ECS 240: Finals

Sugeerth Murugesan

June 6, 2014

1 Apple programming language – Swift

The features that I find very interesting in swift is that:

- Elimination of 2-3 lines of code for the swift programmign language. Like python, Swift seems to be a terse language which eliminates extrneous lines of code to produce the same effect. For example, the initialization of an array in swift requires a like python like syntax.
- Use of dynamic type inference : Swift uses type inference extensively. This takes many type related syntax out of the developers to focus more on logic rather than syntax. The type inference can also flow in the oopposite direction, meaning that one can explicitly define the type of the variable.
- The pattern feature : One of the frequently occuring concepts in programming language is having data patterns in variables and constant declarations. The interesting pattern that swift has is the expression pattern. The expression is compared with the value of the input expression by overloading the $\simeq$ operator in swift.

2 Olio

2.1 Turing complete :

NOTE: I used the wiki page of single-typed lambda calculus as a reference.

A simple-typed lambda calculus is not Turing-complete – meaning it cannot simulate a single-taped Turing machine. As we know, the simply typed lambda caculus is strongly normalizing – meaning all the terms of the simply-typed lambda calculus can be reduced to a term in lambda caculus. One can decide whether or not a simply typed lambda calculus halts. Based on these facts,one can conclude that simply-typed lambda calculus is not Turing complete. Untyped calculus allows us to to represent recursive data structures while it is not possible in singly typed lambda calulus.

2.2 Covariant Array subtyping :

NOTE: I refered a generic Java Hack document from MIT.

One of the "loophole" in covariant array subtyping is that it leads to runtime errors. Example Java code:

```
ClassA[] objectA = new ClassA[1];
objectA[0] = new ClassA("arguments");
Object[] Obj = new ClassA(new Integer(123));
String newS = objectA[0].Variable
```

Here, when we try trace the program, the compiler lets us know that ClassA¡String¿[] becomes a subtype of object[], which is incorrect. Hence, a loophole in Java subtying.

2.3 Statically typed languages

Most of the statically typed languages are not formally safe. Programmers tend to write programs that "fool" the static type checker. One example is using type punning.

Pointer aliases point to the same memory location.

```
int i = 4;
long newVar = *( long )&(i);
```

The newVar will compile, but provide undefined results.

```
ClassA objectA = new ClassA();
ClassA objectB = new ClassA;
```

Although, both the statements are syntactically correct and compile, the results are undefined. The objectB has undefined results.

3 Operational semantics

3.1 Small step and Big-step Semantics

Note: I am assuming that the functions f_1 and f_2 are already defined and the $compose(c_1, c_2)$ have a meaning at this point, similiar to how the other operations have meaning with them.

c :: = $compose(c_1, c_2)$

where c_1,$c_2 \in$ Com

The Big-step semantics for the "compose" function is:

(definition of compose rule)

$$\frac{< c_2(i), \sigma > \Downarrow \sigma_1(i) \;\; < c_1, \sigma(i) > \Downarrow \sigma'}{< (compose(e_1, e_2)), \sigma > \Downarrow \sigma'}$$

This evaluation rule executes the command c_2, and then c_1, it is similiar to c_2 ; c_1. The possible error condition for the Big step semantics is that if the f_2 has a dependency towards f_1, the Big -step operational semantics will not work.

Small step semantics

$$< (compose(e_1, e_2)), \sigma > \rightarrow < c_2(i); c_1, \sigma' >$$

3.2 Counter Example

Yes, the expression is similiar to $c_2; c_1$. But, this is not possible for a certain case where a particular state in c_2 is being assigned to another variable in c_1.

If command c is taken into consideration, we assume that the function f is equalent to a command.

The state changes caused by c_2 is propagated to c_1. As a result, $compose(c_1, c_2)$ is equalent to $c_2; c_1$.

4 Abstract interpretation

- It boils down to all possible values of x at a particular location of program considering all branches. Yes we can use the collecting semantics.
- Abstract Domain : -
  $\{True, False, \top, \bot\}$

  $\top$ is a condition where the value of P(x) is staticaly variable between True and False. Concretizes to { True, False }

  $\bot$ P(x) changes the values during the program execution hence, it is dynamic in nature. Concretizes to

  Concretization function : { True, False }

  The domain : set of values x is assigned to.
- It is same as the collecting semantics for sign analysis.
- It is non-terminating as there are finite locations in the program while the functions being non-terminating.

5 Axiomatic Semantics

5.1 Loop invariants

Loop invariant t = $\sum_{i=n}^{N} i$ - (N - n+1),
if we try to extrapolate the invariant it boils down to t = N*N - n^2
{ n = N > 0 }
t := 0;
{ n ≥ 0 ∧ t = 0 } P_1
while n ≠ 0 **do**
{ t = $N * N - n^2$) ∧ N > n ≥ 0 } P_2
t:= t + (2*n - 1);
{ t = $N * N - (n-1)^2$ ∧ N > (n-1) ≥ 0 } P_3
n:=n-1;
end while { t = N * N}

Proof Obligations
{ n ≥ 0 ∧ n=N } t : = 0 { P_1 }
{ P_1 } → { P_2 }
{ P_2 ∧ (n ≠ 0) } t : = t +(2*n -1); { P_3 }
{ P_3 } n= n-1; { P_2 }
{ P_2 ∧ ¬(n ≠ 0)} → {t : = N*N }

Proof:
When we take t=0 into account the value of n^2=N*N
For the case n : = n-1, we have,
$(n-1)^2$+t=N*N
$(n)^2$-2*n+t=N*N

P_2=$(n)^2$-2*n+t=N*N
Using assignment inference

t := t + (2 * n - 1) in $(n)^2$-2*n+t=N*N gives -

$n^2 + t = N * N$

Evaluates to be

$P_2 \wedge (n \neq 0)$

Preconditions

{ P } *q := *p + *q { *p = *q }

Deriving from the post condition we can say that the value of "q" if it is equal to value of "p" then *q = 0.

*q-*q=*p

*p=0

Because of the post condition where (*p = *q) we have,

p =(*p=0)∧(*q=0)

6 How to reach here

Note: I referred to the wikipedia page of Data-flow analysis. The different types of program analysis and reasoning methodologies are :

- Linear chaining of variables, If one knows all the locations of the program where a particular variable will be accessed, then if there is an arbitary input i assigned to a variable i. We can definitely say using the pre-condition. The other methods that can be incorporated are symbolic execution, where iteratively one determines symbolic values instead of the actual values itself.(**NOTE:** I have referred to wiki page of Symbolic execution,)[**?**]
- Flow sensitive analysis, where the order of statements are taken into account. So for this example, if we can let the compiler know that variables X and Y goto a particular location after statement n then the problem boils down to finding the inputs with the overall knowledge of location l.
- Liveness analysis and control graph, determining the sections where one particular value assigned to a variable is found out or traced.

References

[1] http://www.cis.upenn.edu/ bcpierce/sf/HoareAsLogic.html *Hoare's Logic*

[2] http://www.cis.upenn.edu/ bcpierce/sf/HoareAsLogic.html *Hoare's Logic*

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley *Enhancing Symbolic Execution with Veritesting*