An Emulator for Utah Common Lisp's Abstract Virtual Register Machine

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

A virtual machine emulator (written in Forth) is being used to aid in the development of the Utah Common Lisp compiler. The compiler emits code for an abstract machine with an unlimited number of virtual registers. The register allocator then assigns these virtual registers to real registers and stack frames of the target machine. Once the registers have been assigned, the abstract operations are then mapped to target machine instructions. This paper shows the use of the emulator on virtual register code (before register allocation).

Introduction

The developing Utah Common Lisp (UCL) system is a successor to the Portable Standard Lisp (PSL) [Griss 82] and Portable Common Lisp Subset (PCLS) [Shebs 86] systems. Like its predecessors, UCL has portability as a major goal, initial targets being the CRAY, VAXEN and 680?0s. Work is advancing on three fronts: a bootstrap kernel (written in C), the runtime system (which is loaded into the bootstrap kernel), and the compiler (which compiles the runtime system code, as well as user code).

Instead of waiting for the compiler's register allocator and machine-specific mapping phases to be completed, we wrote a virtual register machine emulator in Forth to exercise code generated by its initial code generation phase. We have used the emulator on numeric functions to validate the code being generated. The rest of this paper will use a simple example to illustrate how the emulator works.

Virtual Register Machine Code Format

The code generation phase of the UCL compiler is given a Lisp function. It produces a control-flow graph of basic blocks. Each basic block contains a straight line sequence of abstract machine operations. Each function is assumed to start with a fresh set of unlimited virtual registers which become the operands to the abstract machine operations. We will use a linear iterative version of FACTORIAL as an example:

```
(defun factorial (n)
  (labels ((iter (product counter)
             (if (> counter n)
               product
               (iter (* counter product)
                     (+ counter 1)))))
    (iter 1 1)))
```

We intercept the results of the compiler before register allocation, formating the basic blocks as a series of Forth words:

```
:: ***initcode***
    quote factorial entry bb1002 entry setf-symbol-function %call drop
    quote factorial %move 30 vr !
    \ %return
;

***initcode***

:: bb1002
    2 vr !
    1 1 entry bb1001 %call 27 vr !
    27 vr @ \ %return
;

:: bb1001
    6 vr ! 5 vr !
    6 vr @ 2 vr @ entry > %call 9 vr !
    nnil 9 vr @ entry eq %call true? if entry #:else-1002 %jump then
    entry #:ft-1004 %fall
;

:: #:ft-1004
    5 vr @ %move 4 vr !
    entry #:end-if-1003 %jump
;

:: #:else-1002
    6 vr @ 5 vr @ entry * %call 13 vr !
    6 vr @ 1 entry + %call 16 vr !
    13 vr @ 16 vr @ entry bb1001 %call 4 vr !
    entry #:end-if-1003 %fall
;

:: #:end-if-1003
    4 vr @ \ %return
;
```

: : is a defining word which acts like : if the word being defined is not already in the dictionary. Otherwise it enters a headless word for the new definition into the dictionary, and changes the existing definition to be a colon word, with its first parameter field pointing to the headless word, and the second parameter field pointing to `exit`. `vr` is an array which places its base address plus the offset (which precedes it) times element size on the stack.

The words which represent the basic blocks make forward references, so before the basic block file is loaded, a file of stubs is loaded first:

```
::: factorial undefined-function ;
::: bb1002 undefined-function ;
::: setf-symbol-function undefined-function ;
::: factorial undefined-function ;
::: bb1001 undefined-function ;
::: > undefined-function ;

... (remaining forward references omitted)
```

This file is created by passing over the basic block file and making a stub for any word preceded by `quote` or `entry` (not to be confused with Forth's `entry`). : : : is a defining word which acts like : if the word being defined is not already in the dictionary. Otherwise, if the word already exists, it ignores the : : : definition.

Operation

`***initcode***` is the first colon definition encountered when loading the basic block code file. It is executed immediately after it is defined. `quote` and `entry` are immediate words

which take the words following them in the input stream and compile their name field address and code field addresses (respectively) as literals. When `***initcode***` is executed `setf-symbol-function` makes `factorial` an alias for `bb1002`.

The entry basic block for each function expects its arguments in the Forth stack. These values are immediately stored into virtual registers. The final basic block of each function leaves its result in the Forth stack. All other communication between the basic blocks of a single function is done through the virtual registers.

`bb1002` is the entry block for the `factorial` function. It takes its single argument from the stack and stores it in virtual register 2. It then places two ones on the stack and `%calls` `bb1001`, the entry block for the internal `iter` function. The result of this internal operation is stored in virtual register 27. Finally, the value stored in virtual register 27 is placed on the stack as the return value of the `factorial` function.

Abstract Machine Operations

Abstract operations are words which begin with `%`. This example uses the basic control operations.

- `%call` - a "saved-pc" function call (which could be optimized away in the Forth emulator, since typical Forth word nesting will implement it).
- `%jump` and `%fall` - perform "no-save-pc" jumps to the given blocks (popping the return stack in the process, so the word that executes them is never returned to).
- `%return` - is commented out, since the normal Forth return mechanism is used.

The only data operation, `%move` is a noop in this implementation.

Limitations

We can only emulate single iterative functions. Recursive functions, with their chain of deferred operations, do not produce correct results because the virtual registers are not unique between the recursive calls. This can be seen clearly in the code to the standard recursive version of factorial:

```
(defun fact (n)
  (if (= n 1)
    1
    (* n (fact (- n 1)))))

:: ***initcode***
    quote fact entry bb1003 entry setf-symbol-function %call drop
    quote fact %move 20 vr !
;

***initcode***
```

```
:: bb1003
    2 vr ! \ %entry 1003 vr 17
    2 vr @ 1 entry = %call 6 vr !
    nnil 6 vr @ entry eq %call true? if entry #:else-1006 %jump then
    entry #:ft-1008 %fall
;

:: #:ft-1008
    1 %move 17 vr !
    entry #:end-if-1007 %jump
;

:: #:else-1006
    2 vr @ 1 entry - %call 11 vr !
    11 vr @ entry fact %call 12 vr !
    2 vr @ 12 vr @ entry * %call 17 vr !
    entry #:end-if-1007 %fall
;

:: #:end-if-1007
    17 vr @ \ %return
;
```

The instantiation of `fact` making the recursive call to itself in basic block `#:else-1006` expects all of its virtual registers to remain untouched, but the recursive call operates out of the same registers. This also makes it impossible to emulate more than single functions.

Extensions

The emulator has been useful to validate the compiler's code generators. When the register allocator is completed we will use the emulator on allocated code, rather than virtual register code. This will be an interesting mix, where we simulate the real registers of, say, a 68020, but still emulate the abstract machine operations. At this point, the restrictions on single, iterative function emulation will be removed, since the allocated code will do the proper context saving and restoring. The ideas discussed in this paper can be extended to provide a method to bootstrap a full Lisp system to a Forth base [Carr 87].

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 87] Carr, H; Kessler, R. R.
Putting Lisp on Forth Base.
In *Proceedings of the 1987 Rochester Forth Conference*. 1987.

[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.

[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.