Astec: A New Approach to Refactoring C

E.M. Hielscher

28 November 2007
Problems with CPP

- Purely lexical and thus can’t be parsed with a standard parser
- Makes source code analysis difficult, as most analysis tools operate on syntax trees
Current Analysis Tools

Current analysis tools do one of three things:

• Ignore the preprocessor and perform simple lexical analysis (too simple)
• Operate on post-processed program text (can’t generate readable output)
• Try to emulate the preprocessor while doing analysis (use ad-hoc heuristics)
• Idea: Replace CPP with an AST-based macro language (Astec)

• Advantages:
  • Refactoring a syntactic macro language is much easier as original source code is parsed and macros analyzed directly
  • Many common errors made when using CPP are eliminated (such as the need to surround arguments with parentheses)
  • After #ifdefs are evaluated, only one possible configuration of the source is processed by current tools. With Astec, all configurations can be parsed at once
  • More readable error messages near macro usage or in macro definitions are generated
  • Macros become first-class objects and can be refactored just as can functions or variables
Macroscope: Translates CPP to Astec

- In order to get the benefits of Astec for existing code, a translator is provided
- Able to translate large open source projects (e.g. a minimally-configured Linux kernel) with almost no manual intervention
- Detects subtle macro errors
Astec Design Considerations

- Astec must be backward compatible with C (i.e. C code without any preprocessor directives must be valid Astec code)
- All commonly used CPP idioms must be expressible in Astec, including those which enhance the power of C
- None of the potential errors involving precedence or side-effects in CPP should be expressible in Astec
- Limit scope to C, ignoring C++ for now
# Astec: A New Approach to Refactoring C

## E.M. Hielscher

## Common CPP Idioms Expressible in Astec

<table>
<thead>
<tr>
<th>Feature</th>
<th>CPP example</th>
<th>ASTEC example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include files</td>
<td><code>#include &quot;header.h&quot;</code></td>
<td><code>@import &quot;header.h&quot;</code></td>
</tr>
<tr>
<td>Conditional compilation</td>
<td><code>#if defined(X) &amp;&amp; Y &gt; 3</code></td>
<td><code>@if(@defined(X) &amp;&amp; Y &gt; 3) int z;</code></td>
</tr>
<tr>
<td>Macros</td>
<td><code>#define M(x) ((x)+2)</code></td>
<td><code>@macro int M(int x) = x+2;</code></td>
</tr>
<tr>
<td></td>
<td><code>#define RETURN(x) return x;</code></td>
<td><code>@macro RETURN(x) { return x; }</code></td>
</tr>
<tr>
<td></td>
<td><code>#define u32 unsigned int</code></td>
<td><code>@type u32 = unsigned int;</code></td>
</tr>
<tr>
<td>Dynamic scoping</td>
<td><code>#define Z(ptr) ptr-&gt;x-&gt;y-&gt;z</code></td>
<td><code>@macro char *Z([T *ptr]) = ptr-&gt;x-&gt;y-&gt;z;</code></td>
</tr>
<tr>
<td>Reference arguments</td>
<td><code>#define M(x, y) x = 2*(y);</code></td>
<td><code>@macro M(int &amp;x, int y) { x = 2*y; }</code></td>
</tr>
<tr>
<td>First-class types</td>
<td><code>#define SIZE(T) \n (sizeof(T) + sizeof(int))</code></td>
<td><code>@macro SIZE(@type T) = sizeof(T) + sizeof(int);</code></td>
</tr>
<tr>
<td>First-class statements</td>
<td><code>#define FOR_EACH(x, list) \n for (x=(list); x; x = x-&gt;next)</code></td>
<td><code>@macro FOR_EACH(x, List *list, @stmt S) { for (x=(list); x; x = x-&gt;next) S; }</code></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td><code>... FOR_EACH(it, list, { ... });</code></td>
</tr>
<tr>
<td>First-class names</td>
<td><code>#define offsetof(T, field) \n (size_t)(&amp;(T*)0)-&gt;field</code></td>
<td><code>@macro offsetof(@type T, @name field) = (size_t)(&amp;(T*)0)-&gt;$field</code></td>
</tr>
<tr>
<td>Declaration “decorators”</td>
<td><code>#define NONNULL(args...) \n __attribute__((nonnull(args)))</code></td>
<td><code>@decorator NONNULL(args...) = __attribute__((nonnull(args)))</code></td>
</tr>
<tr>
<td>Declaration macros</td>
<td><code>#define DECLARE_LIST(name) \n List name = { .head = NULL }; ... DECLARE_LIST(hello)</code></td>
<td><code>@module DECLARE_LIST(@name name) { List $name = { .head = NULL }; ... @import DECLARE_LIST($hello);</code></td>
</tr>
<tr>
<td>Stringization and concatenation</td>
<td><code>#define DECLARE_STRING(name, s) \n char *name##_str = $s;</code></td>
<td><code>@module DECLARE_STRING(@name name, @expr s) { char *$(name ## $_str) = @stringize(s); }</code></td>
</tr>
</tbody>
</table>
Some Differences between CPP and Astec

- @if statements must expand to declarations, statements, or expressions
- Macros must expand to a complete syntactic unit of a single one of the following types: expression, statement, type, declaration, or attribute of a declaration or type. Different macros must be written which expand to the same syntactic unit but are used as different types
- Macro arguments are evaluated exactly once
Error-Prone Constructs of CPP

All solved by Astec:

- Unparenthesized Macro Bodies
- Unparenthesized Formal Macro Arguments
- Multiple Formal Argument Uses
- Type Macros with Pointers
- Dangling Semicolons
- Macros Swallowing Else
Translation with Macroscope

- Macro definitions aren’t translated; rather, each use of a macro is translated separately
- Afterward, all translations of a given macro are merged and an attempt is made at unifying similar translations
- Original formatting is preserved when possible
- Some user intervention is required, but kept to a minimum
- Type annotations not automatically generated in current version
Extracting Macros

- Macroscope preprocesses its input, keeping track of the tokens in macros both before and after expansion.
- Next, the input is parsed by a C parser.
- Next, Macroscope attempts to back out each macro expansion, cutting out the smallest complete subtree containing the post-tokens.
- This subtree becomes a macro translation, and macroscope inserts a call to a newly generated macro.
- Arguments to Astec macros which weren’t arguments to their CPP counterparts may be extracted, since the subtrees may contain additional tokens.
- Only syntactically complete arguments are extracted, which helps in detecting some bugs.
Discussion of Macroscope

- Macros with empty bodies and no post-tokens require insertion of special placeholder tokens
- Unused macro arguments aren’t extracted (doesn’t affect semantics, but may be problematic if programmer still wants them)
- Only extracts macros actually used by a program - problematic for libraries. Expectation is to use the libc test suite to extract all interface macros from libc
Manual Macroscope Hints

- The following are various manual hints that Macroscope may require to complete a translation:
  - Ignore - ignore the macro (e.g. C++ macros)
  - Parser Hints - when the parser doesn’t understand its input, e.g. when it can’t determine whether a placeholder token should be a declaration, statement, or attribute
  - User-specified translations
  - Macro changes - manual changes to strange macros, e.g. macros which allow code to support both K&C and ANSI styles
Translating Extra-Linguistic Features

- Include Files - simply keep track of which tokens correspond to a file
- Dynamic Scoping - postprocesses generated macros to ensure that all variables and referenced macros are declared. If a macro depends on another macro which hasn’t been declared, a macro prototype is inserted. If a variable or type is free in a macro, the argument is declared implicit
- Reference Arguments - any arguments modified by a macro are declared as reference arguments
- First-class types, statements, and names - extracted just like other arguments
Error Detection

• According to Ernst et al, over 20% of macros may contain errors
• Two types: true errors and potential errors
• True errors are errors in use of a macro, while potential errors are macros which have the potential to be misused
• Macros which have potential errors but are used correctly simply get translated to safe Astec macros
• Warnings are printed for each true error, but the program’s semantics is left unchanged to guard against false positives - user intervention required to fix them
Error Example

#define M(x) (x*2)
int x = M(3);
int y = M(1+2);

Only the assignment to y is a true error.
Translating Conditional Directives

- Expecting a Macroscope user to have all OS config files available so as to explore all conditional directives is infeasible.
- At the moment, Macroscope only translates C99 under Linux and glibc.
- Users can explicitly instruct Macroscope to explore both sides of a conditional directive, such as DEBUG to explore both production and debug modes.
Evaluation

- Tested on various FOSS programs
- OpenSSH (55,000 lines) translated with no hints, minimally configured Linux (150,000 lines) translated with one hint per 10000 lines
- Warnings generated for the following cases:
  - Extractions which included more tokens than the original directives
  - Multiple translations of a macro
  - Unused arguments for a macro not being extracted
  - Moved macro definition - Astec only allows macros to be defined at the top level
## Evaluation - Warnings

<table>
<thead>
<tr>
<th>Warning</th>
<th>gzip 1.2.4</th>
<th>rcs 5.7</th>
<th>OpenSSH 3.9p1</th>
<th>Linux 2.6.10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7,324 lines</td>
<td>17,178 lines</td>
<td>55,153 lines</td>
<td>163,154 lines</td>
</tr>
<tr>
<td>imperfect macro</td>
<td>7 (0.5% of macro defs)</td>
<td>10 (0.6%)</td>
<td>10 (0.1%)</td>
<td>88 (0.6%)</td>
</tr>
<tr>
<td>imperfect ifdef</td>
<td>18 (2.0% of #ifdefs)</td>
<td>59 (5.8%)</td>
<td>41 (1.9%)</td>
<td>62 (2.7%)</td>
</tr>
<tr>
<td>imperfect #include</td>
<td>0 (0.0% of #includes)</td>
<td>0 (0.0%)</td>
<td>2 (0.1%)</td>
<td>3 (0.1%)</td>
</tr>
<tr>
<td>multiple translations</td>
<td>3 (0.2% of macro defs)</td>
<td>0 (0.0%)</td>
<td>9 (0.1%)</td>
<td>76 (0.5%)</td>
</tr>
<tr>
<td>argument failure</td>
<td>1 (0.1% of macro defs)</td>
<td>0 (0.0%)</td>
<td>8 (0.1%)</td>
<td>82 (0.5%)</td>
</tr>
<tr>
<td>unused argument</td>
<td>14 (0.9% of macro defs)</td>
<td>7 (0.4%)</td>
<td>8 (0.1%)</td>
<td>240 (1.6%)</td>
</tr>
<tr>
<td>moved macro definition</td>
<td>1 (0.1% of macro defs)</td>
<td>0 (0.0%)</td>
<td>18 (0.2%)</td>
<td>58 (0.4%)</td>
</tr>
</tbody>
</table>
Refactoring

- 1000-line prototype refactoring tool provided which generates human-readable code
- Able to find and rename variables or functions based on their names or arguments - e.g. changing uses of `strcpy` to use the safe `strlcpy` function. In this case, the tool attempts to determine array sizes statically and succeeded 75% of the time in the case of `gzip`
- Able to change algorithm implementations which expose macro interfaces, unlike traditional tools