Partial Inlining Using Local Graph Rewriting

Irène Durand & Robert Strandh

LaBRI, University of Bordeaux

April, 2018

Context: The SICL project

https://github.com/robert-strandh/SICL

In particular, the Cleavir implementation-independent compiler framework that is currently part of SICL.

High-level Intermediate Representation

Cleavir uses (at least) two intermediate representations:

- Abstract Syntax Trees (ASTs) created from source code and a global environment.
- High-level Intermediate Representation (HIR) created from ASTs.

High-level Intermediate Representation

HIR is similar to the kind of flow graphs used in traditional compiler design.

Main difference: In HIR, only Common Lisp objects are manipulated.

By restricting HIR data this way, we can apply most of our optimization techniques to this representation, including type inference.

HIR instruction categories

The following categories exist:

- 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 particular HIR instructions

Two HIR instructions types have no correspondence in Common Lisp source code:

- The `enter` instruction is the first instruction of a sub-graph corresponding to a function.
- The `enclose` instruction creates a callable function from an `enter` instruction and the current environment.

Previous work

Most work focuses on *when* to inline.

How to inline is not discussed much, because as Chang and Hwu put it: “The work required to duplicate the callee is trivial”

Trivial in functional programming

The mechanism of inlining is trivial in the context of functional programming.

Simply replace the call by a copy of the body of the callee, with each occurrence of a parameter replaced by the corresponding argument (β-reduction).

```
(defun f (x y) (+ x (* x y)))
```

```
(defun g (a) (f (+ a 2) 234))
```

becomes

```
(defun g (a) (+ (+ a 2) (* (+ a 2) 234)))
```

Not trivial in the presence of side effects

The mechanism of inlining is not trivial in the context of a language that allows side effects. We can not use simple β-reduction.

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

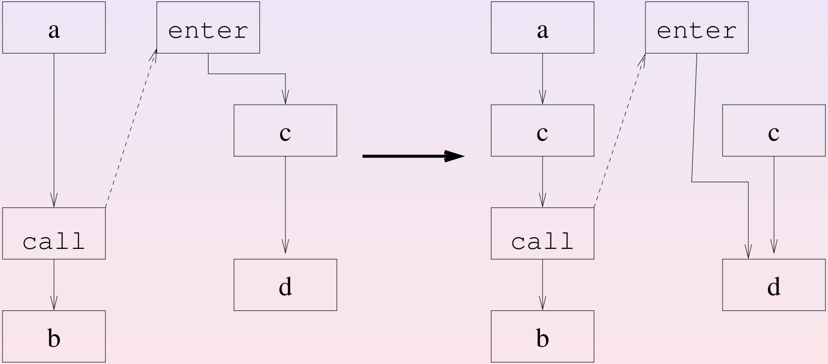
```
(defun g (a) (f a 3) a)
```

becomes

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

Our technique: local graph rewriting

Basic idea:



Our technique: restrictions

Only avoiding call/return is no longer important.

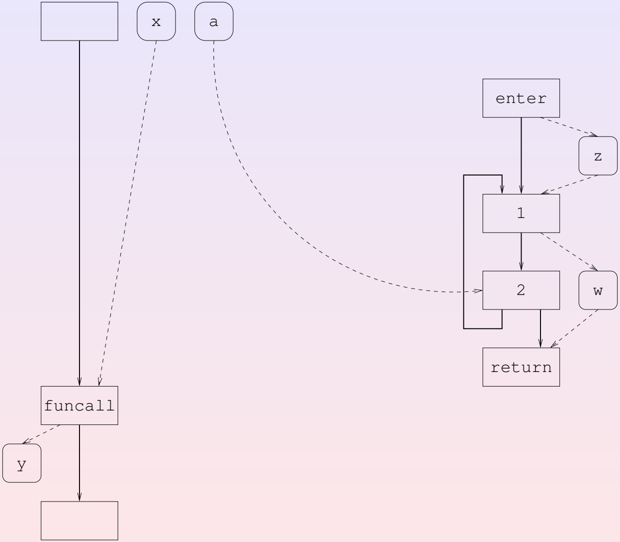
We also want to allocate the environment of the callee in the caller.

This restriction excludes some situations:

- Some cases when the environment of the callee is captured.
- When the callee is directly or indirectly recursive.

We have yet to work out necessary and sufficient conditions.

Running example



Our technique: worklist

We maintain a worklist containing:

- A `funcall` instruction (caller).
- An `enter` instruction (callee).
- The successor instruction of the `enter` instruction, called the *target instruction*.
- A mapping from lexical variables in the callee that have already been duplicated in the caller.

Our technique: global information

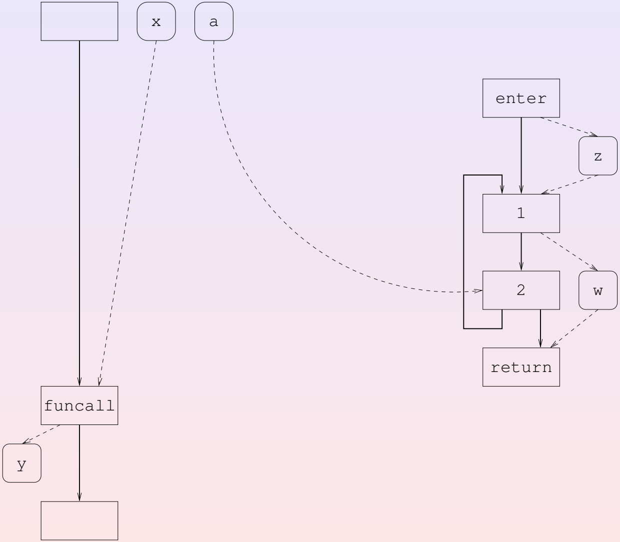
We also maintain the following global information:

- A mapping from instructions in the callee that have already been inlined, to the corresponding instructions in the caller.
- Information about the ownership of lexical variables referred to by the callee.

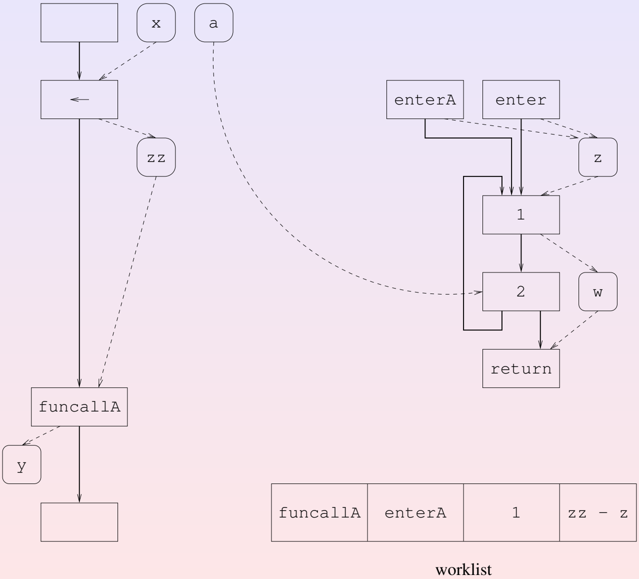
Our technique: initialization

- Create a copy of the initial callee environment in the caller.
- Create an initial worklist containing:
  - 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.

Initial instruction graph



Instruction graph after initialization

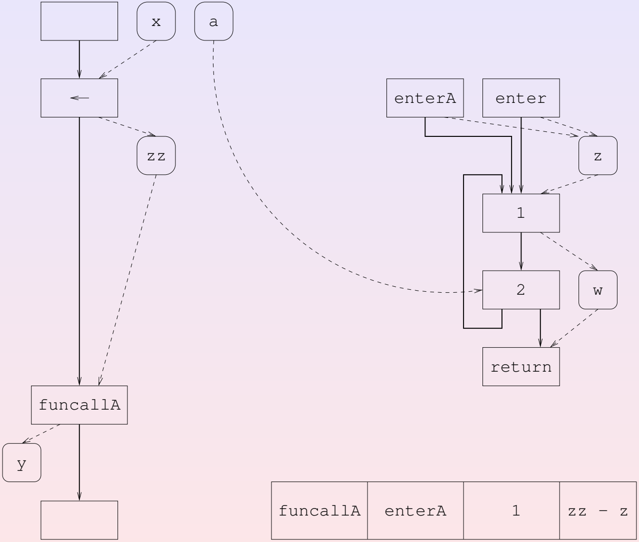


Our technique: one of four rules

In each iteration of our technique, one of the following rules is applied:

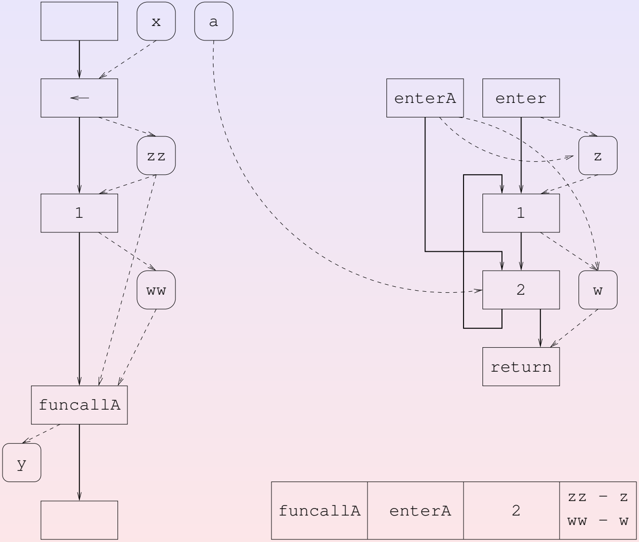
1. If the target instruction has already been inlined, then use the existing inlined copy. No new worklist item is created.
2. If the target instruction is a `return` instruction, then remove call and fix up. No new worklist item is created.
3. If the target instruction has a single successor, then inline it, mapping lexical variables. Create one new worklist item.
4. If the target instruction has two successors, then inline it, mapping lexical variables. Also replicate the `funcall` and `enter` instructions. Create two new worklist items.

Instruction graph after initialization



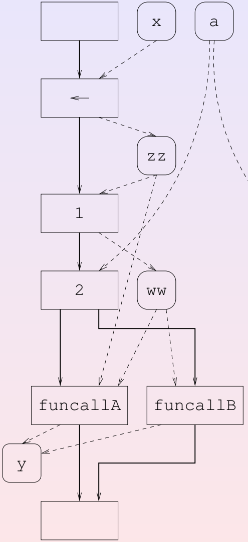
worklist

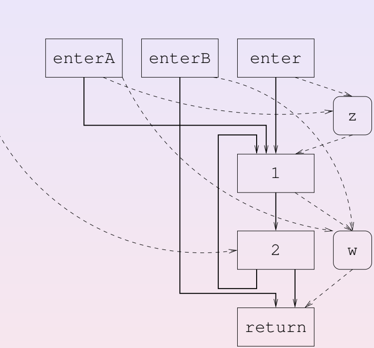
Instruction graph after one inlining step



worklist

Instruction graph after two inlining steps

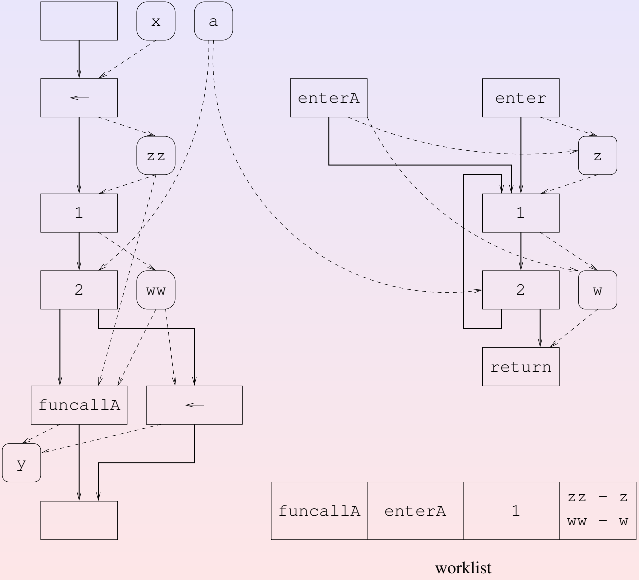




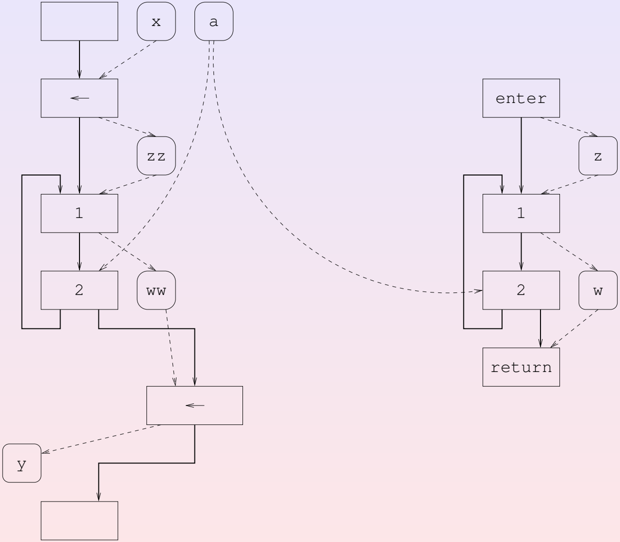
funcallB	enterB	return	ww – w
funcallA	enterA	1	zz – z ww – w

worklist

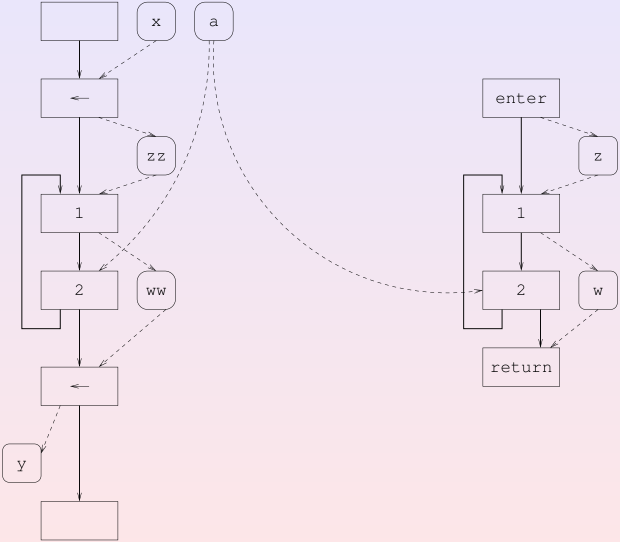
Instruction graph after three inlining steps



Instruction graph after four inlining steps



Final instruction graph



Our technique: characteristics

- Each iteration preserves the overall semantics.
- Inlining can be stopped at any point, making it partial.
- We prove termination even in the presence of loops.

Future work

- Determine necessary and sufficient conditions for our technique to be valid.
- Investigate consequences of multiple entry points for other optimization techniques and analyses.

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.

Thank you

Questions?