Summary: Superoperators in Interpreters

Sam Martin
January 27, 2011

The most pressing issue when using an interpreter is of course, speed. Interpreted programs run significantly slower than compiled versions of the same code. This penalty can be as severe as 9-16 times slower. A large portion of the slow down is dude to the overhead of executing each instruction. The interpreter must read the bytecode one line at a time, decode the instruction that it has been passed - and then finally actually execute that instruction. This process is expensive, but unavoidable. A keen insight is to realize that a lot of slowdown is due to this decoding. If we could somehow make each byte opcode more meaningful and powerful, we could speed up our program by a large amount. Take for instance the following example:

a = 2 + x
b = 3 + x
c = 5 + x
d = 1 + x

If we were to convert this to bytecode (assuming 1 is assignment, 2 is add and 3 is push constant, 4 is push variable:

.byte 3
.word 2
.byte 4
.word address(x)
.byte 2
.byte 4
.word address(a)
.byte 1

.byte 3
.word 3
.byte 4
.word address(x)
.byte 2
.byte 4
.word address(b)
.byte 1

.byte 3
.word 5
.byte 4

```
.word address(x)
    .byte 2
    .byte 4
.word address(c)
    .byte 1

    .byte 3
    .word 1
    .byte 4
.word address(x)
    .byte 2
    .byte 4
.word address(d)
    .byte 1
```

Now: if we instead create a super operator (5) will now be a unary operator that adds x to a constant

```
    .byte 3
    .word 2
    .byte 5
    .byte 4
.word address(a)
    .byte 1

    .byte 3
    .word 3
    .byte 5
    .byte 4
.word address(b)
    .byte 1

    .byte 3
    .word 5
    .byte 5
    .byte 4
.word address(c)
    .byte 1

    .byte 3
    .word 1
    .byte 5
    .byte 4
.word address(d)
    .byte 1
```

The hybrid translator/interpreter actually compiles small pieces of code to optimize the run speed. These compiled pieces are the function prologues of the code. Function prologues are the various operations that need to be done before one can execute a function call, such as stack manipulation and register management. This saves the interpreter a lot of time decoding instructions that have to be executed every time a function is called and thus greatly increase the speed.

The hti can produce superoperators to optimize for either speed or size. If one wishes to optimize for speed, it is useful to find the subtrees of the abstract syntax tree that will be most frequently executed at run-time and create superoperators from those. If one wishes to optimize the size of the input program instead, it is most useful to find the subtrees that occur most frequently in the abstract syntax tree (though not necessarily at run time) and create superoperators from those instead. To actually optimize fully is a NP-complete problem, so the hti instead uses heuristics like these to obtain reasonable results.

The hti uses the output from lcc to obtain the abstract syntax tree and processes that file rather than raw source code. The lcc has the ability to produce 109 operators in its translation. Using all of the possibilities in a byte in our bytecode, hti now has 147 options with which to use superoperators. In practice, hti could even use more superoperators by replacing some of the 109 operators that lcc can produce but never uses. Imagine a program that uses absolutely no floating point operations. The abstract syntax tree produced by the lcc will not feature any nodes with floating point operations in this case. For the sake of argument, assume that lcc has a dozen operators related to floating point arithmetic. Hti is now free to generate up to 159 superoperators to optimize the interpreted code. In practice, this ends up being many more superoperators than necessary to achieve reasonable results. The author of the paper postulates that twenty superoperators should be sufficient to obtain desired results for an improved interpreter.

The heuristic used by hti is a very simple greedy algorithm. Depending on whether the code is being optimized for run-speed or code length, the tree generated by lcc will be weighted by the number of times we expect a section to run or the number of times a section appears, respectively. It then finds the most heavily weighted pair of parent/child nodes and merges all instances of those into superoperator. This process is repeated multiple times until the code is of a desired size or the number of desired superoperators has been achieved. It is very possible that this process will not obtain an optimal layout of superoperators, but it creates a useful set that enables significant speed up.

The hti tool is an effective way to create very fast and compact code to be interpreted. While the slow-down factor without the use of superoperators is between 8 and 16, the slow-down factor of the interpreted code with the addition of superoperators is only 3 to 9. This gives us a speed-up of between 2 and

3 on average, which is very important for interpreted code. Useful extensions to the hti could include support of different front-end parsers. The lcc has inherent attributes of the abstract syntax trees that it creates that limit the hti's effectiveness. For example, the lcc cannot express a ? b in a single tree node, which hampers hti's ability to create superoperators.