

Assignment 3: Middle End

15-411: Compiler Design

Jan Hoffmann, Jonathan Burns, DeeDee Han, Anatol Liu, Alice Rao

Due Thursday, October 12, 2017 (11:59PM)

Reminder: Assignments have to be completed individually, not in pairs. The complete work must be your own. Hand in your solutions as a PDF file on Gradescope. Please read the late policy for written assignments on the course web page.

Problem 1: Static Semantics, IR Translation (30 points)

In class, we've seen the way that typing judgments are structured. Take, for example, the typing judgment for if statements:

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash s_1 \text{ valid} \quad \Gamma \vdash s_2 \text{ valid}}{\Gamma \vdash \text{if}(e, s_1, s_2) \text{ valid}}$$

Essentially, the judgment says: for a statement $\text{if}(e, s_1, s_2)$, if e is of type **bool** in context Γ , and s_1, s_2 are *valid* in Γ , then the whole if statement is *valid* in context Γ .

We also have the following rule for the ternary (?) operator:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (e_1 ? e_2 : e_3) : \tau}$$

- if statements and the ? operator both branch based on a boolean value. Explain why the rule for the if statement judges the statement to be *valid*, while the rule for the ? operator judges the expression to have the type τ .
- Suppose we want to add support for integer comparisons to our language syntax. One way to do this is to introduce a new type **cmp**, which can take on the values **lt**, **eq**, and **gt**. We can also introduce the expression $\text{CMP}(e_1, e_2)$. The **CMP** operator will take in two integers and evaluate to **lt**, **eq**, or **gt** depending on how the arguments compare to each other.

Finally, we will introduce the statement $\text{casecmp}(e, s_1, s_2, s_3)$. The execution of $\text{casecmp}(e, s_1, s_2, s_3)$ evaluates e , and then executes s_1, s_2 , or s_3 if e evaluates to **lt**, **eq**, and **gt** respectively. The following rules begin to describe the statics of our new constructs:

$$\overline{\Gamma \vdash \text{lt} : \text{cmp}} \quad \overline{\Gamma \vdash \text{eq} : \text{cmp}} \quad \overline{\Gamma \vdash \text{gt} : \text{cmp}}$$

Write down the typing judgments for CMP and casecmp.

- c. Write a naive translation for this new construct into an IR tree as defined in lecture and the lecture notes. You may assume we have a translation $\text{tr}(e) = (c, p)$, where c is a sequence of statements and p is a pure expression containing the result of the evaluation of the expression. In our case, $\text{tr}(\text{lt})$, $\text{tr}(\text{eq})$, $\text{tr}(\text{gt})$ will return p 's that hold the values -1, 0, and 1 respectively.
- d. Write a more robust translation for our new language constructs. You may want to look at the lecture notes for translating boolean expressions.

Problem 2: Stuck in the Middle (30 points)

We generate a *Collatz sequence* c_i , starting from some positive integer n , with the following mathematical definition:

$$a_0 = n$$

$$a_{i+1} = \begin{cases} a_i/2 & \text{if } a_i \text{ is even} \\ 3a_i + 1 & \text{otherwise} \end{cases}$$

The *stopping time* of a Collatz sequence is the smallest index i such that $a_i = 1$. It is currently not known if every Collatz sequence reaches 1 (and thereby stops). The following C0 function is intended to compute the *maximum number* in the Collatz sequence for n before it stops.

```
int collatz(int n)
//@requires n >= 1;
{
  int r = n;
  while (n > 1) {
    if (n > r) r = n;
    if (n % 2 == 0)
      n = n / 2;
    else
      n = 3*n + 1;
  }
  return r;
}
```

The following is a valid three-address abstract assembly translation:

```
collatz(n):
  r <- n
  goto .loop
.loop:
  if (n > 1) then .body else .done
.body:
  if (n > r) then .l1 else .l2
.l1:
  r <- n
  goto .l2
.l2:
  m <- n % 2
  if (m == 0) then .l3 else .l4
.l3:
  n <- n / 2
  goto .loop
.l4:
  n <- n * 3
  n <- n + 1
  goto .loop
.done:
  ret r
```

- (a) Show the control flow graph of the program pictorially, carefully encapsulating each basic block. Label each basic block with the label from the abstract assembly code.
- (b) Convert the abstract assembly program from (a) into SSA with parameterized labels.
- (c) Transform your SSA code into minimal SSA. You may want to use ϕ -functions to aid you. It is sufficient to submit the final result of the transformation.
- (d) Apply the de-SSA transformation to obtain a program where labels are no longer parameterized.
- (e) Identify the two extended basic blocks in this program, and write what they are.