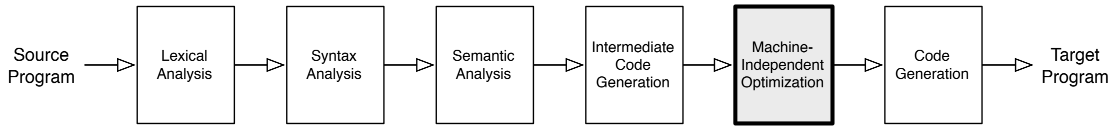
CS 301 Spring 2019
Tutorial Assignment
16 April

Data-Flow Analysis Framework

1 Plan



This week, we practice with data-flow optimizations on Roost programs and develop theoretical foundations for a general approach to data-flow analysis. Understanding this theory will help us build a single data-flow engine for our Roost compiler and implement several optimizations using this one engine. It will also provide a relatively direct way to reason about analyses' correctness, termination, and precision.

2 Readings

- Dragon 9.3 – 9.5.2.
  - Focus on definitions and examples in 9.3 – 9.3.3 and the characterization of Constant Progation in 9.4. Skim proofs and discussions of precision.
  - Interpret "the family, F, of transfer functions" to mean a collection of transfer functions, one per basic block. For the analyses we explore, this "family" is all derived from the same pattern, e.g., for Reaching Definitions, $gen_B \cup (\text{IN}_B - kill_B)$ for a given basic block B. We happen to use many analyses that deal with sets to track information about many variables at once, but we could also consider analyses dealing with simpler atomic values.
- Alternative: for a second perspective on specific data-flow analyses from last week and this week, see EC 8.6.1, 9.1 – 9.2, 10.3

3 Exercises

1. In this exercise, we apply data-flow analyses to Roost programs to practice in preparation for implementing a data-flow engine and a few in our compiler (next project phase).
   (a) For each of the independent single-instruction basic blocks below, describe the *e_gen*, *e_kill*, *def*, and *use* sets:
      i. `t = a[i]`
      ii. `b[j] = s`
      iii. `x = p.g`
      iv. `o.f = y`
      v. `z = q.m(r)`
   (b) Optimize each of the following TAC codes. (We provide the original Roost code for context.) Be sure to use common subexpression elimination, copy propagation, and dead code elimination, and optionally others. What sequence of optimizations did you apply?

i.
```
struct A {
  field f: i64
  field g: i64
}
fn boom(a: A, z: i64) -> A { ... }
fn f(a: A) -> i64{
   let v = a.f
       w = a.g
       b = boom(a, v) in {
     b.f = w;
     (a.f + b.f) * a.g
   }
}
```

```
fn f(a: A) -> i64 {
  v = a.f
  w = a.g
  b = boom(a, v)
  b.f = w
  t1 = a.f
  t2 = b.f
  t3 = t1 + t2
  t4 = a.g
  t5 = t3 * t4
  return t5
}
```

ii.
```
fn f(b: int[]) -> unit {
   let i: i64
       j: i64
       k: i64 in {
     ... // any code
     x = b[i] + b[j];
     b[j] = b[i] + b[j];
     b[i] = b[i] + b[j];
   }
}
```

```
fn f(b: int[]) -> unit {
  t1 = b[i]
  t2 = b[j]
  x = t1 + t2

  t3 = b[i]
  t4 = b[j]
  t5 = t3 + t4
  b[j] = t5

  t6 = b[i]
  t7 = b[j]
  t8 = t6 + t7
  b[i] = t8
}
```

2. Given $P = \{\text{red}, \text{blue}, \text{yellow}, \text{purple}, \text{orange}, \text{green}\}$, let us define the partial order $\sqsubseteq$ as follows:

$$\begin{aligned}
\text{red} &\sqsubseteq \text{purple} \\
\text{blue} &\sqsubseteq \text{purple} \\
\text{yellow} &\sqsubseteq \text{orange} \\
\text{red} &\sqsubseteq \text{orange} \\
\text{blue} &\sqsubseteq \text{green} \\
\text{yellow} &\sqsubseteq \text{green} \\
p &\sqsubseteq p \quad \text{for all } p \in P
\end{aligned}$$

(a) Draw the lattice diagram for P based on this partial order. (See Dragon Figure 9.22 for an example.)

(b) Add purple $\sqsubseteq$ yellow and purple $\sqsubseteq$ green. Is this still a partial order? Why or why not? What if we also add yellow $\sqsubseteq$ red?

(c) Now add two colors to the original P (without the proposed additions above):

$$\begin{aligned}
\text{white} &\sqsubseteq p && \text{where } p \in \{\text{red}, \text{blue}, \text{yellow}\} \\
p &\sqsubseteq \text{black} && \text{where } p \in \{\text{purple}, \text{orange}, \text{green}\}
\end{aligned}$$

i. Draw the resulting semilattice.

ii. Compute all lower bounds for $\{\text{black}, \text{purple}, \text{orange}\}$.

iii. Compute the greatest lower bound for $\{\text{black}, \text{purple}, \text{orange}\}$.

iv. Which of the following functions $f_i : P \to P$ are monotone?

$$f_1(p) = \begin{cases} \text{white} & \text{if } p \in \{\text{red}, \text{blue}, \text{yellow}\} \\ p & \text{otherwise} \end{cases}$$

$$f_2(p) = \begin{cases} \text{yellow} & \text{if } p \in \{\text{red}, \text{blue}, \text{green}\} \\ p & \text{otherwise} \end{cases}$$

$$f_3(p) = \begin{cases} \text{orange} & \text{if } p \text{ contains the letter } a \\ \text{blue} & \text{otherwise} \end{cases}$$

3. Design a data-flow analysis to detect *unreachable code* that cannot possibly execute in any execution of the program. For example, your analysis should recognize the following specific unreachable code:

```
fn f() -> i64 {
  let x = 7 in {
    return x;
    Roost.println("unreachable"); // unreachable
    7 * x // unreachable
  }
}

fn g() -> i64[] {
  let a = new int[4] in {
    let i = 0 in
    while (i < a.length) {
      a[i] = 2*i;
      i = i + 1;
      break;
      a[i] = a[i] * a[i]; // unreachable
    }
    a // reachable
  }
}

fn h(x: i64, y: i64) -> i64 {
  if (x < y) {
    return x;
  } else {
    return y;
  }
  Roost.println("unreachable"); // unreachable
  -1  // unreachable
 }
```

Your analysis is not required to recognize:

```
fn m() -> unit {
  let a = new int[4] in {
    let i = 0 in
    while (i < a.length) {
      a[i] = 2*i;
    }
  }
  Roost.println("unreachable"); // unreachable
}
```

```
fn n() -> unit {
  while (true) { }
  Roost.println("unreachable"); // unreachable
}
```

(a) Design the data-flow analysis by describing the following:

- The direction D (forward/backward) of your analysis.
- The domain V of data-flow values.
  Unlike other analyses, these values will not be sets themselves. In other words, while other analyses we have described use products, this analysis uses atomic values.
- The meet operation $\wedge$ and the order $\leq$ induced on V by the meet operator. Draw the associated lattice diagram for the elements of V.
- The set of transfer functions F, where $f_I \in F$ is the transfer function for TAC instruction I. Show a transfer function for an instruction I only if it is not the identify function. (Most will be the identity function.)
- The initial value for either OUT[ENTER] or IN[EXIT], depending on whether your analysis is forward or backward.

(b) Is your framework monotone? Distributive? Explain.

(c) Consider: What are some limitations of your analysis? In what ways is it conservative? Can you imagine another analyses that could provide extra information that could support less conservative (but correct) identification of unreachable code?

4. (a) Draw the product of the following two lattices:

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & & & 0 \arrow[d] \\
B \arrow[dr] & & C \arrow[dl] & & 1 \arrow[d] \\
 & D & & & 2
\end{tikzcd}$$

(b) Suppose V is the set of all subsequences of "wee: { wee, we, ee, w, e, ϵ }. Let $x \wedge y$ be the longest common subsequence of x and y.

i. Draw the lattice diagram for V.

ii. Consider the function $f : V \to V$, where $f(x)$ is x with all e's removed. Is f distributive? Why or why not?

5. Design an optimization to remove redundant run-time `null` checks. For simplicity, you may assume that each basic block contains a single TAC instruction.

(a) Design your data-flow analysis by describing the following:

- The direction D (forward/backward) of your analysis.
- The domain V of data flow values.
- The meet operation $\wedge$. Describe the order $\leq$ induced on V by your meet operator.
- The set of transfer functions F, where $f_I \in F$ is the transfer function for TAC instruction I. You only need to consider the following forms: `x = y`, `x = new C()`, `check_null x`, and `x = null`.
- v, the initial value for either OUT[ENTER] or IN[EXIT], depending on whether your analysis is forward or backward.

Dragon Figure 9.21 is a good baseline reference. Consider similarities between your analysis and those we have studied.

(b) Is your framework monotone? Distributive? Explain.

(c) How would you use results from the analysis to optimize a program?

(d) Show the results of your analysis and optimization on the following:

```
    y = new A
    check_null y
    z = w
  Loop:
    cjump ... EndLoop
    check_null y
    check_null z
    jump Loop
  EndLoop:
  cjump ... EndIf
    y = k
    check_null y
    check_null z
  EndIf:
    check_null y
    x = y
    check_null x
```

(e) In Dragon, page 629, the authors claim that for a lattice-theoretic partial order $\leq$:

- Any answer greater than IDEAL is incorrect.
- Any answer smaller than or equal than IDEAL is conservative, i.e., safe.

Explain in one or two sentences why your analysis never yields incorrect results. Is your analysis *strictly* conservative in the sense that it sometimes computes answers strictly smaller than IDEAL? If not, explain why in one or two sentences. If it is strictly conservative, write a short program for which your analysis computes a different value than IDEAL. What are the consequences of being conservative in this case?

6. Consider an optimization to eliminate redundant array bounds checks, i.e., those that are provably unnecessary. What information must you know to show that a given array bounds check is redundant? Do our existing analyses capture enough information? Could we design others? (No need to show formally, but give it some thought.) Here is an example to consider:

```
let len = array.length
    i = 0
    sum = 0 in {
  while (i < len) {
    sum = sum + array[i];
    i = i + 1;
  }
  // ...
}
```