15-411: Compiler Design — Spring 2024

Recitation 6: Dynamic Semantics — 1 March

About Code Review Week

As you finish up Lab 3, you should spend some time on improving your codebase and making up for technical debt. If you used any terrible hacks to get register allocation or calling conventions to work, fix them. If you think any part of your code is a jumbled mess, refactor it. You'll want to have a solid base upon which to build Lab 4.

If you haven't already, you should sign up for a code review timeslot (by tonight!) using the link on Piazza. We will release a form soon for you to submit the commit hash or branch that you would like us to review. We will read your code beforehand and ask you questions about it during your team's code review meeting. While we will give you pointers on your style and structure, what we really want to check up on how well you **understand** the code you and your partner have written. We will be using your git commit log to guide our understanding of who implemented what.

If there's any significant section of your compiler that your partner implemented and you did not read, you should read it. If you don't understand how part of your compiler works, you should ask your partner to explain it to you. That said, we don't expect you to remember every detail of your implementation—we just want to make sure that both team members are participating roughly equally and have a thorough understanding of the compiler's structure.

Recap: Dynamic Semantics for L2

A configuration of an L2 program could be modeled as one of the two forms

- $\eta \vdash s \blacktriangleright K$ for *executing* statement s
- $\eta \vdash e \triangleright K$ for *evaluating* expression e

where η represents a map from variables to values and K represents the continuation (what to do next with the result of evaluating the current expression, or the next statement).

We're interested in the judgment $c \to c'$, indicating that a configuration c of the form above steps to a configuration c'. Here are a few rules for L2:

$$\begin{aligned} \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K &\longrightarrow \eta \vdash e \triangleright (\mathsf{assign}(x, \_), K) \\ \eta \vdash v \triangleright (\mathsf{assign}(x, \_), K) &\longrightarrow \eta[x \mapsto v] \vdash \mathsf{nop} \blacktriangleright K \\ \eta \vdash \mathsf{nop} \blacktriangleright (s, K) &\longrightarrow \eta \vdash s \blacktriangleright K \end{aligned}$$

We omit many rules—for a more complete set, refer to Lecture 12.

Let c_1 be the initial configuration, and suppose $c_i \to c_{i+1}$. If c_n is a final configuration of the form $\eta \vdash v \triangleright (\mathsf{return}(\_), K)$, then we say that $c_1, c_2, \ldots c_n$ is the *execution trace* of c_1.

Checkpoint 0

Draw the execution trace of configurations starting from:

$$\cdot \vdash \mathsf{seq}(\mathsf{assign}(x, 3), \mathsf{return}(x)) \blacktriangleright \cdot$$

Dynamic Semantics for L3

L3's dynamic semantics is slightly more interesting in that returning from a function call should restore state and control to the configuration prior to the call. We amend our configuration to hold a fourth

element, the call stack S, which consists of tuples of the form $\langle \eta, K \rangle$. We reproduce the rules for single-argument functions below:

$$
\begin{array}{rcl}
S; \eta \vdash f(e) \triangleright K & \longrightarrow & S; \eta \vdash e \triangleright (f(\_), K) \\
S; \eta \vdash v \triangleright (f(\_), K) & \longrightarrow & (S, \langle \eta, K \rangle); [x \mapsto v] \vdash s_f \blacktriangleright \cdot \\
 & & \textit{supposing that } f \textit{ is defined as } f(x)\{s_f;\} \\
(S, \langle \eta, K \rangle); \eta' \vdash v \triangleright (\mathsf{return}(\_), K') & \longrightarrow & S; \eta \vdash v \triangleright K
\end{array}
$$

Checkpoint 1

Draw the execution trace of the following program, starting execution at the beginning of `main`:

```
int f(int x) { return x; }
void g() { 4; }
int main() { int y = f(3); g(); return y; }
```

Dynamic Semantics for Memory

In L4, you will implement memory operations. But before we get there, let's get an understanding of how their dynamic semantics work! First, we amend our configuration to hold a fifth element, the heap H, which map addresses to stored values. For any non-memory operations and statements that don't involve memory operations, this H will remain the same on either side of our stepping rule. See Lecture 13 and Lecture 14 for more detailed expansion of the dynamic semantic rules to handle memory operations.

How do you think the following piece of code should behave?

```
int main() {
    int *x = NULL;
    *x += 1 / 0;
    return 0;
}
```

Well, if we had elaborated

```
*x += 1 / 0;
```

out to be

```
*x = *x + 1 / 0;
```

we might expect a memory exception because of a null dereference. Well it turns out that C0 actually raises a floating point exception due to a division by zero for the example code.

So, we would like to wait to unfold the += so that we can evaluate the RHS before loading anything into memory. We can represent this behavior by adding the following rules for our dynamic semantics for memory operations:

$$
\begin{array}{rcl}
H; S; \eta \vdash *v \triangleright (\_ += e, K) & \longrightarrow & H; S; \eta \vdash e \triangleright (*v += \_, K) \\
H; S; \eta \vdash c \triangleright (*v += \_, K) & \longrightarrow & H; S; \eta \vdash *v + c \triangleright (\mathsf{assign}(*v, \_), K)
\end{array}
$$

Checkpoint 2

With these new rules, draw the execution trace of the questionable line from earlier:

```
*x += 1 / 0;
```