CS345H: Programming Languages

Lecture 16: Imperative Languages II

Thomas Dillig

Overview

- Last time, we have seen how we can give meaning to a simple imperative language
- Specifically, we wrote operational semantics for the IMP1 language
- Today: How to give semantics to more feature-rich imperative languages

Pointers

- In the language IMP1, perhaps the biggest missing feature is pointers
- A pointer is a reference to a memory location
- Pointers are naturally supported by hardware through load and store instructions
- In fact, pretty much all code turns into pointer manipulation at the assembly level

Why Pointers?

- What are pointers good for?
  - Call-by-reference in a call-by-value language
  - Clever and efficient data structures
  - Avoid copying of data if it can be shared
- It is not uncommon for pointers to to be 100x faster than copying data!
- For this reason, pointers are essential for most performance-critical task.

A Simple Pointer Language

- Let us consider the following simple language with pointers we will call IMP2:

$$\begin{array}{lcl} P & \rightarrow & \varepsilon \mid P_1; P_1 \mid S \\ S & \rightarrow & \text{if}(C) \text{ then } S_1 \text{ else } s_2 \text{ fi } \mid id = e \mid *id = e \\ & & \mid \text{while}(C) \text{ do } S \text{ od } \mid id = \text{alloc} \\ e & \rightarrow & id \mid e_1 + e_2 \mid e_1 - e_2 \mid int \mid *id \\ C & \rightarrow & e_1 \leq e_2 \mid e_1 = e_2 \mid \text{not} C \mid C_1 \text{ and } C_2 \end{array}$$

- This is the same as IMP1, just with a load and store operation
- Here, I am using C syntax for loading and storing
- Addition: Alloc allocates fresh memory

Operational Semantics with Pointers

- We want to give operational semantics to this language
- But how can we handle pointers?
- Recall: So far, we only had a environment.
- The environment mapped variables to values
- But how can we look up the value of a pointer?

Operational Semantics with Pointers Cont.

- Idea: Add one level of indirection in the environment.
- We used to have one environment that maps variables to values
- Now, we will have:
  - An environment E mapping variables to addresses
  - A store S mapping addresses to values stored at this address
- The store is emulating memory when executing a program!

The Store

- This means that our operational semantics will now be of the form

$$\frac{\ldots}{E, S \vdash \ldots}$$

- Specifically, expression rules will be of the form:

$$\frac{\ldots}{E, S \vdash e : v}$$

- Conditional rules are of the form:

$$\frac{\ldots}{E, S \vdash e : bool}$$

- And statement rules are of the form:

$$\frac{\ldots}{E, S \vdash e : E', S'}$$

- Statements now both change the environment and the store!

The Store in Action

- Let start with expressions and take a look at the rule for id
- Recall, in IMP1 the operational semantics for id just returned $E(id)$
- Now, let's write the same rule for IMP2:

$$\frac{\begin{array}{c} l_1 = E(id) \\ v = S(l_1) \end{array}}{E, S \vdash id : v}$$

The alloc Statement

- Intended semantics of alloc: Return a fresh address in S that is not used by anyone else
- Here are the operational semantics of alloc:

$$\frac{\begin{array}{c} l_f \text{ fresh} \\ S' = S[l_f \leftarrow 0] \\ S'' = S'[E(v) \leftarrow l_f] \end{array}}{E, S \vdash id = \text{alloc} : E, S''}$$

Load in IMP2

- Next: The load expression
- What do we have to do to load a value?
  - Look up the address l_1 of the variable in E
  - Look up the value of l_1 in S as v_1
  - Look up the value of v_1 in S
- Here is the rule for load:

$$\frac{\begin{array}{c} l_1 = E(id) \\ v_1 = S(l_1) \\ v_2 = S(v_1) \end{array}}{E, S \vdash *id : v_2}$$

Store in IMP2

- Next: The store statement $*id = e$
- What do we have to do to store a value?
  - Look up the address l_1 of the variable v in E
  - Look up the value of l_1 in S as l_2
  - Change the value of l_2 in S to e's value
- Here is the rule for store:

$$\frac{\begin{array}{c} E, S \vdash e : v \\ l_1 = E(id) \\ l_2 = S(l_1) \\ S' = S[l_2 \leftarrow v] \end{array}}{E, S \vdash *id = e : E, S'}$$

Storage for Variables

- So far, we have been sloppy about the storage associated with variables
- Specifically, we have assumed that every variable can be looked up in E
- But this is clearly not the case unless some rule adds them to E!
- Question: How can we solve this problem?

Storage for Variables Cont

- Solution 1: Two cases for each rule where we use a variables
  - One case if the variable is already in E
  - One case if variable is not yet in E
- Solution 2: Add variable declarations to our language
- Specifically, add a `declare id` statement
- Semantics of `declare id`:

$$\frac{\begin{array}{c} l_f \text{ fresh} \\ E' = E[id \leftarrow l_f] \end{array}}{E, S \vdash \text{declare } id : S, E'}$$

- This is the solution preferred by most imperative languages

Aliasing

- As soon as we allow pointers, we also allow aliasing
- Two pointers alias if they point to the same memory location
- Here is a simple example program:

```
declare x, y;
x = alloc;
y = x;
*x = 3;
*y = 4;
```

- What is the value of `*x`?

Aliasing Cont.

- In one sense, aliasing is great.
- In fact, many the cases where pointers are really useful involve some kind of aliasing
- However, in another sense, aliasing is awful
- Because of aliasing, storing a value into any location can potentially change every other location's value!
- This is very bad news for any kind of expressive type system

Run-time errors

- Question: What kind of new run-time errors can happen in IMP2?
- Run-time errors everywhere!
- This is another typical "side effect" of adding pointers to a language

Even More Features

- Another popular feature of imperative languages: arrays
- Array is nothing but a list of values
  - Indexed by position
  - Corresponds to a contiguous region of memory
- Popular because fast
  - Accessing an element only requires adding to the base pointer
  - Can perform in-place updates of values

Arrays

- Important: What is called an array in Python is not what we are talking about here!
- Python arrays are lists of values
- These lists can even contain elements of different type
- We are talking about the C/Java style array here

Arrays

- Important: What is called an array in Python is not what we are talking about here!
- Python arrays are lists of values
- These lists can even contain elements of different type
- We are talking about the C/Java style array here

Array Language

- Consider the following modified language we will call IMP3:

$$
\begin{array}{lcl}
P & \rightarrow & \varepsilon \mid P_1; P_1 \mid S \\
S & \rightarrow & \text{if}(C) \text{ then } S_1 \text{ else } s_2 \text{ fi } \mid id = e \mid id[e_1] = e_2 \\
 & & \mid \text{while}(C) \text{ do } S \text{ od } \mid id = \text{alloc} \mid \text{declare } id \\
e & \rightarrow & id \mid e_1 + e_2 \mid e_1 - e_2 \mid int \mid id[e] \\
C & \rightarrow & e_1 \leq e_2 \mid e_1 = e_2 \mid \text{not } C \mid C_1 \text{ and } C_2
\end{array}
$$

- Observe that load and store are replaces with array load and store $\Rightarrow$ pointer arrays are a generalization of pointers
- Also, assume that `alloc` allocates arrays of infinite size

Semantics of IMP3

- The only new statements are array load and array store
- They replace load and store from IMP2
- Question: How can we emulate pointer load and store in IMP3?
- Answer: Pointer load `id = *e` is the same as `id = e[0]`
- Pointer store `*id = e` is the same as `id[0] = e`

On to the Operational Semantics

- Fortunately, the only change from IMP2 are the array load and store
- Therefore, we only need to write two new rules
- First order of business: Array load

Load in IMP3

- What do we have to do to load a value in an array?
- Specifically, how do we process $id[e]$?
  - Evaluate e to v_i
  - Look up the address l_1 of the variable id in E
  - Look up the value of l_1 in S as v_1
  - Add the index v_i to v_1 as v_2
  - Look up the value of l_2 in S

Load in IMP3

- Here is the rule for load:

$$\frac{\begin{array}{c} E, S \vdash e : v_i \\ l_1 = E(id) \\ v_1 = S(l_1) \\ v_2 = v_1 + v_i \\ v_3 = S(v_2) \end{array}}{E, S \vdash id[e] : v_3}$$

- Observe how this is a generalization of the earlier rule for pointer load

Store in IMP3

- Next: The store statement $id[e_1] = e_2$
- What do we have to do to store a value?
  - Evaluate e_2 to v_i
  - Look up the address l_1 of the variable v in E
  - Look up the value of l_1 in S as l_2
  - Change the value of $l_2 + v_i$ in S to e_1's value

Store in IMP3 Cont.

- Here is the rule for store:

$$\frac{\begin{array}{c} E, S \vdash e_1 : v_i \\ E, S \vdash e_2 : v \\ l_1 = E(id) \\ l_2 = S(l_1) \\ l_3 = l_2 + v_i \\ S' = S[l_3 \leftarrow v] \end{array}}{E, S \vdash id[e_1] = e_2 : E, S'}$$

- Again, this is a direct generalization of the store rule in IMP3

Arrays Discussion

- We have seen how to add pointer arrays to an imperative language
- However, it is also possible to add arrays without introducing pointers
- In this case, it is possible to get away without using a store
- You will write semantics for an array language without pointers on the homework

Further Features

- The imperative languages we studied still lack many features of real languages
- Some features we did not discuss:
  - How to handle different types
  - Casting
  - Expressions with side effects (e.g., `i++`)
  - ...
- But we covered all the important basics!
- If you think a little, you can now write semantics for any missing feature

Further Features Cont.

- If you want to add a new feature, first think if you need more information!
- Sometimes, you need another mapping (like environment E and store S)
- In general, there are many correct ways to add features to operational semantics
- But your goal is to add them cleanly!