CSE P 501 – Compilers

Dataflow Analysis
Hal Perkins
Autumn 2011
Agenda

- Initial example: dataflow analysis for common subexpression elimination
- Other analysis problems that work in the same framework
The Story So Far…

- Redundant expression elimination
  - Local Value Numbering
  - Superlocal Value Numbering
    - Extends VN to EBBs
    - SSA-like namespace
  - Dominator VN Technique (DVNT)
- All of these propagate along forward edges
- None are global
  - In particular, can’t handle back edges (loops)
Dominator Value Numbering

- Most sophisticated algorithm so far
- Still misses some opportunities
- Can’t handle loops
Available Expressions

- Goal: use dataflow analysis to find common subexpressions whose range spans basic blocks
- Idea: calculate *available expressions* at beginning of each basic block
- Avoid re-evaluation of an available expression – use a copy operation
“Available” and Other Terms

- An expression $e$ is defined at point $p$ in the CFG if its value is computed at $p$
  - Sometimes called definition site
- An expression $e$ is killed at point $p$ if one of its operands is defined at $p$
  - Sometimes called kill site
- An expression $e$ is available at point $p$ if every path leading to $p$ contains a prior definition of $e$ and $e$ is not killed between that definition and $p$
Available Expression Sets

- For each block $b$, define
  - $\text{AVAIL}(b)$ – the set of expressions available on entry to $b$
  - $\text{NKILL}(b)$ – the set of expressions not killed in $b$
  - $\text{DEF}(b)$ – the set of expressions defined in $b$ and not subsequently killed in $b$
Computing Available Expressions

- **AVAIL(b)** is the set

  \[ \text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x))) \]

- **preds(b)** is the set of b’s predecessors in the control flow graph

- This gives a system of simultaneous equations – a dataflow problem
Name Space Issues

- In previous value-numbering algorithms, we used a SSA-like renaming to keep track of versions.
- In global dataflow problems, we use the original namespace.
  - The KILL information captures when a value is no longer available.
GCSE with Available Expressions

- For each block b, compute DEF(b) and NKILL(b)
- For each block b, compute AVAIL(b)
- For each block b, value number the block starting with AVAIL(b)
- Replace expressions in AVAIL(b) with references to the previously computed values
Global CSE Replacement

- After analysis and before transformation, assign a global name to each expression $e$ by hashing on $e$

- During transformation step
  - At each evaluation of $e$, insert copy $\text{name}(e) = e$
  - At each reference to $e$, replace $e$ with $\text{name}(e)$
Analysis

- Main problem – inserts extraneous copies at all definitions and uses of every $e$ that appears in any AVAIL(b)
  - But the extra copies are dead and easy to remove
  - Useful copies often coalesce away when registers and temporaries are assigned

- Common strategy
  - Insert copies that might be useful
  - Let dead code elimination sort it out later
Computing Available Expressions

- Big Picture
  - Build control-flow graph
  - Calculate initial local data – DEF(b) and NKILL(b)
    - This only needs to be done once
  - Iteratively calculate AVAIL(b) by repeatedly evaluating equations until nothing changes
    - Another fixed-point algorithm
Computing DEF and NKILL (1)

- For each block $b$ with operations $o_1, o_2, \ldots, o_k$
  
  KILLED = $\emptyset$
  
  DEF(b) = $\emptyset$
  
  for $i = k$ to 1

  assume $o_i$ is “x = y + z”

  if (y $\notin$ KILLED and z $\notin$ KILLED)

    add “y + z” to DEF(b)

  add x to KILLED

  ...

...
After computing DEF and KILLED for a block \( b \),

\[
\text{NKILL}(b) = \{ \text{all expressions} \}
\]

for each expression \( e \)

for each variable \( \nu \in e \)

if \( \nu \in \text{KILLED} \) then

\[
\text{NKILL}(b) = \text{NKILL}(b) - e
\]
Computing Available Expressions

- Once DEF(b) and NKILL(b) are computed for all blocks b:
  
  Worklist = \{ all blocks b \}
  
  while (Worklist ≠ ∅)
    remove a block b from Worklist
    recompute AVAIL(b)
    if AVAIL(b) changed
      Worklist = Worklist ∪ successors(b)
Comparing Algorithms

- LVN – Local Value Numbering
- SVN – Superlocal Value Numbering
- DVN – Dominator-based Value Numbering
- GRE – Global Redundancy Elimination
Comparing Algorithms (2)

- LVN => SVN => DVN form a strict hierarchy – later algorithms find a superset of previous information
- Global RE finds a somewhat different set
  - Discovers e+f in F (computed in both D and E)
  - Misses identical values if they have different names (e.g., a+b and c+d when a=c and b=d)
    - Value Numbering catches this
Scope of Analysis

- Larger context (EBBs, regions, global, interprocedural) sometimes helps
  - More opportunities for optimizations
- But not always
  - Introduces uncertainties about flow of control
  - Usually only allows weaker analysis
  - Sometimes has unwanted side effects
    - Can create additional pressure on registers, for example
Code Replication

- Sometimes replicating code increases opportunities – modify the code to create larger regions with simple control flow

- Two examples
  - Cloning
  - Inline substitution
Cloning

- Idea: duplicate blocks with multiple predecessors
- Tradeoff
  - More local optimization possibilities – larger blocks, fewer branches
  - But: larger code size, may slow down if it interacts badly with cache
Original VN Example

A
- \( m = a + b \)
- \( n = a + b \)

B
- \( p = c + d \)
- \( r = c + d \)

C
- \( q = a + b \)
- \( r = c + d \)

D
- \( e = b + 18 \)
- \( s = a + b \)
- \( u = e + f \)

E
- \( e = a + 17 \)
- \( t = c + d \)
- \( u = e + f \)

F
- \( v = a + b \)
- \( w = c + d \)
- \( x = e + f \)

G
- \( y = a + b \)
- \( z = c + d \)
Example with cloning

\[ m = a + b \\ n = a + b \]

\[ p = c + d \\ r = c + d \]
\[ y = a + b \\ z = c + d \]

\[ q = a + b \\ r = c + d \]

\[ e = b + 18 \\ s = a + b \\ u = e + f \]
\[ v = a + b \\ w = c + d \\ x = e + f \]
\[ y = a + b \\ z = c + d \]

\[ e = a + 17 \\ t = c + d \\ u = e + f \]
\[ v = a + b \\ w = c + d \\ x = e + f \]
\[ y = a + b \\ z = c + d \]
Problem: an optimizer has to treat a procedure call as if it (could have) modified all globally reachable data

- Plus there is the basic expense of calling the procedure

Inline Substitution: replace each call site with a copy of the called function body
Inline Substitution Issues

- **Pro**
  - More effective optimization – better local context and don’t need to invalidate local assumptions
  - Eliminate overhead of normal function call

- **Con**
  - Potential code bloat
  - Need to manage recompilation when either caller or callee changes
Dataflow analysis

- Global redundancy elimination is the first example of a *dataflow analysis* problem
- Many similar problems can be expressed in a similar framework
- Only the first part of the story – once we’ve discovered facts, we then need to use them to improve code
Dataflow Analysis (1)

- A collection of techniques for compile-time reasoning about run-time values
- Almost always involves building a graph
  - Trivial for basic blocks
  - Control-flow graph or derivative for global problems
  - Call graph or derivative for whole-program problems
Dataflow Analysis (2)

- Usually formulated as a set of *simultaneous equations* (dataflow problem)
  - Sets attached to nodes and edges
  - Need a lattice (or semilattice) to describe values
    - In particular, has an appropriate operator to combine values and an appropriate “bottom” or minimal value
Dataflow Analysis (3)

- Desired solution is usually a *meet over all paths* (MOP) solution
  - “What is true on every path from entry”
  - “What can happen on any path from entry”
  - Usually relates to safety of optimization
Dataflow Analysis (4)

Limitations

- Precision – “up to symbolic execution”
  - Assumes all paths taken
- Sometimes cannot afford to compute full solution
- Arrays – classic analysis treats each array as a single fact
- Pointers – difficult, expensive to analyze
  - Imprecision rapidly adds up
- For scalar values we can quickly solve simple problems
Example: Available Expressions

- This is the analysis we did earlier to eliminate redundant expression evaluations

- Equation:
  \[
  \text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} \left( \text{DEF}(x) \cup \left( \text{AVAIL}(x) \cap \text{NKILL}(x) \right) \right)
  \]
Characterizing Dataflow Analysis

- All of these algorithms involve sets of facts about each basic block \( b \)
  - \( \text{IN}(b) \) – facts true on entry to \( b \)
  - \( \text{OUT}(b) \) – facts true on exit from \( b \)
  - \( \text{GEN}(b) \) – facts created and not killed in \( b \)
  - \( \text{KILL}(b) \) – facts killed in \( b \)

- These are related by the equation
  \[
  \text{OUT}(b) = \text{GEN}(b) \cup (\text{IN}(b) - \text{KILL}(b))
  \]

- Solve this iteratively for all blocks
- Sometimes information propagates forward; sometimes backward
Example: Live Variable Analysis

- A variable $v$ is *live* at point $p$ iff there is *any* path from $p$ to a use of $v$ along which $v$ is not redefined.

- Some uses:
  - Register allocation – only live variables need a register (or temporary)
  - Eliminating useless stores
  - Detecting uses of uninitialized variables
  - Improve SSA construction – only need $\Phi$-function for variables that are live in a block (later)
Liveness Analysis Sets

- For each block $b$, define
  - $\text{use}[b] = \text{variable used in } b \text{ before any def}$
  - $\text{def}[b] = \text{variable defined in } b \text{ & not killed}$
  - $\text{in}[b] = \text{variables live on entry to } b$
  - $\text{out}[b] = \text{variables live on exit from } b$
Equations for Live Variables

Given the preceding definitions, we have

\[ \text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b]) \]
\[ \text{out}[b] = \bigcup_{s \in \text{succ}[b]} \text{in}[s] \]

Algorithm

- Set \( \text{in}[b] = \text{out}[b] = \emptyset \)
- Update \text{in}, \text{out} until no change
Example (1 stmt per block)

- Code

```
a := 0
L: b := a+1
c := c+b
a := b*2
if a < N goto L
return c
```

```
1: a:= 0
2: b:=a+1
3: c:=c+b
4: a:=b+2
5: a < N
6: return c
```
Calculation

\[ \text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b]) \]
\[ \text{out}[b] = \bigcup_{s \in \text{succ}[b]} \text{in}[s] \]

1: \text{a:= 0}
2: \text{b:=a+1}
3: \text{c:=c+b}
4: \text{a:=b+2}
5: \text{a < N}
6: \text{return c}
Many problems have more than one formulation. For example, Live Variables...

**Sets**
- USED(b) – variables used in b before being defined in b
- NOTDEF(b) – variables not defined in b
- LIVE(b) – variables live on *exit* from b

**Equation**
\[
\text{LIVE}(b) = \bigcup_{s \in \text{succ}(b)} \text{USED}(s) \cup \text{LIVE}(s) \cap \text{NOTDEF}(s)
\]
Example: Reaching Definitions

- A definition $d$ of some variable $ν$ reaches operation $i$ iff $i$ reads the value of $ν$ and there is a path from $d$ to $i$ that does not define $ν$

- Use:
  - Find all of the possible definition points for a variable in an expression
Equations for Reaching Definitions

- **Sets**
  - `\text{DEFOUT}(b)` – set of definitions in `b` that reach the end of `b` (i.e., not subsequently redefined in `b`)
  - `\text{SURVIVED}(b)` – set of all definitions not obscured by a definition in `b`
  - `\text{REACHES}(b)` – set of definitions that reach `b`

- **Equation**

\[
\text{REACHES}(b) = \bigcup_{p \in \text{preds}(b)} \text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p))
\]
Example: Very Busy Expressions

- An expression \( e \) is considered very busy at some point \( p \) if \( e \) is evaluated and used along every path that leaves \( p \), and evaluating \( e \) at \( p \) would produce the same result as evaluating it at the original locations.

- Use:
  - Code hoisting – move \( e \) to \( p \) (reduces code size; no effect on execution time)
Equations for Very Busy Expressions

- **Sets**
  - **USED(b)** – expressions used in b before they are killed
  - **KILLED(b)** – expressions redefined in b before they are used
  - **VERYBUSY(b)** – expressions very busy on exit from b

- **Equation**
  \[
  \text{VERYBUSY}(b) = \bigcap_{s \in \text{succ}(b)} \text{USED}(s) \cup \left( \text{VERYBUSY}(s) - \text{KILLED}(s) \right)
  \]
Efficiency of Dataflow Analysis

- The algorithms eventually terminate, but the expected time needed can be reduced by picking a good order to visit nodes in the CFG depending on how information flows
  - Forward problems – reverse postorder
  - Backward problems - postorder
Using Dataflow Information

- A few examples of possible transformations...
Classic Common-Subexpression Elimination

- In a statement $s: t := x \text{ op } y$, if $x \text{ op } y$ is *available* at $s$ then it need not be recomputed.

- Analysis: compute *reaching expressions* i.e., statements $n: v := x \text{ op } y$ such that the path from $n$ to $s$ does not compute $x \text{ op } y$ or define $x$ or $y$.

Classic CSE

- If $x \text{ op } y$ is defined at $n$ and reaches $s$
  - Create new temporary $w$
  - Rewrite $n$ as
    
    $n: w := x \text{ op } y$
    
    $n': v := w$
  - Modify statement $s$ to be
    
    $s: t := w$

- (Rely on copy propagation to remove extra assignments if not really needed)
Constant Propagation

- Suppose we have
  - Statement $d: t := c$, where $c$ is constant
  - Statement $n$ that uses $t$
- If $d$ reaches $n$ and no other definitions of $t$ reach $n$, then rewrite $n$ to use $c$ instead of $t$
Copy Propagation

- Similar to constant propagation
- Setup:
  - Statement d: t := z
  - Statement n uses t
- If d reaches n and no other definition of t reaches n, and there is no definition of z on any path from d to n, then rewrite n to use z instead of t
Copy Propagation Tradeoffs

- Downside is that this can increase the lifetime of variable \( z \) and increase need for registers or memory traffic
  - Not worth doing if only reason is to eliminate copies – let the register allocate deal with that

- But it can expose other optimizations, e.g.,
  
  \[
  \begin{align*}
  a & := y + z \\
  u & := y \\
  c & := u + z
  \end{align*}
  \]

- After copy propagation we can recognize the common subexpression
Dead Code Elimination

- If we have an instruction
  \[ s: a := b \text{ op } c \]
  and \( a \) is not live-out after \( s \), then \( s \) can be eliminated

  - Provided it has no implicit side effects that are visible (output, exceptions, etc.)
Aliases

- A variable or memory location may have multiple names or *aliases*
  - Call-by-reference parameters
  - Variables whose address is taken (&x)
  - Expressions that dereference pointers (p.x, *p)
  - Expressions involving subscripts (a[i])
  - Variables in nested scopes
Aliases vs Optimizations

Example:

\[ p.x := 5; \quad q.x := 7; \quad a := p.x; \]

- Does reaching definition analysis show that the definition of \( p.x \) reaches \( a \)?
- (Or: do \( p \) and \( q \) refer to the same variable/object?)
- (Or: *can* \( p \) and \( q \) refer to the same thing?)
Aliases vs Optimizations

- Example
  
  ```c
  void f(int *p, int *q) {
    *p = 1; *q = 2;
    return *p;
  }
  ```

- How do we account for the possibility that p and q might refer to the same thing?
- Safe approximation: since it’s possible, assume it is true (but rules out a lot)
Types and Aliases (1)

- In Java, ML, MiniJava, and others, if two variables have incompatible types they cannot be names for the same location.
  - Also helps that programmer cannot create arbitrary pointers to storage in these languages.
Types and Aliases (2)

- Strategy: Divide memory locations into *alias classes* based on type information (every type, array, record field is a class).
- Implication: need to propagate type information from the semantics pass to optimizer
  - Not normally true of a minimally typed IR
- Items in different alias classes cannot refer to each other
Aliases and Flow Analysis

- Idea: Base alias classes on points where a value is created
  - Every new/malloc and each local or global variable whose address is taken is an alias class
  - Pointers can refer to values in multiple alias classes (so each memory reference is to a set of alias classes)
  - Use to calculate “may alias” information (e.g., p “may alias” q at program point s)
Using “may-alias” information

- Treat each alias class as a “variable” in dataflow analysis problems
- Example: framework for available expressions
  - Given statement \( s: M[a] := b, \)
    
    \[
    \begin{align*}
    \text{gen}[s] &= \{ \} \\
    \text{kill}[s] &= \{ M[x] \mid a \text{ may alias } x \text{ at } s \}
    \end{align*}
    \]
May-Alias Analysis

- Without alias analysis, #2 kills M[t] since x and t might be related
- If analysis determines that “x may-alias t” is false, M[t] is still available at #3; can eliminate the common subexpression and use copy propagation

Code

1: \( u := M[t] \)
2: \( M[x] := r \)
3: \( w := M[t] \)
4: \( b := u + w \)
Where are we now?

- Dataflow analysis is the core of classical optimizations
- Still to explore:
  - Discovering and optimizing loops
  - SSA – Static Single Assignment form