COS320: Compiling Techniques

Zak Kincaid

April 16, 2019

- Deadline for HW4 extended to Monday
- HW5 out today
- Don’t wait until Tuesday to start HW5!

Data flow analysis

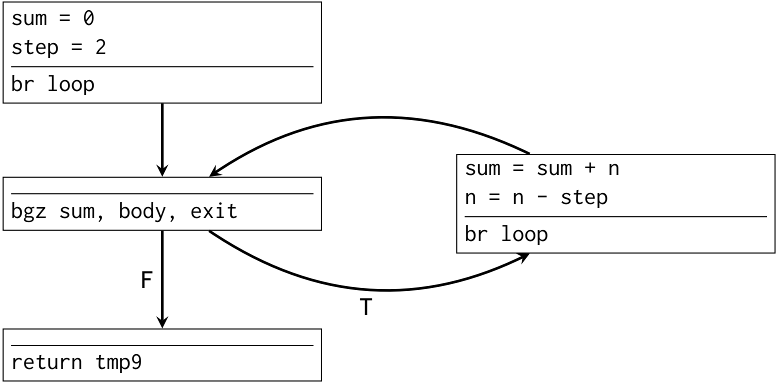
Recall: constant propagation

- Let $G = (N, E, s)$ be a control flow graph.
- *cp* is the *smallest*[1] function such that
  - $cp(s) = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$
  - For each $p \to n \in E$, $post(p, cp(p)) \leq cp(n)$

$\mathit{cp}(s) = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$;
$\mathit{cp}(n) = \{x_1 \mapsto \bot, ..., x_n \mapsto \bot\}$ for all other nodes;
$\mathit{work} \leftarrow N \setminus \{s\}$; `/* Set of nodes that may violate spec */`
while $\mathit{work} \neq \emptyset$ **do**
  Pick some n from work;
  $\mathit{work} \leftarrow \mathit{work} \setminus \{n\}$;
  $C \leftarrow \bigsqcup_{p \in \mathit{pred}(n)} \mathit{post}(p, \mathit{cp}(p))$;
  if $C \neq \mathit{cp}(n)$ **then**
    $\mathit{cp}(n) \leftarrow C$;
    $\mathit{work} \leftarrow \mathit{work} \cup \mathit{succ}(n)$

[1] Pointwise order: $f \leq g$ if for all nodes n and all variables x, $f(n)(x) \preceq g(n)(x)$

```c
int sum2(int n) {
  int sum = 0;
  int step = 2;
  while (n > 0) {
    sum = sum + n;
    n = n - step;
  }
  return sum;
}
```



Common subexpression elimination

- Common subexpression elimination searches for expressions that
  - appear at multiple points in a program
  - evaluate to the same value at those points

  and (possibly) save the cost of re-evaluation by storing that value.

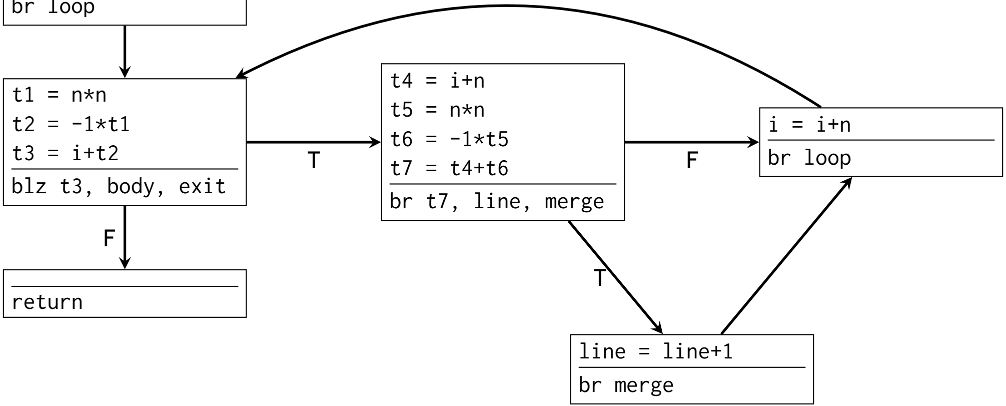
```
void print (long *m, long n) {
  long i,j;
  for (i = 0; i < n*n; i += n) {
    for (j = 0; j < n; j += 1) {
      printf(‘‘ %ld’’, *(m + i + j));
    }
    if (i + n < n*n) {
      printf(‘‘\n’’);
    }
  }
}
```

→

```
void print (long *m, long n) {
  long i,j;
  long n_times_n = n*n;
  for (i = 0; i < n_times_n; ) {
    for (j = 0; j < n; j += 1) {
      printf(‘‘ %ld’’, *(m + i + j));
    }
    long i_plus_n = i+n;
    if (i_plus_n < n_times_n) {
      printf(‘‘\n’’);
    }
    i = i_plus_n;
  }
}
```

Available expressions

- An *expression* in our simple simple imperative language has one of the following forms:
  - `add` <opn>, <opn>
  - `mul` <opn>, <opn>
- Fix control flow graph $G = (N, E, s)$
- An expression e is ***available*** at basic block $n \in N$ if for every path from s to n in G:
  1. the expression e is evaluated along the path
  2. after the *last* evaluation of e along the path, no variables in e are overwritten
- Idea: if expression e is available at node n, then can eliminate redundant computations of e within n


i = 0
br loop
t1 = n*n
t2 = -1*t1
t3 = i+t2
blz t3, body, exit
T
t4 = i+n
t5 = n*n
t6 = -1*t5
t7 = t4+t6
br t7, line, merge
F
i = i+n
br loop
F
return
T
line = line+1
br merge

Propagating available expressions

- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?

Propagating available expressions

- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?
  - $\textit{post}_{\textit{AE}}(x = e, E) = \{e' \in E : x \text{ not in } e'\} \cup \{e\}$

Propagating available expressions

- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?
  - *post*$_{AE}(x = e, E) = \{e' \in E : x$ not in $e'\} \cup \{e\}$
- How do we propagate available expressions through a basic block?

Propagating available expressions

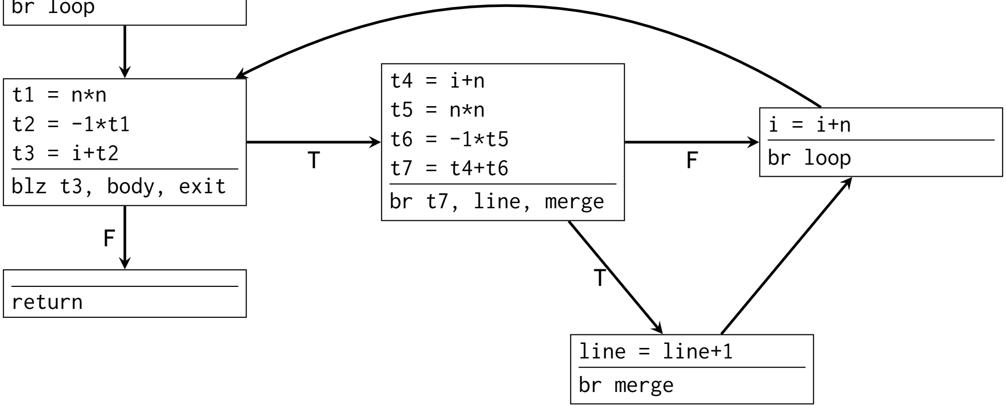
- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?
    - ***post***$_{\textit{AE}}(x = e, E) = \{e' \in E : x$ not in $e'\} \cup \{e\}$
- How do we propagate available expressions through a basic block?
    - Block takes the form $instr_1, ..., instr_n, term$.
      take ***post***$_{\textit{AE}}(block, E) =$ ***post***$_{\textit{AE}}(instr_n, ...$***post***$_{\textit{AE}}(instr_1, E))$

Available expressions

- Let $G = (N, E, s)$ be a control flow graph.
- *ae* is the *smallest*[2] function such that
  - $\textit{ae}(s) = \emptyset$
  - For each $p \to n \in E$, $\textit{post}_{AE}(p, \textit{ae}(p)) \supseteq \textit{ae}(n)$

$\textit{ae}(s) = \emptyset$;
$\textit{ae}(n) = \{\textit{all expressions}\}$ for all other nodes;
$\textit{work} \leftarrow N \setminus \{s\}$; /* Set of nodes that may violate spec */
while $\textit{work} \neq \emptyset$ **do**
  Pick some n from work;
  $\textit{work} \leftarrow \textit{work} \setminus \{n\}$;
  $E \leftarrow \bigcap_{p \in \textit{pred}(n)} \textit{post}_{AE}(p, \textit{ae}(p))$;
  if $E \neq \textit{ae}(n)$ **then**
    $\textit{ae}(n) \leftarrow E$;
    $\textit{work} \leftarrow \textit{work} \cup \textit{succ}(n)$

[2] Pointwise reverse-inclusion order: $f \leq g$ if for all nodes n, $f(n) \supseteq g(n)$


i = 0
br loop
t1 = n*n
t2 = -1*t1
t3 = i+t2
blz t3, body, exit
T
F
return
t4 = i+n
t5 = n*n
t6 = -1*t5
t7 = t4+t6
br t7, line, merge
F
T
i = i+n
br loop
line = line+1
br merge

Constant propagation

cp is the *smallest* function such that

- $cp(s) = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$
- For each $p \to n \in E$,
  $post(p, cp(p)) \leq cp(n)$

Available expressions

ae is the *smallest* function such that

- $\textit{ae}(s) = \emptyset$
- For each $p \to n \in E$,
  $\textit{post}_{\textit{AE}}(p, \textit{ae}(p)) \supseteq \textit{ae}(n)$

- **Commonality**: *cp* and *ae* are least solutions to a system of local constraints
  - "Local": defined in terms of *edges*; contrast with "global", which depends on the structure of the whole graph (e.g., paths)

Constant propagation

$\mathit{cp}(s) = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$;
$\mathit{cp}(n) = \{x_1 \mapsto \bot, ..., x_n \mapsto \bot\}$ for all other nodes;
$\mathit{work} \leftarrow N \setminus \{s\}$;
while $\mathit{work} \neq \emptyset$ **do**
  Pick some n from work;
  $\mathit{work} \leftarrow \mathit{work} \setminus \{n\}$;
  $C \leftarrow \bigsqcup_{p \in \mathit{pred}(n)} \mathit{post}(p, \mathit{cp}(p))$;
  if $C \neq \mathit{cp}(n)$ **then**
    $\mathit{cp}(n) \leftarrow C$;
    $\mathit{work} \leftarrow \mathit{work} \cup \mathit{succ}(n)$

Available expressions

$\mathit{ae}(s) = \emptyset$;
$\mathit{ae}(n) = \{\mathit{all\ expressions}\}$ for all other nodes;
$\mathit{work} \leftarrow N \setminus \{s\}$;
while $\mathit{work} \neq \emptyset$ **do**
  Pick some n from work;
  $\mathit{work} \leftarrow \mathit{work} \setminus \{n\}$;
  $E \leftarrow \bigcap_{p \in \mathit{pred}(n)} \mathit{post}_{AE}(p, \mathit{ae}(p))$;
  if $E \neq \mathit{ae}(n)$ **then**
    $\mathit{ae}(n) \leftarrow E$;
    $\mathit{work} \leftarrow \mathit{work} \cup \mathit{succ}(n)$

- The algorithms for computing *cp* and *ae* are essentially the same

Dataflow analysis

- *Dataflow analysis* is an approach to program analysis that unifies the presentation and implementation of many different analyses
- **Formulate** problem as a system of constraints
- **Solve** the constraints iteratively (using some variation of the workset algorithm)
- What now:
  - General theory & algorithms
  - Conditions under which the approach works
  - Guarantees about the solution
- Not covered: *abstract interpretation* – a general theory for relating program analysis to program semantics
  - What does it mean for a constraint system to be correct?
  - How do we prove it?

A (forward) dataflow analysis consists of:

- An **abstract domain** $\mathcal{L}$
  - Defines the space of program "properties" that we are interested in
- An **abstract transformer** *post*$_{\mathcal{L}}$
  - Determines how each basic block transforms properties
  - i.e., if property p holds *before* n, then *post*$_{\mathcal{L}}(n, p)$ is a property that holds *after* n

Abstract domains

An abstract domain is a set $\mathcal{L}$ equipped with:

- A partial order $\sqsubseteq$
  - $x \sqsubseteq y$ means that x represents more precise information about the program than y[3]
  - Technical requirement: ascending chain condition – any infinite ascending sequence

    $$x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \dots$$

    must eventually stabilize: for some i, we have $x_j = x_i$ for all $j \geq i$.

[3] The other direction also works, and is the one taken in classical compilers literature. In this class, we will stick to this direction, which is the convention established in abstract interpretation.

Abstract domains

An abstract domain is a set $\mathcal{L}$ equipped with:

- A partial order $\sqsubseteq$
  - $x \sqsubseteq y$ means that x represents more precise information about the program than y[3]
  - Technical requirement: ascending chain condition – any infinite ascending sequence

$$x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \ldots$$

  must eventually stabilize: for some i, we have $x_j = x_i$ for all $j \geq i$.
- A *least upper bound* ("join") operator, $\sqcup$
  1. $x \sqsubseteq x \sqcup y$
  2. $y \sqsubseteq x \sqcup y$
  3. $x \sqcup y \sqsubseteq z$ for any z satisfying 1 and 2
- A *least element* ("bottom"), $\bot$
- A *greatest element* ("top"), $\top$

[3] The other direction also works, and is the one taken in classical compilers literature. In this class, we will stick to this direction, which is the convention established in abstract interpretation.

Abstract domains

An abstract domain is a set $\mathcal{L}$ equipped with:

- A partial order $\sqsubseteq$
  - $x \sqsubseteq y$ means that x represents more precise information about the program than y[3]
  - Technical requirement: ascending chain condition – any infinite ascending sequence

$$x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \ldots$$

  must eventually stabilize: for some i, we have $x_j = x_i$ for all $j \geq i$.
- A *least upper bound* ("join") operator, $\sqcup$
  1. $x \sqsubseteq x \sqcup y$
  2. $y \sqsubseteq x \sqcup y$
  3. $x \sqcup y \sqsubseteq z$ for any z satisfying 1 and 2
- A *least element* ("bottom"), $\bot$
- A *greatest element* ("top"), $\top$

What are the abstract domains of constant propagation & available expressions?

[3] The other direction also works, and is the one taken in classical compilers literature. In this class, we will stick to this direction, which is the convention established in abstract interpretation.

Transfer functions

A transfer function $\textit{post}_{\mathcal{L}} : \textit{Basic Block} \times \mathcal{L} \to \mathcal{L}$

- Technical requirement: $\textit{post}_{\mathcal{L}}$ is montone

$$x \sqsubseteq y \Rightarrow \textit{post}_{\mathcal{L}}(n, x) \sqsubseteq \textit{post}_{\mathcal{L}}(n, y)$$

("more information in $\Rightarrow$ more information out")

Transfer functions

A transfer function $\textit{post}_{\mathcal{L}} : \textit{Basic Block} \times \mathcal{L} \to \mathcal{L}$

- Technical requirement: $\textit{post}_{\mathcal{L}}$ is montone

$$x \sqsubseteq y \Rightarrow \textit{post}_{\mathcal{L}}(n, x) \sqsubseteq \textit{post}_{\mathcal{L}}(n, y)$$

  ("more information in $\Rightarrow$ more information out")

- Desirable property: $\textit{post}_{\mathcal{L}}$ is distributive: for all $x, y \in L$,

$$\textit{post}_{\mathcal{L}}(n, x \sqcup y) = \textit{post}_{\mathcal{L}}(n, x) \sqcup \textit{post}_{\mathcal{L}}(n, y)$$

  - $\textit{post}_{AE}$ is distributive
  - $\textit{post}_{CP}$ is not (why?)

Transfer functions

A transfer function *post*$_{\mathcal{L}}$: *Basic Block* $\times \mathcal{L} \to \mathcal{L}$

- Technical requirement: *post*$_{\mathcal{L}}$ is montone

$$x \sqsubseteq y \Rightarrow \textit{post}_{\mathcal{L}}(n, x) \sqsubseteq \textit{post}_{\mathcal{L}}(n, y)$$

  ("more information in $\Rightarrow$ more information out")
- Desirable property: *post*$_{\mathcal{L}}$ is distributive: for all $x, y \in L$,

$$\textit{post}_{\mathcal{L}}(n, x \sqcup y) = \textit{post}_{\mathcal{L}}(n, x) \sqcup \textit{post}_{\mathcal{L}}(n, y)$$

  - *post*$_{AE}$ is distributive
  - *post*$_{CP}$ is not (why?)
  - General family of distributive transfer functions: "gen/kill" analyses.
    - Suppose we have a finite set of data flow "facts"
    - Elements of the abstract domain are *sets* of facts
    - For each basic block n, associate a set of *generated* facts *gen*(n) and *killed* facts *kill*(n)
    - Define *post*$_{\mathcal{L}}(n, F) = (F \setminus \textit{kill}(n)) \cup \textit{gen}(n)$. *post*$_{\mathcal{L}}$ is distributive!

Generic (forward) dataflow analysis algorithm

- Given:
  - Abstract domain $(\mathcal{L}, \sqsubseteq, \sqcup, \bot, \top)$
  - Transfer function
    post$_{\mathcal{L}}$: *Basic Block* $\times \mathcal{L} \to \mathcal{L}$
  - Control flow graph $G = (N, E, s)$
- Compute: *least* function f such that
  1. $f(s) = \top$
  2. For all $p \to n \in E$, *post*$_{\mathcal{L}}(p, f(p)) \sqsubseteq f(n)$

Generic (forward) dataflow analysis algorithm

- Given:
  - Abstract domain $(\mathcal{L}, \sqsubseteq, \sqcup, \perp, \top)$
  - Transfer function
    $\mathit{post}_{\mathcal{L}} : \mathit{Basic\ Block} \times \mathcal{L} \to \mathcal{L}$
  - Control flow graph $G = (N, E, s)$
- Compute: *least* function f such that
  1. $f(s) = \top$
  2. For all $p \to n \in E$, $\mathit{post}_{\mathcal{L}}(p, f(p)) \sqsubseteq f(n)$

$f(s) \leftarrow \top$;
$f(n) = \perp$ for all other nodes;
work $\leftarrow N \setminus \{s\}$;
while *work* $\neq \emptyset$ **do**
  Pick some n from work;
  work $\leftarrow$ *work* $\setminus \{n\}$;
  $v \leftarrow \bigsqcup_{p \in \mathit{pred}(n)} \mathit{post}_{\mathcal{L}}(p, f(p))$;
  if $v \neq f(n)$ **then**
    $f(n) \leftarrow v$;
    work $\leftarrow$ *work* $\cup$ $\mathit{succ}(n)$

Generic (forward) dataflow analysis algorithm

- Given:
  - Abstract domain $(\mathcal{L}, \sqsubseteq, \sqcup, \bot, \top)$
  - Transfer function
    post$_\mathcal{L}$: *Basic Block* $\times\, \mathcal{L} \rightarrow \mathcal{L}$
  - Control flow graph $G = (N, E, s)$
- Compute: *least* function f such that
  1. $f(s) = \top$
  2. For all $p \rightarrow n \in E$, *post*$_\mathcal{L}(p, f(p)) \sqsubseteq f(n)$

$f(s) \leftarrow \top$;
$f(n) = \bot$ for all other nodes;
work $\leftarrow N \setminus \{s\}$;
while *work* $\neq \emptyset$ **do**
    Pick some n from work;
    work $\leftarrow$ *work* $\setminus \{n\}$;
    $v \leftarrow \bigsqcup_{p \in \textit{pred}(n)} \textit{post}_\mathcal{L}(p, f(p))$;
    if $v \neq f(n)$ **then**
        $f(n) \leftarrow v$;
        work $\leftarrow$ *work* $\cup$ *succ*(n)

Invariants:

- *work* contains all $n \in N$ that may violate their constraints (*post*$(p, f(p)) \not\sqsubseteq f(n)$ for some $p \rightarrow n \in E$)
- Use f_i to denote f on the ith iteration and f^* to denote least solution to the constraint system. Then for all n, $f_i(n) \sqsubseteq f^*(n)$.

Generic (forward) dataflow analysis algorithm

- Given:
  - Abstract domain $(\mathcal{L}, \sqsubseteq, \sqcup, \bot, \top)$
  - Transfer function
    post$_{\mathcal{L}}$: *Basic Block* $\times \mathcal{L} \to \mathcal{L}$
  - Control flow graph $G = (N, E, s)$
- Compute: *least* function f such that
  1. $f(s) = \top$
  2. For all $p \to n \in E$, *post*$_{\mathcal{L}}(p, f(p)) \sqsubseteq f(n)$

$f(s) \leftarrow \top$;
$f(n) = \bot$ for all other nodes;
work $\leftarrow N \setminus \{s\}$;
while *work* $\neq \emptyset$ **do**
  Pick some n from work;
  work $\leftarrow$ *work* $\setminus \{n\}$;
  $v \leftarrow \bigsqcup_{p \in \mathit{pred}(n)} \mathit{post}_{\mathcal{L}}(p, f(p))$;
  if $v \neq f(n)$ **then**
    $f(n) \leftarrow v$;
    work $\leftarrow$ *work* $\cup$ *succ*(n)

Invariants:

- *work* contains all $n \in N$ that may violate their constraints (*post*$(p, f(p)) \not\sqsubseteq f(n)$ for some $p \to n \in E$)
- Use f_i to denote f on the ith iteration and f^* to denote least solution to the constraint system. Then for all n, $f_i(n) \sqsubseteq f^*(n)$.

Termination:

- Why does this algorithm terminate?

Generic (forward) dataflow analysis algorithm

- Given:
  - Abstract domain $(\mathcal{L}, \sqsubseteq, \sqcup, \bot, \top)$
  - Transfer function
    $\textit{post}_{\mathcal{L}} : \textit{Basic Block} \times \mathcal{L} \to \mathcal{L}$
  - Control flow graph $G = (N, E, s)$
- Compute: *least* function f such that
  1. $f(s) = \top$
  2. For all $p \to n \in E$, $\textit{post}_{\mathcal{L}}(p, f(p)) \sqsubseteq f(n)$

$f(s) \leftarrow \top$;
$f(n) = \bot$ for all other nodes;
work $\leftarrow N \setminus \{s\}$;
while *work* $\neq \emptyset$ **do**
  Pick some n from work;
  work $\leftarrow$ *work* $\setminus \{n\}$;
  $v \leftarrow \bigsqcup_{p \in \textit{pred}(n)} \textit{post}_{\mathcal{L}}(p, f(p))$;
  if $v \neq f(n)$ **then**
    $f(n) \leftarrow v$;
    work $\leftarrow$ *work* $\cup$ *succ*(n)

Invariants:

- *work* contains all $n \in N$ that may violate their constraints (*post*$(p, f(p)) \not\sqsubseteq f(n)$ for some $p \to n \in E$)
- Use f_i to denote f on the ith iteration and f^* to denote least solution to the constraint system. Then for all n, $f_i(n) \sqsubseteq f^*(n)$.

Termination:

- Why does this algorithm terminate?
- Ascending chain condition $\Rightarrow$ for each n, $f_1(n) \sqsubseteq f_2(n) \sqsubseteq f_3(n) \sqsubseteq \ldots$ must eventually stabilize