Lecture 16: Compilation to Constraints

Motivation for constraints-based reasoning
Compilation to constraints
Announcements

Milestones. Do these **today**, if you haven’t already:

- Contact the team that is reviewing your project
- Sign up for team meetings this week (see Piazza note)

HW4

- Last HW (we promise!)
- Help you prepare for the quiz
- Due this Sunday 11pm (no late days)
Announcements

Final quiz will be next Thursday during sections

- Make sure you attend! 😊
- Quiz to check if you have been attending class
- 2 pages of hand written notes (one-sided)
- Comprehensive, but mostly after-midterm stuff
- Sample exams have been posted on website
Class evaluations

Thanks for working with Jim and staying after class!

Got lots of good feedback:
  – Clarify specs while keeping the assignments open-ended
  – Timing for makeup lectures

Will incorporate them in future versions of 401
  – Looking for TAs interested in improving the courseware
Outline for today

Motivation for constraints compilation

Solver as interpreter

Compiling $401_{CP}$ to constraints
Motivation: Applications of Compiling to Constraint Solvers
Motivation

Imagine you could execute program $P$ backwards

Given an output $y$, compute an input $x$ such that $y = P(x)$.

You could do three exciting applications

**Bug finding:** find an input that fails an assertion

**Oracle execution:** find an input that satisfies all assertions

**Program synthesis:** complete of a program with holes
Finding bugs and security vulnerabilities

Input that fail the assertion exposes the bug.

```python
def main(x) {
    ...
    assert(c!=0)   // division-by-zero error
    a = b/c
}
```

Modeling assertions as program outputs:

i) Introduce the global variable `def retval = true`

ii) Rewrite `assert(E)` to `retval = retval && E`

iii) Make the main function return `retval`
Other applications of bug finding

Check *equivalence* of two programs. That is, do two programs produce the same values on all inputs?

We’ll use this to generate optimal code in a few slides
Oracle execution

Input passing all assertions solves the 8-queen puzzle:

def eight_queen(q1,q2,q3,q4,q5,q6,q7,q8) {
    assert(q1!=q2)     // q1, q2 not in same row
    assert(q1!=q2+1)   // q1, q2 not in same diagonal
    assert(q1+1 != q2) // q1, q2 not in same diagonal
    ...
}

Other applications of oracle execution

Given a buggy execution, find a value for a variable $x$ that rescues the execution (avoids the failure).

The value can be a hint on how to fix the bug.
Synthesize a parallel 4x4-matrix transpose

a functional (executable) specification:

```c
int[16] transpose(int[16] M) {
    int[16] T = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            T[4 * i + j] = M[4 * j + i];
    return T;
}
```

This example comes from a synthesis contest
Implementation idea: parallelize with SIMD

Intel SHUFPP (shuffle parallel scalars) SIMD instruction:

\[
\text{return} = \text{shufps}(x1, x2, \text{imm8} :: \text{bitvector8})
\]

Notes: two bits decide which element is chosen for each return vector slot.
Expression \(x[a::b]\) selects \(b\) elements starting at index \(a\).
High-level insight of the algorithm designer

Matrix $M$ transposed in two shuffle phases

**Phase 1:** shuffle $M$ into an intermediate matrix $S$ with some number of shuffle instructions

**Phase 2:** shuffle $S$ into a result matrix $T$ with some number of shuffle instructions

Synthesis with partial programs helps one to complete their insight. Or prove it wrong.
The SIMD matrix transpose, sketched

```c
int[16] trans_sse(int[16] M) implements trans {
    int[16] S = 0, T = 0;

    S[??::4] = shufps(M[??::4], M[??::4], ??);
    S[??::4] = shufps(M[??::4], M[??::4], ??);
    ...
    S[??::4] = shufps(M[??::4], M[??::4], ??);

    T[??::4] = shufps(S[??::4], S[??::4], ??);
    T[??::4] = shufps(S[??::4], S[??::4], ??);
    ...
    T[??::4] = shufps(S[??::4], S[??::4], ??);

    return T;
}
```

Phase 1

Phase 2
The SIMD matrix transpose, sketched

```c
int[16] trans_sse(int[16] M) implements trans {
    int[16] S = 0, T = 0;
    repeat (??) S[??::4] = shufps(M[??::4], M[??::4], ??);
    repeat (??) T[??::4] = shufps(S[??::4], S[??::4], ??);
    return T;
}
```

```c
int[16] trans_sse(int[16] M) implements trans { // synthesized code
    S[4::4] = shufps(M[6::4], M[2::4], 11001000b);
    S[0::4] = shufps(M[11::4], M[6::4], 10010110b);
    S[12::4] = shufps(M[0::4], M[2::4], 10001101b);
    S[8::4] = shufps(M[8::4], M[12::4], 11010111b);
    T[4::4] = shufps(S[11::4], S[1::4], 10111100b);
    T[12::4] = shufps(S[3::4], S[8::4], 11000011b);
    T[8::4] = shufps(S[4::4], S[9::4], 11100010b);
    T[0::4] = shufps(S[12::4], S[0::4], 10110100b);
}
```

From the contestant email:

Over the summer, I spent about 1/2 a day manually figuring it out.

Synthesis time: <5 minutes.
Key ideas

Many programming questions can be reduced to the question “is there an input $x$ such that $P(x) = y$?”

Sadly, these questions are in general undecidable. No algorithm exists

We’ll sidestep this in one of two ways:

1) Restrict what programs we consider (eg, no loops)
2) Restrict what inputs that we consider (eg 4-bit ints)
Reducing Programming Questions to Constraint Solving

overview of technical ideas
Program as a logical formula

Formula $S_p(x,y)$ holds iff program $P(x)$ outputs value $y$

program: $f(x) \{ \text{ return } x + x \}$

formula: $S_f(x,y): y = x + x$

We introduced variable $y$ to represent $f$’s return value
With program as a formula, solver is versatile

Solver as an **interpreter**: given $x$, evaluate $f(x)$

$$S(x, y) \land x = 3$$

solve for $y$

$y \mapsto 6$

Solver as a program **inverter**: given $f(x)$, find $x$

$$S(x, y) \land y = 6$$

solve for $x$

$x \mapsto 3$

Possible because constraints are non-directional

unlike assignments
Synthesis as constraint solving

\[ S_P(x, h, y) \text{ holds iff sketch } P[h](x) \text{ outputs } y. \]

\[
\begin{align*}
\text{spec}(x) & \{ \text{return } x + x \} \\
\text{sketch}(x) & \{ \text{return } x \ll ?? \} \\
\text{sketch}(x, h) & \{ \text{return } x \ll h \}
\end{align*}
\]

\[ S_{\text{sketch}}(x, y, h): y = x \times 2^h \]

The solver computes \( h \), thus synthesizing a program correct for the given \( x \) (here, \( x=2 \))

\[ S_{\text{sketch}}(x, y, h) \land x = 2 \land y = 4 \quad \text{solve for } h \quad h \mapsto 1 \]

Sometimes \( h \) must be constrained on several inputs

\[
\begin{align*}
S(x_1, y_1, h) & \land x_1 = 0 \land y_1 = 0 \land \\
S(x_2, y_2, h) & \land x_2 = 3 \land y_2 = 6 \quad \text{solve for } h \quad h \mapsto 1
\end{align*}
\]
Inductive synthesis

Our constraints encode **inductive synthesis:**

- We ask for a program $P$ correct on a few inputs.
- We hope (or test, verify) that $P$ is correct on rest of inputs.

How to select suitable inputs?

- Verify a candidate program. If it fails verification, the counterexample (input) is added as an input to synthesis
Key ideas

Programs as non-directional formulas (constraints).

Solver solves constraints, acting as a forward and backward interpreter.
The language of constraints
this is our target language
Constraint solver

Given a set of constraints, the solver

i. finds a solution (often one of many) or
ii. proves that there’s no solution or
iii. runs out of memory or times out 😞

We’ll be using a SAT solver

– it solves the SAT problem (satisfiability of Bool formulas)
– amazingly efficient algorithms now exist
Language of constraints

The language of constraints is our *target language* that is, we compile programs to this language.

This language is idiosyncratic (like JS and cps 😊) so we’ll need a special compilation strategy.

This is what we explain on the next few slides.

We’ll also build an abstraction layer (circuits) over the low-level SAT constraints.
SAT solver

**Input** is a formula in CNF (conjunctive normal form). **Output** is UNSAT or SAT + solution.

\[(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2 \lor x_3) \land \neg x_1\]

SAT solver

- **UNSAT**
- **SAT**

**Solution** \[x_1 = , x_2 = , x_3 =\]
Constraints as circuits

It is sometimes easier to think of boolean constraints as **circuits** (these can be translated to CNF).

Circuit = each value is computed exactly once.
Limitations of boolean circuit constraints

Each variable is computed once ("single assignment")

- we can’t reassign constraint variables

  ==> 

- need multiple constraint variables per program variable

There are no loops

- no recursion either

  ==> 

- need to unroll loops and recursion into circuit form

Bounded unrolling means we can’t execute arbitrary inputs
Turning a program into a circuit

\[ S_{\text{pop}}(x) = \]

\[ W = 4 \]

\[ \text{int pop (bit[W] x) \{} \]
\[ \quad \text{int count} = 0; \]
\[ \quad \text{for(int } i=0; i<W; i++) \]
\[ \quad \quad \text{if (x[i])} \]
\[ \quad \quad \quad \text{count}++; \]
\[ \quad \text{return count;} \]
\[ \} \]
Summary

- Compiling to constraints offer a number of benefits
- Constraint programming differs from imperative code
  - Programs are non-directional
  - All variables get one single value
  - No loops, assignments, and recursions
- Constraint programs are represented using circuits