CMSC 22620/32620 **Implementation of** **Project 4**
Spring 2015 **Computer Languages II** **May 21, 2015**

Flang Optimization

Due: June 10, 2015

1 Introduction

In this final project, you will implement various optimizations for the **Flang** language. These will include a contraction (*i.e.*, shrinking) phase for the LambdaIR representation, as well as two or three optimizations of your choice. We will provide an end-to-end compiler for **Flang** that generates an object file using the LLVM code generation infrastructure.[1] Your optimizations should preserve census information.

2 LambdaIR contraction

The first part of the final project is to implement a contraction phase for the LambdaIR representation. At a minimum, your contraction phase should implement the following optimizations:

- dead-variable elimination
- constant folding for tuple selection
- inlining of functions that are only called once
- let-floating

You may also implement constant folding for primitive operations (*e.g.*, arithmetic) and conditionals.

3 Other optimizations

In addition to contraction, you should implement two or three additional optimizations for either the LambdaIR or Cluster representation. We list a few possible optimizations here, but you should feel free to choose optimizations that are not included here.

[1]See `llvm.org` for information.

3.1 Expansive inlining

In contraction, we only inline non-recursive functions that are called once, which guarantees that the size of the program will shrink. For functional languages, however, it is very useful to inline small functions, even if they are called multiple times. The basic operation of inlining involves substituting the arguments for parameters in a fresh copy of the function's body and the replacing the call with the resulting expression. To avoid excessive growth of the program, one typically uses inlining heuristics based on the size of the candidate function and a budget to limit overall growth. There is an extensive literature of inlining techniques.

3.2 Copy propagation

As described in class, *copy propagation* is the process of replacing the occurrences of bound variables with their values. Contraction with let floating already achieves this optimization for let-bound variables, but does not handle propagation across function applications. A related optimization is *sparse conditional constant propagation*, which combines a form of copy propagation with symbolic evaluation of conditionals.

3.3 Useless variable elimination

As discussed in class, *useless variable elimination* (UVE) is the process of eliminating variables that have no impact on the result or effects of the program. It is most effective when combined with other optimizations that are likely to expose useless variables, such as arity raising, uncurrying, and copy propagation. The implementation consists of a fixed-point analysis that identifies the *useful* variables followed by a transformation pass that removes useless variables (including function parameters) from the program.

3.4 Arity raising

Arity raising (sometimes called *argument flattening*) is an optimization that flattens tuple arguments to a function. Because functions always take a single argument in **Flang**, a naïve implementation will do extra work building and decomposing tuples that are used as arguments. For example, consider the following **Flang** program:

```
{ fun f (x : Integer * Integer) -> Integer =
      case x of { (a, b) => a + b } end;
  f (f (1, 2), 3)
}
```

which would be translated in the following LambdaIR:[2]

```
fix f (x) = let a = #0(x)
            let b = #1(x)
            let t1 = IntAdd(t1, t2)
            in t1
let t2 = [1, 2]
let t3 = f (t2)
```

[2] I have take the liberty to use immediate values in argument positions, instead of variables, since it makes the example more managable.

```
let t4 = [t3, 3]
in
  f (t4)
```

Arity raising rewrites the function `f` into a function with two arguments.

```
fix f (a, b) = let t1 = IntAdd(t1, t2)
               in t1
let t2 = f (1, 2)
in
  f (t2, 3)
```

(Note that we are assuming a pass of contraction to get rid of dead variables).

There is a rich literature about arity raising and many different published approaches. A key design choice is whether to be conservative and only flatten arguments when one is guaranteed a benefit, or be aggressive and flatten whenever possible. In the latter case, some programs may end up doing more work as a result of the optimization.

3.5 Uncurrying

The *uncurrying* optimization converts known calls of curried functions into calls of functions with multiple arguments. It has a similar effect as arity raising, but targets a different programming style. Uncurrying involves identifying functions that have curried definitions and application sites where they are applied to more than one argument. This optimization should be very helpful for **Flang** programs, since it should get rid of significant numbers of type applications (recall that type applications translate to applications of zero arguments). For example, consider the following **Flang** program:

```
{ fun id [b] (x : b) -> b = x;
  id [([c] c -> c) -> ([d] d -> d)] (id [[e] e -> e]) id [Integer] 42
}
```

The resulting LambdaIR is

```
fix id () =
       fix id1 (x) = x
       in
         id1
let t1 = id ()
let t2 = id ()
let t3 = t1 (t2)
let t4 = t3 (id)
let t5 = t5 ()
let t6 = 42
in
  t5 (int<0042>)
```

Analysis shows that `id` is a candidate for uncurrying and that the application `let t3 = t1 (t2)` provides both arguments to the uncurried form of `id`.

```
fix id' (x) = x
fix id () =
       fix id1 (x) = id' (x)
       in
         id1
```

```
let t2 = id ()
let t3 = id' (t2)
let t4 = t3 (id)
let t5 = t5 ()
let t6 = 42
in
  t5 (int<0042>)
```

In this example, only one application of `id` can be optimized directly, but if we subsequently inline the application of `id'`, then further uncurrying will be possible.

3.6 Value numbering

Value numbering (VN) is an optimization that eliminates redundant computation.[3] The basic idea of VN is to map variables to a symbolic representation of their value, such that if the symbolic representation of two variables is *congruent*, then the variables are guaranteed to have the same value at runtime and one may be replaced by the other.

To implement value numbering, we define a mapping from variables to integers (their value number) and a mapping from right-hand-side terms to value numbers. The right-hand-side terms consist of an operator (*e.g.*, a primop, allocation, or select) and a list of the value numbers of the arguments.

The algorithm assigns value numbers to bound variables in a top-down pass. When a variable can be assigned the number of some other variable, then we mark it as redundant. As we return from the value-numbering pass, we rewrite the program bottom up by replacing redundant right-hand-sides with variables. For example, consider the following LambdaIR fragment:

```
let one = 1
let j = IntAdd(i, one)
let k = IntAdd(i, one)
...
```

On the top-down pass, we identify that `k` has the same value number as `j`. As we rewrite the code bottom-up, we replace the binding of `k` with

```
let k = j
```

A subsequent contraction pass will then replace uses of `k` with `j`.

Note: when rewriting the program, your code should update census information. For example, the use count of `i` should be decremented by one when the right-hand-side "`IntAdd(i, 1)`" is replaced with `j` in the above code.

The conventional form of value numbering will treat function parameters as unknown values, but we can use the notion of congruence to get better estimations. For example, consider the following LambdaIR fragment:

```
fix f (x, h) =
    let a = #0(x)
    let b = #1(x)
    ...
    let y = [a, b]
    ...
```

[3]Two other such optimizations are Common Subexpression Elimination (CSE) and Partial Redundancy Elimination (PRE).

The bindings of `a` and `b` imply that `x` could be mapped to `[a, b]`, which is congruent with `y`. Note that to implement this improvement, one will have to enrich the LambdaIR type system.

4 Submission

Sources for Project 4 will be seeded into your phoenixforge repositories. These sources include the solution for Project 3, as well as some basic cluster optimizations and LLVM code generation.

Your submission should include a description of the optimizations (other than contraction) that you have implemented as well as at least one test program per optimization that demonstrates the effect of the optimization.[4]

We will collect projects from the SVN repositories at 10pm on Wednesday, June 10. Please make sure that you have committed your final version before then.

Revision history

May 21, 2015 Original version

[4]By "demonstrate," we mean that the output of the compiler changes if you turn the optimization on or off.