CSSE4630 Assignment One: Pointer Analysis

Mark Utting

23 version 1.0

1 Introduction

This assignment is focussed on several kinds of Pointer Analysis. You will implement the Andersen and Steensgaard algorithms for pointer analysis and compare the results. This assignment is worth 20% of your final mark for this course.

2 School Policy on Student Misconduct

This assignment is to be completed individually. You are required to read and understand the School Statement on Misconduct, available on the School's website at: `https://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct`

3 Understanding Steensgaard's Algorithm with Pictures

Read the `examples/ptr7.tip` program and then draw the union-find graph for this program, using Steensgaard's algorithm rules (Section 11.3 of the textbook).

I suggest you arrange all the variables in four columns (a, b, c, d). The only rule you will need for this program is:

$$X = \&Y \quad \longrightarrow \quad [\![X]\!] = \uparrow [\![Y]\!]$$

Make sure in your union-find diagram that you clearly distinguish the $\uparrow [\![Y]\!]$ terms from the $[\![Y]\!]$ terms. I suggest you use a rectangle for the former, with a circle inside it for the latter. So $\uparrow [\![Y]\!]$ will be drawn as a rectangle for the whole term, with a circle inside for the $[\![Y]\!]$ subterm.

Make sure you follow the correct union-find steps to unify two terms ($Unify(A, B)$) — see Section 3.3 (Solving Constraints with Unification) of the textbook if you need more detail.

1. follow the *parent* pointers of each term as far as you can, to see if they are already equal?
2. If they are not equal, then add an arrow from one to the other (always point the arrow from from variables to $\uparrow$ terms if possible).
3. If you have just added an equality arrow between two pointer terms, $\uparrow C$ and $\uparrow D$, then you must also go inside those $\uparrow$ terms and also unify C and D.

4 Implementing Steensgaard's Algorithm [20%]

The TIP system does not fully implement Steensgaard's pointer analysis algorithm yet. To see this, run the following command and read the exception it generates:

```
tip -steensgaard examples/ptr7.tip
```

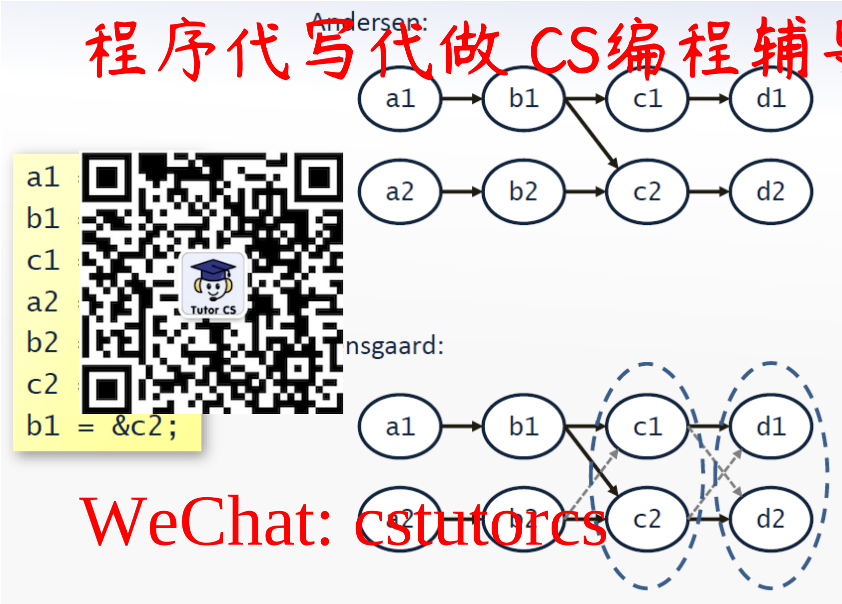

Figure 1: Visualisation of Andersen versus Steensgaard results for ptr7.tip.

Implement the five missing cases in the `visit` method in `SteensgaardAnalysis`. See Section 11.3 (Steensgaard's Algorithm) of the textbook for the rules you should implement.

Hints:

1. The `solver` is already created for you, so you can just call its `unify` method.
2. The *address-of* constructor in the textbook (`&X`) corresponds to `PointerRef(_)` in the Scala code (down the bottom of the file).
3. You should use the `identifierToTerm` method to convert an identifier into an abstract term that represents a set of pointers, and `allocToTerm` to turn an 'alloc' expression into an abstract term.
4. To allocate a fresh variable (called α in the textbook) use the `FreshVariable()` function.

4.1 Checking your results

Use your Steensgaard algorithm to analyse the `examples/ptr7.tip` program. You should see standard output being printed that includes the following results:

```
[info] Points-to:
c1[3:25] -> { d1[3:33],d2[3:37] }
b1[3:17] -> { c1[3:25],c2[3:29] }
a2[3:13] -> { b2[3:21] }
d1[3:33] -> {  }
a1[3:9]  -> { b1[3:17] }
b2[3:21] -> { c1[3:25],c2[3:29] }
d2[3:37] -> {  }
c2[3:29] -> { d1[3:33],d2[3:37] }
```

Figure 1 shows a nice visualisation of these results, from the Week 5 lecture slides. Note how Steensgaard's analysis merges the $c1$ and $c2$ into one set, and therefore must also merge $d1$ and $d2$ into one set.

5 Implementing Andersen's Algorithm [20%]

Next we want to implement the Andersen algorithm, which is slower than Steensgaard's algorithm but can return much more accurate results on some programs.

The TIP system do[illegible]nt Andersen's pointer analysis algorithm yet. To see this, run the followi[illegible]d the exception it generates:

```
tip -andersen[illegible]p
```

Implement the five r[illegible]isit method in `AndersenAnalysis`. See Section 11.2 (Andersen's Algorit[illegible]or the rules you should implement.

Note that in `Anders[illegible]`, a cubic solver is already available (called `solver`) and the set of all Cells in the program is available in `cells`.

6 Comparing Pointer Analyses [20%]

In this part, we want to compare the accuracy of the points-to results from the Steensgaard and Andersen algorithms. Download the provided *.tip test programs from Blackboard and save them in a folder called `tests`. Then for each test program, run *both* of your analyses and save the resulting output into two separate files whose names end with `.andersen.txt` and .steensgaard.txt respectively.

Then use Microsoft Word or some other editing tool to draw the points-to graph produced by the Andersen algorithm (with black arrows), and overlay on that same graph the extra points-to arrows (as red dashed arrows) produced by the Steensgaard algorithm. You should also add dashed red ellipses (or rectangles if you prefer) around the nodes that the Steensgaard algorithm combines into a single node. The resulting graph should look similar to the second graph in Fig. 1 (if your input file was `examples/ptr7.tip`).

Below the graph, add a text box with your explanation of why the results are different for this particular program, or why they are the same. Save your resulting comparison graph and explanation as a PDF file.

For example, given an input file called `test1.tip`, you should create three separate files:

- `tests/test1.andersen.txt` - a text file containing all output from your run of the command: tip -andersen -normalizepointers test1.tip.
- `tests/test1.steensgaard.txt` - a text file containing all output from your run of the command: tip -steensgaard -normalizepointers test1.tip.
- `tests/test1.pdf` - containing your comparison points-to graph and your explanation of any differences between the Andersen and Steensgaard results.

7 Write Circular Data Structure Programs [20%]

In this part we want to explore how these two pointer analysis algorithms handle circular data structures.

Create a directory called `circular`. In that directory write at two short TIP programs that create circular pointer data structures with at least THREE nodes in the loop:

- one program (called `circular/same.tip` that gives the same points-to graph with both the Andersen and Steensgaard algorithms.

- one program (called circular/diff.tip that gives different points-to graphs with the Andersen and Steensgaard algorithms.

For each of these two programs, run both your analyses, save the output into *.andersen.txt and *.steensgaard.txt respectively, draw the points-to graphs and save into *.pdf, as described in the previous part [illegible]

8 Advanced An[illegible] with Records [20%]

This is a more diffic[illegible]ment that you should attempt only after completing all previous parts. T[illegible] your Andersen algorithm so that it can also analyse TIP programs that use records. To simplify the task somewhat, we will focus on immutable records, so you do not have to implement the field write assignments ($r.f = E$ or $(*r).f = E$). You only need to implement the field read expression ($E.f$) and the record allocation expression ($alloc\{f_1 : E_1, \ldots, f_n : E_n\}$) — if this is used without the `alloc` keyword it allocates record values on the stack instead of the heap. You can assume that the values of record fields cannot themselves be records.

For example, we want to analyse programs like examples/record2.tip:


```
main() {
    var n, r1;
    n = alloc {p: 4, q: 2};
    *n = {p:5, q: 6};
    r1 = *n.p; // output 5
    return r1;
}
```


Implement the field-sensitive extension of Andersen's analysis for TIP discussed in the last two pages of Section 11.2 of the textbook (see also Section 3.4 for more examples of record values and types).

9 Submission

Submit your assignment via Gradescope, as a single zip file containing your whole tip directory (with subdirectories called `src`, `tests`, `circular` etc.). The `src` subdirectory will include your modified Scala source files that implement your pointer analysis algorithms. The `tests` subdirectory should include your three output files for each *.tip test file, and the `circular` subdirectory should contain your three `same.*` and three `diff.*` files.

You can submit multiple times before the deadline to check that your submission has the correct structure and that the provided tests work correctly. It is your last submission before the deadline that will count. It will be tested using additional test programs and marked for correctness and elegance.