Fythe Specification

This specification is not yet complete.
It is available in Fythe's mercurial repository, at `https://codu.org/projects/fythe/specchg/`.

Revision 8bb2419d0e43 tip

07 September 2011

Contents

Chapter 1

Overview

Fythe is a design for virtual machines primarily targeting dynamic languages. It is not precisely a language or an interpreter, but a framework for creating interpreters for languages.

The execution of a program by Fythe proceeds as follows:

- The program source is parsed (one statement or declaration at a time) by the Fythe parser. The productions used are defined at runtime, and may be changed at any time. The result of parsing is a tree, the form of which is also defined at runtime.
- The tree produced by parsing is run through a number of transformations. These transformations are defined at runtime, and may be changed at any time. The result of these transformations is a tree which must fit a prescribed grammar, the Fythe Intermediate Representation (Fythe IR).
- The Fythe IR tree is reduced into executable code, generally machine code, by an implementation-specific means, and run. The behavior of the generated code is dictated by the behavioral specification of Fythe IR herein.

The first chapter of this specification describes the Fythe IR in isolation. Chapters 3 and 4 will describe the parser and transform engine, but are yet to be written.

Chapter 2

Fythe Objects and IR

Every IR expression is reduced at runtime to a value. A value in Fythe is an object reference associated optionally with a primitive value. A primitive value is a string, function, integer, floating-point number, or implementation-specific primitive. All primitives are immutable. An object is an array of values; that is, a set of values indexed by keys which range from 0 to the size of the object, exclusively. There are three special objects: Null, which is an object of size 0 used as a nonce; Global, which is to be used as a global data store; and Version, which stores implementation-specific platform, version and feature information. There are also three registers, the values of which are specific to a given context. Two registers are specific to a function call: This, which is initially the function itself; and Arg, which is the argument passed to the function. One register, Caught, is specific to the Catch context, and will be described in the Exceptions section.

The IR is a tree, formed by Fythe values. Although there is no canonical syntax for Fythe values, this specification uses LISP-like list expressions, conforming to the following grammar (using regular expressions as terminals):

```
WhiteN      = /[ \t\r\n]*/

Expression  = /\(/ WhiteN List /\)/ WhiteN
            | /\(/ WhiteN /\)/ WhiteN
            | /[^\(\)\{\}0-9 \t\r\n][^\(\)\{\} \t\r\n]*/ WhiteN
            | /[0-9]+/ WhiteN

List        = Expression
            | Expression List
```

An expression of the first two forms represents an object associated with no primitive. An expression of the third form represents a string associated with Null. An expression of the fourth form represents an integer associated with Null. This syntax has no means of expressing objects other than Null associated with primitive values, as these have no behavior in the Fythe IR.

A Fythe IR expression is an object in which the first element is a string literal, the value of which is the type of expression, and the remaining elements are arguments. For instance, `(Null)` is a Fythe IR expression which

(at runtime) reduces to the value Null (associated with no primitive), `(New 2)` is a Fythe IR expression which reduces to a newly-created object of size 2, and `(Concat (New 2) (New 2))` is a Fythe IR expression which (rather pointlessly) reduces to the concatenation of two newly-produced objects of size 2, to produce a new object of size 4.

The following table is a list of every Fythe IR expression, its parameter count, the types its arguments must have (when reduced at runtime), and any additional notes. The types are expressed as a string of letters, with each letter corresponding to one argument or return type. Where the input types are all generic (just objects, no primitives necessary) the list is elided, as is the return type when it may be any object or primitive type. The association between letters and types is in Figure 2.1. Not all expressions are yet documented[1].

Name	P	Types	Behavior
		Meta	
Function	1	O→c	Result is a function (associated with Null) with the first argument as its body
		Objects	
This	0		Result is the value of the This register, which is initially the value called.
ThisSet	1		The This register is set to the value of argument 0. Result is the value.
Arg	0		Result is the value of the Argument register, which is initially the second argument of the Call expression used to call this function.
Null	0		Evaluates to Null, with no primitive associated.
Global	0		Evaluates to Global, with no primitive associated.
ExpandGlobal	1	i→i	Replaces Global with a new object, the size of which is the original size of Global plus the supplied integer argument. Evaluates to the old size of Global.
Version	0		Evaluates to Version, with no primitive associated.
New	1	i	A new object, the size of which is specified by the first argument, is created. Each field of the new object has the value of Null with no primitive associated. The result is the new object, with no primitive associated.
Length	1	→i	Given an object, evaluates to its size, with Null associated.
Member	2	oi	Given an object and an index, evaluates to the value at that index of the object.
MemberSet	3	oio	Given a "to" object, an index, and a "from" value, sets the value of the "to" object at that index to the "from" value. Result is the "from" value.
Equal	2	→i	Given two objects, if they are referentially identical (ignoring the primitive), results in the integer 1 associated with Null. Otherwise results in the integer 0 associated with Null.
Object	*		Creates a new object, the size of which is the number of arguments. The values of the fields are the values of the arguments, in the order they appear. Result is the new object, with no primitive associated.

[1] Although not yet integrated into this spec, "bignums" (multi-precision integers) are being integrated into Fythe. In time, all integer functions below will never overflow.

Name	P	Types	Behavior
Concat	2		Creates a new object, the size of which is the sum of the size of the two argument objects. Given objects of size N and M, the values of the new object's fields $[0..N-1]$ are set to the values of the first argument object's fields, and the values of the new object's fields $[N..N+M-1]$ are set to values of the second argument object's fields. Result is the new object, with no primitive associated.
ConcatT	2		See chapter 4.
Slice	3	oii	Given an object and two indexes X and Y, creates a new object of size $Y-X$. The values of the new object's fields $[0..Y-X]$ are set to the values of the argument object's fields $[X..Y-1]$. Result is the new object, with no primitive associated.
Temporaries[2]			
Temp	1	i	See the Temporaries section.
TempSet	2	io	
Type detection[3]			
ValidString	1	→i	Will result in the integer 1 or 0, in either case associated with Null. If the value of the argument was created as a string, the result will be 1, but the result 0 does not indicate that the value was not created as a string, only that the VM can distinguish the value from a string. A VM should not return 1 if using the value as a string will result in the VM crashing.
ValidFunction	1	→i	Will result in the integer 1 or 0, in either case associated with Null. If the value of the argument was created as a function, the result will be 1, but the result 0 does not indicate that the value was not created as a function, only that the VM can distinguish the value from a function. A VM should not return 1 if using the value as a function will result in the VM crashing.
ValidInteger	1	→i	Will result in the integer 1 or 0, in either case associated with Null. If the value of the argument was created as an integer, the result will be 1, but the result 0 does not indicate that the value was not created as an integer, only that the VM can distinguish the value from an integer. A VM should not return 1 if using the value as an integer will result in the VM crashing.
ValidFloat	1	→i	Will result in the integer 1 or 0, in either case associated with Null. If the value of the argument was created as a floating-point number, the result will be 1, but the result 0 does not indicate that the value was not created as a floating-point number, only that the VM can distinguish the value from a floating-point number. A VM should not return 1 if using the value as a floating-point number will result in the VM crashing.

[2]Fythe provides infinite temporaries, but the mapping of temporaries to registers or similar is implementation-defined.

[3]These functions are purely heuristic. Although they must evaluate to 1 for any correctly-typed value, they may also evaluate to 1 for other values, although the result of using these values as the indicated type is undefined. That is, any implementation is allowed to produce incorrect results in the form of false negatives if, for instance, it cannot distinguish between types; proper type management is the responsibility of the programmer or language implementer, not Fythe.

Name	P	Types	Behavior
ValidTable	2	oi→i	Will result in the integer 1 or 0, in either case associated with Null. Given an object and an index, if the value of the indexed field of the object was created as a table, the result will be 1, but the result 0 does not indicate that the value was not created as a table, only that the VM can distinguish the value from a table. A VM should not return 1 if using the value as a table will result in the VM crashing.
ValidThread	1	→i	Will result in the integer 1 or 0, in either case associated with Null. If the value of the argument was created as a thread identifier, the result will be 1, but the result 0 does not indicate that the value was not created as a thread identifier, only that the VM can distinguish the value from a thread identifier. A VM should not return 1 if using the value as a thread identifier will result in the VM crashing.
ValidLock	1	→i	Will result in the integer 1 or 0, in either case associated with Null. If the value of the argument was created as a lock, the result will be 1, but the result 0 does not indicate that the value was not created as a lock, only that the VM can distinguish the value from a lock. A VM should not return 1 if using the value as a lock will result in the VM crashing.

Function and call-stack

Call	2	co	Call the given function. The result is the result of the function's body, when evaluated with the value of the This register being the function value (that is, the first argument), and the value of the Arg register being the second argument.
Throw	1		See the Exceptions section.
Catch	2	OO	
Caught	0		

Behavioral

If	3	iOO	The first argument is evaluated. If its value is 0, then the third argument is evaluated and its result is the result of this If. Otherwise, the second argument is evaluated and its result is the result of this If.
While	2	IO	In a loop: • The first argument is evaluated. • If its value is 0, the loop is terminated and the result of this While is the result of the most recent evaluation of the second argument; if the second argument has not yet been evaluated, the result of this While is Null with no primitive associated. • The second argument is evaluated, and its result remembered.
Seq	*		Each argument is evaluated in sequence. The result is the value of the last argument. (Seq must have at least one argument)

Name	P	Types	Behavior
Primitives			
Associate	2	po→p	Results in the primitive value of the first argument associated with the object of the second argument.
Dissociate	1	p→o	Results in the object of the first argument with no primitive associated. Note that the equivalent for getting a dissociated primitive value is to associate it with Null.
Strings			
SConcat	2	ss→s	Given two strings, results in a new string which is the concatenation of the argument strings, associated with Null.
SLength	1	s→i	Results in the length of the argument string as an integer, associated with Null.
SSlice	3	sii→s	Similar to Slice, but slicing over bytes in a string instead of fields in an object.
SEqual	2	ss→i	Results in the integer value 1 if the two strings are equal, 0 otherwise. In either case the value is associated with Null.
SToInteger	1	s→i	Result is the integer value of the string argument, interpreted as a decimal ASCII number, associated with Null.
SToIntegerT	1	s→1	See chapter 4.
Integers			
IWidth	0	→i	Result is the width, in bits, of a traditional signed 2s-compliment which integers provided by the VM are capable of representing. For instance, if the VM actually uses 2s-compliment integers, the result is simply the width of integers it uses. If the VM uses something else (e.g. floating-point numbers), the result is a conservative estimate of their usable width as 2s-compliment integers. The result is associated with Null. The behavior of integers when they do not overflow the size of such an integer is guaranteed to be correct.
IMul	2	ii→i	Result is the integer values of the two arguments multiplied together. If this overflows the VM's representation of integers, the result is implementation-defined. The result is associated with Null.
IDiv	2	ii→i	Result is the integer value of the first arguments divided by the integer value of the second argument, truncated. The result is associated with Null.
IMod	2	ii→i	Result is remainder after dividing the integer value of the first arguments by the integer value of the second argument. If either argument is negative, the integer result is implementation-defined. The result is associated with Null.
IAdd	2	ii→i	Result is the sum of the integer values of the arguments. If this overflows the VM's representation of integers, the result is implementation-defined. The result is associated with Null.
ISub	2	ii→i	Result is the integer value of the second argument subtracted from the integer value of the first argument. If this overflows the VM's representation of integers, the result is implementation-defined. The result is associated with Null.

Name	P	Types	Behavior
ISl	2	ii→i	Result is the integer value of the first argument times (two to the power of the integer value of the second argument), decimal truncated. If this overflows the VM's representation of integers, the result is implementation-defined. The result is associated with Null.
ISr	2	ii→i	Result is the integer value of the first argument times (two to the power of the negated integer value of the second argument), decimal truncated. If this overflows the VM's representation of integers, the result is implementation-defined. The result is associated with Null.
INot	1	i→i	If the argument's integer value is 0, the result is the integer value 1. Otherwise the result is the integer value 0. In either case, the result is associated with Null.
IOr	2	ii→i	If the integer values of both arguments are 0, the result is the integer value 0. Otherwise the result is the integer value 1. In either case, the result is associated with Null.
IAnd	2	ii→i	If the integer values of both arguments are 1, the result is the integer value 1. Otherwise the result is the integer value 0. In either case, the result is associated with Null.
IByte	1	i→s	The result is a string, of length one, the value of the first byte of which is the integer value of the first argument. If the first argument has a value less than 0 or greater than 255, then the result is implementation-defined. The result is associated with Null.
IEqual	2	ii→i	If the integer values of both arguments are equal, the result is the integer value 1. Otherwise the result is the integer value 0. In either case, the result is associated with Null.
INe	2	ii→i	If the integer values of both arguments are not equal, the result is the integer value 1. Otherwise the result is the integer value 0. In either case, the result is associated with Null.
ILt	2	ii→i	If the integer value of the first argument is strictly less than the integer value of the second argument, the result is the integer value 1. Otherwise the result is the integer value 0. In either case, the result is associated with Null.
ILte	2	ii→i	If the integer value of the first argument is strictly less than the integer value of the second argument, the result is the integer value 1. Otherwise the result is the integer value 0. In either case, the result is associated with Null.
IGt	2	ii→i	If the integer value of the first argument is strictly less than the integer value of the second argument, the result is the integer value 1. Otherwise the result is the integer value 0. In either case, the result is associated with Null.
IGte	2	ii→i	If the integer value of the first argument is strictly less than the integer value of the second argument, the result is the integer value 1. Otherwise the result is the integer value 0. In either case, the result is associated with Null.
IToFloat	1	i→f	The integer value is converted to a float value. Due to conflicting ranges of integers and floats, this conversion may lose precision.
Floats			
FMul	2	ff→f	

Name	P	Types	Behavior
FDiv	2	ff→f	
FMod	2	ff→f	
FAdd	2	ff→f	
FSub	2	ff→f	
FEqual	2	ff→i	
FNe	2	ff→i	
FLt	2	ff→i	
FLte	2	ff→i	
FGt	2	ff→i	
FGte	2	ff→i	
FToInteger	1	f→i	

Tables

MInit	2	oi	The index specified by the second argument in the object specified by the first argument is initialized as a mapping of strings to values. Note that the content of both the object and primitive parts of a field being used as a mapping are implementation-defined.
MGet	3	ois	The result is the value associated with the string specified by the third argument in the mapping specified by the first two arguments (object and index). If the string is not contained in the mapping, the result is Null.
MSet	4	oiso	Updates or sets the value associated with the string specified by the third argument to the fourth argument, in the mapping specified by the first two arguments (object and index). Result is the fourth argument.
MList	2	oi	Results in a new object which is an array of every string contained in the mapping specified by the arguments (object and index), with each string associated with Null.
MContains	3	ois→i	If the mapping specified by the first two arguments (object and index) has a value associated with the string specified by the third argument, the result is the integer 1. Otherwise, the result is 0.

Threading

Spawn	2	co→t	
Join	1	t	(FIXME: what's the return?)
Yield	0		(always returns (Null))
LMutexNew	0	→l	
LMutexLock	1	l→l	
LMutexTryLock	1	l→i	
LMutexUnlock	1	l→l	
LMutexDestroy	1	l→o	(alwas returns (Null))
LSemNew	0	→l	
LSemPost	1	l→l	
LSemWait	1	l→l	
LSemDestroy	1	l→o	(alwas returns (Null))
LRWNew	0	→l	

Name	P	Types	Behavior
LRWRLock	1	l→l	
LRWRTryLock	1	l→i	
LRWRUnlock	1	l→l	
LRWRLock	1	l→l	
LRWRTryLock	1	l→i	
LRWRUnlock	1	l→l	
LRWDestroy	1	l→o	(always returns (Null))
Cas	4	oioo→i	(only compares and swaps the object, not the primitive)
PCas	4	oipp→i	(only compares and swaps the primitive, not the object)

Runtime

Include	1	s→s	The result is the content of the file named by the first argument. How the file is found and read is implementation-defined. A typical Fythe interpreter should have a pre-determined list of search paths. If the file does not exist, can't be read, or any implementation-defined error occurs, the result is Null.
Parse	5	siiss	See chapter 3.
Transform	1		See chapter 4.

Grammar[4]

GAdd	3	soo→s	
GRem	1	s→s	

Transform[5]

TAddPhase	2	ss→i	
TAddTree	3	soo→i	
TAddFunction	3	soc→i	
TRem	1	s→i	

[4]For the behavior of these expressions, see chapter 3.
[5]For the behavior of these expressions, see chapter 4.

	Type
o	Object (primitive value ignored)
s	String
c	Function (code)
i	Integer
f	Floating-point number
t	Thread identifier
l	Lock
p	Any primitive value

A capitalized letter indicates that the relevant argument may be evaluated a number of times other than one. That is, the argument will be evaluated conditionally, multiple times, or deferred.

Figure 2.1: Fythe type identifiers

Chapter 3

The Dynamic Parser Generator

(This section is under development)

A Fythe virtual machine has a fixed intermediate representation (Fythe IR), but the grammar of the langauge the VM accepts is defined at runtime, and may be changed at any time during execution. Each file is read, then parsed and executed one statement or declaration at a time.

Fythe's parser's grammar is defined in terms of a set of parsers, each of which may be a nonterminal, a terminal, or a negation. Nonterminals must be explicitly defined through the `GAdd` operation, the others are defined implicitly by their use.

Nonterminals may have any name that begins with a letter and contains no whitespace. By convention, they are named in capitalized camel-case, with namespaces (such as particular languages being defined within Fythe) separated by dots. For instance, the nonterminal representing statements in a language called *A Great Language* might be called `AGreatLanguage.Statement`.

As in any grammar, nonterminals are defined in terms of expressions, where each element of the expression is another parser. Every nonterminal ma have any number of these expressions. Nonterminals may be directly left recursive; that is, the first parser in an expression defining a nonterminal may be the nonterminal itself. They may not, however, be indirectly left recursive. Other than that restriction, all valid context-free grammars must be supported.

Terminals in Fythe's parser are in the form of regular expressions. As such, there is no lexical analysis. The name of a terminal must start and end with slashes (/s), and its name is also the regular expression which it accepts.

Negations are an extension to nonterminals and terminals, and create a form of lookahead during parsing. The name of a negation must start with an exclamation point (!), and the remainder of the name is the parser to be negated. If the negated parser accepts the input, then the negation rejects it. If the negated parser rejects the input, then the negation accepts it. In either case, no input is actually consumed by the negation.

Every parser has a target, which is Fythe value (typically Fythe IR code) generated when the parser accepts input. Terminals generate the string their regular expression accepted, and negations always generate Null. The value that nonterminals generate is defined at the same time as the expression they accept, and may be any non-circular Fythe value. When a nonterminal accepts input, its target value is duplicated, and string literals beginning

with a dollar sign ($) are replaced by the values generated by the subparsers of the nonterminal. For instance, if a nonterminal `A` is defined with expression `/a/ /b/` and target `(Object $0)`, and accepts the input `ab`, then it generates the value `(Object a)`.

For the purpose of documentation, nonterminals are described in this format:

```
NonTerminal = Space-separated list of parsers => Fythe value
```

Because a Fythe VM expects its input to be parsed, it must also provide a built-in grammar for the purpose of bootstrapping. The built-in grammar is:

```
White = /(([ \t\r]*(\/\*([^\*]*\*[^\/])*[^\*]*\*\/)?)*)/ => $0
WhiteN = /(([ \t\r\n]*(\/\/[^\r\n]*[\r\n])?(\/\*([^\*]*\*[^\/])*[^\*]*\*\/)?)*)/ => $0

Top = /#!.*\n/ => (Function (Null))

Top = WhiteN LanguageExpression => (Function $1)

LanguageExpression = Fythe => $0

Fythe = /fythe/ WhiteN /\{/ WhiteN Fythe.Statements /\}/ WhiteN => $4

Fythe.Statements = Fythe.Expression => $0
Fythe.Statements = Fythe.Statements Fythe.Expression => (Seq $0 $1)

Fythe.Expression = Fythe.LanguageExpression => $0
Fythe.Expression = Fythe.Object => $0
Fythe.Expression = Fythe.Literal => $0

Fythe.LanguageExpression = /\{/ WhiteN LanguageExpression /\}/ WhiteN => $2
Fythe.Object = /\(/ WhiteN Fythe.List /\)/ WhiteN => (UnliteralT $2)
Fythe.Object = /\(/ WhiteN /\)/ WhiteN => ()
Fythe.Literal = /([^\(\)\{\}@\-0-9 \t\r\n]([^\(\)\{\}\\ \t\r\n]*(\\.)*)*)/ WhiteN => $0p
Fythe.Literal = /(\/([^\\\/]*\\.)*[^\\\/]*\/)/ WhiteN => $0p
Fythe.Literal = /-?[0-9]+/ WhiteN => (SToIntegerT $0p)

Fythe.List = Fythe.Expression => (Object $0)
Fythe.List = Fythe.List Fythe.Expression => (ConcatT $0 (Object $1))
```

Note that `UnliteralT`, `SToIntegerT` and `ConcatT` are described in chapter 4.

The following table describes the behavior of all parser-related Fythe IR expressions.

Name	P	Types	Behavior

Name	P	Types	Behavior
Parse	5	siiss	The input specified by the fifth argument is parsed by the parser named by the fourth argument. If the parsing succeeds, the result is an object with the following members (respectively): the value generated by parsing, the remaining text associated with Null, the line number if the first character not parsed associated with Null, and the column of the first character not parsed associated with Null. If parsing fails, the result is Null. The first three arguments are a filename, line number and column number, and are only present to allow Parse to create useful error messages.
GAdd	3	soo→s	An expression and target is added to the nonterminal named by the first argument. The second argument is the expression to parse, and must be an object, each element of which is a string naming a parser. The third argument is the target. The result is the first argument.
GRem	1	s→s	All expressions and targets associated with the parsed named by the first argument are removed from the grammar. The result is the first argument.

Chapter 4

The Transform and Macro Engine

(Yet to be written)