SuperC: Parsing All of C by Taming the Preprocessor

Paul Gazzillo and Robert Grimm
New York University

Problem: Parsing All of C

We need better C tools
- Linux x86 is large and complex
- Need source code browsers
- 7,500+ compilation units, 5.5 million lines
- Need bug finders
- 1,000 found by static checkers [Chu et al., ASPLOS '10]
- Need refactoring tools
- 150+ errors due to interface changes [Rudolph et al., Eusys '96]
- These tools all need to parse C first

C source code written in both C and the preprocessor
- Source code contains many programs
- Macros expand to arbitrary C fragments
- Directives appear between arbitrary C fragments
- Turning on all configuration variables yields only 80% of code [Tartler et al., OSR '11]

Solution Approach

Expands macros and includes headers
Preserves conditionals!
Creates an AST for all configurations

How SuperC Works

The Preprocessor

Conditionals Invade the Preprocessor
- The preprocessor leaves conditionals in place
- Conditionals then compose with most preprocessor operations
- Many operations require hoisting

The Power of Hoisting
- Works on: token-pasting, stringification, includes, conditional expressions, macros
- Iterates over conditional branches
- Recurses into nested conditionals
- Doubles tokens across inner-most branches

The Parser

Parsing All Configurations
- Forks subparsers at conditionals
- Merges subparsers in the same state after conditionals
- Joins AST subtrees with static choice nodes
- Preserves mutually exclusive configurations

History Repeats Itself: LR Subparsers
- Organizes state in stacks
- Early forking and merging with DAG
- Is table-driven
- Good performance
- Reuses existing tools and grammars
- The good: most complexity is in table generation
- The bad: shift-reduce & reduce-reduce conflicts

Evaluation

Number of Subparsers Used at Any Given Point
while Parsing Linux x86

<table>
<thead>
<tr>
<th>Subparsers Used</th>
<th>Performance Across Compilation Units of Linux x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 5</td>
<td>Max: 10.4s</td>
</tr>
<tr>
<td>6 - 10</td>
<td>Max: 9.31s</td>
</tr>
<tr>
<td>11 - 20</td>
<td>Max: 9.31s</td>
</tr>
<tr>
<td>21 - 30</td>
<td>Max: 9.31s</td>
</tr>
<tr>
<td>31 - 35</td>
<td>Max: 9.31s</td>
</tr>
<tr>
<td>36 - 40</td>
<td>Max: 9.31s</td>
</tr>
</tbody>
</table>

Follow-set forks fewer subparsers

When to Fork Subparsers?
- Naive strategy: fork on every conditional branch
- Conditions are 40 levels deep, 10 in a row
- Our forking strategy: token follow-set
- All tokens reachable from current position

The follow-set algorithm in action

Find first token of each branch
Recurse into nested conditionals
Stop after all configurations

Follow-set supports further optimization
- Shared reductions
- Reduce one stack for many follow-set tokens before forking
- Limits redundant work by subparsers
- Lazy shifts
- Only fork tokens in the nearest conditional
- Limits number of subparsers needed
- Early Reduces
- Pick reducing subparser before a shifting one
- Improves chances of merging