Hack Your Language!

CSE401 Winter 2016
Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

Lecture 15: Pointer analysis and Intro to Solver-Aided DSLs

Andersen’s algorithm
Constraint programming
Announcements

• Midterm regrade requests due by next Tuesday

• Project milestones
  – We have posted comments to all projects
  – We will pair up projects later this week
Today

- More pointer analysis
- Intro to constraint programming
Flow analysis of pointers
(continued)

(See Lec 14 slides)
Constraint Programming
Constraint Programming

• A programming paradigm where relationships among program vars are stated using constraints

• Constraints are expressed using core language constructs

• Can be used to model many problems
  – Constraint satisfaction problems (CSPs)
  – Linear programming
  – ...

• “Running” the program → an assignment of the program var values such that all constraints are satisfied
  – Language runtime comes with solver
Languages

• Prolog (for constraint logic programming)

• Other languages with support for constraints
  – Wolfram language
  – Verilog
  – Babelsberg / Wallingford

• We will look at how to compile such languages
CP Examples

- Linear programming using the Wolfram language

Minimize $x + y$, subject to constraint $x + 2y \geq 3$ and implicit non-negative constraints:

```math
\text{In}[1]:= \text{LinearProgramming}[[1, 1], \{(1, 2), \{3\}]]
\text{Out}[1]= \{0, \frac{3}{2}\}
```

Solve the problem with equality constraint $x + 2y = 3$ and implicit non-negative constraints:

```math
\text{In}[2]:= \text{LinearProgramming}[[1, 1], \{(1, 2), \{3, 0\}]]
\text{Out}[2]= \{0, \frac{3}{2}\}
```
CP Examples

• SAT
  – Given logic propositions containing variables, find an assignment of each variable to \{T,F\} such that the overall proposition evaluates to T
  – Ex: \((A \lor B \lor \neg C) \land (\neg A \lor D)\)

• N-queens problem
  – Place N queens on a chess board such that they do not attack each other
CP Examples

- Scheduling / Planning
- Resource allocation / optimization
- Visualization layout
- Natural language processing
- Molecular biology / genomics
- VLSI design
Constraint satisfaction problems (CSPs)

- Finite set of variables \((x, y, z, \ldots)\)
- Each variable can range over a nonempty set of values \((x \in \{1, 2, 3\}, y \in \{10, 42\}, \ldots)\)
- Finite set of constraints \((C_1, C_2, \ldots)\)
  - Each constraint limits the possible values that the variables can take

Our goal is to find an assignment of variable values that satisfy all constraints
Example: $$$

Assign distinct digits to the following letters such that:

\[ \text{SEND} + \text{MORE} = \text{MONEY} \]
Example: $$$

- Formulate as CSP:
- Variables: \{ S, E, N, D, M, O, R, Y \}
- Constraints:
  \[
  S \in \{0, 1, 2, 3, \ldots, 9\};
  E \in \{0, 1, 2, 3, \ldots, 9\};
  (1000*S + 100*E + 10*N + D) +
  (1000*M + 100*O + 10*R + E) =
  (10000*M + 1000*O + 100*N + 10*E + Y)
  S \neq 0
  M \neq 0
  S \neq E; E \neq N; N \neq D; \ldots
  \]

- Solution: 9567 + 1085 = 10652
The $401_{\text{CP}}$ Language
Extending 401 with constraints
We will develop $401_{\text{CP}}$ in a few steps

1. Add construct to $401_{\text{CP}}$ to express constraints

2. Using $401_{\text{CP}}$ to write programs that enable verify a given solution
   - We will also need a way to compile and evaluate $401_{\text{CP}}$ programs

3. Add construct to $401_{\text{CP}}$ to enable us search for solutions in addition to verify
Our 401 language can be readily used for CP

• We just need to add the `assert` statement to 401

  – `assert(E)`: add a constraint stating that E is true

  – Multiple constraints can be expressed using multiple `assert` statements
```javascript
// v = { S, E, N, D, M, O, R, Y }
lambda check (v) {
    for (i in {0,1,...,7}) {  // range of each variable
        assert(v[i] == 0 || v[i] == 1 || ... || v[i] == 9)
    }

    for (i in {0,1,...,7}) {  // no two values are equal
        for (j in {0,1,...,7}) {
            if (i != j) {
                assert(v[i] != v[j])
            }
        }
    }

    assert(v[0] != 0 && v[4] != 0)  // S != 0 and M != 0

    assert(1000*v[0] + 100*v[1] + 10*v[2] + v[3]) // the actual math
}

// check solution
check({0=2, 1=3, 2=9, ...})
```
Evaluating 401\textsubscript{CP} programs

- Evaluate `assert(E):`
  \[
  \text{tmp} = E; \\
  \text{if} \ (\neg \text{tmp}) \ {\text{error}}
  \]

- But our interpreter can also evaluate 401\textsubscript{CP} programs using a solver
  - First we need to compile our program into a logical formula
  - Then we run the program by sending the compiled formula to a solver
Program as logical formula

- We need to translate each `assert` into a logical formula
- We will do this manually for now

```plaintext
for (i in {0,1,...,7}) {  // range of each variable
    assert(v[i] == 0 || v[i] == 1 || ... || v[i] == 9)
}

→

let v0, v1,..., v7 in
  (v0=0 ∨ v0=1 ∨ v0=2 ...) ∧ (v1=0 ∨ v1=1 ∨ v1=2 ...) ∧ ...
```
Program as logical formula

\[
\begin{align*}
&\text{for } (i \text{ in } \{0,1,\ldots,7\}) \{ \\
&\quad \text{// no two values are equal} \\
&\quad \text{for } (j \text{ in } \{0,1,\ldots,7\}) \{ \\
&\quad \quad \text{if } (i \neq j) \{ \text{assert}(v[i] \neq v[j]) \} \\
&\quad \}\}
\end{align*}
\]

\[v_0 \neq v_1 \land v_0 \neq v_2 \land v_0 \neq v_3 \ldots \land v_1 \neq v_2 \land v_1 \neq v_3 \ldots\]
Program as logical formula

• Finally
  
  \[
  \text{lambda check}(v) \{ \quad \}
  \]
  
  \[
  \Rightarrow
  
  F(v_0,v_1,v_2..., r): r = (v_0=0 \lor v_0=1 \lor v_0=2 ...) \land ... 
  
  \]

• And
  
  \[
  \text{check}([0=9, 1=5, 2=6, ...])
  \]
  
  \[
  \Rightarrow
  
  F(v_0,v_1,v_2..., r) \land v_0=9 \land v_1=5 \land v_2=6 
  
  \]

• Asking the solver to solve for \( r \Rightarrow r = \text{true} \)
  
  – i.e., our guess worked
But why bother?

• We could just evaluate extend our 401 interpreter to handle assert

• Solver can find missing values s.t. the formula to evaluate to true:
  – We can use it to run our program backwards!
  – This is the first step towards program synthesis
From verification to synthesis
Using the solver as a program inverter

• Back to our example

check(\{0=9, 1=5, 2=6, \ldots\})

\rightarrow

F(v_0,v_1,v_2..., r) \land v_0=9 \land v_1=5 \land v_2=6

Solve for r \rightarrow r = true

• What if we instead write

F(v_0,v_1,v_2..., r) \land r = true

Solve for v_0,v_1,v_2 \rightarrow v_0=9, v_1=5, v_2=6

We just executed the program backwards!
How can we utilize this in $401_{\text{CP}}$?

• Let’s add a new construct: choose
  – Semantics: imagine a code fairy picks a value for each choose such that the resulting formula evaluates to true
  – Reality: Ask the solver to a value
    – The code fairy is known as an oracle
    – This is called angelic execution
One line change to the previous program

```cpp
// Find a solution
check({0=choose(), 1=choose(), 2=choose(), ...})
```

- We translate the program to a formula the same as before
- We now ask the solver to find the values of v₀, v₁, ...

- We still need:
  - A systematic way to compile \( 401_{\text{CP}} \) constructs into logical formulas
  - A way to come up with angelic values that can scale to large programs
A More Complex Example
Sudoku in $401_{\text{CP}}$
lambda sudoku(puzzle, rows, cols)
    // all digits between 1 and n^2
    for (r in {0,...,rows}) {
        for (c in {0,...,cols}) {
            assert(puzzle[r][c] >= 1 && puzzle[r][c] <= square(n))
        }
    }
    // all digits in a row are different
    for (r in {0,...,rows}) {
        for (c1 in {0,...,cols}) {
            for (c2 in {0,...,cols}) {
                if (c1 != c2) { assert(puzzle[r][c1] != puzzle[r][c2]) }
            }
        }
    }
    // likewise for all digits in a column
    ...
    // all digits in a subgrid are different
    ...
}
Define a puzzle and check a solution

```python
def p0=puzzle
    "000000010400000000020000000000050407008000300001090000300400200050100000000806000", 9, 9))
```

Evaluating the program will return false (i.e., not a valid solution)
From Sudoku checker to Sudoku solver

Simply replace each 0 (empty cell / unknown) with a call to the oracle using choose

```python
lambda sudoku(puzzle, rows, cols)
    for (r in {0,...,rows}) {
        for (c in {0,...,cols}) {
            if (puzzle[r][c] == 0) { puzzle[r][c] = choose() }
        }
    }
...
```
Solving

choose will select values that will pass the check for a solution

\texttt{sudoku(p0, 9, 9)}

\begin{array}{cccccccc}
7 & 9 & 3 & 6 & 8 & 4 & 5 & 1 \\
4 & 8 & 6 & 5 & 1 & 2 & 9 & 3 \\
1 & 2 & 5 & 9 & 7 & 3 & 8 & 4 \\
9 & 3 & 2 & 7 & 5 & 1 & 6 & 8 \\
5 & 7 & 8 & 2 & 4 & 6 & 3 & 9 \\
6 & 4 & 1 & 3 & 9 & 8 & 7 & 2 \\
3 & 1 & 9 & 4 & 6 & 5 & 2 & 7 \\
8 & 5 & 7 & 1 & 2 & 9 & 4 & 6 \\
2 & 6 & 4 & 8 & 3 & 7 & 1 & 5 \\
\end{array}