PCp^3: A C Front End for Preprocessor Analysis and Transformation

Greg J. Badros
23 October 1997

Qualifying Exam Presentation
University of Washington
Department of Computer Science
and Engineering

Simple C program

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello world\n");
  return 0;
}
```

- Uses preprocessor
- Outputs nothing if compiled with:
  `gcc -Dprintf(x)= hello.c -o hello`
- C language requires preprocessor

What...

you see is not what **the compiler gets**

```
if (STREQ (&(argv[arg_index][1]),
                    long_args[i].name))

...

else {
#if defined (HISTORY)
#  if defined (BANG_HISTORY)
    history_expansion = 0;
#  endif
    remember_on_history = 0;
#endif /* HISTORY */
    interactive_shell =
        startup_state =
        interactive = 0;
    no_line_editing = 1;
#if defined (JOB_CONTROL)
    job_control = 0;
#endif /* JOB_CONTROL */
}
```

cpp

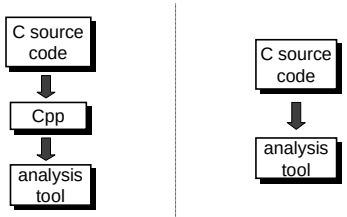
```
if ((( &(argv[arg_index][1]) )[0] ==
   (long_args[i].name )[0] &&
    strcmp( &(argv[arg_index][1]),
    long_args[i].name ) == 0) )
...

else {


    remember_on_history = 0;

    interactive_shell =
        startup_state =
        interactive = 0;
    no_line_editing = 1;

    job_control = 0;

}
```

- Eliminates useful abstractions
- Removes preprocessor directives
- Restricts artifact to a single version configuration

Outline

- Problems with C preprocessor
- Common approaches
- PCp^3 approach
  - integrate parser and preprocessor
  - mapping between unprocessed and preprocessed source
- Automating transformation of preprocessor uses to language features
- Future Work
- Summary

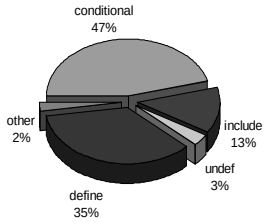
Common approaches



✓	Precise analysis	
✓	Reuses existing front-end	
	Handles non-syntactic programs	✓
	Processes entire source code	✓
	Analyzes same view as engineer	✓

How pervasive is C preprocessor use?

10% of lines contain a preprocesor directive



28% of lines must expand at least one macro

40% of lines have some cpp dependence

The PCp³ Approach

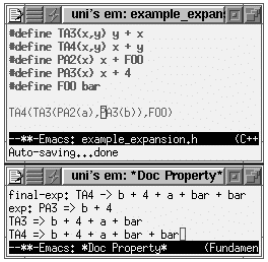
Parser + Cpp + Perl = PCppP = PCp³

- Allow views of both unprocessed and preprocessed code
- Control source code processing so the analysis does not have to
- Provide powerful scripting language for easy development and refinement of analyses

6

Visualizing the mapping

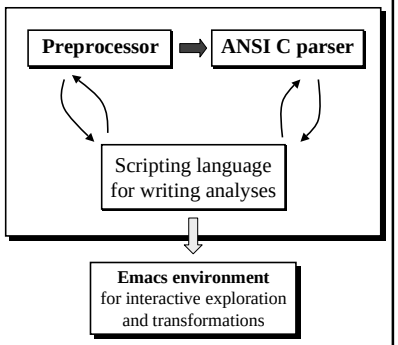
Analysis output Emacs lisp code to mark-up the source code with arbitrary annotations



doc-property-mode lets programmer interact with the source and its annotations.

7

The PCp³ Infrastructure



8

Event hooks

- Preprocessor announces
  - directives
  - definitions
  - expansions
  - ... and more
- Parser announces
  - typedefs
  - declarations
  - function calls
  - ... and more

9

Backcalls

- Expose data structures
  - preprocessor's name table
  - parser's symbol table, AST
  - current processing position
- Control processing
  - add code to be processed (e.g., otherwise unseen branches)
  - alter the parser's state
  - introduce and exit scopes

10

Example PCp³ analysis

```
use pcp3;
AddHook("TYPEDEF",\&Typedef);
AddHook("EXIT",\&Exit);

sub Typedef { my ($name) = @_;
  push @typedefs, $name;
  $num_typedefs{pcp3::Fname()}++;
}

sub Exit { my ($return_value) = @_;
  print "Typedefs:\n",
        join("\n",sort @typedefs), "\n\n";
  print "Number per file:\n";
  foreach my $fname (sort keys %num_typedefs) {
    print $fname, " had ",
          $num_typedefs{$fname}, " typedefs\n";
  }
}
```

11

C++ Language Features to replace uses of Cpp

- Constants (improved over C's)
- Reference parameters
- Inline functions
- Templates (generic code)
- Boolean type
- Enumerations
- Name spaces

12

Replacing macros with language-proper constants

```
#include <stdio.h>

#define CCD2
#ifdef CCD1
#  define FOO 1 ①
#endif
#ifdef CCD2
#  define FOO 2 ①
#endif

#define FUD 3
#define BAR 4 ②
#define BAZ 5
#define BING 6

int
main(char **argv, int argc) {
  int fud = FUD;
  int BONG = -1;
  int rgi[BING];③
  printf("%d, %d, %d, %d, %d, %d\n",
         FOO,FUD,BAR,BAZ,BING,BONG);④
#define BAR -2 ②

#define BONG 7 ④
  printf("%d, %d, %d, %d, %d, %d\n",
         FOO,FUD,BAR,BAZ,BING,BONG);
  return 0;
}

#undef FUD
```

13

PCp³ analysis will notice

① Two definitions of **FOO** may not be mutually exclusive

② **BAR** is redefined, second definition location is inappropriate for a variable definition

③ **BING** is used as an array bounds specifier

④ **BONG** is used as a variable identifier before later being defined as a macro

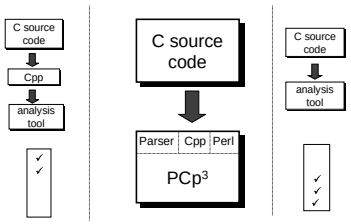
14

Transformed code

```
uni's em: def-example-pcp3-convert.c
#include <stdio.h>
#define CCD2
#ifdef CCD1
#  define FOO 1
#endif
#ifdef CCD2
#  define FOO 2
#endif

static const int FUD = 3;
#define BAR 4
static const int baz = 5;
enum {BING = 6};

int
main(char **argv, int argc)
{
  int fud = FUD;
  int BONG = -1;
  int rgi[BING];
  printf("%d, %d, %d, %d, %d, %d\n",FOO,FUD,BAR,baz,BING,BONG);
#define BAR -2
#define BONG 7
  printf("%d, %d, %d, %d, %d, %d\n",FOO,FUD,BAR,baz,BING,BONG);
  return 0;
}
-----Emacs: def-example-pcp3-convert.c      (C++ Qvwrt)--L22--All
uni's em: *Doc Property*
def: BAR defined [286:300]
xform: Parse stack may not be appropriate for a declaration: 0 2$
Expansions vary, not a constant at def-example.c
use: -2 at def-example.c:360,363
-**-Emacs: *Doc Property*      (Fundamental)--L4--All----------
```

15

How PCp³ compares



PCp³

- ✓ Precise analysis
- ✓ Reuses existing front-end
- ✓ Handles non-syntactic programs
- ✓ Processes entire source code
- ✓ Analyzes same view as engineer

16

Future work

- Complete hooks/backcalls interfaces
- Exploit AST better
  - avoid single-pass problems
  - permit more sophisticated analyses
- Generalized tables for better versioning support
- Multiple translation units
- Convert more preprocessor uses
- Convert to Java

17

Summary

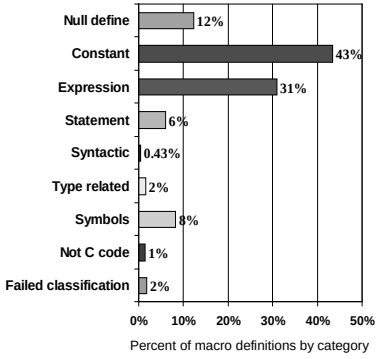
- C preprocessor limits program understanding and tools
- PCp^3 flexible analysis framework
  - integrates preprocessor and parser
  - hooks interface for simplifying analyses
- Interactive view of mapping between un- and pre-processed views
- Transforming C preprocessor use into language features

Extra slides.....

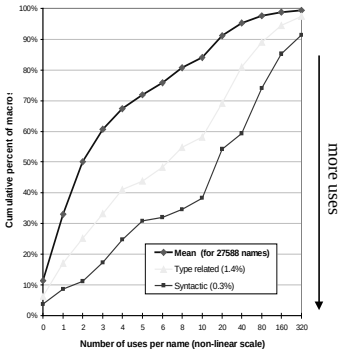
The C Preprocessor

- File inclusion
  - `#include`
- Conditional inclusion
  - `#if`, `#ifdef`, `#ifndef`
  - `#else`, `#elif`
  - `#endif`
- Textual (token-based) macros
  - User-defined via `#define`
  - Pre-defined
    - e.g., `__STDC__`, `__FILE__`
- Stringization, pasting operators
  - #, ##

Most definitions are simple



Difficult macros are used more than simple ones

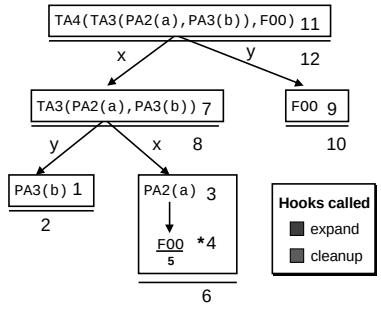


Syntactic macros hinder lexical tools

```
#define START_PARAMS (
#define END_PARAMS )
#define START_CALL (
#define END_CALL )
#define BEGIN {
#define END }
#define ARRAY []
#define NL ;
#define POINTER *
#define COMMA ,
#define STRING(x) #x

#include <stdio.h>

int main START_PARAMS int argc COMMA
        char POINTER argv ARRAY END_PARAMS
BEGIN
  printf START_CALL STRING(Hello world\n)
         END_CALL NL
  return 0 NL
END
```

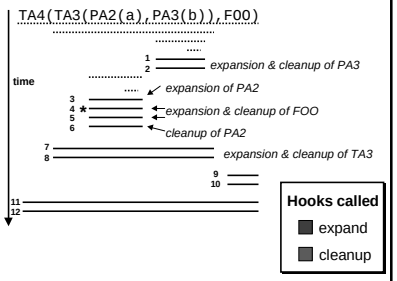
Expansion mechanism

```
#define TA3(x,y) y + x   note the order of arg. use
#define TA4(x,y) x + y
#define PA2(x) x + FOO
#define PA3(x) x + 4
#define FOO bar
```



Nested macro expansion

```
#define TA3(x,y) y + x   note the order of arg. use
#define TA4(x,y) x + y
#define PA2(x) x + FOO
#define PA3(x) x + 4
#define FOO bar
```



Intentional programming group's approach (MS)

1. Convert conditional inclusion directives into (C++) parse-able stylized declarations:

```
#if !defined(NDEBUG)
```

⇓ convert inclusion directives

```
int __cpp_if = !defined(NDEBUG);
```

2. Expand all macros and annotate the resulting tokens. Store those derivations in the AST
3. Encapsulate variant expansion as an implicit argument at all call sites.

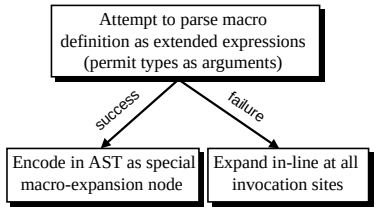
```
#define errmsg(x) \
  fprintf(stderr,x ", line %d", __LINE__)
```

⇓ add arguments to call sites

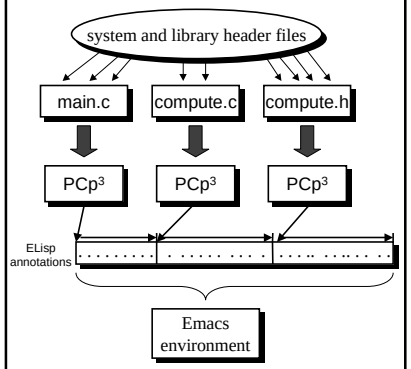
```
errmsg("Missing filename",563)
```

4. "Enzymes" then process the AST and try to recognize known abstractions (intentions)

Griswold & Atkinson's approach for TAWK



Multiple translation units

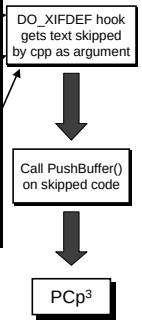


Details of processing entire source

```
use pcp3;
AddHook("DO_XIFDEF",\&do_xifdef);
AddHook("POP_PERL_BUFFER",\&pop_perl_buffer);
AddHook("HANDLE_DIRECTIVE",\&handle_directive);

sub do_xifdef {
  my ($s_start,$s_end,$kind,$conditional,$trailer,
      $skipped_text,$fSkipping,
      $s_branch_start) = @_;
  if ($fSkipping) {
    handle_unincluded_block($s_branch_start,
          $skipped,$kind,$conditional);
  }
}

sub handle_unincluded_block {
  my ($s_branch_start,$s_branch_end,$skipped) = @_;
  $skipped =~ s/^\s*\#\s*include .*$//mg;
  pcp3::YYPushStackState();
  pcp3::EnterScope();
  pcp3::PushHashTab();
  pcp3::PushBuffer($skipped,$s_branch_start);
}

sub pop_perl_buffer {
  pcp3::ExitScope();
  pcp3::PopHashTab();
  pcp3::YYPopAndRestoreStackState();
}

sub handle_directive {
  $cDirectives++;
}
```

Processing all the source code

```
#ifndef COMPUTE_H__
#define COMPUTE_H__
#include <stdio.h>

#ifdef PROFILE
#define PROFILE_OUTPUT(x) \
    fprintf(stderr,"%d\n",x);
#endif
...
#ifdef PROFILE
int ccalls = 0;
#endif
....
#ifdef PROFILE
ccalls++;
PROFILE_OUTPUT(ccalls);
#endif
...
#endif /* COMPUTE_H__ */
```



Other backcalls can save
- parser configuration
- symbol table
- preprocessor name table

30