Puppet **A**s a **L**ibrary (PAL) - predocs

This document describes PAL ("Puppet As a Library"), and API designed to help when writing ruby applications that makes use of Puppet. Earlier, a user wanting to use Puppet as a library would have to assemble all the bits and pieces into a correct configuration and where "correct" varies between versions of Puppet.

Pal only works with Ruby >= 2.0.0 as it is using Ruby "args by name" in the API methods.

This is pre-docs describing functionality being developed as this document is written!

Require 'puppet_pal', not 'puppet'

The first step is to require pal.

```
require 'puppet_pal'
```

This should be done instead of requiring 'puppet' since requiring 'puppet' is not API. While 'puppet_pal' currently requires 'puppet', the API may in the future require a subsection of puppet (the part actually being used) in order to speed up loading.

Configure an Environment

Most operations in puppet require an environment - a definition of where to find puppet logic, modules, hiera configuration and data, as well as certain settings. Setting up and creating one without Pal is a bit of black art due to the flexibility provided in the underlying implementation. Now, in Pal this is quite easy. There are two entry points - one that creates a temporary environment out of what is given to the API, and one that is based on an existing puppet environment (a file system directory).

In both cases, the logic you want to run with the environment in effect is given in a block that is given an instance of `Puppet::Pal` as an argument (to make it easier to further access PAL).

In a Temporary Environment

In a temporary environment - minimum with modules

```
require 'puppet_pal'

result = Puppet::Pal.in_tmp_environment('pal_env', modulepath: ['/tmp/testmodules'])
do |pal|
  # do things with pal here
end
```

Arguments to `in_tmp_environment`

argument name	default value	named param	description
`env_name`	**required**	no	the name of the env must be given as such a name is required in many puppet operations/output
`modulepath`	`[]`	✓	one or several directories where modules can be found
`settings_hash`	`{}`	✓	a hash of setting name to value mapping - *currently unused*
`facts`	`nil`	✓	a hash of fact name to value mapping to use in the operation - when set to `nil`, the facts from the localhost are used (and that is expensive to execute).
`variables`	`{}`	✓	a hash of variable name (without leading '`$`') to value mapping - may set values in any name scope - to be used with caution as variables are immutable
`&block`	**required**	no	the block with code to execute

In an Existing Environment

In an existing environment - minimum with env configured modulepath

```
require 'puppet_pal'

result = Puppet::Pal.in_environment('pal_env') do |pal|
  # do things with pal here
end
```

The "existing environment" is either a directory that has puppet environment compliant layout that is referenced directly with a path to the directory, or it is a directory on the environment path. The former (directly referencing a directory) avoids the potential problem of puppet trying to find an environment in a directory that is not a compliant environment layout. It also enables using a directory with a different name than the wanted env name. See parameters `envpath` and `env_dir` for more details.

Arguments to `in_environment`

argument name	default value	named param	description
`env_name`	**required**	no	the name of an existing env must be given and there must be such an environment on puppet's environmentpath.
`pre_modulepath`	`[]`	✓	one or several directories where modules can be found and where the modules should be added first in the resulting modulepath
`modulepath`	`nil`	✓	one or several directories where modules can be found - uses the env configured modulepath if `nil`, otherwise overrides the env's modulepath.
`post_modulepath`	`[]`	✓	like `pre_modulepath` but appends to the effective modulepath.
`settings_hash`	`{}`	✓	a hash of setting name to value mapping - *currently unused*
`env_dir`	`nil`	✓	a directory to use as the environment directory - instead of finding an environment on the `envpath`. Mutually exclusive with `envpath`. If

			`env_dir` is specified, the directory may have a name different from the given `env_name`.
`envpath`	`nil`	✓	One or several directories where named environments can be found. Mutually exclusive with `env_dir`. Defaults to puppet's configured `environmentpath` if not given and `env_dir` is not given.
`facts`	`nil`	✓	a hash of fact name to value mapping to use in the operation - when set to `nil`, the facts from the localhost are used (which is an expensive operation).
`variables`	`{}`	✓	a hash of variable name (without leading '`$`') to value mapping - may set values in any name scope - to be used with caution as variables are immutable
`&block`	**required**	no	the block with code to execute

NOTE: The PAL API is not recursive - you are not allowed to call in_tmp_environment, or in_environment while in the context of the block given to those calls (applies recursively). There is no detection if you do - just gremlins...

Making Puppet do stuff

In the block given to `in_tmp_environment`, or `in_environment` we want to perform operations with puppet. At this point in the setup the environment has been configured for use, but some slow operations are delayed until it is known what is going to be used. At present, all operations require a "compiler" of some sort - and when it is created there is resolution of facts for a node, scopes etc. are configured and global variables are set, loaders are configured, etc.

It is important to understand that setting up the environment and compiler is a quite costly operation - while you could do that around each operation, it adds significant overhead and the state of scopes etc. is then not preserved. You decide if you want to use the same env and its configuration with multiple separate instances of a compiler, or if you want to use the same compiler instance throughout. You can specify variables and facts at the env level (for those that you want set by default in each compiler instance, and you can override the variables and facts when creating the compiler.

Once there is an environment we can do things and this requires a "compiler" (of some sort). In PAL this is abstracted in `Puppet::Pal::Compiler`, for which there is currently one subclass `Puppet::Pal::ScriptCompiler`. The `ScriptCompiler` is configured to handle the

Puppet Language with task/plan support and restrictions on using catalog compilation expressions and statements. At some point in the future it is expected that PAL will also support catalog compilation.

The rationale for splitting up the PAL API in two (first environment, and then a compiler) is due to the more static nature of the environment (the initial logic and modules typically does not change), whereas you may want to have multiple independent runs (compilations) take place with an environment in effect. This design also enables `ScriptCompiler` and a future `CatalogCompiler` to have compiler specific API.

A script compiler is obtained like this:

Use `with_script_compiler` to get a `ScriptCompiler`

```
require 'puppet_pal'

result = Puppet::Pal.in_environment('pal_env') do | pal |
  pal.with_script_compiler do | compiler |
    # do things with compiler
  end
end
```

Arguments to `with_script_compiler`

argument name	default value	named param	description
`configured_by_env`	`false`	✓	When `false`, `manifest_file` or `code_string` (or neither) defines any initialization logic to evaluate. When `true`, the puppet *settings* for `manifest` and `code` are used.
`manifest_file`	`nil`	✓	A reference to a file with puppet language code to parse and evaluate before any other operations take place in the compiler. Mutually exclusive with `code_string`. Can only be used when `configured_by_env` is `false`.
`code_string`	`nil`	✓	A string containing puppet language code to parse and evaluate before any other operations take place in the compiler. Mutually

			exclusive with `manifest_file`. Can only be used when `configured_by_env` is `false`.
`facts`	`nil`	✓	a hash of fact name to value mapping to use in the operation - when set to `nil`, the facts from the localhost are used. If given, merges on top of the facts set by `in_tmp_environment` or `in_environment`.
`variables`	`{}`	✓	a hash of variable name (without leading '`$`') to value mapping - may set values in any name scope - to be used with caution as variables are immutable. If given, merges on top of the variables set by `in_tmp_environment` or `in_environment`.

Note: that the script compiler will use the default settings for Puppet for things like `manifest`*, and this setting is by default the environment's* `manifests` *directory.* ***It is however illegal to use a directory as the manifest when using the script compiler****.*

Operations on `ScriptCompiler`
`call_function(function_name, *args, &block)`
`evaluate_file(file)`
`evaluate_string(source_string, source_file=nil)`
`evaluate(ast)`
`evaluate_literal(ast)`
`type(type_string)`
`create(data_type, *args)`
`plan_signature(plan_name)`
`task_signature(task_name)`
`function_signature(function_name)`
`list_plans(filter_regexp=nil)`
`list_tasks(filter_regexp=nil)`
`list_functions(filter_regexp=nil)`

`lex_string(code_string, source_file=nil)`		
`lex_file(file)`		
Details per method		
`call_function(function_name, *args, &block)`		
	Calls a function given by name with arguments specified in an `Array`, and optionally accepts a code block.	
parameters		
	`String function_name`	the name of the function to call
	`Any *args`	the arguments to the function
	`Callable block`	an optional block if the given function accepts one - this can be a regular Ruby block as they are compatible with the Puppet Language lambdas
returns		what the called function returns
`evaluate_file(file)`		
	Loads an existing file with puppet language source, parses, validates, evaluates and returns the result of the file's last expression.	
parameters		
	`String file`	the path to the .pp file
returns		what the evaluated result of the last expression in the file is
`evaluate_string(source_string, source_file=nil)`		
	Accepts a string with puppet language source, parses, validates, evaluates and returns the result of the string's last expression	
parameters		
	`String source_string`	a puppet language string
	`String source_file`	a reference to the source location/origin of the `source_string`. Does not have to be an existing path - by convention use `< >` around a symbolic name; for example `<commandline>`. Defaults to `<unknown>` if

		not given. If the string was read from a file, do use the actual filename.
returns		what the evaluated result of the last expression in the `source_string` is
`evaluate(ast)`		
	Evaluates the validated AST obtained from parse_string, or parse_file and returns the result of that evaluation.	
parameters		
	`Puppet::Pops::Model::Program ast`	the parsed "program"
returns		what the evaluated result of the last expression in the file is
`evaluate_literal(ast)`		
	Evaluates the validated AST representing a literal value obtained from parse_string, or parse_file and returns the result of that evaluation. An error is raised if the ast does not represent a literal value.	
parameters		
	`Puppet::Pops::Model::Program ast`	the parsed "program"
returns		the literal value the ast represents
`plan_signature(plan_name)`		

	Returns a `Puppet::Pal::PlanSignature` object describing the signature of a plan (its parameters and return type).
	
	Example: checking if plan can be called with given arguments hash:
	
	`signature = compiler.plan_signature('mymodule::myplan')`
	`raise "Plan not found" if signature.nil?`
	`signature.callable_with?(args_hash) # true if acceptable`	

parameters		
	`String plan_name`	the name of a plan to get a `Callable` for
returns		a `Puppet::Pal::PlanSignature` if the plan exists, or `nil` otherwise.

function_signature(function_name)

Returns a `Puppet::Pal::FunctionSignature` object describing the signature of a function (all its overloaded implementations each with parameters and return type).

Example: checking if function can be called with given set of arguments:

```
signature = compiler.function_signature('mymodule::myfunc')
raise "Function not found" if signature.empty?
signature.callable_with?(args_array)
# or if function accepts/requires a lambda/proc
signature.callable_with?(args_array, a_proc)
```

Example: getting all overloaded implementations as Array[Callable]

```
# The FunctionSignature can return details per overloaded
# implementation. The callables() method returns an array
# of one or more Puppet::Pops::Types::PCallableType,
# one per overloaded implementation.
#
signature.callables
```

parameters	
`String function_name`	the name of a function to get a `Callable` types for - one per possible dispatch
returns	a `Puppet::Pal::FunctionSignature` if the function exists, or `nil` otherwise.

task_signature(task_name)

Returns a `Puppet::Pal::TaskSignature` object describing the signature of a task (its parameters and return type).

For example:

```
signature = compiler.task_signature('mymodule::mytask')
raise "Task not found" if signature.nil?
signature.runnable_with?(args_hash) # true if acceptable
```

parameters	
`String task_name`	the name of a task to get a signature for
returns	a `Puppet::Pal::TaskSignature` if the function exists, or `nil` otherwise.

`parse_file(file)`		
	The same as parse_string, but parsing an entire file. For example: `program = compiler.parse_file(/somehwere/test.pp)`	
parameters		
	`String file`	the puppet language file to parse
returns		The AST element `Puppet::Pops::Model::Program` if the input is a valid puppet language "program"
`list_plans(filter_regexp=nil)`		

	Returns an array with all plans with a name that matches the optional regular expression filter. Returns a `Puppet::Pops::Loader::TypedName` with information.
	
	The TypedName has several attributes - one being name, the fully qualified name of the plan.
	
	Example: output the name of every plan:
	`compiler.list_plans().each { |tn| puts tn.name }`
	
	Example: output the name of all plans in mymodule:
	`compiler.list_plans(/^mymodule::/).each { |tn| puts tn.name }`	

parameters		
	`filer_regexp`	an optional regular expression that all listed elements must match - by default, all available are returned.
returns		Array of `Puppet::Pops::Loader::TypedName` or an empty array if non matched.
`list_tasks(filter_regexp=nil)`		
	Returns an array with all tasks with a name that matches the optional regular expression filter. Returns a `Puppet::Pops::Loader::TypedName` with information. The TypedName has several attributes - one being name, the fully qualified name of the task. **Example: output the name of every task:** `compiler.list_tasks().each { |tn| puts tn.name }`	

	Example: output the name of all tasks in mymodule: `compiler.list_tasks(/^mymodule::/).each { |tn| puts tn.name }`
parameters	
`filer_regexp`	an optional regular expression that all listed elements must match - by default, all available are returned.
returns	Array of `Puppet::Pops::Loader::TypedName` or an empty array if non matched.
`list_functions(filter_regexp=nil)`	

	Returns an array with all functionsks with a name that matches the optional regular expression filter. Returns a `Puppet::Pops::Loader::TypedName` with information.
	
	The TypedName has several attributes - one being name, the fully qualified name of the function.
	
	Example: output the name of every function:
	`compiler.list_functions().each { |tn| puts tn.name }`
	
	Example: output the name of all tasks in mymodule:
	`compiler.list_functions(/^mymodule::/).each { |tn| puts tn.name }`

parameters	
`filer_regexp`	an optional regular expression that all listed elements must match - by default, all available are returned.
returns	Array of `Puppet::Pops::Loader::TypedName` or an empty array if non matched.

Examples

Examples: TBD

- ~~type checking arguments to a Task - shown above in task_signature method~~
- ~~type checking arguments to a Plan - shown above in plan_signature method~~
- ~~type checking arguments to a Function - shown above in function_signature method~~
- general type checking using the type system
- ~~reporting a type mismatch error~~
- examples of evaluation
    - in same compiler - variable values still there

Type checking with PlanSignature, TaskSignature and FunctionSignature

The three kinds of signatures returned from `plan_signature(name)`, `task_signature(name)`, and `function_signature(name)` work the same way - there is one method to check if a given set of arguments are acceptable or not. The method is named "`callable_with?`" for plan and function, and `runnable_with?` for tasks.

The argument checking methods behave in a similar way; they differ in that a `FunctionSignature`'s `callable_with?` takes its arguments in an array, and accepts an optional Proc/lambda. What they have in common is that they return true if arguments are acceptable, and false otherwise. They also accept a block that is yielded to with a type-mismatch error string if arguments were not acceptable. This error message is formatted and may extend over multiple lines (typically if this is a function with overloaded implementations). The error message string may have leading space per line as indentation as that may be required to enable a human to read the output correctly.

Example:

```
signature = compiler.find_function('lookup')
args = [42]
signature.callable_with?(args) do |msg|
  raise ArgumentError.new("Given arguments to 'lookup' does not match
- expected one of: #{msg}")
end

# Produces this very detailed output:

Given arguments to 'lookup' does not match, expected one of:
  (NameType = Variant[String, Array[String]] name, ValueType = Type value_type?, MergeType =
Variant[String[1, default], Hash[String, Scalar]] merge?)
    rejected: parameter 'name' expects a NameType = Variant[String, Array[String]] value, got
Integer
  (NameType = Variant[String, Array[String]] name, Optional[ValueType] value_type,
Optional[MergeType] merge, DefaultValueType = Any default_value)
    rejected: expects 4 arguments, got 1
  (NameType = Variant[String, Array[String]] name, ValueType = Type value_type?, MergeType =
Variant[String[1, default], Hash[String, Scalar]] merge?)
    rejected: parameter 'name' expects a NameType = Variant[String, Array[String]] value, got
Integer
  (OptionsWithName = Struct[{'name' => NameType = Variant[String, Array[String]], 'value_type' =>
Optional[ValueType = Type], 'default_value' => DefaultValueType = Any, 'override' =>
Optional[Hash[String, Any]], 'default_values_hash' => Optional[Hash[String, Any]], 'merge' =>
Optional[MergeType = Variant[String[1, default], Hash[String, Scalar]]]}] options_hash, BlockType
= Callable[NameType = Variant[String, Array[String]]] block?)
```

```
    rejected: parameter 'options_hash' expects an OptionsWithName = Struct[{'name' => NameType =
Variant[String, Array[String]], 'value_type' => Optional[ValueType = Type], 'default_value' =>
DefaultValueType = Any, 'override' => Optional[Hash[String, Any]], 'default_values_hash' =>
Optional[Hash[String, Any]], 'merge' => Optional[MergeType = Variant[String[1, default],
Hash[String, Scalar]]]}] value, got Integer
  (Variant[String, Array[String]] name, OptionsWithoutName = Struct[{'value_type' =>
Optional[ValueType = Type], 'default_value' => DefaultValueType = Any, 'override' =>
Optional[Hash[String, Any]], 'default_values_hash' => Optional[Hash[String, Any]], 'merge' =>
Optional[MergeType = Variant[String[1, default], Hash[String, Scalar]]]}] options_hash, BlockType
= Callable[NameType = Variant[String, Array[String]]] block?)
    rejected: expects 2 arguments, got 1
```

For constructs that do not have overloading ('lookup' is one of the most overloaded) there is naturally just one mismatch entry and one 'rejected'. The more complex 'lookup' is shown as it also shows why you may want to use a simpler "Arguments are not acceptable to the function - see the documentation how it can be called", and then perhaps log the complete error message, or only show it in debug mode or similar.

Checking a Value Against a Data Type

To check a value against a data type, first obtain the data type using the type() method, and then use the methods on that data type.

# is a value in an integer range?
val = 42
compiler.type('Integer[0,100]').instance?

Using hiera and lookup

This is as simple as calling the `lookup` function. Here is an example, where an existing environment is used (and where it is expected that puppet is configured with/without a global hiera.yaml, and that there may be an environment hiera.yaml as well as hiera.yaml files in modules in the module path.

Perform lookup in hiera

```
require 'puppet_pal'

result = Puppet::Pal.in_environment('pal_env') do | pal |
  pal.with_script_compiler do | c |
    array_t = compiler.type('Array[Integer]')
    c.call_function('lookup', 'mymodule::myarray', array_t, 'unique') {|key| [] }
  end
end
```

This example:

- Uses an existing environment on disk named 'pal_env' found on the configured `environmentpath`
- Creates a script compiler without any initialization puppet logic
- Gets a data type to use as validation of the return value from the lookup (an array of integer values is expected), the data type is assigned to the `array_t` variable.
- Looks up the key `'mymodule::myarray'` with `'unique'` merge by calling the compiler's '`call_function` method.
- A Ruby block/lambda is given to `call_function` and it is passed on the call to '`lookup()`'
- If the key `'mymodule::myarray'` was not found the given code block is called and it then returns an empty array.

PAL and Catalog Compilation

Use `with_catalog_compiler` instead of `with_script_compiler`.
To render the catalog call `with_json_encoding` on the compiler given to `with_catalog_compiler` and call its `encode` method. By default that produces pretty printed JSON in rich-data encoding.

```
result = Puppet::Pal.in_tmp_environment('pal_env', modulepath: modulepath, facts: node_facts) do |pal|
 pal.with_catalog_compiler {|c|
   c.evaluate_string("notify {'test': message => /a regexp/}")
   c.with_json_encoding() {|encoder| encoder.encode }
 }
end
```

There is obviously more to say - see https://github.com/puppetlabs/puppet/pull/6949
meanwhile...