

Macrofication: Refactoring by Reverse Macro Expansion

Christopher Schuster, Tim Disney, and Cormac Flanagan

University of California, Santa Cruz
{cschuste, tdisney, cormac}@ucsc.edu

Abstract. Refactoring is a code transformation performed at development time that improves the quality of code while preserving its observable behavior. Macro expansion is also a code transformation, but performed at compile time, that replaces instances of macro invocation patterns with the corresponding macro body or template. The key insight of this paper is that for each pattern-template macro, we can automatically generate a corresponding refactoring tool that finds complex code fragments matching the macro template and replaces them with the equivalent but simpler macro invocation pattern; we call this novel refactoring process *macrofication*.

Conceptually, macrofication involves running macro expansion in reverse; however, it does require a more sophisticated pattern matching algorithm and additional checks to ensure that the refactoring always preserves program behavior.

We have implemented a macrofication tool for a hygienic macro system in JavaScript, integrated it into a development environment and evaluated it by refactoring a popular open source JavaScript library. Results indicate that it is sufficiently flexible for complex refactoring and thereby enhances the development workflow while scaling well even for larger code bases.

1 Introduction

Refactoring is the process of changing code to improve its internal structure without changing its external behavior [15]. Complex restructuring of code usually requires careful design decisions by the developer but refactoring tools still provide support and automation for detecting *code smell*, selecting the right transformation and performing it in a way that preserves behavior.

As an example, consider the JavaScript code fragment in Listing 1, which employs a well-known pattern to define a constructor function `Person` that creates new objects with `sayHello` and `rename` methods.

```
function Person(name) {
  this.name = name;
}
Person.prototype.sayHello = function sayHello() {
  console.log("Hi, I'm " + this.name);
};
Person.prototype.rename = function rename(n) {
  this.name = n;
};
```

Listing 1. JavaScript example with prototype-based inheritance.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log("Hi, I'm " + this.name);
  }
  rename(n) {
    this.name = n;
  }
}
```

Listing 2. Declarative class definition corresponding to Listing 1.

The most recent version of JavaScript (ECMAScript 2015/ES6 [23]) adds declarative class definitions to the language which enable us to simplify the code as shown in Listing 2.

Rewriting existing code to use class definitions instead of the object prototype pattern is a tedious and error-prone process; clearly a refactoring tool to perform this transformation automatically would be desirable.

In addition to class definitions, ES2015/ES6 also adds a more concise arrow syntax (\Rightarrow) for anonymous function definitions, allowing us to rewrite the following example in a more compact way:

```
a.map(function(s) { return s.length; }); // ES5 code
a.map(s => s.length); // equivalent ES2015 code
```

Again, an automatic refactoring tool to perform this transformation for existing code would be most helpful.

Of course, programmers should not have to wait on browser implementations to be able to use such nice syntactic extensions. In fact, many languages allow programmers to define syntactic extensions with *macro* systems. These include string-based macros as commonly used in C [24] or Assembler, and parser-level

```

1 macro class {
2   rule {
3     $cname {
4       constructor $cparams $cbody
5       $($mname $mparams $mbody) ...
6     }
7   } => {
8     function $cname $cparams $cbody
9     $($cname.prototype.$mname = function $mparams $mbody;) ...
10  }
11 }
12 function Person(name) {
13   this.name = name;
14 }
15 Person.prototype.sayHello = function() {
16   console.log("Hello, I'm " + this.name);
17 };

```

Replace with macro?

```

class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log("Hello, I'm " + this.name);
  }
}

```

Fig. 1. The shown sweet.js macro named `class` adds declarative class definitions to JavaScript that expand to a object prototype pattern. Macrofication automatically detects a refactoring candidate in lines 12 to 17, so the development environment highlights the code and shows a preview of the refactored code with the class definition in an overlay.

macros as supported by Lisp [50], Scheme [49], Racket [53] and more recently Rust [1] and JavaScript [10].

In the most general form, a macro is a syntax transformer, a function from syntax to syntax, which is evaluated at compile time. The kind of macros we consider are of a more restricted form called pattern-template macros (in Scheme these macros are defined with `syntax-rules` and are also known as “macro by example” [26,6]). These pattern-template macros are defined with a *pattern* that matches against syntax and a *template* that generates syntax. The template can reference syntax that was matched in the pattern via *pattern variables*. Once all macros have been expanded, the resulting program will be parsed and evaluated according to the grammar and semantics of the target language. In a hygienic macro system [21], the macro expansion also respects the scopes of variables and thereby prevents unintended name clashes between the macro and the expansion context.

Sweet.js [10], a macro system for JavaScript, enables syntactic extensions such as declarative class definitions and arrow notation for functions as described above. As an example, the `class` macro shown in Listing 3 introduces syntax for class definitions by matching a class name, a constructor and an arbitrary number of methods, and expanding to a constructor function and repeated assignments to the prototype of this constructor.

In addition to defining this macro, the programmer also has to rewrite the existing code to benefit from it and be consistent throughout the code base. This involves finding all applicable code fragments (as in Listing 1 above) and replacing them with correct macro invocation patterns (as in Listing 2) which

```

macro class {
  rule {
    $cname {
      constructor $cparams $cbody
      $($mname $mparams $mbody)...
    }
  } => {
    function $cname $cparams $cbody
    $($cname.prototype.$mname = function $mparams $mbody;)...
  }
}

```

Listing 3. Macro for expanding class definitions to ES5 code based on prototypes.

is essentially the class refactoring mentioned above and therefore would equally benefit from automated tool support.

This paper takes of advantage of this similarity between refactoring and macro expansion to introduce *macrofication*, the idea of refactoring via reverse macro expansion. Pattern-template macros, such as the macro for class definitions, allow the algorithm to automatically discover all matching occurrences of a macro template in the program that can be replaced by a corresponding macro invocation.

Fig. 1 shows our development environment with the class macro and the previous code example which uses the object prototype pattern. The macrofication option automatically highlights lines 12 to 17, indicating that the code can be refactored with a macro invocation. Additionally, the environment also shows a preview of the refactored code which is a more readable class definition with the same behavior as the original code. By simply clicking on the preview, the source code will be transformed accordingly.

Conceptually, macrofication is the inverse of macro expansion; macro expansion replaces patterns with templates, whereas macrofication replaces templates with patterns. However, macrofication requires a more complicated matching algorithm than is used in current macro systems due to differences in the handling of macro variables in patterns and templates. For example, variables are often repeated in the template (e.g. `$cname` in Fig. 1) whereas current macro systems do not support repeated variables in patterns. Repetitions (denoted with ellipses ‘...’) introduce additional complexities which we solve with a pattern matching algorithm that takes the nesting level of variables in repetitions into account to enable the correct macrofication of complex macro templates (as illustrated by `$cname` in Fig. 1 line 9 which has to be the same identifier for all methods of a class declaration).

Macrofication should preserve program behavior. Even if the syntax involved in a particular macrofication was replaced correctly, the surrounding code might lead to a different expansion and thereby a different program behavior. Furthermore, a hygienic macro system separates the scopes of variables used in the

macro and in the expansion context, therefore refactoring a scoped variable potentially introduces problems if the refactoring does not account for hygienic renaming. The refactoring algorithm in this paper addresses these issues by ensuring syntactic equivalence after expansion and thereby guarantees that the resulting program behaves the same as the original program.

In addition to the expansion and macrofication algorithm, this paper also evaluates a working prototype implementation for JavaScript based on `sweet.js` including an integration into a development environment which highlights refactoring candidates. This implementation was successfully used to refactor the popular `Backbone.js` JavaScript library by changing its prototype-based code to ECMAScript 6 classes with a complex rule macro. A cursory performance analysis of refactoring both `Backbone.js` and the `ru-lang` library, which uses macros internally, indicates that this approach scales well even for large code bases.

Overall, the contributions of this paper are

- it introduces *macrofication* as a new kind of code refactoring for inferring macro invocations,
- a macrofication algorithm based on reverse expansion that takes the macro expansion order and hygiene into account,
- an advanced matching algorithm for patterns with nested pattern repetitions and repeated pattern variables,
- an implementation for `sweet.js` including an integration into the `sweet.js` development environment,
- and an evaluation of its utility and performance by refactoring `Backbone.js`, a popular JavaScript library.

2 Macro expansion

In order to define macrofication, it is useful to first review how macro expansion works. Our formalism is mostly independent of the target language and only assumes that the code has been lexed into a sequence of tokens which have been further processed into a sequence of *token trees* by matching delimiters such as open and close braces. If k ranges over tokens in the language (such as identifiers, punctuation, literals or keywords), then a token tree h is either a single token k or a sequence enclosed in delimiters $\{s\}$.

$$h ::= k \mid \{s\} \qquad k : Token$$

The *syntax* of the program is simply a sequence s of token trees.

$$s ::= k \cdot s \mid \{s\} \cdot s \mid \epsilon$$

The actual characters used for delimiting token trees are irrelevant for the algorithm, so a sophisticated reader/lexer could support many different delimiters (e.g. `{ }`, `[]` or `()`), including implicit delimiters for syntax trees so this approach supports both Lisp-like and JavaScript-like languages.

As an example, the JavaScript statement “`arr[i+1];`” could be represented as the following token tree sequence where the square brackets “[” and “]” become simple tokens k after delimiter matching.

$$\begin{array}{l} \text{arr} \quad [\quad i \quad + \quad 1 \quad] \quad ; \\ k \quad \cdot \quad \{ k \cdot k \cdot k \cdot k \cdot k \cdot \epsilon \} \cdot k \cdot \epsilon \end{array}$$

A macro has a name n , which is a single token (usually an identifier), and a list of rules. Each rule is a pair of a pattern p and a template t , both of which might include pattern variables x . Pattern variables might be represented with a leading dollar sign $\$$ or question mark $?$, e.g. $\$x$ or $?x$ in the target language but the concrete syntax for pattern variables is insignificant for the algorithm presented in this paper.

Here, we define a pattern or template as a sequence of tokens, variables and pattern/template sequences enclosed in delimiters.

$$p, t ::= k \cdot p \mid \{p\} \cdot p \mid x \cdot p \mid \epsilon \quad x : \text{Pattern Variable}$$

In the context of the expansion and macrofication algorithm, a macro with multiple rules is equivalent to multiple macros with the same name, each having a single rule. Therefore, it is possible to represent all macro rules in the macro environment Σ as an ordered sequence of (name, pattern, template) tuples.

$$\Sigma : (n, p, t)^*$$

A pattern variable x is either unbound or bound to a token tree h . We use Θ to denote the environment of variables bindings.

$$\Theta : x \rightarrow h$$

In the simplest case, all macros are known in advance of the expansion and have global scope¹. Given a fixed list of macros Σ , macro expansion transforms a token tree sequence s (which does not include macros definitions) into a new token tree sequence with all macros matched and expanded:

$$\text{expand}_{\Sigma} : s \rightarrow s$$

For every token k , `expand` will look up its macro environment Σ for a macro named k with a pattern p matching the following tokens. If there is no such macro, it will proceed with the remaining syntax, otherwise the first such macro is used to match the syntax, yielding new variable bindings Θ which are then used to *transcribe* the template t . The resulting token sequence might include other macro calls, so `expand` continues recursively until all macros have been expanded. This process is not guaranteed to terminate as rule macros are Turing-complete. For example, the following macro will result in an infinite expansion:

```
macro omega { rule {} => {omega} }
```

```

macro unless {
  rule { $x $y } => { if (! $x) $y }
}
unless (success) fail();

  expandy (unless · (success) fail());
↔ match (x · y, (success) fail(); , ∅)
→ match (y, fail() ;, [x ↦ (success)])
→ match (ε, ;, [x ↦ (success), y ↦ fail()])
↔ transcribe ( if ( ! x) · y, [x ↦ (success), y ↦ fail()])
→ if (!(success)) fail();

```

Fig. 2. Detailed expansion process of the `unless` macro.

The algorithms for *matching* and *transcribing* generally follow the recursive structure of the provided pattern or template. *Match* uses the pattern p to enforce equivalence with the token tree sequence s while adding variables x to the pattern environment Θ . *Transcribe* uses the template t to generate new syntax s by replacing all free pattern variables x with their substitutions based on Θ .

$$\text{match} : p \times s \times \Theta \rightarrow (\Theta, s) \quad \text{transcribe} : t \times \Theta \rightarrow s$$

Fig. 2 shows a simple example of matching and transcribing as part of macro expansion. The complete algorithm is shown in Fig. 3.

3 Macrofication

The goal of refactoring is to improve the code without changing its behavior. Analogously, macros are often used to introduce a more concise notation for equivalent, expanded code. An automatic refactoring tool in the context of macros could therefore automatically find fragments of code that can be replaced by a corresponding and simpler macro invocation. This section describes an algorithm for this *macrofication* refactoring which is based on pattern-template macros and essentially applies them in reverse, i.e. using the template of the macro for matching code and inserting the macro name and its macro invocation pattern with correct substitutions for variables. However, macro expansion and macrofication are not entirely symmetric due to non-determinism, overlapping macro rules and the way repeated variables are handled.

3.1 Basic reverse matching

Macro expansion uses a deterministic left-to-right recursion to process syntax until all macros have been expanded. This process takes advantage of the fact

¹ A slightly modified algorithm could also match macros and add them to the macro environment during expansion (see also Section 6).

k, n	Token	$h ::= k \mid \{s\}$	Token tree
		$s, r ::= k \cdot s \mid \{s\} \cdot s \mid \epsilon$	Syntax Sequence
x	Pattern Variable	$p, q, t ::= k \cdot p \mid \{p\} \cdot p \mid x \cdot p \mid \epsilon$	Pattern/Template
$\Theta : x \rightarrow h$	Bound variables	$\Sigma : (n, p, t)^*$	Macro Environment

$$\begin{aligned}
& \text{expand}_{\Sigma} : s \rightarrow s \\
& \text{expand}_{\Sigma} (k \cdot s) \quad \hat{=} \text{let } (n, p, t) = \text{first in } \Sigma \text{ s.t. } k = n \wedge \text{match } (p, s, \emptyset) \\
& \quad \quad \quad (\Theta, r) = \text{match } (p, s, \emptyset) \\
& \quad \quad \quad \text{in } \text{expand}_{\Sigma}(\text{transcribe}(t, \Theta) \cdot r) \\
& \text{expand}_{\Sigma} (k \cdot s) \quad \hat{=} k \cdot \text{expand}_{\Sigma} (s) \quad \quad \quad (\text{otherwise}) \\
& \text{expand}_{\Sigma} (\{s\} \cdot s') \quad \hat{=} \{ \text{expand}_{\Sigma} (s) \} \cdot \text{expand}_{\Sigma} (s') \\
& \text{expand}_{\Sigma} (\epsilon) \quad \hat{=} \epsilon \\
& \text{macrofy}_{\Sigma} : s \rightarrow s^* \\
& \text{macrofy}_{\Sigma} (h \cdot s) \quad \hat{=} \{h \cdot r \mid r \in \text{macrofy}_{\Sigma} (s)\} \\
& \quad \quad \quad \cup \{\{r'\} \cdot s \mid h = \{s'\} \wedge r' \in \text{macrofy}_{\Sigma} (s')\} \\
& \quad \quad \quad \cup \{n \cdot \text{transcribe } (p, \Theta) \cdot r' \mid \\
& \quad \quad \quad \quad (n, p, t) \in \Sigma \wedge \text{match } (t, h \cdot s, \emptyset) = (\Theta, r')\} \\
& \text{macrofy}_{\Sigma} (\epsilon) \quad \hat{=} \emptyset \\
& \text{match} : p \times s \times \Theta \rightarrow (\Theta, s) \\
& \text{match } (k \cdot p, k' \cdot s, \Theta) \quad \hat{=} \text{match } (p, s, \Theta) \quad \quad \quad (k = k') \\
& \text{match } (\{q\} \cdot p, \{r\} \cdot s, \Theta) \hat{=} \text{match } (p, s, \Theta') \quad \quad \quad (\text{match } (q, r, \Theta) = (\Theta', \epsilon)) \\
& \text{match } (x \cdot p, h \cdot s, \Theta) \quad \hat{=} \text{match } (p, s, \Theta[x \mapsto h]) \\
& \text{match } (\epsilon, s, \Theta) \quad \hat{=} (\Theta, s) \\
& \text{transcribe} : t \times \Theta \rightarrow s \\
& \text{transcribe } (k \cdot t, \Theta) \quad \hat{=} k \cdot \text{transcribe } (t, \Theta) \\
& \text{transcribe } (\{t\} \cdot t', \Theta) \quad \hat{=} \{ \text{transcribe } (t, \Theta) \} \cdot \text{transcribe } (t', \Theta) \\
& \text{transcribe } (x \cdot t, \Theta) \quad \hat{=} \Theta(x) \cdot \text{transcribe } (t, \Theta) \\
& \text{transcribe } (\epsilon, \Theta) \quad \hat{=} \epsilon
\end{aligned}$$

Fig. 3. Basic macro expansion and macrofication algorithm without repetitions.

that macro invocations always start with the macro name, so if the current head of the syntax sequence does not correspond to a macro, that token will not be part of any other subsequent expansion, so the expansion recursively progresses with the rest of the syntax. During the macrofication process, however, a substitution might cause the refactored code to be part of a bigger pattern that also includes previous tokens as illustrated by the following example.

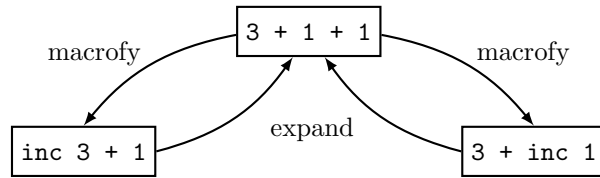
```
macro inc {
  rule { $x } => { $x + 1 }
}
macro inc2 {
  rule { $x } => { 1 + inc $x }
}
```

$$\text{macrofy}_{\Sigma} (1 + \underline{2} + 1) \rightarrow 1 + \underline{\text{inc } 2}$$

$$\text{macrofy}_{\Sigma} (1 + \underline{\text{inc } 2}) \rightarrow \underline{\text{inc2 } 2}$$

Another asymmetry between expansion and macrofication is caused by the fact that different syntax might expand to the same resulting syntax. So, while the expansion process always produces a single deterministic result for a given syntax, the macrofication process produces multiple possible candidates of refactored programs which all expand to the same result and behave identically. In the following example, two different macrofications expand to the same program.

```
macro inc {
  rule { $x } => { $x + 1 }
}
```



For these reasons, macrofication returns a set of programs instead of a single result (see Fig. 3). If h is the head of the syntax and s the tail, the result is the union of three sets:

1. all macrofications of s that do not involve h ,
2. if h is syntax in delimiters $\{s'\}$, then also macrofications of s' , and
3. the program resulting from replacing a matched template t with the macro invocation consisting of the macro name n and substituted pattern p .

It is important to note that algorithm does not recurse on macrofied token trees, so each returned result is a token tree with exactly one step of macrofication. Our development environment based on this algorithm enables the programmer to choose the best refactored program amongst these candidates according to her design decisions and then repeat this process.

3.2 Repeated variables

The pattern matching described in Section 2 as part of the macrofication algorithm processes pattern variables by simply adding the matched token tree to the environment Θ . This corresponds to the common pattern matching behavior in most macro systems. However, the pattern does not enforce variables to be unique, so $x \cdot x$ is a valid pattern. Most existing macros systems including those of Racket [53], Rust [1] and JavaScript [10] do not properly handle repeated pattern variables.

This restriction of the pattern language is usually inconsequential as macro patterns are specially chosen to bind pattern variables in a concise way without unnecessary repetition. However, this repetition is actually intended when pattern variables occur more than once in the template of a macro. For example, the `twice` macro shown in Fig. 4 binds `$f` and `$x` in the pattern `($f $x)` and then uses `$f` multiple times in the template `($f($f($x)))`. The macrofication algorithm described in Section 3 uses this template for pattern matching, therefore it has to handle repeated variable bindings by enforcing the tokens to exactly repeat the previously bound token tree. The following examples illustrate the desired pattern matching for repeated variables in the pattern $x \cdot x$.

$$\begin{aligned} \text{match } (x \cdot x, \quad a \ a, \quad \emptyset) &\rightarrow [x \mapsto a] \\ \text{match } (x \cdot x, \quad a \ b, \quad \emptyset) &\rightarrow \text{no match} \\ \text{match } (x \cdot x, \quad \{a \ b\} \ \{a \ b\}, \quad \emptyset) &\rightarrow [x \mapsto \{a \ b\}] \end{aligned}$$

To support repeated variables, it is possible to extend the match function in the simple algorithm shown in Fig. 3 with an additional case analysis. If the variable x was not assigned before, it gets bound to the corresponding token tree h in the sequence. If, on the other hand, the variable is already part of the pattern environment Θ , then the syntax h has to be identical to the previously bound syntax.

$$\begin{aligned} \text{match } (x \cdot p, \quad h \cdot s, \quad \Theta) &\hat{=} \text{match } (p, \quad s, \quad \Theta[x \mapsto h]) && (x \notin \text{dom}(\Theta)) \\ \text{match } (x \cdot p, \quad h \cdot s, \quad \Theta) &\hat{=} \text{match } (p, \quad s, \quad \Theta) && (\Theta(x) = h) \end{aligned}$$

While this extended matching algorithm correctly handles repeated variables in simple patterns and templates, Section 5 outlines a more sophisticated algorithm which also supports arbitrarily nested pattern repetitions with ellipses.

In contrast to matching repeated variables in patterns, repeated variables in templates are inconsequential for the transcription process. Variables can be used zero or more times in a template without affecting other parts of the transcription process.

4 Refactoring correctness

The macrofication algorithm presented in Section 3 finds all reverse macro matches that could expand again to the original program. In addition, the ad-

```

macro twice {
  rule { $f $x } => { $f($f($x)) }
}
inc(inc(a))

    macrofyΣ (inc(inc(a)))
↔ match (f · (f · (x))), inc (inc(a)), ∅
→ ([x : a, f : inc], ε)
↔ twice · transcribe (f · x, [x : a, f : inc])
→ twice inc a

inc(dec(a))

    macrofyΣ (inc(dec(a)))
↔ match (f · (f · (x))), inc (dec(a)), ∅
→ no match

```

Fig. 4. Macrofication with a macro that uses repeated variables in its template. During the matching process, the pattern variable `$f` will be bound to `inc` at its first occurrence and subsequently matched at all remaining occurrences of `$f`.

vanced pattern matching algorithms described in Sections 3.2 and 5 ensure that even repeated variables are handled correctly. However, this algorithm by itself might inadvertently alter the behavior of the refactored code. In order to guarantee correctness, the refactoring algorithm also needs to take the order of macro expansions and variable scoping in a hygienic macro system into account.

4.1 Problem 1: Conflicts between macro expansions

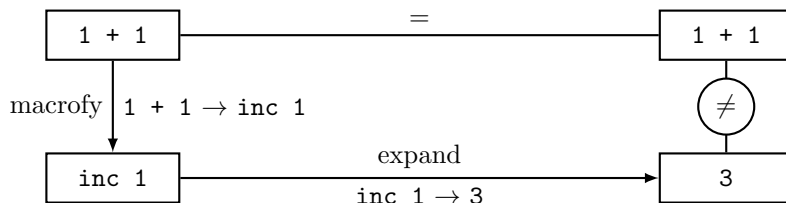
It is possible for multiple macros patterns to overlap. If more than one macro rule matches, the macro expansion algorithm will always expand the first such rule. Due to this behavior, the order of macro rules is significant for the expansion. A naïve refactoring algorithm might inadvertently alter the behavior by refactoring with a rule that is not used during expansion due to other rules with higher priority. As an example, the following macro declares two rules with overlapping patterns.

```

macro inc {
  rule { 1 } => { 3 }
  rule { $x } => { $x + 1 }
}

```

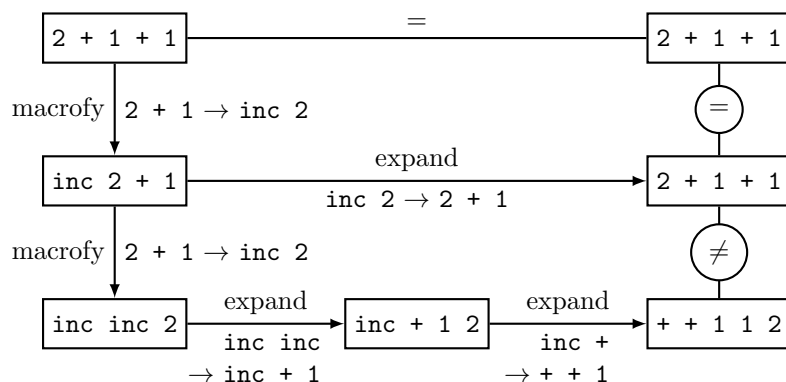
For the program `1 + 1`, macrofication would match the template of the second rule and use it to refactor the program to `inc 1`. However, `inc 1` would macro expand to `3` via the first rule, so macrofication would have changed the behavior of the original program.



In fact, the order of macro expansion also affects refactoring correctness even if there is just one single rule:

```
macro inc {
  rule { $x } => { $x + 1 }
}
```

The program `2 + 1 + 1` can be correctly macrofied to `inc 2 + 1` but a second macrofication on the new program breaks program behavior – despite the fact that both macrofications apply the same rule on the same matched tokens.



In order to prevent the incorrect second macrofication, an improved version of the macrofication algorithm would need to look back at the preceding syntax and consider all macro expansions that might affect the matched code. Unfortunately, there is no clear upper bound on the length of the prefix that has to be considered because macrofication operates on unexpanded token trees which may include additional macro invocations.

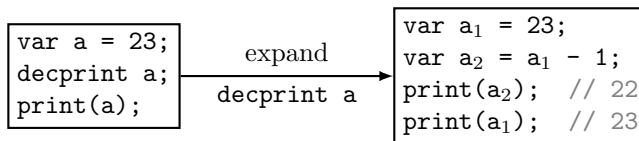
4.2 Problem 2: Hygiene

The basic premise of a hygienic macro system is that macro expansion preserves alpha equivalence, which requires that the scope of variables bound in a macro is separate from the scope in the macro expansion context [21].

So far, the expansion and macrofication algorithms presented in this paper do not address hygiene, scoped variables or the concrete grammar and semantics

of the target language. However, most macro systems used in practice respect hygiene and rename variables accordingly. As an example, the following macro uses an internal variable declaration in its template which will be renamed during hygienic macro expansion.

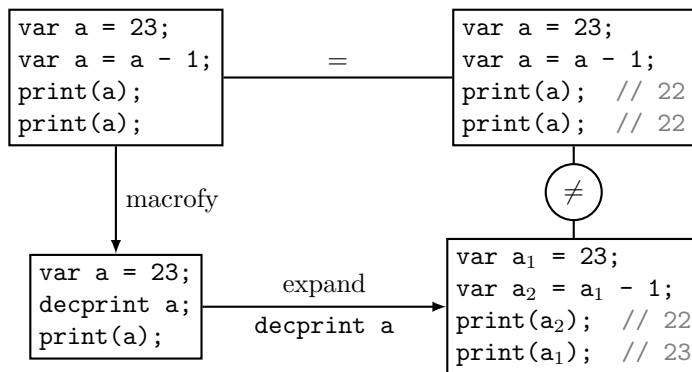
```
macro decprint {
  rule { $x; } => { var a = $x - 1;
                    print(a); }
}
```



The implementation details of hygienic macro expansion are beyond the scope of this paper² but the symmetric relationship between expansion and macrofication suggests that renaming of scoped variables by hygienic expansion also affects macrofication.

In the general case, hygiene is compatible with macrofication as variables with different names in the original code also have different names in the expanded code. The renaming itself is inconsequential for the behavior as long as the expanded macrofied program is α -equivalent to the original program.

However, the same mechanism that ensures that name clashes between the macro and the expansion context are resolved causes problems if the original code actually intended variable names to refer to the same variable binding in the expansion context and the matched macro template. Macrofying this code will result in a macro expansion that inadvertently renames variables and therefore causes the refactored program to diverge from its original implementation.



² For a hygienic macro system for JavaScript, see sweet.js [10].

4.3 Rejecting incorrectly macrofied code

The previous two sections showed macrofied code with different behavior than the original code which has to be avoided for refactoring as behavior-invariant code improvement.

Approaches to fix these problems with an improved matching algorithm are limited by the fact that the correctness of a refactoring operation depends on the surrounding syntax including an arbitrary long prefix (see Problem 1). While this problem could be solved with a complex dynamic check during macrofication, additional difficulties arise from scoped variables that are renamed due to hygiene (see Problem 2). In contrast to the simple expansion and macrofication process, hygiene requires information about variable scopes which are usually defined in terms of parsed ASTs of the program instead of unexpanded token trees that may still include macro invocations.

We address these problems via a simple check performed after the macrofication. It rejects macrofication candidates that, when expanded, are not syntactically α -equivalent to the original program. This simple check successfully resolves these correctness concerns without complicating the macrofication algorithm³.

$$\text{refactor}_{\Sigma}(s) \doteq \{ r \mid r \in \text{macrofy}_{\Sigma}(s) \wedge \text{expand}_{\Sigma}(s) \stackrel{\alpha}{=} \text{expand}_{\Sigma}(r) \}$$

Here, α -equivalence $\stackrel{\alpha}{=}$ is used as alternative to perfect syntactic equivalence which also accommodates hygienic renaming. However, enforcing syntactic equivalence might still reject otherwise valid refactoring opportunities if there are difference that would not affect program behavior, e.g. additional or missing optional semicolons. Further relaxing this equivalence to a broader semantic equivalency might improve the robustness of macrofication but semantic equivalence itself is undecidable in the general case.

5 Repetitions in patterns

Extending the macrofication algorithm described in the previous section with a more expressive pattern and template language does not affect the basic idea of macrofication or the correctness of the results. However, a more sophisticated pattern matching algorithm for matching arbitrarily nested *pattern repetitions* is necessary to correctly support macros like the class macro shown in Fig. 1. The details of the extended algorithm described in this section are not crucial for the remainder of the paper and could be skipped on a first reading.

Pattern repetitions in a pattern allow the use of a single pattern to model an unlimited sequence of that pattern. These pattern repetitions are supported by many macro systems and typically denoted by appending ellipses (...) to the part of the pattern which gets repeated.

$$p, t ::= k \cdot p \mid \{p\} \cdot p \mid x^i \cdot p \mid (p)\dots \cdot p \mid \epsilon$$

³ Additional macro expansions performed as part of this check could potentially be further optimized to improve performance. However, performance problems due to this check did not surface during the evaluation.

Without pattern repetitions, pattern variables can only be assigned a single token tree $h = h^0$. However, if a variable is used in a pattern repetition, it can hold multiple term trees, one for each time the inner pattern was repeated. Pattern repetitions can be nested, so for the purposes of the matching algorithm, every pattern variable x^i has a *level* i which is automatically determined based on the nesting of pattern repetitions. In the simplest case, the level of a variable corresponds to the nesting of repetitions, such that x would have level 0 in the pattern $k \cdot x^0 \cdot k$ and level 1 if in a repetition group like $k \cdot (x^1 \cdot k) \dots$, etc. After a successful match, a variable x^1 will hold a sequence of h^0 token trees, x^2 variables a sequence of sequences of h^0 token trees, and more generally x^i a sequence of h^{i-1} groups.

$$h ::= k \mid \{s\} \qquad h^0 ::= h \qquad h^i ::= h^{i-1} \cdot h^i \mid \epsilon$$

After successfully matching a complete pattern, the final pattern environment Θ always maps variables x^i to groups h^i of the same level. However, the environment used while matching inner patterns builds groups in the pattern environment Θ recursively, so a variable x^i might also hold a group of lower level during the matching process but the level j of its group h^j can never exceed the level i of the variable.

$$\Theta : x^i \rightarrow \bigcup_{0 \leq j \leq i} h^j$$

In order to track the current nesting level during the matching process, the match algorithm shown in Figure 3 has to be extended with an additional parameter $j \in \mathbb{N}$ which will initially be 0 at the top level.

$$\text{match} : p \times s \times \Theta \times \mathbb{N} \rightarrow (\Theta, s)$$

For any nesting level j during the matching process, the intermediate pattern environment Θ always maps free pattern variables x^i in a (sub-)pattern p to groups of level $i - j$.

$$\forall p, s, j. \text{match}(p, s, \emptyset, j) = (\Theta, r) \Rightarrow \forall x^i \in FV(p). \Theta(x^i) \in h^{i-j}$$

5.1 Transcribing templates with repetitions

Transcribing a template $(t) \dots \cdot t'$ with a given environment Θ , *unrolls* all groups used in t and then proceeds with t' . If there is only one group variable x^i in t with length $n = |\Theta(x^i)|$, then the template t will be transcribed n times, each time with a different assignment for x^i . The final result will then be the concatenation of all these repetitions.

$$\begin{aligned} \text{transcribe}(a x^0, & \quad [x^0 \mapsto b]) & \rightarrow a b \\ \text{transcribe}((x^1) \dots, & \quad [x^1 \mapsto [a, b, c]]) & \rightarrow a b c \\ \text{transcribe}((a x^1) \dots, & \quad [x^1 \mapsto [b, c]]) & \rightarrow a b a c \\ \text{transcribe}((x^1) \dots y^0, & \quad [x^1 \mapsto [], y^0 \mapsto a]) & \rightarrow a \\ \text{transcribe}((a (x^2) \dots) \dots, & \quad [x^2 \mapsto [[b, c], [d]]]) & \rightarrow a b c a d \end{aligned}$$

If more than one group variable is used in a repetition, all variables are unrolled at the same time which is equivalent to *zipping* all the groups. The first repetition assigns each x^i the first element of each group, the second repetition assigns each x^i the second element, etc.

$$\text{transcribe}((x^1 y^1)\dots, [x^1 \mapsto [a, b], y^1 \mapsto [c, d]]) \rightarrow a c b d$$

The inner template gets repeatedly transcribed until all the groups are empty which implies that all groups of currently repeating variables need to have the same length.

$$\forall \tilde{\Theta}. \exists n \in \mathbb{N}. \forall x^i \in \text{dom}(\tilde{\Theta}). |\tilde{\Theta}(x^i)| = n$$

As mentioned in Section 3.2, repeated variables in a template are insignificant for the transcription process. The same is true for transcribing templates with pattern repetitions. The complete transcription algorithm is shown in Appendix A/Figure 6.

5.2 Matching patterns with repetitions

Matching a pattern $(p)\dots p'$ is essentially the inverse operation to transcribing a template $(t)\dots t'$. Without repeated variables, the inner pattern p will be greedily matched as many times as possible until finally the remaining syntax s' and a new pattern environment Θ' will be returned and used to match the remaining pattern p' . Instead of destructing groups as in the transcription algorithm, each repetition constructs groups by adding the matched syntax to the corresponding group for all repeating variables.

$$\begin{aligned} \text{match}(a x^0, a b, \emptyset) &\rightarrow [x^0 \mapsto b] \\ \text{match}((a y^1)\dots, a b a c, \emptyset) &\rightarrow [y^1 \mapsto [b, c]] \\ \text{match}(x^0(y^1)\dots, a b c, \emptyset) &\rightarrow [x^0 \mapsto a, y^1 \mapsto [b, c]] \\ \text{match}((x^1 y^1)\dots, a b c d, \emptyset) &\rightarrow [x^1 \mapsto [a, c], y^1 \mapsto [b, d]] \end{aligned}$$

Unfortunately, the greedy matching of repetitions does not support patterns like $(a)\dots a$ as the repetition would have consumed all a tokens at the point the second a would try to match. A more sophisticated pattern matching algorithm might use either lookahead or backtracking to prevent or recover from consuming too many tokens in a repetition. However, this matching would be less efficient and macros with these kinds of pattern repetitions are unusual in practice.

5.3 Matching repeated variables in patterns with repetitions

As explained in Section 3.2, the pattern matching necessary for macrofication also needs to support repeated variables in patterns and templates. If a pattern

variable is repeated at the same group level, the number of times the group matches as well as all matched token trees have to be identical.

$$\begin{aligned} \text{match}(x^0 b x^0, \quad a b a) &\rightarrow [x^0 \mapsto a] \\ \text{match}(x^0 b x^0, \quad a b c) &\rightarrow \text{no match} \\ \text{match}(\{(x^1)\dots\}(x^1)\dots, \{a b\} a b) &\rightarrow [x^1 \mapsto [a, b]] \\ \text{match}(\{(x^1)\dots\}(x^1)\dots, \{a b\} a c) &\rightarrow \text{no match} \\ \text{match}((a x^1)\dots (x^1)\dots, a b a c b c) &\rightarrow [x^1 \mapsto [b, c]] \\ \text{match}((a x^1)\dots (x^1)\dots, a b a c b d) &\rightarrow \text{no match} \end{aligned}$$

If the pattern variable is used once inside and once outside a repetition, all occurrences of that variable within the repetition have to repeat the same syntax as outside the repetition. This means that the variable assignment will be constant while repeatedly matching the pattern repetition, essentially using a lower level than the nesting would indicate.

As an example, the pattern $x^0(x^0 y^1)\dots$ uses two variables x^0 and y^1 where y^1 is only used once and in a repetition, so a successful match will result in a group of tokens h^1 with one assignment per repetition. In contrast, the variable x^0 is used multiple times, so every occurrence of x^0 in the pattern has to match the exact same syntax. Since x^0 is used outside of repetitions, its final assignment has to be a single token h^0 and additionally, all repetitions have to repeat this exact same syntax – instead of building up a group.

$$\begin{aligned} \text{match}(x^0(x^0 y^1)\dots, \quad a a b a c) &\rightarrow [x^0 \mapsto a, y^1 \mapsto [b, c]] \\ \text{match}(x^0(x^0 y^1)\dots, \quad a a b d c) &\rightarrow \text{no match} \\ \text{match}(x^0(x^0 y^0)\dots y^0, \quad a a b b) &\rightarrow [x^0 \mapsto a, y^0 \mapsto b] \\ \text{match}(x^0(x^0 y^0)\dots y^0, \quad a a b a b b) &\rightarrow [x^0 \mapsto a, y^0 \mapsto b] \\ \text{match}(x^0(x^0 y^0)\dots y^0, \quad a a b a b c) &\rightarrow \text{no match} \end{aligned}$$

This causes the matching processes to become more complicated as variable levels can diverge from the level of nesting. If the level of a variable is higher in the template than in the pattern, it will be matched and used as a lower level variable, i.e. as constant in a pattern repetition, in order to be compatible with the pattern. This is especially important for macrofication, as the template might use variables within a repetition that are assumed constant in the pattern. For example, the class name variable `$cname` in the template of the `class` macro in Fig. 1 appears once on the top level and once inside the repetition for every method, so the matching algorithm has to ensure that all methods use the same class name and therefore treat `$cname` as a constant at each repetition.

In order to support repeated variables in patterns with repetitions, it is necessary to extend the match algorithm. Conceptually, the first time a group variable $x^{i \geq 1}$ is encountered in a pattern, the elements are collected by greedily matching syntax and recursively constructing a group h^i . However, once a pattern variable has been assigned, all subsequent uses of that variable in a repetition will cause

the pattern repetition to be unrolled following the approach of the transcribe algorithm described in Section 5.1.

Fig. 6 in Appendix A shows the complete algorithm for matching and transcribing arbitrarily nested *pattern repetitions* with repeated variables to correctly support macros like the class macro shown in Fig. 1.

6 Implementation

Our implementation is part of `sweet.js`, a hygienic macro system for JavaScript which supports pattern-template macros [10]. The source code⁴ as well as a live online demo⁵ are both publicly available, and `sweet.js` is now using the extended pattern matching algorithm for macrofication and regular macro expansion.

Much of our implementation is a straightforward application of the algorithms described in the previous sections. However, there are a few JavaScript specific details. In particular, due to the complexity of JavaScript’s grammar, `sweet.js` provides the ability for a pattern variable to match against a specific *pattern class* in addition to matching on a single or repeating token. A pattern class also allows a macro to match on multiple tokens, e.g. all tokens in an expression. To restrict a pattern variable `$x` to match an expression, the programmer can annotate the variable with the pattern class `:expr` (see Listing 4).

```
macro m {
  rule {
    ($bind:($id:ident = $val:expr) (,)...)
  } => {
    $(var $bind;) ...
  }
}
m a = 2 + 1, // --> var a = 2 + 1;
  b = 3      //      var b = 3;
```

Listing 4. A `sweet.js` macro with pattern classes `:ident`, `:expr`, a named pattern (`$bind`) and ellipses (`...`). In order to support macrofication of this macro, the pattern classes used in the pattern also have to apply to pattern variables in the template.

Considering the code fragment `arr[i + 1]`, a pattern variable `$x` matches just the single token `arr` whereas the pattern `$x:expr` matches the entire expression `arr[i + 1]`. Pattern class annotations only appear in patterns, not templates, so to support pattern classes in macrofication we move pattern class annotations from the pattern to the corresponding variables in the template prior to matching the template with the code.

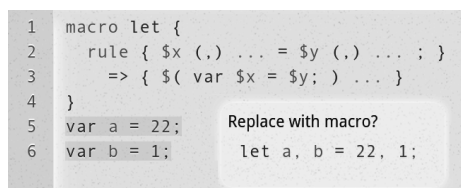
Another difference between the algorithm in Fig. 3 and the implementation in `sweet.js` is the handling of the macro environment Σ . The algorithm assumes that the macro definitions are clearly separated from the program and globally scoped.

⁴ <http://github.com/mozilla/sweet.js> (see `src/reverse.js`, `src/patterns.js`)

⁵ <http://sweetjs.org/browser/editor.html>

In contrast, `sweet.js` macro definitions are defined in the code and cannot be used unless in scope. The current implementation of the refactoring algorithm only supports global macros but could be modified such that the macro environment Σ respects the scopes of macro definitions.

The `sweet.js` refactoring tool is usable from the command line as well as in the web-based `sweet.js` editor. Fig. 1 and 5 show screen shots of this editor integration. As discussed in Section 3, not all refactoring options actually improve the code and could be mutually exclusive. To solve this issue, the development environment displays all options by highlighting code and opening a pop-up overlay of the refactored code on demand. This integration provides unobtrusive visual feedback about refactoring opportunities but other ways to displaying these may be preferable if there is large number of macrofication candidates.



```

1 macro let {
2   rule { $x (, ) ... = $y (, ) ... ; }
3     => { $( var $x = $y; ) ... }
4 }
5 var a = 22;
6 var b = 1;

```

Replace with macro?
let a, b = 22, 1;

Fig. 5. A `sweet.js` macro which expands a parallel `let` declaration to multiple single declarations. The editor automatically detects a refactoring candidate in line 5 and 6 and shows a preview of the substituted code.

7 Evaluation

We evaluated the utility and performance of the macrofication refactoring tool by performing a complex refactoring of a JavaScript library with a specifically tailored macro and second case study on a JavaScript project with a large number of existing macros.

7.1 Experimental Results

Macros can be used to extend the language with additionally facilities that are not part of the grammar. For JavaScript, one of the most requested language features is a declarative `class` syntax, which can be desugared to code with prototypical inheritance (see Fig. 1). Indeed, the most recent version of JavaScript (ECMAScript 2015/ES6 [23]) adds class definitions to the language⁶.

A particularly popular JavaScript framework that relies on inheritance to integrate with user-provided code is Backbone.js⁷. It is open source, widely deployed and has 1633 lines of code. The prototype objects defined by Backbone.js

⁶ See also the `es6-macros` project [31] which includes macros for many ES6 features.

⁷ <http://backbonejs.org/>

```

macro class {
  rule {
    $name extends Events {
      constructor $cargs $cbody
      $( $mname $margs $mbody ) ...
    }
  } => {
    var $name = Backbone.$name = function $cargs $cbody;
    _.extend($name.prototype, Events.prototype, {
      $(
        $mname: function $margs $mbody
      ) (,) ...
    });
  }
}

```

Listing 5. Custom class macro for refactoring Backbone.js.

generally adhere to a simple class-based inheritance approach. Therefore, the code would benefit from declarative class definitions in the language.

Refactoring the Backbone.js code by automatic macrofication required a custom class macro which matches the concrete pattern used by Backbone.js to declare prototypes with the `_.extend` function. Here, ‘`_`’ is a variable in the Backbone.js library with common helper functions like `extend` to add properties to objects. Since the Backbone.js code does not use any *super* calls, the simple macro shown in Listing 5 is sufficient to desugar classes to the prototype pattern used in Backbone.js. As additional manual refactoring step, non-function default properties in the Backbone.js code had to be moved into the constructor since they are not yet supported by the ES2015/ES6 class syntax⁸. After this minor change in the code, the sweet.js macrofication successfully identified all five prototypes used in Backbone.js and refactored these with class declarations without changing the program behavior.

A second case study was performed using the open source project *ru*⁹ which is a collection of 66 macro rules for JavaScript inspired by Clojure. For refactoring the ru-lang library, only 27 macro rules were considered because case macros and custom operators are not currently supported by the macrofication tool. While the tool reported a large number of correct macrofication options, some of these did not improve the code quality. For example, some macrofication candidates introduce an invocation of the `cond` macro with just a single default `else` branch. While this macrofication correctly expands to the original code, it essentially corresponds to replacing a JavaScript statement “`x;`” with “`if (true) x;`”.

⁸ It would also be possible to perform this transformation automatically with a more complex and less generic macro.

⁹ <http://ru-lang.org/>

Project	LOC	Time to read	Time to refactor	Macros	Macrofications
Backbone.js	1633	151ms	984ms	1	5
ru-lang	257	1350ms	17921ms	27	52

Table 1. Results of refactoring the JavaScript libraries Backbone.js and ru-lang.

Table 1 shows the runtime of the macrofication step and the reading step as measured with the `sweet.js` command line running on NodeJS v0.11.13; all times reported averaged across 10 runs. The macrofication step including the expansion of the refactored code was about 6.5 to 13 times slower than the time to read/lex the input and load the macro environment. While future optimizations could improve performance, the runtime of macrofication seems generally feasible.

7.2 Discussion

Overall, the experimental results show that macrofication has major advantages over a manual refactoring approach.

1. Macrofication is guaranteed to preserve the behavior of the program and hence avoids the risks of human error.
2. The time and effort of the refactoring is dominated by the time and effort of writing the macros. Refactoring code with a given macro requires little manual effort, is fast enough for interactive use in an editor and scales well even for large code bases.

However, the experiment also showed three limitations of macrofication.

1. The macro has to be pre-existing or provided by the programmer in advance of the refactoring.
2. While small macros can be generic, larger macros may need to be specifically tailored to the code.
3. Minor differences between the macro template and the code, e.g. the order of statements or additional or missing semicolons in a language with optional semicolons, cause the macrofication algorithm to miss a potential refactoring option due to the strict syntactic equivalence check of the algorithm.

The first limitation could be overcome with an algorithm for automated macro synthesis/inference which might be a promising area for future research (see Section 9).

The second limitation applies to all currently used macro systems to a certain degree. Small, generic macros, e.g. new syntax for loops, may be universally applicable but larger macros are usually specific to the code. For macrofication, this applies both to the pattern as well as its template. For example, the class macro shown in Fig. 1 had to be adapted for refactoring Backbone.js.

The programmer can work around the third limitation by specifying multiple macro rules with the same pattern but in order to tolerate discrepancies between the template and the unrefactored code during the matching process, it would be helpful to remove the syntactic equivalence constraint in favor of behavioral equivalence based on the semantics of the language. This is difficult to integrate into the refactoring as semantic equivalence is generally undecidable. A conservative and decidable approximation of semantic equivalence that is more precise than syntactic equivalence might significantly help macrofication but remains a topic for future work.

8 Related Work

Our tool combines ideas from two streams of research, macro systems that give programmers additional language abstractions through syntactic extensibility and automated refactoring tools for code restructuring.

8.1 Macro Systems

Macros have been extensively used and studied in the Lisp family of languages for many years [14,37]. Scheme in particular has embraced macros, pioneering the development of declarative definitions [26] and hygiene conditions for term rewriting macros (rule macros) [6] and procedural macros (case macros) [22]. In addition there has been work to integrate procedural macros and module systems [13,19]. Racket takes this work even further by extending the Scheme macro system with deep hooks into the compilation process [53,12] and robust pattern specifications [7].

Recently work has begun on formalizing hygiene for Scheme [2]. Prior presentations of hygiene have either been operational [22] or restricted to a typed subset of Scheme that does not include `syntax-case` [21].

Languages with macro systems of varying degrees of expressiveness not based on S-expressions include Fortress [4], Dylan [5], Nemerle [48], and C++ templates [3]. Template Haskell [47] makes a tradeoff by forcing the macro call sites to always be demarcated. This means that macros are always a second class citizen; macros in Haskell cannot seamlessly build a language on top of Haskell in the same way that Scheme and Racket can.

Some systems such as SugarJ [11], and OMeta [56] provide extensible grammars but require the programmer to reason about parser details. Multi stage systems such as mython [42] and MetaML [52] can also be used to create macros systems like MacroML [16]. Some systems like Stratego [54] and Marco [28] transform syntax using their own language, separate from the host language.

As mentioned before, our tool is built on top of `sweet.js` [10] which enables greater levels of macro expressiveness without s-expressions as pioneered by Honu [41,40], a JavaScript-like language. ExJS [55] is another macro system for JavaScript however their approach is based on a staged parsing architecture (rather than a more direct manipulation of syntax as in Lisp/Scheme and `sweet.js`) and thus they only support pattern macros.

While the goal of macrofication is to introduce new syntactic sugar, recent work on *Resugaring* aims to preserve or recover syntactic sugar during the execution to improve debugging [38,39]. In contrast to macrofication, resugaring at runtime operates on ASTs of a concrete language rather than syntax trees.

Macro systems can be generalized to term rewriting systems which have been studied extensively in the last decades. Most noteworthy, it might be possible to statically analyze properties like confluence and overlapping of macro rules (as discussed in Section 4.1) by adapting prior research on orthogonal term rewriting systems [25].

8.2 Refactoring

Refactoring [15,33] as an informal activity to improve the readability and maintainability of code goes back to the early days of programming. Most currently used development environments for popular languages provide built-in automated refactoring tools, e.g. Visual Studio, Eclipse or IntelliJ IDEA.

Early formal treatments look into automated means of refactoring functional and imperative [20] and object-oriented programs [34] that preserve behavior. Since then much work has been done on building tools that integrate automated refactoring directly into the development environment [43], find code smells like duplicated code [29], correctly transform code while preserving behavior [36,44,45,46], and improve the user experience during refactoring tasks [18].

Additionally, prior work on generic refactoring tools includes scripting and template languages for refactoring in Erlang [30], Netbeans [27] and Ekeko/X [8]. However, while these refactoring languages operate on parsed ASTs, macros describe a program transformation in terms of unexpanded token trees.

Much of the work relating refactoring and macro systems have taken place in the context of the C preprocessor (cpp), which introduces additional complexity in traditional refactoring tasks since cpp works at the lexical level rather than the syntactic level and can expand to fragments of code. Garrido [17] addresses many of the refactoring issues introduced by cpp and Overbey et al. [35] systematically address many more by defining a preprocessor dependency graph.

Kumar et al. [51] present a *demacrofying* tool that converts macros in an old C++ code base to new language features introduced by C++11. In a sense they perform the opposite work of macrofication; where demacrofying removes unnecessary macros to aid in the clarity of a code base our refactoring macros add macro invocations to a code base to similar effect.

8.3 Pattern Matching

Pattern matching in macro systems is part of a broad class of pattern matching algorithms. In particular, the handling of repeated variables in the extended pattern matching algorithm in Section 5 is conceptually a first-order syntactical unification which is well known in the context of logic programming languages [32].

In a broader sense, the macrofication algorithm is also related to research on optimizing compilers, e.g. reverse inlining to decrease code size [9].

9 Future Work

While the algorithm is based on refactoring macro invocations, it would also be possible to perform non-macro refactorings with this approach. For example, identifiers can be renamed with a simple, temporary, scoped macro.

As discussed in Section 6, the macrofication algorithm presented in this paper assumes a static macro environment. Future work could extend this algorithm such that it also refactors macro definitions, modifies macro templates, removes existing overlapping macros, or even automatically synthesizes new macros. However, the search space of possible macros is vast, so a carefully designed search which optimizes some metric for code quality would be necessary to provide only the best macro candidates to the programmer.

An additionally promising topic of future research is the extension of the presented algorithm to `syntax-case` macros. In contrast to pattern-template macros, `syntax-case` macros use a generating function instead of a template. Finding refactoring options therefore needs to find syntax that can be generated by a macro which is equivalent to finding the input of a function given its output. Despite the undecidable nature of this problem, it might still be useful to find an incomplete subset of potential macro candidates.

10 Conclusions

The algorithm presented in this paper allows automatic refactoring by macrofying code with a given set of pattern-template macros. The algorithm correctly handles repeated variables and repetitions in the pattern and template of a macro with an extended pattern matching algorithm. The order of macro expansions and hygienic renaming cause a naïve macrofication approach to produce incorrect results. To ensure that the behavior is preserved during refactoring, the algorithm checks syntactic α -equivalence of the fully expanded code before and after the macrofication. The algorithm is language-independent but was evaluated for JavaScript with an implementation based on `sweet.js` and used to refactor `Backbone.js`, a popular JavaScript library with more than one thousand lines of code. The runtime performance indicates that the approach is feasible even for large code bases. Finally, the IDE integration supports and automates the macro development process with promising extensions for future research.

11 Acknowledgements

This research was supported by the National Science Foundation under grants CCF-1337278 and CCF-1421016.

k, n	Token	$h ::= k \mid \{s\}$	Token tree
		$s, r ::= k \cdot s \mid \{s\} \cdot s \mid \epsilon$	Syntax Sequence
x^i	Variable	$p, q, t ::= k \cdot p \mid \{p\} \cdot p \mid x^i \cdot p \mid (p) \dots \cdot p \mid \epsilon$	Pattern/Template
$i, j : \mathbb{N}$	Levels	$h^0 ::= h$ $h^i ::= h^{i-1} \cdot h^i \mid \epsilon$	Group of level i
		$\Sigma : (n, p, t)^*$	Macro Environment
		$\Theta : x^i \rightarrow \bigcup_{0 \leq j \leq i} h^j$	Pattern Environment
		$\tilde{\Theta} : x^i \rightarrow \bigcup_{1 \leq j \leq i} h^j$	Repetition Environment

$\text{match} : p \times s \times \Theta \times \mathbb{N} \rightarrow (\Theta, s)$ (See Fig. 3)

$\text{match}((p) \dots \cdot p', s, \Theta, j) \hat{=} \text{let } (\Theta', s') = \text{matchRep}(p, s, \text{groups}(p, \Theta), \Theta, j)$
 in $\text{match}(p', s', \Theta', j)$

$\text{matchRep} : p \times s \times \tilde{\Theta} \times \Theta \times \mathbb{N} \rightarrow (\Theta, s)$

$\text{matchRep}(p, s, \tilde{\Theta}, \Theta, j) \hat{=} \text{if } \forall x^i \in \text{dom}(\tilde{\Theta}). \tilde{\Theta}(x^i) \neq \epsilon$
 let $\Theta_r = \text{repeating}(\Theta, \Theta')$
 $(\Theta', s') = \text{match}(p, s, \Theta \uparrow \text{head}(\tilde{\Theta}), j + 1)$
 $(\Theta'', s'') = \text{matchRep}(p, s', \text{tail}(\tilde{\Theta}), \Theta' \setminus \Theta_r, j)$
 in $(\Theta'' \uparrow \text{cons}(\Theta_r, \Theta''), s'')$

$\text{matchRep}(p, s, \tilde{\Theta}, \Theta, j) \hat{=} \text{if } \forall x^i \in \text{dom}(\tilde{\Theta}). \tilde{\Theta}(x^i) = \epsilon$
 $(\Theta [x^i \mapsto \epsilon \mid x^i \in FV(p) \wedge x^i \notin \text{dom}(\Theta) \wedge i > j], s)$

$\text{transcribe} : t \times \Theta \rightarrow s$ (See Fig. 3)

$\text{transcribe}((t) \dots \cdot t', \Theta) \hat{=} \text{transcribeRep}(t, \text{groups}(t, \Theta), \Theta) \cdot \text{transcribe}(t', \Theta)$

$\text{transcribeRep} : t \times \tilde{\Theta} \times \Theta \rightarrow s$

$\text{transcribeRep}(t, \tilde{\Theta}, \Theta) \hat{=} \text{if } \forall x^i \in \text{dom}(\tilde{\Theta}). \tilde{\Theta}(x^i) \neq \epsilon$
 $\text{transcribe}(t, \Theta \uparrow \text{head}(\tilde{\Theta})) \cdot \text{transcribeRep}(t, \text{tail}(\tilde{\Theta}), \Theta)$

$\text{transcribeRep}(t, \tilde{\Theta}, \Theta) \hat{=} \text{if } \forall x^i \in \text{dom}(\tilde{\Theta}). \tilde{\Theta}(x^i) = \epsilon$
 ϵ

$\Theta \uparrow \Theta' \hat{=} \Theta [x^i \mapsto \Theta'(x^i) \mid x^i \in \text{dom}(\Theta')]$

$\text{cons}(\Theta, \tilde{\Theta}) \hat{=} [x \mapsto \Theta(x) \cdot \tilde{\Theta}(x) \mid x \in \text{dom}(\Theta) \wedge x \in \text{dom}(\tilde{\Theta})]$

$\text{head}(\tilde{\Theta}) \hat{=} [x^i \mapsto h^{j-1} \mid \tilde{\Theta}(x^i) = h^{j-1} \cdot h^j]$

$\text{tail}(\tilde{\Theta}) \hat{=} [x^i \mapsto h^j \mid \tilde{\Theta}(x^i) = h^{j-1} \cdot h^j]$

$\text{groups}(p, \Theta) \hat{=} [x^i \mapsto h^j \mid x^i \in FV(p) \wedge \Theta(x^i) = h^j \wedge j \geq 1]$

$\text{repeating}(\Theta, \Theta') \hat{=} [x^i \mapsto h^j \mid \Theta'(x^i) = h^j \wedge x^i \notin \text{dom}(\Theta) \wedge i > j]$

Fig. 6. (Appendix A) The complete algorithm for matching and transcribing patterns/templates with repeated variables and arbitrarily nested pattern repetitions as described in Section 5. Differences with the algorithm in Fig. 3 are shown in black.

References

1. The Rust Language. <http://www.rust-lang.org/>, <http://www.rust-lang.org/>
2. Adams, M.D.: Towards the Essence of Hygiene. In: POPL '15 (2015)
3. Alexandrescu, A.: Modern C++ design: generic programming and design patterns applied. Addison-Wesley (2001)
4. Allen, E., Culpepper, R., Nielsen, J.: Growing a syntax. FOOL '09 (2009)
5. Bachrach, J., Playford, K., Street, C.: D-Expressions: Lisp Power, Dylan Style. Style DeKalb IL (1999)
6. Clinger, W.: Macros that work. In: POPL '91 (1991)
7. Culpepper, R.: Refining Syntactic Sugar: Tools for Supporting Macro Development. Ph.D. thesis, Northeastern University Boston (2010)
8. De Roover, C., Inoue, K.: The Ekeko/X Program Transformation Tool. pp. 53–58. SCAM'14 (Sept 2014)
9. Debray, S.K., Evans, W., Muth, R., De Sutter, B.: Compiler techniques for code compaction. ACM Trans. Program. Lang. Syst. 22(2), 378–415 (Mar 2000)
10. Disney, T., Faubion, N., Herman, D., Flanagan, C.: Sweeten Your JavaScript: Hygienic Macros for ES5. In: Proceedings of the 10th ACM Symposium on Dynamic Languages. pp. 35–44. DLS '14, ACM, New York, NY, USA (2014)
11. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: SugarJ: library-based syntactic language extensibility. OOPSLA '11 (2011)
12. Flatt, M., Culpepper, R.: Macros that Work Together. JFP (2012)
13. Flatt, M.: Composable and compilable macros: You Want it When? ICFP '02 pp. 72–83 (2002)
14. Foderaro, J.K., Sklower, K.L., Layer, K.: The FRANZ Lisp Manual. University of California Berkeley, California (1983)
15. Fowler, M.: Refactoring: Improving the design of existing code. Pearson Education India (1999)
16. Ganz, S., Sabry, A., Taha, W.: Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. ICFP '01 (2001)
17. Garrido, A., Johnson, R.: Challenges of refactoring C programs. In: IWPSE '02. pp. 6–14 (2002)
18. Ge, X., DuBose, Q.L., Murphy-Hill, E.: Reconciling manual and automatic refactoring. ICSE '12 pp. 211–221 (2012)
19. Ghuloum, A., Dybvig, R.K.: Implicit phasing for R6RS libraries. ICFP '07 (2007)
20. Griswold, W.G.: Program restructuring as an aid to software maintenance. Ph.D. thesis, University of California, San Diego (1992)
21. Herman, D., Wand, M.: A Theory of Hygienic Macros. In: ESOP '08 (2008)
22. Hieb, R., Dybvig, R., Bruggeman, C.: Syntactic abstraction in Scheme. Lisp and symbolic computation 5(4), 295–326 (1992)
23. International, E.C.M.A.: ECMA-262 ECMAScript Language Specification. ECMA Script, 6 / 2015 edn. (2015)
24. Kernighan, B.W., Ritchie, D.M.: The C programming language, vol. 2. Prentice Hall (1988)
25. Klop, J.W., Klop, J.: Term Rewriting Systems. Centrum voor Wiskunde en Informatica (1990)
26. Kohlbecker, E.E., Wand, M.: Macro-by-example: Deriving syntactic transformations from their specifications. In: POPL '87 (1987)
27. Lahoda, J., Bečička, J., Ruijs, R.B.: Custom Declarative Refactoring in NetBeans: Tool Demonstration. pp. 63–64. WRT '12, ACM, New York, NY, USA (2012)

28. Lee, B., Grimm, R., Hirzel, M., McKinley, K.: Marco: safe, expressive macros for any language. ECOOP'12 (2012)
29. Li, H., Thompson, S.: Clone Detection and Removal for Erlang/OTP Within a Refactoring Environment. PEPM '09, ACM (2009)
30. Li, H., Thompson, S.: Let's Make Refactoring Tools User-extensible! pp. 32–39. WRT '12, ACM, New York, NY, USA (2012)
31. Long, J.: The ES6-macros project. <https://github.com/jlongster/es6-macros>
32. Martelli, A., Montanari, U.: An efficient unification algorithm. TOPLAS'82 (1982)
33. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Transactions on Software Engineering (2004)
34. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis (1992)
35. Overbey, J.L., Behrang, F., Hafiz, M.: A foundation for refactoring C with macros. In: FSE '14 (2014)
36. Overbey, J.L., Johnson, R.E.: Differential precondition checking: A lightweight, reusable analysis for refactoring tools. ASE (2011)
37. Pitman, K.M.: Special forms in Lisp. In: LFP '80 (1980)
38. Pombrio, J., Krishnamurthi, S.: Resugaring: Lifting evaluation sequences through syntactic sugar. pp. 361–371. PLDI '14, ACM, New York, NY, USA (2014)
39. Pombrio, J., Krishnamurthi, S.: Hygienic resugaring of compositional desugaring. pp. 75–87. ICFP'15, ACM, New York, NY, USA (2015)
40. Raffkind, J.: Syntactic extension for languages with implicitly delimited and infix syntax. Ph.D. thesis, The University of Utah (2013)
41. Raffkind, J., Flatt, M.: Honu: Syntactic Extension for Algebraic Notation through Enforestation. GPCE '12 (2012)
42. Riehl, J.: Language embedding and optimization in mython. DLS '09 (2009)
43. Roberts, D., Brant, J., Johnson, R.E.: A Refactoring Tool for Smalltalk. TAPOS 3(4), 253–263 (1997)
44. Schäfer, M., De Moor, O.: Specifying and implementing refactorings. OOPSLA '10 (2010)
45. Schäfer, M., Dolby, J., Sridharan, M., Torlak, E., Tip, F.: Correct Refactoring of Concurrent Java Code. ECOOP '10 (2010)
46. Schäfer, M., Verbaere, M., Ekman, T., De Moor, O.: Stepping Stones over the Refactoring Rubicon. ECOOP '09 (2009)
47. Sheard, T., Jones, S.: Template meta-programming for Haskell. Workshop on Haskell (2002)
48. Skalski, K., Moskal, M., Olszta, P.: Meta-programming in Nemerle. GPCE '04 (2004)
49. Sperber, M., Dybvig, R.k., Flatt, M., Van straaten, A., Findler, R., Matthews, J.: Revised⁶ Report on the Algorithmic Language Scheme. JFP (2009)
50. Steele, Jr., G.L.: Common LISP: The Language. Digital Press, MA, USA (1990)
51. Stroustrup, B., Kumar, A., Sutton, A.: Rejuvenating C++ programs through demacrofication. In: ICSM '12 (2012)
52. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. PEPM '97 (1997)
53. Tobin-Hochstadt, S., St-Amour, V.: Languages as libraries. PLDI '11 (2011)
54. Visser, E.: Program transformation with Stratego/XT. Domain-Specific Program Generation (2004)
55. Wakita, K., Homizu, K., Sasaki, A.: Hygienic Macro System for JavaScript and Its Light-weight Implementation Framework. In: ILC '14 (2014)
56. Warth, A., Piumarta, I.: OMeta: an object-oriented language for pattern matching. DLS '07 (2007)