CS345H: Programming Languages

Lecture 14: Introduction to Imperative Languages

Thomas Dillig

Functional Languages

- All languages we have studies so far were variants of lambda calculus
- Such languages are known as functional languages
- We have also seen that these languages allow us to design powerful type systems
- And even perform type inference

Salient Features of Functional Languages

- The functional languages we studied have a set of defining features:
- Most noticeable feature: No side effects!
- This means that evaluating an expression never changes the value of any other expression
- Example:
  `let x = 3+4 in let y = x+5 in x+y`
- Here, evaluating the expression `x+5` cannot change the value of any other expression

No Side Effects

- No side effects means no assignments and no variables!
- Recall: Let-bindings are only names for values
- The value they stand for can never change
- Example:
  `let x = 3 in let x = 4 in x`

Impact of No Side Effects

- Question: How can we exploit the fact that evaluating expressions never changes the value of any other expression?
- Answers:
  - We can evaluate expressions in parallel
  - We can delay evaluation until a value is actually used
- Question: What kind of side effect can evaluating expressions still have?
- Answer: They may still trigger a run-time error

Impact of No Side Effects Cont.

- Unfortunately, run-time errors negate all the benefits we listed!
- Question: What can we do about this?
- Solution: Type systems
- Any sound type system will guarantee no run-time errors
- Conclusion: We can only fully take advantage of functional features if we use a sound type system

The Alternative to Functional Programming

- However, there is also an alternative (and much more common) way of programming called imperative programming
- Features of imperative programming:
  - Side effects
  - Assignments that change the values of variables
  - Programs are sequences of statements instead of one expression
- Imperative programming is the dominant model
- This style is much closer to the way hardware executes

Imperative Programming Languages

- You have all used imperative programming languages
- Imperative Languages:
  - FORTRAN
  - ALGOL
  - C, C++
  - Java
  - Python
  - ...

Features of Imperative Languages

- At a minimum, a language must have the following features to be considered imperative:
  - Variables and assignments
  - Loops and Conditionals and/or goto
- Observe that features such as pointers, recursion and arrays are optional
- For example, FORTRAN originally only had integers and floats, loops, conditionals and goto statements

Example Compare and Contrast

- Let's look at some example imperative programs
- I will use C style since most of you should be familiar with this
- Adding all numbers from 1 to 10 in L:

```
fun add with n =
  if n == 0 then 0 else n + (add (n-1)) in (n 10)
```

- Here is the same program in C:

```
int res = 0, i;
for(i=0; i < 10; i++) res += i;
return res;
```

- Question: Which style do you prefer?

Very basic imperative programming

- Now, let's get even more basic and only use conditionals and goto statements to write the same program:

```
int res = 0, i;
again:
  res +=i;
  i++;
  if(i<10) goto again;
return res;
```

- Which style do you prefer?

GOTOs in Programming

- All early imperative languages include goto statements
- Rational: 1) Hardware supports only compare and jump instructions 2) GOTOs allow for more expressive control flow
- Example of GOTO use:

```
int i = 0;
int sum;
again:
  i++;
  int z = get_input();
  if(z < 0) goto error:
  n+=z;
  if(i < 5) goto again:
return n;
error:
  return -1;
```

GOTOs in Programming Cont.

- Not so long ago, it was universally accepted that GOTO statements are necessary for expressive programs
- However, as software became larger, GOTO statements started becoming problematic
- Central Problem of GOTO: “Spagetti Code”
- This means that thread of execution is very hard to follow in program text
- Jumps to a label could come from almost anyplace (in extreme cases even from other functions!)

GOTOs in Programming Cont.

- In much early (and also more recent) code, GOTO not only implemented loops but was also used for code reuse
- Real Comment from numerical analyst: “Why bother writing a function if I can just jump to the label?”
- In 1968, Dijkstra wrote a very influential essay called “GOTO Statement Considered Harmful” in which he argued that GOTO statements facilitate unreadable code and should be removed from programming languages

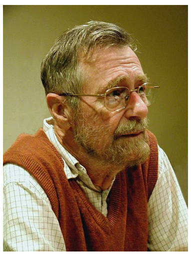
The End of GOTO

- At first, this article was very controversial
- But over time, most programmers started to agree that GOTO constructs should be avoided
- Imperative programming without GOTOs is known as structural programming
- But not everyone was on board...

Side Trip: GOTO and COBOL

- COBOL stands for COmmon Business Oriented Language
- In addition to GOTO, COBOL also includes the ALTER keyword
- After executing `ALTER X TO PROCEED TO Y`, any future `GOTO X` means `GOTO Y` instead
- Can change control flow structures at runtime!
- This was marketed as allowing polymorphism

Side Trip: GOTO and COBOL



- Dijkstra’s comment: “The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense.”

Structured Programming

- Today there is a consensus that GOTOs are not a good idea
- Instead, imperative languages include many kinds of loops and branching constructs
- Examples in C++: `while, do-while, for, if, switch`
- One legitimate use of GOTO: Error-handling code
- This popularized exceptions in most modern languages

A Simple Imperative Language

- Let's start by looking at at a very basic imperative language we will call IMP1:

$$
\begin{array}{lcl}
P & \rightarrow & \epsilon \mid S_1; S_2 \\
S & \rightarrow & \text{if}(C) \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid id = e \\
 & & \mid \text{while}(C) \text{ do } S \text{ od} \\
e & \rightarrow & id \mid e_1 + e_2 \mid e_1 - e_2 \mid int \\
C & \rightarrow & e_1 \leq e_2 \mid e_1 = e_2 \mid \text{not } C \mid C_1 \text{ and } C_2
\end{array}
$$

- This language has variables, declarations, conditionals and loops
- But no pointers, functions, ...
- What are some example programs in IMP1?

Semantics of IMP1

- Let's try to give operational semantics for this language
- First, we will again use an environment E to map variables to their values
- Start with the semantics of expressions
- Question: What do expressions evaluate to?
- Answer: Integers
- Therefore, the result (value after colon) in operational semantics rules for expression is an integer

Semantics of IMP1

- Here are operational semantics for expressions in IMP1 (first cut)
  - Variable:

$$\frac{}{E \vdash v : E(id)}$$

  - Plus

$$\frac{\begin{array}{c} E \vdash e_1 : v_1 \\ E \vdash e_2 : v_2 \end{array}}{E \vdash e_1 + e_2 : v_1 + v_2}$$

  - Minus

$$\frac{\begin{array}{c} E \vdash e_1 : v_1 \\ E \vdash e_2 : v_2 \end{array}}{E \vdash e_1 - e_2 : v_1 - v_2}$$

Semantics of IMP1 Cont.

- On to the semantics of Predicates:
- Question: What do predicates evaluate to?
- Answer: True and False
- Therefore, the result (value after colon) in operation semantics rules for predicates is a boolean

Semantics of IMP1 Cont.

- Here are operational semantics for predicates in IMP1
  - Less than or equal to:

$$\frac{\begin{array}{c} E \vdash e_1 : v_1 \\ E \vdash e_2 : v_2 \\ v_1 \leq v_2 \end{array}}{E \vdash e_1 \leq e_2 : \text{True}}$$

$$\frac{\begin{array}{c} E \vdash e_1 : v_1 \\ E \vdash e_2 : v_2 \\ v_1 \not\leq v_2 \end{array}}{E \vdash e_1 \leq e_2 : \text{False}}$$

  - Or (slightly imprecise) shorthand

$$\frac{\begin{array}{c} E \vdash e_1 : v_1 \\ E \vdash e_2 : v_2 \end{array}}{E \vdash e_1 \leq e_2 : v_1 \leq v_2}$$

  - What about the other predicates?

Semantics of Statements

- Now, all we have left are the statements
- However, there is one big problem: Statements do not evaluate to anything!
- Instead, statements update the values of variables
- In other words, they change E!
- Therefore, the rules for statements will produce a new environment
- Specifically, they are of the form $E \vdash S : E'$
- Changing the environment is the technical way of having side effects in the language

Semantics of Statements Cont.

- Let's start with the sequencing statement $S_1; S_2$:

$$\frac{\begin{array}{c} E \vdash S_1 : E_1 \\ E_1 \vdash S_2 : E_2 \end{array}}{E \vdash S_1; S_2 : E_2}$$

- Observe here that S_1 produces a new environment E_1
- We then use this new environment to evaluate S_2 and return E_2

Basic Statements

- Here is the assignment statement

$$\frac{\begin{array}{c} E \vdash e : v \\ E' = E[id \leftarrow v] \end{array}}{E \vdash id = e : E'}$$

- Observe that it is possible that id already had a value in E
- In this case, this rule overrides the value of id with the current value

Semantics of the Conditional

- Here are operational semantics of the conditional

$$\frac{\begin{array}{c} E \vdash C : true \\ E \vdash S_1 : E' \end{array}}{E \vdash \text{if}(C) \text{ then } S_1 \text{ else } S_2 \text{ fi} : E'}$$

$$\frac{\begin{array}{c} E \vdash C : false \\ E \vdash S_2 : E' \end{array}}{E \vdash \text{if}(C) \text{ then } S_1 \text{ else } S_2 \text{ fi} : E'}$$

- Observe that there are two different proof rules used.
- Expressions and conditionals return values, while statements return environments

Semantics of the While loop

- Let's finish with semantics for the last statement: While loop
- This is tricky because the loop may execute any number of times
- Let's start with the base case where the predicate is false:

$$\frac{E \vdash C : false}{E \vdash \text{while}(C) \text{ do } S \text{ od } : E}$$

Semantics of the While loop Cont.

- Now, what about the case where the condition is true?
- In this case, we want to:
  - Execute one iteration of the loop, producing a new environment E'
  - Repeat the evaluation of the loop with E'
- Here is the rule to do just that:

$$\frac{\begin{array}{c} E \vdash C : true \\ E \vdash S : E' \\ E' \vdash \text{while}(C) \text{ do } S \text{ od} : E'' \end{array}}{E \vdash \text{while}(C) \text{ do } S \text{ od } : E''}$$

Semantics of the While loop Cont.

$$\frac{\begin{array}{c} E \vdash C : true \\ E \vdash S : E' \\ E' \vdash \text{while}(C) \text{ do } S \text{ od} : E'' \end{array}}{E \vdash \text{while}(C) \text{ do } S \text{ od } : E''}$$

- Question: How does this rule make progress?
- Answer: It uses the new environment E' when reevaluating the loop body
- Is it possible that this rule does not terminate? Yes, if the loop is non-terminating

Putting it all together

- We saw how to give operational semantics for a simple imperative language
- Key difference: Side effects
- Side effects are encoded in operational semantics by producing a new environment
- Also observe that for imperative languages, all expressions always evaluate to concrete values