Putting Lisp on Forth Base

Harold Carr and Robert R. Kessler

Utah Portable Artificial Intelligence Support Systems Project
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract: There has been much interest in building a Lisp system on top of Forth, especially a Forth engine. A common approach has been to write the Lisp system from scratch, in Forth. We propose an alternate approach based upon the existing Portable Standard Lisp (PSL) system. The PSL compiler emits code for an abstract machine with 15 registers. Typically, the abstract machine code and registers are mapped to target operations and registers/memory locations. In this paper we explore two approaches to bootstrapping Lisp in Forth: 1) create a Forth wordset to emulate the PSL abstract machine, 2) map PSL's abstract machine operations to Forth primitives.

Introduction: Our group has been involved in building Lisp systems for a decade. Our approaches have ranged from defining a Standard Lisp dialect [Marti 79], which is then implemented upon other Lisp dialects via compatibility packages, through modifying the compiler to emit Pascal code to bootstrap Lisp on Terraks, to the current PSL/PCLS system [Griss 82] [Shebs 86], which is operational on Vax BSD Unix, Vax VMS, Apollo DOMAIN Aegis, HP Series 9000 HP-UX, HP Integrated Personal Computer, Sun BSD, Iris BSD, Gould BSD, Cray CTSS, Cray COS, IBM 370 VM/CMS, and MacIntosh.

We are currently developing Utah Common Lisp (UCL). This differs from our previous PCLS system in that we are writing a new compiler, rather than use the existing PSL compiler. A Forth-based machine emulator [Carr 87a] has been written to help validate the early phases of the new compiler. It became apparent that the ideas used in the emulator could be extended to provide a mechanism to bootstrap our Lisp systems to a Forth base. Although we are moving toward the UCL system, this paper will develop a plan to bootstrap the existing PSL/PCLS system, since it is well understood and time-tested.

The idea is to perform a half-bootstrap of PSL from an existing host implementation to a Forth base. First we compile the PSL runtime system to the Forth base. Once that is operational we load the compiler into the PSL/Forth system and then have it compile the compiler. The half-bootstrap is then complete and we have a running PSL on a Forth base, independent of the porting host.

We will explore three central issues: 1) implementing Lisp data types and their associated operations, 2) handling control operations (jumps, calls, etc.), and 3) obtaining low level support from the operating system (io, signals, saving images, etc.). (All Forth examples will be in Bradley Forthware's CForth-83.)

Data Types and Operations: Implementing Lisp data types turns out to be the easier task. Instead of writing a heap in Forth [Dress 86], we just compile the existing heap and data operations of PSL to Forth. For instance, the definition of `cons`:

```
(de cons (a b)
  (let ((ptr (gtheap (pairpack))))
    (setf (wgetv ptr 0) a)
    (setf (wgetv ptr 1) b)
    (mkpair ptr)))
```

compiles to the following abstract machine code:

```
(*entry cons expr 2)
(*push (reg 2))
(*push (reg 1))
(*move (quote 2) (reg 1))
(*link gtheap expr 1)
(*move (frame 1) (memory (reg 1) (wconst 0)))
(*move (frame 2) (memory (reg 1) (wconst 4)))
(*mkitem (reg 1) (quote 9))
(*exit 2)
```

`*entry` is a pseudoop that indicates the beginning of a procedure. `cons` saves its arguments in the stack then calls (`*link`) `gtheap`, the canonical PSL heap allocator, with an argument of 2 (PSL passes its arguments in registers). `gtheap` returns a pointer to the allocated memory in `(reg 1)`. `cons` then *moves its arguments into the newly allocated pair via the `memory` (indirect plus offset) abstract machine addressing mode. PSL is a tagged architecture, so it tags the `cons` pointer via `*mkitem` with the `cons` tag (9). This is left in `(reg 1)` as the result of the call to `cons`. `(*exit 2)` deallocates the stack and does a return.

We can emulate this code in Forth quite easily by transforming it from prefix to postfix, handling source operands before, and destination operands after, the abstract machine operation (which expect their operands, and leave their results in the Forth stack). `reg` is an array which places its base address plus the offset (which precedes it) times element size on the stack:

```
: cons
   2 reg @ *push
   1 reg @ *push
   2 *move 1 reg !
   *link gtheap
   1 frame *move 1 reg @ 0 memory ! \ frame accesses the stack
   2 frame *move 1 reg @ 4 memory ! \ frame is 1-based
   1 reg @ 9 *mkitem 1 reg !
   2 *exit
;
```

This emulation approach is useful for precise control of abstraction machines operations and addressing modes, but is clearly redundant. Instead, we can map the abstract operations directly to primitive Forth operations:

```
: cons
   2 reg @
   1 reg @
   2 1 reg !
   gtheap
   0 pick 1 reg @ !
   1 pick 1 reg @ 4 + !
   9 27 shift 1 reg @ or 1 reg ! \ assumes tag occupies upper 5 bits
   drop drop                     \ of 32 bit word
;
```

The `car` of a `cons` cell can be obtained by stripping the tag and dereferencing the pointer (no type checking in this example):

```
: car
   th 7ffffff 1 reg @ and        \ strip tag - hex base
   @ 1 reg !                     \ dereference pointer
;
```

In these examples, each Lisp procedure passed arguments and returned results in the abstract registers, which are emulated in Forth. If we were porting to a Forth engine, we would have the compiler emit code for a stack machine, rather than a register machine, to take advantage of the

Forth engine's stack cache.

Handling Control Operations: Abstract machine control operations are straightforward to map to Forth if they have been generated in response to structured control operations in Lisp. For example, the string space allocator:

```
(de allocate-string (len)
  ;; Make str with Len chars, uninitialized
  (if (intp len)
    (if (wlessp len 0)
      (nonpositiveintegererror len 'allocate-string)
      (mkstr (gtstr (- len 1))))
    (nonintegererror len 'allocate-string)))
```

generates abstract machine code with labels and jumps:

```
(*entry allocate-string expr 1)
(*push (reg 1))
(*jumpnotintype (label #:g0217) (reg 1) posint-tag)
(*move (frame 1) (reg 1))
(*jumpwgeq (label #:g0219) (reg 1) (quote 0))
(*move (quote allocate-string) (reg 2))
(*linke 1 nonpositiveintegererror expr 2) ;tail recursive call (exit)
(*lbl (label #:g0219))
(*wplus2 (reg 1) (wconst -1))
(*link gtstr expr 1)
(*mkitem (reg 1) (quote 4))               ;4 is string tag
(*jump (label #:g0137))
(*lbl (label #:g0217))
(*move (quote allocate-string) (reg 2))
(*move (frame 1) (reg 1))
(*linke 1 nonintegererror expr 2)         ;tail recursive call (exit)
(*lbl (label #:g0137))
(*exit 1)
```

One way to handle the branch is to make each basic block of straight line code into a separate Forth word as done in [Carr 87a]. Although this is acceptable in an emulator the overhead is probably too costly for a full implementation of PSL. Instead, we can use the standard Forth branching words:

```
: allocate-string
   1 reg @
   1 reg @ th 7ffffff and posint-tag = not ?branch [ >mark ]
   0 pick 1 reg !
   1 reg @ 0< ?branch [ >mark ]
   quote allocate-string 2 reg !
   drop nonpositiveintegererror unnest  \ tail call
   [ >resolve ]
   -1 1 reg +!
   gtstr
   4 27 shift 1 reg @ or 1 reg !
   branch [ >mark ]
   [ swap >resolve ]
   quote allocate-string 2 reg !
   0 pick 1 reg !
   drop nonintegererror unnest          \ tail call
   [ >resolve ]
   drop
;
```

Although this example was easy to handle (and backward branches can be handled in the same way), the compiler can emit unstructured control sequences, and the user can program them using `tagbody` and `go`. In general, we will need to build a table to associate labels with their addresses for use in resolving branches.

Operating System Services: PSL has standard system routines which are generally interfaced to the existing services provided by the target's operating system. Input/output (including file i/o) is generally straightforward to implement since every system we have encountered provides these

services at some level. The most problematical services are `unexec`, which saves an executing image to be restarted later; `oload`, which load and links foreign (non-Lisp) code into the system; and signal handling (not all machines provide signal catching). `unexec` and `oload` require knowledge of the target's binary file formats. Although conceptually straightforward, the details and limitations in a particular target can be overwhelming. Generally, we write these services in the target's native language, then link them into the PSL kernel.

Conclusion: PSL is ported to new machines via a half-bootstrap process. What we are proposing here is the standard PSL port: Forth being the target instruction set, rather than, say, the Vax or 68K instruction set. There is little reason to port to a Forth implementation running on conventional hardware (although it might be a potential base for portability). This becomes interesting if the target is a Forth engine. Existing Forth engines provide hardware support for 16 bit words. This size is too small to support the full range of Lisp programs. Bank switching can help but portions of the existing PSL system would have to be rewritten to take this into account. If Forth engines are to provide a useful platform for other languages they should directly handle modern word widths (typically 32 bits) and present a flat addressing space. Also, our experience with the FAIM-1 Symbolic Multiprocessor [Carr 87b] indicates that it is beneficial to include some limited general purpose registers in a stack machine to hold frequently referenced items.

Acknowledgments: Work supported in part by the Defense Advanced Research Projects Agency under contract number DAAK11-84-K-0017, and by the Hewlett-Packard Corporation.

References

[Carr 87a] Carr, H; Kessler, R. R.
An Emulator for Utah Common Lisp's Abstract Virtual Register Machine.
In *Proceedings of the 1987 Rochester Forth Conference.* 1987.

[Carr 87b] Carr, H.
Popcorn: A Kernoil Compiler for the FAIM-1 Symbolic Multiprocssing System.
Utah PASS Project OpNote 87-04, University of Utah, Department of Computer Science, April, 1987.

[Dress 86] Dress, W. B.
A FORTH Implementation of the Heap Data Structure for Memory Management.
Journal of Forth Application and Research 3(3):39-49, July, 1986.

[Griss 82] Griss, M. L.; Benson, E.; Maguire, G. Q. Jr.
PSL: A Portable LISP System.
In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, pages 88-97. ACM, Carnegie-Mellon University, Pittsburgh, Pa., 1982.

[Marti 79] Marti, J. B., et al.
Standard LISP Report.
SIGPLAN Notices 14(10):48-68, October, 1979.

[Shebs 86] Shebs, S.; Loosemore, S.
Portable Common Lisp Subset Users Guide.
Utah PASS Project OpNote 86-04, University of Utah, Department of Computer Science, May, 1986.