13. Stack Groups

A *stack group* (usually abbreviated 'SG') is a type of Lisp object useful for implementation of certain advanced control structures such as coroutines and generators. Processes, which are a kind of coroutine, are built on top of stack groups (see chapter 29, page 682). A stack group represents a computation and its internal state, including the Lisp stack.

At any time, the computation being performed by the Lisp Machine is associated with one stack group, called the *current* or *running* stack group. The operation of making some stack group be the current stack group is called a *resumption* or a *stack group switch*; the previously running stack group is said to have *resumed* the new stack group. The *resume* operation has two parts: first, the state of the running computation is saved away inside the current stack group, and secondly the state saved in the new stack group is restored, and the new stack group is made current. Then the computation of the new stack group resumes its course.

The stack group itself holds a great deal of state information. It contains the control stack, or *regular PDL*. The control stack is what you are shown by the backtracing commands of the error handler (Control-B, Meta-B, and Control-Meta-B); it remembers the function which is running, its caller, its caller's caller, etc., and the point of execution of each function (the *return address* of each function). A stack group also contains the dynamic environment stack, or *special PDL*. The name 'stack group' derives from the existence of these two stacks. Finally, the stack group contains various internal state information (contents of machine registers and so on).

When the stack group is running, the special PDL contains all the dynamic bindings that are shadowed by other bindings in this stack group; bindings that are current reside in the symbols' value cells. When the stack group is not running, all of the dynamic bindings it has made reside in its special PDL. Switching to a stack group moves the current bindings from the special PDL to the symbol value cells, exchanging them with the global or other shadowed bindings. Switching out of a stack group does the reverse process. Note that unwind-protect handlers are *not* run by a stack-group switch (see let-globally, page 32).

Each stack group is a separate environment for purposes of function calling, throwing, dynamic variable binding, and condition signalling. All stack groups run in the same address space; thus they share the same Lisp data and the same global (not lambda-bound) variables.

When a new stack group is created, it is empty: it doesn't contain the state of any computation, so it can't be resumed. In order to get things going, the stack group must be set to an initial state. This is done by *presetting* the stack group. To preset a stack group, you supply a function and a set of arguments. The stack group is placed in such a state that when it is first resumed it will apply this function to those arguments. The function is called the *initial function* of the stack group.

13.1 Resuming of Stack Groups

The interesting thing that happens to stack groups is that they resume each other. When one stack group resumes a second stack group, the current state of Lisp execution is saved away in the first stack group and is restored from the second stack group. Resuming is also called *switching stack groups*.

At any time, there is one stack group associated with the current computation; it is called the current stack group. The computations associated with other stack groups have their states saved away in memory and are not computing. So the only stack group that can do anything at all, in particular resuming other stack groups, is the current one.

You can look at things from the point of view of one computation. Suppose it is running along, and it resumes some stack group. The state of the computation state is saved away into its own stack group, and the computation associated with the called stack group starts up. The original computation lies dormant in the original stack group, while other computations go around resuming each other, until finally the original stack group is resumed by someone. Then the computation is restored from the stack group and gets to run again.

There are several ways that the current stack group can resume other stack groups. This section describes all of them.

Each stack group records a *resumer* which is **nil** or another stack group. Some forms of resuming examine and alter the resumer of some stack groups.

Resuming has another ability: it can transmit a Lisp object from the old stack group to the new stack group. Each stack group specifies a value to transmit whenever it resumes another stack group; whenever a stack group is resumed, it receives a value.

In the descriptions below, let *c* stand for the current stack group, *s* stand for some other stack group, and *x* stand for any arbitrary Lisp object.

Stack groups can be used as functions. They accept one argument. If *c* calls *s* as a function with one argument *x*, then *s* is resumed, and the object transmitted is *x*. When *c* is resumed (usually—but not necessarily—by *s*), the object transmitted by that resumption is returned as the value of the call to *s*. This is one of the simple ways to resume a stack group: call it as a function. The value you transmit is the argument to the function, and the value you receive is the value returned from the function. Furthermore, this form of resuming sets *s*'s resumer to be *c*.

Another way to resume a stack group is to use **stack-group-return**. Rather than allowing you to specify which stack group to resume, this function always resumes the resumer of the current stack group. Thus, this is a good way to go back to the stack group which called the current one, assuming that this was done through a function call. **stack-group-return** takes one argument which is the object to transmit. It returns when something resumes the current stack group, and returns one value, the object that was transmitted by that resumption. **stack-group-return** does not change the resumer of any stack group.

The most fundamental way to do resuming is with **stack-group-resume**, which takes two arguments: the stack group, and a value to transmit. It returns when someone resumes the current stack group, returning the value that was transmitted by that resumption, and does not affect any stack group's resumer.

If the initial function of *c* attempts to return a value *x*, the regular kind of Lisp function return cannot take place, since the function did not have any caller (it got there when the stack group was initialized). So instead of normal function returning, a "stack group return" happens. *c*'s resumer is resumed, and the value transmitted is *x*. *c* is left in a state ("exhausted") from which it cannot be resumed again; any attempt to resume it signals an error. Presetting it will make it work again.

Those are the "voluntary" forms of stack group switch; a resumption happens because the computation said it should. There are also two "involuntary" forms, in which another stack group is resumed without the explicit request of the running program.

If an error occurs, the current stack group resumes the error handler stack group. The value transmitted is partially descriptive of the error, and the error handler looks inside the saved state of the erring stack group to get the rest of the information. The error handler recovers from the error by changing the saved state of the erring stack group and then resuming it.

When certain events occur, typically a 1-second clock tick, a *sequence break* occurs. This forces the current stack group to resume a special stack group called the *scheduler* (see section 29.1, page 683). The scheduler implements processes by resuming, one after another, the stack group of each process that is ready to run.

current-stack-group-resumer *Variable*
Is the resumer of the current stack group.

current-stack-group *Variable*
Is the stack group which is currently running. A program can use this variable to get its hands on its own stack group.

13.2 Stack Group States

A stack group has a *state*, which controls what it will do when it is resumed. The code number for the state is returned by the function **sys:sg-current-state**. This number is the value of one of the following symbols. Only the states actually used by the current system are documented here; some other codes are defined but not used.

sys:sg-state-active
The stack group is the current one.

sys:sg-state-resumable
The stack group is waiting to be resumed, at which time it will pick up its saved machine state and continue doing what it was doing before.

sys:sg-state-awaiting-return
The stack group called some other stack group as a function. When it is resumed, it will return from that function call.

sys:sg-state-awaiting-initial-call
: The stack group has been preset (see below) but has never been called. When it is resumed, it will call its initial function with the preset arguments.

sys:sg-state-exhausted
: The stack group's initial function has returned. It cannot be resumed.

sys:sg-state-awaiting-error-recovery
: When a stack group gets an error it goes into this state, which prevents anything from happening to it until the error handler has looked at it. In the meantime it cannot be resumed.

sys:sg-state-invoke-call-on-return
: When the stack group is resumed, it will call a function. The function and arguments are already set up on the stack. The debugger uses this to force the stack group being debugged to do things.

13.3 Stack Group Functions

make-stack-group *name* &optional *options*

Creates and returns a new stack group. *name* may be any symbol or string; it is used in the stack group's printed representation. *options* is a list of alternating keywords and values. The options are not too useful; most calls to make-stack-group don't need any options at all. The options are:

:sg-area
: The area in which to create the stack group structure itself. Defaults to the default area (the value of default-cons-area).

:regular-pdl-area
: The area in which to create the regular PDL. Only certain areas specially designated when they were created may be used for regular PDLs, because regular PDLs are cached in a hardware device called the *pdl buffer*. The default is sys:pdl-area.

:special-pdl-area
: The area in which to create the special PDL. Defaults to the default area (the value of default-cons-area).

:regular-pdl-size
: Length of the regular PDL to be created. Defaults to 3000 octal.

:special-pdl-size
: Length of the special PDL to be created. Defaults to 2000 octal.

:swap-sv-on-call-out
:swap-sv-of-sg-that-calls-me
: These flags default to 1. If these are 0, the system does not maintain separate binding environments for each stack group. You do not want to use this feature.

:trap-enable
: This determines what to do if a microcode error occurs. If it is 1 the system tries to handle the error; if it is 0 the machine halts. Defaults to

1. It is 0 only in the error handler stack group, a trap in which would not work anyway.

:safe — If this flag is 1 (the default), a strict call-return discipline among stack-groups is enforced. If 0, no restriction on stack-group switching is imposed.

sys:pdl-overflow (error) *Condition*

This condition is signaled when there is overflow on either the regular pdl or the special pdl. The :pdl-name operation on the condition instance returns either :special or :regular, to tell handlers which one.

The :grow-pdl proceed type is provided. It takes no arguments. Proceeding from the error automatically makes the affected pdl bigger.

eh:pdl-grow-ratio *Variable*

This is the factor by which to increase the size of a pdl after an overflow. It is initially 1.5.

eh:require-pdl-room *regpdl-space specpdl-space*

Makes the current stack group larger if necessary, to make sure that there are at least *regpdl-space* free words in the regular pdl, and at least *specpdl-space* free words in the special pdl, not counting the words currently in use.

stack-group-preset *stack-group function* &rest *arguments*

This sets up *stack-group* so that when it is resumed, *function* will be applied to *arguments* within the stack group. Both stacks are made empty; all saved state in the stack group is destroyed. stack-group-preset is typically used to initialize a stack group just after it is made, but it may be done to any stack group at any time. Doing this to a stack group which is not exhausted destroys its present state without properly cleaning up by running unwind-protects.

stack-group-resume *s x*

Resumes *s*, transmitting the value *x*. No stack group's resumer is affected.

si:sg-resumable-p *s*

t if *s*'s state permits it to be resumed.

sys:wrong-stack-group-state (error) *Condition*

This is signaled if, for example, you try to resume a stack group which is in the exhausted state.

stack-group-return *x*

Resumes the current stack group's resumer, transmitting the value *x*. No stack group's resumer is affected.

symeval-in-stack-group *symbol sg* &optional *frame as-if-current*

Evaluates the variable *symbol* as a special variable in the binding environment of *sg*. If *frame* is not **nil**, it evaluates *symbol* in the binding environment of execution in that frame. (A frame is an index in the stack group's regular pdl).

Two values are returned: the symbol's value, and a locative to where the value is stored. If *as-if-current* is not **nil**, the locative points to where the value *would* be stored if *sg* were running. This may be different from where the value is stored now; for example, the current binding in stack group *sg* is stored in *symbol*'s value cell when *sg* is running, but is probably stored in *sg*'s special pdl when *sg* is not running. *as-if-current* makes no difference if *sg* actually *is* the current stack group.

If *symbol*'s current dynamic binding in the specified stack group and frame is void, this signals a **sys:unbound-variable** error.

13.4 Analyzing Stack Frames

A stack frame is represented by an index in the regular pdl array of the stack group. The word at this index is the function executing, or to be called, in the frame. The following words in the pdl contain the arguments.

sg-regular-pdl *sg*

Returns the regular pdl of *sg*. This is an array of type **art-reg-pdl**. Stack frames are represented as indices into this array.

sg-regular-pdl-pointer *sg*

Returns the index in *sg*'s regular pdl of the last word pushed.

sg-special-pdl *sg*

Returns the special pdl of *sg*. This is an array of type **art-special-pdl**, used to hold special bindings made by functions executing in that stack group.

sg-special-pdl-pointer *sg*

Returns the index in *sg*'s special pdl of the last word pushed.

The following functions are used to move from one stack frame to another.

eh:sg-innermost-active *sg*

Returns (the regular pdl index of) the innermost frame in *sg*, the one that would be executing if *sg* were current. If *sg* is current, the value is the frame of the caller of this function.

eh:sg-next-active *sg frame*

Returns the next active frame out from *frame* in *sg*. This is the one that called *frame*. If *frame* is the outermost frame, the value is *nil*.

eh:sg-previous-active *sg frame*
Returns the previous active frame in from *frame* in *sg*. This is the one called by *frame*. If *frame* is the currently executing frame, the value is nil. If *frame* is nil, the value is the outermost or initial frame.

eh:sg-innermost-open *sg*
Returns the innermost open frame in *sg*, which may be the same as the innermost active one or it may be within that. In other respects, this is like eh:sg-innermost-active.

eh:sg-next-open *sg frame*
Like eh:sg-next-active but includes frames which are *open*, that is, still accumulating arguments prior to calling the function.

eh:sg-previous-open *sg frame*
Like eh:sg-previous-active but includes frames which are *open*, that is, still accumulating arguments prior to calling the function.

eh:sg-frame-active-p *sg frame*
Returns t if *frame* is active; that is, if the function has been entered.

Running interpreted code involves calls to eval, cond, etc. which would not be there in compiled code. The following three functions can be used to skip over the stack frames of such functions, showing only the frames for the functions the user would know about.

eh:sg-next-interesting-active *sg frame*
Like eh:sg-next-active but skips over uninteresting frames.

eh:sg-previous-interesting-active *sg frame*
Like eh:sg-previous-active but skips over uninteresting frames.

eh:sg-out-to-interesting-active *sg frame*
If *frame* is interesting, returns *frame*. Otherwise, it returns the next interesting active frame.

Functions to analyze the data in a particular stack frame:

sys:rp-function-word *regpdl frame*
Returns the function executing in *frame*. *regpdl* should be the sg-regular-pdl of the stack group.

eh:sg-frame-number-of-spread-args *sg frame*
Returns the number of arguments received by *frame*, which should be an active frame. The rest argument (if any) and arguments received by it, do not count.

eh:sg-frame-arg-value *sg frame n*
Returns the value of argument number *n* of stack frame *frame* in *sg*. An error is signaled if *n* is out of range, if the frame is active. (For an open frame, the number of arguments is not yet known, so there is no error check.)

The second value is the location in which the argument is stored when *sg* is running. The location may not actually be in the stack, if the argument is special. The location may then contain other contents when the stack group is not running.

eh:sg-frame-rest-arg-value *sg frame*
Returns the value of the rest argument in *frame*, or **nil** if there is none.

The second value is **t** if the function called in *frame* expects an explicitly passed rest argument.

The third value is **t** if the rest argument was passed explicitly. If this is **nil**, the rest arg is a stack list that overlaps the arguments of stack frame *frame*. If it was passed explicitly, it may still be a stack list, but not in this frame. See section 5.9, page 112 for more information on stack lists.

eh:sg-frame-number-of-locals *sg frame*
Returns the number of local variables in stack frame *frame*.

eh:sg-frame-local-value *sg frame n*
Returns the value of local variable number *n* of stack frame *frame* in *sg*. An error is signaled if *n* is out of range.

The second value is the location in which the local is stored when *sg* is running. The location may not actually be in the stack; if not, it may have other contents when the stack group is not running.

eh:sg-frame-value-value *sg frame n* &optional *create-slot*
Returns the value and location of the *n*'th multiple value *frame* has returned. If *frame* has not begun to return values, the first value returned is **nil** but the location still validly shows where value number *n* will be stored.

If *frame* was called with **multiple-value-list**, it can return any number of values, but they do not have cells to receive them until *frame* returns them. In this case, a non-**nil** *create-slot* means that this function should allocate cells as necessary so that a valid location can be returned. Otherwise, the location as well as the value is **nil**.

eh:sg-frame-value-list *sg frame* &optional *new-number-of-values*
Returns three values that describe whether *frame*'s caller wants multiple values, and any values *frame* has returned already.

The first value is a list in which live the values being, or to be, returned by *frame*.

The second value is **nil** if this frame has not been invoked to return multiple values, a number which is the number of values it has been asked for, or a locative, meaning the frame was called with **multiple-value-list**. In the last case, the first value includes only the values *frame* has returned already, and the locative points to a cell that points to the cons whose cdr should receive the next link of the list.

The third value is how many values *frame* has returned so far.

If *new-number-of-values* is non-nil, it is used to alter the "number of values already returned" as recorded in the stack group. This may alter the length of the list that is the first value. The value you get is the altered one, in that case.

eh:sg-frame-special-pdl-range *sg frame*

Returns two values delimiting the range of *sg*'s special pdl that belongs to the specified stack frame. The first value is the index of the first special pdl word that belongs to the frame, and the second value is the index of the next word that does not belong to it.

If the specified frame has no special bindings, both values are nil. Otherwise, the indicated special pdl words describe bindings made on entry to or during execution in this frame. The words come in pairs.

The first word of each pair contains the saved value; the second points to the location that was bound. When the stack group is not current, the saved value is the value for the binding made in this frame. When the stack group is current, the saved value is the shadowed value, and the value for this binding is either in the cell that was bound, or is the saved value of another binding, at a higher index, of the same cell.

The bit sys:%%specpdl-closure-binding is nonzero in the first word of the pair if the binding was made before entry to the function itself. This includes bindings made by closures, and by instances (including self). Otherwise, the binding was made by the function itself. This includes arguments that are declared special.

symeval-in-stack-group can be used to find the value of a special variable at a certain stack frame (page 261).

13.5 Input/Output in Stack Groups

Because each stack group has its own set of dynamic bindings, a stack group does not inherit its creator's value of *terminal-io* (see page 460), nor its caller's, unless you make special provision for this. The *terminal-io* a stack group gets by default is a "background" stream that does not normally expect to be used. If it is used, it turns into a "background window" that will request the user's attention. Often this happens when an error invokes the debugger.

If you write a program that uses multiple stack groups, and you want them all to do input and output to the terminal, you should pass the value of *terminal-io* to the top-level function of each stack group as part of the stack-group-preset, and that function should bind the variable *terminal-io*.

Another technique is to use a closure as the top-level function of a stack group. This closure can bind *terminal-io* and any other variables that should be shared between the stack group and its creator.

13.6 An Example of Stack Groups

The canonical coroutine example is the so-called samefringe problem: Given two trees, determine whether they contain the same atoms in the same order, ignoring parenthesis structure. A better way of saying this is, given two binary trees built out of conses, determine whether the sequence of atoms on the fringes of the trees is the same, ignoring differences in the arrangement of the internal skeletons of the two trees. Following the usual rule for trees, **nil** in the cdr of a cons is to be ignored.

One way of solving this problem is to use *generator* coroutines. We make a generator for each tree. Each time the generator is called it returns the next element of the fringe of its tree. After the generator has examined the entire tree, it returns a special "exhausted" flag. The generator is most naturally written as a recursive function. The use of coroutines, i.e. stack groups, allows the two generators to recurse separately on two different control stacks without having to coordinate with each other.

The program is very simple. Constructing it in the usual bottom-up style, we first write a recursive function that takes a tree and **stack-group-returns** each element of its fringe. The **stack-group-return** is how the generator coroutine delivers its output. We could easily test this function by changing **stack-group-return** to **print** and trying it on some examples.

```
(defun fringe (tree)
  (cond ((atom tree) (stack-group-return tree))
        (t (fringe (car tree))
           (if (not (null (cdr tree)))
               (fringe (cdr tree))))))
```

Now we package this function inside another, which takes care of returning the special "exhausted" flag.

```
(defun fringe1 (tree exhausted)
  (fringe tree)
  exhausted)
```

The **samefringe** function takes the two trees as arguments and returns **t** or **nil**. It creates two stack groups to act as the two generator coroutines, presets them to run the **fringe1** function, then goes into a loop comparing the two fringes. The value is **nil** if a difference is discovered, or **t** if they are still the same when the end is reached.

```
(defun samefringe (tree1 tree2)
  (let ((sg1 (make-stack-group "samefringe1"))
        (sg2 (make-stack-group "samefringe2"))
        (exhausted (ncons nil)))
    (stack-group-preset sg1 #'fringe1 tree1 exhausted)
    (stack-group-preset sg2 #'fringe1 tree2 exhausted)
    (do ((v1) (v2)) (nil)
      (setq v1 (funcall sg1 nil)
            v2 (funcall sg2 nil))
      (cond ((neq v1 v2) (return nil))
            ((eq v1 exhausted) (return t))))))
```

Now we test it on a couple of examples:

```
(samefringe '(a b c) '(a (b c))) => t
(samefringe '(a b c) '(a b c d)) => nil
```

As stack groups are large, and slow to create, it is desirable to avoid the overhead of creating one each time two fringes are compared. It can easily be eliminated with a modest amount of explicit storage allocation, using the resource facility (see page 124). While we're at it, we can avoid making the exhausted flag fresh each time; its only important property is that it not be an atom.

```
(defresource samefringe-coroutine ()
   :constructor (make-stack-group "for-samefringe"))

(defvar exhausted-flag (ncons nil))

(defun samefringe (tree1 tree2)
  (using-resource (sg1 samefringe-coroutine)
    (using-resource (sg2 samefringe-coroutine)
      (stack-group-preset sg1 #'fringe1 tree1 exhausted-flag)
      (stack-group-preset sg2 #'fringe1 tree2 exhausted-flag)
      (do ((v1) (v2)) (nil)
        (setq v1 (funcall sg1 nil)
              v2 (funcall sg2 nil))
        (cond ((neq v1 v2) (return nil))
              ((eq v1 exhausted-flag) (return t)))))))
```

Now we can compare the fringes of two trees with no allocation of memory whatsoever.