Marco: Safe, Expressive Macros for Any Language*

Byeongcheol Lee[1], Robert Grimm[2],
Martin Hirzel[3], and Kathryn S. McKinley[4,5]

[1] Gwangju Institute of Science and Technology
[2] New York University
[3] IBM Watson Research Center
[4] Microsoft Research
[5] The University of Texas at Austin

Abstract. Macros improve expressiveness, concision, abstraction, and language interoperability without changing the programming language itself. They are indispensable for building increasingly prevalent multilingual applications. Unfortunately, existing macro systems are well-encapsulated but unsafe (e.g., the C preprocessor) or are safe but tightly-integrated with the language implementation (e.g., Scheme macros). This paper introduces Marco, the first macro system that seeks both encapsulation and safety. Marco is based on the observation that the macro system need not know all the syntactic and semantic rules of the target language but must only directly enforce some rules, such as variable name binding. Using this observation, Marco off-loads most rule checking to unmodified target-language compilers and interpreters and thus becomes language-scalable. We describe the Marco language, its language-independent safety analysis, and how it uses two example target-language analysis plug-ins, one for C++ and one for SQL. This approach opens the door to safe and expressive macros for any language.

1 Introduction

Macros enhance programming languages without changing them. Programmers use macros to add missing language features, to improve concision and abstraction, and to interoperate between different languages and with systems. With macros, programmers use concrete syntax instead of tediously futzing with abstract syntax trees or, worse, creating untyped and unchecked strings. For instance, Scheme relies heavily on macros to provide a fully featured language while keeping its core simple and elegant. Programmers use the C preprocessor to derive variations from the same code base (e.g., with conditional compilation) and abstract over the local execution environment (e.g., defining types and variables in system-wide header files). Web programmers use macros in PHP and similar languages to generate HTML code. Programmers also use macros

* This research was supported by the Samsung Foundation of Culture, and NSF grants CCF-1018271, CCF-1017849, and SHF-0910818.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 589–613, 2012.
© Springer-Verlag Berlin Heidelberg 2012

to generate strings containing SQL queries that interoperate with databases. As illustrated by they last two examples, macros do not only improve individual languages, but are also indispensable for building increasingly prevalent multilingual applications.

Programmers typically write macros in a *macro language* and the macro system generates code in a *target language*. Programmers embed macros in a *host language*. The host and target languages may differ. Macros are resolved before the target-language code is compiled or interpreted. Macro systems fall into two main categories. (1) Some macro systems are well encapsulated from the compiler or interpreter but are unsafe, e.g., the C preprocessor executes before the target-language compiler, but may generate erroneous code. (2) Some macro systems are safe but are tightly integrated with the target languages, e.g., the Scheme interpreter hygienically implements the core language together with its macro system [13,5].

Neither option is particularly attractive. Well encapsulated but unsafe macro systems lead to buggy target code. Notably, the C preprocessor operates on tokens, is not guaranteed to produce correct code, and C programs with macros consequently contain numerous errors [6]. Safe but tightly-integrated macro systems are limited to a prescribed combination of host and target languages and cannot be shared across languages. Developers must learn macro programming for every host and target language combination. Likewise, language designers need to design and implement macros for every combination. For example, while JSE [2] adapts Dylan's macros [20] to Java, it requires design changes and a fresh implementation. This burden increases the temptation to omit macros or use an encapsulated but unsafe macro system, such as GNU M4 [15]. Given the utility of macros and the diversity of current and future languages, expressive safe macros that scale across host and target languages are clearly desirable.

This paper introduces the Marco macro system, which delivers encapsulation and safety, making macros *language scalable*. Our key insight is that the macro system does not need to implement every target-language rule for target-language safety, but rather, it can reuse off-the-shelf target-language compilers or interpreters to do the job. Specifically, prior work on *syntactically* safe macros required the macro system to have a target-language grammar; this paper shows how to enforce syntax safety without that. Similarly, prior work on macros with safe *naming discipline* required the macro system to implement target-language scoping rules; this paper shows how to enforce naming discipline without reimplementing those either. Consequently, Marco composes a language-independent macro translator with unmodified target-language compilers and interpreters that check for most rule violations. The only requirement on the target-language processors is that they produce descriptive error messages that identify locations and causes of errors.

We designed Marco to meet three criteria: expressiveness, safety, and language scalability. For expressiveness, the Marco language has static types, conditionals, loops, and functions, making it Turing-complete. Marco supports target-language *fragments* as first-class values. As indicated by the name, fragments

need not be complete target-language programs. Rather, they contain portions of a target-language program along with *blanks* that other fragments fill in. For safety, Marco uses macro-language types to check target-language syntax, and uses dataflow analysis to check target-language naming discipline. For language scalability, Marco relies on error messages from target-language processors, making it the first safe macro system that is independent of the target language. We demonstrate Marco for two target languages. For depth, we chose C++, which is relatively complex due to its rich syntax, types, and many other features. For breadth, we chose SQL, which differs substantially from C-like languages and is of critical importance to many web applications. Marco currently checks for syntactic well-formedness and naming discipline; we leave type checking target-language code for future work. Furthermore, Marco programs are currently stand-alone; we leave host-language integration for future work, e.g., by using compositional techniques from Jeannie for C and Java [10].

In summary, this paper's contributions are: (1) The design of a safe and expressive macro language that is scalable over target languages. (2) A macro safety verification mechanism that uses unmodified target-language compilers and interpreters. (3) An open-source implementation[1] of the target-language independent macro processor and plug-ins for C++ and SQL.

2 Marco Overview

Fig. 1 illustrates the Marco architecture. It shows the Marco *static checker* taking a Marco program as input and verifying that target-language fragments are correct at macro definition time. Because Marco supports code generation based on external inputs, including additional target-language fragments, some syntax and/or name errors may survive static checking undetected. Consequently, the Marco *dynamic interpreter* verifies fragments again at macro instantiation time, after Marco fills in all blanks. This double checking is critical for developing macro libraries, where one group of programmers writes the macros and another group instantiates them with application-specific inputs. Both the static checker and dynamic interpreter rely on common *oracles* to verify target-language code syntax and naming discipline and detect any errors. The oracles abstract over the different target languages. They query a target-language

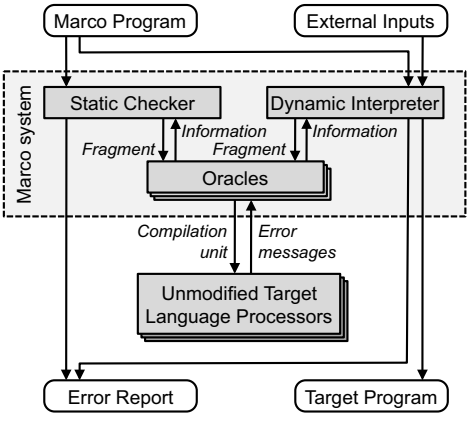

Fig. 1. The Marco architecture

[1] Available at `http://cs.nyu.edu/xtc/`

processor (currently, gcc for C++ macros and SQLite for SQL macros) by submitting specially crafted small compilation units.

Marco achieves language scalability by composing off-the-shelf target-language compilers and interpreters with a common translation engine. To add a new target language, developers implement a simple lexical analysis module that recognizes target-language identifiers and the end of target-language fragments. They also implement a plug-in with three oracles that (1) check for syntactic well-formedness, (2) determine a fragment's free names, and (3) test whether a fragment captures a given name. Everything else is target-language independent. In particular, Marco includes a reusable dataflow analysis, which propagates free and captured identifiers and reports accidental name capture.

3 The Marco Language

This section describes the Marco language, using examples, grammar rules, and type rules. The Marco language is a statically typed, procedural language. It supports macros using three constructs: code types, fragments, and blanks.

The example *synch* C++ macro in Fig. 2 ensures that lock acquire and release operations are properly paired (modulo exceptions), which C++ does not ensure. Lines 1–3 contain the signature of the Marco function *synch*, which takes two parameters, a C++ identifier and a C++ statement, and returns a C++ statement. The **code** type is parameterized by the target language and a nonterminal. Line 5 uses the back-tick operator (`) to begin a *fragment*, which is a quoted piece of target-language code. Line 6 uses the dollar operator (**$**) to begin a *blank*, which is an escaped piece of Marco code embedded in a fragment. The evaluation rule for a fragment first evaluates embedded blanks, then splices their results into the fragment's target-language code:

```
1 code<cpp,stmt>                    # code type
2 synch(code<cpp,id> mux,
3       code<cpp,stmt> body) {
4   return
5     `cpp(stmt)[{                  # C++ fragment
6       acquireLock($mux);
7       $body                       # blank
8       releaseLock($mux);
9     }];
10 }
```

Fig. 2. Marco code to generate C++

$$\frac{\forall i \in 1 \ldots n : \mathsf{Env} \vdash e_i \longrightarrow \beta_i}{\mathsf{Env} \vdash \text{`}lang(nonT)\,[\alpha_0 \$e_1 \alpha_1 \ldots \$e_n \alpha_n] \longrightarrow \text{`}lang(nonT)\,[\alpha_0 \beta_1 \alpha_1 \ldots \beta_n \alpha_n]} \text{ (E-FRAGMENT)}$$

Each α_i is a sequence of target-language tokens, each $\$e_i$ is a blank, and each β_i is the result of evaluating a blank to a sequence of target-language tokens. The result is the concatenation of all α_i and β_i.

Fig. 3 presents the Marco grammar. The interesting grammar rules are *fragment* and its helpers. A fragment, such as `` `cpp(stmt)[...$x...$y...] ``, consists of a head and a sequence of fragment elements. The head specifies the target language, a nonterminal, and an optional list of captured identifiers. We use this list when checking the target code's naming discipline (see Section 7).

```
program       ::= functionDef+
functionDef   ::= type ID '(' (formal (, formal)*)? ')' '{' stmt* '}'
formal        ::= type ID
stmt          ::= '{' stmt* '}'                                  # block
                | type ID '=' expr ';'                           # variable declaration
                | 'if' '(' expr ')' stmt ('else' stmt)?          # conditional
                | 'for' '(' ID 'in' expr ')' stmt                # loop
                | 'return' expr ';'                              # function return
                | expr ';'                                       # expression statement
expr          ::= fragment                                       # fragment
                | '(' expr ')'                                   # parentheses
                | ID                                             # variable use
                | expr INFIX_OP expr                             # infix operation
                | ID '(' (expr (, expr)*)? ')'                   # function call
                | expr '.' ID                                    # record attribute
                | expr '[' expr ']'                              # list subscript
                | '[' (expr (, expr)*)? ']'                      # list literal
                | '{' ID '=' expr (, ID '=' expr)* '}'           # record literal
                | 'true' | 'false' | INT | STRING                # primitive literal
type          ::= 'code' '<' language ',' nonTerm '>'            # fragment type
                | 'list' '<' type '>'                            # list type
                | 'record' '<' formal (, formal)* '>'            # record type
                | 'boolean' | 'int' | 'string'                   # primitive type
fragment      ::= fragmentHead '[' fragmentElem* ']'
fragmentHead  ::= '`' language '(' nonTerm (',' capture)? ')'
language      ::= ID
nonTerm       ::= ID
capture       ::= 'capture' '=' '[' ID (, ID)* ']'
fragmentElem  ::= TARGET_TOKEN | blank
blank         ::= '$' baseExpr
```

Fig. 3. Marco grammar

There are two kinds of fragment elements in Marco: target-language tokens and blanks. Since Marco identifies fragment elements with square brackets, the Marco parser must count matching square brackets in the fragment itself to find the end, e.g., in `` `cpp(expr)[arr[idx]] ``. It should, however, ignore square brackets that appear in target-language strings or comments, e.g., in `` `cpp(expr)[printf("[")] ``. To enforce the naming discipline, the Marco parser must find the fragment's identifiers. It should not treat a target language's keywords or numerical suffixes as identifiers, e.g., in `3.1e4` or `111u`. Since different languages have different keywords, literals, and comments, Marco must be configured with target-language specific lexers. To select lexers based on the syntactic target-language context identified by the fragment's head, we use the *Rats!* [8] scannerless and modular parser generator. The corresponding target-language lexers are very simple and recognize only the target-language tokens listed above.

The static type system includes the primitive types **`boolean`**, **`int`**, and **`string`**; **`list`** parameterized by element type; **`record`** parameterized by attribute names and types; and **`code`** parameterized by target language and nonterminal.

```
1 code<sql,query>
2 genTitleQueryInSQL(code<sql,expr> pred){
3   return `sql(query)[select title
4       from moz_bookmarks where $pred
5     ];
6 }
```

Fig. 4. Marco code to generate SQL

The latter is key to *target language scalability*. For example, Fig. 4 shows a Marco macro generating SQL. Compared to Fig. 2, the Marco syntax and semantics remain the same, but the target language and therefore the code type parameters differ. The Marco engine uses code types to invoke the appropriate target-language oracles, which determine syntactic well-formedness as well as free and captured identifiers. It tracks code types to check syntax when filling in blanks in fragments. And it tracks names during its dataflow analysis, which ensures that identifiers from multiple macros generate consistent bindings. In other words, the strongly typed quote and unquote mechanism lets us maximize Marco's target-language independent functionality while delegating checking to the target-language specific oracles.

The type rule for a Marco fragment first checks the types for each of the embedded blanks, which must result in code belonging to the same target language (*lang*). It then uses the language *lang*, the nonterminal *nonT* of the fragment, the nonterminals $nonT_i$ of each of the blanks, and the contents of the fragment as inputs to a syntax oracle. As far as the Marco type system is concerned, the syntax oracle is a black-box that either succeeds or fails. If the oracle succeeds, the type of the fragment is **code**<*lang*, *nonT*>.

$$\frac{\forall i \in 1 \ldots n : \Gamma \vdash e_i : \textbf{code}\langle lang, nonT_i\rangle \quad syntaxOracle(lang)(nonT, [nonT_1, \ldots, nonT_n], \alpha_0 \$1 \alpha_1 \ldots \$n \alpha_n)}{\Gamma \vdash \text{`}lang(nonT)\,[\,\alpha_0 \$e_1 \alpha_1 \ldots \$e_n \alpha_n\,] : \textbf{code}\langle lang, nonT\rangle} \quad \text{(T-Fragment)}$$

4 The Marco Analysis Framework

At its core, the Marco system provides a static checker and a dynamic interpreter. The static checker verifies correctness at macro development time. The dynamic interpreter generates target-language code and verifies correctness at macro instantiation time. The two components share target-specific oracles, which check syntactic well-formedness and naming discipline in target-language fragments.

Each *oracle* mediates between the target-language independent Marco engine and an off-the-shelf target-language *processor*, i.e., compiler or interpreter. The oracle converts fragments into compilation units, passes the compilation units to the target-language processor, and then parses any error messages in the output. The only target-language specific parts newly developed for Marco are the target-specific lexers (see Section 3) and oracles, which reuse existing target-language processors. A key advantage of Marco over other safe macro systems is that it does not require new or even modified target-language processors.

In more detail, each target-language plug-in provides three oracles: *syntax*, *free-names*, and *captured-name*. Marco itself is implemented in Java, and each

target-language plug-in implements the same three Java interfaces. Since Java already integrates database access through JDBC, the SQL plug-in uses this API. In contrast, the C++ plug-in interacts with gcc through the file system. Either way, all target-language interactions share two characteristics. First, target-language processors receive programs as sequences of characters: strings for JDBC and files for gcc. In other words, we lower the tokenized fragments into character strings. Second, the processor outputs are strings that indicate syntactic or semantic errors, which are then parsed by the plug-in. At the same time, the concrete error reporting mechanism depends on the target language, e.g., Java exceptions for JDBC and standard error output for gcc.

To check target-language syntax, the system first parses a Marco program and tokenizes the target-language fragments. It then ensures that the target-language fragments are consistent with their declared **code** type parameters. For example, consider `` `cpp(expr)[x = 1;] ``. The Marco type checker applies rule T-Fragment from Section 3, which triggers a call to the C++ syntax oracle: *syntaxOracle*(`cpp`)(`expr`, [], '`x = 1;`'). The C++ syntax oracle then generates the following compilation unit for the unmodified C++ compiler, i.e., gcc: `int check_expr(){return (x = 1;);}` For this input, gcc reports an error complaining about the spurious semicolon after `x = 1`. Based on this error message, the oracle deduces that the fragment was syntactically ill-formed for nonterminal `expr`. Since the oracle fails, Marco type-checking fails, and the system reports an error. This example ignores idiosyncrasies of blanks and C++, which Sections 5 and 6 explore.

For naming disciplines, consider a fragment f_1 that fills a blank in fragment f_2. Fragment f_1 is `` `sql(expr)[birthYear >= 1991] ``, and fragment f_2 is `` `sql(query)[select name from Patrons where $pred] ``. Using the SQL free-names oracle, Marco discovers that f_1 contains the free identifier `birthYear`. It uses dataflow analysis to discover that f_1 flows into the blank in f_2. Finally, Marco uses the SQL captured-name oracle to check if identifier `birthYear` is captured by blank `$pred`. Programmers use annotations to tell Marco when a capture is intentional; otherwise, Marco reports an accidental-capture error. Subsequent sections describe how the oracles turn errors from target-language processors into information for Marco's static checker and dynamic interpreter.

5 Checking Syntactic Well-Formedness

This section describes how Marco checks target-language syntax. The *syntax oracle* is the interface between the target-language agnostic Marco system and the black-box target-language processors. The signature of the syntax oracle, as embodied in type rule T-Fragment from Section 3, is:

$$\mathit{syntaxOracle} : \mathit{lang} \to (\mathit{nonT}, \mathbf{list}\langle \mathit{nonT} \rangle, \alpha_0 \$1 \alpha_1 \ldots \$n \alpha_n) \to \mathbf{list}\langle \mathit{error} \rangle$$

For example, consider the following invocation of the syntax oracle:

$$\mathit{syntaxOracle}\ (\ \texttt{sql}\ )\ (\ \texttt{query}, [\mathit{expr}], \text{`}\mathbf{select}\ a\ \mathbf{from}\ B\ \mathbf{where}\ \$1\text{'}\ )$$

Table 1. Helper fragments for syntax oracles. Place-holder fragments fill in blanks. Completion fragments turn a fragment into a self-contained compilation unit. Marco fills in **$***fresh* blanks with fresh identifiers, and **$***orig* blanks with the original fragment.

Marco type	Place-holder fragment	Completion fragment
code<*sql,expr*>	`0`	**select** $*fresh1* **from** $*fresh2* **where** (**$***orig*)
code<*sql,query*>	**select** *	**$***orig*
code<*sql,qlist*>	*/*empty*/*	**$***orig*
code<*cpp,expr*>	`0`	**int** $*fresh*() { **return** (**$***orig*); }
code<*cpp,stmt*>	`;`	**void** $*fresh*() { **if**(1) **$***orig* **else**; }
code<*cpp,id*>	**$***fresh*	**int** $*fresh*() { **return** (**$***orig*); }
code<*cpp,type_sp*>	**int**	**$***orig* $*fresh*;
code<*cpp,type_id*>	**int**	**int** $*fresh*() { **return** *sizeof*(**$***orig*); }
code<*cpp,fdef*>	**void** $*fresh*(){}	**$***orig*
code<*cpp,mdecl*>	**int** $*fresh*;	**class** $*fresh* { **$***orig* };
code<*cpp,decl*>	**int** $*fresh*;	**$***orig*
code<*cpp,cunit*>	*/*empty*/*	**$***orig*

In this example, the target language is SQL, the nonterminal of the fragment is *query*, and there is one blank, whose nonterminal is *expr*. The fragment contents have the form α_0**$**1$\alpha_1$, where α_0 is the token sequence before the blank, **$**1 marks the location of the blank, and α_1 is the token sequence after the blank. In the example, α_0 is '**select** *a* **from** *B* **where**' and α_1 is empty. The remainder of this section describes the syntax oracle algorithm for producing compilation units, interpreting the results, and iterating when necessary.

5.1 Syntax Oracle Algorithm

The syntax oracle algorithm has four steps. The key challenge is to fill in each *blank* in the target-language fragment.

Step 1: Fill in blanks. The syntax oracle starts by filling in each blank with a place-holder fragment. The middle column of Table 1 shows the place-holder fragments for each code type in Marco's SQL and C++ plug-ins. Each such place-holder fragment is syntactically valid for a given nonterminal. In the example above, the nonterminal for blank 1 is *expr*, so the syntax oracle fills in blank 1 with the place-holder fragment for SQL expressions, which is `0`. The result is the fragment **select** *a* **from** *B* **where** `0`. Intuitively, filling in blanks with fixed fragments works because target languages have (more or less) context-free grammars, and the syntax oracle can check syntactic validity even when there are semantic errors. In the example, the place-holder fragment is of type integer and the blank expects type boolean, but this semantic mismatch is irrelevant to syntactic well-formedness.

Step 2: Complete the fragment. The syntax oracle completes fragments to obtain self-contained compilation units. In the example, the fragment is already a full query. The right column of Table 1 shows the completion fragments for each of the code types in Marco's SQL and C++ plug-ins. In addition to turning a fragment into a compilation unit, Step 2 generates boiler-plate syntax. For SQL,

it adds code to begin and then abort a transaction, which prevents side-effects from sending the SQL query to a live database during analysis.

Step 3: Run the target-language processor. The syntax oracle next sends the completed fragment to the target-language processor and collects any error messages. For SQL, Marco makes a JDBC call and catches any exceptions. For C++, Marco generates a file with the fragment, compiles it with gcc, and reads any error messages from `stderr`.

Step 4: Determine oracle results. Finally, the syntax oracle translates errors from the target-language processor into oracle results. It must distinguish syntax errors from other errors, as a fragment only fails the syntactic well-formedness test if there are syntax errors. In C++, other errors may mask syntax errors, so the oracle may iterate to determine if the fragment also has a syntax error, as explained in Section 6. If the syntax oracle fails, the oracle maps error message line-numbers back to the original Marco code, and reports the error.

5.2 Syntax Oracle Example

Consider the example fragment `` `sql(expr)[type = ] ``, which is missing its right operand. Type rule T-FRAGMENT invokes the syntax oracle as follows: *syntaxOracle*(`sql`)(`expr`, [], '`type =`'). The oracle goes through its four steps:

1. Fill in blanks. This step is a no-op, since there are no blanks.
2. Complete the fragment. The oracle consults Table 1 to find the completion fragment for **code**<`sql,expr`>, yielding **select** `x` **from** `T` **where** (`type =`).
3. Run the target-language processor. The oracle uses JDBC to send the completed fragment to SQLite, and then catches the resulting SQLException, which contains the error message "Syntax error near '='.".
4. Determine result. Since the error from the target-language processor was a syntax error, the oracle reports this error back to the user.

Assume the user fixes the fragment, writing `` `sql(expr)[type = 1] ``, and then runs Marco again.

1. Fill in blanks. This step is still a no-op, since there are no blanks.
2. Complete the fragment yields **select** `x` **from** `T` **where** (`type = 1`).
3. Run the target-language processor. If there is no table `T` with an attribute `type` in the database, the error message is "No attribute '`type`' in table '`T`'.".
4. Determine result. Since the error is not a syntax error, the oracle succeeds and indicates that the fragment is syntactically well-formed.

5.3 Discussion

Good completion fragments (see Table 1) satisfy three properties: they are complete, conservative, and accurate. A *complete* fragment yields a complete compilation unit in the target language. A *conservative* fragment has a blank that accepts all fragments conforming to the original nonterminal. An *accurate* fragment rejects fragments that do not conform to the original nonterminal. Of these three properties, the first two help avoid spurious error messages, i.e., false positives, while the third helps avoid missed syntax errors, i.e., false negatives.

Consider the completion fragment for a C++ expression. One complete and conservative solution would be `int $fresh(){ return $orig; }`. This completion fragment is not accurate, since it accepts `0;`, which is a C++ statement and not a C++ expression. To increase accuracy, Marco adds parentheses around the blank, as in `int $fresh(){ return ($orig); }`. As another example, consider the completion fragment for a C++ statement. One complete and conservative solution is `void $fresh(){ $orig }`. However, this completion is not accurate, since it accepts `x=1;y=2;`, a sequence of two statements instead of a single C++ statement. Marco resolves this problem by inserting a conditional statement around the blank: `void $fresh(){ if(1) $orig else; }`. In our experience, enclosing fragments and blanks in delimiters or embedding them in other target-language constructs makes completion fragments more accurate.

6 Context-Sensitive Syntax

Most programming languages, including SQL, Java, ML, and Scheme, have context-free syntax. In this case, the syntax oracle from Section 5 works directly as described. It checks the syntactic well-formedness of a fragment in isolation based on the declared language and nonterminal. However, our goal is to handle any language, including languages with context-sensitive syntax such as C++. Prior work on safe macro systems does not address this issue. This section extends our syntax oracle to correctly deal with context sensitivity.

6.1 Context-Sensitive Syntax Examples

As a first example, consider the following C++ fragment:

```
`cpp(mdecl)[void* method(typeless o) { return 0; }]
```

The nonterminal *mdecl* stands for a member declaration. The fragment is syntactically well-formed for this nonterminal, since a method is a special case of a member. However, after using the completion fragment for *mdecl* from Table 1, gcc reports the following errors:

```
error: expected ';' at end of member declaration
error: expected ')' before 'o'
```

These are syntax errors, even though the root cause is a semantic problem: identifier *typeless* has not been declared as a type. When gcc cannot parse *typeless* as a declaration specifier, it speculates that *method* is a variable name. But the downstream tokens make no sense for a variable declaration. This case shows how a semantic problem, the missing declaration context for *typeless*, *induces* a syntax error.

To resolve such cases, our C++ syntax oracle enumerates all identifiers in the input fragment, and speculates one by one that they are type names (i.e., the opposite of gcc's speculation). In the example, the syntax oracle finds three identifiers: *method*, *typeless*, and *o*. At first, the syntax oracle speculates that

method is a type, but declaring it as such does not advance the location of the first error message. Then, the oracle speculates that *typeless* is a type and issues the following query to gcc:

```
class typeless { };                          // speculative context
class id1 {                                  // completion fragment
  void* method(typeless o) { return 0; } // input fragment
};
```

Since gcc reports no syntax errors for this query, the syntax oracle correctly concludes that the input fragment is syntactically well-formed and succeeds. In theory, a fragment with n ambiguous identifiers may require up to 2^n speculative queries for determining syntactic well-formedness. However, in practice, our simple heuristic of always advancing the location of the first reported error avoids this exponential explosion.

As a second example, consider another C++ fragment (based on [19]):

```
`cpp(stmt)[ A(*x)[4] = y; ]
```

The C++ compiler can parse this in two ways: either as a variable declaration or as an expression statement. If `A` is a type, then the code declares variable `x` as a pointer to an array of 4 elements of type `A`, and initializes it to `y`. On the other hand, if `A` is a function name, then the code calls the function with parameter `*x`, accesses element `[4]` of the result, and assigns it `y`. The ambiguity between declarations and statements is so prevalent in C++ that the language specification has a disambiguation rule of preferring declarations over expression statements [23, p. 802]. Though, C++, unlike C, does treat declarations as statements for its syntax.

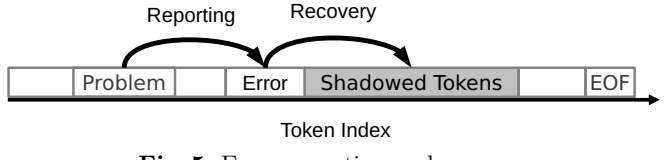

Fig. 5. Error reporting and recovery

If part of a program is not well-formed, language processors report the error and try to recover, so that they can report more than one error per invocation. Fig. 5 depicts how language processors scan through lexical tokens, detect a syntactic or semantic problem, generate an error message, skip several tokens, and continue analysis. For instance, ANTLR-generated parsers report syntax errors and then recover by either inserting a token or skipping several downstream tokens. The hand-written parsers in gcc report both syntax and semantic errors, and may skip all or some downstream tokens.

If the skipped tokens contain a syntax error, then the error recovery for the first error *shadows* the syntax error. Therefore, the absence of syntax errors does not imply syntactic well-formedness. The example fragment `A(*x)[4] = y;` triggers the following error messages:

```
error: 'x' was not declared in this scope
error: 'A' was not declared in this scope
error: 'y' was not declared in this scope
```

These semantic errors may shadow downstream syntax errors, so our oracle speculates that an identifier in a semantic error may be either a type or variable name. In the example, making `A` a type name and `x` and `y` variable names eliminates the missing declaration errors, as shown in the following oracle query:

```
class A {}; class {} x; class {} y;  // speculative context
void query() {                       // completion fragment
  A(*x)[4] = y;                      // input fragment
}
```

This fragment still yields a semantic error ("cannot convert '<anonymous class>' to '`A (*)[4]`'"), but the error cannot shadow syntax errors. Hence, the oracle concludes that the fragment is syntactically well-formed.

6.2 Error Classification

The syntax oracle is concerned with syntax errors, and, in an ideal world, it would not have to deal with semantic errors. However, as demonstrated by the above examples, semantic problems affect syntax errors in two cases: a semantic problem can *induce* a syntax error, or a semantic problem can *shadow* a syntax error. In the induced-error case, it appears as if the fragment is syntactically ill-formed, but it is actually well-formed. In the shadowed-error case, it appears as if the fragment is syntactically well-formed, but it is actually ill-formed.

We need not handle syntax errors that induce or shadow another error; the first error suffices to conclude that the fragment is syntactically ill-formed. Neither do we need to handle the case of a semantic problem inducing or shadowing a semantic error; as long as that error in turn does not induce or shadow a syntax error, it does not affect the syntax oracle. Consequently, the syntax oracle must recognize two classes of errors: (1) syntax errors and (2) semantic errors that may shadow syntax errors. We systematically investigated all C++ errors generated by gcc to validate that our syntax oracle handles these cases correctly. Our investigation found that there are two kinds of syntax errors: parsing errors, which are generated while parsing, and post-parsing errors, which are generated after parsing but still capture violations of syntactic constraints such as **`case`** labels always appearing inside **`switch`** statements. Parsing errors are easy to recognize (they all begin with the word "expected" and include a token or nonterminal symbol), and there are only a few post-parsing errors.

To collect all shadowing error messages, we identified the seven error recovery routines in gcc that update the parser state to skip tokens until a synchronization token. For example, one such routine is `skip_until_sync_token()`. Next, we enumerated all call-sites for the recovery routines. We found that code leading up to these call-sites commonly looks as follows:

```
bool ok = perform_semantic_check();
if (ok)
  error("A");
else
  error("B");
if (!ok) {
  error("C");
  skip_until_sync_token();
}
```

If *ok* is false, the compiler invokes *skip_until_sync_token*() and thus skips tokens, which may contain syntax errors. Consequently, errors "B" and "C" may shadow syntax errors but error "A" may not. In most cases, we only had to look at a single routine to understand error shadowing, though in a few cases multiple routines were involved. We found that shadowing errors are most commonly lookup errors, which indicate that an identifier has not been declared; though a few non-lookup errors shadow other errors.

6.3 Iterative Syntax Oracles in Marco

In summary, context sensitivity prevents a Marco oracle from concluding syntactic well-formedness based solely on the absence of syntax errors. In particular for C++, a semantic problem can either induce or shadow a syntax error. Therefore, Marco's syntax oracle for C++ speculatively resolves syntax errors and shadowing semantic errors by issuing repeated queries to gcc, each with a different speculative context. In other words, the oracle speculates declarations for identifiers as variables or types. If Step 4 from Section 5.1 detects semantic errors, the algorithm iterates back to Step 2 and resolves them by enumerating the possible declarations for identifiers.

7 Checking Naming Discipline

This section describes how Marco uses dataflow analysis to ensure that macro expansion does not cause accidental name capture in the target language. Some macro systems, notably Scheme, prevent capture by automatically renaming variables, but that requires deep target-language specific knowledge and is therefore not an option for Marco, which is target-language agnostic. Accidental name capture is a typical bug when using the C preprocessor, as illustrated in Fig. 6.

```
1 #define swap(v,w) {int temp=v; v=w; w=temp;}
2 int temp = thermometer();
3 if (temp<lo_temp) swap(temp, lo_temp)
```

Fig. 6. Example for accidental name capture bug with C preprocessor [5,6]

Line 1 declares a macro *swap*, which uses a local variable *temp* standing for "temporary." Line 2 declares a different variable *temp* standing for "temperature" that is outside the scope of the macro. Line 3 passes the name *temp* as an

actual parameter to the formal `v` of `swap`. This use of `swap` captures the name `temp`. Since the code outside the macro uses `temp` for "temperature" and not "temporary," the name capture is called *accidental*.

More generally, accidental name capture happens when a first fragment f_1 contains a free name x; a second fragment f_2 unintentionally captures name x at blank b; and f_1 flows into b. Marco detects capture as follows. The *freeNames-Oracle* discovers all free names in fragment f_1. Marco's forward dataflow analysis propagates free names to blanks. The *capturedNameOracle* discovers whether f_2 captures name x at blank b. In dataflow terminology, the analysis state is a map from meta-language variables to free target-language names. The GEN-set of a fragment is the set of free names in the fragment. The KILL-set of a fragment at a blank is the set of names that it captures at that blank.

Marco uses static dataflow analysis at macro definition time and dynamic dataflow analysis in the interpreter at macro instantiation time. The oracles are target-language specific and use the off-the-shelf target-language processor as a black-box to generate error messages that reveal information about free and captured names. The dataflow analysis itself is target-language independent.

7.1 Free-Names Oracle

The signature of the free-names oracle is:

$$\mathit{freeNamesOracle} : \mathit{lang} \rightarrow (\mathit{nonT}, \mathbf{list}\langle \mathit{nonT} \rangle, \alpha_0 \$1 \alpha_1 \ldots \$n \alpha_n) \rightarrow \mathbf{list}\langle \mathit{ID} \rangle$$

For example, given fragment `` `cpp(expr)[100 * (1.0 / (foo))] ``, Marco invokes the free-names oracle as follows:

$$\mathit{freeNamesOracle}(\texttt{cpp})(\texttt{expr}, [\,], \text{`}\texttt{100 * (1.0 / (foo))}\text{'})$$

A target-language name is free if it is not bound inside the fragment. In the example, `foo` is free, and the oracle should return the list [`foo`]. To obtain this result, the free-names oracle first consults Table 1 to instantiate a completion fragment that will be sent to gcc:

```
int query_expr() { return (100 * (1.0 / (foo))); }
```

For this query, gcc returns the error message "name '`foo`' was not declared in this scope." The free-names oracle looks exactly for this kind of error message, which indicates that a name is undefined. In the example, the oracle speculates that `foo` is free. To validate this hypothesis, it executes one more query. It prepends a declaration of the name `foo` to the compilation unit and sends it to gcc again. In the example, the test becomes:

```
int foo;
int query_expr() { return (100 * (1.0 / (foo))); }
```

This modification resolves the declaration error and confirms the hypothesis as correct. Hence, the oracle adds the name `foo` to the list of free names. It repeats this process until it does not observe any more declaration errors. In summary, Marco exploits the fact that a name in a fragment is free, as long as it can be bound by a declaration in the enclosing scope.

7.2 Captured-Name Oracle

The captured-name oracle checks if a target-language name is captured by a blank in a fragment and thus if it is safe to fill in the blank with a fragment in which the name is free. The signature of the captured-name oracle is:

$$\begin{aligned} \mathit{capturedNameOracle} : \ & \mathit{lang} \\ & \rightarrow (\, \mathit{nonT}, \mathbf{list}\langle \mathit{nonT} \rangle, \\ & \quad \alpha_0 \mathbf{\$}1 \alpha_1 \ldots \mathbf{\$}n \alpha_n, \mathbf{int}, \mathit{ID}) \\ & \rightarrow \mathbf{boolean} \end{aligned}$$

We check the blank number **int** and the free name *ID* for capture. Consider swapping two integers: `` `cpp(stmt)[{int temp=$v; $v=$w; $w=temp;}] ``. The following oracle call checks whether blank 1 captures name *temp*:

```
capturedNameOracle ( cpp )
                   ( stmt, [expr, expr, expr, expr],
                     `{int temp=$1; $2=$3; $4=temp;}', 1, temp )
```

Since blank 1 in the fragment does in fact capture the target-language name *temp*, the oracle returns **true** as expected. SQL's scoping rules differ from C++ and implement semantics similar to a **with**-statement. For example, consider the fragment `` `sql(query)[select name from Patrons where $pred] ``. The blank in this fragment captures any names referring to column names in the *Patrons* table in the database. Our *capturedNameOracle* algorithm handles both target languages and their scoping rules with the same approach.

Consider invoking the *capturedNameOracle* to check if free name x is captured at blank number i. Similar to the other oracles, the captured-name oracle fills in all blanks j with $i \neq j$ using the nonterminal-specific place-holders from Table 1. However, for blank i, our analysis hypothesizes that x is captured at the blank. To find counter-evidence, it places x in the blank, wrapping it in boiler-plate code as necessary for syntactic well-formedness. If the target-language processor reports an "x is unknown" error message, then the oracle concludes that x is not captured at blank i, and returns **false**. Otherwise, it returns **true**.

7.3 Intentional Capture

The dataflow analysis propagates free target-language names through variables referencing target-language fragments to blanks. It reports an error if a blank accidentally captures a free name. To determine whether a capture is accidental or intentional, the analysis uses the optional **capture** annotation in the *fragmentHead* clause of the Marco grammar (see Fig. 3). If a name is listed in the **capture** annotation, the capture is intentional, otherwise the capture is accidental and Marco reports an error.

For an example of intentional capture, consider the Marco function *boundIf* in Fig. 7. The function implements an if-statement that binds the value of the condition to *it*, with the express purpose of exposing it to the body, i.e., blank 2.

```
1 code<cpp,stmt> boundIf(code<cpp,expr> cond, code<cpp,stmt> body) {
2   return `cpp(stmt, capture=[it])[{
3       int it = $cond;          #blank 1
4       if (it) { $body }        #blank 2
5     }];
6 }
```

Fig. 7. Example for intentional name capture when using Marco to generate C++ code

The annotation **capture**=[*it*] in line 2 indicates that any such capture is intentional. This convention makes it possible to fill blank 2 with a fragment, such as *printf*("%d", *it*);, that contains a free identifier *it* and have the body's *it* refer to the macro's *it*, as declared on line 3. Marco's captured-names oracle still detects the capture of the identifier *it*, but the **capture** annotation suppresses the corresponding error message. As a special case, if a blank captures an identifier provided by a previous blank in a binding position, e.g., **int** $*index*;, Marco assumes that the capture is intentional. No annotation is necessary.

7.4 Dataflow Analysis

The interesting statements for the dataflow analysis are Marco statements with fragments and blanks. Fig. 8 shows the transfer-function for such statements. Given a Marco statement and an input analysis state *inState*, the transfer function computes an output analysis state *outState*. The analysis state only changes for the Marco location w assigned by the statement. The captured-name oracle from Section 7.2 checks whether free names from $inState(v_1)$ through $inState(v_n)$ are captured by the fragment. If they are captured and the capture is not intentional (the set difference *Captured* − *Intentional* is non-empty), the analysis

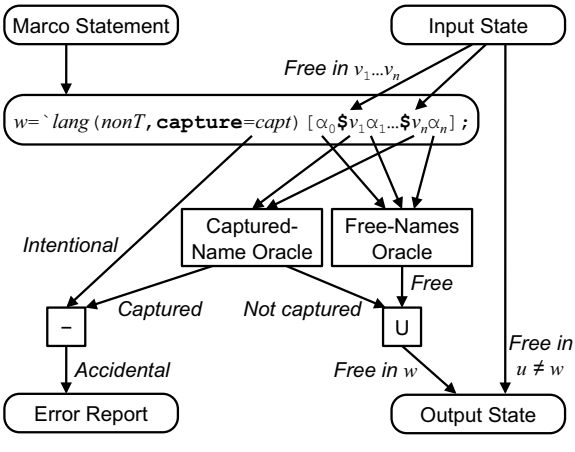

Fig. 8. Transfer functions for the naming-discipline analysis

reports an error. If they are not captured, then they are still free in w. In addition, the free-names oracle checks for free names in the constant portions α_0 thru α_n of the fragment. Those free names are also free in w. The resulting output state $outState(w)$ uses the free names for w as discovered by the oracles. For all other locations $u \neq w$, the transfer function forwards the free names from the input state $outState(u) = inState(u)$.

One pragmatic issue is how to report high-quality error messages in the case of accidental name captures. The analysis remembers which errors it has reported so far and avoids duplicates. Furthermore, the analysis tracks the originating fragment for each free name to more accurately report the source of accidental name captures. When the analysis detects an accidental capture, it reports both the line number of the origin and the line number of the capture.

Marco's static checker uses static dataflow analysis to enforce the naming discipline at macro definition time. It reports accidental name capture errors to macro authors. However, a Marco program may receive fragments as external inputs, and these fragments may contain free names. Consequently, Marco's interpreter uses dynamic dataflow analysis to enforce naming discipline again at macro instantiation time, now reporting accidental name capture errors to macro users. The dynamic dataflow analysis uses the same transfer function as the static analysis and it performs the same oracle queries.

8 Experimental Evaluation

This section experimentally validates the key characteristics of Marco: expressiveness, safety, and language scalability. To evaluate expressiveness, we implemented microbenchmarks from prior work and a code-generation template for a high-performance stream processing module. To evaluate safety, we execute Marco on each microbenchmark and on the stream processing code generator. To evaluate language scalability, we report statistics on the implementation effort for supporting different target languages.

8.1 Methodology

Tools and Environments. We use Marco r278 running on the Sun HotSpot Client JVM 1.6.0_21-ea. For the unmodified target-language processors, we downloaded and built gcc 4.6.1 as well as SQLiteJDBC v056 based on SQLite 3.4.14.2. We conducted all experiments on a Core 2 Duo 1.40 GHz with 4 GB main memory. The machine runs Ubuntu 11.10 on the Linux 3.0.0-12 kernel.

Marco Programs. We wrote 8 Marco microbenchmark programs with 22 macro functions derived from related work [5,26] and the `Aggregate` code generator derived from IBM InfoSphere Streams [16]. The `Aggregate` code generator produces C++ declarations, statements, and expressions that exercise classes, namespaces, and templates. Table 2 presents the microbenchmarks. The first four programs implement C++ macros from the MS^2 paper by Weise and Crew [26]. These macros add new abstractions such as resource management (`paint`), dynamic binding (`dynamic_bind`), rich exception handling (`exceptions`), and

Table 2. Oracle analysis results for the micro-benchmarks fragments

Marco Program	Fragment	Code Type	Size	Backtr.	Queries	Decls
paint	*Painting1*	**code**<*cpp*,*stmt*>	17	5	17	7
dynamic_bind	*dynamic_bind1*	**code**<*cpp*,*stmt*>	13	3	14	8
exceptions	*throw1*	**code**<*cpp*,*stmt*>	23	2	12	7
	throw2	**code**<*cpp*,*stmt*>	28	2	16	7
	catch1	**code**<*cpp*,*expr*>	1	1	3	3
	catch2	**code**<*cpp*,*stmt*>	51	1	8	4
	unwind_protect1	**code**<*cpp*,*expr*>	1	1	3	3
	unwind_protect2	**code**<*cpp*,*stmt*>	44	2	12	6
myenum	*myenum1*	**code**<*cpp*,*decl*>	5	0	1	1
	myenum2	**code**<*cpp*,*stmt*>	9	1	5	4
	myenum3	**code**<*cpp*,*decl*>	15	0	1	1
	myenum4	**code**<*cpp*,*stmt*>	14	2	8	6
	myenum5	**code**<*cpp*,*decl*>	18	0	2	2
discriminant	*discriminant1*	**code**<*cpp*,*expr*>	9	0	1	1
complain	*complain1*	**code**<*cpp*,*stmt*>	4	0	2	2
	main1	**code**<*cpp*,*expr*>	1	1	3	3
	main2	**code**<*cpp*,*stmt*>	13	0	2	2
swap	*swap1*	**code**<*cpp*,*id*>	1	1	3	3
	swap2	**code**<*cpp*,*id*>	1	1	3	3
	swap3	**code**<*cpp*,*stmt*>	28	5	31	12
SQLSyntax	*good1*	**code**<*sql*,*expr*>	3	0	1	0
	good2	**code**<*sql*,*stmt*>	6	0	1	0

multiple declarations (*myenum*). The next three programs implement C++ versions of the examples from "Macros That Work" by Clinger and Rees [5]. These macros illustrate naming issues during macro expansions. The final program generates SQL queries for extracting bookmark titles from a web browser's database.

Data collection methodology. To collect statistical results for fragment analysis, we turned on Marco's -pstat command-line option. To count source lines, we ran the sloccount utility. For the number of error handling rules, we manually examined source files in the Marco system.

8.2 Expressiveness and Safety

In Table 2, Column "Fragment" names the macros using logical function names. Column "Code Type" shows the types of the macros, which indicate the target language and nonterminal. Column "Size" counts the number of target-language tokens and blanks. The remaining columns present the results from running the oracle analysis. The oracle analysis synthesizes query programs to determine whether a fragment is syntactically correct. Column "Backtr." counts how often the syntax oracle needed to backtrack before finishing. Column "Queries" counts the number of compilation units sent to the target-language processor. Column

Table 3. Oracle analysis averages for the fragments in the *Aggregate* operator

Code type	Count	Size	Backtracks	Queries	Decls
code<*cpp*, *id*>	5	1.00	0.80	3.00	3.00
code<*cpp*, *type_spec*>	8	6.88	0.00	3.75	2.75
code<*cpp*, *type_id*>	1	1.00	0.00	2.00	2.00
code<*cpp*, *expr*>	12	4.50	0.08	2.67	2.58
code<*cpp*, *stmt*>	40	13.20	1.58	10.13	6.63
code<*cpp*, *fdef*>	11	31.00	4.09	21.73	9.36
code<*cpp*, *mdecl*>	22	12.36	0.05	3.23	3.00
code<*cpp*, *decl*>	13	11.38	0.00	4.08	3.00
code<*cpp*, *cunit*>	3	7.00	0.00	2.00	2.00

"Decls" shows the number of declarations synthesized by the oracle to provide evidence for syntactic well-formedness.

For the microbenchmark fragments, which contain 1–51 tokens or blanks, our oracle analyzer concludes syntactic well-formedness after evaluating 1–31 queries. The number of queries is proportional to the number of synthesized declarations rather than the size of input fragments. This result is not surprising, because the number of C++ parsing errors for syntactically well-formed fragments should be proportional to the number of undefined identifiers. About 20% of queries result in the oracle backtracking speculations.

This research was originally motivated by language interoperability and concision for IBM's InfoSphere Streams, a stream processing system [16]. A streaming application consists of data streams and operators. Each operator continuously consumes data from one or more input streams, performs its computation, and outputs one or more streams. An *Aggregate* operator uses sum, average, maximum, etc. over a sliding window and must be customized for the particular aggregate and data types. To implement these operator variants, InfoSphere Streams uses "code generation templates," i.e., macros that generate custom code for a specific operator variant. We re-implemented the code generation template for the *Aggregate* operator from InfoSphere Streams in Marco.

Table 3 presents average statistics for the 115 fragments in *Aggregate*. The first column classifies fragments by their code type and the second lists the number of fragments for each type. The remaining columns average the number of tokens and blanks ("Size"), the number of backtracks during oracle query analysis ("Backtracks"), the number of generated C++ compilation units for queries ("Queries"), and the number of helper declarations to disambiguate the C++ syntax ("Decls").

The *Aggregate* operator exercises more C++ specific code types than the micro-benchmarks. For instance, the 22 fragments of type **code**<*cpp*, *mdecl*>, where *mdecl* is the member-declaration nonterminal, generate C++ fields, methods, and constructors. No other macro system generates members of a C++ class and checks syntactic correctness of the generated code. Due to the ambiguity of C++ syntax, our oracle analyzer backtracked speculations 72 times over the

114 fragments. Most backtracking arises form C++ fragments that contain unknown identifiers or expressions statements. Even when an identifier is declared in C++, the gcc parser uses backtracking, so it comes as no surprise that our oracle also backtracks.

8.3 Language Scalability

To add target languages in a traditional safe macro system, the developer must modify the target-language processor, which is usually large and complex. To make matters worse, the modified target-language processor is effectively a branch version, and keeping it up-to-date with the main branch requires additional engineering effort. Adding a target language to Marco requires that the developer write a small plug-in consisting of a simple lexer and three oracles. The oracles wrap unmodified target-language processors. If these processors add or change their error messages, Marco must adapt. We argue that the effort is considerably lesser in the Marco approach.

C++ Plug-in. Like all target-language specific Marco plug-ins, our C++ plug-in consists of a lexical analyzer and three oracles. For the lexical analyzer, we define the *TARGET_TOKEN* terminal in Fig. 3 with a few lines of regular expressions for *identifier* (1), *literal* (5), *keyword* (74), and *preprocessing-op-or-punc* (72) [23]. Most regular expressions are trivial and only *identifier* and *literal* (6) require regular expression operators. Our three oracles consist of 1K+ non-blank source lines of Java. About half of the source lines implement oracle declaration queries, and the other half handle error messages. The error handlers contain 52 regular expressions to classify gcc error messages. In contrast, the gcc source files for the C++ front-end (`cc1plus`) contain 100K+ non-blank C source lines. The hand-written parser in `parser.c` has 14K+ non-blank source lines, and it relies on the semantic analysis in 96K+ non-blank source lines to disambiguate parsing decisions. Our C++ plug-in is much smaller and reuses, unmodified, the sophisticated code base that has been maintained for decades.

Fig. 9 presents an abstracted, static call graph for gcc's 1,400+ error messages. The cc1plus module consists of 53 C source files, 5,400+ procedures, and 67,000+ call sites. Out of 5,400+ procedures, 2,500+ procedures are reachable from the parser (c_parse_file). We use these 2,500+ procedures to over-approximate the syntactic and semantic error messages. We exclude the preprocessing library (libcpp) and backend library (libbackend) by treating them as terminal functions in the call graph. Our analysis assumes that all error messages must go through the three final error reporting functions: error, error_n, and error_at. We exclude the permerror function because it reports "permissive" errors that never shadow any downstream error messages. We label each node with a particular function name in a box or a group of functions in an oval. PARSING represents the functions for recognizing C++ nonterminals in the top-down parser. OTHERS represents the remaining functions. We label several edges with capital letters from A to P because their call sites tentatively characterize the kinds of error messages.

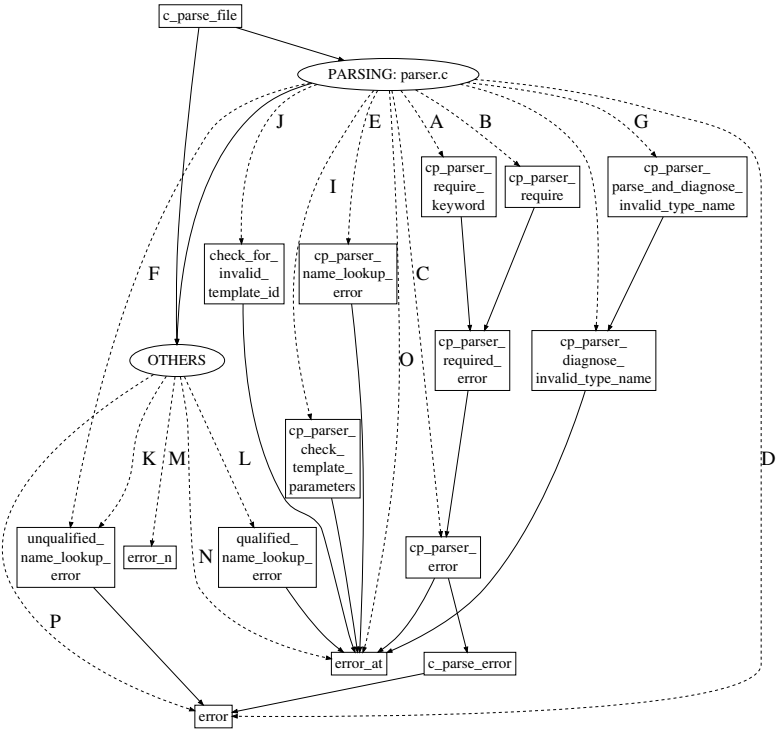

Fig. 9. Abstract Call Graph for error reporting routines

Table 4 maps the labeled call edges to our classifications of error messages. A and B are parsing syntax errors because they expect specific tokens including *keywords*, *punctuation*, and *operators* in C++. C contains 90 syntax errors and 2 semantic errors. E-L are semantic lookups identifying undeclared or unsatisfiable identifiers. D and M-P are mostly semantic errors. Overall, we identified 104 semantic errors that may shadow downstream error messages.

Our oracles recognize 384 critical error messages: 280 parsing error messages, 28 lookup error messages, and 76 other shadowing semantic error messages. A large fraction of these error messages are recognized by a few dozen regular expressions. For instance, all parsing error messages begin with `expected` and end with either a terminal or nonterminal symbol. The lookup error messages begin with `undeclared`.

SQL Plug-in. Our SQL plug-in consists of 40 lines for the lexical analyzer and 400 lines for the three SQL oracles. The lexical analyzer for SQL is simpler than the one for C++. Likewise, SQLite's parser is simpler than gcc's parser: it consists of about 1K source lines in `parser.y` written as an LALR specification. By using SQLite as a black-box language processor, Marco's SQL plug-in reuses not just the parser but also other components for checking naming discipline, all of which have been maintained and tested widely for over a decade.

Table 4. Mapping from calling contexts to error classes

Error Context	Call Sites	Syntax		Semantics		
		Parsing	Post-Parsing	Lookup	Other Shadow	Non-Shadow
A	27	27				
B	176	176				
C	92	73	17		1	1
D	22	3	2		17	
E	5			5		
F	2			2		
G	4			4		
H	2			2		
I	3			3		
J	4			4		
K	3			3		
L	5			5		
M	2					2
N	71					71
O	125	1			7	117
P	1,012				51	961

9 Related Work

Unlike previous macro systems, Marco is safe, well-encapsulated, and target-language agnostic at the same time. In the literature, safe macro systems are deeply coupled with the particular target language and its implementation (Section 9.1), whereas language-agnostic macro systems fail to provide safety guarantees (Section 9.2). Furthermore, while there is previous work that relies on error messages from unmodified language processors, this approach has not previously been applied to macro systems (Section 9.3).

9.1 Language-Specific Safe Macro Systems

Some macro systems check the safety of generated code by deeply coupling the macro language with the target language. Like Marco, these systems enforce macro safety, but, unlike Marco, they are target-language specific.

Syntax. To enforce syntactic well-formedness, previous safe macro systems usually rely on a grammar for the target language. For instance, MS^2 implements a C grammar [26], `metafront` implements grammars for Java and HTML [3], and Ur implements grammars for HTML and SQL [4]. These macro systems approach the level of complexity of extensible compiler toolkits such as ASF+SDF [25], Polyglot [18], or xtc [8,10]. While the approach of implementing a grammar works well enough for targeted language extensions and small domain-specific languages, it is problematic for large, existing languages. Besides the sizable development effort, another issue is compatibility. For example, HTML syntax is deceptively simple, but in practice, HTML processors have so many corner-cases that checkers resort to random testing [1]. Marco side-steps this issue by leveraging unmodified target-language processors for any language.

Scope. In the functional-languages community, a wide-spread technique for ensuring that macros respect scoping rules is *hygiene*. Kohlbecker et al. introduce

hygienic expansion [13]. Clinger and Rees present an improved algorithm for renaming identifiers to guarantee hygiene [5]. Kim et al. formally characterize accidental and intentional capture, but do not implement intentional capture [12]. All these systems depend on the syntax and scoping rules for the chosen target language. In contrast, Marco programmers declare intentional name capture and rely on the system to detect scoping violations with black-box target-language processors. Marco does not automatically rename identifiers, but rather uses dataflow analysis to detect and report errors on accidental name capture.

Semantics. Some macro systems check whether expanded fragments will pass type checking in the target language. Multi-stage extensions generate safe code in, for instance, ML [17], Haskell [21], and Java [27]. C++ concepts add contracts to templates [7]. MorphJ statically verifies some contracts so that expanded code will not have name-resolution conflicts [11]. Quail checks types between SQL queries and the database system [24]. However, target-language agnostic type checking is an open problem that has not been addressed by any of these systems, and we do not yet address it in Marco either.

9.2 Language-Agnostic Unsafe Macro Systems

The idea of macro systems that work for any target language dates back at least to GPM [22]. GPM is credited as an ancestor of M4, a general-purpose preprocessor widely installed on GNU platforms today [15]. Both offer target-language independence, but neither is safe. Perhaps the most used macro system today is the C preprocessor. Ernst et al. present an empirical study that demonstrates numerous violations of safety rules when using the C preprocessor [6]. Furthermore, code-generation based on concrete syntax is widely used in web applications. The PHP language is primarily a code generator for HTML and JavaScript. Programmers often generate SQL code by manipulating strings in general-purpose languages such as Java.

Compared to these systems, Marco adds safety checks while remaining expressive and language-agnostic. Marco relies on high-quality error messages from target-language processors. We believe that our reliance on descriptive error messages aligns well with compiler and interpreter writers who want to provide precise explanations for compilation and execution failures. Fragment code types constrain both the target-language and the nonterminal, enabling syntactic well-formedness checks in isolation. Marco is the first language-agnostic macro system to rely on a dataflow analysis for enforcing naming discipline.

9.3 Using Messages from Black-Box Compilers

A few previous systems rely on error messages from unmodified language processors. Notably, Seminal analyzes error messages from the OCaml and gcc compilers and suggests changes for ill-formed programs [14]. Autoconf compiles specially crafted C/C++ programs, analyzes any error messages, and determines if preprocessor symbols or header files are available in the build environment. The HelpMeOut system mines IDE logs to discover common bug fixes, and then

proposes them to programmers based on currently displayed error messages [9]. Similar to Marco, these systems execute unmodified language processors, and inspect the error messages for clues. Unlike Marco, none of these systems is a macro system. To our knowledge, Marco is the first system that mines error messages from black-box compilers and interpreters for safe code generation.

10 Conclusion

Macros that are expressive, safe, and language scalable at the same time have the potential to significantly improve programmer productivity, particularly for increasingly prevalent multilingual applications. This paper has presented the first such macro system called Marco. Our work is based on two key ideas. First, a plug-in facility provides target-language specific oracles implemented with off-the-shelf compilers and interpreters. In particular, we have identified three simple oracles: syntax, free-names, and captured-name. They are sufficient for ensuring syntactic correctness and naming discipline of macros. Oracles are discharged by submitting specially crafted programs to the target-language processor and then analyzing any resulting error messages. Second, a statically typed quote/unquote facility maximally exploits the target-language independent translation engine. Notably, the type of a fragment specifies its target language and its nonterminal, which the engine uses to invoke the appropriate language-specific oracles. Our evaluation of the Marco prototype supporting C++ and SQL demonstrates the viability of this approach. Future work should explore additional target languages and safety guarantees. Overall, our work demonstrates that safe code generation through macros is orthogonal to language implementation and can be well-encapsulated and language-scalable at the same time.

References

1. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in dynamic web applications. In: ACM International Symposium on Software Testing and Analysis, ISSTA (2008)
2. Bachrach, J., Playford, K.: The Java syntactic extender (JSE). In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2001)
3. Brabrand, C., Schwartzbach, M.I., Vanggaard, M.: The metafront system: Extensible parsing and transformation. Electronic Notes in Theoretical Computer Science 82(3) (December 2003)
4. Chlipala, A.: Ur: Statically-typed metaprogramming with type-level record computation. In: ACM Conference on Programming Language Design and Implementation, PLDI (2010)
5. Clinger, W., Rees, J.: Macros that work. In: ACM Symposium on Principles of Programming Languages, POPL (1991)
6. Ernst, M.D., Badros, G.J., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Transactions on Software Engineering (TSE) 28(12) (December 2002)
7. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Reis, G.D., Lumsdaine, A.: Concepts: Linguistic support for generic programming in C++. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2006)

8. Grimm, R.: Better extensibility through modular syntax. In: ACM Conference on Programming Language Design and Implementation, PLDI (2006)
9. Hartmann, B., MacDougall, D., Brandt, J., Klemmer, S.R.: What would other programmers do? Suggesting solutions to error messages. In: ACM Conference on Human Factors in Computing Systems, CHI (2010)
10. Hirzel, M., Grimm, R.: Jeannie: Granting Java native interface developers their wishes. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2007)
11. Huang, S.S., Smaragdakis, Y.: Expressive and safe static reflection with MorphJ. In: ACM Conference on Programming Language Design and Implementation, PLDI (2008)
12. Kim, I.-S., Yi, K., Calcagno, C.: A polymorphic modal type system for LISP-like multi-staged languages. In: ACM Symposium on Principles of Programming Languages, POPL (2006)
13. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: ACM Conference on LISP and Functional Programming, LFP (1986)
14. Lerner, B.S., Flower, M., Grossman, D., Chambers, C.: Searching for type-error messages. In: ACM Conference on Programming Language Design and Implementation, PLDI (2007)
15. GNU M4 macro processor, `http://www.gnu.org/software/m4/manual/m4.html`
16. Mendell, M., Nasgaard, H., Bouillet, E., Hirzel, M., Gedik, B.: Extending a general-purpose streaming system for XML. In: International Conference on Extending Database Technology, EDBT (2012)
17. Moggi, E., Taha, W., Benaissa, Z.-E.-A., Sheard, T.: An Idealized MetaML: Simpler, and More Expressive (Includes Proofs). In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 193–207. Springer, Heidelberg (1999)
18. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
19. Roskind, J.: Parsing C, the last word. The comp.compilers newgroup (January 1992), `http://groups.google.com/group/comp.compilers/msg/c0797b5b668605b4`
20. Shalit, A.: The Dylan Reference Manual. Addison-Wesley (1996)
21. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. ACM SIGPLAN Notices 37(12) (December 2002)
22. Strachey, C.: A general purpose macrogenerator. The Computer Journal (1965)
23. Stroustrup, B.: The C++ Programming Language. Addison Wesley (2000)
24. Tatlock, Z., Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: Deep typechecking and refactoring. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2008)
25. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: The ASF+SDF compiler. ACM Transactions on Programming Languages and Systems (TOPLAS) 24(4) (July 2002)
26. Weise, D., Crew, R.: Programmable syntax macros. In: ACM Conference on Programming Language Design and Implementation, PLDI (1993)
27. Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: ACM Conference on Programming Language Design and Implementation, PLDI (2010)