Functional Inlining

Course Report

of

CS6113 Function Programming

Instructor: Dr. Owen Kaser

By

Chengyan Zhao

Id: 255 459

Due: July 10, 1998

1. Introduction

As an important compiler optimization technique, inlining is not only used in traditional languages, but also widely applied on functional languages [1,2,3,4,7,8,10]. Similar as the one in compiling C language [12], inlining optimization could also be done on both source-level and intermediate level of compiling functional languages.

In the following report, we give more description on source-level functional language inlining and comparison with the similar work done on the C inliner.

2. Source-level functional inlining

2.1 Introduction

It might be strange to think about inlining at the source level of functional languages, because most of the functional language compilers compile the functional source code to a non-functional intermediate representation before optimizations could be applied. However, it is still possible and practical to perform inlining at the source level of functional languages.

2.2. Source-level inlining

An example to perform source-level functional inlining is given in figure 2.1, which is written in OPAL [1,2] source code.

```
DEF (x: xs) ++ ys = x :: (xs++ys)
DEF [ ] ++ ys   = ys
```

Fig 2.1(a) definition of function ++

```
DEF xs :: ys ==
  IF <>? (xs) THEN ys
  IF ::?(xs) THEN head(xs) :: (tail(xs) ++ ys)
```

Fig 2.1(b) definition of function ::

```
DEF <>? (s) ==
    CASE s OF
      [ ] = true
    | xh :: xl = false
```

Fig 2.1(c) definition for function <>?

```
DEF ::? (s) ==
    CASE s OF
      [ ] = false
    | xh :: xl = true
```

Fig 2.1(d) definition for function ::?

```
DEF head(s) ==
  CASE s OF
    [ ] = ERROR
  | x :: xs = x
```

Fig 2.1(e) definition for function head

```
DEF tail(s) ==
  CASE s OF
    [ ] = ERROR
  | x :: xs = xs
```

Fig 2.1(f) definition for function tail

In Figure 2.1(a), the definition of function "++" (list concatenation) is given. In Figure 2.1(b), we show the definition of function ::. In this definition, we found two other function callsites -

- <>? and ::?, which definitions are given in Figure 2.1(c) and Figure 2.1(d) respectively. Figure 2.1(e) shows the definition to get the head of a list and Figure 2.1(f) shows the function to get the detail of a list.

Both callsites of <>? and ::? in Figure 2.1(b) have been selected for inlining and the inlined code is shown in Figure 2.2(a) and Figure 2.2(b).

```
DEF xs :: ys ==
      CASE xs OF /* inlined callsite <>? */
      [] = ys
 IF ::?(xs) THEN head(xs) :: tail(xs ++ ys)
```

Fig 2.2(a) Inlining callsite <>?

```
DEF xs :: ys ==
      CASE xs OF
       [] = ys
      CASE xs OF   /* inlined callsite ::? */
       x1 :: x2 = head(x1) :: tail(x2 ++ ys)
```

Fig 2.2(b) Inlining callsite ::?

Actually, the OPAL compiler could do a little bit more program restructuring on the code shown in Figure 2.2(b) – combine the separated CASE constructs. The final result is given in Figure 2.2(c).

```
DEF xs :: ys ==
      CASE xs OF  /* inlined callsite <>? */
       [] = ys
     |  /* inlined callsite ::? */
       x1 :: x2 = head(x1) :: tail(x2 ++ ys)
```

Fig 2.2(c) Inlining after combination

2.3 Comparison

Comparing with source-level C inlining, there are great differences in the source-level functional inlining.

• No standardization

In C, every callsite must be in its standardized format before inlining techniques could be applied on and many efforts are dedicated to the standardization processing, including subprocesses of swapping and splitting.

In functional languages, the callsites are more likely to appear in conditional evaluations (if or case) and the callsites do not have to be in the standardized format before inlining. The efforts made on callsite standardization are eliminated.

• Callsite Inlining

In C, the callsite inlining includes relatively large amount of work, such as: body duplication, parameter passing simulation, return statement simulation, conflicted variable renaming, original callsite removal and the duplicated function body replacement.

In source-level functional inlining, the steps are quite similar, but much simpler.

First, the callee's function definition body is explicitly duplicated and takes the place of the original callsite.

Second, the parameter passing is simulated, including parameter renaming and renamed parameter propagation in the duplicated function body.

Because of the prime features of functional programming language, there are no return statement, no labels, no complicated data structures and control constructs. The tiresome works, such as: array type parameter passing simulation, return statement simulation, duplicated body relabeling, etc are all eliminated.

• Decision Algorithm

In our C inliner, we used a rather ambitious decision algorithm, which considers version issue, cache issue, constant-propagation opportunities, hazardous opportunities and aims to get the best performance improvement after inlining.

In the tested functional languages, there are no explicitly and dedicated decision-making algorithms that combine different techniques and expect best performance [2,3,6]. Instead, some simple ideas are chosen. For example, the OPAL [1,2] compiler uses counters. Calls to CASE and primitive constructor functions count zero and calls to other functions count one. The compiler only inlines the callsites that count between 2 and 4. For another example, the FISh [11] compiler explicitly inlines all non-recursive callsites.

3. Intermediate-level inlining

Depends on specific compiler's implementation, intermediate language representation could categorize into three directions.

3.1 Using a functional language

Some functional compilers, such as Clean [9], will compile the function language source code into another well-structured function language [1] that has better features to perform optimizations. This intermediate level inlining falls into the category of source-level functional inlining and details have been discussed in Section 2.

3.2 Using a well-structured traditional language

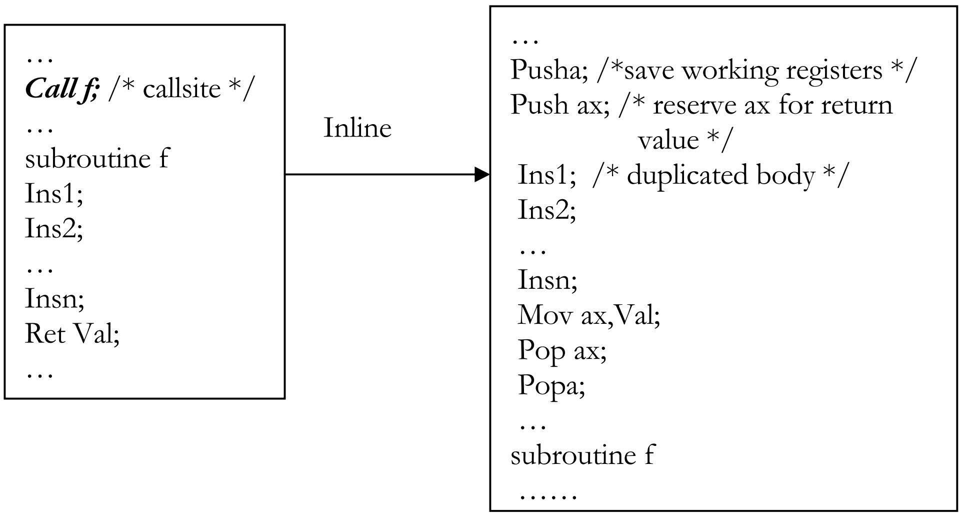
[1] Clean compiler compiles Clean source code to Miranda, which is a better-structured functional language.

Some functional compilers, such as Equals [13], prefer to use a well-structured traditional language[2] to be its intermediate level representation, because there are powerful tools from different researchers to perform optimizations on the selected intermediate language[3].

3.3 Using non-standardized intermediate representation

Most of the other functional compilers [2,4,8,10] use their own intermediate language representations. It could be tuples, p-code or assembler code that people could find standardized specifications. As well, it could be any kind of non-standardized and un-published intermediate language that is only available within the research group. In this case, the compiler writers also need to give specifications on their intermediate level inlining.

On the other hand, intermediate level inlining seems to be easier if the intermediate representation is on low level and well structured. An example of assembler inlining is given in Figure 3.1.



```
…
Call f; /* callsite */
…
subroutine f
Ins1;
Ins2;
…
Insn;
Ret Val;
…
```

```
…
Pusha; /*save working registers */
Push ax; /* reserve ax for return
                value */
 Ins1;  /* duplicated body */
 Ins2;
 …
 Insn;
 Mov ax,Val;
 Pop ax;
 Popa;
 …
subroutine f
 ……
```

Fig 3.1 Assembler-level inlining

[2] Equals compiler compiles Equals source code to gcc, which is an extension of ANSI C with some special features supported by gcc compiler.

In this example, we use assembler as the intermediate language. After inlining, the original callsite is replaced with the duplicated function body of callee f, as shown on the right-hand side of Figure 3.1. First, all the working registers are saved onto stack and a specific register (in this case, we use general-purpose register AX) is reserved for result returning before the control is passed into the duplicated function body. Second, the duplicated instruction sequence from the callee function definition replaces the original callsite. Finally, the return value is popped on the top of stack and working registers need to be restored. Minus modifications might also necessary to adjust registers such as stack pointer and base pointer.

Surprisingly enough, we find that most of the headaches from the source-level inlining are eliminated in this level, such as: callsite standardization, parameter passing simulation, conflicted variable renaming, return simulation, relabeling, etc. In assembler language representation, all callsites have already been in its standardized format, it is not necessarily to standardize them. All parameters are passed on stack through registers and labels appear to be relative addresses, neither do we need to perform parameter passing simulation nor renaming and relabeling. Although there are some adjustments on working-set register protection and return register reservation, the work is relatively small and could be ignored comparing with the efforts we concentrate on the source-level C inlining.

4. Performance Improvement

[3] Detailed discussion on C source-level inlining could be found in [12].

In C inlining, the performance generally increases from 4% to 43%, depending on the property of the testing programs and the range of code expansion. In functional programs, this comes from 3% to 17% [3,4], which is still very impressive.

5. Conclusion

5.1 Functional inlining is easier

Comparing with the efforts made on source-level C inlining, source-level functional language inlining seems to be easier, because functional language definition is more concise and the control structures are far less complicated. People do not need to worry too much about syntax issues.

5.2 Comparable performance increment

Comparing with source-level C inlining, the inlinings performed on source-level functional languages also achieves a relatively high performance increment after the inlining optimization, which is also attractive.

5.3 Implementation dependent

Since the inlining is a kind of background optimization technique, it is up to the compiler writers' responsibility to decide what kind of inlining he likes better – source-level inlining on function languages, source-level inlining on traditional well-structured languages or object-level inlining on assembler languages or even at the linking time. Since the compiler writers never need to read the inliner generated code if the inliner is bug free, it does not really mater in what language and on which level the inlining will actually perform on.

Reference

[1] Robert Büssow, Wolfgang Grieskamp, Winfried Heicking and Stephan Herrmann: An Open Environment for the Integration of Heterogeneous Modelling Techniques and Tools. Submitted to Current Trends in Applied Formal Methods, 1998.

[2] Robert Büssow, Robert Geisler, Wolfgang Grieskamp, Marcus Klar: The mSZ Notation Version 1.0 Technical Report, TU Berlin, FB 13, No. 97-26.

[3] SL Peyton Jones and A Santos, A transformation-based optimiser for Haskell, to appear in Science of Computer Programming, 1998.

[4] SL Peyton Jones, Compiling Haskell by program transformation: a report from the trenches, Proc European Symposium on Programming (ESOP'96), Linkping, Sweden, Springer Verlag LNCS 1058, Jan 1996.

[5] David N. Turner, Philip Wadler and Christian Mossin, Once Upon A Type, pages 1-11, ACM95: Journal of Lisp and Functional Programming

[6] Santos, Andr Lus de Mederios, Compilation by Transformation in Non-Strict Functional Languages, Ph.D. thesis, Department of Computing Science, University of Glasgow , July 1995. Available as Technical Report TR-1995-17.

[7] Jörg Würtz, Oz Scheduler: A Workbench for Scheduling Problems, Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence, Nov16--19 1996, IEEE Computer Society Press.

[8] Thomas Conway, Fergus Henderson, and Zoltan Somogyi, Code generation for Mercury, Proceedings of the 1995 International Symposium on Logic Programming, Portland, Oregon, December 1995, pages 242-256.

[9] Plasmeijer, M.J., Eekelen, M.C.J.D. v, Nöcker, E., Smetsers, J.E.W. (1991), Concurrent CLEAN - Status report, In Proc. of Functional languages: Optimization for parallelism, Schloss Dagstuhl, Saarbrucken, Wilhelm Ed., Schloss Dagstuhl, Saarbrucken, pp. 22-23.

[10] Joe Armstrong, The development of Erlang, In Proceedings of the ACM SIGPLAN International Conference on Functional Programming, pages 196 - 203, ACM Press, 1997.

[11] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. Lisp and Symbolic Computation: An International Journal, 1(2): 147-164, September 1988.

[12] Chengyan Zhao, A Multi-Technique C Inliner, MCS. Thesis, Faculty of Computer Science, University of New Brunswick, July 1998.

[13] Owen Kaser, C.R. Ramakrishnan, I.V. Ramakrishnan, R.C. Sekar, EQUALS – A Fast Parallel Implementation of a Lazy Language, Journal of Functional Programming, 1997.