Errors and related matters in CommonLoops - A Proposal

Henry Thompson
11 August 1985

This proposal represents an attempt to provide a set of control primatives for CommonLoops which will

1) Support the existing Interlisp error handling mechanisms (including ERROR and friends, ERRORSET and friends, RESETLST and friends, ERRORTYPELST, BREAKCHECK and its consequences, and the relationships between ERRORX, FAULT1 and BREAK1, all in the context of spaghetti stacks and the existing process mechanisms;
2) Support the CommonLisp constructs catch, throw, unwindprotect, the relationship of unwindprotect to go and return(-from), error, cerror and warn;
3) Substantially reproduce the functionality of the ZetaLisp signalling facility;
4) Be a reasonably plausible attempt to take the high ground wrt whatever proposals the CommonLisp working party on error handling come up with;
5) Be a Good Thing in its own right.

Needless-to-say trying to satisfy all these goals simultaneously is not possible without some compromises, but I think what follows is a good first cut.

The starting point for this design is the Mesa signal and error mechanism, with one key idea borrowed from ZetaLisp. We start with the notion that it must be possible to unwind the stack, either as a part of non-local transfers of control, or as a consequence of abnormal termination. We add to this the notion that at certain points on the stack we may wish to take some action if such an unwinding is underway. Finally we discrimate between actions mandated at some point on the stack but taking place before the unwinding actually starts, and actions which occur at a point on the stack as the unwinding goes by.

Some terminology is in order at this point.

We call the unwinding process **unwinding**. We call the points on the stack at which action wrt unwinding may be specified *Unwind Control Points*, or *UWCPoints* for short. The process by which the user or the system announce circumstances which may provoke unwinding is called signalling, and the concrete representation of the circumstances at issue is called a condition. A condition is a CommonLoops class, and should be a sub-type of class Condition.

Unwind Control Points

UWCPoints are central to this proposal. They provide a vehicle for all activities associated with unwinding, both before the fact in th***e co***ntext of signall*ing,* and as the stack unwinds. A UWCPoint is created with a call to the spread lambda UWCP:

```
(uwcp body catch exit always-do-exits).
```

Its definition is simple:

```
[prog1  (apply body nil)
        (cond   (always-do-exits
                 (apply exit '(normal nil))],
```

but it is what goes on behind the scenes which is important. A UWCPoint is basically a way of evaluating body (actually applying it as a function of no arguments, to allow for closures) in a context which affects what happens in the case of signalling and/or unwinding occuring within the dynamic extent of that UWCPoint. If no signalling or unwinding occurs, the value of uwcp is the value of body. If the stack is unwound past this point, `(apply exit '(unwind <condition>))` will be performed on the way past. This

suffices for unwindprotect and RESETLST (note I use `(f:l args . body)` throughout as short for `(function (lambda args . body))`:

```
(unwindprotect form . cleanups) =>
(uwcp (f:l () form)
         nil
         (f:l (exit-key c)
            (selectq exit-key
               ((unwind normal) . cleanups)
               nil))
         t)
(RESETLST . forms) =>
(LET ((LISPXHIST LISPXHIST)
         (RESETX RESETVARSLST))
      (DECLARE (SPECVARS RESETX))
   (uwcp (f:l () . forms)
         nil
         (f:l (exit-key c)
            (selectq exit-key
               (normal (RESETRESTORE RESETX))
               (unwind (RESETRESTORE RESETX 'ERROR))
               nil))
         t))
```

RESETSAVE and RESETRESTORE are exactly as before. Note that this means that closures may appear on RESETVARSLST in case of e.g.

```
(RESETSAVE xx (LIST (f:l ...) yy))
```

but this is presumably just what is wanted.

Unwinding

The actual unrolling of the stack is performed by the spread lambda UNWIND!:

```
(unwind! frame exit-key condition).
```

frame is a stack pointer to the frame to unwind to. condition is an instance of some sub-class of Condition, descriptive of what is causing the unwinding. In the simple case frame will be a UWCPoint, in which case exit-key will determine what happens when we get there.

unwind! works by scanning the stack upwards via c-links [Larry - should this be a-links? I notice that e.g. GO and RESET chase a-links, not c-links?] from its own frame until it gets to frame. Along the way, whenever it encounters a UWCPoint, it applies the exit argument of that UWCPoint to (list 'unwind condition). When it gets to frame there are two cases. If frame is a UWCPoint, then the unwinding completes with (retapply frame <the exit argument of frame> (list exit-key condition) t). If frame is not a UWCPoint, then the unwinding completes with (apply exit-key (list frame)). This latter case is for non-local 'go's and 'return(-from)'s, see Note on Non-local Xfers below.

A crucial implementation point is that unwind! releases frame before scanning the stack, and uses raw pointers during its scan. This is to prevent the stack from inadvertently being tied down if some unwind clause pre-empts the unwinding by doing its own non-local transfer, something which cannot (and indeed probably should not) be ruled out. This has the further consequence that if frame is not a UWCPoint then it must be re-constituted before being having exit-key applied to it, see above. What would make sense is for unwind! to be defined to get the raw pointer out of the stack pointer, invoke an opcode which does the actual stack scan in microcode, and then reconstruct the stack pointer and do the final apply or retapply.

Signalling

Conditions are signalled with the spread lambda raise-signal:

```
(raise-signal condition can-resume neednt-catch).
```

condition must be an instance of some sub-class of Condition. It identifies the circumstances which provoked the signalling, and may contain relevant parameters. If can-resume is non-nil, then the signal may be resumed, otherwise not (see below). If neednt-catch is non-nil, then the signal need not be caught, otherwise a condition Uncaught will be signalled if it is not caught.
raise-signal works by scanning the stack upwards looking for UWCPoints. When it finds one it applies the catch argument thereof to (list condition). If the result is nil, it continues the scan. If the result is non-nil we say the signal has been caught at that UWCPoint.

What happens next depends on the type of the result. If it is not a list it is called the exit key, and raise-signal exits the signal by causing the stack to unwind to the UWCPoint which caught the signal by calling (unwind! <the UWCPoint> <the exit key> condition). If it is a list then its first element is considered a resume value. If can-resume is non-nil, then the resume value is returned as the value of the call to raise-signal, otherwise an error condition CantResume will be signalled.

If the stack is scanned all the way to the top without the signal being caught, then if neednt-catch is non-nil, the value of raise-signal is nil. Otherwise, the condition Uncaught will be signalled, with instance variables recording the parameters to raise-signal. If it is exited, fine. If it is resumed, the value returned is the value of the call to raise-signal. Otherwise a break is caused around the call to raise-signal.

Note that the application of the catch argument at each UWCPoint is done in the dynamic context of the call to raise-signal, but as the catch argument is lexically in the context of the call to uwcp, if it is a closure its non-special variable references will be to that context.

Three macros are provided for the common cases:

```
(signal condition) => (raise-signal condition t nil) - can resume, must be caught
(error condition) => (raise-signal condition nil nil) - can't resume, must be caught
(notify condition) => (raise-signal condition t t) - can resume, needn't be caught
```

ERROR! and ERRORSET

ERROR! and control-E are now defined as (error \Abort), where \Abort is an instance of class Abort.

ERRORSET is now defined as

```
(lambda (form flag)
               (uwcp (f:l () (list (eval form)))
                     (f:l (condition)
                        (select-type condition
                           (Abort t)))
                     (f:l (exit-key condition)
                           nil))).
```

t is by convention the 'do-nothing' exit key. Note the semantics of ERRORSET are subtly changed by this definition. It is no longer the case that ERRORSET flatly stops the stack from unwinding. What it does is catch Abort, which means it short-stops ERROR!/control-E/^, as it used to, but not unwinding associated with other signals which have been caught overhead. This seems to me to be what is wanted. One could

of course write a catch phrase for Condition to insure catching anything and everything, but that would be pretty dangerous.

catch and throw

These CommonLisp functions are implemented in terms of a sub-class of Condition called Throw:

```
(catch tag . body) =>
(uwcp (f:l () body)
         (f:l (condition)
            (select-type condition
               (Throw (cond ((eq condition:tag tag) 'caught)))))
         (f:l (exit-key condition)
            (selectq exit-key
               (caught condition:value)
                nil)))
(throw tag form) =>
(let ((value form))
      (raise-signal (create Throw tag_tag value_value) nil nil))
```

ERROR, ERRORX and BREAKCHECK

ERROR has a name conflict with the new (and CommonLisp) error - I propose changing its name to old-error and in the short term dicriminating on the basis of the type of the first argument. The only change to old-error is that it now passes its nobreak argument on to ERRORX, which passes it to ERRORX2. ERRORX is unchanged except for that. FAULT1 calls ERRORX2 instead of replicating it - more on this later. ERRORX2 calls BREAKCHECK as before, and then constructs an instance of class SystemError, which is a sub-class of condition, including the error number, message, position, BREAKCHK and PRINTMSG as instance variables, and signals it.

At the top of every process there is a UWCPoint, which inter alia catches Abort, and also handles most of what used to be in ERRORX2. It catches SystemError in order to implement both the built-in and user specified error type list clauses, declining to catch the signal if they don't apply. It catches Uncaught if what wasn't caught was a SystemError, and then either produces the appropriate call to BREAK1 or raises Abort, depending on the recorded value of BREAKCHK. BREAK1, however invoked, signals AboutToBreak before doing anything else. This is all a bit hairy, but the code has been worked out and will be forthcoming.

Non-Local Transfers of Control

Lexical scoping of goto tags and block labels in CommonLisp represents a bit of bother in the Interlisp context. For instance

```
(prog     ((damnFun (f:l (arg)
     (if (weird arg)
          then (go bother)
          else (process arg)))))
     (return (unwindprotect (apply* damnFun 'foo) (cleanup)))
bother
     (return 'lost))
```

works not only in the sense that if 'foo is weird, the value of the prog is 'lost, but also that in that case the unwindprotect is observed and cleanup is called. Now I don't understand how closures are to be implemented in CommonLoops, but I assume the following must be true:

1) The interpreter will continue to exist independently of the compiler (if this is false that just simplifies things a bit).

2) There is a way of identifying on the stack lexical scoping boundary points - that is to say, I presume, frames created by the application of a function definition or a non-quoted argument to apply.
3) The definition of function is such that a closure knows of every non-local lexical variable reference, goto tag and block label within it. This implies inter alia that when running interpreted all macros are expanded by function, and that evaluating or compiling calls to function may produce uba, no such tag or no such label errors.
4) The stack entry for a local variable which is referenced by a closure contains not its value but an invisible pointer to a 'free-floating' value cell, which is also pointed to by the closure.
5) When the interpreter needs the value of a lexical variable, it scans the stack up to the first boundary point and no further. The compiler will presumably be pretty much as now - collapsing all bindings upwards to the boundary frame in so far as possible, and building in references to the right points in the right frames.
6) Thus when a closure is applied to anything, a frame can be built which has entries for all its non-local lexical variable references which will do the right thing.
7) A similar, although messier, approach will work for labels and tags. Messier because although such 'free-floating' value cells may persist after their 'home' frame has gone, 'free-floating' labels and tags must be invalidated when the frame they are based on goes away (see e.g. page 41 of the CommonLisp book). Most of this hair is probably necessary simply to allow 'go's from inside nested progs in any case:

a) Compiled PROG and interpreted \PROG0 frames have two new sorts of entry for tags and labels. Each has two fields, an atom number and a pointer. Every PROG or \PROG0 frame has one tag entry with the atom number for each goto tag it owns, and one label entry for the label of the block, usually nil. In the case where no closures are involved, the pointer field of each entry corresponding to a goto tag contains the appropriate p-counter to transfer to in the case of compiled PROG frames, and the appropriate tail of the PROG body in the case of interpreted \PROG0 frames. The pointer field of the label entry contains nil. By convention (see page 120 of the CommonLisp book) every bounding frame also has a label entry for its frame-name.
b) When a go, return, or return-from is *evaluated,* the stack is scanned to the first boundary point looking for an appropriate tag/label entry. When one is found we call

```
(unwind! <the frame> (f:l (frame) (do-go frame tag
<value of entry>)))
```

if evaluating a go, otherwise

```
(unwind! <the frame> (f:l (frame) (do-return frame label
<value of entry>
   <arg to return>)))
```

c) When a closure is constructed whose body refers to non-local tags or labels, it constructs (or finds, see below) a stack pointer for the frame in which the tag/label is bound, and includes that together with the tag/label in the closure. A pointer to this stack pointer is left in a distinguished part of the frame, so that it can be re-used by other closures, and so that it can be released when the frame goes away. Note that this means it is a special sort of stack pointer, in that its reference to the frame must not be counted.

This also implies an additional cost both in size and time to return for every frame, but I don't see how it can be avoided.

d) When a closure is applied which includes such tags/labels, tag and label entries containing the appropriate atom numbers and the associated stack pointer are included in the constructed frame.

e) It follows from all this that do-go and do-return implement the distinction between local and non-local transfers. If the value of the entry they are passed is not a stack pointer, then they effect the local transfer, via retfrom for do-return, and by appropriate hacking of the PROG (compiled) or \PROG0 (interpreted) frame followed by retto in the case of do-go. If they do get a stack pointer, then they convert themselves into the local case by getting the entry from for the tag/label from the frame pointed to, and doing a further unwind! to that frame with an appropriate re-call of themselves as the exit-key argument. Needless-to-say, if the stack pointer has been released, we get an error.

f) All this is unnecessary for compiled transfers which don't cross any frame boundaries, which can still be coded open.

g) It is not clear to me what will happen in the case of e.g. (eval '(go foo)). If we take the CommonLisp manual seriously, this will fail, as it probably should, because eval will set up a bounding frame with no lexical variables or tags, but then so will most existing uses of eval... I guess this gets beyond what I can reasonably hope to second-guess...

enable

A special form is provided which will be the standard way of producing UWCPoints. It is modelled on the Cedar Mesa ENABLE form, and looks like this:

```
(enable
   c1 => a1 a2 a3 ...
     . . .
   cn => n1 n2 n3 ...
 form
   k1 -> ea1 ea2 ...
    . . .
   kn -> en1 en2 ...)
```

The double arrow lines above are called catch phrases, the single arrow lines are called exit phrases. Evaluates form so as to catch conditions c1, ... cn if they are signalled during its evaluation. If e.g. c1 is signalled, the forms a1 ... an (the catch phrase for c1) will be evaluated in the context of the call which signalled c1. Catch phrases are implemented with select-type, so the order of the condition names is significant. For a catch phrase to be well formed, all control paths through it must end with one of the following four quit forms:

`(exit)`

Causes the stack to unwind back through the enclosing enable form, which is exited with value NIL.

`(resume form)`

Returns from the call which did the signalling with the value of form as the value of that call, if it is resumable, otherwise generates an error.

`(goto exit-key)`

Causes the stack to unwind back to the enclosing enable form, where the exit phrase for exit-key is evaluated. The value of the last form in the phrase is the value of the enable. Exit phrases are implemented with SELECTQ, so lists of exit keys may precede ->.

```
(reject)
```

Causes the signal handling process to act as if the catch phrase had not been there at all.

There is a special exit key whose name is unwind, which has a special meaning. The exit clause for the unwind key will be evaluated whenever the stack unwinds upwards past this point.

There is another special exit key whose name is normal. Its exit clause will be executed in case of a normal return from the enclosed form, without affecting the value of the enable, which will still be the value of the enclosed form.

There is another special exit key whose name is always. always is a just a synonym for (normal unwind). Thus its exit clause will be executed if the stack ever unwinds past it and it will also be executed in case of a normal return from the enclosed form.

Calls to enable translate into calls to uwcp as follows, taking the above template for enable as the input:

```
(uwcp (f:l () form)
         (f:l (condition)
            (select-type condition
                  (c1 a1 a2 a3 ...)
                     . . .
                  (cn n1 n2 n3 ...)))
         (f:l (exit-key condition)
            (selectq exit-key
               (k1 ea1 ea2 ...)
                  . . .
               (kn en1 en2 ...)
               (t nil)
               nil))
         <if normal or always appeared then t else nil>)
(exit) =>
t
(resume form) =>
(list form)
(goto exit-key) =>
(quote exit-key)
(reject) =>
nil
```

Built in Condition classes

Here follow the definitions of class Condition and its built in sub-classes:

```
(defstruct Condition "an unspecialised condition")
```

[Note: It may turn out to be useful to include here instance variables can-resume and neednt-catch which are used instead of the arguments to raise-signal.]

```
(defstruct (Abort (:include Condition)) "an abort condition")
(defstruct (Throw (:include Condition)) "a condition for throwing
to a catch"
   tag value)
(defstruct (SystemError (:include Condition))
   "a condition resulting from a call to ERRORX"
   number message stack-pos breakchk printmsg)
(defstruct (Uncaught (:include Condition))
```

```
   "a condition resulting from an uncaught signal" condition)
(defstruct (AboutToBreak (:include Condition))
   "a condition signalled by BREAK1 on entry")
```

Relation to CommonLisp

This proposal supports everything relevant in the CommonLisp book.