The Slate Programmer's Reference Manual

Brian Rice and Lee Salzman

22nd October 2002

Contents

1 Introduction

Slate is a member of the Smalltalk family of languages which supports an object model in a similar prototype-based style as Self[2], extended and re-shaped to support multiple-dispatch methods. However, unlike Self, Slate does not rely on a literal syntax that combines objects and blocks, using syntax more akin to traditional Smalltalk. Unlike a previous attempt at providing prototype-based languages with multiple dispatch, Slate is dynamic and more free-form. It is intended that both Smalltalk and Self styles of programs can be ported to Slate with minimal effort. Finally, Slate contains extensions including syntactic macros, optional keywords, optional type-declarations and subjective dispatch, that can be used to make existing programs and environment organizations more powerful.

Slate is currently implemented as an interpreter written in Common Lisp, which loads source files to build a full environment. A complete bootstrap is under development, which will involve many of the optimizations of the Self system.

Conventions Throughout this manual, various terms will be highlighted in different ways to indicate the type of their significance. If some concept is a certain programming utility in Slate with a definite implementation, it will be formatted in a `typewriter-style`. If a term is technical with a consistent definition in Slate, but cannot have a definite implementation, it will be set in SMALL CAPITAL LETTERS. Finally, just emphasis is denoted by *italics*. Finally, when expression/result patterns are entered, typewriter-style text will be used with a `Slate>` prompt before the statement and its result will be set in *`italicized typewritten text`* below the line.

Finally, many of the examples assume that the full standard library set has been loaded, or at least the fundamental set. To perform this, execute `'src/init.slate' fileIn.` when the interpreter is running. Additional libraries can be loaded with a similar syntax.

2 Language Reference

2.1 Objects

OBJECTS are fundamental in Slate; everything in a running Slate system consists of objects. Slate objects consist of a number of slots and roles: slots are mappings from symbols to other objects, and roles are a means of organizing code that can act on the object. Slots themselves are accessed and updated by a kind of message-send which is not distinguishable from other message-sends syntactically, but have some important differences.

Objects in Slate are created by *cloning* existing objects, rather than instantiating a class. When an object is cloned, the created object has the same slots and values as the original one. The new object will also have the access and update methods for those slots carried over to the new object, but other methods will not propagate due to reasons explained in section 2.3 on Methods.

Both control flow and methods are implemented by specialized objects called blocks, which are code closures. These code closures contain their own slots and create activation objects to handle run-time context when invoked. They can also be stored in slots and sent their own kinds of messages.

2.1.1 Block Closures

A block closure represents an encapsulable context of execution, containing local variables, input variables, the capability to execute expressions sequentially, and finally returns a value to its point of invocation.

Block closures have a special syntax for building them up syntactically. Blocks can specify input slots and local slots in a header between vertical bars (| |), and then a sequence of expressions which comprises the block's body. Block expressions are delimited by square brackets. The input syntax allows you to specify the slot names desired at the beginning. For example,

```
Slate> [| :i j k | j: 4. k: 5. j + k - i].
[]
```

creates and returns a new block. Within the header, identifiers that begin with a colon such as `:i` above are parsed as input slots. The order in which they are specified is the order that arguments matching them must be passed in later to evaluate the block. If the block is evaluated later, it will return the expression after the final stop (the period) within the brackets, `j + k - i`. In this block, `i` is an input slot, and `j` and `k` are local slots which are assigned to and then used in a following expression. The order of specifying the mix of input and local slots does not affect the semantics, but the order of the input slots directly determines what order arguments need to be passed to the block to assign them to the correct slots.

In order to invoke a block, the client must know how many and in what order it takes input arguments. Arguments are passed in using one of several messages. By evaluating these messages, the block is immediately evaluated, and the result of the evaluation is the block's execution result.

Blocks that don't expect any inputs respond to `value`, as follows:

```
Slate> [| a b | a: 4. b: 5. a + b] value.
9
```

Blocks that take one, two, or three inputs, each have special messages `value:`, `value:value:`, and `value:value:value:` which pass in the inputs in the order they were declared in the block header. Every block responds properly to `values:` however, which takes an array of the input values as its other argument.

```
Slate> [| :x :y | x quo: y] value: 17 value: 5.
3
Slate> [| :a :b :c | (b raisedTo: 2) -
(4 * a * c)]
 values: {3. 4. 5}.
-44
```

If a block is empty, or contains an empty body, it returns `Nil` when evaluated:

```
Slate> [] value.
Nil
Slate> [| :a :b |] values: {0. 2}.
Nil
```

Blocks furthermore have the property that, although they are a piece of code and the values they access may change between defining the closure and invoking it, the code will "remember" what objects it depends on, regardless of what context it may be passed to as a slot value. This is critical for implementing good control structures in Slate, as is explained later. Basically a block is an activation frame composed with an environment that can be saved and invoked (perhaps multiple times) long after it is created.

2.1.2 Slot Properties

Slots may be mutable or immutable, and explicit slots or delegation slots. These four possibilities are covered by four primitive methods defined on all objects.

Slate provides several primitive messages to manage slots:

`object addSlot: slotSymbol` adds a slot using the symbol as its name, initialized to `Nil`.

`object addSlot: slotSymbol valued: val` adds a slot under the given name and initializes its value to the given one.

`object addDelegate: slotSymbol` and `object addDelegate: slotSymbol valued: val` adds a delegation slot, and initializes it, respectively. It is recommended to use the latter since delegation to `Nil` is unsafe.

Each of the former has a variant which does not create a mutator method for its slot: `addImmutableSlot:valued:` and `addImmutableDelegate:valued:`.

2.2 Expressions

Expressions in Slate consist of message-sends to argument objects. In Slate, the left-most argument is not considered the implicit receiver. This can mostly be ignored when invoking methods, however.

An important issue is that every identifier is *case-sensitive* in Slate, that is, there is a definite distinction between what `AnObject`, `anobject`, and `ANOBJECT` denote even in the same context. Furthermore, the current implementation is whitespace-sensitive as well, in the sense that whitespace must be used to separate identifiers in order for them to be considered separate. For example, `ab+4` will be treated as one identifier, but `ab + 4` is a message-send expression.

There are three basic types of messages, with different syntaxes and associativities: unary, binary, and keyword messages. *Precedence* can of course be overridden by enclosing expressions in parentheses. An implicit left-most argument can be used with all of them.

A concept that will be often used about message-sends is that of the name of a message, its SELECTOR. This is the symbol used to refer to the message or the name of a method that matches it. Slate uses three styles of selectors, each with a unique but simple syntax.

2.2.1 Unary message-sends

A UNARY MESSAGE does not specify any additional arguments. It is written as a name following a single argument.

Some examples of unary message-sends to explicit arguments include:

```
Slate> 42 print.
42
Slate> 'Slate' clone.
'Slate'
```

Unary sends associate from left to right. So the following prints the factorial of 5:

```
Slate> 5 factorial print.
120
```

Which works the same as:

```
Slate> (5 factorial) print.
120
```

Unary selectors can be most any alpha-numeric identifier, and are identical lexically to ordinary identifiers of slot names. This is no coincidence, since slots are accessed via a type of unary selector.

2.2.2 Binary message-sends

A BINARY MESSAGE is named by a special non-alphanumeric symbol and 'sits between' its two arguments. Binary messages are also evaluated from left to right; there is no special *precedence* difference between any two binary message-sends.

These examples illustrate the precedence and syntax:

```
Slate> 3 + 4.
7
Slate> 3 + 4 * 5.
35
Slate> (3 + 4) * 5.
35
Slate> 3 + (4 * 5).
23
```

Binary messages have lower *precedence* than unary messages. Without any grouping notation, the following expression's unary messages will be evaluated first and then passed as arguments to the binary message:

```
Slate> 7 factorial + 3 negated.
5037
Slate> (7 factorial) + (3 negated).
5037
```

Binary selectors can consist of one or more of the following characters:

```
# $ % ^ & * - + = ~ / \ ? < > , ;
```

However, these characters are reserved:

```
@ [ ] ( ) { } . : ! | `
```

2.2.3 Keyword message-sends

A KEYWORD MESSAGE is an alternating sequence of keywords and expressions. Keywords are identifiers beginning with a letter and ending with a colon. Keyword messages start with the left-most argument along with the longest possible sequence of keyword-value pairs. The SELECTOR of the message is the joining-together of all the keywords into one symbol, which is the *name* of the message. For example,

```
Slate> 5 min: 4 max: 7.
7
```

is a keyword message-send named `min:max:` which has 3 arguments: 5, 4, and 7. However,

```
Slate> 5 min: (4 max: 7).
5
```

is a different kind of expression. Two keyword message-sends are made, the first being `max:` sent to 4 and 7, and `min:` sent to 5 and the first result.

Keywords have the lowest *precedence* of message-sends, so arguments may be the results of unary or binary sends without explicit grouping required. For example, in

```
Slate> 5 + 4 min: 7 factorial max: 8.
9
Slate> (5 + 4) min: (7 factorial) max: 8.
9
```

the former basically parses into the latter.

2.2.4 Expression sequences

Expressions occur between stop-marks, which are periods. At the top-level, expressions aren't evaluated until a full stop is entered. The stop mark also means that expression results aren't directly carried forward as an argument to the following expression; side-effects must be used to keep the results.

Slate provides for a bare expression sequence syntax that can be embedded within any grouping parentheses, as follows:

```
Slate> 3 + 4.
7
Slate> (7 factorial. 5
 negated) min: 6.
-5
```

The parentheses are used just as normal grouping, and as you'll note, the `5 negated` expression wraps over a line, but still evaluates that way. (We do not consider this expression good style, but it illustrates the nature of the language.)

2.2.5 Implicit-context sends

Within methods, blocks, and even at the top-level, some expressions may take the surrounding context as the first argument. There is a precedence for the determination of which object becomes the first argument, which is entirely based on lexical scoping. So, within a block, an implicit send will take the block's run-time context as argument, and then at lesser precedences will be the next outer contexts in sequence, up to the top-level and what it inherits from.

There are some very common uses of implicit-context sends. In particular, accessing and modifying local variables of a block or method is accomplished entirely this way, as well as returns. For example,

```
[| :i j k |
 j: i factorial.
 k: (j raisedTo: 4).
 j < k ifTrue: [| m |
   j: j - i. m: j. ^ (m raisedTo: 3)].
 k: k - 4.
 k
].
```

is a block which, when invoked, takes one argument and has another two to manipulate. Notice that the local slot `j` is available within the enclosed block that also has a further slot `m`. Local blocks may also *override* the slots of their outer contexts with their input and local slots. In this case, the identifiers `j` and `j:`, for example, are automatically-generated accessing and update methods on the context. Because `j:` is a keyword message, if the assigned value is a keyword message-send result, it must be enclosed in parentheses to distinguish the keyword pattern. The `^ (m raisedTo: 3)` message causes the context to exit prematurely, returning as its value the result of the right-hand argument. All methods have this method defined on them, and it will return out to the nearest named block or to the top-level.

In some cases, it may be necessary to manipulate the context in particular ways. In that case, it can be directly addressed with a loop-back slot named `thisContext`, which refers to the current activation. The essence of this concept is that within a block, `x: 4.` is equivalent to `thisContext x: 4.`[1]

2.3 Methods

METHODS in Slate are basically annotated block closures, coupled with annotations of the objects roles that dispatch to them.

2.3.1 Method declarations

Method declaration syntax is handled relatively separately from normal precedence and grammar. It essentially revolves around the use of the reserved character "`@`". If any identifier in a message-send is found to contain the character, the rest of the same send is examined

[1]The current named method as distinct from the context is available as `currentMethod`, and its name is available as `selector`. However, these are dependent on the current implementation of Slate, and so may not be available in the future.

for other instances of the symbol. The parser then treats the expression or identifier to the right of the @ character as a dispatch target for that argument position. After the message-send, there is expected a block expression of some kind, whether a literal or an existing block. Whichever is specified, the parser creates a new block out of it with adjustments so that the identifiers in the dispatching message-send become input slots in the closure. The block should be the final expression encountered before the next stop (a period).

There is a further allowance that an input slotname specifier may be solely an underscore (but not an underscore followed by anything else), in which case the argument to the method at that position is *not* passed in to the block closure.

This syntax is much simpler to recognize and create than to explain. For example, the following are a series of message definitions adding to boolean control of evaluation:

```
_@True ifTrue: block ifFalse: _ [block value].
_@False ifTrue: _ ifFalse: block [block value].

bool@(Boolean traits) ifTrue: block
"Some sugaring for ifTrue:ifFalse:."
[
  bool ifTrue: block ifFalse: []
].
```

The first two represent good uses of dispatching on a particular individual object (dispatching the ignored symbol "_" to `True` and `False`, respectively) as well as the syntax for disregarding its value. Within their blocks, `block` refers to the named argument to the method. What's hidden is that the block given as the code is re-written to include those arguments as inputs in the header. The latter method is defined in terms of the first two, since `True` and `False` both delegate to `Boolean traits`.

2.3.2 Lookup semantics

Message lookup in Slate involves all arguments in concert. Each object contains a table separate from its slot table that contain dispatch annotations per ROLE. An object's roles are its set of possible positions within a method that is defined upon it.

Object slots which are designated as delegate slots will be traversed recursively to continue the lookup process if the object does not define a method which matches the message's signature.

The lookup process involves searching the role dictionary of each argument and their delegates in turn to *first* find selectors matching the message. These sets of methods found from each of the role dictionaries are then intersected to provide a filtered set of applicable methods. This final set visits each argument from left to right, determining whether the argument object does *not* support the method. If one of them doesn't support the method, the next method is searched in a similar way. Otherwise, the method is immediately applied. If none of the methods were applicable, then each of the arguments' delegates are considered in turn. Finally, when the delegates are exhausted, an error is reported.

There are two definite orderings that are important to consider. First, the arguments are more significant to the dispatch on the left than to the right. So of two applicable methods, the one that matches the left-to-right order first is the one that is applied. Second, delegate slots that are searched in the reverse order that they were added: more recent additions override older delegates for the same object.

Finally, this procedure of matching messages with methods is convenient in that omitting a dispatch annotation because it is not necessary results in no change in the semantics of the method or whether or not it is applicable. Moreover, a sparser use of dispatch annotations means that fewer methods need to be collated during the lookup process, which generally means that the process will be faster.

So the recommended style of using dispatch annotations is minimalist, which helps for genericity and for one performance aspect. This also allows for a smooth transition from single-dispatch use without any conceptual or otherwise penalty.

2.3.3 Resending messages or dispatch-overriding

Because Slate's methods are not centered around any particular argument, the resending of messages is formulated in terms of giving the method activation itself a message. The simplest type of resend is `resend`, which finds the next most-applicable method and invokes it with the exact same set of arguments. The result of `resend` is the returned result of that method.

- `methodName findOn: argumentArray` locates the method for the given symbol name and group of argument objects.
- `methodName findOn: argumentArray after: aMethod` locates the method following the given one with the same type of arguments as above.
- `methodName sendTo: argumentArray` is an explicit application of a method, useful when the symbol name of the method needs to be provided at run-time.

- `sendWith:`, `sendWith:with:` and `sendWith:with:with:` take one, two, and three arguments respectively as above without creating an array to pass the arguments in.
- `methodName sendTo: argumentArray through: dispatchArray` is an extra option to specify a different signature for the method than that of the actual argument objects.

2.3.4 Type-annotations

Input and local slots' types can be specified statically for performance or documentation reasons, if desired. The special character "!" is used in the same manner as the dispatch annotation "@", but type-annotations can only occur within a block closure's header. The type system and inference system in Slate is part of the standard library, and so is explained later.

2.3.5 Macro-level Methods

The ‘ special character Preceding any selector with a back-tick (`) will cause it to be applied to the parsed pre-evaluated form of its arguments. This provides access to syntax-level methods at run-time and compile-time.

Slate's parser produces syntax trees which are trees of objects with various attributes, so there is some difference from the Lisp family of languages in that simple lists are not the limit of the expression's parsed format.

Quoting and unquoting A few of the macro-methods we have found appropriate already are `` `quote `` and `` `unquote ``, which pass as their run-time result the syntax-level shifted versions of their expressions.

`` `quote `` causes the surrounding expression to use its quoted value as the input for even normal methods.

`` `unquote `` results in an inversion of the action of `` `quote ``, so it can only be provided within quoted expressions. Lisp macro system users will note that this effectively makes ‘quote the same as quasi-quotation. [2]

Labelled quotation In experience with Lisp macros, nested quotation is often found necessary. In order to adequately control this, often the quotation prefix symbols have to be combined in non-intuitive ways to produce the correct code. Slate includes, as an alternative, two

[2] We may also provide these as `` `up `` and `` `down ``, respectively, if there is enough demand for it, and it is not too confusing.

operations which set a label on a quotation and can unquote within that to the original quotation by means of referencing the label.

Most users need time to develop the understanding of the need for higher-order macros, and this relates to users who employ them. For reference, a Lisp book which covers the subject of higher-order macros better than any other is On Lisp. Although it's also been said that Lisp's notation and the conceptual overhead required to manage the notation in higher-order macros keeps programmers from entering the field, so perhaps this new notation will help.

The operators are `expr1 ‘quote: aLiteral` and `expr2 ‘unquote: aLiteral`, and in order for this to work syntactically, the labels must be equal in value and must be literals. As well, the unquoting expression has to be a sub-expression of the quotation. The effect is that nesting an expression more deeply does not require altering the quotation operators to compensate, and it does indicate better what the unquoting is intended to do.

Evaluation at compile-time `‘evaluate` provides compile-time evaluation of arbitrary expressions.

Term or expression substitution (Not Yet Implemented) `‘with:as:` is a protocol for transparent substitution of temporary or locally-provided proxies for environment values and other system elements. This should provide an effective correspondent of the functionality of Lisp's "`with-`" style macros.

Defining new macro-methods Macros must be dispatched (if at all) upon the traits of expressions' syntactic representation. This introduces a few difficulties, in that some familiarity is needed with the parse node types in order to name them. However, only two things need to be remembered:

1. The generic syntax node type is `Compiler SyntaxNode traits`, and this is usually all that is necessary for basic macro-methods.

2. Syntax node types of various objects and specific expression types can be had by simply quoting them and asking for their traits, although this might be too specific in some cases. For example, `4 ‘quote traits` is suitable for dispatching on Integers, but not Numbers in general, or `(3 + 4) ‘quote traits` will help dispatch on binary message-sends, but not all message-sends. Luckily, `[] ‘quote traits` works for blocks as well as methods.

2.3.6 Primitive Methods

Messages prefixed with the underscore character (_) are looked up in a special table in the system for primitive methods.[3]

2.4 Literal Syntax

2.4.1 Characters

Slate's default support for character literals uses the `$` symbol as a prefix. The following printable and non-printable characters require backslash escapes as follows:

Character name	Literal
Escape	`$\e`
Newline	`$\n`
Carriage return	`$\r`
Tab	`$\t`
Backspace	`$\b`
Null	`$\0`
Space	`$\s`
Backslash	`$\\`

All other symbols can be immediately be preceded by `$` in order to construct the Character object for them, for example,
`$a`, `$3`, `$>`, and `$$`
are all Character object literals for `a`, `3`, `>`, and `$`, respectively.

2.4.2 Strings

Strings are comprised of any sequence of characters surrounded by single-quote characters. Strings can include the commenting character (double-quotes) without an escape. Embedded single-quotes can be provided by using the backslash character to escape them (`\'`). Slate's character literal syntax also embeds into string literals, omitting the `$` prefix. All characters that require escapes in character literal syntax also require escapes when used within string literals, with the exception of double-quote marks and the addition of single-quote marks.

The following are all illustrative examples of Strings in Slate:

```
'a string comprises any sequence of charac-
ters, surrounded by single quotes'
'strings can include the "comment delimit-
ing" character'
```

[3]These are subject to change and should not concern the applications programmer until the bootstrap is complete.

```
'and strings can include embedded sin-
gle quote characters by escaping\' them'
'strings can contain embedded
newline characters'
'and escaped \ncharacters'
'' "and don't forget the empty string"
```

2.4.3 Symbols

Symbols start with the pound sign character (`#`) and consist of all following characters up to the next non-escaped whitespace, unless the pound sign is followed exactly by a string literal, in which case the string's contents become the identifier for the symbol. So, for example, `#@`, `#key:word:expression:`, `#something_with_underscores`, and `#'A full string with a \nnewline in it.'` are all valid symbols and symbol literals.

A property of Symbols and their literals is that any literal with the same value as another also refers to the same instance as any other symbol literal with that value in a Slate system. This allows fast hashes and comparisons by identity rather than value hashes. In particular, as with Slate identifiers, a Symbol's value is case-sensitive, so #a and #A are distinct.

2.4.4 Arrays

Arrays can be literally and recursively specified by curly-brace notation using stops as separators. Array indices in Slate are 0-based. So:

```
{4. 5. {foo. bar}}.
```

returns an array with 4 in position 0, 5 at 1, and an array with objects `foo` and `bar` inserted into it at position 2.

Immediate array syntax is provided as an alternative to create the array when the method is compiled, instead of creating a new array on each method invocation. The syntax is identical except that the first opening brace is preceded by the pound sign. The disadvantage is that no run-time values will be usable.

3 The Slate World

3.1 Overall Organization

3.1.1 The lobby

The lobby is the root namespace object for the Slate object system. All 'global' objects are really only globally accessible because the lobby is delegated to by lexical contexts, directly or indirectly. The lobby in turn may delegate to other namespaces which contain different categorized objects of interest to the applications programmer, and this can be altered at run-time.

Every object reference which is not local to a block closure is sent to the enclosing namespace for resolution, which by default is the root namespace, the lobby (nested closures refer first to their surrounding closure). The lobby contains a loopback slot referring to itself by that name. If you wish to add or arrange globals, you can use either implicit sends to the lobby or explicitly reference it. (We usually consider it good style to directly reference it.)

The lobby is essentially a threading context, and in the future bootstrap will be instantiable in that sense.

3.1.2 Naming and Paths

The lobby provides access to the major `Namespaces`, which are objects suitable for organizing things (for now, they are essentially just `Oddball` objects). The most important one is `prototypes`, which contains the major kinds of shared behavior used by the system. Objects there may merely be cloned and used directly, but they should not themselves be manipulated without some design effort. `prototypes` is inherited by the lobby, so it is not necessary to use the namespace path to identify, for example, `Collection` or `Boolean`. However, unless you explicitly mention the path, adding slots will use the lobby or the local context by default.

The `prototypes` namespace further contains inherited namespaces for collections, and can be otherwise enhanced to divide up the system into manageable pieces.

3.2 Core Behaviors

Slate defines several subtle variations on the core behavior of objects:

ROOT The "root" object, upon which all the very basic methods of slot manipulation are defined.

ODDBALL The branch of `Root` representing non-cloneable objects. These include built-in 'constants' such as the Booleans, as well as literals (value-objects) such as Characters and Symbols.

NIL `Nil` is an `Oddball` representing "no-object".

DERIVABLE `Derivable` objects respond to `derive` and `deriveWith:`, which means they can be readily extended.

CLONEABLE `Cloneable` objects are derivables that can be cloned.

METHOD A `Cloneable` object with attributes for supporting execution of blocks and holding compiled code and its attributes.

3.2.1 Default Object Features

Identity == and ~= return whether the two arguments are identical, i.e. the same object. Value-equality (=) defaults to this.

Printing `print` returns a printed representation of the object. This should be overridden. `printOn:` places the result of printing onto a designated Stream.

Delegation-testing `is:` returns whether the first object has the second as one of its delegated objects, directly or indirectly.

Hashing A quick way to sort by object value that makes collections faster to sort through is the `hash` method, which by default hashes on the object's identity, essentially by its address in memory. What's more important about hashing is that this is how value-equality is established for collections; if an object type overrides =, it must also override hash so that `a = b` ⇔ `a hash = b hash`.

Conversion/coercion The `as:` method has a default implementation on root objects. Essentially the purpose of the `as:` protocol is to provide default conversion methods between types of objects in Slate. Some primitive types, such as `Numbers`, override this. For now, if no converter is found or if the objects are not of the same type, the failure answer is `Nil`. Precisely, the behavior of `a as: b` is to produce an object based on `a` which is as much like `b` as possible.

Slot-enumeration For each object, the Symbols naming its slot and delegate slots can be accessed and iterated over, using the accessors `slotNames` and `delegateNames`, which work with the symbol names of the slots, or the iterators `slotsDo:` and `delegatesDo:`, which iterate over the stored values themselves.

3.2.2 Oddballs

There are various Oddballs in the system, and they are non-cloneable in general. However, Oddball itself may be cloned, for extension purposes.

3.3 Traits

Slate objects, from the root objects down, all respond to the message `traits`, which is conceptually shared behavior but is not as binding as a class is. It returns an object which is, by convention, the location to place shared behavior. Most Slate method definitions are defined upon some object's Traits object. This is significant because cloning an object with a traits delegation slot will result in a new object with the same object delegated-to, so all methods defined on that traits object apply to the new clone.

Traits objects also have their own traits object, which is `Traits traits`. This has the important methods defined on it for deriving new prototypes with new traits objects:

- `myObject derive` will return a new clone of the object with a traits object which is cloned from the original's traits object, and a delegation slot set between the traits objects.
- `myObject deriveWith: mixinsArray` will perform the same operation, adding more delegation links to the traits of the array's objects, in the given order, which achieves a structured, shared behavior of multiple delegation. Note that the delegation link addition order makes the right-most delegation target override the former ones in that order. One interesting property of this method is that the elements of `mixinsArray` do not have to be `Derivable`.

3.4 Closures, Booleans, and Control Structures

3.4.1 Boolean Logic

Slate's interpreter primitively provides the objects `True` and `False`, which are clones of `Boolean`, and delegate to `Boolean traits`. Logical methods are defined on these in a very minimalistic way.

Here are the logical methods and their meanings:

Description	Selector
AND/Conjunction	`/\`
OR/Disjunction	`\/`
NOT/Negation	`not`
EQV/Equivalence	`equiv:`
XOR/Exclusive-OR	`xor:`

3.4.2 Basic Conditional Evaluation

Blocks that evaluate logical expressions can be used lazily in other logical expressions. For example,

```
(x < 3) and: [y > 7].
```

only evaluates the right-hand block argument if the first argument turns out to be True.

```
(x < 3) or: [y > 7].
```

only evaluates the right-hand block argument if the first argument turns out to be False.

In general, the basic of booleans to switch between code alternatives is to use `ifTrue:`, `ifFalse:`, and `ifTrue:ifFalse:` for the various combinations of binary branches. For example,

```
x isNegative ifTrue: [x: x negated].
```

ensures that `x` is positive by optionally executing code to make it positive if it's not. Of course if only the result is desired, instead of just the side-effect, the entire expression's result will be the result of the executed block, so that it can be embedded in further expressions.

Conditional evaluation can also be driven by whether or not a slot has been initialized, or whether a method returns `Nil`. There are a few options for conditionalizing on `Nil`:

- `expr ifNil: block` and `expr ifNotNil: block` execute their blocks based on whether the expression evaluates to Nil, and returns the result.
- `expr ifNil: nilBlock ifNotNil: otherBlock` provides both options in one expression.
- `expr ifNotNilDo: block` applies the block to the expression's result if it turns out to be non-Nil, so the block given must accept one argument.

3.4.3 Looping

Slate includes various idioms for constructing basic loops.

- `n timesRepeat: block` executes the block N times.
- `condition whileTrue: block` and `condition whileFalse: block` execute their blocks repeatedly, checking the condition before each iteration.
- `a upTo: b do: block` and `b downTo: a do: block` executes the block with each number in turn from `a` to `b`.
- `a below: b do: block` and `b above: a do: block` act identically to the previous method except that they stop just before the last value. This assists in iterating over array ranges, where the 0-based indexing makes a difference in range addresses by one, avoiding excessive use of `size - 1` calls.

Slate's looping control structures can easily be extended without concern due to the fact that the interpreter unrolls properly tail-recursive blocks into low-level loop code that re-uses the same activation frame. So basically structuring your custom looping code so that it calls itself last within its own body and returns that value will ensure that you don't need increasing stack space per iteration.

3.5 Numeric Objects

All of the normal arithmetic operations (i.e. `+`, `-`, `*`, `/`) are supported primitively between elements of the same type. Type coercion has to be done entirely in code; no implicit coercions are performed by the virtual machine. However, the standard library includes methods which perform this coercion. The interpreter also transparently provides unlimited-size integers, although the bootstrapped system may not do so implicitly.

The following are the rest of the primitive operations, given with an indication of their "signatures":

- `Float raisedTo: Float` is simple floating-point exponentiation.
- `Integer as: Float` extends an integer into a float.
- `Float as: Integer` truncates a float.
- `Integer bitOr: Integer` performs bitwise logical OR.
- `Integer bitXor: Integer` performs bitwise logical XOR.
- `Integer bitAnd: Integer` performs bitwise logical AND.
- `Integer bitShift: Integer` performs bitwise logical right-shift (left-shift if negative).
- `Integer bitNot` performs bitwise logical NOT.
- `Integer >> Integer` performs logical right-shift.
- `Integer << Integer` performs logical left-shift.
- `Integer quo: Integer` returns a quotient (integer division).

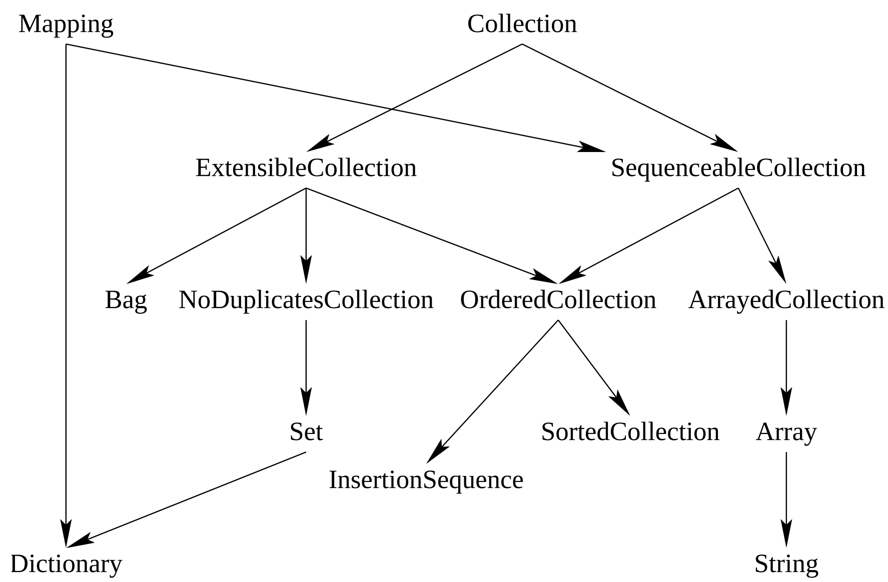
Many more useful methods are defined, such as `mod:`, `reciprocal`, `min:`, `max:`, `between:and:`, `lcm:`, and `gcd:`. Slate also works with `Fractions` when dividing Integers, keeping them lazily reduced.

3.6 Collections

Slate's collection hierarchy makes use of multiple delegation to provide a collection system that can be reasoned about with greater certainty, and that can be extended more easily than other object-oriented languages' collection types.

Figure 1 shows the overview of the collection types, and how their delegation is patterned.

Figure 1: Core Collections Delegations



All collections support a minimal set of methods, including support for basic internal iteration and testing. The following are representative core methods, and are by no means the limit of collection features:

Testing Methods:

- `isEmpty` answers whether or not the collection has any elements in it.
- `collection includes: object` answers whether the collection contains the object.

Properties

- `size` returns the number of elements in it. Sometimes this is calculated dynamically, so it's often useful to cache it in a method.

- `capacity` returns the size that the collection's implementation is currently ready for.

Making new collections

- `newSize:` returns a new collection of the same type that is sized to the argument.
- `newEmpty` returns a new collection of the same type that is sized to some small default value.
- `as:` *alias* `newWithAll:` has extensive support in the collection types to produce copies of the first collection with the type of the second.

Iterating

- `col do: block` executes a block with `:each` (the idiomatic input slot for iterating) of the collection's elements in turn. It returns the original collection.
- `col collect: block` also takes a block, but returns a collection with all the results of those block-applications put into a new collection of the appropriate type.
- `col select: block` takes a block that returns a `Boolean` and returns a new collection of the elements that the block filters (returns True).

3.6.1 Extensible Collections

Collections delegating to `ExtensibleCollection` respond to `add:`, `remove:`, and other protocol messages based upon them, such as the batch operations `addAll:` and `removeAll:`.

3.6.2 Sequences

Sequences (`SequenceableCollection`) are `Mappings` from a range of natural numbers to some objects, sometimes restricted to a given type. Slate sequences are all addressed from a base of 0.

To access and modify sequences, the basic methods `seq at: index` and `seq at: index put: object` are provided.

Arrays `Arrays` are fixed-length sequences and are supported primitively.

Subsequences / Slices `Subsequences` allow one to treat a segment of a sequence as a separate sequence with its own addressing scheme; however, modifying the subsequence will cause the original to be modified.

Cords `Cords` are a non-copying representation of a concatenation of Sequences. Normal concatenation of Sequences is performed with the `;` method, and results in copying both of the arguments into a new Sequence of the appropriate type; the `;;` method will construct a Cord instead. They efficiently implement accessing via `at:` and iteration via `do:`, and `Cord as: SequenceableCollection` will "flatten" the Cord into a Sequence.

Ordered and Sorted Collections An `OrderedCollection` is an Extensible Sequence with some special methods to treat both ends as queues.

Ranges A Range is a sequence of integers between two values, that is ordered consecutively and has some stepping value.

3.6.3 Strings and Characters

`Strings` in Slate are Arrays of characters. Strings and characters have a special literal syntax, and methods specific to dealing with text.

3.6.4 Collections without Duplicates

`NoDuplicatesCollection` forms a special protocol that allows for extension in a well-mannered way. Instead of an `add:` protocol for extension, these collections respond solely to `include:`, which ensures that at least one element of the collection is the target object, but doesn't do anything otherwise. Using `include:` will never add an object if it is already present.

The default implementation of this protocol is `Set`, which stores its elements in a padded array.

3.6.5 Mappings and Dictionaries

Mappings are a general protocol for associating the elements of a set of keys each to a value object. Dictionaries are essentially `Sets` of these associations, but they are generally used with symbols as keys.

Mapping defines the general protocol `at:` and `at:put:` that Sequences use, which also happen to be Mappings. Mappings also support iteration protocols such as `keysDo:`, `valuesDo:`, and `keysAndValuesDo:`.

3.6.6 Trees

Slate includes libraries for binary trees, red-black trees, trees with ordered elements, and tries.

3.6.7 Vectors and Matrices

Slate includes the beginnings of a mathematical vector and matrix library.

3.7 Streams and External Iterators

Streams are objects that act as a sequential channel of elements from (or even *to*) some source.

3.7.1 Basic Protocol

Streams respond to a number of common messages. However, many of these only work on some of the stream types, usually according to good sense:

- `stream next` reads and returns the next element in the stream. This causes the stream reader to advance one element.
- `stream peek` reads and returns the next element in the stream. This does *not* advance the stream reader.
- `stream next: n` draws the next n number of elements from the stream and returns them in a sequence of the appropriate type.
- `stream nextPut: object` writes the object to the stream.
- `stream nextPutAll: sequence` *alias* `stream ; sequence` writes all the objects in the sequence to the stream. The `;` selector allows the user to cascade several sequences into the stream as though they were concatenated.
- `stream do: block` applies the Block to each element of the stream.
- `stream atEnd` answers whether or not the stream has reached some limit.
- `stream upToEnd` collects all the elements of the stream up to its limit into an `OrderedCollection`.

3.7.2 Basic Stream Variants

PositionableStream acts as a means to iterate over a sequence of elements from a Sequence. These streams store their position in the sequence as they iterate.

ReadStream provides input-only access to any source.

WriteStream provides output-only access to any target.

ReadWriteStream allows both read and write access, and caches its input as necessary.

DummyStream is a `ReadStream` that returns `Nil` repeatedly.

BlockStream is a `ReadStream` that targets a no-input Block and returns it's output each time.

FileStream targets a `FileHandle`.

3.7.3 Standard Input/Output

The Slate interpreter provides two Streams primitively, `ConsoleInput` and `ConsoleOutput`, which are `Read`- and `WriteStreams` by default.

3.7.4 Collection Iterator Streams

Each collection type may define its own `Stream` type which goes over its elements in series, even if the collection is not ordered, and only visits each element once. This type's prototype is accessed via the slot Iterator within each collection. So `Set Iterator` refers to the prototype suitable for iterating over `Sets`.

In order to create a new iterator for a specific collection, the `iterator` message is provided, which clones the prototype for that collection's type and targets it to the receiver of the message.

3.8 Files

File access in Slate is currently rudimentary. The interpreter provides an object type `FileHandle` which follows the corresponding protocol:

FileHandle newFor: filename returns a handle for a `String` that names a path to a file.

open opens the file.

exists answers whether there is a file with the handle's pathname.

close closes the file.

read reads the next byte from the file.

position returns the position within the file.

size returns the file size in bytes.

write: char writes one byte to the file.

write: seq writes a sequence of bytes to the file.

name returns the file's pathname.

atEnd answers whether the file's end has been reached.

FileStream newOn: filehandle creates a new Stream to read and write to the file.

Perhaps the most important utility is to load libraries based on path names. `'filename' fileIn` will execute a file with the given path name as Slate source.

3.9 Types

In coordination with the reserved syntax for type-annotation in block headers, Slate's standard libraries include a collection of representations of primitive TYPES as well as quantifications over those types.

The library of types is laid out within the non-delegated namespace `Types` in the lobby.

Any The type that any object satisfies: the universal type.

None The type that no object satisfies: the empty type.

Range A parametrized type over another type with a linear ordering, such as `Integer`. This type is bounded, it has a `start` and a `finish` (least and greatest possible member). In general, any `Magnitude` can be used as a base of a Range type.

Member The type associated with membership in a specific set of objects.

Singleton The type of a single object, as distinct from any other object.

Clone The type of an object and its CLONE FAMILY, the set of objects that are direct copies of it.

Array The Array type is parametrized by an element type and represents arrays of all length of that type.

Block The Block type represents code closures of a given (optional) input and output signature.

Types may be combined in various ways, including `union:`, `intersection:`, and extended via `derive` and `deriveWith:` which preserve type constraints on the derivations.

4 Style Guide

Slate provides an unusual opportunity to organize programs and environments in unique ways, primarily through the unique object-centered combination of prototypes and multiple-argument dispatch.

4.1 Environment Organisation

4.2 Instance-specific Dispatch

4.3 Mini-interpreters Using Macros

References

References

[1] *Multiple Dispatch with Prototypes*. Lee Salzman, 2002

[2] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Holze, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The Self Programmer's Reference Manual*. Sun Microsystems and Stanford University, 4.0 edition, 1995.