# HW6: Dataflow Analysis and Optimizations

## Getting Started

To get started, accept the assignment on Github Classroom [https://classroom.github.com/] and clone your team's repository.

Many of the files in this project are taken from the earlier projects. The new files (only) and their uses are listed below. Those marked with `*` are the only ones you should need to modify while completing this assignment.

| | |
|---|---|
| bin/datastructures.ml | set and map modules (enhanced with printing) |
| bin/cfg.ml | “view” of LL control-flow graphs as dataflow graphs |
| bin/analysis.ml | [illegible]r functions for propagating dataflow facts |
| bin/solver.ml | [illegible] general-purpose iterative dataflow analysis solver |
| bin/alias.ml | [illegible]s analysis |
| bin/dce.ml | * dead code elimination optimization |
| bin/constprop.ml | * constant propagation analysis & optimization |
| bin/liveness.ml | provided liveness analysis code |
| bin/analysistests.ml | test cases (for liveness, constprop, alias) |
| bin/opt.ml | optimizer that runs dce and constprop |
| bin/backend.ml | * you will implement register allocation heuristics here |
| bin/registers.ml | collects statistics about register usage |
| bin/printanalysis.ml | a standalone program to print the results of an analysis |
| PerformanceExperiments.xlsx | * experimental results |



The shared, *public* git submodule to which you should push your public test case is structured as shown below (`spXX` is “spring 20XX”, e.g. sp24 is “spring 2024”):

| | |
|---|---|
| spXX_hw6_tests/ | * Git submodule to store student-generated public tests |
| spXX_hw6_tests/ | * add the info for your test case here |
| spXX_hw6/tests/ | an example test case (though it is not substantive enough!) |
| spXX_hw6_tests/ | * (please modify only *your* file) |



> **Note**
>
> You'll need to have menhir [http://gallium.inria.fr/~fpottier/menhir/] and clang [https://clang.llvm.org/] installed on your system for this assignment. If you have not already done so, follow the provided instructions [../../toolchain.html#toolchain] to install them.

> **Note**
>
> As usual, running `oatc --test` will run the test suite. `oatc` also now supports several new flags having to do with optimizations.
>
> ```
> -O1 :  runs two iterations of (constprop followed by dce)
> --liveness {trivial|dataflow} : select which liveness analysis
> to use for register allocation
> --regalloc {none|greedy|better} : select which register
> allocator to use
> --print-regs : print a histogram of the registers used
> ```

## Overview

The Oat compiler we have developed so far produces very inefficient code, since it performs no optimizations at any stage of the compilation pipeline. In this project, you will implement several simple dataflow analyses and some optimizations at the level of our LLVMlite intermediate representation in order to improve code size and speed.

## Provided Cod

The provided co e use of modules, module signatures, and functors. These nd abstraction. If you need a refresher on OCaml functors, ading through the Functors Chapter [https://dev.realworldocaml.org/functors.html] of Real World OCaml.

In `datastructures.ml`, we provide you with a number of useful modules, module signatures, and functors for the assignment, including:

> - `OrdPrintT`: A module signature for a type that is both comparable and can be converted to a string for printing. This is used in conjunction with some of our other custom modules described below. Wrapper modules `Lbl` and `Uid` satisfying this signature are defined later in the file for the `Ll.lbl` and `Ll.uid` types.
> - `SetS`: A module signature that extends OCaml's built-in set to include string conversion and printing capabilities.
> - `MakeSet`: A functor that creates an extended set (`SetS`) from a type that satisfies the `OrdPrintT` module signature. This is applied to the `Lbl` and `Uid` wrapper modules to create a label set module `LblS` and a UID set module `UidS`.
> - `MapS`: A module signature that extends OCaml's built-in maps to include string conversion and printing capabilities. Three additional helper functions are also included: `update` for updating the value associated with a particular key, `find_or` for performing a map look-up with a default value to be supplied when the key is not present, and `update_or` for updating the value associated with a key if it is present, or adding an entry with a default value if not.
> - `MakeMap`: A functor that creates an extended map (`MapS`) from a type that satisfies the `OrdPrintT` module signature. This is applied to the `Lbl` and `Uid` wrapper modules to create a label map module `LblM` and a UID map

> module `UidM`. These map modules have fixed key types, but are polymorphic in the types of their values.

## Task I: Dataflow Analysis

Your first task is to implement a version of the worklist algorithm for solving dataflow flow equations presented in lecture. Since we plan to implement several analyses, we'd like to reuse as much code as possible between each one. In lecture, we saw that each analysis differs only in the choice of the lattice, the flow function, the direction of the analysis, and how to compute the meet of facts flowing into a node. We can take advantage of this by writing a generic solver as an OCaml functor and instantiating it with these parameters.



### The Algorithm

Assuming only that we have a directed graph where each node's `in` and `out` edges are` labeled with a *dataflow fact* and each node has a *flow function*, we can compute a fixpoint of the flow on the graph as follows:

```
let w = new set with all nodes
repeat until w is empty
  let n = w.pop()
  old_out = out[n]
  let in = combine(preds[n])
  out[n] := flow[n](in)
  if (!equal old_out out[n]),
    for all m in succs[n], w.add(m)
end
```

Here `equal`, `combine` and `flow` are abstract operations that will be instantiated with lattice equality, the meet operation and the flow function (e.g., as might be defined by the gen and kill sets of the analysis), respectively. Similarly, `preds` and `succs` are the graph predecessors and successors in the *flow graph*, and so are not in one-to-one correspondence with the edges found the LL IR control-flow-graph of the program. These general operations can be instantiated appropriately to create a forwards or backwards analysis.

> **Note**
>
> Don't try to use OCaml's polymorphic equality operator (`=`) to compare `old_out` and `o`[illegible]`ence equality`, not *structural equality*. Use the supplied `F`[illegible]`d.`

## Getting Started

Be sure to review the comments in the `DFA_GRAPH` (*data flow analysis graph*) and `FACT` module signatures in `solver.ml`, which define the parameters of the solver. Make sure you understand what each declaration in the signature does – your solver will need to use each one (other than the printing functions)! It will also be helpful for you to understand the way that `cfg.ml` connects to the solver. Read the commentary there for more information.

## Now implement the solver

Your first task is to fill in the `solve` function in the `Solver.Make` functor in `solver.ml`. The input to the function is a flow graph labeled with the initial facts. It should compute the fixpoint and return a graph with the corresponding labeling. You will find the set datatype from `datastructures.ml` useful for manipulating sets of nodes.

To test your solver, we have provided a full implementation of a liveness analysis in `liveness.ml`. Once you've completed the solver, the liveness tests in the test suite should all be passing. These tests compare the output of your solver on a number of programs with pre-computed solutions in `analysistest.ml`. Each entry in this file describes the set of uids that are **live-in** at a label in a program from `./llprograms`. To debug, you can compare these with the output of the `Graph.to_string` function on the flow graphs you will be manipulating.

## printanalysis

The stand-alone program `printanalysis` can print out the results of a dataflow analysis for a given .ll program. You can build it by doing `make printanalysis`. It takes a flag to indicate which analysis to run (run with `--h` for a list).

For example, once the solver is implemented correctly, you can use it to display the results of liveness analysis for the `add.ll` program like this:



```
cis3410:/hw6>                -live llprograms/add.ll

define i64 @m                8** %arcv) {
{_entry={}
  IN : {}
   %1 = add i
  OUT: {%1}
  IN : {%1}
   ret i64 %1
  OUT: {}
}
}
```



## Task II: Alias Analysis and Dead Code Elimination



The goal of this task is to implement a simple dead code elimination optimization that can also remove `store` instructions when we can prove that they have no effect on the result of the program. Though we already have a liveness analysis, it doesn't give us enough information to eliminate `store` instructions: even if we know the UID of the destination pointer is dead after a store and is not used in a load in the rest of the program, we can not remove a store instruction because of *aliasing*. The problem is that there may be different UIDs that name the same stack slot or heap location. There are a number of ways this can happen after a pointer is returned by `alloca`:



> - The pointer is used as an argument to a `getelementptr` or `bitcast` instruction
> - The pointer is stored into memory and then later loaded
> - The pointer is passed as an argument to a function, which can manipulate it in arbitrary ways

Some pointers are never aliased. For example, the code generated by the Oat frontend for local variables never creates aliases because the Oat language itself doesn't have an "address of" operator. We can find such uses of `alloca` by applying a simple alias analysis.

### Alias Analysis



We have provided some code to get you started in `alias.ml`. You will have to fill in the flow functi[illegible]ations. The type of lattice elements, `fact`, is a map from UIDs t[illegible] of type `SymPtr.t`. Your analysis should compute, at ever[illegible]e set of UIDs of pointer type that are in scope and, additionally,[illegible]ter is the unique name for a stack slot according to the [illegible]e comments in `alias.ml` for details.

> 1. `Alias.insn_flow`: the flow function over instructions
> 2. `Alias.fact.combine`: the combine function for alias facts



### Dead Code Elimination



Now we can use our liveness and alias analyses to implement a dead code elimination pass. We will simply compute the results of the analysis at each program point, then iterate over the blocks of the CFG removing any instructions that do not contribute to the output of the program.



> - For all instructions except `store` and `call`, the instruction can be removed if the UID it defines is not live-out at the point of definition
> - A `store` instruction can be removed if we know the UID of the destination pointer is not aliased and not live-out at the program point of the store
> - A `call` instruction can never be removed

Complete the dead-code elimination optimization in `dce.ml`, where you will only need to fill out the `dce_block` function that implements these rules.

## Task III: Constant Propagation

Programmers don't often write dead code directly. However, dead code is often produced as a result of other optimizations that execute parts of the original program at compile time, for instance *constant propagation*. In this section you'll implement a simple constant propagation analysis and constant folding optimization.

Start by reading through the `constprop.ml`. Constant propagation is similar to the alias analysis from the previous section. Dataflow facts will be maps from UIDs to the type `SymConst.t`, which corresponds to the lattice from the lecture slides. Your anal[illegible]he set of UIDs in scope at each program point, and the int[illegible]UID that is computed as a result of a series of `binop` and `icmp` i[illegible]stant operands. More specifically:



> - The flow [illegible] `icmp` whose operands have been determined to be cons[illegible]ing flow with the defined UID to `Const` with the expected constant value obtained by (statically) interpreting the operation.
> - The flow out of any `binop` or `icmp` with a `NonConst` operand sets the defined UID to `NonConst`
> - Similarly, the flow out of any `binop` or `icmp` with a `UndefConst` operand sets the defined UID to `UndefConst`
> - A `store` or `call` of type `Void` sets the defined UID to `UndefConst`
> - All other instructions set the defined UID to `NonConst`



Constant propagation of this form acts a lot like an interpreter–it is a “symbolic” interpreter that can’t always produce an answer. (At this point we could also include some arithmetic identities, for instance optimizing multiplication by 0, but we’ll keep the specification simple.)

Next, you will have to implement the constant folding optimization itself, which just traverses the blocks of the CFG and replaces operands whose values we have computed with the appropriate constants. The structure of the code is very similar to that in the previous section. You will have to fill in:

> 1. `Constprop.insn_flow` with the rules defined above
> 2. `Constprop.Fact.combine` with the combine operation for the analysis
> 3. `Constprop.cp_block` (inside the `run` function) with the code needed to perform the constant propagation transformation

> **Note**

> Once you have implemented constant folding and dead-code elimination, the compiler's `-O1` flag will ask `oatc` to optimize your ll code by doing 2 iterations of (constant prop followed by dce). See `opt.ml`. The `-O1` optimizations are *not* used for te[illegible]hey are *always* performed in the register-allocation qua[illegible]ptimizations improve register allocation (see below).
>
> This coupling [illegible] have a faulty optimization pass, it might cause the quality of your register allocator to degrade. And it might make getting a high score harder.



## Task IV: Register Allocation

The backend implementation that we have given you provides two basic register allocation stragies:

> - **none**: spills all uids to the stack;
> - **greedy**: uses register and a greedy linear-scan algorithm.

For this task, you will implement a **better** register allocation strategy that makes use of the liveness information that you compute in Task I. Most of the instructions for this part of the assignment are found in `backend.ml`, where we have modified the code generation strategy to be able to make use of liveness information. The task is to implement a single function `better_layout` that beats our example "greedy" register allocation strategy. We recommend familiarizing yourself with the way that the simple strategies work before attempting to write your own allocator.

The compiler now also supports several additional command-line switches that can be used to select among different analysis and code generation options for testing purposes:

```
--print-regs prints the register usage statistics for x86 code
--liveness {trivial|dataflow} use the specified liveness analysis
--regalloc {none|greedy|better} use the specified register
allocator
```

> **Note**
>
> 
>
> The flags above *do not* imply the `-O1` flag (despite the fact that we always turn on optimi[illegible]urposes when running with `--test`). You should enable[illegible]

For testing purp[illegible]he compiler with the `-v` verbose flag and/or use the `--print-regs` flag to get more information about how your algorithm is performing. It is also useful to sprinkle your own verbose output into the backend.



The goal for this part of the homework is to create a strategy such that code generated with the `--regalloc better --liveness dataflow` flags is “better” than code generated using the simple settings, which are `--regalloc greedy --liveness dataflow`. See the discussion about how we compare register allocation strategies in `backend.ml`. The “quality” test cases report the results of these comparisons.



Of course, your register allocation strategy should produce correct code, so we still perform all of the correctness tests that we have used in previous version of the compiler. Your allocation strategy should not break any of these tests – and you cannot earn points for the “quality” tests unless *all* of the correctness tests also pass.



## Task V: Experimentation / Validation

Of course we want to understand how much of an impact your register allocation strategy has on actual execution time. For the final task, you will create a new Oat program that highlights the difference. There are two parts to this task.

### Create a test case

Push an Oat program to `sp24_hw6_tests` git submodule. This program should exhibit significantly different performance when compiled using the “greedy” register allocation strategy vs. using your “better” register allocation strategy with dataflow information. See the files `sp24_hw6_tests/regalloctest.oat` and `sp24_hw6_tests/regalloctest2.oat` for uninspired examples of such a program. Yours should be more creative. (For a challenge, try to create an Oat program for which your regis[…]egy yields a significant speedup, but for which clang is not able[…])

## Evaluate the P[…]

Use the unix `time` command to test the performance of your register allocation algorithm. This should take the form of a simple table of timing information for several test cases, including the one you create and those mentioned below. You should test the performance of the compiler in eight configurations for `.oat` source programs:

> 1. using the `--liveness trivial` `--regalloc none` flags (baseline)
> 2. using the `--liveness dataflow` `--regalloc greedy` flags (greedy)
> 3. using the `--liveness dataflow` `--regalloc better` flags (better)
> 4. using the `--clang` flags (clang)

And… all of the above plus the `-O1` flag. These experiments will test our optimizations using the Oat optimizer.

To help you with this task, the `Makefile` in this project includes two new targets `oat_experiments` and `ll_experiments` that generate executables for each case. Each such `make` target takes a parameter `FILE=<filename>` and can optionally include `OPT=-O1` to enable the Oat level 1 optimizations.

```
make oat_experiments FILE=<filename.oat> [OPT=-O1]
make ll_experiments FILE=<filename.ll> [OPT=-O1]
```

Test your compiler on these three programs:

> - `hw4programs/regalloctest.oat`
> - `llprograms/matmul.ll`
> - your own test case

For best results, use a "lightly loaded" machine (close all other applications) and average the timing over at least five trial runs.



The example below shows one interaction used to test the `hw4programs/regalloctest.oat` file in several configurations from the command line:

```
cis3410:/workspaces/upenn-cis3410/hw/hw6/soln> make oat_experiments
FILE=hw4programs/regalloctest.oat
dune build bin/main.exe
echo "Generating executables for hw4programs/regalloctest.oat with
optimization "
Generating executables for hw4programs/regalloctest.oat with
optimization
./oatc -o a_baseline.out --liveness trivial --regalloc none
hw4programs/regalloctest.oat bin/runtime.c
./oatc -o a_greedy.out --liveness dataflow --regalloc greedy
hw4programs/regalloctest.oat bin/runtime.c
./oatc -o a_better.out --liveness dataflow --regalloc better
hw4programs/regalloctest.oat bin/runtime.c
./oatc -o a_clang.out --clang hw4programs/regalloctest.oat
bin/runtime.c

cis3410:/workspaces/upenn-cis3410/hw/hw6/soln> time
./a_baseline.out
./a_baseline.out  0.46s user 0.00s system 99% cpu 0.458 total

cis3410:/workspaces/upenn-cis3410/hw/hw6/soln> time ./a_greedy.out
./a_greedy.out  0.19s user 0.00s system 99% cpu 0.187 total

cis3410:/workspaces/upenn-cis3410/hw/hw6/soln> time ./a_better.out
./a_better.out  0.18s user 0.00s system 99% cpu 0.176 total

cis3410:/workspaces/upenn-cis3410/hw/hw6/soln> time ./a_clang.out
./a_clang.out  0.00s user 0.00s system 65% cpu 0.002 total
```



The example below shows one interaction used to test the `llprograms/matmul.ll` file in several configurations that use the `-O1` flag from the command line:

```
cis3410:/workspaces/upenn-cis3410/hw/hw6/soln> make ll_experiments
FILE=llprograms/matmul.ll OPT=-O1
dune build bin/main.exe
echo "Generating executables for llprograms/matmul.ll with
optimization -O1"
```

```
Generating executables for llprograms/matmul.ll with optimization -
O1
./oatc -o a_baseline-O1.out --liveness trivial --regalloc none -O1
llprograms/matmul.ll
./oatc -o a_g[illegible]iveness dataflow --regalloc greedy -O1
llprograms/ma[illegible]
./oatc -o a_b[illegible]iveness dataflow --regalloc better -O1
llprograms/ma[illegible]
./oatc -o a_c[illegible]ang llprograms/matmul.ll -O1

cis3410:/work[illegible]3410/hw/hw6/soln> time ./a_baseline-
O1.out
./a_baseline-O1.out  1.32s user 0.00s system 99% cpu 1.324 total

cis3410:/workspaces/upenn-cis3410/hw/hw6/soln> time ./a_greedy-
O1.out
./a_greedy-O1.out  0.97s user 0.00s system 99% cpu 0.972 total

cis3410:/workspaces/upenn-cis3410/hw/hw6/soln> time ./a_better-
O1.out
./a_better-O1.out  0.42s user 0.00s system 99% cpu 0.423 total

cis3410:/workspaces/upenn-cis3410/hw/hw6/soln> time ./a_clang-
O1.out
./a_clang-O1.out  0.06s user 0.00s system 98% cpu 0.065 total
```



Don't get too discouraged when clang beats your compiler's performance by many orders of magnitude. It uses register promotion and many other optimizations to get high-quality code!

After collecting this data, use it to fill out `PerformanceExperiments.xlsx`. This spread sheet will compute the speed up for each of these programs under the various optimization configurations. It asks you to report the processor and OS version that are hosting the Docker instance that you used to test your code, as well as the total time (in seconds) for the various configurations.

## Post Your Results

For fun, please post your performance results to Ed [https://edstem.org/us/courses/52477/discussion/] on the designated thread so that everyone can see how your optimizations perform.

## Grading



**Projects that do not compile will receive no credit!**

Your team's grad[illegible]will be based on:

- 90 Point[illegible]mated tests that we provide. Note that the register-[illegible]oints *cannot* be earned with an allocator that generat[illegible]
- 5 Points for committing an interesting test case to the shared `sp24_hw6_tests` repo. (Graded manually.)
- 5 Points for the timing analysis in `PerformanceExperiments.xlsx` (Graded manually.)