# Ruby master - Feature #16254

## MRI internal: Define built-in classes in Ruby with `__intrinsic__` syntax

10/15/2019 06:57 PM - ko1 (Koichi Sasada)

| | |
|---|---|
| **Status:** | Closed |
| **Priority:** | Normal |
| **Assignee:** | ko1 (Koichi Sasada) |
| **Target version:** | |

**Description**

# Abstract

MRI defines most of built-in classes in C with C-APIs like rb_define_method().
However, there are several issues using C-APIs.

A few methods are defined in Ruby written in prelude.rb.
However, we can not define all of classes because we can not touch deep data structure in Ruby.
Furthermore, there are performance issues if we write all of them in Ruby.

To solve this situation, I want to suggest written in Ruby with C intrinsic functions.
This proposal is same as my RubyKaigi 2019 talk https://rubykaigi.org/2019/presentations/ko1.html.

# Terminology

- C-methods: methods defined in C (defined with rb_define_method(), etc).
- Ruby-methods: methods defined in Ruby.
- ISeq: The body of RUbyVM::InstructionSequence object which represents bytecode for VM.

# Background / Problem / Idea

## Written in C

As you MRI developers know, most of methods are written in C with C-APIs.
However, there are several issues.

### (1) Annotation issues (compare with Ruby methods)

For example, C-methods defined by C-APIs doesn't have parameters information which are returned by Method#parameters, because there is way to define parameters for C methods.
There are proposals to add parameter name information for C-methods, however, I think it will introduce new complex C-APIs and introduce additional overhead on boot time.

-> Idea; Writing methods in Ruby will solve this issue.

### (2) Annotation issues (for further optimization)

It is useful to know the methods attribute, for example, the method causes no side-effect (a pure method).
Labeling all of methods including user program's methods doesn't seem good idea (not Ruby-way). But I think annotating built-in methods is good way because we can manage (and we can remove them when we can make good analyzer).

There are no way to annotate this kind of attributes.

-> Idea: Writing methods in Ruby will make it easy to introduce new annotations.

### (3) Performance issue

There are several features which are slower in C than written in Ruby.

- exception handling (rb_ensure(), etc) because we need to capture context with setjmp on C-methods. Ruby-methods doesn't need to capture any context for exception handling.

- Passing keyword parameters because Ruby-methods doesn't need to make a Hash object to pass the keyword parameters if they are passed with explicit keyword parameters (foo(k1: v1, k2: v2)).

-> Idea: Writing methods in Ruby makes them faster.

### (4) Productivity

It is tough to write some features in C:

For example, it is easy to write rescue syntax in Ruby:

```
# in Ruby
def dummy_func_rescue
  nil
rescue
  nil
end
```

But it is difficult to write/read in C:

```
static VALUE
dummy_body(VALUE self)
{
    return Qnil;
}
static VALUE
dummy_rescue(VALUE self)
{
    return Qnil;
}
static VALUE
tdummy_func_rescue(VALUE self)
{
    return rb_rescue(dummy_body, self, dummy_rescue, self);
}
```

(trained MRI developer can say it is not tough, though :p)

-> Idea: Writing methods in Ruby makes them easy.

### (5) API change

To introduce Guild, I want to pass a "context" parameter (as a first parameter) for each C-functions like mrb_state on mruby.
This is because getting it from TLS (Thread-local-storage) is high-cost operation on dynamic library (libruby).

Maybe nobody allow me to change the specification of functions used by rb_define_method().

-> Idea: But introduce new method definition framework, we can move and change the specification, I hope.
Of course, we can remain current rb_define_method() APIs (with additional cost on Guild available MRI).

## Written in Ruby in prelude.rb

There is a file prelude.rb which are loaded at boot time.
This file is used to define several methods, to reduce keyword parameters overhead, for example (IO#read_nonblock, TracePoint#enable).

However, writing all of methods in Ruby is not possible because:

- (1) feasibility issue (we can not access internal data structure)
- (2) performance issue (slow in general, of course)
- (3) atomicity issue (GVL/GIL)

To solve (1), we can provide low-level C-methods to implement high-level (normal built-in) methods. However issues (2) and (3) are not solved.
(From CS researchers perspective, making clever compiler will solve them, like JVM, etc, But we don't have it yet)

-> Idea: Writing method body in C is feasible.

# Proposal

(1) Introducing intrinsic mechanism to define built-in methods in Ruby.
(2) Load from binary format to reduce startup time.

## (1) Intrinsic function

### Calling intrinsic function syntax in Ruby

To define built-in methods, introduce special Ruby syntax __intrinsic__.func(args).
In this case, registered intrinsic function func() is called with args.

In normal Ruby program, __intrinsic__ is a local variable or a method.
However, running on special mode, they are parsed as intrinsic function call.

Intrinsic functions can not be called with:

- block
- keyword arguments
- splat arguments

### Development step with intrinsic functions

(1) Write a class/module in Ruby with intrinsic function.

```
# string.rb
class String
  def length
    __intrinsic__.str_length
  end
end
```

(2) Implement intrinsic functions

It is almost same as functions used by rb_define_method().
However it will accept context parameter as the first parameter.

(rb_execution_context_t is too long, so we can rename it, rb_state for example)

```
static VALUE
str_length(rb_execution_context_t *ec, VALUE self)
{
  return LONG2NUM(RSTRING_LEN(self));
}
```

(3) Define an intrinsic function table and load .rb file with the table.

```
Init_String(void)
{
  ...
  static const rb_export_intrinsic_t table[] = {
    RB_EXPORT_INTRINSIC(str_length, 0), // 0 is arity
    ...
  };
  rb_vm_builtin_load("string", table);
}
```

### Example

There are two examples:

(1) Comparable module: https://gist.github.com/ko1/7f18e66d1ae25bb30c7e823aa57f0d31
(2) TracePoint class: https://gist.github.com/ko1/969e5690cda6180ed989eb79619ca612

## (2) Load from binary file with lazy loading

Loading many ".rb" files slows down startup time.

We have ISeq#to_binary method to generate compiled binary data so that we can eliminate parse/compile time.
Fortunately, [Feature #16163] makes binary data small.
Furthermore, enabling "lazy loading" feature improves startup time because we don't need to generate complete ISeqs.
USE_LAZY_LOAD in vm_core.h enables this feature.

We need to combine binary. There are several way (convert into C's array, concat with objcopy if available and so on).

# Evaluation

Evaluations are written in my RubyKaigi 2019 presentation: https://rubykaigi.org/2019/presentations/ko1.html

Points:

- Calling overhead of Ruby mehtods with intrinsic functions
  - Normal case, it is almost same as C-methods using optimized VM instructions.
  - With keyword parameters, it is faster than C-methods.
  - With optional parameters, it is x2 slower so it should be solved (*1).
- Loading overhead
  - Requiring ".rb" files is about x15 slower than defining C methods.
  - Loading binary data with lazy loading technique is about x2 slower than C methods. Not so bad result.
  - At RubyKaigi 2019, the binary data was very huge, but [Feature #16163] reduces the size of binary data.

[*1] Introducing special "overloading" specifier can solve it because we don't need to assign optional parameters. First method lookup can be slowed down, but we can cache the method lookup results (with arity).

```
# example syntax
overload def foo(a)
  __intrinsic__.foo1(a)
end
overload def foo(a, b)
  __intrinsic__.foo2(a, b)
end
```

# Implementation

Done:

- Compile calling intrinsic functions (.rb)
- Exporting intrinsic function table (.c)

Not yet:

- Loading from binary mechanism
- Attribute syntax
- most of built-in class replacement

Now, miniruby and ruby (libruby) load '*.rb' files directly. However, ruby (libruby) should load compiled binary file.

# Discussion

## Do we rewrite all of built-in classes at once?

No. We can try and migrate them.

## Do we support intrinsic mechanism for C-extension libraries?

Maybe in future. Now we can try it on MRI cores.

## __intrinsic__ keyword

On my RubyKaigi 2019 talk, I proposed __C__, but I think __intrinsic__ is more descriptive (but a bit long).
Another idea is RubyVM::intrinsic.func(...).

I have no strong opinion. We can change this syntax until we expose this syntax for C-extensions.

## Can we support __intrinsic__ in normal Ruby script?

No. This feature is only for built-in features.
As I described, calling intrinsic function syntax has several restriction compare with normal method calls, so that I think they are not exposed as normal Ruby programs, IMO.

## Should we maintain intrinsic function table?

Now, yes. And we need to make this table automatically because manual operations can introduce mistake very easily.

Corresponding ".rb" file (trace_point.rb, for example) knows which intrinsic functions are needed.
Parsing ".rb" file can generate the table automatically.
However, we need a latest version Ruby to parse the scripts if they uses syntax which are supported by latest version of Ruby.

For example, we need Ruby 2.7 master to parse a script which uses pattern matching syntax.
However, the system's ruby (BASE_RUBY) should be older version. This is one of bootstrap problem.
This is "chicken-and-egg" problem.

There are several ideas.

(1) Parse a ".c" file to generate a table using function attribute.

```
INTRINSIC_FUNCTION static VALUE
str_length(...)
...
```

(2) Build another ruby parser with source code, "parse-ruby".

- 1. generate parse-ruby with C code.
- 2. run parse-ruby to generate tables by parsing ".rb" files. This process is written in C.
- 3. build miniruby and ruby with generated table.

We can make it, but it introduces new complex build process.

(3) Restrict ".rb" syntax

Restrict syntax which can be used by BASE_RUBY for built-in ".rb" files.
It is easy to list up intrinsic functions using Ripper or AST or ISeq#to_a.

(3) is most easy but not so cool.
(2) is flexible, but it needs implementation cost and increases build complexity.

## Path of '*.rb' files and install or not

The path of prelude.rb is <internal:prelude>. We have several options.

- (1) Don't install ".rb" files and make these path <internal:trace_point.rb>, for example.
- (2) Install ".rb" and make these paths non-existing paths such as <internal>/installdir/lib/builtin/trace_point.rb.
- (3) Install ".rb" and make these paths real paths.

We will translate ".rb" files into binary data and link them into ruby (libruby).
So the modification of installed ".rb" files are not affect the behavior. It can introduce confusion so that I wrote (1) and (2).

For (3), it is possible to load ".rb" files if there is modification (maybe detect by modified date) and load from them. But it will introduce an overhead (disk access overhead).

## Compatibility issue?

There are several compatibility issues. For example, TracePoint c-call events are changed to call events.
And there are more incompatibles.
We need to check them carefully.

## Bootstrap issue?

Yes, there are.

Loading .rb files at boot timing of an interpreter can cause problem.
For example, before initializing String class, the class of String literal is 0 (because String class is not generated).

I introduces several workarounds but we need to modify more.

# Conclusion

How about to introduce this mechanism and try it on Ruby 2.7?
We can revert these changes if we found any troubles, if we don't expose this mechanism and only internal changes.

**Related issues:**

| | |
|---|---|
| Related to Ruby master - Misc #17502: C vs Ruby | **Open** |

**Associated revisions**

**Revision 46acd007 - 11/08/2019 12:09 AM - ko1 (Koichi Sasada)**

support builtin features with Ruby and C.

Support loading builtin features written in Ruby, which implement
with C builtin functions.
[Feature #16254]

Several features:

(1) Load .rb file at boottime with native binary.

Now, prelude.rb is loaded at boottime. However, this file is contained
into the interpreter as a text format and we need to compile it.
This patch contains a feature to load from binary format.

(2) __builtin_func() in Ruby call func() written in C.

In Ruby file, we can write __builtin_func() like method call.
However this is not a method call, but special syntax to call
a function func() written in C. C functions should be defined
in a file (same compile unit) which load this .rb file.

Functions (func in above example) should be defined with
(a) 1st parameter: rb_execution_context_t *ec
(b) rest parameters (0 to 15).
(c) VALUE return type.
This is very similar requirements for functions used by
rb_define_method(), however rb_execution_context_t *ec
is new requirement.

(3) automatic C code generation from .rb files.

tool/mk_builtin_loader.rb creates a C code to load .rb files
needed by miniruby and ruby command. This script is run by
BASERUBY, so *.rb should be written in BASERUBY compatbile
syntax. This script load a .rb file and find all of _builtin
prefix method calls, and generate a part of C code to export
functions.

tool/mk_builtin_binary.rb creates a C code which contains
binary compiled Ruby files needed by ruby command.

**Revision c8703a17 - 06/19/2020 09:46 AM - nobu (Nobuyoshi Nakada)**

[Feature #16254] Allow __builtin.func style

**Revision d863f4bc - 06/19/2020 09:46 AM - nobu (Nobuyoshi Nakada)**

[Feature #16254] Use __builtin.func style

**Revision 49f0fd21 - 06/19/2020 09:46 AM - nobu (Nobuyoshi Nakada)**

[Feature #16254] Allow Primitive.func style

**Revision 63aadc23 - 06/19/2020 09:46 AM - nobu (Nobuyoshi Nakada)**

[Feature #16254] Use Primitive.func style

## History

**#1 - 10/15/2019 08:01 PM - Eregon (Benoit Daloze)**

This sounds great.

We have very a similar mechanism in TruffleRuby, inherited from Rubinius, which is called "primitives" (ala Smalltalk).
Compared to Rubinius we changed the syntax and always use the invoke_primitive form (like __intrinsic__ above), not the "try intrinsic, if it fails fallback to the Ruby code below" (Smalltalk-style).

It looks like this:

```
class WeakRef < Delegator
  def initialize(obj)
    TrufflePrimitive.weakref_set_object(self, obj)
  end
end
```

(from https://github.com/oracle/truffleruby/blob/44e61173f0661c41dbf9a4c7a229091cf6ab83e3/lib/truffle/weakref.rb
see more examples with https://github.com/oracle/truffleruby/search?q=TrufflePrimitive&unscoped_q=TrufflePrimitive )

We recently changed from Truffle.invoke_primitive(:weakref_set_object, self, obj) to TrufflePrimitive.weakref_set_object(self, obj) because:

- It's more concise and arguably easier to read.
- As it's like a normal method call, we can generate stub definitions in IDEs (e.g., in IntelliJ), which allows to jump from Ruby code to the corresponding Java code of TruffleRuby.

Implementation-wise, the code above will generate an AST of the form

```
MethodNode(name=initialize, args=obj, body=[
  ReadArgumentNode,
  WeakRefSetObjectNode(children=[ReadSelfNode, ReadLocalNode(name=obj)])
])
```

I.e., the intrinsic node is directly in the AST of the method in TruffleRuby.
Only a fixed number of positional arguments is allowed. That way, there is no code for argument handling, arguments values are just passed directly to the WeakRefSetObjectNode as arguments are simply child nodes.

The main reason I'm detailing this is I think this an opportunity to standardize the syntax for "primitives"/"intrinsics".
No matter the implementation language, it seems the concept of "primitives"/"intrinsics" is universal.

A common syntax for intrinsics/primitives would allow to share Ruby code for core classes using these intrinsics/primitives.
It might look like it's little Ruby code, but I'm confident it will grow.
For instance, I wouldn't be surprised to see some of the argument processing/validation moved to Ruby, as it might just be easier.
At the very least, method definitions (the argument names and their default values) could be shared and avoid duplication.

What do you think?

**#2 - 10/16/2019 02:04 AM - Dan0042 (Daniel DeLorme)**

There's something I'm not sure I understood so I'd like to clarify if this proposal can be described as

A) write the boilerplate rb_define_class and rb_define_method using a ruby-like macro language;

B) write core classes and methods with the full power of ruby plus a few "invoke C function" macros, which are then compiled to VM instructions and serialized to become part of the binary.

It sounds to me like the proposal is (B) which is really an amazing idea and implementation, but the examples provided for Comparable and TracePoint could be trivially written as (A) so I'm not sure what is the advantage of the heavyweight (B) approach. It would really help to have an example that does more than just wrap the intrinsic function calls.

**#3 - 10/18/2019 06:08 AM - naruse (Yui NARUSE)**

Dan0042 (Daniel DeLorme) wrote:

> It would really help to have an example that does more than just wrap the intrinsic function calls.

Below is an example which solves Problem: Writen in C: (3) Performance issue: keyword parameters.
https://gist.github.com/ko1/969e5690cda6180ed989eb79619ca612#file-trace_point-rb-L195-L197

#### #4 - 11/07/2019 09:32 PM - ko1 (Koichi Sasada)

Design change:

(1) Table auto generation

> Should we maintain intrinsic function table?

I wrote "yes", but I found it is too difficult by human being. So I decide to generate this table by parsing .rb files.

As I wrote:

> Restrict syntax which can be used by BASE_RUBY for built-in ".rb" files.
> It is easy to list up intrinsic functions using Ripper or AST or ISeq#to_a.

There is this kind of restriction. You can not use pattern matches in .rb files :p

(2) __intrinsic__.func(...) to __builtin_func(...)

Reasons:
(a) similar to gcc's intrinsic format.
(b) easy to introduce special inline pragmra with __builtin_, like __builtin_attribute(:pure) and so on to teach the special information to Ruby interpreter. In this case, __builtin_attribute(:pure) can specify this method is "pure" (no-side effect) and so on.
(c) easy to parse (find out this format) by external tools. Without AST module, it is a bit difficult to parse __intrinsic__.foo() with compiled VM asm. However, AST module was introduced from Ruby 2.6 and the BASERUBY can be more older versions (the oldest version of BASERUBY on rubyci is ruby 2.2). This restriction can be relaxed by making analyzing microruby from source code (microruby is small subset of ruby interpreter to generate miniruby).

Completed code is https://github.com/ruby/ruby/pull/2655
I'll merge it soon.

#### #5 - 11/07/2019 09:35 PM - ko1 (Koichi Sasada)

Eregon (Benoit Daloze) wrote:

> A common syntax for intrinsics/primitives would allow to share Ruby code for core classes using these intrinsics/primitives.
> It might look like it's little Ruby code, but I'm confident it will grow.
> For instance, I wouldn't be surprised to see some of the argument processing/validation moved to Ruby, as it might just be easier.
> At the very least, method definitions (the argument names and their default values) could be shared and avoid duplication.
>
> What do you think?

I understand your concern. But I'm not sure we can share these code because they are "implementation" and depend on backend interpreter.

Anyway, it is very first stage and if this approach becomes mature, we can discuss more again.

#### #6 - 11/08/2019 12:09 AM - ko1 (Koichi Sasada)

*- Status changed from Open to Closed*

Applied in changeset git|46acd0075d80c2f886498f089fde1e9d795d50c4.

---

support builtin features with Ruby and C.

Support loading builtin features written in Ruby, which implement
with C builtin functions.
[Feature #16254]

Several features:

(1) Load .rb file at boottime with native binary.

Now, prelude.rb is loaded at boottime. However, this file is contained
into the interpreter as a text format and we need to compile it.
This patch contains a feature to load from binary format.

(2) __builtin_func() in Ruby call func() written in C.

In Ruby file, we can write __builtin_func() like method call.
However this is not a method call, but special syntax to call
a function func() written in C. C functions should be defined
in a file (same compile unit) which load this .rb file.

Functions (func in above example) should be defined with
(a) 1st parameter: rb_execution_context_t *ec
(b) rest parameters (0 to 15).
(c) VALUE return type.
This is very similar requirements for functions used by
rb_define_method(), however rb_execution_context_t *ec
is new requirement.

(3) automatic C code generation from .rb files.

tool/mk_builtin_loader.rb creates a C code to load .rb files
needed by miniruby and ruby command. This script is run by
BASERUBY, so *.rb should be written in BASERUBY compatbile
syntax. This script load a .rb file and find all of _builtin
prefix method calls, and generate a part of C code to export
functions.

tool/mk_builtin_binary.rb creates a C code which contains
binary compiled Ruby files needed by ruby command.

#### #7 - 04/08/2020 09:23 PM - Eregon (Benoit Daloze)

ko1 (Koichi Sasada) wrote in #note-5:

> Eregon (Benoit Daloze) wrote:
>
> > A common syntax for intrinsics/primitives would allow to share Ruby code for core classes using these intrinsics/primitives.
> > It might look like it's little Ruby code, but I'm confident it will grow.
> > For instance, I wouldn't be surprised to see some of the argument processing/validation moved to Ruby, as it might just be easier.
> > At the very least, method definitions (the argument names and their default values) could be shared and avoid duplication.
> >
> > What do you think?
>
> I understand your concern. But I'm not sure we can share these code because they are "implementation" and depend on backend interpreter.
>
> Anyway, it is very first stage and if this approach becomes mature, we can discuss more again.

It would be great to unify the style primitives are written between Ruby implementations.
__builtin_foo(1, 2) is MRI-specific and not really pretty in Ruby code.

ko1 (Koichi Sasada) Could you reconsider this?

I think Primitive.foo(1, 2) is significantly superior to __builtin_foo(1, 2) because:

- It's possible to have "jump to definition" working in the first form, in fact that already works in RubyMine with TruffleRuby and so one can jump from Ruby code to the Java code implementing that primitive. To make __builtin_foo work with jump to definition it would have to be an Object or Kernel method, which is not clean.
- Primitive.foo(1, 2) clearly namespaces those special calls, and from the name it's easy to guess there is some intrinsified behavior.
- primitive is the name chosen by Smalltalk, and Ruby has a strong Smalltalk inheritance. This is syntax that is in Ruby code. __builtin_foo doesn't look like Ruby code at all (it's a C-style private function).
- We shouldn't make something that can be shared between implementations use a C-specific syntax, that would hurt compatibility between implementations for no much reason. It's used in Ruby (core) code, it should use a Ruby-like syntax.

Of course the function implementing the primitive on the C side can have a different convention (e.g., rb_primitive_foo), that doesn't matter.

Regarding parsing one could just use a regexp, I doubt any of these core files would use syntax like Primitive.something. Ripper would also be enough if we want to be more precise.

cc headius (Charles Nutter)

#### #8 - 04/23/2020 05:53 AM - ko1 (Koichi Sasada)

*- Status changed from Closed to Assigned*

Eregon:

There are two poionts: (1) sharing the code between Ruby implementations and (2) prettier syntax.

# (1) sharing the code between Ruby implementations

I assume it is using same .rb code between Ruby implementations.
My conclusion is it is bad idea to share the same code between interpreters.

There are some points:

- (a) If we need to unify the source code, the body should be empty and call a function (or Java method, and so on) which is decided by convention.

For example, a method empty? in String class should be defined as:

```
class String
  def empty?
  end
end
```

and call rb_builtin_str_empty_p() implicitly. The name convention should be customize.

- (b) Each interpreter should use their own body.

However, sometimes we want to write a code in Ruby because it is easy to write a code than to write native code (this is one motivation of this feature).

The body is written in Ruby, but an interpreter want to use a ruby code in a body, but another interpreter want to write the code in native code, because of several problem such as performance interest.
I think the advantage of sharing the core ruby code is less than the disadvantage because of this point.

- (c) (MRI-specific)

Also I plan to write a C code part in .rb code, which is easy to write for me.
Example: https://github.com/ko1/ruby/blob/ractor/ractor.rb#L69

And also we want to introduce annotation methods "(2) Annotation issues (for further optimization)" and I think the annotations maybe be the same and different.

I know the point (c) can be solved by ignoring them.

# (2) prettier syntax

Honestly speaking, I have no strong opinion about the syntax.
For example, I have a frustration that I can't select a function name by double clicking a __builtin_foo() (__builtin_foo is selected.)
A (small) concern about the name Primitive is, it is too common class name and conflicts with other code (if there is a reader). This is why I use __ prefix for it.

A practical reason of __builtin_foo syntax is, it is easy to implement.
I used VM iseq disassemble result to implement a detector (a utility to list a builtin functions in C code). It is difficult to specify receiver code from VM disassemble because it is stack machine, in general.
We can make a detector by using RubyVM::AST or ripper, but ripper has several method call nodes and so on.

Maybe the most correct approach to make a detector is to implement it by parse.y code (separate binary from miniruby).
With this technique, it is easy to parse (because we can modify the parser for this purpose) and we can use any newer syntax in core .rb files (such as pattern matching in Ruby 2.7).
Now, a detector is implemented by a BASERUBY (system installed ruby) and it should be older than master.

#### #9 - 04/23/2020 09:55 AM - Eregon (Benoit Daloze)

ko1 (Koichi Sasada) wrote in #note-8:

> My conclusion is it is bad idea to share the same code between interpreters.

I believe any alternative Ruby implementation would tell you the opposite: the more shared code the less duplicated efforts.
All Ruby implementations use a lot of code (not only Ruby files) from MRI, let's make it easier for this case.

If this doesn't happen, I would still expect other Ruby implementations to copy those files and then transform them.
It just seems a big missed opportunity to me to not standardize on such a simple thing.
MRI is now using more Ruby code and primitives, like Rubinius and TruffleRuby did before, let's make it into something that can be shared nicely.

I'd like to have a call so we can discuss that more directly.

> There are some points:
>
> - (a) If we need to unify the source code, the body should be empty and call a function (or Java method, and so on) which is decided by convention.

That leaves no room for any code in the Ruby definition, so it's much more restrictive.

> - (b) Each interpreter should use their own body.
>
> However, sometimes we want to write a code in Ruby because it is easy to write a code than to write native code (this is one motivation of this feature).
>
> The body is written in Ruby, but an interpreter want to use a ruby code in a body, but another interpreter want to write the code in native code, because of several problem such as performance interest.
> I think the advantage of sharing the core ruby code is less than the disadvantage because of this point.

In TruffleRuby a large number of files are copied from MRI: Ruby stdlib, C-ext stdlibs and many other things.
In that process it's still fine to choose whether we want to use the Ruby definition of a method or something else.
It's Ruby code so we can also have the Ruby definition and override it later if needed.
Since those files are special (core files) we could also have the implicit semantics that a method definition is skipped if the method exists on the same Module.
There are many ways, MRI doesn't need to worry about that.

> - (c) (MRI-specific)
>
> Also I plan to write a C code part in .rb code, which is easy to write for me.
> Example: https://github.com/ko1/ruby/blob/ractor/ractor.rb#L69

Yes, that cannot be shared, and that's fine not everything needs to be shared.

Such an example makes me think it would be nicer and easier to maintain/lint/syntax highlight/etc if the logic was in a __builtin, and the Ruby method would call that builtin.
From experience in TruffleRuby (e.g., we had snippets of Ruby in Java code but no more),
mixing 2 languages at a finer level than functions/methods results in more confusion and more drawbacks than advantages in the longer term.

> And also we want to introduce annotation methods "(2) Annotation issues (for further optimization)" and I think the annotations maybe be the same and different.
> I know the point (c) can be solved by ignoring them.

Yes, that's harmless they can be ignored or maybe other implementations find such annotations useful too, it's all optional.

> A (small) concern about the name Primitive is, it is too common class name and conflicts with other code (if there is a reader). This is why I use __ prefix for it.

Since builtins are only allowed in core files, this doesn't matter.
A Primitive module doesn't actually need to exist, and the user can still define a Primitive constant without conflicts.

> A practical reason of __builtin_foo syntax is, it is easy to implement.

That approach sounds more complicated than needed to me.
I think a simpler regexp approach would go a long way and something like /\bPrimitive\.(\w+)\(.+\)/ is very easy and will realistically never be ambiguous in core Ruby files.

How about I make a PR replacing __builtin_foo with Primitive.foo?

#### #10 - 05/09/2020 05:25 PM - ko1 (Koichi Sasada)

Eregon (Benoit Daloze) wrote in #note-9:

> > My conclusion is it is bad idea to share the same code between interpreters.
>
> I believe any alternative Ruby implementation would tell you the opposite: the more shared code the less duplicated efforts.
> All Ruby implementations use a lot of code (not only Ruby files) from MRI, let's make it easier for this case.
>
> If this doesn't happen, I would still expect other Ruby implementations to copy those files and then transform them.
> It just seems a big missed opportunity to me to not standardize on such a simple thing.
> MRI is now using more Ruby code and primitives, like Rubinius and TruffleRuby did before, let's make it into something that can be shared nicely.

It is one big advantage. I have no objection if it does not hurt MRI development.

There are several concerns:

- (1) how to manage the code?
  - MRI repository?
- (2) how to introduce MRI specific code?
  - __builtin_cexpr!, __builtin_attribute..., ...
  - I will also introduce *overload* special form for the performance, is it acceptable?
- (3) how to choose the Ruby defs and C defs?
  - Because of several reasons, we can't choose Ruby version (C version)
  - maybe the goal is all of defs in Ruby, but we need more effort to achieve it.

I think the current situation is not matured to share the code.
Maybe we don't expose this specification for the C extensions, it means we can change the spec later.

Note about the *overload* syntax. Some methods use optional arguments and the straight forward implementation with builtin functions is like:

```
def foo(o1=nil, o2=nil)
  __builtin_foo(o1, o2)
end
```

Using optional arguments with nil is common usage and on many cases the optional parameters are not given.
However, the setup optional parameters will be an additional overhead and it is slower than C-impl (https://rubykaigi.org/2019/presentations/ko1.html shows it).
If we allow the overload method definition, the overhead will be reduced like:

```
overload do
  def foo
    __builtin_foo0()
  end
  # general definition
  def foo(o1 = nil, o2 = nil)
    __builtin_foo(o1, o2)
  end
end
```

I doubt we can introduce such syntax to the Ruby, but builtin library can accept it.
Another option is write such branch in the method body:

```
def foo o1=nil, o2=nil
  __builtin_argc(0,   __builtin_foo0(),
                 nil, __builtin_foo(o1, o2)
end
```

But this syntax is not nice (and complicate syntax is hard to analyze).

Ah, this discussion is denoted in the ticket body:

> [*1] Introducing special "overloading" specifier can solve it because we don't need to assign optional parameters. First method lookup can be slowed down, but we can cache the method lookup results (with arity).
>
> - (c) (MRI-specific)
>
> Also I plan to write a C code part in .rb code, which is easy to write for me.
> Example: https://github.com/ko1/ruby/blob/ractor/ractor.rb#L69

> Yes, that cannot be shared, and that's fine not everything needs to be shared.
>
> Such an example makes me think it would be nicer and easier to maintain/lint/syntax highlight/etc if the logic was in a __builtin, and the Ruby method would call that builtin.
> From experience in TruffleRuby (e.g., we had snippets of Ruby in Java code but no more),
> mixing 2 languages at a finer level than functions/methods results in more confusion and more drawbacks than advantages in the longer term.

If there are long lines, it should be.

> A (small) concern about the name Primitive is, it is too common class name and conflicts with other code (if there is a reader). This is why I use __ prefix for it.

> Since builtins are only allowed in core files, this doesn't matter.
> A Primitive module doesn't actually need to exist, and the user can still define a Primitive constant without conflicts.

Yes. But I meant reader's cost can be increase. __xxx is strange name and it is highlighted.

> A practical reason of __builtin_foo syntax is, it is easy to implement.

> That approach sounds more complicated than needed to me.
> I think a simpler regexp approach would go a long way and something like /\bPrimitive\.(\w+)\(.+\)/ is very easy and will realistically never be ambiguous in core Ruby files.

We need to know the parameter's number so the regexp approach doesn't work.

#### #11 - 05/13/2020 09:17 PM - Eregon (Benoit Daloze)

ko1 (Koichi Sasada) wrote in #note-10:

- (1) how to manage the code?
  - MRI repository?

Yes, that's fine, at least for now.
Maybe later we can share such code in a shared repository, but one step at a time, and probably not needed (other Rubies import from MRI anyway).

- (2) how to introduce MRI specific code?
  - __builtin_cexpr!, __builtin_attribute..., ...

I think we can just skip those for other implementations (e.g., ignore method definitions or files including __builtin_cexpr!).

- I will also introduce *overload* special form for the performance, is it acceptable?

Yes, although IMHO it would be much better and simpler to optimize optional arguments.
On slide Evaluation With optional arguments you show

```
static VALUE
dummy_func_opt(int i, VALUE *v, VALUE self) {
  VALUE p1, p2;
  rb_scan_args(i, v, "11", &p1, &p2);
  return NIL_P(p2) ? p1 : p2;
}
```

That seems sub-optiomal, I would expect rb_scan_args() is slow-ish.
How about defining it as:

```
static VALUE
dummy_func_opt(VALUE self, VALUE p1, VALUE p2) {
  return NIL_P(p2) ? p1 : p2;
}
```

I expect that's as fast or faster than C, and so overload is unnecessary (or insignificant gain for large complexity).
If you need argc to know if an argument was passed or not, that can probably be passed some way (another way is to have Qundef default argument values).

Duplicating method bodies doesn't sound optimal at least for some cases,
and in many cases there are checks on the arguments (not just "passed or not", but e.g. "nil or not", "is Integer", etc) so those should be reasonably fast.

- (3) how to choose the Ruby defs and C defs?
  - Because of several reasons, we can't choose Ruby version (C version)
  - maybe the goal is all of defs in Ruby, but we need more effort to achieve it.

I think MRI can choose this as time goes by.
e.g. yield_self/then seems a good example.
Probably for MRI JIT, more in Ruby in almost always better.

> I think the current situation is not matured to share the code.
> Maybe we don't expose this specification for the C extensions, it means we can change the spec later.

Yes, only available for special "core library Ruby files".
In TruffleRuby we use a magic comment # truffleruby_primitives: true for this to have a bit more flexibility (e.g., to use in tests):
https://github.com/oracle/truffleruby/blob/3a7d56c0f170a46eb0cc36cefb49a62b14fb085c/spec/truffle/array/array_storage_equal_spec.rb#L1

But just based on file path (*.rb from MRI source dir) seems fine too.

> However, the setup optional parameters will be an additional overhead and it is slower than C-impl ( https://rubykaigi.org/2019/presentations/ko1.html shows it).

I think we should measure again, and try to optimize it if not fast enough.

> If we allow the overload method definition, the overhead will be reduced like:

Seems complicated and the code will still need to choose the right version by checking argc.

> Another option is write such branch in the method body:

```
def foo o1=nil, o2=nil
  __builtin_argc(0,   __builtin_foo0(),
                 nil, __builtin_foo(o1, o2)
end
```

Maybe we can have Primitive.argc?
Then:

```
def foo o1=nil, o2=nil
  if Primitive.argc == 0
    __builtin_foo0()
  else
    __builtin_foo(o1, o2)
  end
end
```

(but again, I think optimizing optional arguments is better)

> Yes. But I meant reader's cost can be increase. __xxx is strange name and it is highlighted.

Ruby has no built-in Primitive module so I think Ruby-core people will get learn very quickly what it does.

> We need to know the parameter's number so the regexp approach doesn't work.

Could you point me to where we need to know the number of parameters?
Probably BASERUBY Ripper is a good approach then (or as you say, a variant of parse.y, or maybe just just lexer is enough).

#### #12 - 05/14/2020 02:42 PM - Eregon (Benoit Daloze)

Some examples of Primitive.name used in TruffleRuby:

https://github.com/oracle/truffleruby/blob/af97b03a55b757688a62455cd56672c53cd56d0d/lib/truffle/weakref.rb#L36
https://github.com/oracle/truffleruby/blob/af97b03a55b757688a62455cd56672c53cd56d0d/src/main/ruby/truffleruby/core/array.rb#L111
https://github.com/oracle/truffleruby/blob/96541ee9ea0f399248d568a37600ef0787e87f59/src/main/ruby/truffleruby/core/string.rb
https://github.com/oracle/truffleruby/blob/3a7d56c0f170a46eb0cc36cefb49a62b14fb085c/src/main/ruby/truffleruby/core/truffle/feature_loader.rb#L224

General search (not fully precise):
https://github.com/oracle/truffleruby/search?l=Ruby&q=Primitive.

#### #13 - 05/14/2020 02:57 PM - Eregon (Benoit Daloze)

As an example, https://github.com/ruby/ruby/blob/d7d0d01401a8082e514eb2cb3cec5410e7acba7d/trace_point.rb could be used as-is in TruffleRuby if using the Primitive.name syntax.
It might not be a lot, but it's a start and it's already nice it extracts keyword arguments, etc.

#### #14 - 05/14/2020 03:06 PM - Eregon (Benoit Daloze)

An even better example, dir.rb could be used in TruffleRuby & other implementations easily:
https://github.com/ruby/ruby/blob/d7d0d01401a8082e514eb2cb3cec5410e7acba7d/dir.rb#L14-L25

There is already quite a bit of logic there.
All it'd take is using the same builtin syntax.

#### #15 - 06/19/2020 09:47 AM - nobu (Nobuyoshi Nakada)

*- Status changed from Assigned to Closed*

Applied in changeset git|c8703a17ceedae4a72f83c962a4d098731a85dda.

---

[Feature #16254] Allow __builtin.func style

#### #16 - 06/21/2020 08:47 AM - k0kubun (Takashi Kokubun)

I have no preference on whether it should be called "primitive" or "builtin", but can we at least make the naming consistent? The Primitive change introduced naming inconsistency inside CRuby and it's causing cognitive overhead when reading any code related to this ticket.

I hope one of the following sets of naming will be used.

- Primitive.xxx
- primitive.c
- tool/mk_primitive_loader.rb
- vm_invoke_primitive, invoke_pf, rb_primitive_function

or

- Builtin.xxx
- builtin.c
- tool/mk_builtin_loader.rb
- vm_invoke_builtin, invoke_bf, rb_builtin_function

#### #17 - 06/21/2020 09:54 AM - Eregon (Benoit Daloze)

Great to see Primitive.name is used now :)
Somehow Redmine didn't send me any notification for the issue being closed.

k0kubun (Takashi Kokubun) agreed for the first one.

#### #18 - 01/03/2021 03:09 AM - k0kubun (Takashi Kokubun)

*- Related to Misc #17502: C vs Ruby added*