Lecture 17: Constraints and Code Generation

More compilation to constraints
Using solvers for code generation
Announcements

• HW4
  – Due this Monday 11pm (no late days)

• Project meetings tomorrow
  – Milestones due tonight

• Final poster session
  – 3/15, 2:30-4:20pm, CSE atrium
  – Format: we will drop by for 4 mins for your show and tell
  – You will have to demo your work
  – You will peer review two other posters
Outline for today

Compiling $401_{CP}$ to constraints

Techniques for code generation
  - Classical and solver-based techniques
Translating $401_{CP}$ to Constraints
Our target language is based on SAT

- SAT
  - Language based on Boolean formulas
  - A language of assertions (for expressing constraints)

- Constructs:
  - Identifiers (Booleans or integers of finite length)
  - Arithmetic operators (+, -, ...)
  - Logical connectives and equality (∨, ∧, ¬)
  - Constraints: $c^\dagger(E)$

- The compiled program will be interpreted by the SAT solver
Example program

(a Int)
(b Int)
ct(a = b ∧ a < 10)

Translates to: ∃ a, b . a = b ∧ a < 10
Solving gives: a = 9, b = 9

• Each variable holds a constant value throughout
  – Remember that programs are non-directional!
Meaning of constraints

- They are used to describe *program states*
- Example:

  ```
  // imperative code
  def a;
  def b;
  a = 10;
  b = a;
  assert(a == b);
  ```

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>10</td>
</tr>
</tbody>
</table>
Meaning of constraints

- They are used to describe *program states*
- Example:

```cpp
// imperative code
def a;
def b;
a = 10;
b = a;
assert(a == b);
```

// SAT constraints
(a Int)
Meaning of constraints

- They are used to describe program states
- Example:

```plaintext
// imperative code
def a;
def b;
a = 10;
b = a;
assert(a == b);
```

```plaintext
// SAT constraints
(a Int)
(b Int)
```

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>
Meaning of constraints

- They are used to describe *program states*
- Example:

  ```
  // imperative code
  def a;
  def b;
  a = 10;
  b = a;
  assert(a == b);
  ```

  Constraints describe the *effects* of imperative stmts on program state

  ```
  // SAT constraints
  (a Int)
  (b Int)
  ct(10 = a ∧ b = a)
  ```

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>10</td>
</tr>
</tbody>
</table>
Same constraints can describe many states

// imperative code
lambda double (x) {
    def r = 2*x
    r
}

// SAT constraints
(x Int)
(r Int)
ct(2 * x = r)

Allows us to solve for inputs

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>??</td>
</tr>
<tr>
<td>r</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>r</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>r</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>r</td>
<td>4</td>
</tr>
</tbody>
</table>

1. ✅
2. ✅
3. ✅
4. ❌
Translating $401_{CP}$ to Constraints

- Goal is to describe the *effect* of each $401_{CP}$ statement on the program state

- We will do that using syntax-directed translation
Steps in translation

- We will translate each construct in $401_{\text{CP}}$:
  - Identifiers
  - Expressions
  - Assignments
  - Conditionals
  - Function declarations and Calls
  - Loops
  - Arrays
  - assert and choose
Translating identifiers

- **Booleans:**
  \[
  \text{def } b \rightarrow (b \text{ Bool})
  \]

- **Integers:**
  \[
  \text{def } i \rightarrow (i \text{ Int})
  \]

- This assumes we have performed type inference
Expressions

- id \rightarrow id
- E_1 \text{ op } E_2 \rightarrow E_1 \text{ op } E_2

Example:

\begin{align*}
\text{def } & a; & (a \text{ Int}) \\
\text{def } & b; & (b \text{ Int}) \\
\text{def } & c; & (c \text{ Int}) \\
\text{c } = & a + b & \text{ct}(a + b = c)
\end{align*}
Translating statements

- We will translate each $401_{\text{CP}}$ statements into constraints
- Combine them together at the end in as a single constraint using AND

```python
def a;
def b;
def c;
def d;
c = a + b;
d = a;
```

<table>
<thead>
<tr>
<th>Collected constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a + b = c$</td>
</tr>
</tbody>
</table>
Translating statements

- We will translate each $401_{\text{cp}}$ statements into constraints
- Combine them together at the end in as a single constraint using AND

```python
def a;
def b;
def c;
def d;
c = a + b;
d = a;
```

<table>
<thead>
<tr>
<th>Collected constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a + b = c$</td>
</tr>
<tr>
<td>$a = d$</td>
</tr>
</tbody>
</table>
Translating statements

- We will translate each $401_{CP}$ statements into constraints
- Combine them together at the end in as a single constraint using AND

```python
def a;
def b;
def c;
def d;
c = a + b;
d = a;
```

Collected constraints:

<table>
<thead>
<tr>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = a + b$</td>
</tr>
<tr>
<td>$a = d$</td>
</tr>
</tbody>
</table>

$\text{ct}(c = a + b \land a = d)$
Assignments

• Our target language does not have assignments
• This can be problematic:

```
def c:
    c = 10;
    c = 20;
```

(c Int)

\[ct(10 = c \land 20 = c)\]

C cannot be both 10 and 20!

```
def a:
    a = c;
```

But we can’t just throw away \(c = 10\) either

```
def c:
    c = 10;
    c = 20;
```

```
Handling assignments

- Solution: rewrite $401_{cp}$ statements such that each variable is assigned exactly once

```python
def c:
c = 10;
def a:
a = c;
c = 20;
def c1:
c1 = 10;
def a:
a = c1;
def c2:
c2 = 20;
```

```
(c1 Int) (a Int) (c2 Int)
```

This is called Static Single Assignment (SSA) form

- Conversion is done using reaching definition analysis (see section last week)
Conditionals

if \( (E_1) \) {
    E_2
} else {
    E_3
}

c\_t( \text{ite}(E_{1CP}, E_{2CP}, E_{3CP}) )

/* \( E_1 \) translates to \( E_{1CP} \)
\( E_2 \) translates to \( E_{2CP} \)
\( E_3 \) translates to \( E_{3CP} \) */

Example:

```python
def a; def b; def c;
if (a == 10) {
    b = 1
} else {
    c = 2
}
```

c\_t( \text{ite}(10 = a, 1 = b, 2 = c) )

\textbf{\textit{ite}} is just syntactic sugar:

\textit{ite}(c, e_1, e_2) \Rightarrow (c \land e_1) \lor (\neg c \land e_2)
Conditionals

But what about:

```python
def a; def b; def c;
if (a == 10) {
    b = 1
} else {
    b = 2
}
c = b
```

```python
def a; def b_1; def b_2; def c;
if (a == 10) {
    b_1 = 1
} else {
    b_2 = 2
}
c = b // which b??
```

Both \( b_1 \) and \( b_2 \) can flow to \( c \)
Keeping track of branches

```plaintext
def a; def b_1; def b_2;
def c;
if (a == 10) {
    b_1 = 1
} else {
    b_2 = 2
}
c = phi(b_1, b_2)
```

Keeps track of which branch we came from.

```
(a Int) (b_1 Int) (b_2 Int) (c Int)
```

```plaintext
c(t(ite(10 = a, 1 = b_1, 2 = b_2) \land
    ite(10 = a, b_1 = c, b_2 = c)))
```

```plaintext
optimize
```

```
(a Int) (b_1 Int) (b_2 Int) (c Int)
```

```plaintext
c(t(ite(10 = a,
    1 = b_1 \land b_1 = c,
    2 = b_2 \land b_2 = c))
```

There are no functions in our SAT language

- Just inline function body at call site!

\[
\text{lambda dbl(x)} \\
\{ \ r = 2*x; \ r \ \}
\]

\[
\ldots
\]

\[
\ldots
\]

\[
\text{ct}( \ 2*a = r_1 \land r_1 = x \land \\
2*b = r_2 \land r_2 = y )
\]

- In doing so, we need to:
  - Create new variables as needed
  - Replace function parameters with actual arguments

- This will blow up size of the SAT program
Loops

- Loops can create infinitely many assignments

```python
def c:
    c = 0;
    for (e : l):
        c = c + e
    def a:
        def a:
            a = c
def a;
```

\[
\text{ct}(\ 0 = c \land c + e = c \land c = a)\]

\(\times\)
Unroll loops

• We assume loop runs for a fixed number of times

```python
def c:
c = 0
for (e : l):
{ c = c + e }

def a:
a = c

assert(l[2] == null)
```

Ensure that loop indeed terminates!
Arrays

• If array length is known, then we desugar the array using extra variables

```python
def a = {}
da[0] = 10
a[1] = 20
def a_0; def a_1 (a_0 Int)
da_0 = 10 (a_1 Int)
a_1 = 20 (c Int)
def c
ct( 10 = a_0 ∧ 20 = a_1 ∧ c = a_0 = c)
c = a[0]
c = a_0
```

• If length is unknown, then we set an upper bound on the number of elements we will consider
  – This is similar to loop unrolling
Arrays accesses

• What about \( a[i] \)?
• It can one of \( a_0, \ldots, a_N \) where \( N = \text{length of } a \)
• Let’s create a new construct in SAT for translation

\[
a[i] \rightarrow \text{mux}(i, a_0, \ldots, a_N)
\]

\text{mux is just syntactic sugar:}

\[
\text{mux}(i, a_0, \ldots, a_N) \rightarrow
\]

\[
(\text{ite } 0 = i, a_0,
\]

\[
(\text{ite } 1 = i, a_1,
\]

\[
(\text{ite } 2 = i, a_2, \ldots)
\]

\]
Example

if (c == 0) {
    i = 0
} else {
    i = 1
}

// a has length 2

// 

i = phi(i₀, i₁)
c = a[i]

c = a[i]

\[
(a₀ \text{Int}) (a₁ \text{Int}) \\
(i₀ \text{Int}) (i₁ \text{Int}) (i \text{Int}) (c \text{Int})
\]

\[
\text{ct}(\text{ite}(0 = c, 0 = i₀, 1 = i₁) \land \text{ite}(0 = c, i₀ = i, i₁ = i) \land \text{mux}(i, a₀, a₁) = c)
\]

\[
\text{optimize}
\]
choose

• With no code fairies in sight, we need to translate this to constraints:
  – Enumerate all choices and let solver decide!

\[
\text{def } x \\
\text{x = choose()}
\]

\[(x \text{Int})
\]

\[
\text{ct}(1 = x \lor 2 = x \lor 3 = x \lor ...)
\]

• We first need to know the possible choices for x
End-to-End Example

\[
\text{lambda pop (x) \{} \\
\quad \text{def count} = 0; \\
\quad \text{for(e : x) \{} \\
\quad \quad \text{if (e == 1)} \{} \\
\quad \quad \quad \text{count} = \text{count}+1; \\
\quad \quad \}\text{else} \{} \\
\quad \quad \quad \text{count} = \text{count}; \\
\quad \}\text{count} \\
\text{def c = pop([choose(), choose(), choose(), choose()])} \\
\text{assert(c == 2)}
\]

\[
\text{lambda pop (x) \{} \\
\quad \text{def count}_0 = 0; \\
\quad \text{def e}_0 = x[0]; \\
\quad \text{if (e}_0 \text{ == 1)} \{} \\
\quad \quad \text{count}_1 = \text{count}_0+1 \} \\
\quad \text{else} \{} \\
\quad \quad \text{count}_1 = \text{count}_0 \} \\
\quad \text{def e}_1 = x[1]; \\
\quad \text{if (e}_1 \text{ == 1)} \{} \\
\quad \quad \text{count}_2 = \text{count}_1+1 \} \\
\quad \text{else} \{} \\
\quad \quad \text{count}_2 = \text{count}_1 \} \\
\quad \ldots \\
\quad \text{assert(x[4] == null) \\
\quad \text{count}_4 \\
\quad \text{def c = pop(...)} \\
\quad \text{assert(c == 2)}
\]
End-to-End Example

\[
\begin{align*}
\text{lambda pop (x) } & \{ \\
\quad \text{def count } = 0; \\
\quad \text{def e}_0 = x[0]; \\
\quad \text{if (e}_0 \text{ == 1) } \{ \text{ count}_1 = \text{ count}_0 + 1 \} \\
\quad \text{else } \{ \text{ count}_1 = \text{ count}_0 \} \\
\quad \text{def e}_1 = x[1]; \\
\quad \text{if (e}_1 \text{ == 1) } \{ \text{ count}_2 = \text{ count}_1 + 1 \} \\
\quad \text{else } \{ \text{ count}_2 = \text{ count}_1 \} \\
\quad \ldots \\
\quad \text{assert(x}[4] \text{ == null)} \\
\quad \text{count}_4 \\
\} \\
\text{def c } = \text{ pop(...)} \\
\text{assert(c } == 2) \\
\end{align*}
\]

\[
\begin{align*}
\text{x}_0 &= \text{ choose()}; \text{x}_1 &= \text{ choose()} \ldots \\
\text{def count}_0 = 0; \\
\text{def e}_0 = x_0; \\
\text{if (e}_0 \text{ == 1) } \{ \text{ count}_1 = \text{ count}_0 + 1 \} \\
\text{else } \{ \text{ count}_1 = \text{ count}_0 \} \\
\text{def e}_1 = x_1; \\
\text{if (e}_1 \text{ == 1) } \{ \text{ count}_2 = \text{ count}_1 + 1 \} \\
\text{else } \{ \text{ count}_2 = \text{ count}_1 \} \\
\ldots \\
\text{assert(x}_4 \text{ == null)} \\
\text{c } = \text{ count}_4 \\
\text{assert(c } == 2)
\end{align*}
\]
End-to-End Example

\[ x_0 = \text{choose}(); \ x_1 = \text{choose}(); \ ... \]
\[ \text{def} \ \text{count}_0 = 0; \]
\[ \text{def} \ e_0 = x_0; \]
\[ \text{if} \ (e_0 == 1) \{ \ \text{count}_1 = \text{count}_0 + 1 \} \]
\[ \text{else} \{ \ \text{count}_1 = \text{count}_0 \} \]
\[ \text{def} \ e_1 = x_1; \]
\[ \text{if} \ (e_1 == 1) \{ \ \text{count}_2 = \text{count}_1 + 1 \} \]
\[ \text{else} \{ \ \text{count}_2 = \text{count}_1 \} \]
\[ \ldots \]
\[ \text{assert}(x_4 == \text{null}) \]
\[ c = \text{count}_4 \]
\[ \text{assert}(c == 2) \]

\[ \text{ct}( \ (0 = x_0 \lor 1 = x_0) \land \]
\[ (0 = x_1 \lor 1 = x_1) \ldots \land \]
\[ 0 = \text{count}_0 \land \]
\[ x_0 = e_0 \land \]
\[ \text{ite} \ (1 = e_0, c_0 + 1 = c_1, c_0 = c_1) \land \]
\[ x_1 = e_1 \land \]
\[ \text{ite} \ (1 = e_1, c_1 + 1 = c_2, c_1 = c_2) \land \]
\[ \ldots \]
\[ \text{null} = x_4 \land \]
\[ c_4 = c \land \]
\[ 2 = c) \]
End-to-End Example

c\top ( 0 = x_0 \lor 1 = x_0 ) \land \\
( 0 = x_1 \lor 1 = x_1 ) \ldots \land \\
0 = \text{count}_0 \land \\

x_0 = e_0 \land \\
\text{ite} \ ( 1 = e_0, c_0 + 1 = c_1, c_0 = c_1 ) \land \\

1 = e_1 \land \\
\text{ite} \ ( 1 = e_1, c_1 + 1 = c_2, c_1 = c_2 ) \land \\
\ldots \\
\text{null} = x_4 \land \\
c_4 = c \land \\
2 = c \\

S_{\text{pop}}(x) = c_4
End-to-End Example

• Check Z3 code on course website for details
• You can run the code online: http://rise4fun.com/z3
Code optimization
Scope of optimizations

**Scope** of study for optimizations:

- **peephole**: look at adjacent instructions
- **local**: look at straight-line sequence of statements
- **global(intraprocedural)**: look at whole procedure
- **interprocedural**: look across procedures

Larger scope $\Rightarrow$ better optimization,  
but more cost & complexity
Style of optimizations

How is the program is improved

• **naïve**: no optimization after code generation
• **rewrite rules**: used in peephole optimization
• **instruction selection**: tree covering
• **deductive**: derive equivalent programs
• **superoptimization** and synthesis: search for a correct program
Naïve code generation
Naïve code generation

For each AST node, generate a sequence of instructions.

Pros: simple

Cons: suboptimal code

The same as bytecode generation (see previous lectures).

Generation of assembly code is the same but with labels.
Peephole optimization
Peephole optimizations

Replace a sequence of adjacent instructions with a more optimal sequence

\[
\begin{align*}
\text{sw } & \text{n8, 12(fp)} \\
\text{lw } & \text{n12, 12(fp)} \\
\text{⇒ } & \\
\text{sw } & \text{n8, 12(fp)} \\
\text{mv } & \text{n12, n8}
\end{align*}
\]

\[
\begin{align*}
\text{sub } & \text{sp, 4, sp} \\
\text{mov r1, 0(sp)} \\
\text{⇒ } & \\
\text{mov r1, -(sp)}
\end{align*}
\]
Instruction selection via tree coverage
Better code-gen rules

Rather than translating one AST node to an instruction sequence, we map multiple nodes to a sequence.