Loop Headers in λ-calculus or CPS

Andrew W. Appel
Princeton University
appel@princeton.edu

CS-TR-460-94
Princeton University
June 15, 1994

Abstract

As is well known, the introduction of a "loop header" block facilitates the hoisting of loop-invariant code from a loop. But in a λ-calculus intermediate representation, which has a notion of scope, this transformation is particularly useful.

Loop headers with scope also solve an old problem with in-line expansion of recursive functions or loops: if done naively, only the first iteration is inlined. A loop header can encapsulate the loop or recursion for better in-line expansion.

This optimization improves performance by about 5% in Standard ML of New Jersey.

I have previously described [2, pp. 83–92] the in-line expander of the Standard ML of New Jersey compiler. Its purpose is not merely to avoid function-call overhead by inserting the bodies of functions in place of their calls. What is more important is that further optimizations (β-reductions, constant folding, dead variable elimination) are enabled—in λ-calculus, one β-reduction may produce more redexes.

But the 1992-vintage CPS optimizer (which I will call "naive") had trouble with loops. Consider this example (already CPS-converted):

```
fun f(x,r,s,k) = if x=0 then k(s)
                         else f(x-1, r, s + #1(r), k)

fun outer(v,c) =
   let fun j(i) = c(i*2)
      val t = (v, v>2)
    in f(v,t,0,j)
   end

fun elsewhere =  . . .   f(. . .) . . .
```

What in-line expansions are useful here? The function calls in lines 1 and 4 are to "unknown" functions k and c (their function bodies are not trivially identifiable), so they cannot be expanded. Only the calls to f in lines 2 and 6 can be expanded. Expanding the former is equivalent to unrolling the loop a little bit; expanding the latter means putting the first iteration of the loop inside *outer*:

```
1    fun f(x,r,s,k) = if x=0 then k(s)
2                         else f(x-1, r, s + #1(r), k)

3    fun outer(v,c) =
4       let fun j(i) = c(i*2)
5          val t = (v, v>2)
6.1     in if v=0 then j(0)
6.2              else f(v-1,t,0+#1(t),j)
7       end

8    fun elsewhere =  . . .   f(. . .) . . .
```

Now the call in line 6.1 can be expanded, but on the whole this step was not very useful.

Loop headers

The new idea is to wrap a kind of "loop header" function around every recursive function. The header *contains* the loop function and calls it. Recursive calls go to the loop function; non-recursive calls (calls from outside) go to the header.

Loop header basic blocks have long been used [1], but in CPS or λ-calculus the notion of nested scope makes this transformation more powerful and useful where in-line expansion is concerned.

In the example,

```
1.1 fun fh(x',r',s',k') =
1.2  let fun fl(x,r,s,k) = if x=0 then k(s)
2.1                              else fl(x-1, r, s + #1(r), k)
2.2   in fl(x',r',s',k')
2.3  end

3    fun outer(v,c) =
4       let fun j(i) = c(i*2)
5          val t = (v, v>2)
6        in fh(v,t,0,j)
7       end

8    fun elsewhere =  . . .   fh(. . .) . . .
```

At the same time the loop header is installed, induction variable elimination can eliminate arguments that are just passed around the loop without change [4]. These variables become free variables of the loop, bound in the header. In this case, k and r are such variables:

```
1.1 fun fh(x',r',s',k') =
1.2  let fun fl(x,s) = if x=0 then k'(s)
2.1                               else fl(x-1, s + #1(r'))
2.2   in fl(x',s')
2.3  end

3   fun outer(v,c) =
4      let fun j(i) = c(i*2)
5         val t = (v, v>2)
6       in fh(v,t,0,j)
7      end

8   fun elsewhere =  . . .   fh(. . .) . . .
```

In-line expansion of loops

Now, the function call in line 6 can be in-line expanded. This is a "speculative" step, since the body of *fh* will be copied, possibly making the program bigger. It is to be hoped that further contractions will make up for this; the criteria for the expansion heuristic have been previously described [2, pp. 87-92].

After this expansion, we have:

```
1.1 fun fh(x',r',s',k') =
1.2  let fun fl(x,s) = if x=0 then k'(s)
2.1                           else fl(x-1, s + #1(r'))
2.2   in fl(x',s')
2.3  end

3   fun outer(v,c) =
4      let fun j(i) = c(i*2)
5         val t = (v, v>2)
6.1.1   in let fun fl(x,s) = if x=0 then j(s)
6.2.1                             else fl(x-1,s+#1(t))
6.2.2       in fl(v,0)
6.2.3      end
7      end

8   fun elsewhere =  . . .   fh(. . .) . . .
```

Now, `j(s)` in line 6.1.1 is calling a function with no other calls, so it may be β-reduced; `#1(t)` in line 6.2.1 may be contracted to `v`, and then `t` is dead so line 5 may be removed:

```
1.1 fun fh(x',r',s',k') =
1.2  let fun fl(x,s) = if x=0 then k'(s)
2.1                          else fl(x-1, s + #1(r'))
2.2   in fl(x',s')
2.3  end

3   fun outer(v,c) =
6.1.1      let fun fl(x,s) = if x=0 then c(i*2)
6.2.1                                else fl(x-1,s+v)
6.2.2       in fl(v,0)
6.2.3      end

8   fun elsewhere =  . . .   fh(. . .) . . .
```

The important things accomplished by this series of transformations are the reduction of `#1(t)` and `j(s)`. In general, suppose t had been a function and line 2 had contained something like $t(s)$, then this function call can now be in-lined when the "naive" in-line expander could not do so.

The question of whether to unroll the loop 1.2 or the loop 6.1.1 may be taken up separately by the expander. This is a nontrivial question, since the premature unrolling of line 1.2 will make the expansion of `fh` less attractive. In general this problem is not computable, but a useful heuristic suggested by Trevor Jim is to delay unrollings (expansion of recursive calls) until after other in-line expansions have quiesced.

Loop invariant arguments

Even without in-line expansion, hoisting invariant arguments out of loops is important—and it is only possible with a loop-header that can provide a binding site for these variables. The efficient callee-save closure representation of Shao and Appel [3] takes particular advantage of this. In this example,

```
fun exists(L,f,c) =
   if L = nil then c(false)
   else let fun k(x) = if x then c(true)
                       else exists(cdr L, f, c)
         in f (car L, k)
        end
```

the naive compiler would allocate a closure in memory for each instance of k, even though the free variables f and c are invariant and only L differs in

Benchmark	Compiler	Execution Time usr	gc	sys	real	Heap Allocation	Compile Time Ratio	Run Time Ratio
Barnes-Hut	Naive	25.49	2.25	0.51	28.32	353.2Mb		
	Headers	24.54	2.26	0.49	27.36	341.0Mb	1.003	0.966
Boyer	Naive	1.21	1.45	0.23	2.93	23.8Mb		
	Headers	1.13	1.38	0.25	2.77	23.1Mb	0.983	0.955
CML-sieve	Naive	16.11	18.63	0.76	35.56	177.0Mb		
	Headers	16.25	16.15	0.58	33.04	164.9Mb	0.914	0.929
Knuth-Bendix	Naive	6.82	1.20	0.27	8.32	141.1Mb		
	Headers	6.00	1.06	0.22	7.30	122.0Mb	0.802	0.878
Lex	Naive	9.85	0.80	0.25	11.12	81.9Mb		
	Headers	9.59	0.75	0.25	10.83	62.0Mb	0.997	0.972
Life	Naive	1.31	0.17	0.02	1.51	8.6Mb		
	Headers	1.29	0.15	0.03	1.49	7.0Mb	0.904	0.980
Yacc	Naive	2.92	1.06	0.31	4.67	43.9Mb		
	Headers	2.95	1.08	0.26	4.60	44.3Mb	0.988	1.000
Ray	Naive	24.68	0.38	1.02	26.80	408.7Mb		
	Headers	21.83	0.36	1.06	23.88	397.7Mb	0.933	0.891
Simple	Naive	15.71	0.65	0.35	16.75	225.8Mb		
	Headers	15.16	0.76	0.31	16.25	202.4Mb	1.100	0.971
VLIW	Naive	13.84	0.62	0.17	14.82	116.0Mb		
	Headers	12.63	0.79	0.22	13.81	113.2Mb	0.481	0.932
Average							0.911	0.947

Table 1: Benchmark performance

each iteration. (A stack-based compiler could stack-allocate these closures, but this still requires memory traffic in each iteration.) The "callee-save closure" algorithm can make the closure (containing f and the several callee-save registers c) in the loop header, and hold L in callee-save registers, so that each iteration can call f and return to k without *any* memory traffic.

Results

Table 1 shows the effect on execution time of this optimization on several benchmark programs, which are briefly described by Shao and Appel[3]. The loop-header transformation improves execution time by about 5% on the average, partly by reducing the amount of heap allocation.

Compilation time *improves* by 8% on the average, because the loop-header optimization reduces the amount of work for the back-end phases (closure conversion, instruction selection, register allocation, scheduling).

This optimization is implemented in SML/NJ versions 0.96 and after.

Conclusion

This technique relies critically on the nested scope of the *lambda*-calculus intermediate representation. It does not rely so much on continuation-passing; a direct-style version of this algorithm would also be effective.

With an intermediate representation sufficiently powerful to express functions with nested scope, the introduction of "loop header" functions makes in-line expansion of recursive functions much more useful.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, page (to appear). ACM Press, 1994.

[4] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991. CMU-CS-91-145.