Hack Your Language!

CSE401 Winter 2016

Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

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 ($\vee$, $\wedge$, $\neg$)
  - Constraints: ct(E)
- The compiled program will be interpreted by the SAT solver

Example program

(a Int)
(b Int)
ct($a = b \wedge a < 10$)

Translates to: $\exists a,b \,.\, a = b \wedge a < 10$

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);
```

sym	value
a	10
b	10

Meaning of constraints

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

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

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

sym	value
a	10
b	10

sym	value
a	

Meaning of constraints

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

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

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

sym	value
a	10
b	10

sym	value
a	
b	

Meaning of constraints

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

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

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

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

sym	value
a	10
b	10

sym	value
a	10
b	10

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

sym	value
x	~~??~~ 4
r	8

sym	value
x	2
r	4

sym	value
x	10
r	20

sym	value
x	5
r	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_{CP}:
  - Identifiers
  - Expressions
  - Assignments
  - Conditionals
  - Function declarations and Calls
  - Loops
  - Arrays
  - `assert` and `choose`

Translating identifiers

- Booleans:

  `def b` → (b Bool)

- Integers:

  `def i` → (i Int)

- This assumes we have performed type inference

Expressions

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

- Example:

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

$\longrightarrow$

(a Int)
(b Int)
(c Int)
ct(a + b = c)

Translating statements

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

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

(a Int)
(b Int)
(c Int)

Collected constraints
a + b = c

Translating statements

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

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

```
(a Int)
(b Int)
(c Int)
```

Collected constraints
a + b = c
a = d

Translating statements

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

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

(a Int)
(b Int)
(c Int)
(d Int)

Collected constraints
c = a + b
a = d

ct(c = a + b $\wedge$ a = d)

Assignments

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

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

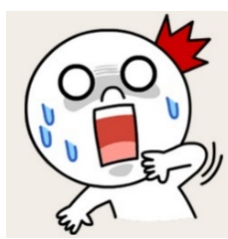
(c Int)

ct(10 = c $\wedge$ 20 = c)

c cannot be both 10 and 20!

```
def c;
c = 10;
def a;
a = c;
c = 20;
```

But we can't just throw away c = 10 either



Handling assignments

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

```
def c;
c = 10;
def a;
a = c;
c = 20;
```

→

$$\begin{array}{l}\texttt{def } c_1;\\ c_1 \texttt{ = 10;}\\ \texttt{def a;}\\ \texttt{a = } c_1;\\ \texttt{def } c_2;\\ c_2 \texttt{ = 20;}\end{array}$$

→

$(c_1 \text{ Int})$
$(a \text{ Int})$
$(c_2 \text{ Int})$

$ct(10 = c_1 \wedge 20 = c_2)$

- This is called Static Single Assignment (SSA) form
- Conversion is done using reaching definition analysis (see section last week)

Conditionals

```
if (E1) {
   E2
} else {
   E3
}
```

ct(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:

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

(a Int)
(b Int)
(c Int)

ct(ite(10 = a, 1 = b, 2 = c))

ite is just syntactic sugar:
$ite(c, e_1, e_2) \rightarrow (c \wedge e_1) \vee (\neg c \wedge e_2)$

Conditionals

But what about:

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

$\longrightarrow$

```
def a; def b₁; def b₂; def c;
if (a == 10) {
  b₁ = 1
} else {
  b₂ = 2
}
c = b // which b??
```

Both b_1 and b_2 can flow to c

Keeping track of branches

```
def a; def b₁; def b₂;
def c;
if (a == 10) {
  b₁ = 1
} else {
  b₂ = 2
}

c = phi(b₁, b₂)
```

Keeps track of which branch we came from

(a Int)
(b$_1$ Int)
(b$_2$ Int)
(c Int)

ct(ite$\boxed{(10 = a,}$ $1 = b_1$, $2 = b_2$) $\wedge$
ite$\boxed{(10 = a,}$ $b_1 = c$, $b_2 = c$))

optimize

(a Int) (b$_1$ Int) (b$_2$ Int) (c Int)

ct(ite(10 = a,
$1 = b_1 \wedge b_1 = c$,
$2 = b_2 \wedge b_2 = c$))

Function declarations and call

- There are no functions in our SAT language
  - Just inline function body at call site!

```
lambda dbl(x)
{ r = 2*x; r }
...
x = dbl(a)
y = dbl(b)
```

→

```
...
r1 = 2*a
x = r1
r2 = 2*b
y = r2
```

→

$$ct(\ 2*a = r1 \wedge r_1 = x \wedge 2*b = r_2 \wedge r_2 = y)\ )$$

- 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

```
def c;
c = 0;
for (e : 1)
{ c = c + e }

def a;
a = c
```

(c Int)
(a Int)
(e Int)

$ct(\, 0 = c \wedge c + e = c \wedge c = a)$

Unroll loops

- We assume loop runs for a fixed number of times

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

def a
a = c
```

```
c₀ = 0
if (l[0] != null) {
  e₀ = l[0]
  c₁ = c₀ + e₀
} else {
  c₁ = c₀
}
if (l[1] != null) {
  e₁ = l[1]
  c₂ = c₁ + e₀
} else {
  c₂ = c₁
}
a = c₂
assert(l[2] == null)
```

Ensure that loop indeed terminates!

Arrays

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

```
def a = {}
a[0] = 10
a[1] = 20

def c
c = a[0]
```

→

```
def a₀; def a₁
a₀ = 10
a₁ = 20

def c
c = a₀
```

→

$(a_0\ \mathrm{Int})$
$(a_1\ \mathrm{Int})$
$(c\ \mathrm{Int})$

$ct(\ 10 = a_0 \wedge\ 20 = a_1 \wedge\ a_0 = c)$

- 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`$_0$`,...,a`$_N$ where N = length of a
- Let's create a new construct in SAT for translation

  `a[i]` → mux(i, a_0 , …, a_N)

mux is just syntactic sugar:

mux(i, a_0 , …, a_N) →
(ite 0 = i, a_0,
    (ite 1 = i, a_1,
        (ite 2 = i, a_2, …)
)
)

Example

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

// a has length 2
c = a[i]
```

→

```
if (c == 0) {
  i₀ = 0
} else {
  i₁ = 1
}

i = phi(i₀, i₁)
c = a[i]
```

→

$(a_0\ \mathrm{Int})\ (a_1\ \mathrm{Int})$
$(i_0\ \mathrm{Int})\ (i_1\ \mathrm{Int})\ (i\ \mathrm{Int})\ (c\ \mathrm{Int})$

$\mathrm{ct}(\ \mathrm{ite}(0 = c, 0 = i_0, 1 = i_1) \wedge$
$\quad \mathrm{ite}(0 = c, i_0 = i, i_1 = i) \wedge$
$\quad \mathrm{mux}(i, a_0, a_1) = c)$

↓ optimize

$(a_0\ \mathrm{Int})\ (a_1\ \mathrm{Int})$
$(i_0\ \mathrm{Int})\ (i_1\ \mathrm{Int})\ (i\ \mathrm{Int})\ (c\ \mathrm{Int})$

$\mathrm{ct}(\ \mathrm{ite}(0 = c, a_0 = c, a_1 = c)\ )$

choose

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

```
def x
x = choose()
```

$\longrightarrow$

(x Int)

ct($1 = x \lor 2 = x \lor 3 = x \lor \ldots$)

- We first need to know the possible choices for x

End-to-End Example

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

unroll

```
lambda pop (x) {
  def count_0 = 0;
  def e_0 = x[0];
  if (e_0 == 1) { count_1 = count_0+1 }
  else { count_1 = count_0 }
  def e_1 = x[1];
  if (e_1 == 1) { count_2 = count_1+1 }
  else { count_2 = count_1 }
  …
  assert(x[4] == null)
  count_4
}
def c = pop(...)
assert(c == 2)
```

End-to-End Example

```
lambda pop (x) {
  def count = 0;
  def e₀ = x[0];
  if (e₀ == 1) { count₁ = count₀+1 }
  else { count₁ = count₀ }
  def e₁ = x[1];
  if (e₁ == 1) { count₂ = count₁+1 }
  else { count₂ = count₁ }
  …
  assert(x[4] == null)
  count₄
}
def c = pop(...)
assert(c == 2)
```

inline

$\longrightarrow$

rewrite arrays

```
x₀ = choose(); x₁ = choose() ...
def count₀ = 0;
def e₀ = x₀;
if (e₀ == 1) { count₁ = count₀+1 }
else { count₁ = count₀ }
def e₁ = x₁;
if (e₁ == 1) { count₂ = count₁+1 }
else { count₂ = count₁ }
…
assert(x₄ == null)
c = count₄
assert(c == 2)
```

End-to-End Example

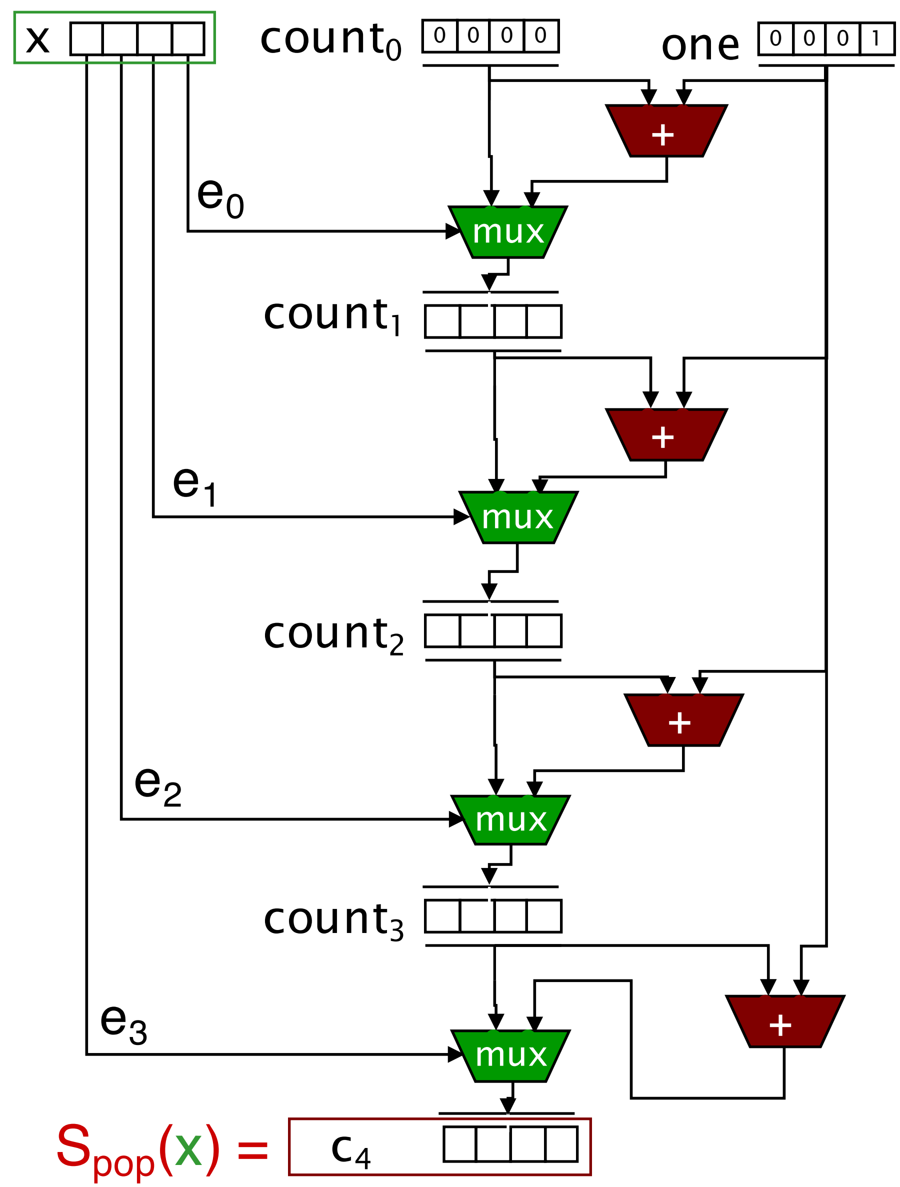
x_0 = choose(); x_1 = choose() ...
def $count_0$ = 0;
def e_0 = x_0;
if (e_0 == 1) { $count_1$ = $count_0$+1 }
else { $count_1$ = $count_0$ }
def e_1 = x_1;
if (e_1 == 1) { $count_2$ = $count_1$+1 }
else { $count_2$ = $count_1$ }
…
assert(x_4 == null)
c = $count_4$
assert(c == 2)

translate $\longrightarrow$

$$
\begin{aligned}
&ct(\,(0 = x_0 \vee 1 = x_0\,) \wedge \\
&\quad (0 = x_1 \vee\ 1 = x_1) \ldots \wedge \\
&\quad 0 = count_0 \wedge \\
&\\
&\quad x_0 = e_0 \wedge \\
&\quad ite\ (1 = e_{0,}\ c_0 + 1 = c_1,\ c_0 = c_1) \wedge \\
&\\
&\quad x_1 = e_1 \wedge \\
&\quad ite\ (1 = e_{1,}\ c_1 + 1 = c_2,\ c_1 = c_2) \wedge \\
&\quad \ldots \\
&\quad null = x_4 \wedge \\
&\quad c_4 = c \wedge \\
&\quad 2 = c)
\end{aligned}
$$

End-to-End Example

$$
\begin{aligned}
ct(\ & (0 = x_0 \lor 1 = x_0) \land \\
& (0 = x_1 \lor 1 = x_1) \ldots \land \\
& 0 = count_0 \land \\
& \\
& x_0 = e_0 \land \\
& ite\,(1 = e_0,\ c_0 + 1 = c_1,\ c_0 = c_1) \land \\
& \\
& 1 = e_1 \land \\
& ite\,(1 = e_1,\ c_1 + 1 = c_2,\ c_1 = c_2) \land \\
& \ldots \\
& null = x_4 \land \\
& c_4 = c \land \\
& 2 = c)
\end{aligned}
$$



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

<u>Scope</u> of study for optimizations:

- **peephole**: look at adjacent instructions
- **local**: look at straight-line sequence of statements
- **global**(**int*ra*procedural**): look at whole procedure
- **int*er*procedural**: 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.

each node code-generated individually

The same as bytecode generation (see previous lectures).

Generation of assembly code is the same but with labels.

Pros: simple

each node code-generated individually

Cons: suboptimal code

each node code-generated individually

Peephole optimization

Peephole optimizations

Replace a sequence of adjacent instructions with a more optimal sequence

```
sw $8, 12($fp)
lw $12, 12($fp)
```

⇒

```
sw $8, 12($fp)
mv $12, $8
```

```
sub sp, 4, sp
mov r1, 0(sp)
```

⇒

```
mov r1, -(sp)
```

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.