PA 4: IC Dataflow Analysis & Optimization

CSCI 434T
Spring, 2019

Overview

In this assignment, you will extend your compiler to support a general dataflow analysis framework, and then you will optimize the TAC for each method in a program using dataflow information.

Implementation Details

[Note: I use the class names from my implementation below — you will probably need to convert them to the names in your compiler.]

Control-flow Graph. You will first build a control-flow graph representation of the TAC for a method. More specifically, you should design a package that builds a CFG for a method's `TACList`. I suggest that you use a simple representation where each CFG node is a single instruction. I provide a few classes — all you need to write is a class to convert a `TACList` into a `ControlFlowGraph` object.

Generic Dataflow Analysis Framework. Your implementation of the dataflow analysis framework will include a generic dataflow engine. This engine is implemented once and reused as the base class for each analysis instance. Each analysis instance will describe the specific lattice and transfer functions that it uses. The dataflow analysis provides a method that solves the dataflow equations using the iterative algorithm explored in the homeworks. Additional methods will access to the IN and OUT values for each basic block in the CFG, once the solution is computed. The dataflow analysis engine is capable of performing both forward or and backward analyses.

Analysis Instances. You will then implement the following analysis instances:

- *Live variables analysis*: compute the variables that may be live at each program point.
- *Reaching Copies*: compute the definitions generated by *Copy* instructions that always reach each program point.
- At least one of:
  - *Constant folding analysis*: determine the variables whose values are constant at each program program.
  - *Available expressions*: compute the expressions available at each program point.

Optimization. Finally, you will use the results of the analyses that you have implemented to perform the following optimizations:

- *Dead code elimination*: Removes code that updates variables whose values are not used in any executions. This optimization will use the results from the live variable analysis.
- *Copy Propagation*: Use the results from the reaching copies analysis to perform copy propagation.
- Depending on which analyses you implemented, at least one of:
  - *Constant folding*: Uses the results from constant folding analysis and replaces each constant expression with the computed constant. (The Dragon book describes this optimization in detail.)

– *Common subexpression elimination*: Reuses expressions already computed in the program. Here you will use the information computed with the available expressions analysis. If an expression is available before an instruction that re-computes that expression, you have to replace the computation by the variable holding the result of that expression in all executions of the program. If there is no such variable, you will create a temporary variable to store the result of the expression at the site where the expression is initially computed.

  Common subexpression elimination should also remove redundant run-time checks such as array bounds checks or null pointer checks. (In HW 10, redundant null-pointer check removal was phrased as a separate analysis and optimization. You may do it that way if you prefer, although it should fit nicely into CSE if you consider a test like `check_null x` as an expression that can be removed if it is always available.)

  Note that your CSE implementation will only find syntactically equivalent expressions. To do a better job at finding common expressions, you may want to run a Copy Propagation pass before CSE, and possibly do each multiple times via the command line options.

  (As you design the code for your Available Expressions analysis domain, please see the note at the end of the handout about case classes and equality.)

Command Line Invocation. In addition to all of the options from the previous assignments, your compiler should support the following command-line options:

- Option `-dce` for dead code elimination;
- Option `-cfo` for constant folding;
- Option `-cse` for common subexpression elimination;
- Option `-cpp` for copy propagaion;
- Option `-opt` to perform all optimizations once.
- Option `-printDFA` to print the dataflow facts computed for each program point.

The compiler will perform the optimizations in the order they occur on the command line. The above arguments may appear multiple times in the same command line — in that case the compiler will execute them multiple times, *in the specified order*. The compiler should only perform the analyses necessary for the optimizations requested.

When the `-printIR` also occurs on the command line, you must print the TAC before or after optimizations, depending on where it occurs in the command line. For instance, with: `-cfo -printIR -dce`, you must print the TAC after the compiler performs constant folding, but before it removes dead code.

Your compiler must also print out the computed dataflow information when supplied with the command line option `-printDFA`. Specifically, the compiler should print the dataflow information at each point in the program for each analysis implemented. *Make sure your output is readable.* Each dataflow fact must clearly indicate the program statement that it refers to, and whether it represents the information before or after the statement.

Code Structure

You should extend your code base with three additional packages:

- `cfg`: Classes to represent and build a CFG for a `TACList`.
- `dfa`: Classes for the general dataflow solver, as well as the specific instances necessary for this assignment and any supporting classes.
- `opt`: Classes to implement the optimizations listed above.

I will give you a few starter files this week that will help you organize these three packages. Feel free to use any or all of them, or ignore them if you prefer to write them in a different way.

BasicBlock and ControlFlowGraph. These two classes in `cfg` represent one basic block and a control flow graph. Each basic block is restricted to one instruction, which is sufficient for this assignment. (Larger basic blocks avoid some computation and space overhead and facilitates more sophisticated instruction selection schemes that operate on the block level, but they are not necessary for the analyses we are performing.) When constructing the CFG, don't forget to include dummy nodes for `enter` and `exit`. (These can hold a `TACComment` instruction, a `TACNoOp` instruction, etc.). You should not need to modify these two classes, although you are free to do so if you wish.

In addition to `toString`, the `ControlFlowGraph` supports generating dot files to show the control flow graph graphically with the `dotToFile` method. If you generate `a.dot`, the following commands will show you the graph:

```
dot -Tpdf < a.dot > a.pdf
```

DataFlowAnalysis. This general class for solving dataflow instances in `dfa` will be the superclass of all analysis instances:

```
abstract class DataFlowAnalysis[T](val cfg : ControlFlowGraph) {

    def solve() : Unit = { ... }
    def in(b : BasicBlock) : T = { ... }
    def out(b: BasicBlock) : T = { ... }

    // return true iff the analysis is a forward analysis
    def isForward() : Boolean;

    // initial value for out[enter] or in[exit], depending on direction.
    def boundary() : T;

    // Top value in the lattice of T elements.
    def top() : T;

    // Return the meet of t1 and t2 in the lattice.
    def meet(t1 : T, t2 : T) : T;

    // Return true if t1 and t2 are equivalent.
    def equals(t1 : T, t2 : T) : Boolean;

    // Return the result of applying the transfer function for instr to t.
    def transfer(instr : TInstr, t : T) : T;
}
```

This class is parameterized by the type `T`, which is the type of value contained in the lattice. The `solve` method is responsible for computing the solution for the CFG passed into the constructor. After calling `solve`, the `in` and `out` methods can be used to access the dataflow facts for each basic block.

To use the framework, you extend this class with a new class — `LiveVariableAnalysis`, for example — which defines the six abstract methods describing the lattice, transfer functions, meet operator, boundary value, and direction of the analysis.

The starter code contains `ReachableAnalysis`, a very simple example analysis that determines which TAC instructions are unreachable (because there are return statements on all paths leading to them). You can use this analysis to help debug your code, and to give you ideas on how to structure your other analyses. You can often implement the transfer functions using a large case statement in the `transfer` method.

There are a number of clearly marked debugging statements in `DataFlowAnalysis.scala`. These may be useful as you develop your analyses and can be turned on with the "-d" command line flag. You may comment them out or replace them with whatever would be most useful to you.

When implementing your analyses, make reasonable design choices about how to represent the dataflow facts, and feel free to use the `scala.collection` collection classes wherever possible. *And remember: clarity of design and ease of implementation should be the primary motivation for any initial design choice.*

Optimization. This class in `opt` is the superclass for all optimizations:

```
abstract class Optimization {

    // apply the optimization to each method in p.
    def optimize(p : Program) : Unit = { ... }

    // apply the optimization to the method md.
    def optimize(md : MethodDecl) : Unit;
}
```

To create an optimization, simply create a subclass of `Optimization` and define `optimize(md)` to compute dataflow information and perform the optimization on `md`. That method should replace the method's TAC list with the optimized version. You can either provide methods in your `TACList` class to modify an existing list, or you can construct a new list to replace the old one. For your reference, I have provided a very simple `UnReachableCodeElimination` optimization that uses the `ReachableAnalysis` to eliminate unreachable code.

The `Optimization` methods return `Unit`, but you can change them to return a `Boolean` indicating whether the TAC changed. While not required, this allows you to provide a `-iter` command-line option that iteratively applies all optimizations over and over again until no additional changes happen (or a fixed upper bound on the number of iterations is reached).

Schedule

There are two milestones for PA 4:

Thursday, May 2: Your compiler should generate the Control Flow Graph for a `TACList` and support the `-printDFA` option for at least two dataflow analyses. (Reaching Copies and Live Variables may be the most straight forward, followed by Constant Folding, and then Available Expressions.) Include a brief status update in your `README.md` that tells me which analyses you have implemented and which test programs I should use to verify the correctness of your analyses.

Thursday, May 9: PA 4 is due. Your compiler should support the command line options listed above for the optimizations you have implemented.

Include a brief writeup in your `README.md` to describe any important details about the dataflow and optimization passes, a summary of your testing methodology, and any known bugs.

In your write up, demonstrate your analyses and how your compiler performs optimizations on several small representative IC programs. (That is, show me a few small IC programs, and both their unoptimized and optimized TAC.) Do you see any performance improvement? There may be other limitations of your back end preventing the optimizations from making a huge difference. What factors may be limiting how efficient the code is in this regard?

Extensions

There are many, many optimizations that are possibly in your framework. If you want to try others, have a look at: Partial Redundancy Elimination, Loop Invariant Code Motion, or Any other analysis from our discussions. These would all make excellent extensions to PA 4.

Scala Case Classes and Equality. Keep in mind that, by default, Scala uses structural equality for == tests on case classes. That is, two distinct objects of a case class will be considered equals if their fields are all equal. Thus, the following prints true, even though `x` and `y` are distinct objects.

```
case class TACNot(dst: TACOperand, src: TACOperand) extends TACInstr { }

val x = TACNot(t1, t2);
val y = TACNot(t1, t2);

println(x == y); // prints true.
```

That means that if you are storing TAC instructions in a set, searching for them in a list, or creating a map where the domain is TAC instructions, you may not be able to distinguish between two occurences of the same instruction. So, if the TAC instruction for "`t1 = !t2`" appears in multiple places in a TAC list, the standard Scala set. map, and list implementations will consider them all equal to each other.

This will potentially be an issue for analyses like CSE, for which you may choose to construct sets of TAC instructions that compute the same expression. In such a case, you will likely want to use reference equality so that you can keep track of different occurences of the same instruction. To switch to reference equality, add the following to your `TACInstr` base class:

```
abstract class TACInstr {
  override def equals(o : Any) = {
    o.isinstanceOf[AnyRef] && this eq o.asInstanceOf[AnyRef];
  }
}
```

The function `eq` is a pointer equality test. With this extension, the above test "`x == y`" yields false.