NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 21 part 1: Available expressions analysis

Michael Engel

Overview

- Data-flow analyses
  - Global properties of expressions
  - Expressions and availability
  - Semantic vs. syntactic analysis
  - Available expressions analysis

Discovering global properties of expressions

- We look at analyses for eliminating **redundant computations** of expressions
- Programs may contain code whose result is needed, but in which some computation is simply a redundant repetition of earlier computation within the same program
- Our first analysis – **available expressions** – involves **replacing an expression** by its precomputed value:
  - Given a program point **u**, this analysis discovers the expressions whose results at **u** are same as the their previously computed values regardless of the execution path taken to reach **u**

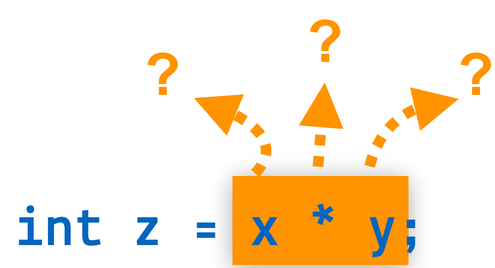
Expressions and availability

- Any given program contains a finite number of expressions (i.e. computations which potentially produce values), so we may talk about the ***set of all expressions*** of a program:

```
int z = x * y;
print s + t;
int w = u / v;
```

program contains expressions
`{x*y, s+t, u/v}`

- **Availability** is a data-flow property of expressions:
"Has the value of this expression already been computed?"



Availability

- At each instruction, each expression in the programis either available or unavailable
- We consider availability from an instruction's perspective: each instruction (or node of the CFG) has an associated ***set of available expressions:***

```
    int z = x * y;
    print s + t;
n:  int w = u / v;      avail(n)={x*y, s+t}
```

- This is all familiar from live variable analysis
- Expression availability and variable liveness share many similarities
  - both are simple data-flow properties
  - however, they do differ in important ways

Semantic vs. syntactic analysis

- Availability differs from earlier examples in a subtle but important way:
  - we want to know which expressions are ***definitely available*** (i.e. have already been computed) at an instruction, not which ones ***may be available***
- An expression is ***semantically available*** at a node `n` if its value gets computed (and not subsequently invalidated) along every execution sequence ending at `n`

```
int x = y * z;
      :
return y * z;
```

`y*z` **available**

```
int x = y * z;
      :
y = a + b;
      :
return y * z;
```

y invalidated here

`y*z` **unavailable**

Semantic vs. syntactic analysis

- An expression is ***syntactically available*** at a node `n` if its value gets computed (and not subsequently invalidated) along every path from the entry of the CFG to `n`
  - semantic availability: ***execution behaviour*** of the program
  - syntactic availability: program's ***syntactic structure***

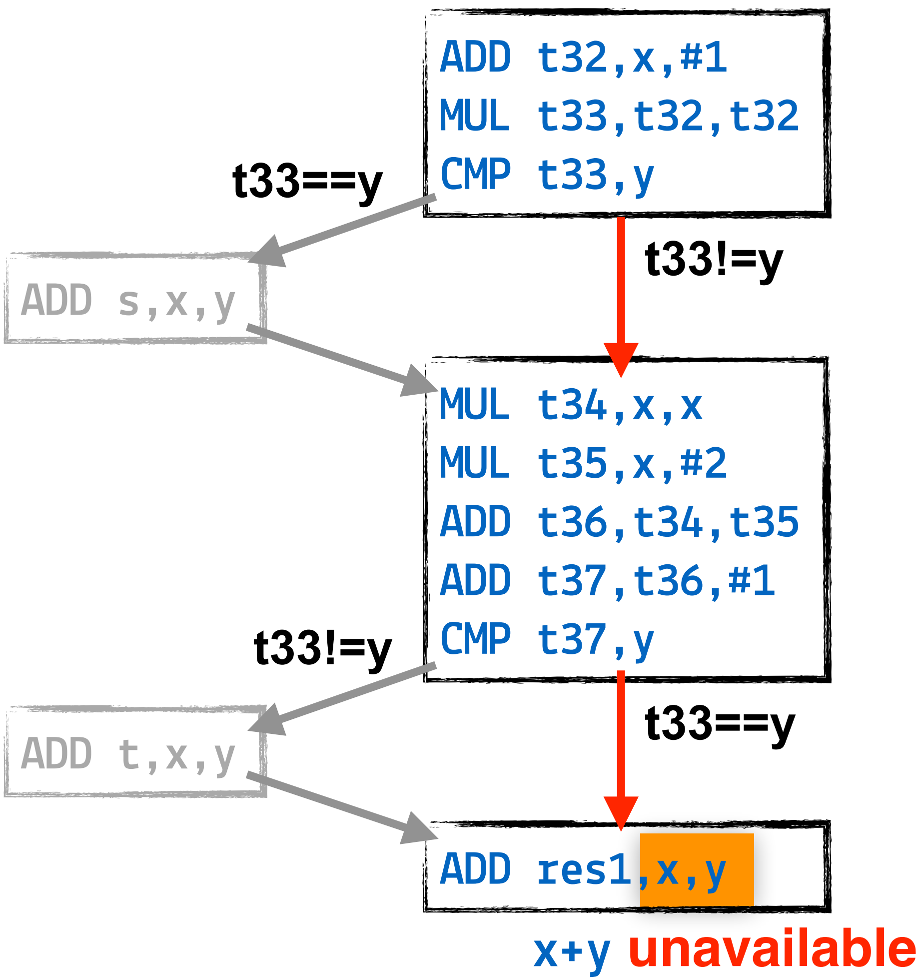
```
if ((x+1)*(x+1) == y) {
   s = x + y;
}
if (x*x + 2*x + 1 != y) {
   t = x + y;
}
return x + y;   // x+y available
```

Semantically:
one of the conditions will be true, so on every execution path `x+y` is computed twice

⇒ the recomputation of `x+y` is **redundant**

Semantic vs. syntactic analysis

Translation to IR representation:



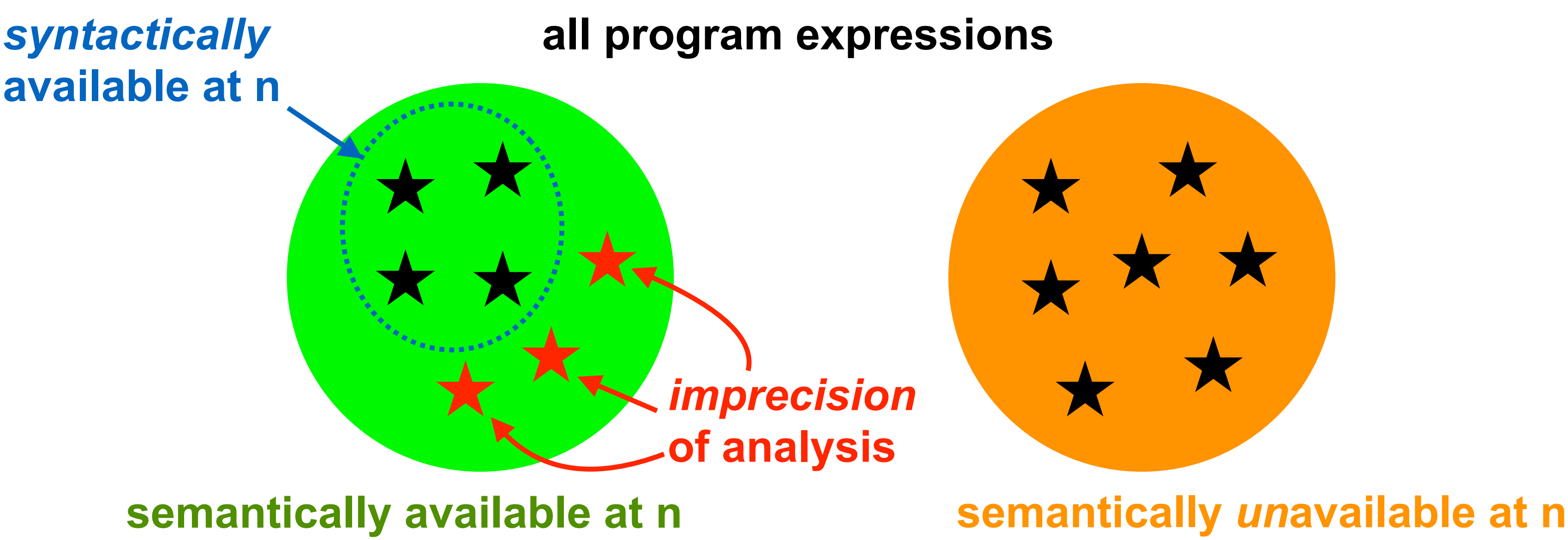

```
if ((x+1)*(x+1) == y) {
   s = x + y;
}
if (x*x + 2*x + 1 != y) {
   t = x + y;
}
return x + y;
```

On the **red** path through the flowgraph, `x+y` is never computed, so `x+y` is ***syntactically unavailable*** at the last instruction

Note that this **red** path ***never actually occurs*** during execution

Semantic vs. syntactic analysis

- If an expression is deemed to be available, we may do something ***dangerous*** (e.g. remove an instruction which recomputes its value)
- In contrast, with live variable analysis we found safety in assuming that more variables were live, here we find safety in assuming that fewer expressions are available



Available expression analysis

- Available expressions is a forward data-flow analysis: information from past instructions must be propagated forward through the program to discover which expressions are available

```
                 t = x * y;
print x * y;                       if (x * y > 0)

                int z = x * y;
```

- Unlike variable liveness, expression availability flows forwards through the program
- As in liveness, each instruction has an effect on the availability information as it flows past

Available expression analysis

- An instruction makes an expression **available** when it ***generates*** (computes) its current value

```
     {}
print a*b;        // Generate a*b
     {a*b}
c = d+1;          // Generate d+1
     {a*b,d+1}
e = f/g;          // Generate f/g
     {a*b,d+1,f/g}
```

Available expression analysis

- An instruction makes an expression **unavailable** when it ***kills*** (invalidates) its current value

```
  {a*b,c+1,d/e,d-1}
a=7;                // Kill a*b
  {c+1,d/e,d-1}
c=11;               // Kill c+1
  {d/e,d-1}
d=13;               // Kill d/e,d-1
  {}
```

Available expression analysis

- As in LVA, we can devise functions $\texttt{Gen}_n$ and $\texttt{Kill}_n$ which give the sets of expressions generated and killed by the instruction at node `n`
- The situation is slightly more complicated here:
  an assignment to a variable `x` kills ***all expressions in the program*** which contain occurrences of `x`
- In the following, E_x is the set of expressions in the program which contain occurrences of `x`:

```
Gen(x=3) = {}                 Gen(print x+1) = {x+1}
Kill(x=3) = Ex                Kill(print x+1) = {}
```

```
Gen(x=x+y) = {x+y}
Kill(x=x+y) = Ex
```

Available expression analysis

- As availability flows forwards past an instruction, we want to modify the availability information by adding any expressions which it generates (they become available) and removing any which it kills (they become unavailable)

```
      {y+1}
Gen(print x+1) = {x+1}
      {x+1,y+1}
```

```
      {x+1,y+1}
Kill(x=3) = Eₓ
      {y+1}
```

Available expression analysis

- If an instruction both generates and kills expressions, we must **remove the killed** expressions **after adding the generated** ones

`{x+1,y+1}`

`x = x+y;`

`Gen(x=x+y) = {x+y}`

`Kill(x=x+y) =` E_x

`{y+1}`

- If we consider `in-avail(n)` and `out-avail(n)`, the sets of expressions which are available immediately before and immediately after a node, the following equation must hold:

$$\texttt{out-avail(n)} = (\texttt{in-avail(n)} \cup \texttt{gen(n)}) - \texttt{kill(n)}$$

Available expression analysis

$$\texttt{out-avail(n)} = (\texttt{in-avail(n)} \cup \texttt{gen(n)}) - \texttt{kill(n)}$$

in-avail(n) = {x+1,y+1}

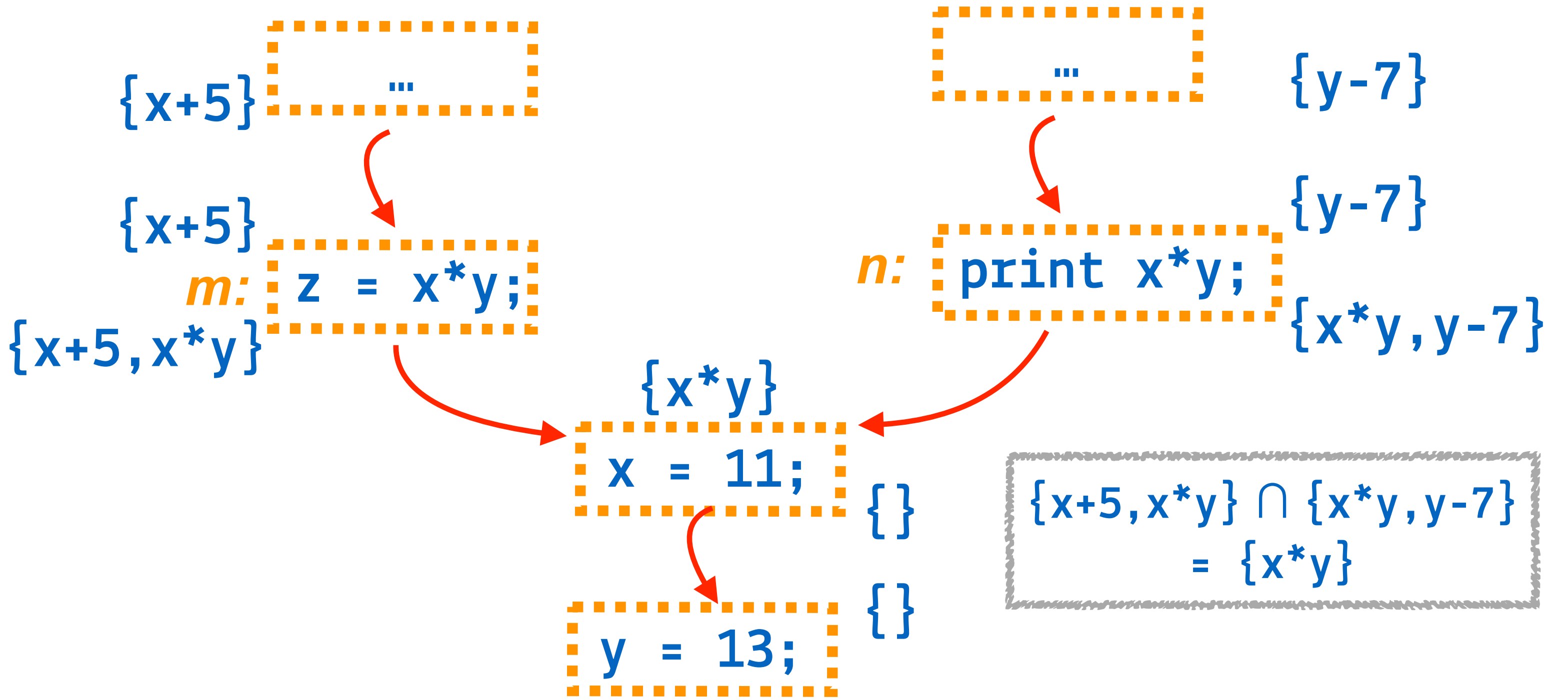
n: $\boxed{\texttt{x = x+y;}}$

out-avail(n) = (in-avail(n) ∪ gen(n)) - kill(n)
= ({x+1,y+1} ∪ {x+y}) - {x+1,x+y}
= {x+1,y+1,x+y} - {x+1,x+y}
= {y+1}

$\texttt{Gen}_n$ = {x+y} $\texttt{Kill}_n$ = {x+1,x+y}

Available expression: joined control flow

- When a node `n` has a single predecessor `m`, information propagates along the CFG edge as you expected: *in-avail(n)* = *out-avail(m)*
- When a node has multiple predecessors, the expressions available at the entry of that node are exactly those **expressions available** at the exit of **all of its predecessors** (cf. “any of its successors” in LVA)



Data-flow equations

- The data-flow equations for available expression analysis tell us everything we need to know about how to propagate availability information through a program:

$$\textit{in-avail}(n) = \bigcap_{p \in \mathtt{pred}(n)} \textit{out-avail}(p)$$

$$\textit{out-avail}(n) = (\ \textit{in-avail}(n) \cup \mathtt{Gen}_n\ ) - \mathtt{Kill}_n$$

- Each is expressed in terms of the other, so we can combine them to create one overall availability equation:

$$\textit{avail}(n) = \bigcap_{p \in \mathtt{pred}(n)} (\textit{avail}(p) \cup \mathtt{Gen}_n\ ) - \mathtt{Kill}_n$$

Available expressions equations

- An expression $\texttt{e} \in \texttt{Expr}$ is **available** at a program point $\texttt{u}$ if all paths from $\texttt{Start}$ to $\texttt{u}$ contain a computation of $\texttt{e}$ which is not followed by an assignment to any of its operands
- The data flow equations to define the analysis are:

$$\texttt{In}_n = \begin{cases} \texttt{BI} & \text{if } n \text{ is } \texttt{Start} \text{ block} \\ \bigcap\limits_{p \in pred(n)} \texttt{Out}_p & \text{otherwise} \end{cases}$$

$$\texttt{Out}_n = (\texttt{In}_n - \texttt{Kill}_n) \cup \texttt{Gen}_n$$

  where $\texttt{In}_n$, $\texttt{Out}_n$, $\texttt{Gen}_n$, $\texttt{Kill}_n$, and $\texttt{BI}$ are sets of definitions
- Note the use of ∩ to capture the *“any path”* nature of data flow
  - This is **different** to liveness and reaching def. analysis
  - The direction of data flow is forward like in reaching def.

Use: common subexpression elimination

- Optimization that searches for instances of identical expressions
    - i.e. all evaluate to the same value
- Analyzes whether it is worthwhile replacing the instances with a single variable holding the computed value
- Example:

```
a = b * c + g;
d = b * c * e;
```

- can be transformed into:

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```