Team Phase 4 and 5 Documentation Notes

Design

Introduction

We designed our data structures with correctness as the highest priority, and scalability as the second. The reason why we chose to do so was that incorrect compilers are useless even if they are performant. Furthermore, at the beginning of our design, we weren't sure which optimizations we were doing, and we anticipated many iterations as well as debugging, hence the secondary emphasis on scalability.

Each phase of the compiler generation roughly corresponds to one package. Much of our phase 3 infrastructure[1] was left alone, except we restructured CFG and Assembly Generation, which we describe briefly in this document.

Control Flow Graph Design

Our optimizations are designed around our representation of a control flow graph. We build one control flow graph (CFG) for each method in the program. Our control flow graph structure is found in packages `cfg` and `innercfg`. A representation of CFGs can be printed out using the `./run.sh` script with command line option `--target=cfg`.

Our equivalent of basic blocks are represented by the several `OuterCFNode` variants. Each basic block is split up into chunks based on short-circuiting logic, resulting in a tree of `InnerCFNode` nodes that is associated with each `OuterCFNode.` While outer `CFConditionals` only branch on a single temp, inner `InnerCFConditionals` branch on either a temp or a binary expression, because each binary expression has a special assembly instruction.

We have two variants of `CFStatement`: `CFAssign` and `CFMethodCall.` The former represents assignments and the latter represents a call to either an import or a method defined in another CFG. We represent a method call like `a = call();` where the return value is assigned to a variable, as two `CFStatements:` a method call and then an assignment of the return register `%rax` to the variable `a.`

A CFG then looks something like Figure 1.

[1] docs/phase3.pdf

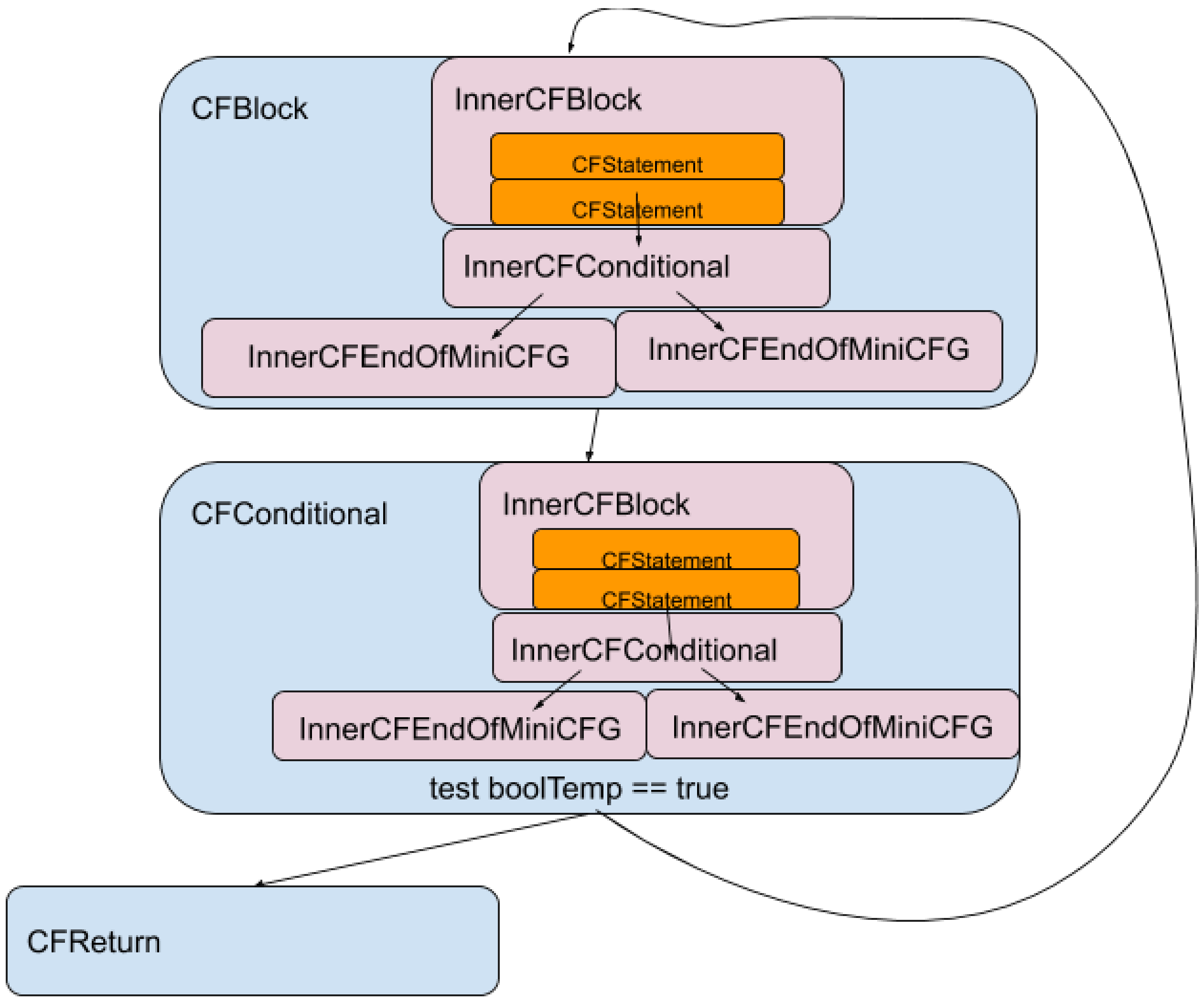

Figure 1: Structure of Control Flow Graph

Note that outer `CFReturn` nodes can also have a miniCFG to calculate a non-void return expression.

The expressiveness of the control flow graph classes makes creation of the CFG and assembly code generation very straightforward. However, some optimizations are difficult to perform (particularly, register allocation). This is why we later wrote an interface to the CFG that is appropriate for register allocation, which is found in package `reg: CFNodeIterator.`

To simplify our assumptions for assembly code generation, we use `MergeBasicBlocksAndRemoveNops` to remove all no-ops generated during the CFG generation phase. We also end every void method with an implicit void `CFReturn`. The void return, together with the runtime exception for control falling off of a non-void method, allows us to guarantee execution of the postlogue for every method CFG (see `Assembly Code Generation Overview`).

Usage of Compiler

Along with the posted compiler flags, we offer two command line options for optimization:

1) `--opt=cse` performs local and global common sub-expression elimination
2) `--opt=reg` performs register allocation for temporaries.

As specified, `--opt=all` or `--opt=cse,reg` perform both optimizations, with CSE first then Register allocation. Optimization in the opposite order would result in possibly incorrect code, so this is the only feasible ordering of the optimizations.

An example full command would thus look like the following:

```
./run.sh --target=assembly --opt=all code.dcf -o code.s
```

Assembly Code Generation Overview (package `assembly`)

AssemblyFactory

Our assembly code is generated in package assembly by AssemblyFactory. The high-level procedure is to write the method prologue, then write labels and body for each node in the CFG.

To construct the method prologue, we first use the visitor pattern to count the maximum number of non-global variables used in a Basic Block (OuterCFNode), then assign them stack offsets by augmenting their stackOffset field. Non-global variables include SharedTemps, Temps, Params, and Locals.

Our method prologue has the following structure:
- Allocate memory on the stack for all non-global variables, and initialize to 0.
- Save callee-saved registers to make them available to the body. Initialize them to 0. There are 5 callee-saved registers, so push an extra register to maintain 16-byte alignment.

Our postlogue, included in the assembly generated for any CFReturn,

MethodAssemblyGenerator

`MethodAssemblyGenerator` can be viewed as a reference table for how to convert each of our data structures into assembly code. It is called into by `AssemblyFactory`. Much of the code is copy-pasted within this file, and the file could be refactored to share code more effectively.

The reason why we generate assembly code using this hairy file is unfortunately historical. In our earlier iterations of the design, each object was responsible for its own toAssembly function, but in PR #20 (Modularize Assembly)[2], we chose to put all of the assembly generation closer together because assembly generation is highly bug-prone and we needed the code close together to perform further refactoring[3] for register allocation.

EliminateShadowingVisitor

For correctness, we label every ID with its scope using the EliminateShadowingVisitor. This is our alternative implementation to using Single Static Assignment. We effectively treat the same variable name in different scopes as a totally different variable.

Optimizations descriptions

Phase 4 Optimization

Global CSE (package `cse`)

1) First, we performed global available expressions analysis on our internal representation of basic blocks. The bulk of this analysis occurs in `GlobalAvailableSubExpressionsAnalyzer.` Analysis is broken into several steps:
    a) Collect all possible subexpressions (each assembly statement is augmented with the canonical expression it may represent). This is handled by the the CFNode (CFNodes are the basic building blocks of out CFG) visitor `CollectSubExpressions`.
    b) Calculate the gen and kill for each basic block — implemented as a visitor in `GlobalAvailableSubExpressionsAnalyzer.`
    c) Calculate INs and OUTs of each basic block using a fixed point algorithm for available expressions — also implemented in `GlobalAvailableSubExpressionsAnalyzer.`
2) The actual optimization algorithm is handled in `CommonSubExpressionEliminator.` After analysis we traverse *forwards* through the program. Whenever we reach a statement that uses an expression that is available as denoted by the INs and OUTs from earlier analysis, we employ `ExpressionSaver` which then assigns the available expression a temporary, and traverses backwards along every parent path to ensure that the expression is correctly saved.
3) We augment the control flow statements with two new optional fields: srcOptionalCSE and dstOptionalCSE. When a common subexpression is eliminated, we can augment its uses with srcOptionalCSE and augment its definitions with dstOptionalCSE.
4) In most cases, the net number of instructions is reduced because we only need to compute an expression once. Global common subexpression elimination excels when

[2] https://github.com/6035/team/pull/20
[3] Starts at https://github.com/6035/team/pull/21/commits/0bdf09da8f6516a6d96d8ba4fcde7f77368d4d1a

there are unruly repeated expressions that appear in disjoint chunks of code. See `doc/examples/global-cse`.

Local CSE

5) We performed available expressions analysis within our internal representation of basic blocks. For our control flow graph, this constitutes the InnerCFNode class. Analysis and elimination of common subexpressions on the local scale is done together in `LocalCommonSubExpressionEliminator`. The Analysis is broken into several steps:
    a) Collect all possible subexpressions (each assembly statement is augmented with the canonical expression it may represent). This is handled by the the InnerCFNode visitor `LocalCollectSubExpressions`.
    b) Perform a topological sort of the nodes in the MiniCFG.
    c) In one pass, calculate the INs and OUTs for each chunk of the basic block in topological order. Every node's parents' OUTs are calculated before it, if it has parents. We use helper functions to calculate the GENs and KILLs of each particular statement.
6) In the same class (`LocalCommonSubExpressionEliminator`), the actual optimization algorithm is in the function `handleCommonExpr`. After analysis we traverse *forwards* through the topological sort (`eliminateCommonExprs`). The only InnerCFNode that contains statements is InnerCFBlock, so we loop through the statements and keep track of INs and OUTs on the statement granularity. Whenever we reach a statement that uses an expression that is available as denoted by the INs and OUTs from earlier analysis, we assigns the available expression a temporary, and traverses backwards along previous statements within the InnerCFBlock (chunk of a basic block)
7) In most cases, the net number of instructions is reduced because we only need to compute an expression once. Local common subexpression elimination excels when there are repeated expressions appearing in local chunks of code. See `doc/examples/local-cse`.

To avoid erroneously eliminating method calls, we filter out expressions that contain a method call from the available expressions. Eliminating an expression containing a method call is not safe because it could have side effects or result in a different value on different calls. This filtering is done using an object oriented model.

Summary

Together, Global and Local CSE optimized the derby code from 2.5 to 2.25 seconds.

Phase 5 Optimization

Web Building

Our algorithm for building webs runs in roughly quadratic time relative to the size of the program. For each statement in the program, we build a web starting at that statement for each of the defined variables in the statement (for a = b, we start building a web for a). To build a web, we explore forwards in the CFG, taking branches and storing each branch we take until reaching another DEF, USE, or death of the variable. If we reach a DEF or death, we backtrack to the previous branch because the web for the variable cannot extend past it. If we reach a USE, we add it to the web, along with all nodes along the direct path from the DEF, and we also merge with webs for the same variable that contain that USE.

Note that we only attempt to build webs for appropriate variables — they must be local variables, temps, or shared temps (from global CSE). Notably, we do not attempt to allocate registers for method parameters or global variables.

`CFNodeIterator`

The algorithm described benefits from an interface to the CFG that is implemented by the class `CFNodeIterator`[4]. The main functionalities of this class are as follows:

- Constructor — either a copy of another CFNode iterator or a starting node
- boolean hasNext(), CFNode next() — retrieve the next CFNode while exploring
- void backtrackToPreviousBranch() — if any branch points were taken, backtrack to them and take the other branch this time.
- Set<CFNode> getActivePath() — retrieve the CFNodes directly between the starting node and the current location

The class is implemented by an internal representation of the location in the CFG, including, if applicable: an OuterCFNode, an InnerCFNode, an index into the statements of the InnerCFBlock, and which branch to take next.

This interface was very difficult to implement, and the resulting code is pretty hairy due to the use of global variables. This is the tradeoff for maintaining modularity with the different class variants, which is better for both correctness and scalability.

Quirks / Helpful observations of our webs:

- If Web A ends and Web B starts on the same instruction, these two webs interfere unless the assignment is a compound assignment. There may be a way to work around this, but due to our assembly generation scheme. When we generate assembly for a statement like a = t1 + b, we break it down into mov t1 → a, add b → a. If a and b are assigned the same register, this results in mov t1 → a, add t1 → a, because now the value in t1 overwrites b's register.
- For a compound assignment, our generated assembly code is conservative and doesn't assume the "USE" and "DEF" are assigned to the same register.
- When iterating through nodes, when backtracking after reaching a previously visited node, we always add the loop to the web even if there doesn't appear to be a USE. This

[4] Detailed spec written in the javadoc: src/edu/mit/compilers/reg/CFNodeIterator.java

prevents bugs[5] illustrated in Figure 2. The visited nodes could technically be omitted from the spanning statements until the head of the loop was found to be a spanning statement. However, this condition is complex to track, and the conservative thing to do is to include the visited statements always. The worst that happens is that an extra web is spilled, and we never sacrifice correctness.

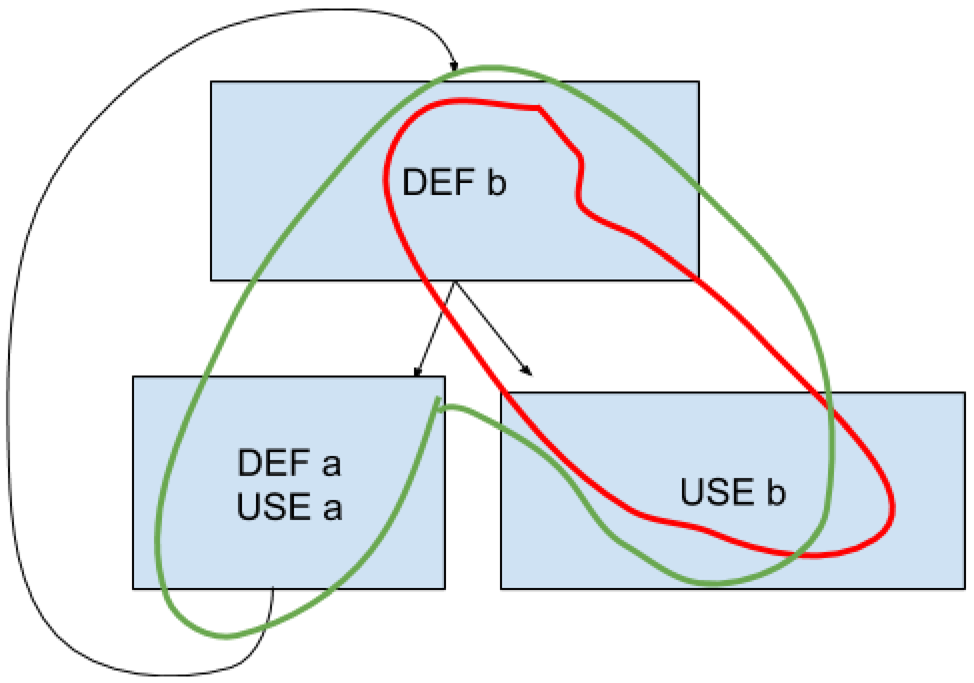

Figure 2: adding loop to web when reaching a "visited node". Green: correct web, and our behavior. Red: without such a consideration

Graph Coloring

We use the spill-cost heuristic strategy to color our interference graph. The logic for the graph coloring is found in `RegisterAllocator` and it closely follows the paper G.J. Chaitin / 1982.

Register Allocation

Our actual allocation of registers is done according to the colors of the graph by augmenting the control flow nodes with extra fields (`registerAssignmentsForUses|Defs`). We pay attention to these fields when generating assembly code (in `MethodAssemblyGenerator`).

Summary

Register Allocation, as shown in lecture, optimizes programs because register accesses are much cheaper than memory accesses on our architecture. After CSE, additional optimization of code using Register Allocation for callee-saved registers brought down the runtime of the derby code from 2.25 to 1.98 seconds.

[5] Fixed in https://github.com/6035/team/pull/21/commits/212f6fcdf62d09abee38c65471a1ad85c9405ff9

Example

One type of program in which register allocation excels is when many temps are generated that do not interfere with each other. For example, handling large expressions. See the example in `doc/examples/reg-alloc`.

Extras

We addressed difficulties from phase 3: we were thinking of making an extra MiniCFNode class to better distinguish between inner CFGs and outer CFGs. We executed this by splitting off some of the `cfg` package into `innercfg`.

Much previous code could be refactored using `CFNodeIterator`. The file abstracts away a lot of confusion. The code is very hairy and uses global variables in a message-passing style to accommodate the visitor pattern in a generic way.

Debug Strategies

Example Control Flow Graph output.

When debugging any assembly, it is helpful to have the control flow graph open side-by-side. You can obtain this output by using `--target=cfg,` in a command like:

```
./run.sh --target=cfg --debug code.dcf -o code-cfg.err
```

CFG output for each of our examples are found with suffixes `*-cfg.err`.

If this was a far longer-term project, at least a couple of years, this would warrant a prettier cfg print à la Figure 1. However, in the shorter term, by providing UIDs and pointers to UIDs in the string representations of our CFNodes, the CFGs are easily readable.

Example Web output.

When debugging register allocation and web-building logic, it is helpful to view the clusters of statements and what our analysis says about their interference. You can obtain this output by using `--target=web,` in a command like:

```
./run.sh --target=web --debug code.dcf -o code-web.err
```

Web output for each of our examples are found with suffixes `*-web.err`.

Example Debug session for segmentation fault.

To debug segmentation faults, we can run using `gdb` after compiling with debug flags, and then examine the backtrace using the `frame` command.

Output is shown in the file `doc/examples/debug/segmentation-fault.err`.

Examples of other useful commands: "start", "break _block_33", "continue".

Example Debug session for unknown register allocation bug.

To debug unknown register allocation problems, we can print out the webs, spanning statements, and the construction process of the webs.

Output is shown in the file `doc/examples/debug/register-allocation-bug.err`.

Difficulties

We had many difficulties with register allocation due to the catastrophic interaction of both our assembly generation code and interference graph code. The CFNodeIterator, while being very useful, comes with the drawback of being difficult to debug. The assembly generation code is very prone to typos due to its immenseness and dearth of code sharing.

Known Issues

- .equals() Reflexivity: To determine whether expressions are available, we use the "equals()" method on the Expr to search in the set of available expressions. However, our equals() implementation violates the contract of *reflexivity* because any Expr.METHOD_CALL does not equal itself. This leads to unexpected behavior when searching for a method call in a set of available expressions — although it should never appear because method calls are never available, it might appear due to Java's implementation of Set.contains() using reference equality as a shortcut to object equality. As a workaround, we filter method calls from the set of available expressions (described in CSE).
    - For example, Line 571 in the OpenJDK copy of Java 8[6] assumes reflexivity.
- Implicit Initialization:
    - Instead of initializing variables to 0, we simply inject a virtual DEF in the web analysis for any such statement, so that a register cannot be allocated between the beginning of the method and that variable. This leads to sub-optimal register allocation, and could be possibly fixed by generating "a = 0" statements when going to the AST to the CFG.

[6] http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashMap.java

- Register Allocation for Caller-Saved Registers (for example, `%rsi`): We fail to allocate registers for caller-saved registers. We attempted an implementation to push and pop caller-saved registers around method calls, but tests were still failing, so we did not try to allocate caller-saved registers.
  - We suspect the bug is either related to our assembly generation for these method calls, or failing to initialize them to 0.

Future Work

- We are interested in investigating Dead code Elimination and Peephole Optimization in the next couple of days.

Contribution

We each contributed code whenever we were available to do so. Register allocation was very difficult. We refactored code and planned our phase 4 and 5 implementations intermittently starting from the phase 3 checkpoint, then we steadily implemented the compiler optimizations starting at around Thanksgiving. Records of our iterations can be found as pull requests in our github repo[7].

Here are the code / comment breakdowns, courtesy of `gitinspector`[8]. Due to working from multiple machines with different SSH Key signatures, commits are under the names of multiple different people (`Henry Hu == hanryhu; Jordan Docter == Jordan E Docter == jdocter`). We excluded information about authors we didn't recognize (skeleton and test code). We include files with (`filetype=java,c,cc,cpp,h,hh,hpp,py,glsl,rb,js,sql`).

```
Author                    Commits      Insertions    Deletions     % of changes
Allan Garcia-Zych               5            87            19            0.29
Henry Hu                        7            49            13            0.17
hanryhu                       140          8789          6938           43.40
Jordan Docter                 101          9434          3664           36.14
Jordan E Docter                24          4363          1219           15.40
jdocter                         1             1             1            0.01

Below are the number of rows from each author that have survived and are still intact
in the current revision:

Author                    Rows   Stability         Age    % in comments
Allan Garcia-Zych         54           62.1        1.0                1.85
Henry Hu                  32           65.3        0.1                0.00
hanryhu                   5849         66.5        1.2                9.35
Jordan Docter             5225         55.4        1.3                5.63
jdocter                   1           100.0        1.0                0.00
```

[7] https://github.com/6035/team/pulls?q=is%3Apr+is%3Aclosed
[8] https://github.com/ejwa/gitinspector