CIS 341 Final Examination
2 May 2022

SOLUTIONS

1. Inference Rules and Typechecking (23 points total)

The following questions refer the inference rules given in Appendix A, which contain a simplified version of the Oat type system from HW 5.

a. (2 points) Suppose we add a new type `pos` of *positive integers* (i.e., integers greater than or equal to 0). Which subtyping relation could you add to the type system while retaining type soundness? (choose one)

☐ $\vdash$ `int` $\leq$ `pos`
☒ $\vdash$ `pos` $\leq$ `int`

b. (5 points) Which of the following are legal subtyping relations according to the rules in the appendix. That is, which of the following judgments are derivable according to the rules? (Mark all that are correct.)

☐ $\vdash$ `(int)` $\rightarrow$ `int?` $\leq$ `(int)` $\rightarrow$ `int`
☒ $\vdash$ `(int)` $\rightarrow$ `int` $\leq$ `(int)` $\rightarrow$ `int?`
☒ $\vdash$ `(int?)` $\rightarrow$ `int` $\leq$ `(int)` $\rightarrow$ `int?`
☐ $\vdash$ `int?[]` $\leq$ `int[]?`
☒ $\vdash$ `int[]` $\leq$ `int[]?`

Consider the following typechecking rule for Oat's null check statement. It is simplified from "full" Oat because we omit the global context, record type definitions, and return type analysis parts of the judgment.

$$\frac{L \vdash e : \mathit{ref}_1\texttt{?} \quad \vdash \mathit{ref}_1 \ \leq \ \mathit{ref} \quad L, x : \mathit{ref} \vdash \mathit{block}_1 \quad L \vdash \mathit{block}_2}{L \vdash \texttt{if?(}\mathit{ref}\ x\ \texttt{=}\ e\texttt{)}\ \mathit{block}_1\ \ \texttt{else}\ \ \mathit{block}_2 \Rightarrow L}\ [\text{IFQ}]$$

c. (5 points) Recall that L is the local context, and that the notation $\Rightarrow L$ means that the local context isn't changed by this statement (it is the same before and after the statement runs). Suppose we modify the rule above so that the result context includes x, that is: $\Rightarrow L, x : \mathit{ref}$, but we don't otherwise modify the frontend of the compiler. Would the resulting language be sound? Briefly explain why or why not.

Answer: It would not be sound because a subsequent statement could use x as a non-null value of type ref, but when e actually is `null` that would lead to a null pointer exception. x is only known to be non-`null` and in definitively in scope in the "then" branch.

d. (5 points) Suppose that we instead change the first premise of the IFQ rule to be $L \vdash e : \mathit{ref}?$ (that is, we replace ref_1 with ref when checking e). Would the resulting language be sound? Briefly explain why or why not.

Answer: This is sound (but will typecheck fewer programs) because after the null check, the “then” branch treats e as having type ref, which it does.

e. (6 points) Recall that we can define *scope checking* using inference rules of the form $\Gamma \vdash e$, where Γ is the set of variables in scope and e is the expression being checked. For example, the rules below show how to scope-check a variable and an OCaml-style anonymous function:

$$\frac{x \in \Gamma}{\Gamma \vdash x}\ [\mathit{var}] \qquad \frac{\Gamma, x \vdash e}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e}\ [\mathit{lam}]$$

Suppose we want to write the scope checking rule for OCaml’s `let rec` construct (simplified to have just two mutually recursive functions). An incomplete scoping rule for this expression is shown below—it is missing the premises.

$$\frac{?}{\begin{array}{lll} \Gamma \vdash \texttt{let} & \texttt{rec} & f\ x\ \texttt{=}\ e_1 \\ & \texttt{and} & g\ y\ \texttt{=}\ e_2 \\ \texttt{in} & & \\ & e & \end{array}}\ [\mathit{rec}]$$

Which of the following judgments should be used for the premises to the rec rule so that the resulting scope checking matches OCaml semantics? (Mark all that apply)

☐ $\Gamma, f \vdash e_1$	☒ $\Gamma, f, g, x \vdash e_1$
☐ $\Gamma, g \vdash e_2$	☒ $\Gamma, f, g, y \vdash e_2$
☐ $\Gamma, x \vdash e_1$	☐ $\Gamma \vdash e$
☐ $\Gamma, y \vdash e_2$	☐ $\Gamma, x, y \vdash e$
☐ $\Gamma, g, x \vdash e_1$	☒ $\Gamma, f, g \vdash e$
☐ $\Gamma, f, y \vdash e_2$	☐ $\Gamma, f, g, x, y \vdash e$

2. Compilation (26 points total)

The Oat language and compiler as implemented for this class supports one dimensional (1D) arrays that are reminiscent of Java's arrays. Of course such arrays can be nested, so the type `int[][]` is legal and works as expected. However, Oat (like Java) incurs a cost: if `a : int[][]` in Oat, the LLVM representation of `a` is a reference to an (LLVM) array of references to (LLVM) arrays. This arrangement has the benefit of letting us store length information with each array (for bounds checks) and is flexible (such arrays can be "ragged"). However, these levels of indirection can be very inefficient.

In this problem, we consider how to modify Oat to give better native support for two dimensional (2D) arrays, whose types we will write like `int[,]`. Unlike nested arrays, 2D arrays are always rectangular and cannot be accessed without reference to *both* indices. If `e : int[,]` then we will use the syntax `e[x,y]` to mean the element of `e` at coordinate (x,y). The index operation for such an element should just "compute" the location of the corresponding array element in memory from the coordinates and shouldn't require an extra indirection.

Despite the addition of this new form of arrays, we still want Oat to be type safe—it should still do proper array bounds checking. That means we will have to introduce a way of creating 2D arrays that generalizes Oat's explicit initializers. We write this as: `new t[exp1,exp2]{id1,id2 -> exp3}`

We consider each phase of the compiler in turn.

a. Lexing (3 points) Will we need to add any new *tokens* to the Oat lexer to support 2D arrays? Why or why not? (No need to fill the space!)

Answer: No: Oat already has syntax that uses `'['`, `']'`, `'{'`, `'}'`, `','`, and `'->'`, as well as the `'new'` keyword, which are the only special tokens needed for 2D arrays.

b. Parsing We will have to modify the grammars for types, expressions, and assignment left-hand sides, as well as the corresponding parts of the abstract syntax definitions. The relevant parts are shown below:

```
 1  (* In ast.ml *)
 2  type ty = | TBool | TInt | TRef of rty | TNullRef of rty
 3  and rty = | RArray of ty
 4
 5  (* In parser.mly *)
 6  ty:
 7    | TINT   { TInt }
 8    | r=rtyp { TRef r } %prec LOW
 9    | r=rtyp QUESTION { TNullRef r }
10    | LPAREN t=ty RPAREN { t }
11    | TBOOL  { TBool }
12
13  %inline rtyp:
14    | t=ty LBRACKET RBRACKET { RArray t }
```

(6 points) What changes would you make to the above abstract syntax representation to support 2D array types?

- ☐ Add a new constructor `RArray2D of ty` to `ty` on line 2.
- ☐ Add a new constructor `RArray2D of ty * int * int` to `ty` on line 2.
- ☒ Add a new constructor `RArray2D of ty` to `rty` on line 3.
- ☐ Add a new constructor `RArray2D of ty * int * int` to `rty` on line 3.

What corresponding changes would you make to the parser code? (Make sure you choose a case that corresponds to your change above. Both parts are graded together.)

- ☐ Add `| t=ty LBRACKET COMMA RBRACKET { RArray2D t }` after line 11.
- ☐ Add `| t=ty LBRACKET x COMMA y RBRACKET { RArray2D (t,x,y) }` after line 11.
- ☒ Add `| t=ty LBRACKET COMMA RBRACKET { RArray2D t }` after line 14.
- ☐ Add `| t=ty LBRACKET x COMMA y RBRACKET { RArray2D (t,x,y) }` after line 14.

(4 points) Would you expect the changes above to introduce any ambiguities to the parser? (Recall that menhir is an LR1 parser generator.) Briefly, explain why or why not.

Answer: No: The syntax cdt[,] can be distinguished from `t[]` with one character of lookahead that sees either a comma `,` or a '`]`'.

c. Typechecking (3 points) The Oat type checker will have to be adapted to check the new expression forms, which is mostly straight forward, so we skip it here. We might also consider adding one or both of the subtyping relations shown below:

$$\frac{}{\vdash \texttt{t[,]} \;\leq\; \texttt{t[][]}}\;[\textsc{2DTo1D}] \qquad\qquad \frac{}{\vdash \texttt{t[][]} \;\leq\; \texttt{t[,]}}\;[\textsc{1DTo2D}]$$

However, one rule is unsound and the other would require a very expensive runtime "coercion" to covert between array representations. Which is which? (choose one)

- ☐ Rule [2DTo1D] is unsound and [1DTo2D] is expensive.
- ☒ Rule [1DTo2D] is unsound and [2DTo1D] is expensive.

d. Frontend (i) (3 points) Recall that the representation of a 1D Oat array type `t[]` at the LLVM IR level was defined by the type: `{i64, [0 x T]}*`, where `T` is the LLVM IR translation of the source Oat type `t`. Which of the following should we pick as the type translation of the 2D array type `t[,]` (assuming that `T` is again the LLVM translation of `t`)?

- ☐ `{i64, [0 x T]}*`
- ☒ `{i64, i64, [0 x T]}*`
- ☐ `{i64, [0 x T], [0 x T]}*`
- ☐ `{i64, [0 x {i64 x [0 x T]}]}*`

e. Frontend (ii) (3 points) The Oat compiler translates a nested 1D array access like `arr[3][7]` using two `bitcast` instructions, two calls to `@oat_assert_array_length`, two `getelementptr` and two `load` instructions—one of each for each index operation. The new 2D array compilation strategy for `arr[3,7]` can eliminate half of those LLVM instructions, provided we do which of the following?

- ☒ Add a new runtime operation called `@oat_assert_array2D_lengths`, a C function that checks that *both* array indices are in bounds, and use it for the bounds check.
- ☐ Make sure to use `bitcast` to convert the 2D array type to the 1D array type before the call to `@oat_assert_array_length`.
- ☐ Use the `getelementptr` instruction to access both array bounds simultaneously when calling `@oat_assert_array_length`.
- ☐ Use the `getelementptr` instruction to dereference the second array index as part of the "path" from the first index.

f. Backend/Optimization (4 points) Happily, the backend already supports everything we need for the 2D array feature. However some optimizations would be particularly helpful for Oat's new 2D (and 1D) arrays. Name one, and briefly explain why:

Answer: Many answers were accepted for this problem. Two examples include: (1) **Function inlining of the array bounds checking** code would improve the code, since it would remove a function call and allow better register usage. Also, the bounds checks are short . (2) **common subexpression elimination** can reduce redundant calculations for array indices.

3. Dataflow Analysis (22 points total)

This problem explores how to implement *escape analysis* as an instance of the general dataflow analysis framework from HW6. An escape analysis identifies pointer values that might "escape" the scope of the function in which they are available, either by being passed as an argument to a function or by being stored in memory. Such an analysis is useful at the LLVM level for identifying which uids defined by the `alloca` instruction can be "promoted" to registers: any such pointer that does not escape.

To instantiate the dataflow framework for escape analysis, recall that we need to define (1) the lattice of flow "facts" and, (2) the flow functions that explain how instructions transform those facts. As with liveness analysis, the idea behind escape analysis is that we propagate each "escaping use" of a uid backwards through the control flow graph, so the lattice of facts will be defined as *sets* of uids. Unlike liveness analysis, "escaping" uids are never "killed" by their definition—if a uid escapes anywhere in the control flow graph it is considered to escape. The result of an escape analysis is the set of uids that escape at the entry block of the control flow graph.

a. (3 points) Recall that we need to specify the *join* $\sqcup$ (or *combine*) operation of the analysis. For this analysis, since we want to determine whether there exists a path along which the uid might escape, we should use what for $\sqcup$? (choose one)

- ☐ $\sqcup$ should be *set intersection*
- [✓] $\sqcup$ should be *set union*
- ☐ $\sqcup$ should be *set difference*
- ☐ $\sqcup$ should be *set complement*

b. (3 points) This is a *backwards* dataflow analysis. That means that we use which of the following update rules? (choose one)

☐ $out[n] = \bigsqcup_{m \in pred(n)} in[m]$ ☐ $in[n] = \bigsqcup_{m \in pred(n)} out[m]$

[✓] $out[n] = \bigsqcup_{m \in succ(n)} in[m]$ ☐ $in[n] = \bigsqcup_{m \in succ(n)} out[m]$

c. (8 points) We also need to define the *flow functions* F_I for each instruction I to specify the escapes analysis. There are a couple of considerations. First, a pointer escapes if it is stored to memory or if it is the argument of a call. Second, because escapes analysis concerns pointer values, we have to be careful about *aliasing*: if p escapes and q may alias p, then q also may escape. Given these constraints, complete the following table describing F_I by marking an "X" in the appropriate cell(s) for each instruction. Here, `t` is an LLVM type and `id1` and `id2` are LLVM uid operands (we don't care about other operands like literals so we ignore them). Also, for simplicity, we consider only functions of one argument.

Each row will have at least one "X." As an example, we have done the last three rows of the table for you.

instruction	nothing escapes	`id1` escapes if `uid` does	`id2` escapes if `uid` does	`id1` escapes if `t` is a pointer	`id2` escapes if `t` is a pointer
`uid = add i64 id1, id2`	X				
`uid = alloca t`	X				
`uid = load t id1`	X				
`store t id1, id2`				X	
`icmp cnd t id1, id2`	X				
`uid = call t1 f(t id1)`				X	
`uid = bitcast t* id1 to t2*`		X			
`uid = gep t* id1, path`		X			
`ret t, id1`				X	
`br lbl`	X				
`br i1 id1, label lbl1, label lbl2`	X				

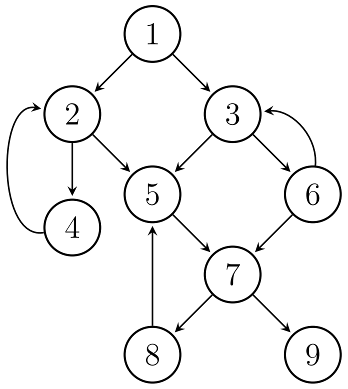
d. (8 points) Annotate the following LLVM example program with the results of computing the escape analysis described above. There are six uids of pointer type: `%ptr1` to `%ptr6`. The `IN:` and `OUT:` comments in the code below indicate which set of uids may escape after control reaches the corresponding edge of the control-flow graph. For each such annotation, mark the set of pointers that the escapes analysis will identify as escaping.

Note: As an example, we have completed the `merge` block for you. Because only `%ptr2` escapes after control reaches the entry of the `merge` block, only that uid is marked in the set.

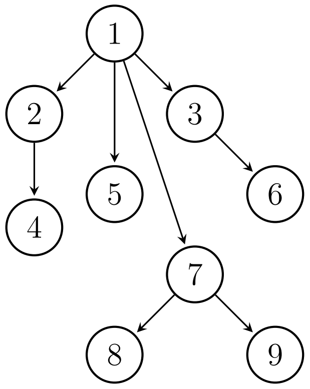
```
declare void @f(i64** %x)

define i64** @example(i64* %ptr1, i1 %flag) {
_entry:
; IN : { ☒%ptr1, ☒%ptr2, ☒%ptr3, ☒%ptr4, ☐%ptr5, ☒%ptr6 }
  %ptr2 = alloca i64*
  %ptr3 = alloca i64
  %ptr4 = alloca i64
  store i64* %ptr1, i64** %ptr2
  br i1 %flag, label %then, label %else
; OUT: { ☐%ptr1, ☒%ptr2, ☒%ptr3, ☒%ptr4, ☐%ptr5, ☒%ptr6 }

then:
; IN : { ☐%ptr1, ☒%ptr2, ☐%ptr3, ☒%ptr4, ☐%ptr5, ☐%ptr6 }
  %ptr5 = bitcast i64* %ptr3 to i64**
  store i64* %ptr4, i64** %ptr5
  br label %merge
; OUT: { ☐%ptr1, ☒%ptr2, ☐%ptr3, ☐%ptr4, ☐%ptr5, ☐%ptr6 }

else:
; IN : { ☐%ptr1, ☒%ptr2, ☒%ptr3, ☐%ptr4, ☐%ptr5, ☒%ptr6 }
  %ptr6 = bitcast i64* %ptr3 to i64**
  call void @f(i64** %ptr6)
  br label %merge
; OUT: { ☐%ptr1, ☒%ptr2, ☐%ptr3, ☐%ptr4, ☐%ptr5, ☐%ptr6 }

merge:
; IN : { ☐%ptr1, ☒%ptr2, ☐%ptr3, ☐%ptr4, ☐%ptr5, ☐%ptr6 }
  ret i64** %ptr2
; OUT: { }
}
```

4. Control-flow Analysis (25 points total)

The following questions concern the following control-flow graph, where the nodes numbered 1–6 represent *basic blocks* and the arrows denote control-flow edges. The node labeled 1 is the *entry block* of the control-flow graph.



a. (8 points) Add edges below to complete the *dominator tree* for this control flow graph:
Solution:



b. (6 points) Recall that a control-flow graph edge is a *back edge* if its target dominates its source. Recall that a *natural loop* is a strongly-connected component with a unique entry node (the header) that is the target of a back edge. For each natural loop of the control-flow graph above, identify the header node and the set of nodes making up the loop. Write each one in the form: "Header: X, Nodes: $\{$X,Y,Z$\}$"

Answer:
Header 2, Nodes: $\{2, 4\}$

Header 3, Nodes: $\{3, 6\}$

c. (3 points) The control-flow graph above has an "un-natural" loop—there is a cycle without a back edge. Which nodes participate in this cycle?

Answer: Nodes: $\{5, 7, 8\}$

d. (8 points) Mark each of the following statements about the graph above as True or False.

- This control-flow graph can be the result of compiling a well-formed Oat program using the frontend from Projects 5 and 6.

  True ☐ or False ☒

- It is possible to create this control-flow graph structure using the LLVM IR we used for the course projects.

  True ☒ or False ☐

- If we remove the control-flow edge from node 2 to node 5, the resulting graph has only natural loops.

  True ☐ or False ☒

- If we remove the control-flow edge from node 6 to node 7, the resulting graph has only natural loops.

  True ☒ or False ☐

5. Oat Compilation and Optimizations (24 points total)

The questions in this problem refer to the code shown in Appendix B Oat / LLVM Optimizations. The code shown corresponds to a simple Oat program with a function `prog`, the (slightly cleaned up) LLVM IR code generated by `oatc` for that function, and an improved version obtained by using Clang to optimize the LLVM IR code.

Oat Compilation

a. (2 points) Which LLVM uid corresponds to the Oat source variable `a`?

☐ `%0` ☐ `%1` ☒ `%3` ☐ `%4` ☐ `%5` ☐ `%6`

b. (2 points) Which LLVM uid corresponds to the Oat source variable `x`?

☐ `%0` ☐ `%1` ☐ `%3` ☐ `%4` ☒ `%5` ☐ `%6`

Liveness Analysis and Register Allocation

c. (3 points) What set of uids will be *live* upon entry to the instruction defining `%14` in `opt-O0.ll`?

`%6`

d. (3 points) What set of uids will be *live* upon entry to the instruction defining `%13` in `opt-O0.ll`?

`%3, %6, %11, %12`

e. (3 points) Consider the more optimized version of the LLVM code shown in `opt-O2.ll`. Which pairs of uids *cannot* be assigned to the same register by register allocation? (mark all that apply)

☒ `%0` and `%1` ☐ `%0` and `%3` ☐ `%0` and `%4` ☐ `%1` and `%3`

☐ `%1` and `%4` ☐ `%3` and `%4`

f. (3 points) Suppose we translate `opt-O2.ll` to x86 assembly following the x64 ABI calling conventions. The resulting code for the function `prog` *must* have a `movq` instruction. Briefly explain why.

Answer: The calling conventions require that the that the first two function arguments be in registers `%rdi` and `%rsi` and that the returned value be placed in `%rax`. Because x86 instructions take two input registers and update one of them, no matter how we assign registers to the uids, the code will have to move some value into `%rax` (either before doing the operations, or after).

General Optimizations These questions pertain to the process of optimizing the LLVM code from `opt-01.ll` to `opt-02.ll`.

g. (3 points) Which three optimizations, from those listed below, are *necessary* to achieve the transformation from `opt-01.ll` to `opt-02.ll`? (pick 3)

- ☐ common subexpression elimination
- ☒ register promotion (a.k.a. mem2reg)
- ☒ function inlining
- ☐ loop unrolling
- ☒ strength reduction
- ☐ constant propagation

h. (5 points) Suppose we add a global variable g and replace the definition of the function `foo` from the Appendix with the version shown on the left below. The resulting optimized code will be almost the same as before, but will need a few more instructions added at the `??` as shown on the right.

```
global g = 0;

void foo(int x, int y, int z) {
  g = x + y + z;
  return;
}
```

```
define i64 @prog(i64 %0, i64 %1) {
  %3 = add i64 %1, %0
  %4 = shl i64 %3, 1
  ??
  ret i64 %4
}
```

Which (if any) of the following sequences of instructions can fill in the hole `??` above to produce a correct optimization of `prog`? (choose one)

☐
```
call void @foo(i64 %4, i64 %3, i64 %3)
```

☐
```
store i64 %4, i64* @g
```

☒
```
%5 = add i64 %4, %0
store i64 %5, i64* @g
```

☐
```
%5 = shl i64 %4, 1
store i64 %5, i64* @g
```

☐ None of the above is correct.

CIS341 Final Exam 2022 Appendices

(Do not write answers in the appendices. They will not be graded)

APPENDIX A: Inference Rules

Consider a simplified variant of the Oat type system whose types, t, and reference types, ref, are generated by the following grammars:

$$\begin{array}{rcl} t & ::= & \texttt{int} \mid \mathit{ref} \mid \mathit{ref}\texttt{?} \\ \mathit{ref} & ::= & t\texttt{[]} \mid \texttt{(}t_1\texttt{)} \rightarrow t_2 \end{array}$$

Unlike full Oat, this subset leaves out `bool`, `string`, and (named) records, and has only functions of one input (rather than many inputs). The subtyping relation for this language is given by the following collection of inference rules (which are adapted from those used in HW5):

$$\boxed{\vdash t_1 \ \leq \ t_2}$$

$$\frac{}{\vdash \texttt{int} \ \leq \ \texttt{int}}\ [\text{INT}]$$

$$\frac{\vdash_r \mathit{ref}_1 \ \leq \ \mathit{ref}_2}{\vdash \mathit{ref}_1 \ \leq \ \mathit{ref}_2}\ [\text{RRREF}] \qquad \frac{\vdash_r \mathit{ref}_1 \ \leq \ \mathit{ref}_2}{\vdash \mathit{ref}_1\texttt{?} \ \leq \ \mathit{ref}_2\texttt{?}}\ [\text{NNREF}] \qquad \frac{\vdash_r \mathit{ref}_1 \ \leq \ \mathit{ref}_2}{\vdash \mathit{ref}_1 \ \leq \ \mathit{ref}_2\texttt{?}}\ [\text{RNREF}]$$

$$\boxed{\vdash_r \mathit{ref}_1 \ \leq \ \mathit{ref}_2}$$

$$\frac{}{\vdash_r t\texttt{[]} \ \leq \ t\texttt{[]}}\ [\text{ARR}] \qquad \frac{\vdash t_3 \ \leq \ t_1 \quad \vdash t_2 \ \leq \ \tau_4}{\vdash_r \tau_1 \rightarrow \tau_2 \ \leq \ \tau_3 \rightarrow \tau_4}\ [\text{FUN}]$$

Appendix B: Oat / LLVM Optimizations

`opt.oat`: An Oat source program.

```
void foo(int x, int y, int z) {
  return;
}

int prog(int a, int b) {
  var x = a + b;
  var y = x;
  foo(x, y, a);
  return y + y;
}
```

`opt-O0.ll`: Unoptimized LLVM IR code for `opt.oat`, obtained by doing `oatc -S opt.oat` (and putting the result in a file called `opt-O0.ll`):

```
define void @foo(i64 %0, i64 %1, i64 %2) {
  ret void
}

define i64 @prog(i64 %0, i64 %1) {
  %3 = alloca i64
  %4 = alloca i64
  %5 = alloca i64
  %6 = alloca i64
  store i64 %0, i64* %3
  store i64 %1, i64* %4
  %7 = load i64, i64* %3
  %8 = load i64, i64* %4
  %9 = add i64 %7, %8
  store i64 %9, i64* %5
  %10 = load i64, i64* %5
  store i64 %10, i64* %6
  %11 = load i64, i64* %5
  %12 = load i64, i64* %6
  %13 = load i64, i64* %3
  call void @foo(i64 %11, i64 %12, i64 %13)
  %14 = load i64, i64* %6
  %15 = load i64, i64* %6
  %16 = add i64 %14, %15
  ret i64 %16
}
```

`opt-O2.ll`: Optimized LLVM IR code obtained by asking LLVM to optimize the program above by doing `clang -O2 -S -emit-llvm opt-O0.ll`:

```
define i64 @prog(i64 %0, i64 %1) {
  %3 = add i64 %1, %0
  %4 = shl i64 %3, 1
  ret i64 %4
}
```