An optimization framework for toolc

Compiler Construction ’16 Final Report

Rodrigo Raya
EPFL
rodrigo.raya@epfl.ch

1. Introduction

During the development of the front-end and back-end of toolc compiler we were able to translate our programs into java bytecode and execute them using our own installation of the java virtual machine. This process is very straight forward, for the same constructs you always do the same translations. The advantage of it is that we were able to tell the eventual programmer of toolc the errors of the syntax or the semantics of his code.

Now we address the task of improving the performance of the code. Not all ways of writing code are as efficient as others and some inefficiencies can be treated in a machine-independent fashion. In this report, I explain the implementation of live variable analysis and dead code elimination together with the development of a framework that makes possible to implement further optimizations. I restrict my analysis to intra-procedural analysis which has important consequences in the implementation aspects.

2. Examples

Dead code elimination should remove code that depends of dead variables as it is formalized in section 3.1 of this document. In what follows I specify the expected output in some program examples.

Listing 1. Dead code elimination

```
//Expected output: 9 8 7 6 5 4 3 2 1 0 0
program Example1{
 println(new Ex1().compute(10));
}

class Ex1{
 def compute(num : Int) : Int = {
  var aux : Int;
  var num_aux : Int;

  while(!(num < 0) && !(num == 0)){
   aux = num;
   num = num - 1;
   println(num);
  }
  num_aux = num;
  return num;
 }
}
```

Here the expected output would look at follows:

Listing 2. Output pseudo-code program expected for first example

```
-----------------------------
compute
-----------------------------
L0:
cjump if (!((!(num < IntLit(0)))
   && (!(num == IntLit(0))))) to L1:
Removed dead code: aux = num
num = (num - IntLit(1))
println(num)
jump L0:
L1:
Removed dead code: num_aux = num
return num
```

where the two assigments to variables that are not used in the future are removed.

However if for instance, *aux* was a class variable then we could only remove the second assignment because the computation could be reused in future calls and as we are doing intra-procedural analysis we have no control on it:

Listing 3. Output pseudo-code program expected when aux is class variable

```
-------------------------------
compute
-------------------------------
L0:
cjump if (!((!(num < IntLit(0)))
    && (!(num == IntLit(0))))) to L1:
aux = num
num = (num - IntLit(1))
println(num)
jump L0:
L1:
Removed dead code: num_aux = num
return num
```

3. Implementation

3.1 Theoretical Background

3.1.1 Intermediate representations

The abstract interpretation phase of a compiler is traditionally placed between the front-end and the back-end of the compiler being built. In our case, this would be between type checking and code generation phases.

However, normal optimization frameworks work with intermediate representations and in our work we introduced two of them. The second representation is a small variation of the first and it is introduced for clearness purposes.

If we were to justify why an intermediate representation is suitable for our problem, we could give some reasons. First, this representation is independent from the source language so it could be reused for other languages as well. Second, it is a sequential representation which can be easily modified. In our case, the two constructs that motivate the introduction of intermediate representation are if and while statements.

Once we have justified the introduction of intermediate codes for our framework we can explain its design. We followed the instructions in chapter six of [Aho 2007] implementing a three-address code. Three address code is characterised by the fact that there is at most one operator on the right side of an instruction. For this purpose the language introduces temporary variables that hold partial results. For instance, the expression $x + y * z$ would be translated into two instructions with on operand each, namely $t_1 = y * z$ and $t_2 = x + t_1$. We will refer to the process of decomposing source code instructions in this format as unfolding.

On the one hand, the collection of instructions includes some conceptual elements which where absent in the original toolc language to translate if and while constructs, namely unconditional and conditional jumps and labels. On the other hand, addresses are variables, constants and temporary variables introduced by the unfolding process.

We used a second intermediate representation to approach code generation. Due to the scope of the project we could not focus on the implementation of code generation optimizations that are described in chapter six of [Aho 2007]. To avoid generating a significant number of load and store instructions for temporary variables we were compelled to fold the code again (after applying the optimizations) into a tree-like representation. Not all optimizations are suitable for this process as we will discuss in the implementation details section.

3.1.2 The control-flow graph

First step to analyze toolc programs is to build a control flow graph.

The representation of such graph is based on basic blocks which are defined as groups of instructions such that the control flow only enter them through its first instruction and control leaves the block without halting or branching except perhaps the last instruction of the block. In our case each basic block will be holding only one instruction.

To connect different basic blocks we used these principles:

Take basic block B_1 and B_2. There is an edge from B_1 to B_2 if:

- B_1 is an unconditional or conditional jump with target B_2.
- B_2 follows B_1 in the three-address-code and B_2 is not an unconditional jump.
- B_1 is the entry block and B_2 is the first instruction of the three-address code.
- B_1 is the return instruction of the three-address code and B_2 is the exit block.

Note that applying these rules we are effectively implementing intra-procedural analysis, that is, we are restricting our analysis to each method individually.

3.1.3 The data-flow framework

We first review the basic definitions on which our code rely, they are taken from [Aho 2007]:

Definition 3.1 (Data-flow analysis framework)**.**
A data-flow analysis framework consists of

- A direction of the data flow D, either forwards or backwards.
- A semilattice with a domain of values V and a meet operator $\wedge$.
- A family of transfer functions F from V to V that contains the identity function and is closed under composition.

We have the following two equivalent definitions for a meet-semilattice:

Definition 3.2 (Semilattice-Algebraic definition)**.**
A set V with a binary operator $\wedge$ is a meet-semilattice if $\wedge$ is idempotent, commutative and associative. Furthermore, there exists a neutral element for $\wedge$, $\top \in V$.

Definition 3.3 (Semilattice-Partial order definition)**.**
A set V with a partial order $\leq$ relation is a meet-semilattice if every two elements in V have a greatest lower bound.

The connection between the two definitions is as follows $x \leq y \iff x \wedge y = x$.

To study the rate of convergence of a data-flow analysis algorithm we introduce the following notion:

Definition 3.4 (Height of a semilattice)**.**
Define $x < y \iff x \leq y$ and $x \neq y$. An ascending chain in a semilattice is a finite sequence of elements related with $<$ relation as $x_1 < x_2 < \cdots < x_n$. Then we define the height of that semilattice as the length of the largest ascending chain counted as the number of $<$ symbols employed.

To prove properties of convergence of the algorithm employed we need to define some further properties on the frameworks used:

Definition 3.5 (Monotone framework)**.**
A data-flow framework is monotone if given any function $f \in F$ and any pair of elements $x, y \in V$ we have $x \leq y \implies f(x) \leq f(y)$ or equivalently $f(x \wedge y) \leq f(x) \wedge f(y)$.

Definition 3.6 (Distributive framework)**.**
A data-flow framework is distributive if given any function $f \in F$ and any pair of elements $x, y \in V$ we have $f(x \wedge y) = f(x) \wedge f(y)$.

Clearly distributivity implies monotonicity. The converse does not hold.

3.1.4 Modelling our optimizations to fit the data-flow framework

Here we describe in detail the set-up for live variable analysis. There are other analysis for which we gave a working implementation but need some adjustment in the intermediate codes to provide full optimization capabilities. These are commented in the last section of this document.

Consider the problem of live variable analysis:

Definition 3.7 (Live variable analysis)**.**
For each variable x and each program point p, determine if the value of x at p is used along some path in the flow graph starting at p.

If so, we will say that x is live at p. Otherwise, we will say that x is dead at p.

Clearly, no variable is live after the return instruction, that is, $OUT[EXIT] = \emptyset$. If we could tell what variables are live before the execution of an instruction given that we know the live variables after its execution, by induction, we could determine the live variables at every point of any program no matter its size. The equations that tell us how to do are our tranfer functions.

So the framework to solve this problem would have the following parameters: $D = backwards$, V the set of variables of the program, $\wedge$ would be $\cup$ and the transfer function would be $IN[B] = use_B \cup (OUT[B] \cup def_B)$ where:

- use_B is the set of variables whose values may be used in B prior to any definition of the variable.
- def_B is the set of variables defined in B prior to any use o the variable in B.

Why is $\cup$ the meet operator? Because of the equation $OUT[B] = \cup_{S:\text{S is successor of B}} IN[S]$. This tells us that the infimum of a set of lattice values is computed with the union operator. This can be visualized using Hasse diagrams described in the literature.

Definition 3.8 (Dead code elimination)**.**
Eliminate code that only affects dead variables.

This optimization is clear by itself some details dealing with the fact that we are doing an intra-procedural are discussed in further sections.

3.1.5 The solving algorithm

We present the algorithm to solve problems formulated in terms of our framework as Algorithm 1:

Algorithm 1: Iterative algorithm for general frameworks

```
Input  : 1.A data-flow graph, with special nodes ENTRY and EXIT.
         2.A direction of the data-flow.
         3.A set of values V.
         4. A meet operator $\wedge$.
         5. Set of functions $F$. $f_B$ is the transfer function for block B.
         6. Constant $v_{ENTRY}$ if forward or constant $v_{EXIT}$ if backward.
Output: Values $IN[B]$,$OUT[B]$ for each block in the graph.

if $D == forward$ then
    $OUT[ENTRY] = v_{ENTRY}$
    for each basic block B other than ENTRY do
        $OUT[B] \leftarrow \top$
    end
    while changes to any $OUT$ occur do
        for each basic block B other than ENTRY do
            $IN[B] \leftarrow \wedge_{\text{P a predecessor of B}} OUT[P]$
            $OUT[B] \leftarrow f_B(IN[B])$
        end
    end
end
else
    $IN[EXIT] = v_{EXIT}$
    for each basic block B other than EXIT do
        $IN[B] \leftarrow \top$
    end
    while changes to any $IN$ occur do
        for each basic block B other than EXIT do
            $OUT[B] \leftarrow \wedge_{\text{S a successor of B}} IN[S]$
            $IN[B] \leftarrow f_B(OUT[B])$
        end
    end
end
```

We have the following theorem:

Theorem 3.1.
1. If the general algorithm converges, then the result is a solution to the data-flow equations.
2. If the framework is monotone and of finite height, then the algorithm is guaranteed to converge.

Since the set of variables is finite, live variable analysis lattice will have a finite height and since their transfer functions are monotone we have the following corollary:

Corollary 3.1.
The implementation of live variable analysis converges to a valid solution.

3.2 Implementation Details

The solution is implemented using five different packages. It may seem like a wasteful effort for just implementing one optimization. However, this framework can be easily extended to hold a lot of different optimizations. As an example I provide a copy propagation optimization. I did not have enough time to fit it into the framework I describe why in what follows.

3.2.1 Handling temporary variables and side-effects

My first (naive) approach was to translate three address instructions (defined in package tac) directly into the code generation phase. This produced of course a great quantity of extra load and store instructions. Again, [Aho 2007] studies some optimizations to get rid of these. However, the big problem was an exception in the CafeBabe library [Suter 2017]. The pertinent error is *"Wide is unsupported for now."*. According to [Schwartzbach 2017] WIDE is used: *to extend the range of local variables available to the instruction from 8 bits (i.e. 0-255) to 16 bits*. Having around two hundred temporal variables I had run out of space.

My second approach was to fold the program into another intermediate representation (defined in package untac) that would get rid of temporal variables before code generation. But it turns out that there are certain situations when you cannot get rid of temporal variables at least if you combine some optimizations. In my case I did the work with dead code optimization and copy propagation.

Basically, the problem goes as follows. Without any optimization, if you scan the list of three-address code instructions and you get an instruction that defines a temporary variable, you will always find one and only one same temporary variable ahead in the list. If you introduce dead code elimination then you may or may not find that temporary variable in the list but if you find it, there will be only one occurrence. Combining dead code elimination and copy propagation changes the situation. Then you could find several times a temporary variable ahead.

The situation would not be so difficult if toolc did not have side effects. Basically, side-effects are incorporated to the language by constructs like println or class variable accesses which could appear in method calls. Therefore, propagating method calls is inefficient and incorrect. So doing copy propagation implies keeping temporary variables.

3.2.2 Improving the code for variable analysis

At the end, although we tried different optimizations we decided to be conservative an restrict ourselves to live variable analysis and dead code elimination. However, the previous work proved useful to improve the quality of the code for this analysis.

In package opt we define dead code elimination. There, method *removeIfNotLive* defines the conditions under which dead code can be eliminated. We state that class variables dependent code cannot be removed because it can be reused in other computation producing different results. Also, method calls cannot be removed because they may change class variables or produce other side effects which would change program state or behaviour.

Interestingly, there exist [Spuler 1994] techniques to detect if method calls have side effects so that we could decide more accurately if we need to optimize or not a given method call or remove code that depends on class variables.

4. Possible Extensions

From the implementation part of this report, it is pretty clear that although I got a working implementation for a particular optimization it would be easy to extend it with other optimizations. Normally, applying different optimizations in a sequence leads to a much optimized code.

I provide a reaching copies and a copy propagation implementation. If it was to be integrated in the system some changes should be made. First, in package untac the generation of the untac code should be changed to consider the case in which we had several copies of temporal variables ahead. As described in the implementation part, a careful analysis is required specially on side effects. Also, code generation which is implemented in package *code* in file *UNTAC_CodeGeneration* should take into account the existence of temporal variables.

I provide a fully working reaching definition analysis in package dfa (as the rest of analysis of the framework)it may be extended to provide a loop invariant code optimization. Some good slides on how to do this bit can be found in [Pingali 2013]. This kind of optimization is vital as most imperative languages as toolc spend most of their time in loops.

Finally, another interesting optimization which is less difficult to handle than copy propagation and that presents an example of a monotone but non distributive framework as well as an example of an infinite domain is constant folding. For this framework more details may be found in chapter nine of [Aho 2007].

References

A. V. Aho. *Compilers. Principles, Techniques and Tools*. Pearson, 2nd edition, 2007.

K. Pingali. Loop optimizations and pointer analysis, 2013. URL `https://www.cs.utexas.edu/~pingali/CS380C/2013/lectures/strengthReduction.pdf`.

M. I. Schwartzbach. Jvm documentation, 2017. URL `https://cs.au.dk/~mis/dOvs/jvmspec/ref-wide.html`.

D. Spuler. Compiler detection of function call side effects. Technical report, James Cook University, 1994.

P. Suter. Cafebabe library implementation, 2017. URL `https://github.com/psuter/cafebabe/blob/49dce3c83450f5fa0b5e6151a537cc4b9f6a79a6/src/main/scala/cafebabe/CodeHandler.scala`.