**COMP50006 Compilers**

## Exercise 7: Points-to analysis

In this exercise we explore a dataflow analysis for pointers. We begin with a Haskell data type for instructions for a simple machine with registers:

```
data Instr = Define String   -- "label:"
           | Mov Reg Reg     -- "mov.l xxx yyy" (yyy:=xxx)
           | Cmp Reg Reg     -- "cmp.l xxx yyy" (test yyy-xxx)
           | Bgt String      -- "bgt label" (branch if greater than zero)
           | New Int Reg     -- "yyy = malloc(xxx)" (storage allocation)
data Register = D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7
```

For the purposes of this question, an extra instruction, called `New`, has been added. For example, this instruction, with identifier *id*,

```
id:     New 8 D0
```

allocates 8 bytes of memory and sets register `D0` to point to it. Our goal is to define a dataflow analysis that computes, for each `Register`, the set of identifiers of the allocations that the register might point to. For example, given:

```
1:     New 8 D0
2:     New 8 D3
3:     Cmp D1 D2
4:     Bgt L
5:     Mov D0 D3
6:  L:
```

then, at line 6, we can say that `D0` points to the allocation at line 1, and `D3` may point to either the allocation at line 1 or the allocation at line 2. We represent this as a points-to-set { (D0, 1), (D3, 1), (D3, 2) }. The effect of an instruction on the points-to-set before its execution depends on the instruction; we define a function `effect`:

```
effect ::  PointsToSet → CFGNode → PointsToSet
effect pts (Node id (Cmp r1 r2)) = pts
effect pts (Node id (Bgt label)) = pts
effect pts (Node id (New n r)) = pts ∪ {(r, id)}
effect pts (Node id (Mov r1 r2)) =
```

*(i)* Complete the missing definition of `effect` above, for `Mov`.

*(ii)* Write down the defining equation for *pointsIn(n)*, the points-to-set just before node *n* of the control-flow graph, and the defining equation for *pointsOut(n)*, the points-to-set just after node *n*.

*(iii)* Show how `effect` can be improved by enhancing the rule for `New`.

*(iv)* What do you think points-to information might be used for in an optimising compiler? Could you also use it for static detection of software defects?

*Paul Kelly*   *Imperial College*   *February 2024*

## Exercise 7: Points-to analysis - solution

*(i)* : We need to add points-to relations saying that `r2` might now point to anything `r1` might point to:

```
effect pts (Node id (Mov r1 r2)) = pts ∪ [(r2, id) | (r1,id) ← pts]
```

Actually we also know that after this move, r2 no longer points to what it pointed to before, so we can remove its targets first:

```
removeTargets r1 pts = [(r2, id) | (r2, id) <- pts, r1 != r2]

effect pts (Node id (Mov r1 r2))
   = (removeTargets r2 pts) ∪ [(r2, id) | (r1,id) ← pts]
```

*(ii)*

$$pointsIn(n) = \bigcup_{p \in pred(n)} pointsOut(p)$$

$$pointsOut(n) = effect(pointsIn(n))(instruction_n)$$

*(iii)* The rule for `New` does not account for the points-to elements that are killed by the assignment. To do better we need to remove them:

```
effect pts (Node id (New n r))
   = pts’ ∪ {(r, id)}
   where pts’ = [(reg,t) | (reg,t) ← pts, reg ≠ r]
```

*(iv)* Points-to analysis can be used to determine whether two pointers might point to the same location - which is important for understanding dataflow and potential dependence or interference. It can also be used to determine whether a pointer can escape from the method in which it allocated - and more generally, to automate storage reclamation.

You might also wonder whether pointer analysis might enable you to reason about which parts of the code might be able to reach private (or vulnerable) data.

## Follow-up things to think about

- This exercise has dodged a huge issue in pointer analysis: tracking pointer variables *in the heap*. This is pretty important for many of the things we might care about - such as the question of what the pointers in a linked list might point to.

  This question concerns instructions that dereference heap pointers. We might add indirect load and store instructions:

  ```
  LoadIndirect Reg Reg "mov.l (xxx) yyy" (yyy:=*xxx)
  StoreIndirect Reg Reg"mov.l xxx (yyy)" (*yyy:=xxx)
  ```

  (let’s assume, as some pointer analysis research work assumes, that we aggregate all the possible pointer values in any field of a heap object. “Field-sensitive” pointer analysis adds this refinement back in).

  Now the compiler needs an abstract model not only of what registers might point to — but also what (abstract) heap cells might point to as well. This leads us to a graph representation, with heap cells as nodes and edges representing "might hold a pointer to".

This lies beyond the scope of this course — in fact beyond all three of the textbooks we've been referring to. There are many (*many!*) papers on this. This one (the work of my PhD graduate David Pearce) provides a nice introduction:

> David J. Pearce, Paul H.J. Kelly, and Chris Hankin. **Efficient field-sensitive pointer analysis of C**. *ACM Trans. Program. Lang. Syst. 30, 1* (November 2007), DOI:`https://doi.org/10.1145/1290520.1290524`

*Paul Kelly*    *Imperial College*    *February 2024*