk and Javascript don't care so much about declared types, and will let you pass an object of any class where something different was expected. This works if the object faithfully accepts all the methods called on it. It fails at run time otherwise. Perl 5 falls into this category, as it doesn't do any kind of static type checking or care about the relationships of objects.

Perl 6 provides for *optional* strong type checking. This will bring more significant software engineering features to Perl, without sacrificing the flexibility and freedom that makes Perl easier to write in. So, it needs to support all the checking of strongly-typed languages like C++ and all the freedom of untyped languages like Smalltalk, at the same time!

Perl 6 will let you override methods and change the parameter types. It will even let you retype inherited content in a general way. Clearly this is beyond what C++ allows, and we know why C++ doesn't like that. How can Perl 6 manage it, with strong type checking present?

Perl 6 has a more advanced notion of strong-typed polymorphism than existing mainstream languages. This article discusses the issues and introduces the features that make it work.

Why strong typing?

The earliest Perl 6 requirements documentation included the desire for optional strong typing. Some typing was introduced in Perl 5, but it doesn't help much because the language itself doesn't do much with that information.

- The ability to overload functions and operators, with full multi-parameter dispatch.
- Less verbose code, since the compiler knows more about what you mean. E.g. `my Dog .= new("fido")` doesn't make you specify the type again with `new`.
- Up-front checking that modules are used correctly.
- Program correctness analysis without needing full run-time coverage to detect "wrong object" problems.
- Improved performance.
- More higher-order metaprogramming capabilities.

Perl 6 syntax

Perl 5	**Perl 6**
A class is just a package	Still is, but now use the **class** keyword instead of package to enable all the built-in syntax.
methods are just subs in the package	methods use the **method** keyword, not sub.
attributes are stored however you like, typically by making the class a blessed hash.	built-in abstraction (if you choose to use it), attributes declared with the **has** keyword.
always have to reference calls or hash keys using an explicit self object. E.g. $self->foo, $$self{x}.	A dotted call will use $_ if you don't specify an object. Attributes use access methods, .foo, .x. means same as $_.foo, $_.x. Private attributes use ! instead of . to access the real location rather than using the accessor: self!x = 5;
Inherit by setting up search path in package @ISA variable	Use **is** after the class name. Also uses **does** for roles.
You must extract invocant from first parameter.	The **self** function accesses the invocant even if you did not save it in a variable yourself. It is also placed in $_ on method entry.
Parameter lists are just @_.	Declared parameters, with optional types, and return type after the --> symbol[1]. sub plus(Num $x, Num $y --> Num)

Roles are beyond the scope of this article. They can work like Interfaces, but they are different from inheriting base classes because they are incomplete and flatten into the final class.

Built-in types include **Num** for numbers (highest-precision native floating point), **Int** for integers (arbitrary precision), and **Str** for strings. There is also **Any**, which is untyped.

1 But TIMTOWTDI.

Classic derived types and subtypes

In C++, derived types are subtypes. A subtype is one that can be substituted for another. Of course, the rules of C++ require that the virtual functions have the exact same parameter lists, so that the object can indeed be substituted and the virtual dispatch table is the same.

What happens if we relax this?

Let's start with a simple class and some code that uses it.

```
class Point {
  has $.x;
  has $.y;

  method new ($class: $x?, $y? --> Point)
  {
   return $class.bless(x=>$x, y=>$y);
  }

  method midpoint (Point $other --> Point)
  {
   my $midX = (self!x + $other!x) / 2;
   my $midY = (self!y + $other!y) / 2;
   return .new($midX, $midY);
  }

  } # end of Point


sub foo ($p1, $p2)
 {
  my $result = $p1.midpoint($p2);
  $result.x = 0 if $result.x < 0;
  return $result;
 }
```

Clearly foo has no trouble using the Point, since that is what it was written for.

Now we'll extend the class to create a Point3D.

```
class Point3D is Point {
  has $.z;

  method new ($class: $x?, $y?, $z? --> Point3D)
  {
   return $class.bless(x=>$x, y=>$y, z=>$z);
  }

  method midpoint (Point3D $other --> Point3D)
  {
   my $midX = (.x + $other.x) / 2;
   my $midY = (.y + $other.y) / 2;
   my $midZ = (.z + $other.z) / 2;
   return .new($midX, $midY, $midZ);
  }

 } # end of Point3D
```

And call foo with object of the derived type:

```
my $first = Point3D.new(-10,-10,5);
my $second = Point3D.new(5,10,12);
my $result = foo($first, $second);
```

No surprise, it works.

Now suppose that the program is growing from a quick script into something that needs to be maintained and updated with new features. As part of shift from Q&D to "software engineering", we add strong typing.

Here is the revised foo.

```
sub foo (Point $p1, Point $p2 --> Point)
{
 my Point $result = $p1.midpoint($p2);
 $result.x = 0 if $result.x < 0;
 return $result;
}
```

Unfortunately, the line

```
my $result = foo($first, $second);
```

that worked fine before now gives a run-time error. It won't let me pass a Point3D where the function expected a Point.

Why not? Aren't derived classes supposed to be substitutable for the base class? Well, in C++ the override of midpoint would not have been allowed, because it changed the parameter type. A call to the method Point::midpoint wants to be passed a Point. But the call to Point3D.midpoint wants to be passed a more-restrictive value

of Point3D.

The subtyping rules state that overridden methods must have parameter types that are the same or more general, and the return types are the same or more specialized. This method violates this rule, so class Point3D is *not* a subtype of class Point. The "isa" relationship does not hold.

Derived classes are not necessarily subtypes.

If you really want the derived class to be a subtype, for sure, you can use the :subtype modifier to is, and then the compiler will make sure you follow the subtyping rules and check that your overridden methods are all Kosher.

```
class C is :subtype B { ... }
```

Generic Types

Now C++ programmers might recognise the reuse pattern in foo as something that ought to use generic programming, like with the Standard Template Library in that language. Templates take classes based on what they actually manage to do, not based on what parts were reused when they were written. And the signature really ought to say it wants two of the same thing and returns another of the same. If the arguments are the same type, than the internal use will certainly call midpoint properly so the substitutability isn't an issue.

In C++ foo would be written as a template with one type parameter. In Perl 6, this exactly translates as:

```
sub foo (::PointType $p1, PointType $p2 --> PointType)
{
my PointType $result = $p1.midpoint($p2);
$result.x = 0 if $result.x < 0;
return $result;
}
```

Notice that you don't have a separate line saying that it's a template, as in C++. Rather, the generic type ::PointType is declared (by using the :: sigil) in its first use in the signature. Now, whatever is passed for $p1 is noted, and $p2 must conform, and the function returns one of the same. Likewise, the local variable $result has the matching type.

As with C++, this will take anything and then complain if the actual type is unsuitable for the code. For example, if you passed in something that didn't have a midpoint method or an x method, it would not work!

Unlike C++, and like Eiffel, you could declare foo to constrain the generic types allowed. Besides giving errors earlier if called wrong, it is important that you do this if you want to overload foo, providing one form that works on point-like things and another form that is totally different.

Now since we already know that "types that isa Point" doesn't work, this requires a more powerful way of representing type families. We'll return to that notion later.

There are "type families" that are more general than simple "isa" relationships with a base class or interface.

Parameterized Types

Perl 6 offers parameterized types that are most similar to C++'s parameterized types, and more powerful than C# or Java 5 equivalent features.

Notice that the original Point class does not give any type for its x and y attributes. It will really change to pick up whatever I use in the constructor, as long as the arithmetic works at run-time. This is untyped in the classic Perl 5 or Smalltalk way.

As written, it shows that I can mix typed and untyped notions quite effectively. I don't really worry about that type, and if I decided to pass some Rat numbers or extended precision real numbers, I would expect it to work. If I passed in Complex numbers or quaternions, I could hold my breath for a moment and see if it blew up or seemed to work OK, at least in this case. After all, the checking is done at run time, if all the operations are present and behave properly.

Here I'll make the Point class use a generic parameter for the individual values. This means that both x and y (and later z) will use the same type, and accessors will be strongly typed.

```
class Point [::T = Any] {
  has T $.x;
  has T $.y;

  method new ($class: T $x?, T $y? --> Point)
  {
   return $class.bless(x=>$x, y=>$y);
  }
```

```
method midpoint (Point[T] $other --> Point[T])
{
 my T $midX = (self!x + $other!x) / 2;
 my T $midY = (self!y + $other!y) / 2;
 return .new($midX, $midY);
}

} # end of class Point
```

The type's parameter list is placed in square brackets, seen above in bold. The generic type parameter ::T is declared. The default value is Any, so if you leave off the parameter list the Point is the same as it was before. Existing code is not affected! Updating the class was a matter of adding T's to all the declarations that were untyped before.

Notice that method midpoint uses Point[T] in the parameter list, rather than just Point. In C++, using the template name alone within the template means to use the same arguments as the one you're in. In Perl, Point by itself will mean Point[Any]. You have to pass in the same arguments explicitly. This would be a good reason to use the special symbol ::?CLASS instead of restating the class name within the class.

```
my Point $p1 .= new(1,2.0);   # still works, exact same results
my Point[Rat] .= new(1,2);  # do it with rational arithmetic
my Point[Num] .= new(1,2);
my Point[Str] .= new("hello","world");
```

That last one doesn't make sense. What happens when midpoint tries to add two strings together and divide by 2? In C++, this would cause a compile-time error if you had tried to call midpoint. If you avoid methods that wouldn't work, you can get away with pushing the reasonable boundaries of type type arguments. In Perl, this might get a compile-time error in the same manner as C++, or it might not notice until run-time when Point[Str]::midpoint is invoked, or it might not notice until it actually gets inside midpoint and hits the bad line, just like with the untyped case. With the code shown, it will not give an error at all but return a midpoint of Str "0", because strings are automatically parsed as numbers if you try to do arithmetic with them.

As mentioned, existing code is unchanged, relying on the default argument of Any. This includes the use of Point as a base class in the definition of Point3D. Point3D continues to be derived from a Point[Any]. It is a simple matter to update this class to be parameterized, too. As you see, this can be done a little at a time, adding type information within the existing program.

```
class Point3D [::T = Any] is Point[T] {
  has T $.z;
```

```
method new ($class: T $x?, T $y?, T $z? --> CLASS)
{
 return $class.bless(x=>$x, y=>$y, z=>$z);
}

method midpoint (CLASS $other --> CLASS)
{
 my T $midX = (.x + $other.x) / 2;
 my T $midY = (.y + $other.y) / 2;
 my T $midZ = (.z + $other.z) / 2;
 return .new($midX, $midY, $midZ);
}

} # end of Point3D
```

Again, this was just a matter of adding T's where no types were before, and using CLASS to refer to this class itself.

Constraining types

Suppose you were using Java/C# style interfaces, or abstract base classes in C++ for the purpose, to separate out individual code features. Then you can write code that takes anything supporting that interface, which is more abstract than any real class.

The problem is that these languages only allow one interface to be specified when declaring the variable or parameter!

```
sub foo (ISort IBend IFile $x)
{
 $x.sort;
 $x.bend;
 $x.save("filename");
}
```

Clearly Perl 6 does not have the same limitation. Here I can pass anything to foo provided that it "isa" ISort, IBend, and an IFile all at the same time. This does not only work for "interfaces", but any class or role or any kind of type at all. Basically, you can list more than one type in juxtaposition and the thing being declared will be required to be all of them at once.

```
sub bar (Goat Lion $x) { ... }
```

You might have a tough time calling this function with anything, since the parameter has to be both a Goat and a Lion. But then one day you make a class:

```
class Chimera is Goat is Lion { ... }

my Chimera $beast .= new;
bar($beast);  # hey, I can use it now!
```

Now if you're wondering where this is going, the key is that a "juxtaposition of multiple types" may include generic types as well as anything else. Let's revisit foo:

```
sub foo (ISort IBend IFile ::T $x)
{
 my T $y= $x.sort;
 $x.bend;
 $y.save("filename");
}
```

Now whatever I pass to foo has its type constrained to those three "isa" relationships, and also has the actual type captured by the symbol ::T, which can be used within the code to create more variables of the same type.

Now look again at this code:

```
sub foo (::PointType $p1, PointType $p2 --> PointType)
{
 my PointType $result = $p1.midpoint($p2);
 $result.x = 0 if $result.x < 0;
 return $result;
}
```

To constrain the function to only take point-like things, it could be something like this:

```
sub foo (Pointlike ::PointType $p1, PointType $p2 --> PointType)
```

Now I could make a dummy class Pointlike that doesn't do anything, and just use it as a base class for all those I want to mark as being point-like. Interfaces should work here but we run into the same problem of derived classes not being subtypes.

Interfaces or not?

To mimic the Interfaces idea in Perl 6, you might write a base class like this:

```
class ISort {
  method sort (-->ISort) { ... }
  }

class IFile {
  method save (Str $filename) { ... }
  method ^load (Str $filename --> IFile) { ... }
  }
```

This is typical of Java or C# or COM, where you have to use the interface as the only

type information because the methods in the interface don't know about the full type. So how do you end up using a class derived from these interfaces?

```
sub foo (ISort IBend IFile ::T $x)
{
 my T $y= $x.sort;  # woops, return value needs downcasting!
 my T $z = T.^load("filename");  # woops, return value needs downcasting!
 ...
}
```

Well, you get the idea. If you like putting up with this, you can continue using C#.

In Perl, you can declare the methods to work with the proper types. That is another thing about using CLASS rather than the actual class name—it is virtual! It will change to the real dynamic type when the method is inherited. All[2] class names will behave that way, and we'll look into it more later.

Also, I'll change the example to use role instead of class. It doesn't matter for this example, but all sibling roles flatten as they combine and can contain implementation code that refers to the final class, like templates do.

```
role ISort {
  method sort (-->CLASS) { ... }
  }

role IFile {
  method save (Str $filename) { ... }
  method ^load (Str $filename --> CLASS) { ... }
  }


class Thingie {
  does ISort;
  does IFile;
  does IBend;
  method sort (-->CLASS) { ... }
  method save (Str $filename) { ... }
  method ^load (Str $filename --> CLASS) { ... }
  }
```

Now look at what happens when you pass a Thingie to foo:

2 All unqualified non-global names. CLASS is virtual, ::?CLASS is not.

```
sub foo (ISort IBend IFile ::T $x)
{
 my T $y= $x.sort;
 my T $z = T.^load("filename");
 ...
}
```

The return values from the methods are correctly typed. But return values are easy. Return types may be more specialized in overrides[3], even in C++. This is called *covariance*, meaning they vary in the same direction. Derived types may change return values to the derived type, too. So Thingie has an "isa" relationship to ISort and to IFile.

But things get more challenging with parameter types. Consider the IBend role:

```
role IBend {
  method merge (CLASS $other --> CLASS) { ... }
  }
```

Perhaps Thingie overrides merge to take care of details specific to the class, or perhaps version inherited from IBend (something Java interfaces can't do!) automatically adapts to the concrete class without help. Either way, there is now a method in Thingie that is typed like so:

```
  method merge (Thingie $other --> Thingie)
```

Now what happens if you try to pass a Thingie to foo? Compile time error?

Compare this with the method that follows Java-like rules:

```
  method merge (IBend $other --> IBend)
```

and you see the problem. The parameter is more specialized in the overridden version, and that violates subtyping rules. The Thingie cannot substitute for IBend in code that was written with the method signatures in IBend.

On the other hand, if you don't change the parameter type but only change the return type (which is not a problem), you don't get the proper type checking within the implementation of functions written to IBend:

```
role IBend {
  method merge (IBend $other --> CLASS) { ... }
  }

# method inside Thingie is:
  method merge (IBend $other --> Thingie)
```

3 Actually, the ::?CLASS does not exist in a role body, but for purposes of "isa" checking, it acts like it has the role's type, the same as if you were using an abstract base class instead of a role.

```
sub bendit (IBend ::T $p -->T)
{
 IBend $q = get_something;
 my T $result= $p.merge($q);
 return $result;
}
```

The generics and the covariant return types are still in place. But the parameter is not updated to match the actual class. So, it can't be checked by the compiler either. In bendit, the value of $q can be anything derived from IBend, with no constraint that it match the type of $p, even though the type of $p is captured and usable as type T.

Presumably, the implementation of Thingie::merge does a downcast of the parameter, and checks for the correct type then. This is unnecessary since Perl will in fact allow you to put the desired parameter type in merge, so it follows the derived class.

The language lets us write the desired method, and use it properly within generic code. The only thing missing is being able to use IBend as a way to constrain the allowed types of the generic.

Beyond "isa"

The solution is to design the language feature so it does what we want of it. In fact, using it is easy.

```
sub bendit (£ IBend ::T $p -->T)
{
 T $q = get_something;
 my T $result= $p.merge($q);
 return $result;
}
```

This works now even when the parameter type of Thingie::merge is changed to accept only another Thingie, and the signature in the role is defined to take "another like me" just like the return type.

```
role IBend {
  method merge (CLASS $other --> CLASS) { ... }
  }
```

Even though Thingie does not have an "isa" relationship to IBend in the manner of C++, Java, and C#, it clearly does follow in the sense that the parameter is "another like me" no matter what the class is, even though it changes from class to class. We see that it does in fact allow reuse with different classes when the code doing the reusing employs generics in a suitable way, rather than assuming only unchanging method signatures.

In case you missed it, the difference in this version of bendit is the £ symbol[4]. Typing the parameter as IBend is rejected because Thingie is not a subtype of IBend. But typing the parameter as £ IBend does work, because the £ gives us a deeper level of matching.

The £ specifies a deeper meaning of matching.

It works by considering IBend not as it was originally defined, but how it is transformed when it is used in the context of Thingie. In this case, we know that ::?CLASS follows the actual class, so it's obviously a match. More generally, the constraint type might contain type parameters and virtual type names that all change when it is inherited.

The meaning of £ does not require that the type being passed is actually derived from the constraint—only that it conform to the generalized meaning of the constraint.

So let's revisit the "point-like" constraint from earlier.

```
role pointlike [::T = Any] {
  method x (-->T) { ... }
  method y (-->T) { ... }
  method new ($class: T $x?, T $y? --> CLASS)  { ... }
  method midpoint (CLASS $other --> CLASS)  { ... }
  }
```

This interface-like role describes what my code expects of a point-like thing. It has access methods x and y, a new method that takes two optional parameters, all of the same type; and the midpoint method that takes and returns the same class as the concrete class in question.

```
sub foo (£ pointlike ::PointType $p1, PointType $p2 --> PointType)
```

Even though the Point and Point3D classes were not written to inherit from pointlike (I just made it up to use a a constraint here), the £ works its magic and says "yes, Point[Num] is point-like" and "yes, Point3D[Rat] is point-like".

It has to figure out that CLASS is OK, like the earlier example, but also that the T type will become Num or Rat in these examples, in order that the concrete class fit with the pointlike role.

In short, £ gives a general higher-order type substitution matching that works well with generic types.

Formally, £ pointlike means the OR-junction of all specializations of pointlike, across

4 It can also be spelled like. All symbols in standard Perl 6 have ASCII equivalents.

the type parameters and changing out CLASS and any other virtual type names. In practice, it works by pretending that the class (e.g. Point3D[Rat]) did in fact inherit from pointlike, plugging in the virtual type names based on that position, and seeing if the methods in the class conform to the specialized pointlike role. Because pointlike has a type parameter, it also has to figure out that this can take on some value (Rat) that works. It does that by matching the methods in Point3D[Rat] against the methods in pointlike, and where the latter had a T, seeing what it matched up with in the concrete class. If at least one of the matches, when used as T throughout, allows Point3D[Rat] to conform to the subtyping rules, than the match is a success. Since the methods in Point3D[Rat] use Rat in every place that pointlike has a T, this is simple, and T must be Rat. It handles some more complex cases too, and if it's too complex to figure out automatically, you can declare how it will resolve without having to change the code in Point3D.

Beyond interfaces

As the project continues to be refactored, it makes sense to push the implementation of Point up into pointlike. Unlike Java interfaces, and like C++ abstract base classes, roles may contain method implementations.

```
role pointlike [::T = Any] {
  has T $.x;
  has T $.y;

  method new ($class: T $x?, T $y? --> ::?CLASS)
  {
   return $class.bless(x=>$x, y=>$y);
  }

  method midpoint (::?CLASS $other --> ::?CLASS)
  {
   my T $midX = (.x + $other.x) / 2;
   my T $midY = (.y + $other.y) / 2;
   return .new($midX, $midY);
  }

  } # end pointlike role

class Point [::T = Any] {
  does pointlike[T];
  }
```

```
sub foo (£ pointlike ::PointType $p1, PointType $p2 --> PointType)
{
 my PointType $result = $p1.midpoint($p2);
 $result.x = 0 if $result.x < 0;
 return $result;
}
```

If I had another class that was not derived from Point or declared to use the pointlike role, I could still pass one to foo if it fit the constraints mandated by pointlike. For example,

```
class FoxPoint {
  method x (-->Num) { ... }
  method y (-->Num) { ... }
  method new ($class: Num $x?, Num $y? --> FoxPoint) { ... }
  method midpoint (FoxPoint $other --> FoxPoint) { ... }
  }

my FoxPoint $p1.= new(1,2);
my FoxPoint $p2 .= new(1,2);
my FoxPoint $result = foo($p1,$p2);
```

FoxPoint doesn't use autogenerated accessors for $.x and $.y attributes. How it actually stores the data is not shown. But that's OK, because the constraint matching looks at the access methods, not the private attributes, and finds that it has methods x and y.

So £ pointlike figures out that T is Num and CLASS is FoxPoint, and decides that the match is good.

Virtual types and parallel hierarchies

You saw the behavior of CLASS, but that is not a special isolated example. In general, types used by classes can be redefined by derived classes.

Here is an example adapted from *Theory of Classification* by A J H Simons[5].

The idea is to have parallel hierarchies of vehicles and locations, so that each vehicle has a corresponding location, such as Car/Garage and Airplane/Hanger.

```
class Vehicle { ... }  # pre-declare because of circular reference
```

5Anthony J H Simons: "The Theory of Classification Part 20: Modular Checking of Classtypes", in *Journal of Object Technology*, vol. 4, no. 17, September-October 2005, pp. 7-18 http://www.jot.fm/issues/issue_2005_09/column1
but find it at http://www.dcs.shef.ac.uk/~ajhs/classify/index.html

```
class Location {
  has Str $.address;
  has Vehicle $.keeps;
  method new ($class: String $address, Vehicle $keeps?  --> CLASS)
  {
   return $class.bless(:$address, :$keeps);
  }
  method keep (Vehicle $v)
  {
   self!keeps = $v;
   $v.keeps(self)  unless $v.keptAt =:= self;
  }

 }

class Vehicle {
  has Person $.owner;
  has Location $.keptAt;
  method new ($class: Person $owner, Location $keptAt  --> CLASS)
  {
   return $class.bless(:$owner, :$keptAt);
  }
  method keepAt (Location $.keptAt)
  {
   $.keptAt.keep(self)  unless $.keptAt.keeps =:= self;
  }

 } # end of Vehicle

class Car is Vehicle {
   # whatever is special about Car goes here.
   }

my Person $wen .= new("Wendolene");
my Car $car .= new($wen);
my Location $loc .= new("3 Town Square", $car);
```

So far, so good. The call to Location.new passes a Car where it wanted a Vehicle. That is accepted in the usual "isa" way. However, it loses the actual type on the accessor, and the line

```
$car = $loc.keeps;
```

produces a type check error because the right-hand-side returns a Location. You would need an explicit downcast, and that is unacceptable. We need the other half.

```
class Garage is Location {
  my ::Vehicle ::= Car;
  }
class Car is Vehicle {
  my ::Location ::= Garage;
  }
```

This creates a class Garage, and inherits everything from Location. But, it does more than just copy all the inherited stuff unchanged. It changes all the uses of Vehicle into uses of Car instead!

That works because type names are virtual. The original definition of Location refers to a symbol named Vehicle, and the compiler finds that symbol in the Vehicle class. But in class Garage, the local symbol Vehicle (which is aliased to Car) is found and that hides the Vehicle in the outer scope.

This name change doesn't just affect code written in the derived class. It affects everything inherited from the base class! Again, this is more like generic template programming in C++, where symbols are bound when the template is used, not when it is created.

It means that I can write:

```
my Person $wen .= new("Wendolene");
my Car $car .= new($wen);
my Garage $loc .= new("3 Town Square", $car);
$car = $loc.keeps;
```

and everything is typechecked properly. The last line works because the return type from Garage::keeps is Car, not the base class Location. No downcast is necessary. More subtly, the second parameter to Car.new is checked to make sure it is indeed a Car, not just anything derived from Vehicle. In fact, all the inherited methods and attribute storage from Vehicle are rewritten to work specifically to Cars.

By now you should understand the downside: Car is not a subtype of Vehicle even though it is derived from it. Likewise, Garage is not a subtype of Location. That didn't seem to bother the use thus far, because both classes were re-coded to work with the corresponding class. But other code that accepts Vehicle, for example, will be bothered by the lack of the "isa" relationship and not let you reuse it for Cars. But, you already know what to do about that: use the £ symbol at the very least. And use generic types to get the proper type for use within the function.

```
sub process_vehicle (£ Vehicle ::VType $p)
{
 my VType::Location $loc = $p.keptAt;
 my VType $v = $loc.keeps;
}
```

If I pass a Car to process_vehicle, it will correctly type the local variable $loc to Garage,

and $v to another Car, and it knows that $p.keptAt returns a Garage not just a generic Location, and that $loc.keeps returns a Car and not just a generic Vehicle.

Conclusion

Perl 6 has been under design for many years, and incorporates cutting edge features that go beyond anything available in other mainstream languages. This paper explores one aspect of this: higher level polymorphism, and unifying concepts of base/derived relationships and templates.

Concepts discussed in this paper that are not on the Synopses include

- Use of the £ symbol to indicate higher-order type conformance. This is simply introducing a unique symbol to go with a necessary concept. With no established mathematical symbol as an obvious choice, an abstract symbol from the Latin-1/8859-1/Windows Western character set was chosen. Criticism including alternate proposals is welcomed. ASCII alternative is like.
- Derived classes are not necessarily subtypes. Subtyping rules follow from substitutability, and adherence to method overriding restrictions that are not necessary in an untyped system including Perl 5 and Perl 6 without heavy use of type annotations.
- The is keyword may take a :subtype modifier that turns on traditional subtype conformance checks.
- Details of defining parameterized types are never shown in the synopses.

For more information

A companion to this article for more detail on the underlying *isa relationships and inheritance*, http://www.dlugosz.com/Perl6/web/isa-inheritance.html

The Theory of Classification by Anthony J H Simons, MA PhD, http://www.dcs.shef.ac.uk/~ajhs/classify/index.html uses formal methods to analyze mainstream and laboratory languages, and explores the use of F-bounded polymorphism and higher-order typing.

The Perl 6 Synopses, http://svn.perl.org/perl6/doc/trunk/design/syn/ including S12 which covers Objects, including “roles”.

*Traits: Com*posable Units of Behaviour by Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black, http://www.iam.unibe.ch/~scg/Archive/Papers/Scha03aTraits.pdf, describes the feature Perl 6 calls “roles”.

http://en.wikipedia.org/wiki/Subtype

Languages: Eiffel, Gradual Typing in ECMAScript 4, Scala, BETA, Trellis/Owl

Changes/Progress

::?CLASS changed to CLASS.

The ? twigil generally means things that turn into literals at compile time. CLASS is consistent with rules for which types are virtual (unqualified names) and with items like BUILD which are also inherited from Object. Note that the sentiment also applies to the use of ::?CLASS in Roles, so that should change too.

::?CLASS is changed to the literal name of the class at compile time. CLASS is a virtual type defined in Object and automatically overridden in every class:

```
my ::CLASS ::= ::?CLASS;
```

isa changed to modifier

Works on does too. Default is :generic, value to turn on checking is :subtype.