Strawman proposal to X3J20, for variable sized argument lists as an alternative to a dynamic array constructor syntax.

Extending the Message Pattern

Brian Wilkerson[1]

Digitalk Inc.

Revision 0

The method pattern specifies both the selector with which the method is associated and the names bound to the arguments provided when the message is sent. In the standard keyword form of the message selector, there is a one-to-one correspondence between keywords in the pattern and keywords in the message expression (and correspondingly between parameter names and the arguments passed to the method).

In keeping with the theme that methods are specified with a "pattern", it seems fitting to explore ways of extending the syntax to allow for more interesting patterns. In particular, we describe two extensions to the keyword form: allowing a keyword to be repeated zero or more times (which we call repeated keywords), and allowing a keyword to appear zero or one times (which we call optional keywords).

Repeated Keywords

Repeated keywords are keywords that can appear zero or more times in a keyword message selector (and correspondingly in a keyword message expression). It effectively defines a variable-argument-count (varargs) syntax.

Semantics

When a method with a repeating keyword parameter is defined, it is semantically equivalent to defining all possible expansions of the selector. For example, if the pattern:

```
a: first b: *second c: third
```

were defined, any message with a selector of a:c:, a:b:c:, a:b:b:c:, a:b:b:b:c:, etc. would be understood.

When a message is sent using one of these selectors, the arguments to the repeated keyword are collected together in an argument array (an object that supports the same protocol as a literal array, but might not be an instance of the same class) which is then bound to the formal parameter associated with the keyword in the method pattern. The size of the argument array is the same as the number of times the keyword was repeated. For example, using the pattern defined above, of a message with the selector a:b:b:c: were sent, the parameter named second would be bound to an argument array with two elements. If a message with the selector a:c: were sent, second would be bound to an empty argument array.

---

[1] Current contact information: Brian Wilkerson, Instantiations Inc., Brian_Wilkerson@Instantiations.com

Multiple repeated keywords are allowed, and may be the only kind of keyword occuring in the message pattern. Of course, no zero length selectors would be defined, because no messages with zero length selectors could be sent.

To allow greater flexibility in implementation, we explicitly specify that the result of sending a message with a selector that matches multiple patterns is undefined.

Example

The standard collection creation protocols—with:, with:with:, with:with:with:, and with:with:with:with:—could be uniformly extended to allow any number of with: keywords, while at the same time the implementation could be simplified by having a single method, by writing the method like the following:

```
with: *initialElements
    "Answer an instance of the receiver containing the contents of the <initialElements> array."

    | collection |
    collection := self new.
    initialElements
        do:
            [:each |
            collection add: each].
    ^collection
```

Implementation Considerations

This section sketches a possible implementation. We make no claims concerning the appropriateness of such an implementation for any existing implementations, we present this implementation merely to demonstrate the feasibility of the feature.

When the compiler compiles a method with a repeating keyword message pattern, it compiles the method as usual, but constructs a special selector, called a *pattern*, that cannot normally appear as the selector for a message and which contains information concerning which keywords can be repeated. A simple implementation of a pattern, valid if method look-up tests for selector identity, would be a string containing special characters before repeating keywords. In the first example above, the pattern might be the string 'a:*b:c:'.

When a message is sent, method look-up proceeds as usual, looking for an exact match. If one is found, the look-up algorithm returns that method. If a match is not found, rather than immediately sending doesNotUnderstand:, the algorithm then looks for any patterns that match the message's selector. The pattern matching algorithm would produce the information required to collect in an array the arguments to the repeating keywords, the original argument list would be replaced with the new list, and the method would be evaluated. If no matching patterns could be found, the message doesNotUnderstand: would be sent as usual.

Such an implementation has the advantage that the cost for supporting repeating keywords is only incurred when a message is sent that matches a pattern (that is, when it is used) or when a message is sent that is not understood.

Note that this implementation implies that the text of the message selectors would be available at run-time. Other implementations might exist that would remove this restriction.

Optional Keywords

Another useful extension to the current semantics is the notion of optional keywords. An optional keyword is one that might or might not be included in the actual message selector.

Semantics

When a method with an optional keyword parameter is defined, it is semantically equivalent to defining the method both with and without the optional keyword. The method without the keyword is equivalent to a method consisting of a send to self of the message that includes the optional keyword, with the expression following the optional parameter's name as the argument to the optional keyword. The default value expressions may not reference any optional parameters.

Example

It is possible to create an Array of a given size, and to optionally supply an initial value for the elements of the array to be used in place of nil. This is currently implemented as two separate methods. Using optional keywords we could implement this in a single method.

```
new: initialSize withAll: [defaultValue := nil]
    "Answer an instance of the receiver with <initialSize> elements, each of which has been initialized to
the <defaultValue>."

    | collection |
    collection := self basicNew: initialSize.
    defaultValue == nil
        ifTrue: [^collection].
    1 to: initialSize
        do:
            [:index |
            collection at: index put: defaultValue].
    ^collection
```

This is equivalent to writing the method with a method pattern of

```
new: initialSize withAll: defaultValue
```

and writing the additional method

```
new: initialSize
    "Answer an instance of the receiver with <initialSize> elements."

    ^self new: initialSize withAll: nil
```

Implementation Considerations

Unlike repeating keywords, when the compiler compiles a method containing optional keywords in its message pattern it can create compiled methods for all of the possible selectors (because there are a finite number of them).

There is no run-time cost associated with optional keywords, and the development cost is only incurred when they are used.

Combined (partial) BNF

The following is a partial BNF for a message pattern that includes the syntax for both of the extensions discussed above.

<message pattern> ::=
    <unary pattern>
    <binary pattern>
    <keyword pattern>

<keyword pattern> ::=
    <keyword specification>
    <keyword specification> <keyword pattern>

<keyword specification> ::=
    <standard keyword specification>
    <repeated keyword specification>
    <optional keyword specification>

<standard keyword specification> ::= <keyword> <parameter name>

<repeated keyword specification> ::= <keyword> '*' <parameter name>

<optional keyword specification> ::= <keyword> '[' <parameter name> ':=' <default value expression> ']'

The semantics of a message pattern is ambiguous if there are two or more non-standard keyword specifications with the same keyword in the same message pattern with no intervening keyword specifications. For example, the following message pattern is ambiguous:

```
with: *firstCollection with: *secondCollection
```

Which arguments would be collected into the first collection, and which into the second? Ambiguous message patterns are therefore specified to be illegal and subject to further specification.