Programming in *Lisp

In Parallel Software Bulletin No. 4 April 1989

Programming in *Lisp *In Parallel*

Place this subsection of the *In Parallel* bulletin at the front of the volume entitled *Programming in *Lisp*, which was distributed with Version 5.0 of CM System Software. Each month, place the new *In Parallel* subsection on top of the one for the previous month.

Reports in This Issue

Programming in *Lisp *In Parallel*

The following restrictions in *Lisp, Versions 5.0 and 5.0.1, were not reported in previous issues of *In Parallel.*

---

*Lisp Language Restrictions

The following restrictions in the *Lisp language, Version 5.0 and 5.0.1, were not previously reported.

ID star-setf-pref-bug

Environment

*Lisp, Versions 5.0 and 5.0.1, any front-end/CM configuration.

Description

The form **(*setf (pref foo (grid i j)) k)** does not work outside of **foo**'s vp set.

Reproduce By

```
> (*cold-boot)
4096
(64 64)
> (def-vp-set matrix (128 128)
           :*defvars ((a (!! 0.0) nil (float-pvar))))
MATRIX
> (ppp a :end 10)
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
> (grid 1 1)
#S(ADDRESS-OBJECT GEOMETRY-ID 2 CUBE-ADDRESS 5)
> (pref a (grid 1 1))
0.0
```

```
> (*setf (pref a (grid 1 1)) 25)
NIL
> (pref a (grid 1 1))
0.0                  ;;; ???
> (ppp a :end 10)
0.0 0.0 0.0 0.0 0.0 25.0 0.0 0.0 0.0 0.0
> (set-vp-set matrix)      ;;; This fixes it
#<VP-SET Name: MATRIX, Allocation form: (CREATE-GEOMETRY
:DIMENSIONS (QUOTE (128 128))), Dimensions (128 128), Ge-
ometry-id: 5, Nesting-level: 0>
> (grid 1 1)
#S(ADDRESS-OBJECT GEOMETRY-ID 5 CUBE-ADDRESS 3)
> (ppp a :end 10)
0.0 0.0 0.0 0.0 0.0 25.0 0.0 0.0 0.0 0.0
> (pref a (grid 1 1))
0.0
> (*setf (pref a (grid 1 1)) 23)
NIL
> (pref a (grid 1 1))
23.0
>
```

Workaround

Call **set-vp-set** before the form (***setf (pref ...)**), as shown in the code example above. Or use **(cube-from-vp-grid-address (pvar-vp-set foo) i j)** instead of **(grid i j)**.

ID star-locally-bug

Environment

*Lisp, Versions 5.0 and 5.0.1, any UNIX front end with any CM configuration.

Description

The *Lisp ***locally** macro occasionally generates code that encounters a restriction in the Lisp compiler. This usually manifests itself as an error message that "the object does not match its declared type."

Reproduce By

```
(defun foo ()
  (let ((i 1))
    (*locally
       (declare (type fixnum i))
       (!! i))))
```

Workaround

Change the optimization levels **speed** and **safety**. This restriction occurs at some levels and not at others.

Alternatively, set the *Lisp compiler option ***verify-type-declarations*** to **nil**. Or get the same effect by setting the *Lisp compiler option ***safety*** to 0.

ID cond-bang-bang-bug

Environment

*Lisp, Versions 5.0 and 5.0.1, any front-end/CM configuration.

Description

The *Lisp macro **cond!!** produces an incorrect expansion when there are no value forms in an arm. For example, the following code does not work:

```
(cond!! ((zerop!! (!! 0))))
```

But the code below does work:

```
(cond!! ((zerop!! (!! 0)) t!!))
```

Workaround

Either provide an explicit value for the arm, or rewrite the **cond!!** expression as an **or!!** expression.

ID sideways-aref-bug

Environment

*Lisp, Versions 5.0 and 5.0.1, any front-end/CM configuration.

Description

The *Lisp function **sideways-aref!!** shares the **aref32** restriction documented in the Paris section of the January 1989 issue of *In Parallel* (page 42). The value of the *index* parameter is checked in unselected processors.

---

*Lisp Simulator Restrictions

The following restrictions in the *Lisp simulator, Version F15, were not previously reported.

ID nested-star-with-vp-set-sim-bug

Environment

*Lisp simulator, Version F15; *Lisp, Versions 5.0 and 5.0.1.

Synopsis

Repeatedly executing code that uses nested ***with-vp-set** forms eventually causes the *Lisp simulator to transfer control to the debugger.

Description

The state of the mechanism that keeps track of context within a vp set is not reset properly when a vp set is exited. This can cause errors while executing code that uses nested ***with-vp-set** forms and that restricts the currently selected set while inside a nested vp set.

Reproduce By

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *SIM-I;
;;; Base: 10 -*-

(IN-PACKAGE "*SIM-I")

(defun test-pref (count &optional (collisions :no-collisions))
  "Useless function for pref!! test."
  (*cold-boot)
  (*all
    (let ((vp-fine (create-vp-set ´(8 8)))
     (vp (create-vp-set ´(4 4 4))))
      (*with-vp-set vp-fine
   (*let (source!!)
     (*with-vp-set vp
       (*let (dest!!)
         (dotimes (x count)
      (print *sim-i::*css-current-level*)
      (if (zerop (mod x 10)) (format t "~D " x) (princ #\.))
      (*set dest!! (pref!! source!! (self-address!!)
                 :collision-mode collisions))
      ))))))))

(test-pref 50)
```

Workaround

There is no complete workaround. In the above example, simply not using **:no-collisions** allows the code to execute, because only **:no-collisions** traverses a path that causes code with nested vp sets to be used.

ID setf-aref-sim-bug

Environment

*Lisp simulator, Version F15; *Lisp, Versions 5.0 and 5.0.1.

Description

For one-dimensional arrays, the *Lisp simulator confuses array indices with array values when it attempts to do indirect addressing by executing a statement of the following form:

```
(*setf (aref!! ( ...))
```

Reproduce By

```
;;; Create an array, set it to all zeros, and print it out. Next,
;;; try to setf-aref element 1 to value 7. This works in the
;;; *Lisp interpreter, but the simulator sets element 7 to value
;;; 1 instead.

(in-package '*lisp)

(defun setf-bug ()
  (*let ((buff (make-array!! 8 :element-type '(unsigned-byte 8)))
    (index (!! 1))
    (value (!! 7)))
    (dotimes (i 8)
      (*setf (aref!! buff (!! i)) (!! 0)))
    (format t "~%Buff should be all 0's~%")
    (ppp buff :end 1)
    (*setf (aref!! buff index) value)
    (format t "~%Buff should have 7 in position 1 ~%")
    (ppp buff :end 1)))
```

Here is the output from the *Lisp interpreter:

```
> (setf-bug)

Buff should be all 0's

#(0 0 0 0 0 0 0 0)
Buff should have 7 in position 1

#(0 7 0 0 0 0 0 0)     <---- [This is correct.]
NIL
>
```

Here is the output from the *Lisp simulator:

```
> (*cold-boot)
Thinking Machines *Lisp Simulator.  Version 15.0
32
(8 4)
> (setf-bug)

Buff should be all 0's

#(0 0 0 0 0 0 0 0)
Buff should have 7 in position 1

#(0 0 0 0 0 0 0 1)    <----  [Instead, it put a 1 in position 7!]
NIL
>
```

Workaround

Reverse the *value* and *index* arguments.

ID star-defvar-array-or-struct-sim-bug

Environment

*Lisp simulator, Version F15; *Lisp, Versions 5.0 and 5.0.1.

Description

If a proclaimed general pvar or an unproclaimed pvar is initialized to an array pvar or to a structure pvar, the initializing **`*defvar`** form results in an error.

Reproduce By

Compile the following file within the *Lisp simulator environment:

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *SIM-I;
;;; Base: 10 -*-

(IN-PACKAGE "*SIM-I")
```

```
(*cold-boot)

(defconstant TERRA-WIDTH 10)
(defconstant TERRA-LENGTH 10)

;;; TERRA-LENGTH is simply the larger dimension of the TERRAIN.

(defconstant ARMY-GROUP-SIZE 10)

(def-vp-set TERRA `(,TERRA-WIDTH ,TERRA-LENGTH))

(*defstruct (M-Squared)
   (Terrain-Type #\C :type string-char)
   (Occupant 0 :type (unsigned-byte (logcount ARMY-GROUP-SIZE))))


(*defvar TERRAIN (Make-M-Squared!!)
      "The land where all this happens"
      TERRA)
```

At this point, the debugger reports an error in **return-pvar-array-to-pool**.

Workaround

Before initializing a ***defvar** to a structure or array pvar, first ***proclaim** the ***defvar** as a structure or array pvar. For example, insert

```
(*proclaim ' (type (pvar M-Squared) TERRAIN))
```

before the **(*defvar TERRAIN . . .)** form in the example above.

Programming in *Lisp

In Parallel **Software Bulletin** **March 1989**

Programming in *Lisp *In Parallel*

Place this subsection of the *In Parallel* bulletin at the front of the volume entitled *Programming in *Lisp*, which was distributed with Version 5.0 of CM System Software. Each month, place the new *In Parallel* subsection on top of the one for the previous month.

*Lisp Hints

The following are not restrictions, but hints for using Paris programming utilities from *Lisp, Version 5.0.

ID timing-code-hint

Please note that future releases of CM System Software may include a better timing facility, and the current **CM:time** may not continue to be supported.

Environment

*Lisp, Version 5.0, any front-end/CM configuration. **CM:time** is not available when using the *Lisp simulator.

Description

Use the Paris macro **CM:time** to record the execution time of *Lisp code. For example, the following code:

```
(*cold-boot :initial-dimensions '(256 256))
4096
(256 256)

(cm:time (scan!! (!! 1) '+!!))
```

produces a response like the following:

```
Evaluation of (SCAN!! (!! 1) '+!!) took 0.004006 seconds of
elapsed time, during which the CM was active for 0.002058 sec-
onds or 51.38% of the total elapsed time.
```

As the example shows, the **CM:time** macro reports the following information:

- Total elapsed time (0.004 seconds)
- The amount of time the Connection Machine itself was running (0.002 seconds)
- The ratio of these two numbers (51.38%)

This ratio is normally referred to as *CM utilization.* In general, CM utilization increases as the number of virtual processors per physical processor (the vp ratio) increases.

The **CM:time** macro cannot be nested. For example, the following code is incorrect:

```
(CM:TIME
  (progn
    (CM:TIME (subroutine-1))
    (CM:TIME (subroutine-2))
    ))
```

The best way to time code is, therefore, to do the timing layer by layer. For example, to time a program like the following:

```
(defun main ()
  (initialize)
  (step-1)
  (step-2)
  (step-3)
  (cleanup)
  )
```

rewrite it as:

```
(defun main ()
  (cm:time (initialize))
  (cm:time (step-1))
  (cm:time (step-2))
  (cm:time (step-3))
  (cm:time (cleanup))
  )
```

Once you determine how much time each routine takes, remove the **CM:time** calls from this outer layer and put **CM:time** calls around the subroutines that constitute the body of **initialize, step-1, step-2, step-3,** and **cleanup.** For instance:

```
(defun step-1 ()
  (cm:time (substep-1))
  (cm:time (substep-2))
  (cm:time (*set the-answer (substep-3)))
  )
```

In this way you can, layer by layer, determine which procedures and subprocedures are using the most Connection Machine time.

ID determining-memory-use-hint

Please note that future releases of CM System Software may include a better memory-space facility, and the current **CMI::cm-room** may not continue to be supported.

Environment

*Lisp, Version 5.0, any front-end/CM configuration. **CMI::cm-room** is not available when using the *Lisp Simulator.

Description

Use the Paris macro **CMI::cm-room** to determine how much memory you have left. For example, the following code produces the response shown:

```
(cmi::cm-room nil)

Total number of bits per processor: 65536
Number of bits used by connection machine system software:
1536
Number of bits allocated in heap: 3874
Number of bits free in the heap (fragmentation): 0
Number of bits allocated in the stack: 288
Number of free bits: 59837
NIL
```

As the example shows, **CMI::cm-room** reports the following information:

- The total number of bits of memory per physical processor (65536)
- The number of bits reserved for use by the CM System Software (1536)
- The number of bits allocated for use by permanent pvars (i.e., those allocated using ***defvar** and **allocate!!**) (3874)
- A *fragmentation* statistic, which reports the number of bits used up by "holes" in memory. As with any storage management system, memory space can develop gaps between allocated memory areas. With *Lisp, holes can be created when a user deallocates permanent pvars, using ***deallocate** or ***deallocate-*defvars**. In the example shown, no holes in memory space have been created yet.
- The number of bits allocated for use by temporary pvars (i.e., those allocated using ***let**) (288)
- The number of bits available (59837)

Programming in *Lisp

In Parallel Software Bulletin February 1989

Programming in *Lisp *In Parallel*

Place this subsection of the *In Parallel* bulletin at the front of the volume entitled *Programming in *Lisp*, which was distributed with Version 5.0 of CM System Software. Each month, place the new *In Parallel* subsection on top of the one for the previous month.

Contents

*Lisp Restrictions Corrected in Version 5.0.1

The following list of previously reported *Lisp restrictions have been corrected in CM System Software Version 5.0.1. These restrictions were reported in *In Parallel*, Number 1.

allocate-bang-bang-bug
array-to-pvar-bug-1
array-to-pvar-bug-2
heap-memory-not-reclaimed-when-vp-set-deallocated
lisp-too-big
setf-pref-with-address-object-bug
star-defun-bug
star-pset-with-add-bug
star-setf-pref-does-not-reclaim-stack
star-when-bug

Following are additional *Lisp restrictions corrected in CM System Software Version 5.0.1. These were not previously reported.

ID array-to-pvar-grid-bug-1

Environment

*Lisp and Lisp/Paris, Version 5.0, any front-end/CM configuration.

Description

The Lisp/Paris functions that write array data to the CM (i.e., **CM:write-news-array**, **CM:write-array-by-cube-address**, and **CM:write-array-by-news-address**) overloaded the CM input first-in first-out (FIFO) queue in certain situations. Overloading the FIFO had several possible consequences: the data written might have been corrupted, the CM might have crashed, the CM might not have executed following instructions correctly, and if the front end was a VAX it might have crashed. The situation that usually caused the problem was performing an operation that took a long time immediately before calling the write data function. Such operations include communications instructions and other array data-writing functions.

This restriction has been corrected in CM System Software Version 5.0.1. It was caused by the Paris restriction called **bitblt-cross-seq**, which has been corrected in CM System Software Version 5.0.1.

ID array-to-pvar-grid-bug-2

Environment

*Lisp and Lisp/Paris, Version 5.0, VAX front end using any CM configuration.

Description

The Lisp/Paris functions that write array data (**CM:write-news-array**, **CM:write-array-by-cube-address**, and **CM:write-array-by-news-address**) may have caused VAX front ends to crash if there was an error while writing the data. This restriction has been corrected in CM System Software Version 5.0.1. It was caused by the Paris restriction called **bitblt-cross-seq**, which has been corrected in CM System Software Version 5.0.1.

ID self-bang-bang-bug

Environment

The *Lisp compiler, Version 5.0, any front-end/CM configuration.

Description

This restriction has been corrected in CM System Software Version 5.0.1. The operation **self!!** returns a structure pvar containing two slots: one for the send address and one for the geometry ID. Because of an oversight in the *Lisp compiler implementation, only the send address slot was initialized.

ID var-len-pvar-bug

Environment

*Lisp, Version 5.0, any front-end/CM configuration.

Description

If ***let** was called with a variable-length pvar and the pvar was given an initial value, the pvar was allocated in heap memory instead of on the stack where it belonged. Because ***let** was allocating variable-length pvars on the heap, this memory was never de-allocated when the ***let** was exited, unnecessarily reducing the available CM memory. A variable-length pvar is of any one of the following types:

```
(unsigned-pvar *)
(signed-pvar *)
(float-pvar * *)
(complex-pvar * *)
```

This restriction has been corrected in CM System Software Version 5.0.1.

ID vpset-damaged-by-coldboot-detach

Environment

*Lisp, Version 5.0, any front-end/CM configuration.

Description

This restriction has been corrected in CM System Software Version 5.0.1. In one particular circumstance, a defined and instantiated VP set was not re-instantiated after ***cold-boot**. The following series of actions resulted in a run-time error in the VP set initialization code: defining a VP set, cold-booting, detaching, attaching to a CM portion of a different physical size, then cold-booting again.

A *Lisp Interpreter Restriction

The following restriction in Versions 5.0 and 5.0.1 has not been reported previously.

ID star-defstruct-bug

Environment

The *Lisp interpreter, Versions 5.0 and 5.0.1, any front-end/CM configuration.

Description

When a `*defstruct` form is interpreted instead of compiled, attempting to use one of the accessor functions results in an infinite recursion, causing a stack overflow or core dump.

Reproduce By

Evaluate a `*defstruct` form from a Lisp Listener; then call one of its accessor functions.

Workaround

Always compile `*defstruct` forms. It is best to place `*defstruct` forms in a separate file and always use only the binary version of that file.

*Lisp

In Parallel **Software Bulletin** **January 1989**

*Lisp *In Parallel*

Put this section of the *In Parallel* bulletin at the front of the volume entitled *Programming in *Lisp*, which was distributed with Version 5.0 of *Lisp. Each month, put the new *Lisp *In Parallel* section on top of the one for the previous month. This way, the most current notes on using *Lisp will always be available for reference.

*Lisp Language Restrictions

What follows are descriptions of previously undocumented restrictions on various *Lisp language constructs.

Restrictions on Vp Sets

Several restrictions on the creation and use of vp sets with *Lisp Version 5.0 have recently been discovered. These apply to both the *Lisp interpreter and the *compiler.

ID def-vp-set-bug-1

Synopsis

The construct **(def-vp-set foo nil :*defvars ((bar)))** can not be run twice in a row.

Description

If this is attempted, the second call results in an error, complaining that **bar** is unbound. This bug typically occurs during recompilation of a file containing a call to **def-vp-set.**

Reproduce by

```
(def-vp-set foo nil :*defvars ((bar)))
(def-vp-set foo nil :*defvars ((bar)))
```

Workaround

Either ***cold-boot** or deallocate the vp set before re-executing the code.

ID def-vp-set-bug-2

Synopsis

A **def-vp-set** form can not be called twice in a row with intervening ***cold-boot** and **allocate-vp-set** calls.

Description

If this is done, an error message complains of an attempt to use a vp set which has not been instantiated.

Reproduce by

```
(def-vp-set foo nil :*defvars ((bar)))
(*cold-boot)
(allocate-processors-for-vp-set foo '(128 128))
(def-vp-set foo nil :*defvars ((bar)))
```

Workaround

***deallocate** the :*defvar **bar** before the second call to **def-vp-set**.

ID vp-set-redefinition-bug

Synopsis

Trying to redefine a vp set that hasn't had its processors allocated generates an error if a :*defvar from the original vp set definition isn't in the redefinition of that vp set.

Reproduce by

```
(def-vp-set c-vp-set nil :*defvars ((c-var1 t!!)))
(def-vp-set c-vp-set nil :*defvars ((c-var2 t!!)))
```

results in this error message

```
Trap: The variable *LISP::C-VAR1 is unbound.
While in the function *LISP-I:RE-EVALUATE-STILL-EXISTING-
OLD-*DEFVARS   SI:*EVAL   EVAL
```

Workaround

Deallocate the vp set before redefining it.

ID heap-memory-not-reclaimed-when-vp-set-deallocated

Synopsis

A few bits of heap memory are not reclaimed when a vp set is deallocated.

Reproduce by

Allocate a vp set, then deallocate it. Use **CMI::CM-ROOM** before and after. The heap usage will not be the same.

Workaround

Execute

```
(cm:deallocate-heap-field (*lisp-i::vp-set-border-bits my-vp-set))
```

immediately before deallocating **my-vp-set**.

ID setf-pref-with-address-object-bug

Synopsis

The **pref** operation, when composed with **setf**, does not properly reference address objects.

Reproduce by

```
(*defvar sfl (self-address!!) nil big-2d-vp-set)

   (defun foo (x)
     (*with-vp-set 2d-vp-set
       (setf (pref sfl x) 3.4)
       (pref sfl x)))
```

```
 (foo (grid 1 1))
3.0 <<<<<< should be 3.4
 (pref sfl 5)
3.4
```

Workaround

none.

Restrictions on Array Pvars

There are several newly-discovered restrictions on the use of array pvars with *Lisp Version 5.0. These apply to both the *Lisp interpreter and the *compiler.

ID array-to-pvar-bug-1

Synopsis

The **array-to-pvar** operation can *not* write only a portion of a front end array into the CM; the destination pvar must be large enough to receive all front end array elements.

Description

The **array-to-pvar** operation signals an error if given **:cube-address-start** and **:cube-address-end** arguments specifying a number of processors that is less than the number of effective elements in the array—as dictated by the array offset argument. This should be legal and should have the effect of writing the first

```
(- :cube-address-end :cube-address-start)
```

array elements into the pvar processors.

Reproduce by

```
TEST
> (cm:attach)
;;; Loading source file "/usr/local/etc/cm_configuration.lisp"
8192
> (*cold-boot)
```

```
8192
(128 64)
> (array-to-pvar (make-array 100 :initial-element 1.0)
                 test :cube-address-end 50)
>>Error: Starting at array-offset 0, the array provided has 100
elements. But you are attempting to write 50 elements
into the CM
 ...
```

In 4.3 this worked without complaint, putting 1.0 in the first 50 processors.

Workaround

Make a smaller front-end array and use it as the source-array argument to array-to-pvar.

ID array-to-pvar-bug-2

Synopsis

A call to **array-to-pvar** yields incorrect results if the *dest-pvar* is a mutable integer pvar.

Description

The *Lisp **array-to-pvar** operation does not treat variable length destination pvars correctly. It fails to grow the *dest-pvar* to accommodate the source data.

Reproduce by

```
(*defvar integer-pvar (!! 0))
(ppp (array-to-pvar (make-array 10 :initial-element 33)
       integer-pvar :cube-address-end 10) :end 10)
```

This yields:

```
1 1 1 1 1 1 1 1 1 1
```

instead of:

```
33 33 33 33 33 33 33 33 33 33
```

whereas

```
(ppp (array-to-pvar (make-array 10 :initial-element 33)
       nil :cube-address-end 10) :end 10)
```

prints ten 33's, as it should.

Workaround

Don't provide a *dest-pvar* argument within the **array-to-pvar** form. The code above can be made to work thus:

```
(*when (<!! (self-address!!) (!! 10))
  (*set integer-pvar (array-to-pvar
                       (make-array 10 :initial-element 33)
                       nil :cube-address-end 10))
     (ppp integer-pvar :end 10))
```

Alternatively, provide a *dest-pvar* with a definite length (e.g., **(field-pvar 8)**), or initialize the *dest-pvar* with a value that ensures it is large enough to hold all of the data in the source array.

ID nested-array-declare-within-star-let-bug

Synopsis

A nested pvar array declaration does not work properly if variables are used to specify inner dimension lists.

Reproduce by

```
(setq x '(5))
(setq y '(4))
```

This doesn't work:

```
(*let (temp)
  (declare (type (pvar (array (array single-float x) y))
temp))
  nil
  )
```

Workaround

This does work:

```
(*let (temp)
  (declare (type (pvar (array single-float x)) temp))
  nil
  )
```

And so does this:

```
(*let (temp)
  (declare (type (pvar (array (array single-float (10)) y))
temp))
  nil
  )
```

ID allocate-bang-bang-bug

Synopsis

Using **allocate!!** to allocate array pvars whose element type length must be evaluated at run time causes a lisp run time error.

Reproduce by

```
(allocate!! nil nil
 '(pvar (array (unsigned-byte
                 *current-send-address-length*) (3))))
```

Workaround

There is no general workaround. Use backquote if possible:

```
`(pvar (array (unsigned-byte
                ,*current-send-address-length*) (3)))
```

Restriction on *defun Declarations in Lucid Lisp

ID star-defun-bug

Synopsis

In some cases ***defun** does not work in *Lisp running under Lucid Lisp.

Description

The operation ***defun** is a macro. Unless the first forms are **declare** forms, *Lisp will macroexpand them, looking for **declare** forms. If the first forms within a ***defun** need to be macroexpanded and if they implicitly reference the Common Lisp ***safety*** compiler variable, then the ***defun** will not be correctly interpreted or compiled.

The reason for this is that the Lucid compiler erroneously binds ***safety*** to nil.

Reproduce by

```
(*defun foo (x y)
  (*locally (declare (type float-pvar x y))
    (BODY)))
```

Workaround

Only use **declare** forms as the first forms in a ***defun**.

Problems with Memory Use

Three problems with memory usage in *Lisp Version 5.0 have recently been discovered.

ID pref-bang-bang-runs-out-of-memory

Synopsis The message

```
Foward sprint-send-with-trace has exceeded its allowed space for
saving out trace data.
CM Microcode Function: CMI::SAVE-OUT-PETIT-CYCLE-TRACE
```

is indicative of running out of memory using **pref!!** without a **:collision-mode** argument (i.e., using backwards routing).

Description

Repeated calls to **pref!!** will cause *Lisp code to run out of heap space. This is true of both *compiled and *interpreted code.

Workaround

Use a **:collision-mode** argument of **:collisions-allowed** or **:no-collisions**.

ID star-setf-pref-does-not-reclaim-stack

Synopsis

Under certain circumstances, using **(*setf (pref** ... does not reclaim the *Lisp stack after it finishes execution. This is true of both *compiled and *interpreted code.

Description

This occurs when the destination of a **(*setf (pref** ... is not a symbol, but rather an expression. If used in a tight loop, this can result in stack overflow.

Reproduce by

```
(*setf
  (pref (discrete-attribute-value!!
          (aref!! (record-discrete-attribute-array!! *record!!*)
            (!! (the fixnum i))))
     processor)
   pos))
  (print (list 'after (length *lisp-i::*temp-pvar-list*)))
)
```

Workaround

```
(*let () (*setf (pref ...
```

That is. wrap a **(*let () ...)** around the offending form..

ID lisp-too-big

Synopsis

The VAX *Lisp image uses more virtual memory than it should on a VAX front end.

Description

As distributed, the VAX Lisp bands have many more dynamic free segments allocated than are strictly necessary. This causes the Lisp to consume up to 26 megabytes more VM than they need on startup.

Workaround

VAX Customers can reduce Lisp memory usage greatly by reducing the number of free segments in their disksaved Lisp bands.

To reduce this memory usage, do the following:

```
% starlisp
;;; Lucid and TMC copyright messages

> (room t)                   ;;; display amount of memory being used
;;; 42142 words [168568 bytes] of dynamic storage in use.
;;; 2987872 words [11951488 bytes] of free storage available
;;; before a GC.
;;; 6017886 words [24071544 bytes] of free storage available
;;; if GC is disabled.
;;; Semi-space Size: 11840K bytes [185 segments]
;;; Current Dynamic Area: Dynamic-1-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 0K bytes [0 segments]
;;; Memory Growth Limit: 49152K bytes [768 segments], total
;;; Memory Growth Rate: 2048K bytes [32 segments]
;;; Reclamation Ratio: 25% desired free after garbage collection
;;; Area Information:
;;; Name                        Size [used/allocated]
;;; ----                        ----
;;; Foreign-Area                21K/64K bytes,     1/1 segment
;;; Dynamic-0-Area              0K/11836K bytes,   0/185 segments
;;; Lots of free segments
;;; Dynamic-1-Area              165K/11836K bytes, 3/185 segments
;;; Static-Area                 8799K/8832K bytes, 138/138 segments
;;; Read-Write-Area             837K/896K bytes,   14/14 segments
;;; Readonly-Pointer-Area       1546K/1600K bytes, 25/25 segments
;;; Readonly-Non-Pointer-Area   12392K/12416K bytes,194/194 segments
NIL
> (sys:disksave "/usr/local/starlisp-new" :full-gc t :verbose t
         :dynamic-free-segments 32 :reserved-free-segments 16)


;;;lots of messages from disksave
> (sys:quit)
%
```

Miscellaneous *Lisp Language Restrictions

Two problems to avoid are described below: one bug and one common user error.

ID star-pset-with-add-bug

Synopsis

A bug in the **CM:send-with-f-add-1L** Paris operation results in errors when the *Lisp ***pset** operation is called using the **:add** combiner with floating point or complex data.

Reproduce by

The results obtained follow the code below. Note that the results printed show the answers when the combiner is specified as **:default**, **:no-collisions**, and **:add**.The first two combiner values (**:default** and **:no-collisions**) produce the results expected. The third combiner value (**:add**) gets the wrong numbers.

```
;;; -*- Package:*lisp; Syntax:Common-lisp; Mode:lisp -*-

(in-package '*lisp)
(defmacro !!tf (x) `(!!p (the fixnum ,x)))

(defun buggy ()
  (*locally
    (*let (v test-vp-set m)
      (declare (type (pvar single-float) v))
      (declare (type (pvar (unsigned-byte
                              cm:*cube-address-length*)) m))
      (setq test-vp-set (create-vp-set '(4096)))
      (*set v (!! 1.0))
      (*with-vp-set test-vp-set
        (*let ( (r0 (!! 0.0)) (r1 (!! 0.0)) (r2 (!! 0.0)) )
          (declare (type (pvar single-float) r0 r1 r2))

          ; assembly into a residual vector.

          (*with-vp-set *default-vp-set*
            (*set m (self-address-grid!! (!! 0)))
            (*when (<!! (self-address-grid!! (!! 0)) (!! 10))
              (*pset :no-collisions v r0 m :vp-set test-vp-set)
              (*pset :default v r1 m :vp-set test-vp-set)
              (*pset :add v r2 m :vp-set test-vp-set)
            ) ; end *when.
          ) ; end *with-vp-set.

          (print " final residual vector after assembly.....")
          (dotimes (i 10)
            (format t "~% i=~d; r (no-collisions) =~d;
                                  r (default) =~d;    r (add) =~d"
              i (pref r0 (grid i))
                (pref r1 (grid i)) (pref r2 (grid i))))
        ) ; end *let.
      ) ; end *with-vp-set.
    ) ; end *let.
  ) ; end *locally.
) ; end defun.
```

```
(buggy)
" final residual vector after assembly....."
 i=0; r (no-collisions) =1.0;   r (default) =1.0;    r (add) =1.0
 i=1; r (no-collisions) =1.0;   r (default) =1.0;    r (add) =0.0
 i=2; r (no-collisions) =1.0;   r (default) =1.0;    r (add) =0.0
 i=3; r (no-collisions) =1.0;   r (default) =1.0;    r (add) =0.0
 i=4; r (no-collisions) =1.0;   r (default) =1.0;    r (add) =0.0
 i=5; r (no-collisions) =1.0;   r (default) =1.0;    r (add) =0.0
 i=6; r (no-collisions) =1.0;   r (default) =1.0;    r (add) =0.0
 i=7; r (no-collisions) =1.0;   r (default) =1.0;    r (add) =0.0
 i=8; r (no-collisions) =1.0;   r (default) =1.0;    r (add) =1.0
 i=9; r (no-collisions) =1.0;   r (default) =1.0;    r (add) =0.0
NIL
```

Workaround

There is no obvious workaround for the ***pset** with **:add** bug.

ID copy-bang-bang

Synopsis

The **copy!!** operation may only be used in conjunction with a segment pvar. This is a documented *Lisp restriction but users stumble over it all the time. The *Lisp documentation notes that **'copy!!** may only be used in conjunction with a segment pvar. (See pages 46–47 of the **Lisp Reference Manual*, Version 5.0.)

Reproduce by

Here is a non-inclusive copy scan and its result.

```
(ppp (scan!! (!! 10) 'copy!! :include-self nil) :end 20)
```

```
0 0 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

Instead, we would expect the non-inclusive copy scan to return:

```
0 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

In contrast, an inclusive copy scan correctly returns all 10's:

```
(ppp (scan!! (!! 10) 'copy!! :include-self t) :end 20)
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

Workaround

Always use a segment pvar when using **scan!!** with **copy!!**. For example:

```
(scan!! (!! 10) 'copy!! :include-self nil
  :segment-pvar (zerop!! (self-address!!)))
```

Compiler Restrictions

What follows are descriptions of previously undocumented restrictions on the *Lisp compiler.

ID off-grid-border-p-not-compiling

Synopsis

The **off-grid-border-p!!** operations can not be *compiled by Version 5.0 of the *Lisp compiler. This fact should have been included in the **Lisp Release Notes*, Version 5.0 on page 22.

ID star-pset-not-compiled-properly

Synopsis

The *compilation of ***pset** is invalid in some circumstances.

Description

A *Lisp ***pset** form does not compile properly when the *address-pvar* parameter is an experimental function that doesn't compile (such as **address-plus-nth!!**) and that contains some other *Lisp expression (such as !!) that would have compiled had it not been inside an experimental function form.

Reproduce by

```
(*pset :no-collisions (the (field-pvar length) pvar)
                      (the (field-pvar length) dest)
          (address-plus-nth!! start-address-object rank
            (!! (the fixnum dimension-constant))))
```

Workaround

Put a ***nocompile** around ***pset** forms.

Alternately, don't declare the parameters to the inner form that would have compiled. For the example above, this would yield:

```
(*pset :no-collisions (the (field-pvar length) pvar)
                      (the (field-pvar length) dest)
      (address-plus-nth!! start-address-object rank
        (!! dimension-constant)))
```

ID star-set-fun-dest-mashes-stack

Synopsis

Using a function as the destination argument to ***set** causes the compiler to generate code that incorrectly overwrites a portion of the stack.

Reproduce by

```
(*proclaim '(ftype (function () (pvar bit)) bug-fcn))
(*proclaim '(type (pvar bit) bug-var))
(*defvar bug-var)
(defun bug-fcn () bug-var)

(defun demo-bug ()
  (*let (x!)
    (declare (type (pvar (unsigned-byte 16)) x!))
    nil
    (format t "~% Stack before *SET = ~D"
      cmi::*next-available-stack-maddr*)
    (*set (bug-fcn) (!! 0))
    (format t "~% Stack after *SET = ~D"
      cmi::*next-available-stack-maddr*)
    ))
>>> DEMO-BUG

(demo-bug)
>
>  Stack before *SET = 19
>  Stack after *SET = 4
>>> NIL
```

Obviously, the stack has been bashed.

Workaround

None.

ID star-when-bug

Synopsis

The *Lisp operation ***when** may have trouble *compiling if ***cold-boot** has not yet been called for the first time.

Description

The problems, when they occur, can manifest in several different ways. Essentially, the compiler does not know that it is doing an operation that affects which processors are active.

Reproduce by

*Compile either of the following ***when** expressions in a *Lisp that has not ever executed ***cold-boot**. The first expression causes an internal inconsistency message.

```
(*when
  (and!!
    (not!! (off-grid-border-relative-p!! (!! 1) (!! 1)))
    (news!! (the boolean-pvar new-edge!!) 1 1))
  nil)

(*when
  (and!!
   (not!! (contour-point-head-p!! contour-points))
```

```
        (local-point-real-point-p!! hull-points))
       nil)
```

Workaround

Call ***cold-boot**, or do the following:

```
(*lisp-i::setup-context-flag)
(*lisp-i::setup-test-flag)
```

ID star-proclaim-star-defun-bug

Synopsis

The *Lisp construct **(*proclaim '(*defun...** fails to allow the *Lisp compiler to use proclaimed type information for forward references.

Description

If an operation is ***proclaimed** as a ***defun**, or if the return value of a function is ***proclaimed**, or both, then code containing forward references to the ***proclaimed** operation will nonetheless *not* be *compiled.

Reproduce By

```
(*proclaim '(*defun foo))
(*proclaim '(ftype (function () (pvar (unsigned-byte 10))) foo))


(*defun function-using-foo ()
   (*let ((some-pvar (foo nil)))
     (declare (type (pvar (unsigned-byte 10)) some-pvar))
       some-pvar))
```

The **some-pvar** initialization expression in this code can not be *compiled. There is no error message.

Workaround

Avoid forward references or use **the** to give type information for the forward references. For example:

```
(*let
  ((some-pvar (the (pvar (unsigned-byte 10))(foo nil))))
...)
```

Simulator News

The *Lisp simulator, Version 5.0, is now available. All customer sites should have received a copy of the simulator. Call TMC Customer Support if your site does not yet have the 5.0 *Lisp simulator.