Optimization with Moose

General optimization tips

Friday, June 7, 13

I'd like to start by discussing some general optimization tips. Moose itself is code too, so it's important to briefly cover how to optimize code in general. After that we'll dig into specific optimizations you can watch for in your Moose code.

The first rule of optimization is:

Friday, June 7, 13

Can anyone tell me the first rule of optimization?

The first rule of optimization is:

don’t!

Friday, June 7, 13

Optimization is *not* free by any means. Its costs are real.

Is your time better spent on features?

Bug fixes?

Friday, June 7, 13

Optimization takes your time and attention away from features and bug fixes. Is performance really the most important thing you can be spending your time on? Probably not.

"*There are only two hard problems in Computer Science: cache invalidation and naming things.*"

Friday, June 7, 13

You've probably seen this quote before.
Caching is strictly an optimization technique. So hidden is this quote is the suggestion that optimization is hard. I'd say so is estimating. It will probably take more time to optimize than you think.

Optimization often comes at the expense of maintainability

Friday, June 7, 13

Optimization can involve taking shortcuts you know you can get away with. You break encapsulation "just this for this one hot codepath". You might need to litter your code with special cases to get that extra bit of performance.
Optimized code is code too. Code is buggy, and optimized code doubly so.

You're expensive!

Friday, June 7, 13

Once computers became smaller than entire rooms, programmer time became more expensive than CPU time. Is it worth spending a week of your salary to tackle some optimization, or can you just throw another computer at it and get on with your work?

The first rule of optimization is:

don’t!

Friday, June 7, 13

So that's why we say the first rule of optimization is: don't. But what happens when you DO need to optimize?

The second rule of optimization:

profile!

Friday, June 7, 13

If you've decided you need to optimize, then be sure to profile first.

Don’t assume

Friday, June 7, 13

Measure where your actual bottlenecks are. It doesn't make sense to optimize JavaScript by replacing semicolons with commas and if-statements with && if 90% of your wait time is the network latency of downloading the JavaScript.The right and useful optimization there would be allowing the browser to cache your JS.

Be scientific

- Start by profiling
- Make changes
- Profile again!

Friday, June 7, 13

Profile before you make changes, but also profile AFTER you make changes. If your changes didn't make a difference, then don't commit them!

Modern computers are complex

- Instruction pipelining
- CPU cache lines
- Filesystem cache
- OS process and thread scheduling

Friday, June 7, 13

There's a LOT going on in your laptop and in your servers. All this powerful infrastructure makes it difficult to reason about your changes _a priori_. If you're a guru and you know all this stuff, you probably know you should still profile anyway.

Profile *real* workloads

Friday, June 7, 13

You should profile real workloads, not trivial subsets. If your production table is a hundred million rows, don't profile how your system handles just the first hundred. For example, your system might be able handle just a hundred all in memory, but that is not how your real workload rolls.

One-time and amortized costs

Friday, June 7, 13

If you're only optimizing the first hundred rows instead of all hundred million, you're inflating the effects of one-time costs and other amortized costs, like the compilation of all your code, connecting to the database, and all your application's other initialization.

Be wary of micro-benchmarks

Friday, June 7, 13

Microbenchmarks are benchmarks of trivial workloads in order to persuade you to use their project. They don't tell you the whole story. It's easy to win a database microbenchmark if you don't fsync, which means you might lose data. Which is just about the worst thing a database can do.

You workload changes

Your code changes

Friday, June 7, 13

The data you're dealing with is probably changing over time. Maybe it's 150 million rows now. Profile again to make sure your optimizations are still valid, and to see if you can squeeze out other savings.

The second rule of optimization:

profile!

Friday, June 7, 13

Hence the second rule. Measure, don't just guess and assume.

Profilers

- Devel::NYTProf
- DTrace

Friday, June 7, 13

There are two profilers I'll cover quickly today, Devel::NYTProf and DTrace.

Devel::NYTProf

- Perl-specific
- Line-, sub-, block-, and opcode-level profiling
- Low overhead
- Accurate

Friday, June 7, 13

Devel::NYTProf is an excellent Perl-specific profiler. Don't even bother looking at older Perl profilers. They're far more buggy, slow, ugly, and obtuse. Devel::dprof was removed from Perl core because it sucks compared to NYTProf.

Live demo!

- `cd Moose`
- `export PERL5OPT=-d:NYTProf`
- `ack Moose >/dev/null`
- `export PERL5OPT=`
- `nytprofhtml --open`

No demo?

Performance Profile Index

For /Users/sartak/.perl/perls/perl-5.16.0/bin/ack

Profile of /Users/sartak/.perl/perls/perl-5.16.0/bin/ack for 1.45s (of 1.82s), executing 846546 stat
source files and 1 string evals.

Top 15 Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
536	1	1	473ms	1.23s	App::Ack::search_resource
78937	2	1	289ms	331ms	App::Ack::Resource::Basic::next_text
5157	1	1	94.1ms	94.1ms	App::Ack::Resource::Basic::CORE:ftbinary (opcode)
5157	1	1	90.0ms	215ms	App::Ack::print_match_or_context
79620	5	1	67.1ms	67.1ms	App::Ack::CORE:regcomp (opcode)
5838	2	1	49.6ms	49.6ms	App::Ack::CORE:subst (opcode)
15471	3	1	44.4ms	51.0ms	App::Ack::print
78937	1	1	42.0ms	42.0ms	App::Ack::Resource::Basic::CORE:readline (opcode)
83872	13	1	27.7ms	27.7ms	App::Ack::CORE:match (opcode)
5157	1	1	25.2ms	119ms	App::Ack::Resource::Basic::is_binary
1	1	1	20.6ms	1.39s	App::Ack::print_matches
673	1	1	20.6ms	20.6ms	App::Ack::Resource::Basic::CORE:sysread (opcode)
5157	1	1	12.9ms	27.9ms	App::Ack::print_line_no
1	1	1	12.8ms	12.8ms	utf8::SWASHNEW
5157	1	1	12.8ms	31.1ms	App::Ack::print_filename

See all 383 subroutines

No demo?

2106	536	60µs			my $has_lines = 0;
2107	536	57µs			my @lines;
2108	536	236µs			if (defined $opt->{lines}) {
2109					$has_lines = 1;
2110					@lines = (@{$opt->{lines}}, -1);
2111					undef $regex; # Don't match when printing matching line
2112					}
2113					else {
2114	536	2.95ms	1072	1.14ms	$regex = qr/$opt->{regex}/; # spent 986µs making 536 calls to App::Ack::CORE:qr, avg 2µs/call # spent 158µs making 536 calls to App::Ack::CORE:regcomp, avg 295ns/
2115					}
2116					
2117					# for context processing
2118	536	62µs			$last_output_line = -1;
2119	536	43µs			$any_output = 0;
2120	536	160µs			my $before_context = $opt->{before_context};
2121	536	67µs			my $after_context = $opt->{after_context};
2122					
2123	536	86µs			$keep_context = ($before_context || $after_context) && !$passthru;
2124					
2125	536	30µs			my @before;
2126	536	11µs			my $before_starts_at_line;
2127	536	56µs			my $after = 0; # number of lines still to print after a match
2128					
2129	536	78.0ms	73780	306ms	while ($res->next_text) { # spent 306ms making 73780 calls to App::Ack::Resource::Basic::next_text
2130					# XXX Optimize away the case when there are no more @lines to find.
2131					# XXX $has_lines, $passthru and $v never change. Optimize.
2132	78401	298ms	156802	91.7ms	if ($has_lines # spent 66.5ms making 78401 calls to App::Ack::CORE:regcomp, avg 849r # spent 25.2ms making 78401 calls to App::Ack::CORE:match, avg 321ns/
2133					? $. != $lines[0] # $lines[0] should be a scalar
2134					: $v ? m/$regex/ : !m/$regex/) {

DTrace

- Not Perl-specific
- System profiler
- Profile your kernel too!
- Solaris, OS X, FreeBSD
- Linux support iffy
- Low overhead
- Production safe!

Friday, June 7, 13

DTrace is an excellent system-level profiler. It lets you see what your kernel is doing in relation to your code. Most other tools stop at the syscall but DTrace can show you kernel stack traces, can investigate how other processes are affecting yours, and so much more. DTrace is so low-overhead and safe that you can use it in production. Very handy if you're having trouble replicating results.

DTrace

- Capture many kinds of events
- syscall
- memory allocation
- thread scheduling
- process lifecycle
- Perl function call
- Perl global phase change (BEGIN, END, etc)

Friday, June 7, 13

The way DTrace works is it reports tons of interesting events to the user. It offers a simplistic language for describing events and operating on them. You can slice, dice, and combine the events any way you want to get at the data you're interested in.

Live demo!

- How much time does each of ack's function calls spend in syscalls?

```
sudo dtrace -qZn 'perl::sub-entry /substr(copyinstr(arg3), 0, 8) == "App::Ack"/ { self->wanted = pid; self->profiling = strjoin(copyinstr(arg3), strjoin("::", copyinstr(arg0))) } syscall::entry /self->wanted == pid/ { self->started = timestamp } syscall:::return /self->started/ { @syscalls[self->profiling] = sum((timestamp - self->started) / 1000); self->started = 0 } proc:::exit /self->wanted == pid/ { exit(0) }'
```

- `ack Moose >/dev/null`

No demo?

```
App::Ack::print                                  11
App::Ack::read_ackrc                             38
App::Ack::exit_from_ack                          61
App::Ack::print_matches                          63
App::Ack::Resource::Basic::reset                250
App::Ack::Resource::Basic::next_text            470
App::Ack::Resource::Basic::close                774
App::Ack::BEGIN                                1124
App::Ack::Resource::Basic::needs_line_scan     1893
App::Ack::Repository::Basic::close             3077
App::Ack::Resource::Basic::new                 3248
App::Ack::ignoredir_filter                     3809
App::Ack::is_searchable                        9581
App::Ack::Resource::Basic::is_binary          10877
```


Friday, June 7, 13

This is how long each function in ack spends in syscalls. The number on the right is the number of microseconds. We can see that ack spends a lot of system time in is_binary, so maybe that's something to look at.

General optimizations

Friday, June 7, 13

So now let’s talk about some of the ways you can optimize your code that aren’t Moose-specific. I don’t want to spend to much time here so I’ll just hit the highlights.

Algorithm and data structure changes

Friday, June 7, 13

Algorithm and data structure changes are likely to be your biggest savings. Not changing multiplication to bitshifting. Let the Perl optimizer and your CPU handle that.

Array vs Hash

"*Doing linear scans over an associative array is like trying to club someone to death with a loaded Uzi.*"

- Larry Wall

Friday, June 7, 13

Hashes are great. But if you don't *really* need the behavior of a hash (fast lookup), consider whether you can use an array instead.
Are your keys small integers? Maybe you can treat an array like a hash.

Big-oh complexity

- O(n)
- $O(n^2)$
- $O(n^3)$
- O(log n)
- O(n log n)

Friday, June 7, 13

You should be familiar with big-oh complexity. It measures how a resource like time or memory change based on how much input you give it. Just having a sense of big-oh complexity is enough to discard really terrible implementations before you write any code. Do you need to make a network request for each row in your HTML table? That might be fine for three rows, but not for three hundred.

CPAN modules

- Set::Object
- Heap::Fibonacci
- Graph::Implicit
- Algorithm::

Friday, June 7, 13

If you need a special algorithm or data structure don't dive straight into implementing it yourself. Investigate what's on CPAN. There’s probably something there for what you need. They've probably debugged the code, had other people use it, and wrote lots of tests for it.

Cache?

- `use Memoized; memoize ‘fib’;`
- Memcached

Friday, June 7, 13

Can you insert a cache somewhere to stop redoing work? One of the most accessible caches is Memoize, which maintains a table of function input -> output to avoid recalculation. If you need to cache across multiple machines, can you use Memcached? There are some gigantic websites using memcached, so you're unlikely to hit design limits. Or at least you'll be in good company if you do.

Moose optimizations

Friday, June 7, 13

Now we can discuss some of the optimizations pertinent to Moose code.

Moose is pretty fast

Friday, June 7, 13

We've spent a *lot* of effort making Moose itself as fast as possible, especially compile time.
Our goal for Moose's speed is to make what it gives you as fast as what you would write in Perl by hand. Sometimes we don't hit that goal. Sometimes we beat that goal because Moose does some stuff in C.

Moose gives you knobs

Friday, June 7, 13

Moose offers you some knobs for deciding what tradeoffs make the most sense for your application.

make_immutable

Friday, June 7, 13

make_immutable is an invocation you perform to optimize your class.

```
__PACKAGE__->meta->make_immutable;
```


Friday, June 7, 13

You usually see it written like this at the end of a class declaration.

make_immutable

- string evals a constructor (“new”)
- and a destructor (“DESTRUCTALL”)
- memoizes meta-object methods

Friday, June 7, 13

make_immutable does three primary things. We’ll cover in each in turn. The goal of each of these is to spend a little bit of time up front to make your class faster.

string eval a constructor

(“new”)

Friday, June 7, 13

By default, your constructor is a slow, generic method that consults the metaobject protocol for your class's attributes and how they should be constructed. When you call make_immutable, Moose pulls all that out and writes a lean and mean constructor specific to your class that is much much faster.

string evals a destructor

DEMOLISHALL

Friday, June 7, 13

Moose inlines a DEMOLISHALL for you too, which is a method that calls each DEMOLISH method in turn. By default it again has to consult the metaobject protocol for which classes in your hierarchy provide that DEMOLISH method. So by inlining a class-specific DEMOLISHALL, we make garbage collection a little faster.

Memoizes meta-object methods

Friday, June 7, 13

Metaobject methods answer questions like: what local and inherited attributes does this class have, what are all the methods it can respond to, etc. By invoking make_immutable you're declaring these won't change, so Moose can cache them.
This really only affects code that uses the metaobject protocol, so this will only speed up the MooseX modules you use. But if you peek into ->meta, you'll enjoy the speedups too.

Use `make_immutable`

Friday, June 7, 13

Use make_immutable. It makes your code faster, basically for free.

Use `make_immutable` except when you shouldn't!

Friday, June 7, 13

But there are times when you shouldn't use make_immutable.

"*There ain't no such thing as a free lunch*"

Friday, June 7, 13

All that additional MOP work isn't free.
To compile a lean and mean constructor for your class, Moose has to investigate your class hierarchy and generate Perl code. Then Perl has to compile that code. This takes time.

Amortized cost

Friday, June 7, 13

You only have to pay that make_immutable cost once per class. If you create only a couple objects of this class, then maybe it's not worth it to use make_immutable.
But if you're making tons of objects, then it's probably worth it.

Profile real workloads!

Friday, June 7, 13

Remember, you need to profile real workloads to determine whether removing make_immutable is worth it.

Be kind:

Leave a note

Friday, June 7, 13

If you remove the make_immutable call because you determine it to be not worth its costs, be sure to leave a comment in the source code. The next guy looking at your code might think you overlooked the make_immutable and helpfully add it for you. I personally do this.

Use `make_immutable` by default

Friday, June 7, 13

Your standing orders should be to use make_immutable for all classes. Don't let my cautionary tales scare you, make_immutable is pretty much always the right answer. But sometimes it’s not which is why we’re discussing it.

Sorry...

Friday, June 7, 13

If we had the right tools for automatically invoking make_immutable on your behalf, we would. But at this point even if Perl gains such tools it's probably too late to change that default.

Attributes vs Methods

Friday, June 7, 13

Next up is something we've been considering lately. We love Moose's attributes because they have so many great features and go a long way toward building your classes.
But lately we've been pondering whether we haven't been overusing attributes.

Memory usage

Friday, June 7, 13

Attributes increase memory usage. No mincing words here. Each object has a copy of, or reference to, a value for that attribute. It's also going to take up however much memory a hash key takes up.

Lazy

Friday, June 7, 13

Lazy alleviates the memory usage to some extent by only storing a copy as needed. So you don't pay that memory usage until you actually look at the attribute for that object.
We'll talk about lazy more in a bit.

“get” isn’t free

- is there a @_?
- pull the value out of the object
  - (which is a hash lookup)
- laziness slows this down further

Friday, June 7, 13

"get" accessors have a runtime cost. The accessor has to check the parameter list, even for a getter, to report an error (since getters take no parameters). Then the accessor has to consult the object for its value which involves a hash lookup. If you're using lazy and there's no value, this can also involve an additional method call, type constraint check, etc.!
Regular methods don’t need to do all this work.

Attributes aren't bad...

Friday, June 7, 13

Attributes aren't bad, you just should be aware that there are costs to using them.

Maybe use a method?

Friday, June 7, 13

For some situations where you normally reach for an attribute, a method might work better.

Does it really need to be stored for each object?

Friday, June 7, 13

If you don't need the storage for each object then don't pay the cost! Just make it a method.

Class constant

```
has some_value => (
    is      => 'ro',
    default => sub { 100 },
);

sub some_value { 100 }
```


Friday, June 7, 13

We've long been using attributes for class constants. In this case, the simple method might be better.
This is the specific case that prompted our thinking on using methods instead of attributes.

Method instead of default/ builder?

Friday, June 7, 13

Do you really need to cache the value as an attribute? If your calculation is simple enough, this might be faster! Remember the slide about everything "get" does. A hash lookup is cheap but not free; maybe your calculation is cheaper.

Role application

Friday, June 7, 13

You might wonder whether role application "costs" anything. It does. Moose might have to create composite roles, compile more accessors, copy methods in, etc. But all that work is compile time.

Role application

≈

inlining

Friday, June 7, 13

The net effect of role application is that we inline methods and attributes directly into your class. That means there SHOULD be no runtime overhead.

Trust me,
I benchmarked it

Friday, June 7, 13

I benchmarked it and I found an interesting result that I want to share.

```
Class::WithInheritance:
1882648/s

Class::WithRoles:
1923234/s
```


Friday, June 7, 13

So here's what I found when I benchmarked a simple class hierarchy implemented first with inheritance and with roles. They're pretty close. Close enough that the difference in iterations per second is statistically insignificant. Run the benchmark again and it might favor the other side. I grayed out the less significant digits so you could tell what the difference is easier.

```
Class::WithInheritance
567344/s

Class::WithRoles
659223/s
```


Friday, June 7, 13

But the first time I benchmarked it I saw that roles were chugging through 16% more work in the same amount of time. Which IS significant.
This really surprised me. Is inheritance that slow? But I thought Perl was caching method lookups so the timing of the simple method calls I was doing should have been similar.

Class::WithInheritance::AndClassNamesReallySeemToMatter

434438/s

Class::WithRoles

643734/s

Friday, June 7, 13

I thought about it more and I realized what was going on. The way I wrote my benchmark was forcing Perl to traverse stashes every time I called a method in the benchmark iteration. The way I fixed it was to bless the object so that Perl could presumably do some caching.
I thought I was writing just about the most trivial benchmark and even there the results were surprising.

Lazy

```
has dbh => (
    is      => 'ro',
    default => sub { DBI->connect(...) },
    lazy    => 1,
);
```


Friday, June 7, 13

Lazy is a feature of attributes that defers the initialization of that attribute's value until it's needed. Without lazy, attributes are initialized immediately in the constructor.
If we never actually look at the value of dbh then we never connect to the database, which is an expensive operation.

Lazy saves time and memory

Friday, June 7, 13

If you never look at the attribute's value then Moose never runs the initialization code, which saves runtime. And it never stores the value, which saves memory.

"*There ain't no such thing as a free lunch*"

Friday, June 7, 13

Lazy can be a big win but there is still a cost associated with it.

Lazy's overhead: the accessor

Friday, June 7, 13

Lazy does incur some overhead. It's in the accessor.

“get” isn’t free

- is there a @_?
- pull the value out of the object
  - (which is a hash lookup)
- laziness slows this down further

Friday, June 7, 13

This slide showed up a few minutes ago.
Let's talk about how lazy slows down your accessors.

Accessors - lazy

- Moose blindly returns the attribute's value

Friday, June 7, 13

Without Lazy, Moose just blindly returns the attribute's value. It doesn't ask if the object has a value for that attribute. If it doesn't, then the normal Perl behavior of returning undef for a hash miss will take over.

Accessors + lazy

- object, do you have a value for this attribute?
  - `exists $self->{$attribute}`
- if so, pull it out and return it
- if not:
  - invoke the default/builder
  - set the value
    - which checks type, might run coercions
  - return it

Friday, June 7, 13

When you use lazy, Moose has to make your accessors quite a bit more complicated. The real overhead is that additional existence check. You pay that cost every single time you access the attribute. The rest only happens that first time this attribute is initialized for each object.

Don’t use lazy blindly everywhere

Friday, June 7, 13

Blindly making all of your attributes lazy is not productive. If you always consult the value of the attribute, then lazy is *slower*. You still pay that initialization runtime cost. And the memory cost of storing the value. And you just made your accessor slower for no benefit with that existence check.

Use a method?

Friday, June 7, 13

Does your lazy calculation really need to be an attribute or can it just be a method instead?

Measure!
Profile!

Don’t assume!

Friday, June 7, 13

Try it, see what effects it has on your system.

Detour: lazy's other function

Friday, June 7, 13

Lazy is useful for two unrelated goals. The first is the one we've been discussing: deferring expensive attribute initialization.

Consulting other attributes during initialization

Friday, June 7, 13

The other allows you to consult other attributes in an attribute's builder or default. This is because Moose's attribute's initialization order is explicitly undefined. So you can't depend on another attribute's value being initialized even if it has a default or builder.
So if you want to look at other attributes, you can use lazy to defer your initialization until after construction when your object is complete.

Lazy's functions

- deferring expensive attribute initialization
- consulting other attributes during initialization

Friday, June 7, 13

You have to figure out which effect of lazy this attribute wants, whether it's deferring expensive initialization, or working around Moose's undefined initialization order. You can remove "lazy => 1" from attributes using the former goal and things shouldn't break. But if you remove "lazy => 1" from the latter and you risk running into nasty, subtle side effects.

Type constraints
Type coercions

Friday, June 7, 13

Now let's talk about the last major system that has important performance consideration: types.

Defining types

```
use Moose::Util::TypeConstraints;
subtype 'Price',
      as 'Str',
      where {
/^ \p{Currency_Symbol} \d+ (\.\d+)? $/x
      };
```


Friday, June 7, 13

Moose gives you tools to build types. Here we're creating a Price subtype which is a string that has a currency symbol, some numbers, and then an optional decimal component too.

Defining types

```
package Car;
use Moose;
has sticker_price => (
        is  => 'rw',
        isa => 'Price',
    );
```


Friday, June 7, 13

Then we can start using our new Price type like any other type. When we store a value in sticker_price, that regular expression on the previous slide confirms that the value fits what we want.

Tricky types

- check as many aspects of the value as you want
- database lookups
- network requests?
- Mechanical Turk?
- arbitrary code

Friday, June 7, 13

Types can get as complicated as you want because you can run arbitrary code.

subtype ‘Price’

- number of decimal digits ($ £ €, ¥, gas prices)
- optional commas in the right places
- localization (e.g. “1.000.000,00”)

Friday, June 7, 13

Some ways we could make our Price subtype more robust is by consulting a database using the currency symbol and pulling out how many decimal points are valid for that currency. American dollars; English pounds; and the Euro use two digits, but Japanese yen uses whole numbers. Gas prices in America use three digits! We can also check whether commas and decimals are in the right places, even when considering localization.

Expensive!

Doing all this work in your type checks can get very expensive, especially since a value will typically pass through type checks many times.
Would a simpler type check suffice?

Data normalization?

Friday, June 7, 13

Can you normalize data as it comes into your system?

```
package Car;
use Moose;

has sticker_price => (
    is  => 'rw',
    isa => 'Int',
);

sub formatted_price { ... }
```


Friday, June 7, 13

For example if you're working in a web context, take the input from the user, possibly applying your complex type constraint just once. But then convert it into an unadorned integer number of cents. That way you can use the simpler, faster Int type constraint internally.

Boundary types

Friday, June 7, 13

You could even go so far as to use type constraints only at the “boundary” of your API. When you’re taking input from other systems, validate it. Bugs breed in the boundaries between APIs so type constraints there will win you the most bang for the buck. Flag bad data at the edge of your API before it becomes your fault!
But when you’re just passing data around you don’t need to keep revalidating that the same input still matches the same type.

Type coercion can be expensive too

Friday, June 7, 13

Type coercion is a Moose feature that lets you transform data of one type into another. But it can be expensive.

Type coercion

- Involves many type constraint checks
  - Check each potential “from” constraint
  - Run the coercion’s transform code
  - Ensure the output matches the “to” constraint

Friday, June 7, 13

Type coercion can run potentially very many type constraint checks. If Moose determines that it can coerce a value, then it first needs to figure out which coercion to apply. This involves running potentially many type constraint checks to see whether it can match the input side. Then it transforms the value according to the coercion. Then it checks that the output matches the expected type.

Type coercion

- Makes your API more complex
  - “I can accept a string or an arrayref or a regular expression or a DateTime!”
- You can afford to be strict in your APIs

Friday, June 7, 13

Using type coercions means you can accept multiple different kinds of values. Do you really need to support this or can you ask your users to simply supply the data in the desired format to begin with?

Avoid coercions

- or, use them only at the boundary of your API

Friday, June 7, 13

My personal advice is to avoid coercions because they’re expensive and confuse your API. But I admit they are still useful. Like type constraints, maybe only use them at the boundary of your API where you’re dealing with the harsh realities of the outside world. But internally, inside your API, you shouldn’t really need them.

Types are expensive

Friday, June 7, 13

No matter how much Moose optimizes types, they’re going to be expensive because we’re dealing with Perl which falls deep into the dynamic end of the static-dynamic type spectrum.

But don't discard them

Friday, June 7, 13

But just because they're expensive doesn't mean you should discard them out of hand. They're great for letting you notice bugs *at their source*, instead of somewhere down the line, when some later assumption is broken.

“Not a HASH reference”

Friday, June 7, 13

How many times have you seen an error like this. Your assumption of what data you’d be getting has been broken. Maybe this faulty reference was passed through several layers before it got to your system so you don’t have a good sense of who is responsible for producing this bad data. If you’re in the habit of using type constraints, then you’ll flag this kind of problem sooner and have a better chance of debugging it.

Type inheritance and `inline_as`

Friday, June 7, 13

The next and last performance consideration I want to discuss is type inheritance and a feature called inline_as.

Type validation

- Types have hierarchy
- Each type (except the top-level “Any”) has a supertype
- Each type *specializes* its supertype

Friday, June 7, 13

When you’re declaring a type, recall that you use the ‘subtype’ keyword. This is because your custom types always have a parent type. So in effect there’s a tree of types. The top of the tree is “Any”. Subtypes are stricter than its supertype. In other words, the subtype lets fewer values through than its supertype.

Type validation

- Types have hierarchy
- Each type (except the top-level “Any”) has a supertype
- Each type *specializes* its supertype
- Each type check first validates against the supertype’s constraint, recursively

Friday, June 7, 13

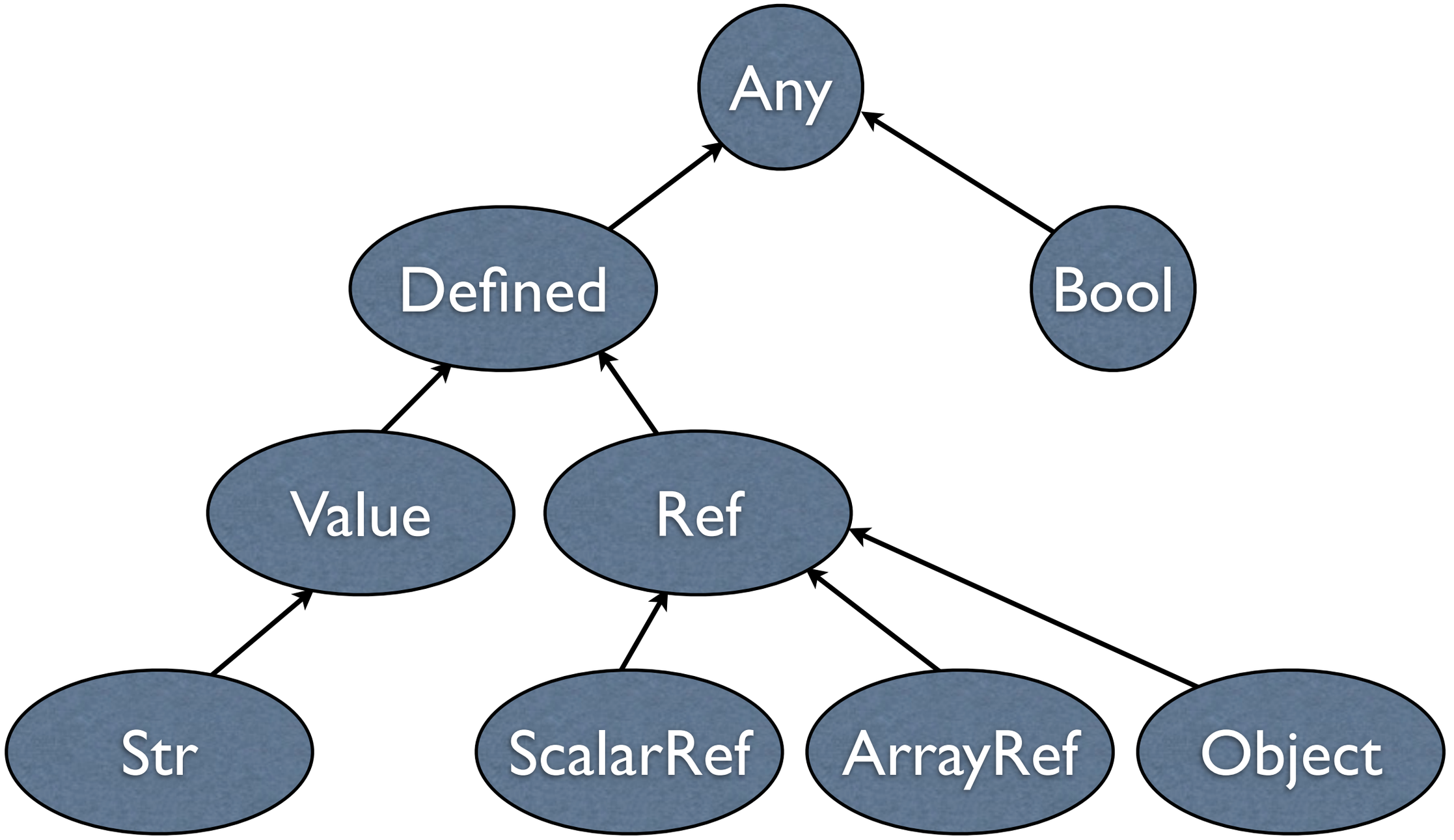
And here’s the performance part of this: when you’re validating against a type, we first check the supertype’s type constraint code. This way you only need to specify how your subtype is different from the supertype.

Type validation

```
subtype ‘PositiveInt’
    as ‘Int’,
where { $_ > 0 };
```

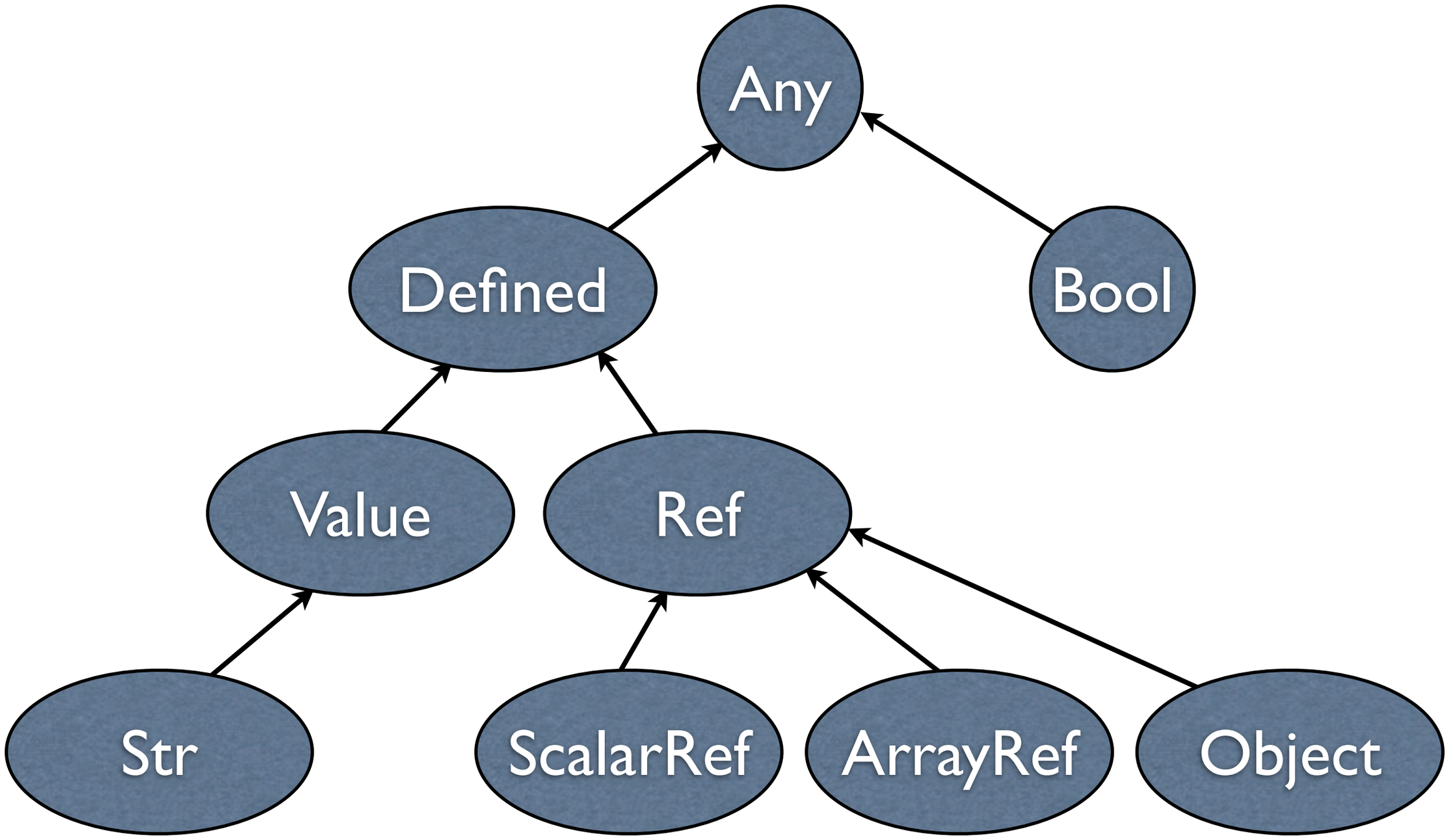

Friday, June 7, 13

This is a complete type constraint declaration. PositiveInt doesn’t need to confirm that the value is an integer, because it relies on its Int supertype to do that. PositiveInt only needs to confirm that the value is more special than what its supertype checks. It doesn’t need to poorly reimplement its supertype validators.



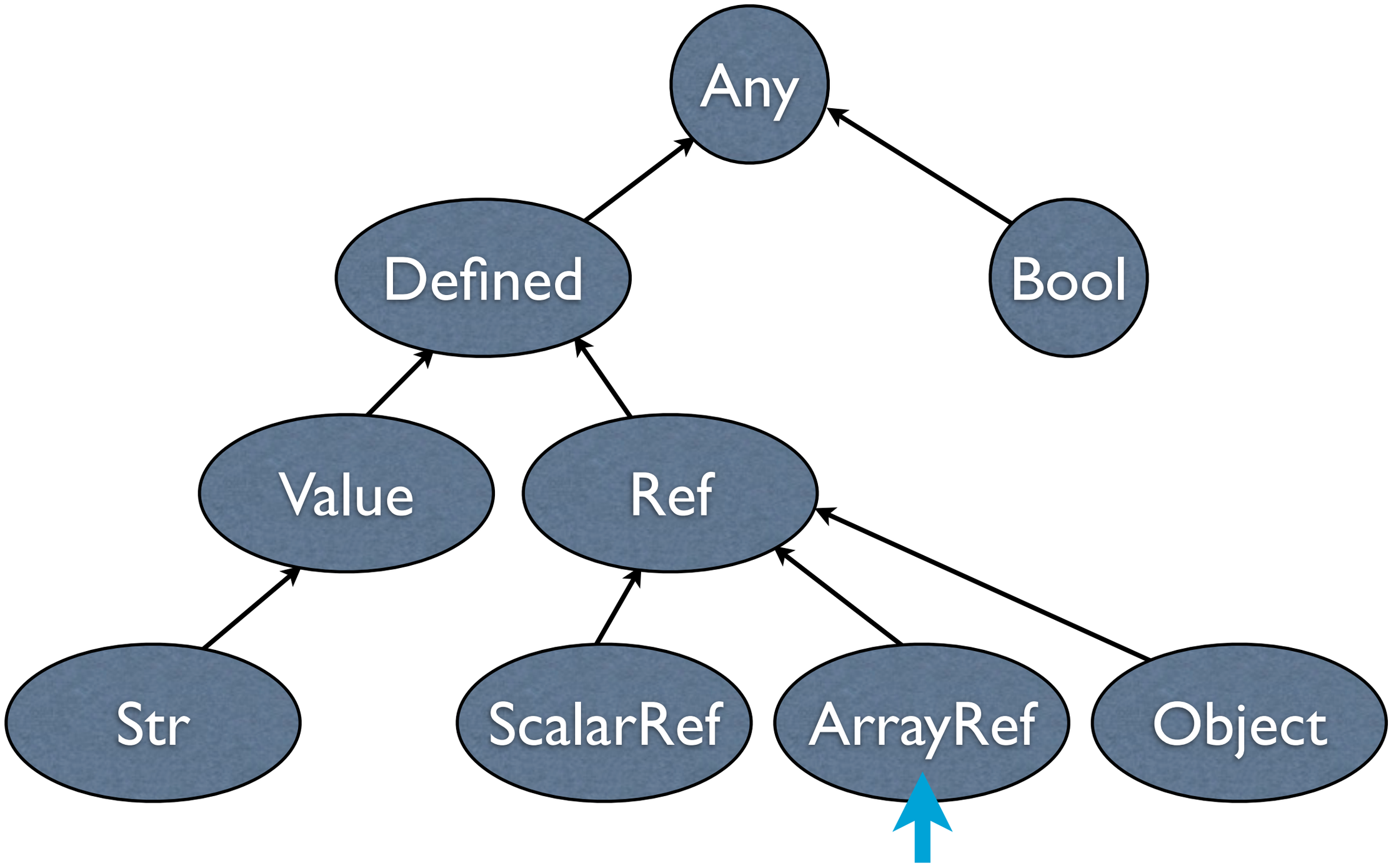
Friday, June 7, 13

Here's part of the builtin type tree. I've excluded a lot just to fit it on the slide. You can see that, for example, Object is a subtype of Ref, which is a subtype of Defined, which is a subtype of Any. Bool is not a subtype of Defined because Bool can include the value undef for false, but Defined can't.



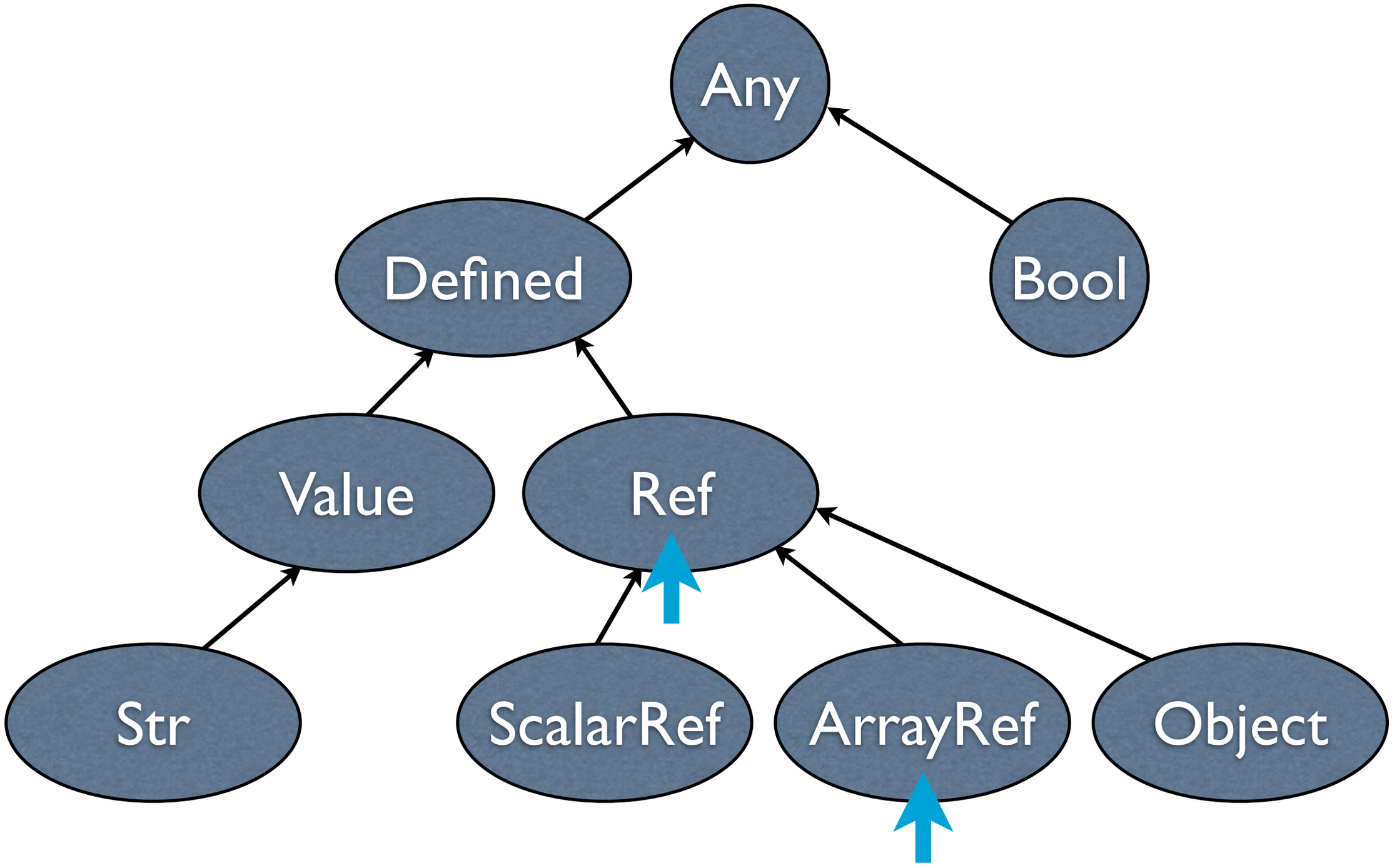
Friday, June 7, 13

As I mentioned, when you're validating a type, you also validate all of its supertypes in turn.



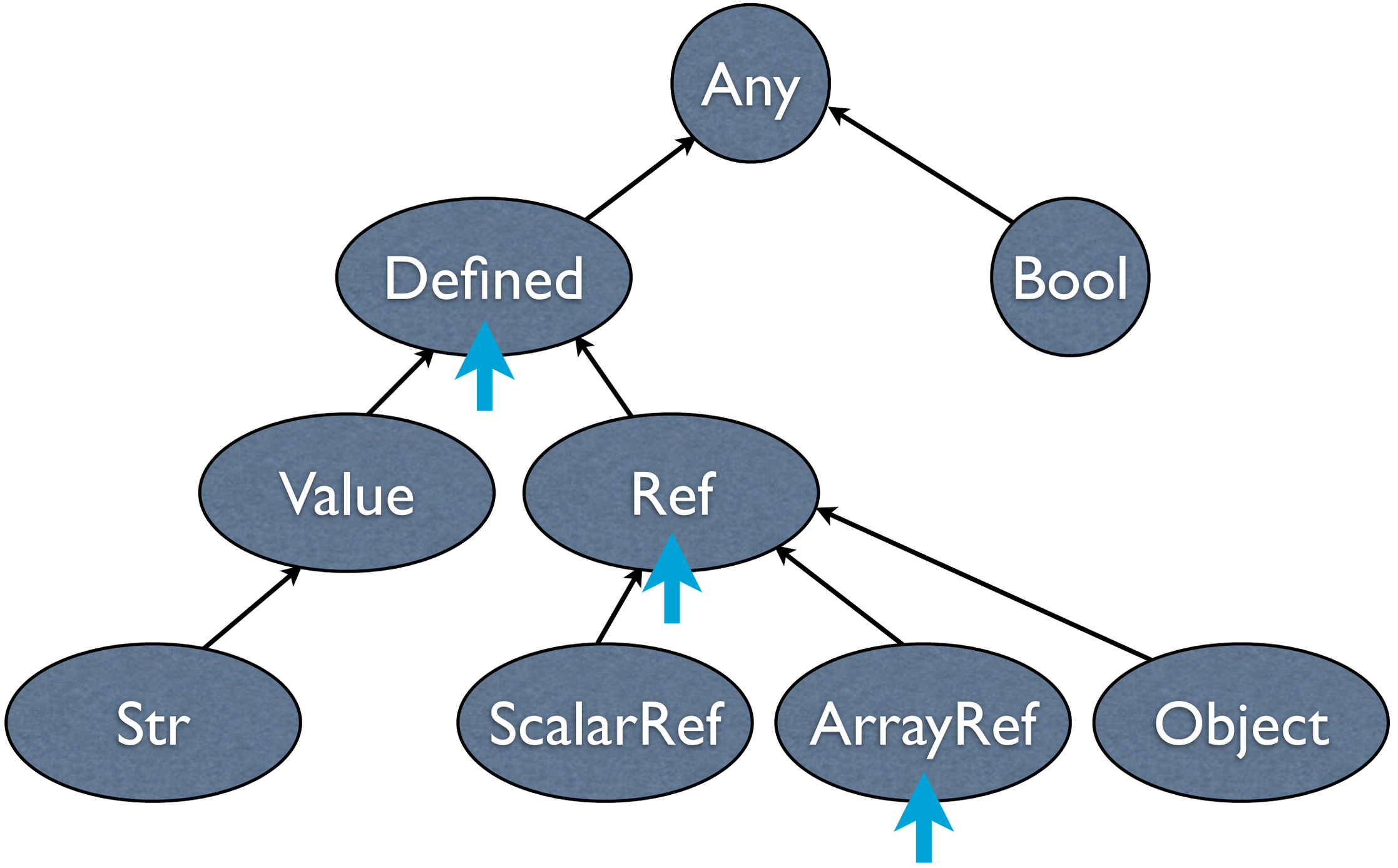
Friday, June 7, 13

So let's say we want to validate that something is an ArrayRef.



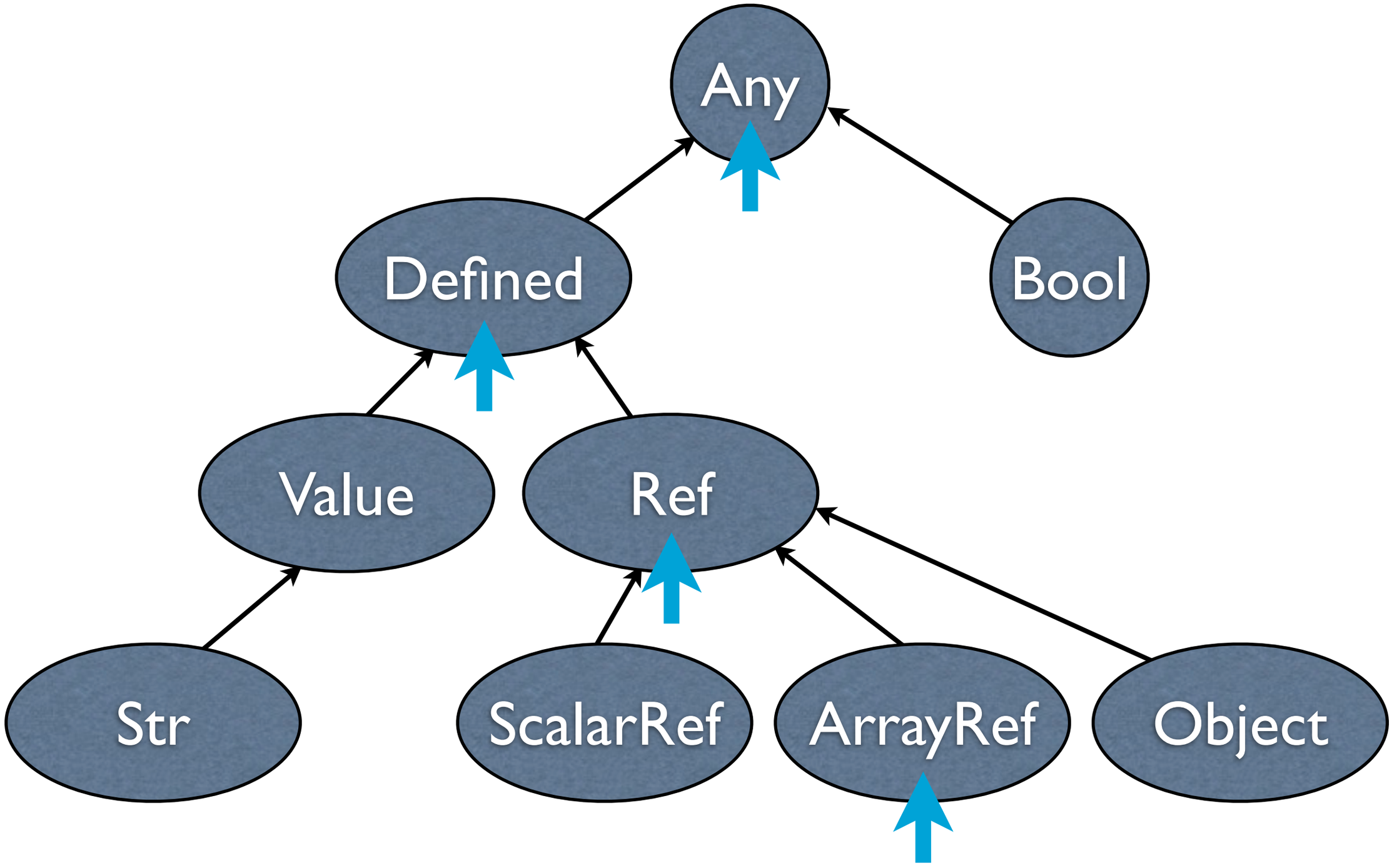
Friday, June 7, 13

We start by validating against ArrayRef's supertype, which is Ref.



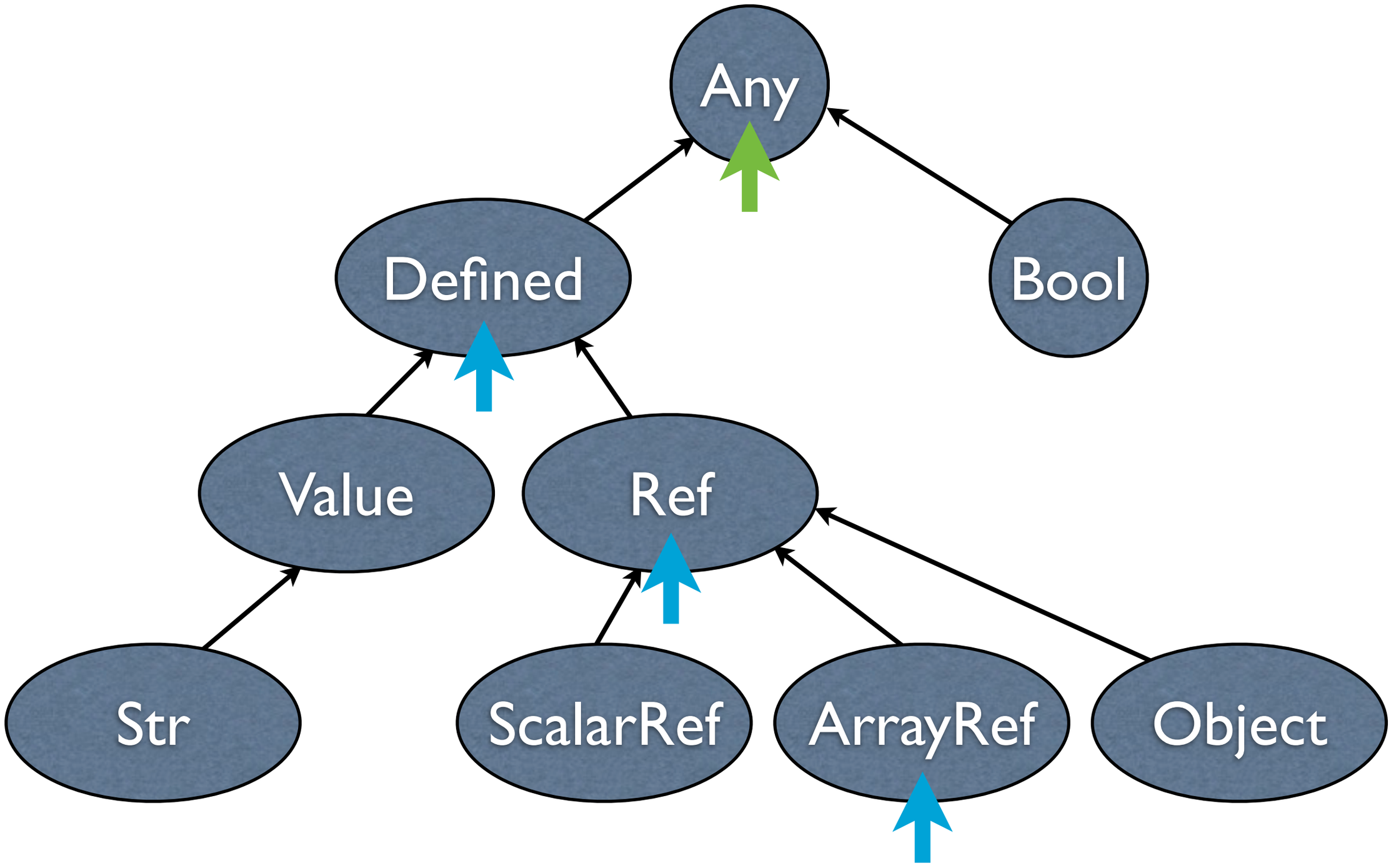
Friday, June 7, 13

Then we validate against Ref's supertype, which is Defined.



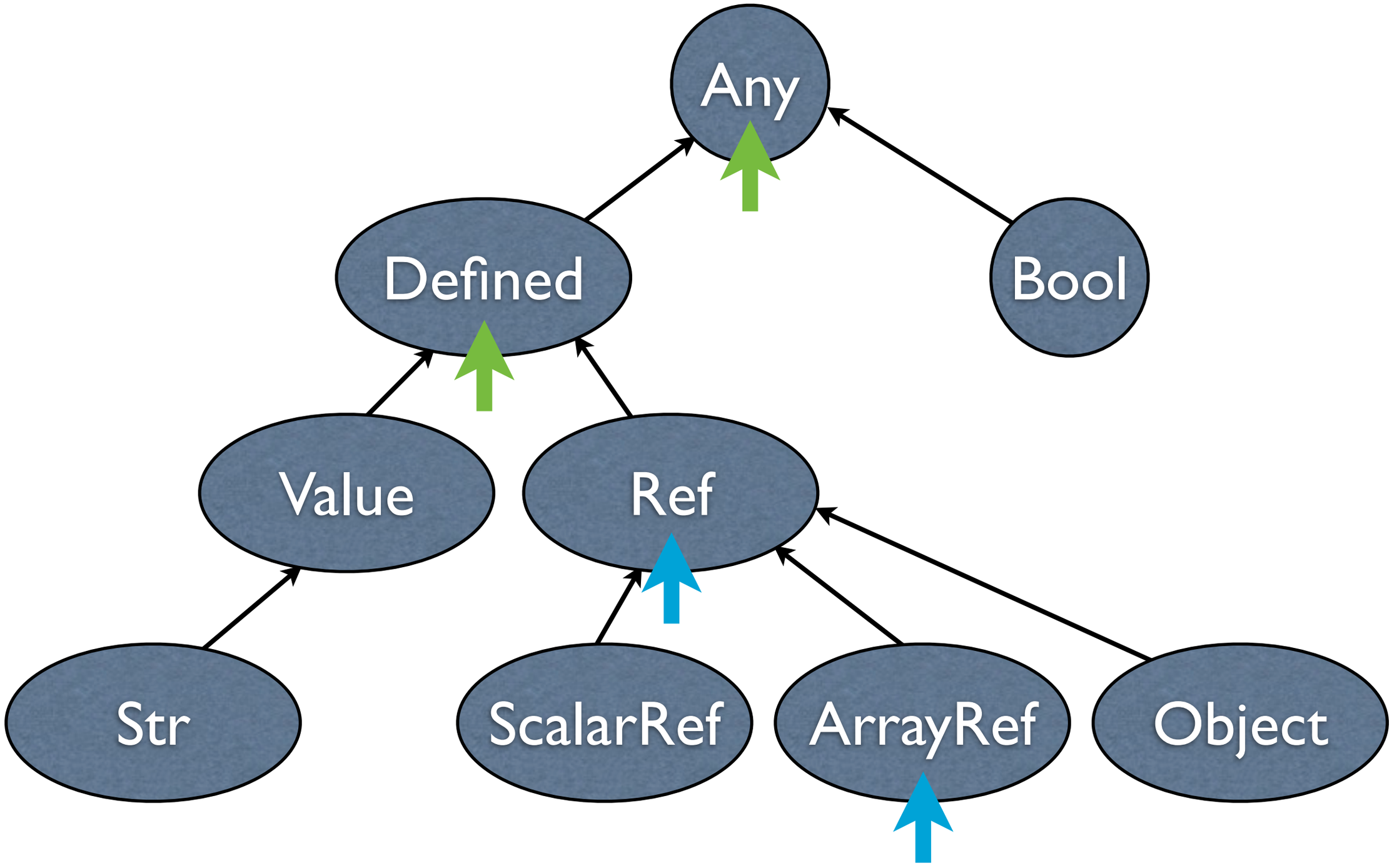
Friday, June 7, 13

Then we validate against Defined's supertype, which is Any.



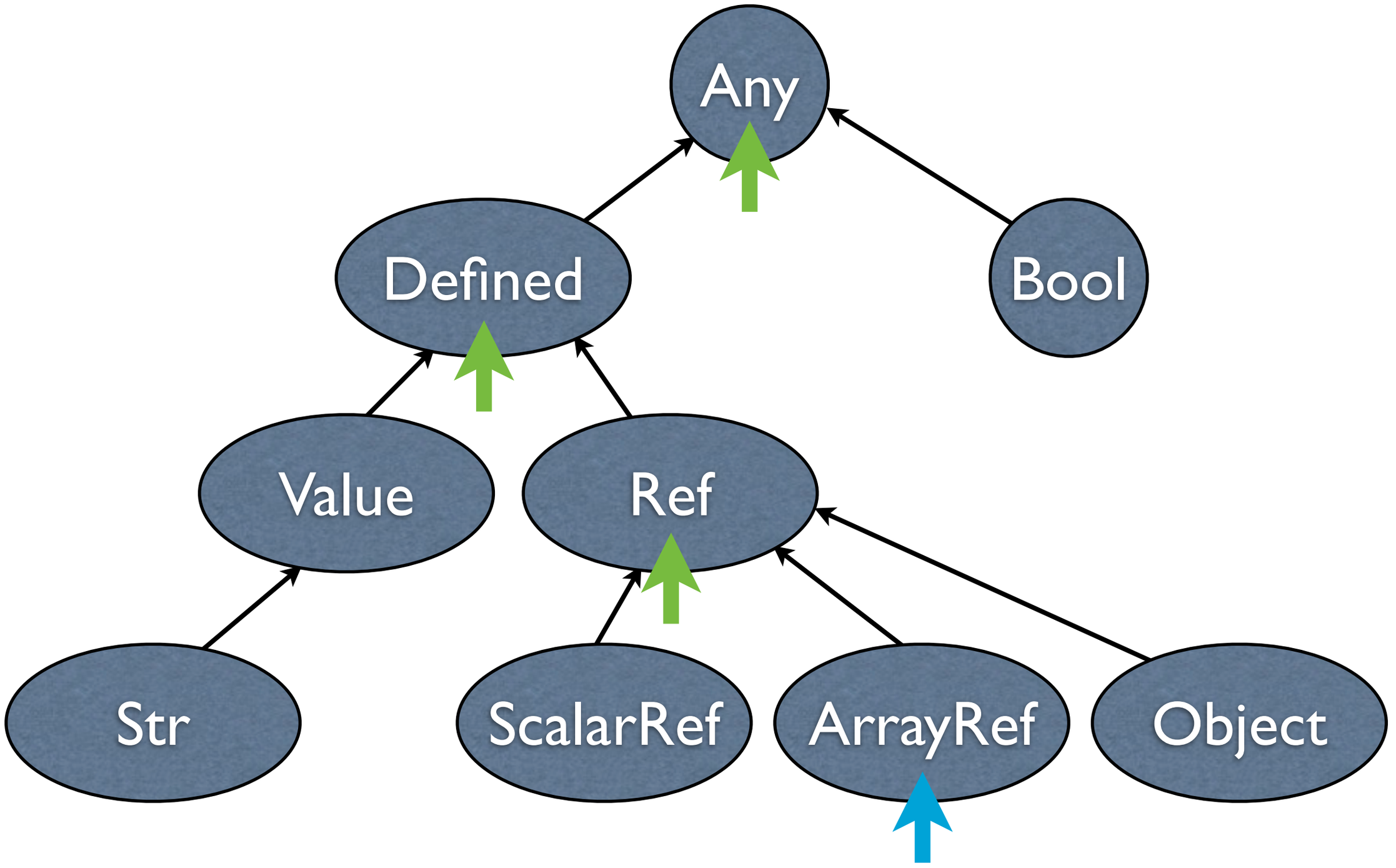
Friday, June 7, 13

Any is at the top of the true, and it always passes anything you're validating, so that's a green.



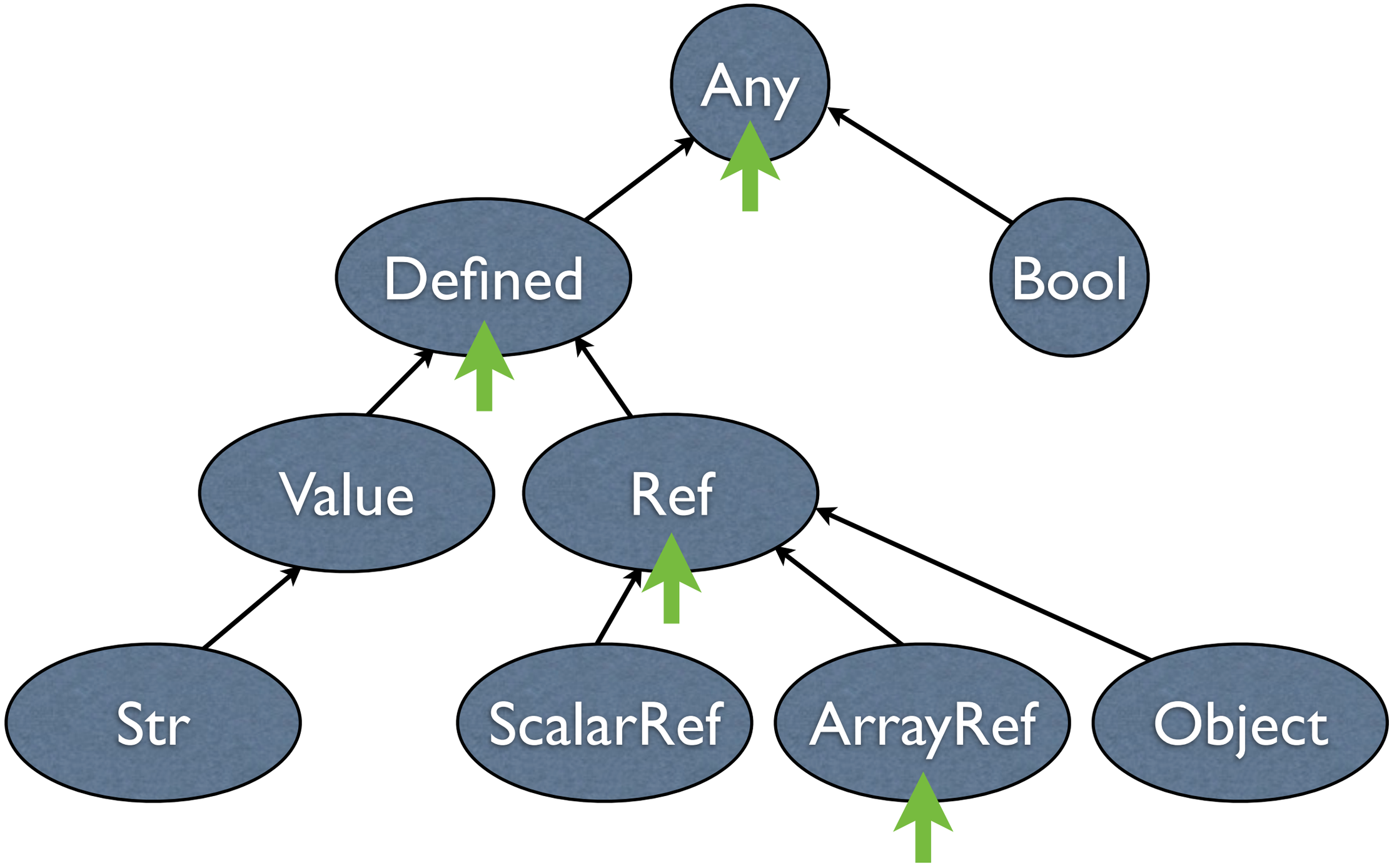
Friday, June 7, 13

Then we check the type constraint of Defined. As you'd expect, the Defined type checks that the defined builtin returns true.



Friday, June 7, 13

Next, Ref. Similarly that type just checks whether the ref builtin returns true.



Friday, June 7, 13

Finally, we're back down to ArrayRef. So at this point we know that the value is defined and that it's a reference. Now we just need to check that the type of reference is ARRAY. So we call the ref builtin again but this time make sure that it returns the particular string ARRAY.

Types add up!

- `{ 1 }` `# Any`
- `{ defined($_) }` `# Defined`
- `{ ref($_) }` `# Ref`
- `{ ref($_) eq ‘ARRAY’ }` `# ArrayRef`

Friday, June 7, 13

We had to run these four functions, three of which call additional functions, just to confirm that something is an array reference. All those checks, and the overhead of those function calls, add up.

Types add up!

- `{ 1 }` `# Any`
- `{ defined($_) }` `# Defined`
- `{ ref($_) }` `# Ref`
- `{ ref($_) eq ‘ARRAY’ }` `# ArrayRef`

Friday, June 7, 13

This is really the only check we need, right? If the value is undefined, or if it’s not a reference, then the return value of ref won’t be ‘ARRAY’ anyway.

inline_as

- Optimize your type constraint check!
- Your parent constraints will not be called
- You give Moose a string, not a subroutine
- Formerly “optimized_as”

Friday, June 7, 13

Moose provides a tool, inline_as, to optimizes your type constraints. inline_as enables you to specify a complete, self-contained type constraint check so that Moose won’t need to consult your parent type constraints.
inline_as produces a string of code that will be invoked on values to check their values. This is so we can inline them directly into accessors and constructors, saving function call overhead.
inline_as started out as a different option called optimized_as, but we made it better so we gave it the new name and deprecated the old optimized_as.

Types add up!

- `{ 1 }` `# Any`
- `{ defined($_) }` `# Defined`
- `{ ref($_) }` `# Ref`
- `{ ref($_) eq ‘ARRAY’ }` `# ArrayRef`

Friday, June 7, 13

inline_as is exactly for this kind of situation. As we discussed, we don’t need those other checks because ref already has all that logic for discarding undef and non-references.

Type validation

```
subtype ‘PositiveInt’
    as ‘Int’,
    where { $_ > 0 },
    inline_as {
        $_[1] . ‘ && ‘ .
        $_[1] . ‘ =~ /^\d+$/’
};
```


Friday, June 7, 13

It’s a little bit tricky to write an inline_as, but it’s worth it. Moose will invoke the function you pass into inline_as at compile time, and it returns a string of code which will validate the value. Moose will pass in your type constraint object as the first parameter, in case you need it. The second parameter, $_[1], contains the name of the variable where the value is stored. So here our PositiveInt check is first confirming that the value is true (to weed out undef, especially zero, etc.) and then it checks that the value matches digits only. It works differently from the ordinary $_ > 0 because we don’t have the benefit of all our supertype checks.

`inline_as`: Our free lunch?

Friday, June 7, 13

This is about as close as Moose can get to giving you a free lunch. In exchange for writing a tiny bit more code, you get blazing fast type constraints everywhere. It goes without saying that all of Moose's type constraints declare an inline_as. Do yours?

Recap

- First rule of optimization: don’t!
- Second rule of optimization: profile
- Devel::NYTProf, DTrace
- Fix your data structures and algorithms
- Can you cache?

Friday, June 7, 13

So to briefly recap what we talked about today.
The first rule of optimization is: don’t. You almost certainly have better ways to spend your time.
The second rule is: profile. Computers are complicated and you want to be sure that you’re spending your time fixing the right bottlenecks.
Devel::NYTProf and DTrace are great tools for investigating your performance.
Improving your data structures and algorithms are going to give you the biggest bang for the buck.
And, ask yourself if you can cache with tools like the Memoize module or memcached.

Recap

- `make_immutable` (almost) always
- Don’t overuse attributes
- Role overhead is compile-time
- Lazy
- Types are expensive...
- ...so use `inline_as` to make them fast

Friday, June 7, 13

You should use make_immutable, but when you intentionally remove it, leave a note.
Only use attributes for things that should be attributes; methods are still useful in Moose.
Role application might seem like it has runtime overhead but because it inlines methods and attributes, its overhead is compile-time only.
Lazy is useful for deferring expensive computation but don’t use it for all of your attributes blindly because of its accessor overhead.
Types, especially coercions, canbe expensive. Consider using them only at the boundaries of your APIs where they’ll catch the most bugs.
And when you do declare a type, spend the extra few minutes to declare an inline_as because it will make your type as fast as possible.