Partial Inlining Using Local Graph Rewriting [*]

Irène Durand
Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr

ABSTRACT

Inlining is an important optimization technique in any modern compiler, though the description of this technique in the literature is informal and vague. We describe a technique for inlining, designed to work on a *flow graph* of instructions of intermediate code.

Our technique uses *local graph rewriting*, making the semantic correctness of this technique obvious. In addition, we prove that the algorithm terminates.

As a direct result of the preservation of the semantics of the program after each local rewriting step, the algorithm can stop after any iteration, resulting in a *partial inlining* of the called function. Such partial inlining can be advantageous in order to avoid the inlining of code that is not performance critical, in particular for creating arguments and calls to error-signaling functions.

CCS Concepts

•Software and its engineering → Abstraction, modeling and modularity; Software performance; Compilers;

Keywords

Common Lisp, Compiler optimization, Portability, Maintainability, Graph rewriting

1. INTRODUCTION

Inlining represents an important optimization technique in any modern compiler. It avoids the overhead of a full function call, and it allows further optimization in the calling function in the form of type inference, loop optimizations, and more.

While the advantages of inlining are well known and well documented, inlining also entails some disadvantages. It increases the size of the code, with a possible negative impact on processor cache performance. It also increases pressure on register allocation, possibly making it necessary to spill registers to the stack more often. Most importantly, though, as Ayers et al. point out [1, 2], since many optimization algorithms do not have linear-time complexity in the size of the code, inlining can have a serious impact on the execution time of the compiler.

Some authors distinguish between *procedure integration* and *inline expansion* [7]. Both techniques are often referred to with the abbreviated form *inlining*. Our use of *inlining* corresponds to *procedure integration*.

Most literature sources define inlining as "replacing a call to a function with a copy of the body of the called function" (see e.g., [3, 4, 8]). This definition suggests that inlining is an all-or-nothing transformation. In this paper, we present a technique that allows for *partial* inlining. More precisely, it allows for a *prefix* of the callee to be copied into the caller. We obtain this property by using *local graph rewriting* at the level of instructions in intermediate code. A single instruction is inlined in each step, preserving the overall semantics of the program, and thereby allowing us to stop the process at any time.

The traditional definition of inlining is too vague for our purpose. It suggests that the sole purpose of inlining is to avoid overhead in the function-call protocol. However, on modern processors, this overhead is insignificant. For the purpose of this paper, we would also like to avoid the creation of a local *environment* that would normally be necessary for each invocation of the callee. This additional requirement poses additional restrictions as to when inlining is appropriate.

In this paper, we discuss only the inlining technique itself. We do not consider the policy to determine when it is advantageous to perform the technique, and, although our technique allows for partial inlining, we also do not consider the policy of when inlining should stop.

2. PREVIOUS WORK

Before inlining was applied to so-called "structured programming languages", the technique was applied to languages such as Fortran, that do not allow recursion, and therefore do not need for subroutines to allocate their own environments upon entry. And it was then referred to as "open-coding of subroutines". Scheifler [8] is probably one of the first to apply inlining to more modern programming languages. The language used by Scheifler is CLU [6].

Ayers et al [1] consider the benefit of inlining consisting

[*]This work was supported by the French National Research Agency (ANR project GraphEn / ANR-15-CE40-0009).

11th ELS April 16–17, 2018, Marbella, Spain

of the elimination of the overhead of a procedure call to be a "side benefit", and we agree. They cite the main benefit as the opportunity for more optimizing code transformations when the code of the called function is exposed in the context of the calling function.

In their paper, they also mention *cloning* as an alternative to inlining, i.e., the duplication and specialization of the called function according to the context of the calling function. However, they consider inlining to be strictly superior to cloning in terms of the possible additional optimizations made possible, so they recommend cloning only as a means to avoid too large an increase in the code size, which could slow down subsequent non-linear optimizations. Cloning, and especially the specialization of the cloned code in the context of the caller, is one technique used in *partial evaluation* [5]. Inlining, however, whether total or partial, is not a technique of partial evaluation. Inlining may of course enable such techniques by exposing the code of the called function in the context of the caller.

Most existing work is concerned with determining when inlining is to be performed, based on some analysis of the benefits as compared to the penalties in terms of increased compilation time in subsequent optimization passes. The inlining technique itself is considered trivial, or in the words of Chang and Hwu ([4, 3]) "The work required to duplicate the callee is trivial". Inlining might be trivial in the context of purely function programming, in that it suffices to replace occurrences of local variables in the called function by the argument expressions in the function call. However, for a language such as Common Lisp that allows for assignments to lexical variables, inlining can be non-trivial. Consider the following example:

```
(defun f (x y) (setq x y))

(defun g (a) (f a 3) a)
```

If simple renaming is applied, we obtain the following code which does not preserve the semantics of the original code:

```
(defun g (a) (setq a 3) a)
```

The use of continuation-passing style for compiling Common Lisp programs often requires a priori elimination of side effects by confiding these side effects to updates on *cells*. Such a conversion transforms the program so that it respects a purely functional style, making inlining trivial as indicated above. However, such a conversion has a significant impact on program performance, especially in the context of modern processors, where memory access are orders of magnitude more expensive than register operations.

Because of issues such as this one, this paper discusses only a technique for inlining in the context of arbitrary Common Lisp code that might contain such side effects. It does not discuss the more complex issue of determining a strategy for when inlining should or should not be applied.

Although the paper by Ayers et al explains that their technique is applied to intermediate code, just like the technique that we present in this paper, their paper contains little information about the details of their technique.

3. OUR TECHNIQUE

The work described in this paper is part of the Cleavir compiler framework. Cleavir is currently part of the SICL project[1], but we may turn it into an independent project in the future.

In our compiler, source code is first converted to an *abstract syntax tree*. In such a tree, lexical variables and lexical function names have been converted to unique objects. When a globally defined function F is inlined into another function G, we incorporate the abstract syntax tree of F as if it were a local function in G. No alpha renaming is required. Notice that this step in itself does not count as inlining. The function F is still invoked using the normal function-call protocol at this stage.

In the second phase, the abstract syntax tree is translated to intermediate code in the form of a flow graph of instructions. Our inlining technique is designed to work on this intermediate representation.

There are several advantages of using this intermediate representation over higher-level ones such as source code or abstract syntax trees, as we will show in greater detail below, namely:

- Each iteration of the algorithm defined by our technique is very simple, and we can be shown to preserve the semantics of the program.
- Because each iteration preserves the semantics, the process can be interrupted at any point in time, resulting in a *partial* inlining of the called function.

Furthermore, this intermediate code representation is similar to the one used in many traditional compiler optimization techniques, making it possible to reuse code for similar transformations.

One potential drawback of this representation is that operations on programs represented this way are inherently imperative, i.e. they modify the structure of the flow graph. The use of techniques from functional programming is therefore difficult or impractical with this representation. Moreover, the flow graph resulting from some arbitrary number of iterations of our technique does not necessarily have any correspondence as Common Lisp source code.

3.1 Intermediate code

The intermediate code on which our technique is designed to work is called *High-level Intermediate Representation*, or *HIR* for short. This representation takes the form of a *flow graph* of *instructions* as used by many traditional compiler optimization techniques. The main difference between HIR and the intermediate representation used in compilers for lower-level languages is that in HIR, the only data objects that the instructions manipulate are Common Lisp objects. Arbitrary computations on addresses are exposed in a later stage called *Medium-level Intermediate Representation*, or *MIR*.

Most HIR instructions correspond directly to Common Lisp operators such as the ones in the categories described below. Notice that, although the names of the instructions often resemble the names of Common Lisp operators, the instruction typically requires more precise objects than the corresponding Common Lisp operator does. Thus, the `car` instruction requires the argument to be a `cons` object, and the `funcall` instruction requires its first argument to be a function. The following such categories exist:

[1]https://github.com/robert-strandh/SICL

- Low-level accessors such as `car`, `cdr`, `rplaca`, `rplacd`, `aref`, `aset`, `slot-read`, and `slot-write`.
- Instructions for low-level arithmetic on, and comparison of, floating-point numbers and fixnums.
- Instructions for testing the type of an object.
- Instructions such as `funcall`, `return`, and `unwind` for handling function calls and returns.

Two of the HIR instructions are special in that they do not have direct corresponding Common Lisp operators, and in that they are essential to the inlining machinery described in this paper:

- The `enter` instruction. This instruction is the first one to be executed in a function, and it is responsible for creating the initial local lexical environment of the function from the arguments given by the calling function. This initial environment is typically augmented by temporary lexical variables during the execution of the function. Variables may also be eliminated from the local environment when they are no longer accessible by any execution path.
- The `enclose` instruction. This instruction takes the *code* of a nested function (represented by its `enter` instruction) and creates a *callable function* that may be a *closure*.

3.2 Algorithm

The algorithm that implements our technique maintains a *worklist*. An item[2] of the worklist contains:

- A `funcall` instruction, representing the call site in the calling function.
- An `enter` instruction, representing the called function.
- The successor instruction of the `enter` instruction, called the *target instruction*, or *target* for short. The target instruction is the one that is a candidate for inlining, and it is used for generic dispatch.
- A mapping from lexical variables in the called function that have already been duplicated in the calling function.

In addition to the contents of the worklist items, our algorithm maintains the following global information, independent of any worklist item:

- A mapping from instructions in the called function that have already been inlined, to the corresponding instructions in the calling function. This information prevents an instruction from being inlined more than once. Without this information, and in the presence of loops in the called function, our inlining algorithm would go into an infinite computation.
- Information about the ownership of lexical variables referred to by the called function. This ownership information indicates whether a lexical variable is created by the called function itself, or by some enclosing function. When an instruction to be inlined refers to a variable that is created by some enclosing function, the reference is maintained without modification. When the reference is to a variable created by the function itself, the inlined instruction must refer to the corresponding variable in the calling function instead.

Prior to algorithm execution, assignment instructions are inserted before the `funcall` instruction, copying each argument to a temporary lexical variable. These lexical variables represent a copy of the initial environment of the called function, but allocated in the calling function. The pair consisting of the `funcall` and the `enter` instruction can be seen as transferring this environment from the calling function to the called function. The variable correspondences form the initial lexical variable mapping to be used in the algorithm.

Initially, the worklist contains a single worklist item with the following contents:

- The `funcall` instruction representing the call that should be inlined.
- A *private copy* of the initial `enter` instruction of the function to inline.
- The successor instruction of the initial `enter` instruction, which is the initial target.
- The initial lexical variable mapping described previously.

In each iteration of the algorithm, a worklist item is removed from the worklist, and a generic function is called with four arguments, representing the contents of the worklist item. Each iteration may result in zero, one, or two new worklist items, according to the mappings and ownership information, and according to the number of successors of the target instruction in this contents.

When the generic function is called in each iteration, one of the following four rules applies. As we show in Section 3.4, each of the following rules preserves the overall operational semantics of the code:

1. If the target instruction has already been inlined, i.e. it is in the mapping containing this information as described previously, then replace the `funcall` instruction by the inlined version of the target. There are two ways of doing this replacement. Either the predecessors of the `funcall` instruction are redirected to the inlined version of the target instruction, effectively making the `funcall` instruction unreachable, or else, the funcall instruction is replaced by a `no-operation` instruction with the inlined version of the target instruction as its successor. When this rule applies, no new item is added to the worklist.
2. If the target instruction is a `return` instruction, then replace the `funcall` instruction by one or more assignment instructions mapping inputs of the `funcall` instruction to outputs of that same instruction. Again, in this case, no new item is added to the worklist.

[2]In the code, an item also contains an `enclose` instruction, but we omit this instruction from our description, in order to simplify it.

3. If the target instruction has a single successor, insert a copy of the next instruction before the `funcall` instruction, and make the `enter` instruction refer to that successor. Update the mappings, the inputs of the `funcall` instruction, and the outputs of the `enter` instruction as described below. In this case, the `funcall` instruction, the `enter` instruction, the new successor of the `enter` instruction, and the updated lexical variable mapping are inserted as a new item on the worklist for later processing.

4. If the target instruction has two successors, insert a copy of the target instruction before the `funcall` instruction, and replicate the `funcall` instruction in each branch. Also replicate the `enter` instruction so that each replica refers to a different successor of the original instruction. Update the mappings, the inputs of the `funcall` instruction, and the outputs of the `enter` instruction as described below. In this case, two new items are inserted on the worklist for later processing. Each item contains a `funcall` instruction, an `enter` instruction, the successor of the `enter` instruction, and a lexical variable mapping, corresponding to each successor branch of the inlined instruction.

For rules 3 and 4, when a new instruction is inlined, the mappings, the inputs to the `funcall` instruction, and the outputs of the `enter` instruction are updated as follows:

- An entry is created in the mapping from instructions in the called function to instructions in the calling function, containing the inlined instruction and its copy in the calling function.

- If some input `i` to the inlined instruction is present in the lexical variable mapping (mapping to (say) `ii` in the calling function) and in the outputs of the `enter` instruction, but `i` is no longer live after the inlined instruction, then the entry `ii - i` is eliminated from the mapping, `i` is eliminated from the outputs of the `enter` instruction, and `ii` is eliminated from the inputs to the `funcall` instruction. It would be semantically harmless to leave it intact, but it might harm performance if the inlining procedure is stopped when it is still partial. Notice that, when an instruction with two successors is inlined, variable liveness may be different in the two successor branches.

- If some output `o` of the inlined instruction is a new variable that is created by that instruction, then we proceed as follows. Let `I` be the instruction in the called function that has been inlined, and let `II` be the copy of `I` in the calling function. We create a new variable `oo` in the calling function that takes the place of `o` in `II`. We add `oo` as an input to the `funcall` instruction, `o` as an output of the `enter` instruction, and we add `oo - o` to the lexical variable mapping. Again, if the inlined instruction has two successors, the lexical variable mapping may have to be updated for one or the other or both of the successors.

3.3 Example

As an example of our technique, consider the initial instruction graph in Figure 1. On the left is the calling function. It has three lexical variables, namely `x`, `a`, and `y`. The variable `a` is referenced by the called function, but it is owned by the calling function. The called function has a single variable named `z` in its initial lexical environment. A temporary variable `w` is created as a result of the execution of one of the instructions in the called function.

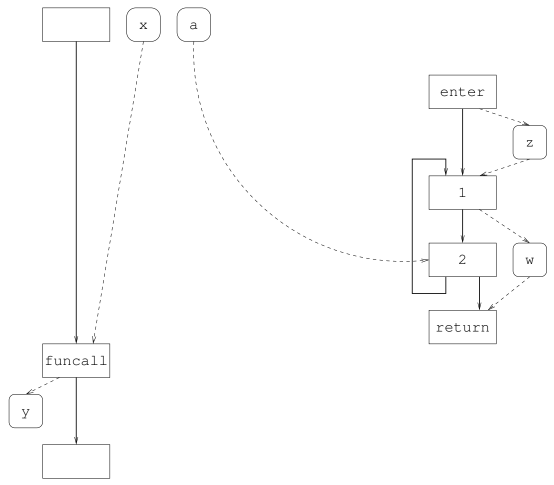

Figure 1: Initial instruction graph.

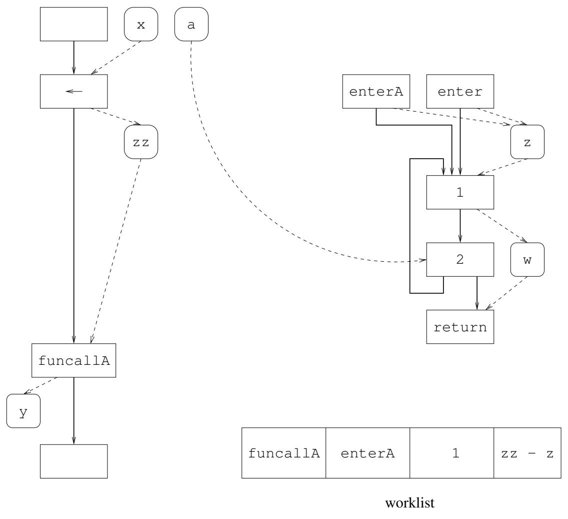

Figure 2: Instruction graph after initialization.

Before the inlining procedure is started, we create temporary variables in the calling function for the variables in the initial environment of the called function. We also create a private copy of the `enter` instruction so that we can mutate it during the inlining procedure. The result is shown in Figure 2.

As we can see in Figure 2, an assignment instruction has been created that copies the value of the lexical variable `x` into a variable `zz` that mirrors the initial lexical variable `z` in the called function. We also see that there are now two identical `enter` instructions. The one labeled `enterA` is the private copy.

Step one of the inlining procedure consists of inlining the successor of our private `enter` instruction, i.e. the instruc-

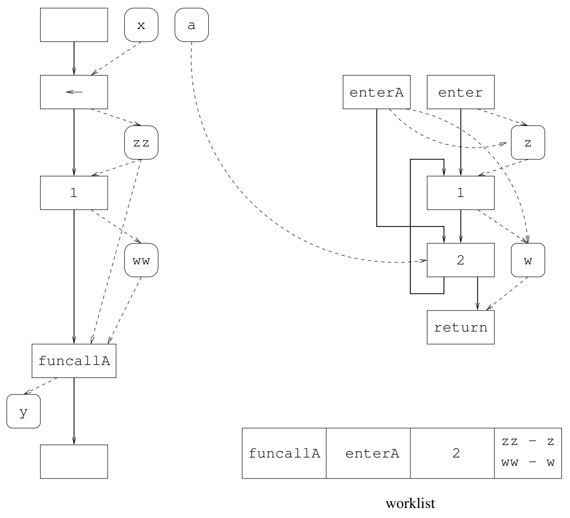

Figure 3: Instruction graph after one inlining step.

tion labeled `1` in Figure 2. That instruction has a single successor, and it has not yet been inlined. Therefore, rule 3 applies, so we insert a copy of that instruction before the `funcall` instruction. Furthermore, since the input to the original instruction is the lexical variable `z`, and that variable is mapped to `zz` in the calling function, the inlined instruction receives `zz` as its input. The output of the original instruction is the temporary variable `w` that is not in our lexical variable mapping. Therefore, a temporary variable `ww` is created in the calling function, and an entry is created in the mapping that translates `w` to `ww`. The private `enter` instruction (labeled `enterA`) is modified so that it now refers to the next instruction to be considered as a target. The result of this step is shown in Figure 3.

In step two of the inlining procedure, we are considering inlining an instruction with two successors, i.e. the one labeled `2` in Figure 3. It has not yet been inlined, so rule number 4 applies. As rule number 4 stipulates, we must replicate both the `enter` instruction and the `funcall` instruction. The result is shown in Figure 4.

In Figure 4, the `funcall` instruction labeled `funcallA` is paired with the `enter` instruction labeled `enterA` and the `funcall` instruction labeled `funcallB` is paired with the `enter` instruction labeled `enterB`.

In step three of the inlining procedure, we consider the `funcall` instruction labeled `funcallB`. The corresponding `enter` instruction has a `return` instruction as its successor, so rule number 2 applies. We must therefore replace the `funcall` instruction by an assignment instruction, assigning the value of the variable `ww` to the variable `y`. The result of this operation is shown in Figure 5.

In step four of the inlining procedure, we consider the `funcall` instruction labeled `funcallA` in Figure 5 and the corresponding `enter` instruction. The successor of the `enter` instruction is the instruction labeled `1`, and that instruction has already been inlined, so rule number 1 applies. We therefore remove the `funcall` and redirect its predecessors to the inlined version of the instruction labeled `1`. The result is shown in Figure 6, and that completes the inlining procedure.

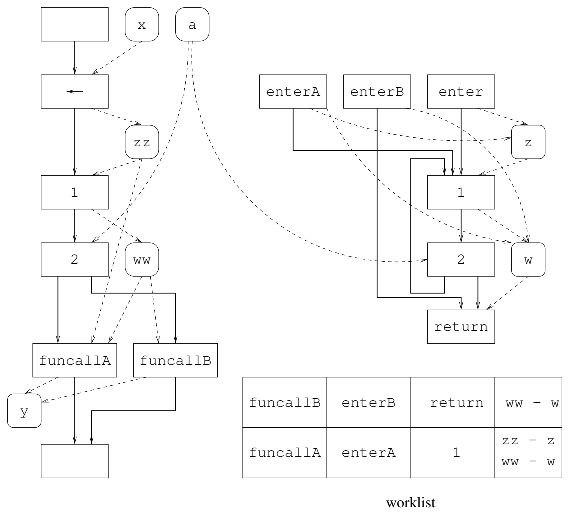

Figure 4: Instruction graph after two inlining steps.

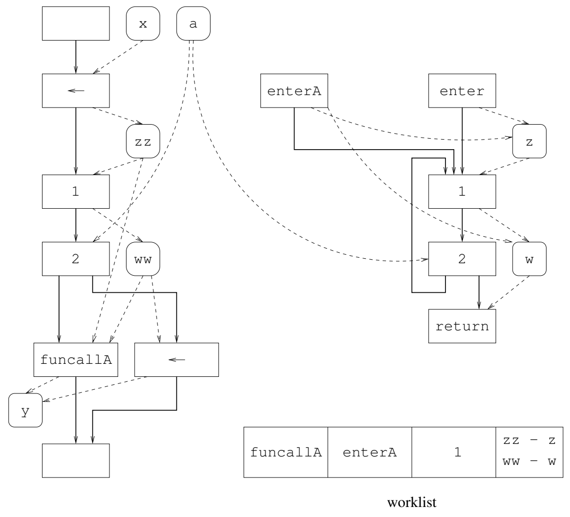

Figure 5: Instruction graph after three inlining steps.

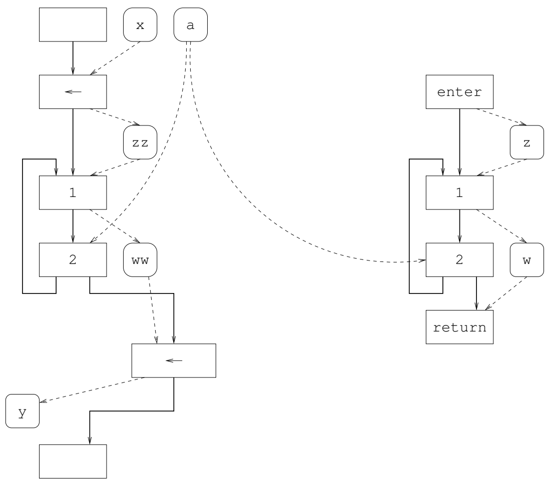

Figure 6: Instruction graph after four inlining steps.

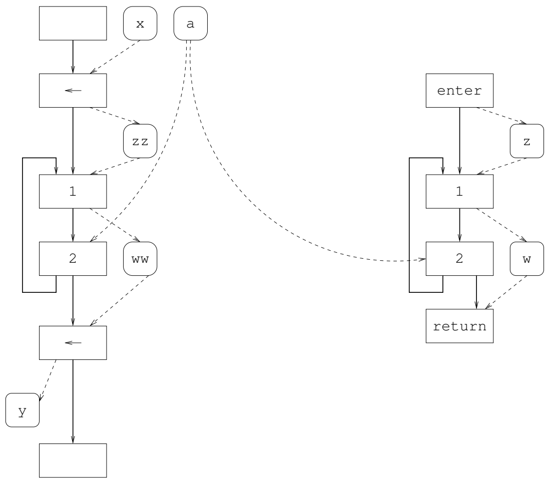

Figure 7: Final instruction graph.

After some minor reorganization of the instructions in Figure 6, we obtain the final result shown in Figure 7. Clearly we have an inlined version of the called function now replicated in the calling function.

3.4 Correctness of our technique

In order to prove total correctness of our technique, we must show that two conditions hold:

1. Partial correctness, i.e. the technique must preserve the semantics of the program.
2. Termination.

3.4.1 Partial correctness

Our technique preserves a very strong version of the semantics of the program, namely the *operational* semantics. This fact makes it unnecessary to create a precise definition of the program semantics, as might have been the case for some weaker type of semantics. Instead, we only need to show that the exact same operations are performed before and after each inlining step.

After a copy of the initial environment of the called function has been made in the environment of the calling function, we can see a pair of `funcall/enter` instructions as defining a morphism σ, mapping the copy of this environment in the calling function to its original version in the called function. The inputs of the `funcall` instruction are mapped to the outputs of the `enter` instruction. The lexical variable mapping used in our technique is simply the inverse, i.e σ^{-1} of this morphism. Similarly, a pair of `return/funcall` instructions can be seen as defining a morphism τ, mapping the environment in the called function to the environment in the calling function. The inputs of the `return` instruction are mapped to the outputs of the `funcall` instruction. These morphisms are illustrated in an example of an initial situation in Figure 8.

Applying rule 3 or rule 4 copies one instruction from the called function to the calling function, applying the morphism σ^{-1} to its inputs and outputs. Two applications of rule 3 from the initial situation are illustrated in Figure 9 and Figure 10. Applying rule 4 is a bit more involved, but the same mechanism is used. As we can see from these figures, thanks to the morphism, the instructions operate the same way whether inlined or not. The semantics are thus the same in both cases.

When rule 2 is applied, the `return` instruction is not copied. Instead, a number of assignment instructions are created in the calling function. Together, these assignment instructions define the composition of the two morphisms τ and σ, i.e. $\tau \circ \sigma$. Applying this rule therefore does not alter the semantics of the program. It merely maps the returned values to their copies in the calling function. Applying this rule is illustrated in Figure 11.

Finally, applying rule 1 merely avoids the control transfer from the calling function to the called function, by replacing the `funcall` instruction by an existing copy of the instruction that would have been inlined by rule 3 or rule 4. The existing copy obviously already operates in the environment of the calling function.

3.4.2 Termination

In order to prove termination, we invent a metric with the following properties:

- It has a lower bound on its value.
- Its value decreases with each iteration of our inlining procedure.

The metric we have chosen for this purpose is called *remaining work*, and it is represented as a pair $r = (I, F)$ where I is the number of instructions that have yet to be inlined, and F is the number of `funcall` instructions that have yet to be processed as part of the worklist items. Clearly, it has a lower bound on its value, namely $r_{min} = (0, 0)$.

Initially, the remaining work has the value $r_0 = (N, 1)$ where N is the number of instructions in the called function. We consider the metric to be lexicographically ordered by its components, i.e. $(I_1, F_1) < (I_2, F_2)$ if and only if either $I_1 < I_2$ or $I_1 = I_2$ and $F_1 < F_2$. We show that each step yields a value that is strictly smaller than before the step.

Consider some iteration k of our inlining procedure, so that $r_k = (I_k, F_k)$ is the remaining work before the iteration, and $r_{k+1} = (I_{k+1}, F_{k+1})$ is the remaining work after the iteration.

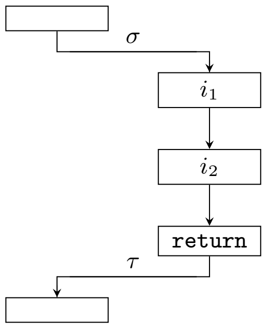

Figure 8: Initial situation.

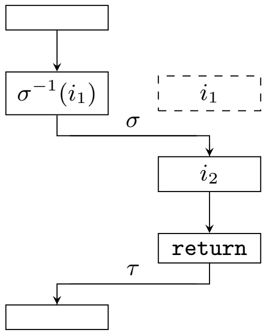

Figure 9: Situation after one application of rule 3.

- If rule number 1 applies, then one `funcall` instruction is eliminated in the iteration, so that $I_{k+1} = I_k$ and $F_{k+1} = F_k - 1$. Clearly, $r_{k+1} < r_k$ in this case.
- If rule number 2 applies, then again one `funcall` instruction is eliminated in the iteration, so that $I_{k+1} = I_k$ and $F_{k+1} = F_k - 1$. Again, $r_{k+1} < r_k$.
- If rule number 3 applies, then another instruction is inlined, but the number of `funcall` instructions remains the same, so that $I_{k+1} = I_k - 1$ and $F_{k+1} = F_k$. Again, $r_{k+1} < r_k$.
- Finally, if rule number 4 applies, then another instruction is inlined, but the number of `funcall` instructions increases by 1, so that $I_{k+1} = I_k - 1$ and $F_{k+1} = F_k + 1$. Again, $r_{k+1} < r_k$.

4. CONCLUSIONS AND FUTURE WORK

We have presented a technique for inlining local functions that uses local graph rewriting techniques. We have proved our technique to be correct in that it preserves the semantics of the original program, and it is guaranteed to terminate.

Although our iterative technique can be stopped at any point, thus giving us *partial inlining*, there are some practical aspects of such partial inlining that still need to be investigated:

- When the inlining is not complete, the called function has multiple entry points. Many optimization techniques described in the literature assume that a function has a single entry point. We plan to investigate the consequences of such multiple entry points on the optimization techniques that we have already implemented, as well as on any optimization techniques that we plan to incorporate in the future.
- In our intermediate code, we treat multiple values with an unknown number of values as a special type of datum. It is special in that it must store an arbitrary number (unknown at compile time) of values. During the execution of our inlining procedure, such a datum may become part of the mapping between variables of the called function and the calling function. When the inlining procedure continues until termination, such a datum will be handled in the calling function in the same way that it is handled in the called function. However, if the inlining procedure is stopped with such a datum in the mapping, we would somehow need to transmit it as an argument to the called function. Doing so may require costly allocation of temporary memory and costly tests for the number of values that would not be required when the procedure continues until termination. However, it is rare that code needs to store intermediate multiple values. It only happens in a few cases such as when `multiple-value-prog1` is used. Therefore, one solution to this problem is to avoid inlining in this case. Another possible solution is to let the inlining procedure continue until termination for these cases.

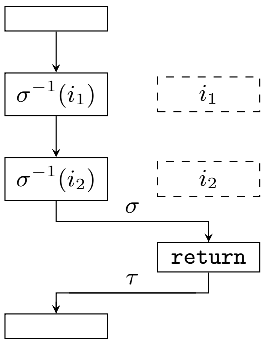

Figure 10: Situation after two applications of rule 3.

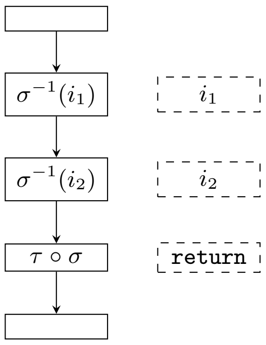

Figure 11: Situation after an applications of rule 2.

As presented in this paper, our technique handles only functions with very simple lambda lists. It is probably not worth the effort to attempt to inline functions with lambda lists containing keyword arguments, but it might be useful to be able to handle optional arguments. We intend to generalize our technique to such lambda lists.

We have implemented the technique described in this paper, but have yet to implement a decision procedure for determining whether this technique could and should be applied. The details of this decision procedure are currently being investigated.

5. ACKNOWLEDGMENTS

We would like to thank Bart Botta, Jan Moringen, John Mercouris, and Alastair Bridgewater for providing valuable feedback on early versions of this paper.

6. REFERENCES

[1] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 134–145, New York, NY, USA, 1997. ACM.

[2] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. *SIGPLAN Not.*, 32(5):134–145, May 1997.

[3] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. *SIGPLAN Not.*, 24(7):246–257, June 1989.

[4] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 246–257, New York, NY, USA, 1989. ACM.

[5] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[6] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction Mechanisms in CLU. *Commun. ACM*, 20(8):564–576, Aug. 1977.

[7] S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[8] R. W. Scheifler. An analysis of inline substitution for a structured programming language. *Commun. ACM*, 20(9):647–654, Sept. 1977.