C. Optional Arguments

This proposal for managing optional arguments and "rest" arguments in messages is consistent with the existing syntax of the language, Energetic Secrets (see Appendix D), and efficient implementation.

C.1. Overview

Optional arguments and rest arguments are both extremely useful facilities. They are realizable (more or less conveniently) in languages such as C++ and Scheme. This proposal supports them both directly. It starts by allowing only one method to respond to a given selector or unsealer; that one method can supply optional parameters to handle multiple cases. This keeps a method name associated with a single semantics.

C.2. Receiving Messages

Here is a template that illustrates all the argument-passing idioms:

```
Server serverName
    op sel1: arg1 arg2 arg3
        body...
    op sel2: arg1 arg2 optional arg3 = exp1, arg4 = exp2
        body...
    op sel3: arg1 rest num fn
        body...
endServer
```

The first operation, `sel1:`, is the standard message passing pattern: several argument names following the selector that will be matched against the incoming arguments.

The second operation, `sel2:`, illustrates handling of optional arguments. It includes the keyword **optional**, which signals that the pattern following is for optional arguments only. Each optional argument name is followed by "=" and an expression specifying a default value to be used if that argument is not supplied in a message.

The third operation, `sel3:`, illustrates handling of "rest" arguments, for use when any number of supplied arguments are to be handled generically. The identifier `num` is bound to the number of arguments remaining. The identifier `fn` is bound to a function which can reveal the arguments to the message. Any of the "rest" arguments can then be revealed by calling the argument function `fn` with the index of the argu-

The current definition for "rest" arguments is in terms of a function and arguments. This may be changed to make use of a virtual collection type.

ment to reveal and a distributor on which to reveal it. `fn` can only be called once per index, with the calls in any order. (On further calls with the same index, it must either return the same result or signal an exception.)

C.3. Sending Messages

Sending messages with optional arguments is just like normal message sending. Each optional argument can be either supplied or not; the receiving server accommodates either case. Similarly, messages can be sent to servers that will treat all the arguments generically. Special handling is needed to forward messages generically when manipulating the arguments. This requires the support of Energetic Secrets (see Appendix D).

Sealers for Energetic Secrets support direct protocol that can seal using the same kind of argument count and argument function that the "rest" arguments mechanism supplies. Programs can provide a function directly that supplies the arguments dynamically to the message send. Thus:

```
• receiver (msg:sealer seal*: num fn arg1 arg2)
```

would supply 3 static arguments and any number of dynamic arguments at call time (determined by the combination of `num` and `fn`).

C.4. Other Changes

To support the implicit result argument convention in the presence of optional and "rest" arguments, the implicit result argument will be the first argument in a message. (Previously, it was assumed to be the last argument.) Thus, the "plus" operation would be defined with:

op + result> addend

This allows operations that are used in a functional style to also use optional and rest arguments.