An Optimized R5RS Macro Expander

Sean Reque

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

February 2013

ABSTRACT

An Optimized R5RS Macro Expander

Sean Reque
Department of Computer Science, BYU
Master of Science

Macro systems allow programmers abstractions over the syntax of a programming language. This gives the programmer some of the same power posessed by a programming language designer, namely, the ability to extend the programming language to fit the needs of the programmer. The value of such systems has been demonstrated by their continued adoption in more languages and platforms. However, several barriers to widespread adoption of macro systems still exist.

The language Racket [6] defines a small core of primitive language constructs, including a powerful macro system, upon which all other features are built. Because of this design, many features of other programming languages can be implemented through libraries, keeping the core language simple without sacrificing power or flexibility. However, slow macro expansion remains a lingering problem in the language's primary implementation, and in fact macro expansion currently dominates compile times for Racket modules and programs. Besides the typical problems associated with slow compile times, such as slower testing feedback, increased mental disruption during the programming process, and unscalable build times for large projects, slow macro expansion carries its own unique problems, such as poorer performance for IDEs and other software analysis tools.

In order to improve macro expansion times for Racket, we implement an existing expansion algorithm for R5RS Scheme macros, which comprise a subset of Racket's macro system, and use that implementation to explore optimization opportunities. Our resulting expander appears to be the fastest implementation of a R5RS macro expander in a high-level language and performs several times faster than the existing C-based Racket implementation.

Keywords: Macros, R5RS, Optimizations, Expander

ACKNOWLEDGMENTS

Without the support and guidance provided by Dr. McCarthy, this work would not have been possible. I am also indebted to BYU for providing an excellent learning environment and culture. I last of all thank my family for being the teachers in the most important school of my life.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Statement

R5RS macro expansion can be optimized to enable the expansion of Scheme programs in a fraction of the time required by existing implementations.

To support this claim, we implement an optimized R5RS macro expander based on an existing algorithm. We also show our methods for verifying the correctness and performance of our expander, and provide our results from benchmarking our expander against other major Scheme implementations. We show that our macro expander significantly outperforms all other open source macro expander implementations we tested and is competitive with the fastest known macro expander, Chez Scheme's proprietary expander.

1.2 Scheme Macros

Macros systems as a programming language feature have been present in both mainstream and research languages for decades [20]. Macro systems are usually defined by the following:

1. An abstraction for syntax. This abstraction can be as simple as a sequence of characters or as complex as a disjoint union encoding of an abstract syntax tree.
2. An abstraction for defining macros, also known as syntax transformers, functions whose inputs and outputs are syntax objects.
3. A mechanism for invoking macros and making use of the resulting syntax in the compilation or execution process. Macro invocation is also referred to as macro

expansion, and to expand a macro means replace the reference to a macro with the results of invoking it on its syntax arguments.

Depending on the language, macros are most often used to implement functionality that is unique to the power of macros and cannot be implemented in the base language itself. Common uses of macros include:

- Guide compilation through external parameters, such as compile flags or configuration. One simple example is conditional compilation.
- Change the evaluation semantics of the language. This allows one to, for instance, define some form of lazy evaluation in an otherwise strict language without requiring native language support.
- Optimize the program by transforming syntax fragments into forms more easily compiled to efficient machine code.
- Generate syntax programmatically, such as boilerplate code that can't be abstracted over in the base syntax of the host language.
- Define domain-specific constructs and mini-languages, often referred to as DSLs or domain-specific languages. These languages are often more elegant, readable, and maintainable to domain experts than the syntax of the base language.

With the power and expressiveness of macro systems comes added complexity. Correctly manipulating language syntax can prove difficult even in the simplest cases. For this reason, most research in macros has occured historically in languages with simple grammars, most notably the Lisp family of programming languages. Also, as with any other tool for abstraction, debugging, which involves peeling away abstractions to analyze the underlying low-level details, can be made more difficult by macro systems, which may not provide means for tracing the original program syntax to its final syntax. Lastly, macros add an extra layer to the compilation process, complicating other syntax-processing tools such as code editors, IDEs, and quality analysis tools like LINT. In particular, the speed of compilation times and IDE refresh cycles can be slowed by complex macro systems that require significant computation time to complete expansion of code.

Racket, a Scheme-derived programming language, is designed with both a simple core set of language primitives and a powerful, flexible macro system as a foundation on which to extend the capabilities of the language. Language features that are primitive in other languages, such as object systems, exceptions, co-routines, lazy evaluation, and asynchronous code written in synchronous style, are all implemented as libraries on top of the core language, in part due to the power of its macro system. Perhaps because of its sophistication and power, the Racket macro system currently suffers from performance issues, and for some programs macro expansion accounts for the overwhelming majority of program compilation time.

By optimizing the speed of the macro expansion process in Racket, we aim to make macros more useful as a productive tool in writing software. Because of the size of Racket's macro system, we focus on a subset of the system defined by the R5RS standard, also known as **syntax-rules** macros. We show that our expander is both functionally correct and faster than the current Racket expander, as well as all most other major R5RS expander implementations.

1.3 R5RS Scheme

The word "Scheme" can currently refer to an entire family of programmming languages. Since the first description of a language called Scheme appeared in 1978, multiple universities created dialects of the language that diverged so far that researchers began to have difficulty reading code written using a different dialect. Therefore, in 1984 representatives from different universities met and published the first revised report on Scheme to provide a specification that implementations could unify on. Since then, 5 other revised reports have been been published, and each of these reports is referred to by their publishing order using the notation **RXRS**, where X is the number of the revision. For reasons explained later, this proposal focuses on R5RS Scheme [11], or the language specified by the 5th revised report on Scheme. The Racket language, while derived from Scheme, does not closely conform to any of the revised reports. It can be thought of as a superset of R6RS with many incompatible changes.

R5RS Scheme defines a minimal homoiconic S-expression based programming language with a simple but expressive and Turing-complete macro system. In this language, syntax is represented using the same primitive types available at runtime, namely lists, vectors, symbols, characters, strings, and numbers. Macros can be defined and associated with a globally- or lexically-scoped keyword. These macros can then be invoked using the same syntax for invoking regular procedures at runtime. A simple example macro is the **and** macro. This macro implements the short-circuit "and" operator commonly found in programming languages but generalized to an arbitrary number of arguments. To use the macro, one invokes it just as if it were an ordinary procedure, i.e (**and** (< *x* 5) (> *x* 0)).

To define an **and** macro globally for the rest of the program to use, one must place a **define-syntax** form at the top level of the program as follows:

```
(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...)
     (if test1 (and test2 ...) #f))))
```

This example binds the macro specified by the **syntax-rules** expression to the keyword "and" for the duration of the program. It showcases the **syntax-rules** system Scheme provides for defining macros. **syntax-rules** macros use patterns to match input syntax and templates to output the resulting syntax. In all **syntax-rules** forms, all arguments but the first are a list containing two values, representing a pattern and template respectively. When a **syntax-rules** macro is invoked, the macro matches the input syntax to the invocation against each pattern successively. The template corresponding to the first matching pattern is then used to create the output syntax for the invocation, and the output syntax is spliced into the program, replacing the syntax corresponding to the macro invocation. If no patterns match then the macro invocation results in a syntax error.

The above definition of **and** takes advantage of the fact that Scheme's **if** forms are actually expressions that return the value of the branch taken. In addition, as in most languages, an **if** form only evaluates the conditional expression and the appropriate branch expression, leaving the other branch expression un-evaluated. With these two features an **and** expression taking an arbitrary number of arguments can be re-written using **if** expressions. Lastly, in Scheme all values but false, written as #`f`, are considered true, written #`t`.

Specified in English, an invocation of **and** returns true if no arguments are provided. If one argument is provided, that argument is evaluated and its result returned. If more than one argument is provided, the first argument is evaluated. If it's value is not false, then the *macro* need only process the rest of the argument list recursively. The ... syntax allows a single identifier to bind to a variable-length list of values, so that the pattern (**and** *test1 test2* ...) matches a list beginning with the keyword "**and**" and containing one or more other values, the first of which is bound to the identifier *test1* and the rest of which are bound in a list to the identifier *test2*. Lastly, identifiers in a template expression whose names are identical to a pattern identifier in the corresponding pattern expression are replaced with the value that was bound to that identifier during the matching process.

Given the above information, one can see that the definition of the **and** macro above follows this English prose faithfully, matching its arguments and recursively rewriting the **and** expression into a series of nested **if** expressions. This example also showcases the implementation of a short-circuiting operator, an operator that cannot usually be implemented in a call-by-value language without both syntactic and runtime overhead.

To correctly expand a program, any Scheme program processor, including a compiler, interpreter, or syntax analysis tool, must maintain an environment mapping keywords to syntax transformers. Whenever an expression of the form (<*keyword*> *arguments* ...) is encountered, the environment must be consulted to see if the keyword corresponds to a

```
(and
 (> (+ 1 2) 2)
 (not (eqv? 'a 'b))
 (string-ci<? "a" "B"))
```

(a) A simple usage example.

```
(if
 (> (+ 1 2) 2)
 (and
  (not (eqv? 'a 'b))
  (string-ci<? "a" "B"))
 #f)
```

(b) The code after one expansion.

```
(if
  (> (+ 1 2) 2)
  (if
   (not (eqv? 'a 'b))
   (and (string-ci<? "a" "B"))
   #f)
  #f)
```

(c) The code after two expansions.

```
(if
  (> (+ 1 2) 2)
  (if
   (not (eqv? 'a 'b))
   (string-ci<? "a" "B")
   #f)
  #f)
```

(d) The code after three expansions.

Figure 1.1: An example showing the use of the ”and” macro with its subsequent expansion.

defined macro. If so, the corresponding transformer is invoked with the specified arguments, and the entire expression is replaced with the results of the transformer invocation. Then, processing of the program continues with the newly created syntax. In this way, macros can generate syntax that also invokes macros, potentially recursively.

Figures 1.1a through 1.1d illustrate an expander iteratively converting an invocation of **and** to its final form.

1.4 Problems with Conventional Macro Systems

Macros in traditional programming languages like C are often viewed as dangerous and unwieldy. Unless written carefully, macro definitions in such languages can contain latent errors that reveal themselves only when activated under certain conditions. To understand why, we will define several syntax constructs and explain how they can make macros dangerous.

- **Free variable**: any variable which, in a given block of code, is unbound.

- **Bound variable**: any variable, which, in a given block of code, is bound to a definition (binding occurrence).
- **Binding occurrence**: also known as a binding, an expression or statement that binds one or more variables to a particular denotation or location.
- **Capture**: a binding occurrence is said to capture a variable if that variable is bound by the occurrence, or in other words, the denotation of the variable is the denotation introduced by the occurence.
- **Binding scope**: the region of code under which a binding occurrence is effective and can capture variables.
- **Environment**: a dictionary mapping identifiers to denotations. Binding occurrences extend environments for the duration of their scope, while global binding occurrences extend an environment permanently. Some languages, including R5RS, conceptually have two environments for a program: the global, or top-level, environment, and the local, or lexical, environment.

In the context of the above definitions, traditional macros are dangerous because they can introduce new identifiers and binding occurrences, possibly mixing original program syntax into binding scopes that the user never intended. Macros must be written carefully as it is possible for users to unwittingly bind variables introduced by a macro, giving them new meaning and thus changing the overall meaning of the macro. Also, poorly constructed macros can mistakenly bind original program variables by placing them in binding scopes that the user never intended.

To summarize, macros look like regular procedure calls and we would like to treat them that way. In fact, macros for the most part never need to cause side effects or mutate state, so we would like to be able to treat them as pure functions. However, issues related to variable capture prevent us from doing so.

Referential Transparency and Hygiene

In order to allow macro users to transparently invoke a macro as if it were a pure function, macros satisfy two important properties that macro facilities of languages like Common Lisp and C do not: referential transparency and hygiene. These features help reduce the burden of the macro writer in producing error-free macros and enable macros to be used consistently and reliably in any part of a program. All standards-conforming implementations of R5RS Scheme provide macros that satisfy these two properties. Although they can be reasoned about separately, we will, after defining them, refer to both properties at once as "hygiene" unless otherwise noted.

Referential Transparency

The first property, referential transparency, means that the free variables in a macro expansion refer to bindings located in the lexical environment of the macro definition, not in the environment at the time of macro use. As such, a macro invocation, given the same input, will always result in the same expansion, regardless of the program location where the macro is invoked.

The following Common Lisp code illustrates macros that violate referential transparency.

```
(defun wrap (v) (list v))
(defmacro wrap-macro (v) `(wrap ,v))
(wrap-macro 1)
(flet ((wrap (v) (vector v))) (wrap-macro 1))
```

In the above snippet, the procedure *wrap* is defined that takes a single value and returns a list containing that value. Next, the macro **wrap-macro** is defined that accepts a value and generates syntax to invoke the *wrap* procedure on the value, such that this macro serves as a simple alias for *wrap*. The return value of the third line is the list (1). However, in the final line, the return value is the vector #(1).

This disparity in results can be explained as follows. The **flet** form is used to define a local version of the *wrap* procedure that wraps a single value inside of a vector. When the invocation of **wrap-macro** is expanded into the syntax (*wrap* 1), the identifier *wrap*, which was free in the definition of **wrap-macro**, is placed into a scope where it is bound by the locally-defined *wrap* procedure. The identier is bound to a denotation in the lexical environment at the site of the macro use, not at the site of the macro definition. This means that, given the same input, a macro transformer can produce different results, which is usually undesirable.

A C example that highlights a violation of referential transparency is a macro definition and use of the trigonometric secant function. Note that C uses the term "function" to refer to what Scheme and LISP refers to as a "procedure".

```
# define SECANT(rad) (1.0 / cos(rad))
  printf("%.3f\n", SECANT(1.0));
  double cos = 5;
  printf("%.3f\n", SECANT(1.0)); //fails to compile
```

In this example, at the point of definition of the macro SECANT, the identifier **cos** is free and therefore refers to the top-level binding of **cos**. If the math.h header is imported, then the top-level binding will be a function. After the first use of the **SECANT** macro, however, a local variable **cos** is defined that shadows the top-level binding. Therefore, the second invocation of **SECANT** generates an identifier that refers to the local binding of **cos** as a double and subsequently causes a compiler error.

Hygiene

A hygienic macro enforces the rule that a binding occurrence can only bind identifiers created in the same expansion step as the identifier used in the binding, where all original program syntax belongs in one expansion step that is unique from all others. One example of a non-hygienic macro is the following definition of the built-in **or** procedure as a macro named **macro-or** in Common Lisp. The **macro-or** macro works analogously to the Scheme **and** macro, described earlier, in that it returns the first argument that is considered “true”, or otherwise a “false” value, which is the keyword *nil* in Common Lisp.

```
(defmacro macro-or (&rest args)
  (if args
    `(let ((v ,(car args)))
    (if v v (macro-or ,@(cdr args))))
    nil))

(macro-or nil 2)
(let ((v 2)) (macro-or nil v))
```

In this example, **macro-or** inspects its arguments, stored as a list in the variable args. If the list is non-empty, it expands to an expression that binds the value of the first list argument to a variable v. It then places a reference to that binding as the first two arguments of the final if expression, with the third argument being a recursive invocation of **macro-or** on the remaining arguments.

Both invocations of **macro-or** following its definition should return the value 2. However, the second invocation returns the value nil. Performing one expansion step on the second invocation yields the following syntax:

```
(let ((v nil)) (if v v (macro-or v)))
```

From this one can see that the expansion introduced a binding of a variable v, and the identifier v provided by the macro invoker now references this local binding instead of the intended binding that stored the value 2.

Another interesting example in C is a basic swap macro that exchanges the values of two variables.

```
# define SWAP(a, b) {typeof(a) tmp = a; a = b; b = tmp; }
  int a = 1, b = 2, tmp = 3;
  SWAP(a, b);
  SWAP(tmp, a);
  //expect 3, 1, 2
  //actual 2, 1, 3
```

In the above example, the two uses of the **SWAP** macro expand to the following:

```
{typeof(a) tmp = a; a = b; b = tmp; }
{typeof(tmp) tmp = tmp; tmp = a; a = tmp; }
```

While the first use of **SWAP** works as expected, the second does not. In the first statement of the second expanded block, a new variable **tmp** is created that is assigned the value of the **tmp** variable in the outer scope, or 3. In the second statement, this new **tmp** variable is assigned the value of **a**, or 2. Finally, **a** is assigned the value of the new **tmp** variable, which had just been set to 2. Because of this, instead of swapping the values of **tmp** and **a**, the **tmp** created by the user never gets assigned a new value and **a** is only ever assigned its original value. All of this occurs because the **SWAP** macro introduces a binding occurence for **tmp** that captures uses of the identifier **tmp** supplied by the user.

1.5 Related Work

Hygienic macro systems are safer to use because they avoid accidental variable capture between macro-introduced syntax and user-introduced syntax. However, they complicate the implementation of the macro system, and they must also perform significantly extra bookkeeping to maintain hygiene. This results in slower implementations and increased compilation times. We present here an overview of the past research effort dedicated to designing hygienic macro systems, along with efficient algorithms for their implementation.

1.5.1 Gensym

Common Lisp's macro system is unhygienic. However, the language provides one construct to assist in writing hygienic macros in an ad-hoc fashion, the *gensym* operator, which generates a unique identifier. With this operator, a macro writer can write a hygienic macro if the macro generates syntax that only binds identifiers that were created using gensym during the expansion of the syntax. Such macros, however, are still not referentially transparent, so that care must be taken by the user when defining lexically-scoped macros with *macrolet* that generate free identifiers referring to local bindings. In C, because there is no gensym facility, users often create and use identifiers with long prefixes or suffixes that are hoped to be globally unique, which, while cumbersome, has a similar effect.

1.5.2 Timestamp Expander

Kohlbecker et. al. proposed a change to naive macro expansion algorithms to satisfy the hygiene condition [16]. This algorithm works by transforming syntax into a new universe where every identifier is associated with a timestamp. The timestamps are then used to distinguish identifiers from different expansion steps. The algorithm works as follows:

1. The expander maintains a global timestamp beginning at zero, and begins by stamping every identifier in the original program with this value.
2. At the end of each expansion step the expander increments the timestamp and afterwards traverses the output syntax of the expansion, stamping every freshly-generated identifier with the new timestamp. All identifiers generated outside the expansion step are guaranteed to already be stamped.
3. Once finished, the expander brings the resulting program back into the original syntax universe by unstamping each identifier. The unstamp process renames every bound identifier with α-conversions so that the resulting syntax has the same meaning it would have if the timestamps were retained. Unbound identifiers are merely unstamped and retain their original symbol.

The timestamp expander guarantees that macros cannot create binding occurrences that bind identifiers from another expansion step. The algorithm does not, however solve referential transparency because it is still possible for local bindings at a macro use site to unintentionally bind free identifiers generated by a macro. Also, their algorithm exhibits $O(n^2)$ complexity relative to an unhygienic expander.

1.5.3 Syntactic Closures

Bawden and Rees proposed a low-level macro facility for writing hygienic macros [1]. This system requires the syntax trasnformer to explicitly associate its output syntax elements with their syntactic environments rather than using the environment at the macro use site. The system also allows to provide a list of symbols that should be dynamically scoped in order to allow controlled breaking of hygiene. Syntactic closures also compose and support a quasiquote syntax for embedding syntactic closures inside other syntactic closures.

Syntax transformers are functions that receive the use-site environment and the use-site syntax arguments as a regular S-expression. If the transformer wishes to use elements of the input expression in its output expression, it can tease apart the input syntax as normal and bind the desired elements to the use-site environment. It can then embed these elements in its final expression, which may have a different syntactic environment.

This system is amenable to $O(n)$ implementations, allows for writing hygienic macros by virtue of requiring all binding environments to be made explicit by the macro writer, and supports controlled hygiene-breaking in like matter. For similar reasons, the system is also unwieldly compared to unhygienic solutions and requires exposing low-level details about syntactic environments to the macro writer. More importantly, the system cannot be used to implement a high-level, automatically hygienic macro system like R5RS's pattern/template system because such a system cannot determine what environment to close identifiers under until macro expansion has already been completed. Any such algorithm must provide a way to defer the judgement of deciding the denotation for an identifier until expansion has been completed for that identifier.

1.5.4 Explicit Renaming

Clinger and Rees built on the timestamp expander and syntactic closures to design an algorithm specifically to implement R5RS-style macros efficiently [3]. At a low-level, the system associates each syntax transformer with its definition-site syntactic environment.

During expansion, the algorithm explicitly renames all identifiers introduced during expansion. Importantly, it also extends the use-site syntactic environment with bindings for each newly introduced identifier such that the denotation of the new identifier is identical to the denotation of the identifier it originated from.

The explicit renaming and use-site binding serves two purposes. First, it enforces the property that all macro-introduced bindings refer to denotations from the definition-site environment. Second, it is impossible for binding occurrences, either in the use-site lexical scope or introduced by the macro itself, to interfere with one another. The explicit renaming algorithm is therefore fully hygienic, is referentially transparent, and, as with syntactic closures, always runs in linear time relative to the size of the input program.

The algorithm's primary weakness compared to unhygienic macros is that it does not support an automatically hygienic system that also allows for the full use of the runtime capabilities of the host language. Unhygienic Lisp macros and syntactic closures, by contrast, allow a macro writer to program macros in the same host language and with the same libraries that are used to write programs themselves. Clinger and Rees later discovered through implementing their algorithm that they could easily support a low-level unhygienic system called explicit renaming.

1.5.5 Mark/Antimark

Dybvig and Bruggeman presented a macro system that also ran in $O(n)$ time and added the ability to program macros in the host language without losing automatic hygiene or controlled hygiene-breaking [10]. This algorithm creates a new abstraction for identifiers, like the timestamp algorithm, but unlike the timestamp expander, this abstraction is the primary abstraction used for identifiers rather than the temporary one. To implement hygiene, an identifier is defined as a tuple of a symbolic name, a variable name, and a set of marks. During different parts of the macro expansion process, marks are added and removed from identifiers to differentiate between identifiers passed into an expansion and those generated by the expansion itself. Then, substitutions change the variable name of identifiers to fresh symbols based on their mark sets. The variable name is what is eventually used to maintain referential transparency and hygiene.

Dybvig and Bruggeman also take advantage of redefining the representation of an identifier to solve another problem with macro systems, source-object correlation. When reporting errors in a stack trace or highlighting syntax in an IDE, it is necessary to be able to associate each piece of syntax with the original program syntax responsible for creating it. The mark/antimark algorithm makes it trivial to associate source location information with syntax as it is expanded by annotating expressions with source location information during the expansion process.

The mark/antimark algorithm is the basis of the current implementation of R5RS and R6RS macros in Racket, Chez, Ikarus, and other conforming Scheme implementations.

1.5.6 Fast Imperative Expander

When the R6RS macro system was being developed and standardized, the mark/antimark algorithm provided the initial reference implementation. Later, Van Tonder provided an alternative, though unpublished, implementation inspired by the explicit renaming system devised by Clinger and Rees [22]. This algorithm foregoes defining a new syntax representation, instead eagerly renaming fresh syntax as it is generated.

1.5.7 MetaML and MacroML

MetaML [21] is a programming language designed to generalize macros into a completely type-safe multi-stage execution environment, meaning that sub-programs can be generated, typed, and executed even at runtime. MetaML's primary focus lies in creating a stage- and type-safe system; if a metaprogram is well-typed, then the programs it generates are guaranteed to be well-typed. This, also means, however, that metaprogramming is limited to the capabilities of the type system and cannot use the full expressivity of the runtime language in defining macros.

MacroML [7] builds on MetaML by defining a macro system whose semantics can be reduced to MetaML semantics. The macro system allows for typesafe, hygienic, generative macros and allows for the generation of locally hygienic binding constructs by encoding the binding occurrences a macro creates, as well as the free variables an expression needs to have bound before it can be used. This technique suffers from the limitation that the type system must be able to infer this information directly, which in practice means that MacroML only supports new binding constructs that have the same shape as the existing binding constructs in the language. As another serious limitation, MacroML only supports generative macros. It does not support arbitrary decomposition syntax and rearrangement of syntax, as the type system would be unable to track binding dependency information needed to type the macros properly.

1.5.8 Template Haskell

Template Haskell [17] also implements a hygienic macro system, employing its type system in novel ways to accomplish hygiene. At its lowest level, Template Haskell exposes a low-level unhygienic API by representing Haskell syntax as a strongly-typed AST through Haskell data records. On top of this system Template Haskell implements Hygiene via a monadic library, the quotation monad. This monad encapsulates the generation of fresh names and contains constructors for each low-level data type that lifts the corresponding data type to the quotation monad. On top of this, Haskell provides a splice operator to allow one to splice binding occurences into the top level that become visible to the rest of the program, and a quasiquote operator to make syntax generation feel more natural. Template Haskell foregoes the full flexibility that Scheme macros offer in defining new binding occurences in order to feasibly statically type the inputs and outputs of macro invocations.

1.5.9 Nemerle

The Nemerle programming language [19] was designed to fuse imperative, object-oriented concepts prevalent on the .NET CLR with several functional concepts, including automatically hygienic, high-level and low-level macro facilities. It is one of the most recently designed languages containing macro systems, and its algorithm borrows heavily from the algorithms used in Scheme macro systems.

The algorithm follows the timestamp algorithm in every way but two. First, free identifiers in a Nemerle expression generated by macro expansion refer to the definition-site environment stored with the macro to implement referential transparency. Second, it uses the abstraction of identifiers as a tuple of symbols and timestamp integers permanently, so that it doesn't need to parse the output of every macro expansion to search for unstamped identifiers. Instead, whenever a macro constructs new syntax, it must associate a timestamp with that syntax immediately. This timestamp can be the global timestamp or the timestamp of another identifier for controlled hygiene-breaking. This means that the algorithm can run in $O(n)$ time. Nemerle's hygienic macro system provides similar flexibility and power to Scheme macros but in a statically typed language. Nemerle does not, however, support an arbitrarily number of phases of execution per compilation, a requirement for R6RS macros. Also, Nemerle macros must be compiled separately before they can be used. Overall, the algorithm appears to be sufficient to serve as a potentially viable alternative for an optimized R5RS expander.

1.6 Optimization Approach

Our strategy to implement an optimized expander is as follows

1. Implement a full-fledged expander and a benchmark suite.
2. Use manual profiling techniques to develop an intuition for possible performance bottlenecks.

3. Use the results of manual profiling to develop and test optimizations.

We wished to use a statistical or instrumenting profiler to automate the collection of performance data for our expander. Unfortunately, we attempted to use the available Racket profiler with little success. The profiler as written cannot measure time spent in native code, and we found that our expander spent most of its time executing the native code that implements many core standard library functions. This meant that, in many cases, the profiler output held little or no value and we were forced to rely on intuition alone in determining possible performance improvements. Our attempts to use native statistical and instrumenting profilers also failed to yield useful profiling data.

1.7 Summary

Thus far we have defined macro systems, explained their value in software development, and showed some of the pitfalls of existing macro systems that limit their widespread adoption. We have also shown how the R5RS macro system addresses some of these issues through hygiene and referential transparency, though at the cost of an increased performance burden on macro expanders. Lastly, we have highlighted the previous body of work dedicated to developing and improving the performance of hygienic macros systems, including that of R5RS Scheme.

Our contribution is to show that existing macro algorithms can be optimized to implement macro expanders that run much more performantly than existing implementations. We do so by implementing a new expander using an existing algorithm, explicit renaming, and then showing how the application of several optimizations to our expander can increase its performance to surpass other existing implementations.

Chapter 2

Explicit Renaming Algorithm

We use the explicit renaming algorithm as the basis of our new macro expander implementation. This algorithm was designed specifically to implement R5RS, or **syntax-rules**, macros efficiently. Its advantage is that it is simpler to implement than other more advanced macro algorithms that are also capable of implementing the more general R6RS macro system. Choosing this algorithm also allowed us to explore possible optimizations that are specific to the R5RS specification but that do not generalize to R6RS. Since many fundamental macros in newer versions of Scheme are still written using a **syntax-rules** system similar or identical to that defined by R5RS, optimizing R5RS **syntax-rules** macros can yield performance benefits for these Scheme implementations as well.

2.1 Algorithm Overview

The explicit renaming algorithm has at its foundation the concepts of identifiers, denotations, and environments. As defined earlier, an environment maps an identifier to its meaning, or denotation. Environments are conceptually immutable; they can be extended or reduced to create new environments, but never modified in place. For our purposes, we do not use denotations to distinguish between lexically shadowed variables, but only to distinguish between identifiers introduced at different macro expansion steps.

Within a given scope, an identifier is unbound if there is no entry in the environment that maps the identifier to a denotation. Whenever a binding occurrence is encountered, the environment is updated with a mapping of the bound identifier to a fresh denotation for the

Syntax	Environment

(**let** ((*x* "outer"))
  (**let-syntax**
    ((*m* (**syntax-rules** () ((*m*) *x*))))
    (**let** ((*x* "inner"))
      (*m*))))
	()

(**let-syntax**
    ((*m* (**syntax-rules** () ((*m*) *x*))))
    (**let** ((*x* "inner"))
      (*m*)))
	(x → den1:”outer”)

(**let** ((*x* "inner")) (*m*))	(m → den2:*macro*, x → den1:“outer”)
(*m*)	(m → den2:*macro*, x → den3:“inner”)

x.3	(m → den2:*macro*,
	    x → den3:“inner”,
	    x.3 → den1:“outer”)

Figure 2.1: Explicit Renaming Example

scope of the binding. Whenever a macro is invoked, it is passed at least two arguments: the input syntax for the invocation and the environment at the invocation site. The macro itself outputs both the resulting syntax and a new environment. The new environment consists of the old environment updated with mappings for each fresh identifier generated by the macro. The denotation for each freshly-generated identifier is the denotation of the parent identifier in the macro definition relative to the environment of the macro definition site.

The explicit renaming algorithm distinguishes between symbols introduced at different expansion steps by renaming macro-generated symbols as soon as they are introduced into a program. When introduced, the identifiers share the same denotation with their parent identifiers as expected, but they have separate entries in the environment table. A binding can only update one environment entry, which means that a binding can never capture identifiers introduced at different expansion steps. This satisifies both hygiene and referential transparency.

Without yet delving into other relevant details, we give figure 2.1 to show how explicit renaming interacts with the step-by-step parsing and macro expansion of a Scheme expression to maintain hygiene. This example is taken from the original paper [3]. The steps shown in the figure are as follows:

1. At the beginning, we have a single expression and an empty lexical environment.
2. The top-level **let** expression introduced one new binding for the identifier x. The environment is updated with a mapping of this identifier to a fresh denotation, which stores the value `"outer"`.
3. The **let-syntax** expression defines a single lexically-scoped macro, m. The identifier is bound to a new denotation which stores the macro itself.
4. The **let** expression provide a new binding for x, and the environment is updated accordingly.
5. The macro corresponding to the identifier m is invoked. The macro specifies to output a single identifier, x. Instead, a fresh identifier, *x.3*, is output. Its denotation is the denotation of x in the environment in which the macro m was defined, which is *den1*. The value of this variable when evaluated is therefore `"outer"`.

2.2 R5RS *syntax-rules*

2.2.1 *syntax-rules* Grammar

Table 2.1 gives a slightly simplified grammar for **syntax-rules** definitions, which serve as the only way to define macros in R5RS Scheme. At its core, a **syntax-rules** definition consists of a list of pattern/template pairs (along with a literals list that we ignore for our discussion). A pattern can be one of the following:

- an identifier
- a proper list of patterns

<transformer spec> -> (syntax-rules (<identifier>*) <syntax rule>*)
<syntax rule> -> (<pattern> <template>)
<pattern> -> <pattern identifier> | (<pattern>*) | (<pattern>+ . <pattern>) | (<pattern>* <pattern> <ellipsis>) | <pattern datum>
<template> -> <pattern identifier> | (<template element>*) | (<template element>+ . <template>) | <template datum>
<template element> -> <template> | <template> <ellipsis>
<pattern identifier> -> <any identifier except '...'>
<ellipsis> -> <the identifier '...'>

Table 2.1: Simplified R5RS *syntax-rules* Grammar

- an improper list of patterns
- an ellipsis list, which is a proper list containing a sequence of zero or more patterns followed by a pattern and an ellipsis.
- a datum (string, number, character, or symbol)

A template is also defined recursively. One of the major differences in its grammar is its handling of ellipses. The possible types of templates are given below:

- an identifier
- a proper list of templates
- an improper list of templates (the tail template cannot be applied to an ellipsis)
- a datum (string, number, character, or symbol)
- a template applied to one or more ellipses

2.2.2 Interpreting *syntax-rules* Definitions

The grammar given in the previous section defines the syntax for **syntax-rules** definitions. In this section we explain how to interpret their semantic meaning. A **syntax-rules** definition specifies a program or function that accepts a list of syntax and an environment as input arguments and outputs either a match error or else a piece of syntax and its associated environment. In turn, each pattern and template defines a sub-program or sub-function. A pattern function also accepts syntax and an environment as input. It outputs either a match error or a pattern environment. A template in turn accepts an environment as input and outputs a new syntax/environment pair. A **syntax-rules** interpreter could execute a definition as follows:

1. Match the input syntax against each pattern sequentially until one of the patterns matches the input syntax. If none match, give up and output an error.
2. Create fresh bindings for each template identifier in the corresponding template that does not refer to a pattern identifier.
3. Create a new environment that merges the pattern environment returned from the pattern match with the regular identifier environment created in step 2.
4. Use the template and environment from step 3 to generate the output syntax.
5. Create an output environment that extends the input syntax environment with the new fresh identifier bindings created in step 2.
6. Return the output syntax from step 4 with the environment from step 5.

At a high level, all that remains is to define how each pattern and template specify a sub-program for matching or outputting syntax. Patterns, as functions that accept an input environment and output an error or pattern environment, define sub-programs as follows:

- a pattern identifier returns an environment that maps the identifier to the input syntax.

- a literal datum outputs an empty environment.
- a proper list of patterns returns an input error if the input list is not a proper list of the same length. It matches each sub-pattern against each sub-element of the input syntax. It returns an error if any sub-matches fail. Otherwise, it returns an environment that is the union of all environments returned by the sub-matches.
- an improper list of patterns operates similarly to a proper list. It requires the input syntax to have the same shape and similarly merges it sub-match results.
- an ellipsis list containing n sub-patterns can match an input syntax list if it is at minimum of length $n-1$. The input syntax matches only if its first $n-1$ syntax elements match the first $n-1$ patterns in the ellipsis list. All remaining elements of the input syntax are matched against the final pattern in the ellipsis list. If all matches succeed, the output environment contains all the merged environments for the first $n-1$ sub-patterns. All ellipsis pattern-match results are also merged into a single environment mapping identifiers to lists of values, and this environment is merged into the final result.

Templates define the sub-programs that take environments as input and output new syntax forms as explained below:

- an identifier looks up its value from the input environment. The value will either come from a corresponding pattern identifier, or else an identifier freshly-generated for this expansion step.
- a datum ouputs itself as syntax.
- a template applied to one or more ellipses returns a syntax list based on the length of the values in the pattern environment corresponding to the identifiers contained in the template. The identifiers in the template must have corresponding pattern identifiers that were also applied to ellipses.

- a proper list of templates outputs a proper list of syntax values corresponding to the sub-programs specified by its sub-templates.
- an improper list of templates behaves similarly to a proper list but outputs an improper syntax list.

2.2.3 Implementation Concerns

Several other issues must also be addressed when designing an implementation for explicit renaming.

1. A macro expander must parse and track all binding information in a program. In fact, the parsing and expansion of a Scheme program are interleaved and difficult to separate. A full macro expander implementation must therefore also include a Scheme syntax parser.
2. Most R5RS implementations implement a more advanced form of ellipses handling in templates than is specified in the standard. The extensions to the R5RS in terms of ellipses in templates were eventually codified in the R6RS standard. We decided to follow the lead of other R5RS implementations in providing them. In summary, the extended rules allow more ellipses to apply to a template identifier than the number applied to a pattern, as long as there is at least one "anchor" identifier in the template that makes unambiguous how much to replicate the template.
3. The **quote** form complicates the explicit renaming algorithm. This form converts its single argument syntax into a runtime value. In order for the form to work properly, no symbol inside a quote form can be renamed. However, an expander cannot know for certain beforehand whether a given symbol denotes the **quote** form. Our implementation handles this fact by tracking the original identifier of every renamed identifier. Whenever a **quote** form is recognized, the expander undoes all renaming of symbols inside the form.

```
 ; macro definition

(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))))

 ; macro invocation

(let
  ((a 1) (b 2))
  (display "adding two numbers\\n")
  (+ a b))
```

Figure 2.2: Simplified let definition and example use

2.3 Example Macro Expansion

To illustrate the execution of the explicit renaming algorithm, we provide in figure 2.2 an example definition and use of a simplified version of the **let** macro. The **let** macro as defined contains one pattern-syntax pair. The pattern itself is an ellipsis-terminated pattern list containing two sub-patterns, the pattern ellipsis list ((*name val*) ...) and the pattern identifier *body1*, followed by the tail ellipsis pattern, the pattern identifier *body2*. The nested ellipsis pattern itself contains one sub-pattern, which is a pattern list with two pattern identifiers as sub-patterns. One representation of the parsed syntax tree can be viewed as follows:

```
(ellipsis-plist
  ((ellipsis-plist
     () ;contains zero sub-patterns before the tail pattern
     (plist ((pattern-id name) (pattern-id val))))
   (pattern-id body1))
  (pattern-id body2))
```

The invocation of **let** causes its argument syntax to be packaged as a list and matched against the pattern as shown above. The match algorithm will first verify that the input syntax list contains at least two elements. Then it will match the first two syntax elements against the first two sub-patterns in the ellpsis list. Lastly, it will match the tail pattern against the remaining syntax elements. The top-level sub-patterns, the syntax elements that each pattern matches against, and the resulting environments are given in table 2.2.

Pattern	**Matched Syntax**	**Output Environment**
((name val) ...)	((a 1) (b 2))	(name → '(a b), val → '(1 2))
body1	(display "adding two numbers\n")	(body1 → (display "adding two numbers\n"))
body2 ...	((+ 1 2))	(body2 → ((+ 1 2))

Table 2.2: The Top-level Sub-Patterns, Matched Syntax, and Output Environments

The final pattern environment output by the matcher function is:

```
(
  name → '(a b),
  'val → '(1 2),
  'body1 → '(display "adding two numbers\n"),
  'body2 → '((+ a b)))
```

The template corresponding to our pattern is at the top level a template list. Note that there is no concept of an ellipsis-terminated list in our model of templates, as ellipses can appear anywhere in a template list, not just at the end as with patterns. Our top-level list contains exactly two sub-templates, one a template list and the other an ellipsis template with one ellipsis applied to it and a pattern identifier as its sub-template. The first nested template list contains a regular identifier **lambda**, a template list (*name* ...), a pattern identifier *body1*, and an ellipsis template *body2* .... The conversion of the template syntax to our template model can continue to be applied recursively to the remaining sub-template. We show a parsed representation below.

```
(tlist
  ((tlist
    ((regular-id lambda)
     (tlist
      ((ellipsis-template (pattern-id name) 1)))
     (pattern-id body1)
     (ellipsis-template (pattern-id body2) 1)))
   (ellipsis-template (pattern-id val) 1)))
```

Before rewriting, the expander takes the use environment for our macro invocation and augments it with the pattern environment returned by the match procedure. Following that it augments the environment with an environment mapping the regular identifier **lambda** to a fresh identifier, which for our example, is *lambda.1*. The final environment used for syntax rewriting is as follows:

```
(
  'name → '(a b)
  'val → '(1 2)
  'body1 → '(display "adding two numbers\\n")
  'body2 → '((+ a b))
  'lambda.1 → 'lambda)
```

The expander accepts the passed-in environment and constructs a syntax list by invoking each of its two sub-templates with the same environment. The top-level template itself is composed of two sub-templates: one that creates a lambda expression and one that outputs the arguments to be applied to it. Table 2.3 shows the sub-templates for the lambda template and the syntax they output. The final output syntax for the entire lambda expressions is: (*lambda.1* (*a b*) (*display* "adding two numbers\n") (+ *a b*)).

Template	**Output Syntax**
lambda	lambda.1
(name ...)	(a b)
body1	(display "adding two numbers\n")
body2 ...	((+ 1 2))

Table 2.3: Sub-Templates of lambda Template and Their Output Syntax

The second sub-template corresponds to the tail template of the top-level ellipsis template. It simply extracts the list of values from the environment keyed to the identifier *val* and splices them in after the lambda expression. The final output is therefore:

((*lambda.1* (*a b*) (*display* "adding two numbers\n") (+ *a b*)) 1 2)

The output environment for this syntax is simply the environment at the macro invocation site extended with the fresh regular identifier environment created for the output syntax.

2.4 Summary

The explicit renaming algorithm renames all generated identifiers during a macro expansion step, at the same time aliasing these identifiers to have the same denotations as their parent identifiers. Because identifiers from different macro expansion steps hold separate entries in the denotation environment, newly introduced bindings cannot inadvertently capture identifiers from different macro expansion steps. Explicit renaming does not interact well with the **quote** form. Extra book-keeping is required to undo identifier renaming inside of **quote** forms.

In the next chapter, we discuss our methods for verifying expander correctness and benchmarking expander performance. We then use our benchmarks to identify areas where our initial explicit renaming-based expander implementation performs well relative to existing implementations, as well as areas where it struggles. In the subsequent chapter, we give our optimizations to our expander, along the resulting performance gains.

Chapter 3

Correctness and Performance Verification

3.1 Correctness and Performance Tests

We verify the correctness and measure the performance of our expander with a suite of programs that showcase advance uses of Scheme macros. All of these programs share one important quality; they perform most or all of their computations as part of macro expansion, taking advantage of the fact that **syntax-rules** is Turing-complete. This property of is valuable for two reasons:

1. Our expander cannot execute Scheme code and we wish to avoid relying on an external Scheme interpreter to verify the correctness of our expander when possible.
2. We can write relatively small programs that perform non-trivial calculations at expansion time to stress test our implementation.

We use mostly the same set of programs for verifying both correctness and comparing the performance of our expander against existing expanders.

3.2 Testing Hardware

We performed all correctness and performance tests on a Dell Vostro 3750 Notebook with the following specifications:

- Windows 7 SP1 64-bit Operating System
- Intel I5-2410M 2.3 GHz Processor
- 4 GB DDR3 1.333 GHz RAM

3.3 Performance Goals

There are two primary use cases we wish to represent with our benchmarks. For the first, many Scheme source units make light use of macros, mostly by using macros provided by other libraries or by defining simple syntax extensions. Example of this pattern of usage can be found in our Schelog and monads benchmarks. For this type of use, an expander will spend the majority of its time parsing syntax forms and performing simple macro expansions. A fast macro expander must therefore be able to process Scheme syntax as quickly as possible. This processing includes handling **quote** forms, maintaining syntactic environments, handling binding occurrences, recognizing macro applications, and so on.

Performance for this type of code is important when compiling large code bases, where thousands or hundreds of thousands of lines of code may get processed at once. Furthermore, integrated development environments and advanced text editors, which often strive for near instantaneous edit-time feedback, must expand Scheme code to provide the kind of semantic analysis typically enjoyed by toolchains for other languages. A fast expander is therefore essential for such editors to respond in a timely manner to user edits of individual source units.

On the other hand, we categorize the other major use of Scheme macros to be cases where macro writers intend to perform non-trivial calculations during the compilation of a program. These uses, while less common, represent a valid and useful application of the macro systems. In these cases, an expander will spend less time parsing user syntax forms and more time matching syntax and generating syntax. A performant expander must therefore be able to execute all forms of syntax-rules definitions as quickly as possible.

Our suite of benchmarks captures both of these use cases. While we would like to have access to more large Scheme code bases, the monads and Schelog benchmarks, and to a lesser extent, the CK benchmark, consitute programs that stress the handling of binding and quote forms and the general parsing of Scheme syntax. The other benchmarks stress different permutations of pattern matching and syntax generation.

3.4 Test Programs

We provide descriptions for the programs we use to verify and benchmark our expander below.

3.4.1 CK

CK contains an implementation of a CK abstract machine that is used to implement mutually-recursive, higher-order applicative macros without resorting to CPS style [14]. As examples, the program implements the following functions as macros:

- **permute**: a function that, given a list of values, returns a list of all the permutations of the original list.
- **factorial**: computes factorials using a unary encoding of natural numbers.
- **delete-assoc**: A function that removes an element from an association list given the key of the element.

3.4.2 Dirty R5RS

Normally, if a macro wishes to capture identifiers in the input expression it must accept an identifier as an argument and then place that argument in a binding position with the remainder of the code inside that binding. However, R5RS macros also have the ability to arbitrarily parse and extract portions of input syntax. Using this knowledge, Al Petrofsky devised a system whereby a macro digs inside of its argument syntax for specific identifiers and places them in binding positions. With this technique, the macro invoker does not have to explicitly specify the identifier(s) it wishes bound and the macro appears to be unhygienic.

Oleg Kiselyov takes this behavior further to implement expansion time environments, associative data structures that can be operated on at macro expansion time [12]. The library is designed in such a way that user identifiers are overriden using the extraction pattern by Petrofsky.

This particular program tests unique uses of R5RS hygiene and the ability for macros to match arbitrary Scheme syntax.

3.4.3 Macro Lambda

The macro-lambda program implements an apply macro that mimics the ability to create anonymous macros and apply syntax arguments to those macros in much the same way as one would create and use anonymous procedures [14]. The library properly handles nested lambdas and variable shadowing as with normal procedures. The apply macro is itself a fairly complex macro-generating macro, with each invocation of apply generating seven local macro definitions. In addition, the macro must parse the entirety of the argument syntax to discover and apply the lambda forms. These properties make it a useful program for verification and benchmarking.

3.4.4 Prime Numbers and the Sieve of Eratosthenes

Kiselyov built a prototype for a compiler that can translate Scheme code into **syntax-rules** code [13]. He then used this compiler to translate a Scheme procedure which uses the sieve of Eratosthenes to determine whether a number is prime into **syntax-rules** definitions that can be executed at expansion time [14]. Since number types do not exist for **syntax-rules** macros they are unary-encoded using nested parenthesis.

The algorithm as implemented has $O(n^2)$ complexity based on the size of the input number, so it can be easily modified to require more CPU time. It also heavily stresses an expander's ability to parse and apply macro definitions. For instance, computing whether the number five is prime involves the creation of over 5000 local macros.

3.4.5 Fibonacci

The sieve of Eratosthenes mostly traverses a finite list repeatedly. To see if the results for Kiselyov's compiler might vary based on different algorithms, we enhanced the compiler to be able write a program to compute the nth Fibonacci number. We discovered that the performance results for this program are similar to, but not identical to those obtained by the sieve program.

3.4.6 Schelog

Dorai Sitarum developed a DSL library in Scheme for coding in the style of Prolog [18], providing a useful test that our expander works with real-world Scheme programs. Since Schelog operates at runtime we must make use of an interpreter to test our implementation with it. However, the library makes heavy use of macros, so that if we feed the output of expanding a Schelog program into an interpreter we can be confident that our expander is not generating bad syntax. This form of test, on the other hand, is insufficient to catch certain classes of errors such as an expander not expanding a program completely.

The library itself makes heavy use of literal identifier patterns, macros that mix argument syntax with new bindings, and patterns and templates with ellipses, including forms that require our implementation to go beyond the R5RS standard.

3.4.7 Monads

The monads benchmark is a small library [15] written by Edward Kmett and made available on github. It provides a DSL for writing monadic-style code in Scheme using Haskell's "do" notation. Since such a notation amounts to creating a new binding form, it is implemented using macros. This library does not provide any opportunity for testable compile-time computation. Instead, it is included to be used as a useful performance benchmark, representing a simple real-world example of Scheme macro use.

3.5 Benchmarked Implementations

To provide a comparison for our expander, We implemented benchmarkable macro expander applications using several other Scheme implementations. The other expander implementations are described below.

3.5.1 Gambit

Gambit [5] is an R5RS Scheme implementation designed for performance and portability. Gambit did not appear to provide an API for fully expanding an entire Scheme program, and on our system the Gambit *pp* and ##decompile forms did not work properly, so we used their *load* procedure, which both expands and evaluates a Scheme source file. Gambit's expander uses a version of the portable syntax expander, an implementation of the mark-antimark algorithm written to be able to run on any R5RS-compliant Scheme implementation. We used Gambit version 4.6.2.

3.5.2 Chicken

Chicken Scheme [24] includes both an interpreter and Scheme-to-C compiler for R5RS Scheme programs and focuses on strong FFI-based integration with the underlying native platform. Their expander uses an explicit-renaming system as of version 4.6.5.

3.5.3 Ikarus

Ikarus [8] is an incremental, mostly self-hosted R6RS Scheme compiler. Ikarus was the first public implementation of a large part of R6RS, whose macro system is a superset of R5RS. Ikarus uses another version of the portable syntax expander [9] designed by Waddell and Dybvig [23]. We used a source build of Ikarus with the following version: 0.0.4-rc1+ (revision 1870, build 2011-11-22).

3.5.4 Chez

Chez [4] Scheme is a proprietary implementation of R6RS Scheme. As such, its current expander implementation is unknown. Its runtime interpreter, Petite Chez, is available for free and is bulit from the same sources as the Chez compiler. We used version 8.4.

3.5.5 Racket

Racket [6] is a Scheme-derived programming language that provides both R5RS and R6RS-compliant interfaces as sub-languages. We used Racket's *expand-top-level-with-compile-time-evals* form to expand s-expressions. The Racket interpreter implements its macro expander using a C-based implementation of the mark-antimark algorithm. We used Racket version 5.2.1.

3.5.6 Fast Imperative Expander

As far as we are aware, no Scheme implementation uses the "fast imperative" algorithm for macro expansion. We benchmark its reference implementation using both Racket and Petite Chez.

3.5.7 Other expanders

We attempted to benchmark Bigloo but its expander had several major flaws that rendered it incapable of correctly expanding many of the benchmarks. Furthermore, the interpreter had a flaw that prevented its use on Windows with all versions we tried, including versions 3.6, 3.7, and 3.8.

Table 3.1 summarizes the information for all benchmarked expander implementations.

3.6 Initial Benchmark Results

Expander	Version	Standard	Algorithm
Optimized Explicit Renaming	N/A	R5RS	explicit renaming
Gambit	4.6.2	R5RS	mark-antimark
Chicken	4.6.5	R5RS	explicit renaming
Ikarus	0.0.4	R6RS	mark-antimark
Chez	8.4	R6RS	unknown
Racket	5.2.1	R6RS	mark-antimark
Fast Imperative Reference Implementation	N/A	R6RS	Fast Imperative

Table 3.1: Benchmark Programs

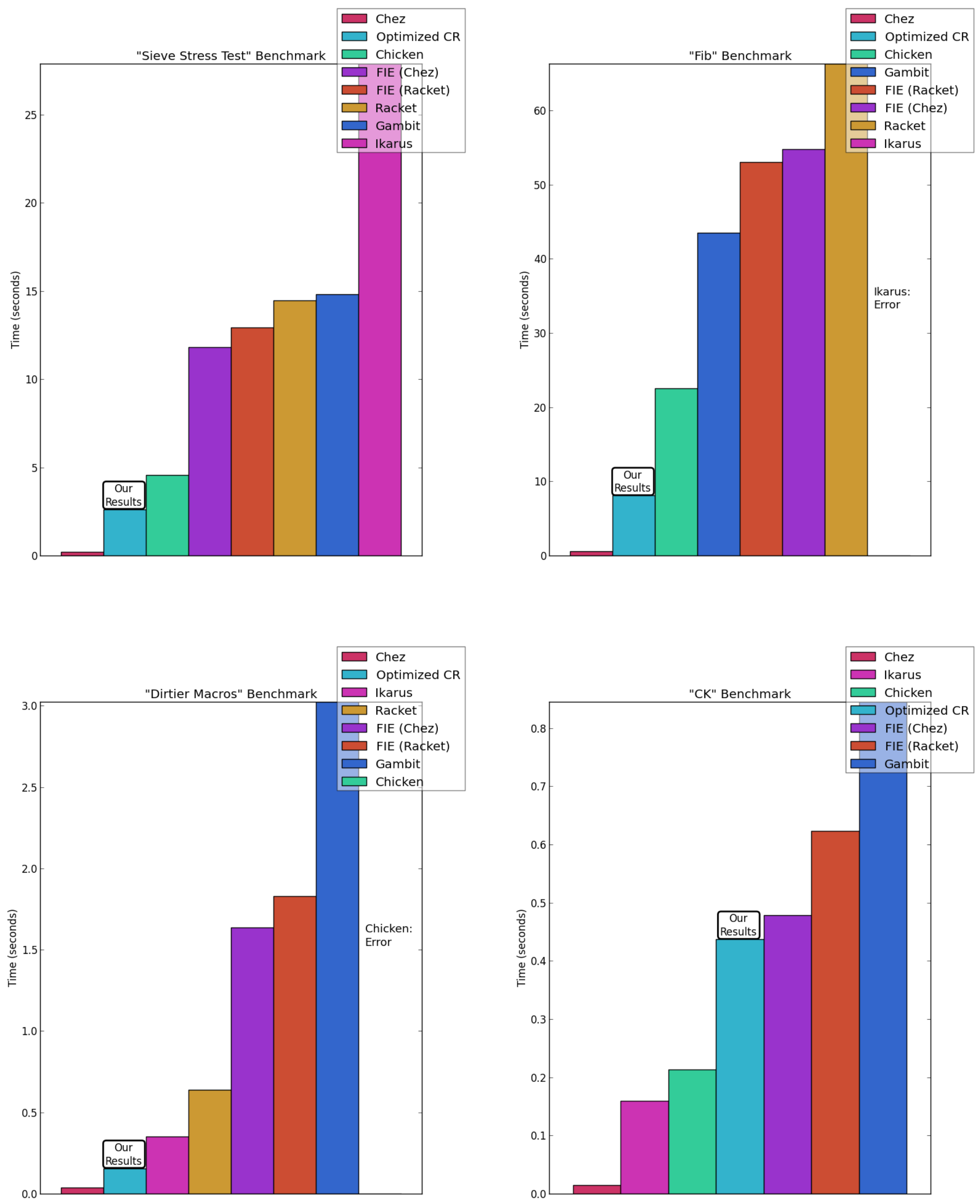

Figure 3.1: Initial "Heavyweight" Benchmarks

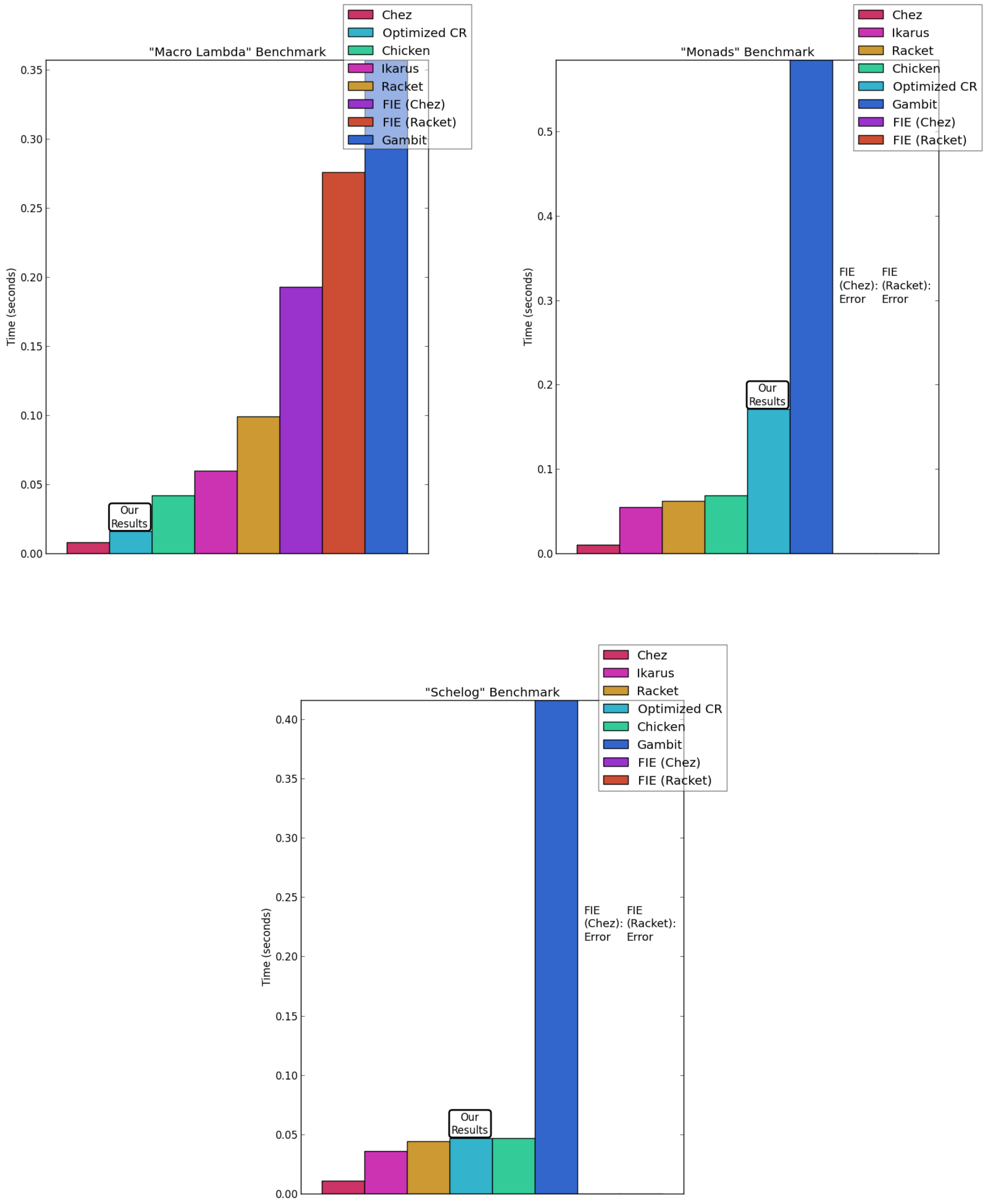

Figure 3.2: Initial "Lightweight" Benchmarks

Figures 3.1 and 3.2 show the benchmark results comparing all expanders against our initial implementation of an optimized explicit renaming-based expander. We label our expander as “Optimized CR”. This represents our first attempt at writing an expander, with correctness as our primary focus. The Chez expander performs the best by a large margin. For the benchmarks representing significant macro-level computations, which we have labeled as “heavyweight”, our expander already outperforms all other expanders. The next most performant expander in all but one of these is Chicken, which also uses explicit renaming. This provides some empirical evidence that the explicit renaming algorithm provides a good base for writing a fast R5RS expander.

The remaining benchmarks, which we have dubbed as “lightweight”, represent programs for which macro expansion does not necessarily dominate compile times. Instead, much of the time is spent, for instance, simply parsing Scheme expressions that are not macro invocations. In these benchmarks, our initial expander fares worse. Our expander gives a relatively mediocre performance for the CK and Schelog benchmarks and barely avoids last place on the monads benchmark.

3.7 Summary

We developed a suite of seven Scheme programs to use as input to our expander to verify its correctness. We used these same programs to create a benchmarking suite to measure the performance of our expander and identify potential performance bottlenecks. We set up and benchmarked several other expander implementations against this same suite to rank our expander against them.

Our results show that our initial expander already performs exceptionally well on benchmarks that heavily stress the expander by performing computations for algorithms such as the sieve of Eratosthenes. For other benchmarks that make less exotic use of our expander and whose parse times are less dominated by the expansion phase, our expander appears average at best. In the next chapter, we show how we improve our expander to significantly outperform all other open source Scheme macro expanders.

Chapter 4

Optimizations

Our initial benchmarking confirmed that we had implemented a functional R5RS expander with above-average performance. To show that R5RS macro expansion can be optimized to run in a fraction of the time required by existing implementations, we applied a series of optimizations that, when combined, improved the overall performance of our expander by almost an order of magnitude.

The optimizations we applied to our expander can be roughly grouped into four categories:

1. **Data structures**. This denotes either a change in the data structures used, or a change in their overall shape, along with any associated adjustments to the algorithm.
2. **Lazy Computation**. Avoid performing work until it is needed, especially if there is a chance that the work may not be needed at all.
3. **Removal of redundant computation**. Avoid performing redundant computations, including those related to error handling, constraint checking, or algorithm-specific computations.
4. **Static analysis of syntax-rules definitions**. Optimize macro expansion by analyzing and altering macro definitions prior to using them.

We discuss each of our optimizations in turn, organized into these categories.

4.1 Data Structure Optimizations

4.1.1 Pattern Matching Environment

Conceptually, a pattern matcher function, given a syntax element and its environment, returns either an error or an environment mapping pattern identifiers to matched syntax elements. We initially implemented this behavior as directly as possible using Racket's core data structures. Environments were represented as functional hashes and identifiers as Racket symbols. Given the observation that we know at **syntax-rules** definition-parsing time the exact number of pattern identifiers in a pattern, we can model our environment differently to improve performance.

When compiling a pattern into a procedure, we first map every unique pattern identifier in the pattern to an incrementing number starting at zero. We then redefine a match procedure to accept a third argument, a vector of length n, where n is the number of unique pattern identifiers in the pattern. In this way, each cell of the vector corresponds to a pattern identifier in the pattern. A match procedure returns only a value indicating whether the match was a success or failure. If the match was a success, then the vector is modified to include matched syntax elements.

The only source of complication comes from ellipsis patterns, since they match the same identifiers multiple times. For these patterns the vector cells corresponding to the pattern identifiers in the pattern hold lists of values, and the ellipsis matcher accumulates matched elements onto the lists.

4.1.2 Template Regular Identifier Environments

Once we converted our pattern environment representation into a vector, we modified the template procedure code, which originally expected a hash composed of both the pattern matches and the fresh identifier mappings created for all the regular identifiers in the template.

Instead, we modified it to take in two vectors, one representing the pattern identifiers and the other the fresh identifiers generated for the template's regular identifiers.

4.1.3 Removing the Quote Environment

The expander has to maintain a separate environment to support the **quote** form. Identifiers inside a quote form that were generated by a macro invocation must be restored to the value of the identifier in the unexpanded source program from which they originated. Each time a fresh identifier is generated, we therefore track its originating identifier in case it ends up inside a **quote** form.

We believed that merging the use environment and the quote environment into one would result in better performance due to the fact that a *cons* allocation should be less expensive than the allocations required to update a functional hash. To this end, we changed the definition of the use environment from a map of identifiers to denotations into a map of identifiers to a two-tuple of denotation and originating identifier.

4.2 Lazy Computation Optimizations

4.2.1 Lazy List Matching

Pattern lists represent trees of patterns, and our implementation compiles them into trees of procedures of the same shape as their original syntactic forms. Tree-shaped match procedures delegate to child procedures and then reduce the results of invoking these procedures into a single value.

Our initial implementation of compound patterns computed all sub-results for a pattern before merging the sub-results into a final result. However, it is a common occurrence that compound pattern matches fail in a non-fatal way. Most macros are written using multiple patterns/template pairs, and it is often the case that a macro invocation succeeds even though one or more of its matchers fail to match the input syntax.

One example of this is an R5RS macro implementation of the **letrec** form shown in figure 4.1. We note here that part of the complexity in implementing **letrec** is that the R5RS specification requires that all initial values be evaluated before any are assigned to the locations for the variables provided by the user. This requirement can only be met in standard Scheme by assigning these initial values to temporary locations, and the definition of **letrec** in figure 4.1 does so by creating a list of temporary variables to match the variables given by the user. The expressions for the initial values are first evaluated and assigned to these temporary variables. Once all expressions have been evaluated, they are then assigned to the locations corresponding to the macro invoker's variables.

This definition in figure 4.1 contains a recursive component that at each step extracts a value from one list and adds values to other lists; the base case occurs when the first list is finally empty. We show the two patterns corresponding to these recursive steps in figures 4.2 and 4.3.

When expanding a **letrec** form using the definition in 4.1, our macro expander prior to this optimization would match all sub-expressions in a syntax list before merging the results into a single value. For this macro, the top-level list in each pattern contains five sub-expressions, four of which are identical in both the base case pattern and recursive case pattern. Therefore, during each recursion step, our initial expander will perform several unneeded matches.

Given these observations, one can potentially reduce the time required to run compound match procedures by causing them to exit immediately as soon as they discover an error value returned from one of their child procedures. This avoids otherwise unnecessary comptuation, since when reducing child match results the error value always takes precedence.

```
(define-syntax letrec
  (syntax-rules ()
    ((letrec ((var1 init1) ...) body ...)
     (letrec "generate_temp_names"
               (var1 ...)
               ()
               ((var1 init1) ...)
               body ...))
    ((letrec "generate_temp_names"
               ()
               (temp1 ...)
               ((var1 init1) ...)
               body ...)
     (begin
       (define var1 #f) ...
       (define temp1 init1) ...
       (set! var1 temp1) ...
       body ...))
    ((letrec "generate_temp_names"
               (x y ...)
               (temp ...)
               ((var1 init1) ...)
               body ...)
     (letrec "generate_temp_names"
               (y ...)
               (newtemp temp ...)
               ((var1 init1) ...)
               body ...))))
```

Figure 4.1: R5RS letrec Definition

```
(letrec "generate_temp_names"
          (x y ...)
          (temp ...)
          ((var1 init1) ...)
          body ...)
```

Figure 4.2: Recursive rule of letrec

```
(letrec "generate_temp_names"
          ()
          (temp1 ...)
          ((var1 init1) ...)
          body ...)
```

Figure 4.3: Base case of letrec

4.2.2 No Match Failure Diagnostics

When a macro invocation fails, a user needs diagnostic information to determine the cause of the failure. We initially wrote our macro expander to report the reasons for match failures when invoking a macro. However, match failure in R5RS Scheme is an allowable and common occurence. A macro invocation only fails if all its patterns fail to match the input syntax. Therefore, the error diagnostic information calculated for each match failure wastes CPU cycles unless all matches fail.

We can avoid building unnecessary diagnostic information in one of two ways. The first is by passing a runtime value to the match procedure indicating whether we wish for diagnostic information on match failure. A second option is to use macros to create two versions of each match procedure, one that reports diagnostic information and one that doesn't. In either scenario, we invoke a macro once, and, if the invocation fails, we then perform extra work to report to the user the failure's cause. Since invocation failure leads to program termination it does not need to be optimized as with partial match failures; thus the performance penalty for gathering this information is paid rarely.

We chose to do away with diagnostic information entirely, since such information is not required to explore implementing an optimized expander. However, based on the implementation ideas given above, we believe the feature can be added with minimal overhead to the expander.

4.3 Redundant Computation Optimizations

4.3.1 Unsafe Operations

Because all variables in Racket's type system have the same type, runtime type-checking that could otherwise be omitted in languages with more powerful type systems is required for most operations to maintain memory safety. As an example, the implementation of the procedure *car* must verify at runtime that its argument is a *cons* object. User code often must also perform its own type-checking using procedures such as *car?* or *vector?*. When a piece of code performs specific operations on a value depending on its type, it's type checking combined with the type checking of built-in procedures leads to redundant and unnecessary overhead.

As of this writing, the Racket VM does not attempt to optimize away many of these redundant type checks. However, it does provide a *match* form to efficiently combine type checking a value with extracting values from that value based on its type. It also provides efficient iteration constructs that can outperform procedures like *map* and *foldl*. These constructs are the *for* forms combined with forms that designate the expected type of iterable values such as *in-list* and *in-vector*. Given these type hints, the *for* operators can often more efficiently iterate through data structures than procedures like *map* while maintaining memory safety.

Even with the optimized Racket forms, there are cases in our expander implementation where we know extra information about the types of values being stored in variables that the runtime isn't aware of. For instance, we may know that a variable should always hold proper lists, that two variables hold references to lists of the same length, or that one variable holds a valid integer index into a vector.

With this information in hand, we wish to avoid runtime memory-safety checking, including general type checking, list shape checking, and vector bounds checking. Fortunately, Racket provides unsafe operators corresponding to many memory-safe forms. These forms, if used improperly, could read from or write to invalid memory locations and result in program errors. On the other hand, if used correctly, these operators are more efficient than their memory-safe counterparts. Examples of these unsafe operators are:

- *unsafe-car* and *unsafe-cdr* for traversing lists
- *unsafe-vector** forms for referencing and setting vector cells and retrieving vector lengths
- *unsafe-fx* forms for performing unsafe arithmetic and comparisons on tagged integers.

```
 ; This is not a valid R5RS syntax-rules definition

(define-syntax letrec-optimized
  (syntax-rules ()
    ((letrec ((var init) ...) body ...)
     ((lambda (var ...)
       ((lambda (tmp-var ...) (set! var tmp-var) ...) init ...)
       ((lambda () body ...))) 'undefined ...))))
```

Figure 4.4: Optimized letrec pseudocode

4.3.2 Optimizing body expansion

Our expander rewrites **define** forms at the top of *body* forms using **letrec** as shown in the R5RS specification. Our expander also in general uses **syntax-rules** definitions of syntax forms whenever possible, and **letrec** is implementable as an R5RS macro. The accepted R5RS macro definition of **letrec**, while functional, is less efficient than a lower level implementation. A lower level implementation can more easily rewrite the **letrec** form into forms that do not require further expansion, which in, the case of our expander, is the **lambda** expression. Such an implementation may also take advantage of full access to the host programming language to perform optimizations not possible in the language of **syntax-rules**.

To optimize our implementation of **letrec**, we therefore devised a low-level macro system and used it to implement a more performant version of **letrec**. We discovered after the fact that this system is very similar to the one devised by Clinger for explicit renaming [2]. In our system, one defines a macro as a regular Racket procedure that accepts a syntax list and a use environment as its arguments. It in turn returns new syntax and a new use environment for the syntax.

Figure 4.4 gives a pseudocode definition for our optimized letrec implementation in a syntax that is identical to R5RS **syntax-rules**. The reasons this definition will not compile are as follows:

1. Our definition contains an identifier, *tmp-var*, to denote a list of unique identifiers that is the same length as the list denoted by the pattern identifier *var*. We have not defined the creation of this value explicitly. A pure R5RS **syntax-rules** implementation must create this temporarily list through relatively costly recursive macro invocations.

2. We apply an ellipsis to the symbol 'undefined in the last line of our definition, which would result in a syntax error due to its ambiguity. In this case, we wish to produce a list of the same length as the list denoted by *var* that contains the repeated value 'undefined, which would again be costlier in a pure R5RS **syntax-rules** implementation.

Our actual implementation follows our pseudocode directly, but using our lower-level macro system. In this system, one must be explicitly hygienic and explicitly referentially transparent. One does so by only using identifiers that are either provided by the argument syntax or else are freshly generated. For instance, instead of outputting the **lambda** identifier directly, we generate a fresh identifier on each invocation. Our output environment will contain a mapping from this freshly-generated identifier to the denotation of **lambda**, which means that this freshly-generated identifier will be guaranteed to always denote **lambda** and never be captured inadvertently to have another meaning.

As a final note to our implementation, to create the list of identifiers for *tmp-var* we maintain a cache of symbols that we reuse between **letrec** invocations to avoid calls to *gensym* when possible.

In addition to improving **letrec**, we also made another simple but effective change. If a body expression contains no **define** form, there is no need to rewrite it into a **letrec** form. We can instead use the unmodified body expression.

4.3.3 Template List Rewriting

During the rewriting phase, a template list procedure must fuse together the results of its sub-template procedures. To do so, it maintains a list of connector or fuser functions corresponding to the list of sub-rewriter procedures, where each element in the list contains a

```
(define (unsafe-map f _xs final-value)
  (let loop ([xs _xs])
    (if (null? xs)
        final-value
        (cons (f (unsafe-car xs)) (loop (unsafe-cdr xs))))))

(define (unsafe-foldr proc init _xs)
  (let loop ([xs _xs])
    (if (null? xs)
        init
        (proc (unsafe-car xs) (loop (unsafe-cdr xs))))))
```

Figure 4.5: Specialized Definitions of map and foldr

reference to either the *cons* or the *append* procedure. The values in the list track whether the corresponding sub-procedure is for an ellipsis template, whose resulting list value must be spliced, or any other template, whose resulting value is not spliced.

For template lists that contain no ellipsis templates, the fuser list is unnecessary. We can simply use a Racket *for/list* operator or a *map* procedure. Since this branch of code is so common, we decided to write an optimized *map* procedure that relies only on unsafe operations and is specialized to operate on exactly one list.

Often times a template list will contain an ellipsis template only at the end of the list. In this case, we can avoid creating and using a fuser list by writing our own *map* procedure that, instead of ending a list with the null value, ends the list with the result of an ellipsis template procedure. We can perform a similar optimization with improper template lists.

When writing our own *map* procedure, we do not replicate the built-in procedure's ability to handle arbitrary arity procedures and number of argument lists. We also skip type-checking that the input value is a proper list. We similarly write an optimized *unsafe-foldr* procedure, which our list fuser function still relies on for the common case where a template contains one or more ellipsis templates in a non-tail position. The definitions of both procedures are given in figure 4.5.

4.4 Static Analysis Optimizations

4.4.1 Constant Wrapping

In our expander, templates are represented in memory as tagged discriminated unions, as is common for representing heterogeneous tree-like data structures. We noticed that, for one benchmark, our expander was spending the majority of its time allocating and freeing memory. This benchmark represented numbers using a Peano encoding based on nested lists, where the value of a number was the depth of the list. The benchmark also included a macro-generating macro with peano-encoded numbers in the templates. Parsing such macros involved wrapping these nested lists in tagged structs recursively, requiring n memory allocations, where n is the depth of the list. To avoid this excessive memory pressure, we opted to avoid wrapping constant values, including string, number, and character literals and lists of the same.

4.4.2 Template Pre-expansion

Based on the structure of the template language, we can sometimes infer the use of identifiers using static analysis. For macros that generate identifiers which provably also refer to macros, we can attempt to pre-expand the template statically, constrained to what we can reason about the pattern identifiers that are in the macro definitions. The most major limiting factor comes from the fact that pattern identifiers can refer to arbitrary syntax forms. Ellipsis patterns and templates provide another source of variability that cannot be resolved statically.

Figure 4.6 provides an example of a possible template pre-expansion via static analysis using macro definitions taken from the monads benchmark. As shown in the figure, the **assert** macro can be used to test program invariants and print a warning to the console when invariants are not met. The **assert-eq** macro simply rewrites itself into a use of the **assert** macro. A static analyzer, upon encountering the definition of this macro, can verify that the regular identifier **assert** in the template will always refer to the top-level binding of **assert**.

```
(define-syntax assert
  (syntax-rules ()
    ((assert proposition)
      (when (not proposition)
        (begin
          (display "Assertion Failed: ")
          (display 'proposition)
          (newline))))))
```

(a) Definition of the assert macro

```
(define-syntax assert-eq
  (syntax-rules ()
    ((assert-eq x y)
      (assert (equal? x y)))))
```

(b) Definition of the assert-eq macro

```
(define-syntax assert-eq
  (syntax-rules ()
    ((assert-eq x y)
      (when (not (equal? x y))
        (begin
          (display "Assertion Failed: ")
          (display (equal? x y))
          (newline))))))
```

(c) assert-eq after static pre-expansion

Figure 4.6: An example of template pre-expansion using the assert-eq macro

If such an analyzer knew the definition of **assert**, it could rewrite **assert-eq** by inlining the body of the appropriate template in **assert**, as we have shown in the figure.

The analyzer is unfortunately assuming that the user will not rebind **assert** to another value later on, which means it is potentially generating an incorrect program. However, a programmer may decide that the improvement in macro expansion performance may justify this violation of Scheme semantics, especially if he or she knows that **assert** will never be overwritten. Another possible solution to avoid this violation is to create a system where two versions of a macro are maintained in memory, the original macro and the pre-expanded macro. The pre-expanded macro is also associated with all the top-level bindings it assumes exists. when a macro is invoked, these bindings could be compared to the current top-level bindings. If all necessary bindings are still present, then the pre-expanded macro implementation may be used safely; otherwise, the original macro could be selected to preserve full R5RS semantics.

Our prototype template pre-expander operates by generating a macro transformer as normal when encountering a macro definition. However, it wraps this transformer in another procedure. When invoked, this procedure performs static analysis on the original templates in the macro, pre-expanding them when possible, and finally generating an optimized version of the macro to replace the original macro. Lazily optimizing the macro in this manner allows macros to refer to themselves, and allows macros to reference as-yet undefined macros. By delaying optimization until the macro is first used, the static analyzer will have both a full definition of the macro and full definition of all the macros that the macro depends on, and can therefore use these implementations during template pre-expansion.

When pre-expanding a template, our analyzer tracks new bindings introduced in the template, giving up on any sections of the template where it cannot statically determine the exact bindings that are introduced in the new syntax. Whenever the analyzer encounters a form that it can prove invokes a macro, it attempts to prove which of the templates in the invoked macro will be selected. If it can with certainty select the appropriate template, it inlines the contents of that template into the invoking template.

4.4.3 Identifier Analysis

We can attempt other optimizations based on static analysis. For instance, if we can determine the shape of the binding sites introduced by the macro, we can statically determine which regular identifiers will eventually revert to their parent identifiers and which will be rewritten permanently. If no instances of a regular identifier in a macro ever get placed in a binding position, then all instances of that identifier will be rewritten into the parent identifier from which they descended. Therefore, no fresh regular identifier need be generated for it. Beyond avoiding unnecessary fresh identifier generation, we can mark identifiers at expansion time as already resolved to their final denotation to reduce the number of use environment lookups during macro expansion. The cost for doing so, unfortunately, is more memory allocations, as the identifiers must be wrapped so that they can can be tagged. With these tags come

```
(define-syntax glambda
  (syntax-rules ()
    ((_ (bind-symbol lookup-symbol) ((_id ida ...)) body)
     (lambda (temp)
       (bind-symbol (((ida ...) temp)) body)))))
```

Figure 4.7: Definition of glambda from the "dirtier macros" benchmark

more conditional branches in our primary macro expansion loop, which must handle the new tags as separate cases.

In figure 4.7, copied from the dirtier macros benchmark, we see an example of a template for which the boundedness of each regular identifier can be accounted for. The template for this macro contains mostly pattern identifiers; the only two regular identifiers are **lambda** and *temp*. For the former, we can confidently declare that there is no possible way for the **lambda** symbol to ever be placed in a binding position and will therefore always be free in the output expressions generated by this template. There is as such no need to generate a fresh identifier each time the macro is invoked.

The first instance of the identifier *temp*, on the other hand, assuming that **lambda** refers to its normal denotation, is guaranteed to be placed in a binding position; the second instance is also guaranteed to be captured by this binding. We can therefore mark all instances of this identifier in the output expression so that the expander knows when it encounters these symbols that it does not need to look up their denotation to see if they need to be rewritten.

Overall, to improve efficiency, macro static analysis must strike a careful balance in the amount of effort it spends optimizing macros and intruding on the existing algorithm, as the cost of the analysis and changes can dwarf the potential performance gains when the macro is actually invoked.

4.5 Other Racket-Specific Optimizations

4.5.1 Racket Keyword Arguments

Our expander originally made use of keyword arguments in several heavily-called procedures. We assumed that keyword arguments were handled in a phase prior to the runtime phase of execution. We later learned that this is not the case; there is a runtime overhead to Racket's keyword arguments, and we subsequently removed all uses of keyword arguments from performance critical procedures in the expander.

4.5.2 Hasheq vs Hash

The hashes we use in our expander primarly store symbols only. In Scheme it is much faster to compare symbols for equality using *eq?*, so we converted all *hash* constructors into *hasheq* constructors.

4.6 Optimization Results

Table 4.1 shows the mean and median effects of each optimization on benchmark times for our expander relative to the previous version not including the optimization. They are given in the same order that they were actually applied to the expander. Most optimizations resulted in a net performance increase, and most of those yielded significant improvements. The overall mean runtime reduction for all optimizations suite was 88.70%, or an 8.8x speedup. We discuss the effect of each optimization on the benchmark results in turn.

4.6.1 Lazy List Matching

Lazily matching sub-lists resulted in a minor net improvement on performance. We manually added log statements to discover that, in the course of running all benchmarks, a total of over 250,000 list matcher function applications occur that result in a match error. Of these, about 38,500, or 15.3%, terminate earlier because of lazy list matching. On average, lazy matching

Optimization	Mean Change	Median Change
lazy list match	-1.79%	0.00%
optimized/unsafe ops	-19.86%	-12.82%
no error diagnosis	-25.75%	-24.41%
optimized body expansion	-13.42%	-3.30%
pattern env vectors	-39.93%	-43.18%
regular identifier vectors	-21.42%	-20.00%
quote env merge	-18.46%	-19.05%
optimized template list rewriting	-16.29%	-19.35%
hasheq	-7.02%	-4.10%
no keyword args	-11.96%	-10.34%
pre-expand templates	8.11%	9.24%
binding inference	4.73%	-1.23%
best overall	-87.05%	-87.03%

Table 4.1: Mean and Median Performance Deltas Per Optimization

resulted in over 50% less sub-match functions being called inside a list matcher that fails to match. Lastly, about 39% of all list match function calls result in a match failure.

Based on the above numbers, we can see that pattern list match failures are a fairly common occurrence. However, overall, only about 6% of all list match calls result in a match failure that occurs before the last pattern element, which means that we would expect based on these numbers alone to avoid only about 3% of all match calls inside a list through lazy matching. Indeed, if we count the total number of sub-matcher invocations, only 3.3% of all possible match procedure invocations are avoided due to lazy list matching. This information explains the overall modest performance increase resulting from this optimization.

4.6.2 Unsafe Operations

Using Racket's optimized matching and looping constructs, along with unsafe list and vector operations, resulted in a 13% average decrease in benchmark run times. It is difficult to analyze the reasons for this performance increase without delving into the actual JIT-generated executed assembly code. As this optimization specifies no algorithmic or data structure changes in the expander itself, we do not investigate it further. It does show that avoiding runtime safety checks imposed by a high-level language can yield a significant performance improvement.

4.6.3 No Error Diagnosis

Measuring the precise impact of Racket match error diagnosis through Racket's statistical profiler proved very difficult. The profiler does not measure time spent in native code, and most of the cost of error diagnosis code lies in Racket's core string formatting procedures that are implemented with native code. Therefore, the decision to disable all such code was based on intuition alone. These string formatting functions in general are used to render potentially complex syntax elements into strings.

The optimization, however, yielded one of the best performance increases of all our attempted optimizations. We logged over 250,000 match failures occur over the course of running our benchmarks. Overall, the optimization saved 3.67 seconds total, which means that each string formatting operation required approximately 15 microseconds to complete. After further analysis, we discovered that, of these 3.67 seconds, almost 90% is spent converting Racket structs and lists into strings. This means that, if one does not wish to design a system where user-friendly match failure diagnosis is delayed until a separate step, one may be able avoid most of the costs of doing unnecessary error diagnosis through the use of a simple runtime delay mechanism such as closures.

4.6.4 Optimized Body Expansion

Optimized body expansion yielded an average 13.4% reduction in benchmark runtimes, but only a median 3% reduction. This large difference between the mean and median performance gains suggests that the optimization disproportionately effects the benchmarks, which is indeed the case. Optimizing **letrec** had the effect of halving the runtime of the monads benchmark, which makes extensive use of **letrec** forms and local **define** forms. The other benchmarks eschew the use of both constructs almost entirely.

The second optimization, avoiding a **letrec** rewrite when possible, once again halved the runtime of the monads benchmark. In addition, it also slightly improved the performance of several other benchmarks.

4.6.5 Pattern Environment Vectors and Regular Identifier Vectors

The hash-to-vector optimization halved the execution time of the majority of the benchmarks. Simple logging statements revealed that this optimization converted over five million hash lookups to vector lookups and over 3 million hash updates to vector updates. This, however, accounts for a relatively small proportion of the speedup. The remaining difference is most likely caused by reduced GC pressure and from avoiding repeatedly iterating over all elements in a hash.

While less dramatic an improvement, the regular identifier environment vector optimization also yielded a significant 20% drop in benchmark runtimes for similar reasons.

4.6.6 Quote Environment Removal

As this optimization involves again a decreased use of hash operations, it likewise again illustrates the relatively high cost of using hashes in Racket. Switching from updating two hashes to updating one hash with a pair of values resulted in an 18% average reduction in benchmark times.

4.6.7 Optimized Template List Rewriting

The optimizations for list rewriting yielded another significant improvement. We list each of the three sub-optimizations applied below and their average effect on benchmark runtimes when each is added cumulatively.

- Optimized *foldr* for general template lists: -14.2%.
- Optimized *map* for lists with no ellipsis templates: -2.3%.

- Optimized *map* for lists with one ellipsis template at the tail position: -1.8%.

To summarize, the majority of speedup from this optimization came not from optimizing based on the absence of ellipses in a list but by using an unsafe version of *foldr* for fusing the output lists together in the general case.

4.6.8 Hasheq and Keyword Arguments

Our final successful optimization attempts are both Racket-specific. Our environments are represented as hashes with symbols for keys, which can be compared for equality using pointer equality. Choosing this version of equality when constructing our hashes resulted in a 7% average reduction in benchmark times. We also verified that there is a significant runtime overhead to keyword arguments, and, by removing them from the expander's main procedures, average benchmark times were reduced by a further 11%.

4.6.9 Template Pre-expansion

After applying our pre-expansion optimization, we measured that our expander spends less than ten milliseconds pre-expanding templates over the course of running all benchmarks, which shows that the pre-expansion algorithm itself introduces minimal overhead. However, in our prototype implementation we added a layer of indirection in key locations in the code to support optimized macros. The implementation also creates two versions of each macro, one to expand normal syntax and one to expand templates. Finally, when optimizing a template one has to create a third macro, the final optimized macro for normal syntax. On the other hand, the optimized macros did not yield an appreciable increase in performance to make up for this overhead, resulting in a net performance loss.

4.6.10 Binding Inference

A similar story holds for binding inference. While it helps prevent in some cases some hash lookups and updates, it adds overhead in two key areas. First, each symbol whose scope is inferred must be wrapped inside a struct that serves as a tag, resulting in more memory allocations. Second, all other code that handles symbols must handle these tagged variants, including pattern matching logic in the expander's main loop. Overall, the costs associated with this optimization outweighed the benefits.

4.6.11 Final Results

Figures 4.8 and 4.9 show the incremental effect of each applied major optimization attempt on the individual benchmarks. For the most part, each optimization resulted in a cumulative reduction in runtimes up to the last two attempted optimizations. The notable exception to this is lazy list matching: this benchmark yielded a very minimal overall speed increase and in several benchmarks resulted in a speed decrease. Optimizations were more likely to have no impact at all in the lightweight benchmarks. This illustrates that these benchmarks are smaller and provide less coverage over the code base of an expander.

Figures 4.10 and 4.11 compare our optimized expander against the other expanders. The figures show that our expander is now competitive with the C-based Chez expander. The mean relative benchmark time of our expander compared to Chez's is 1.11. The relative runtime of the entire suite of benchmarks, 1.57, is less meaningful, as the run time of the suite is dominated by a single benchmark, the Fibonacci test. The geometric mean runtime of our expander relative to Chez is 1.0, suggesting that our optimized expander is very close to Chez for these benchmarks.

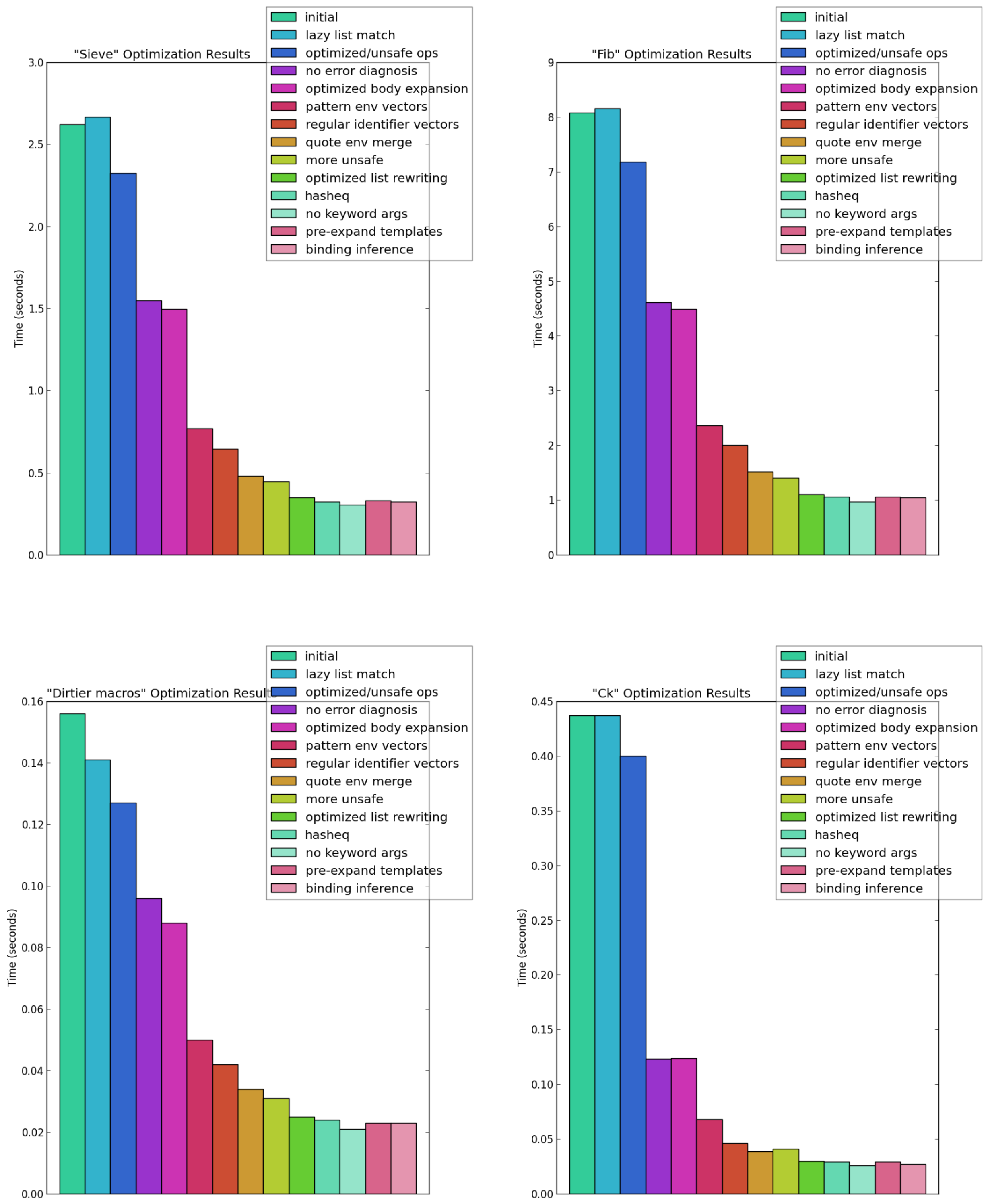

Figure 4.8: Impact of Optimizations on 'Heavyweight' Benchmarks

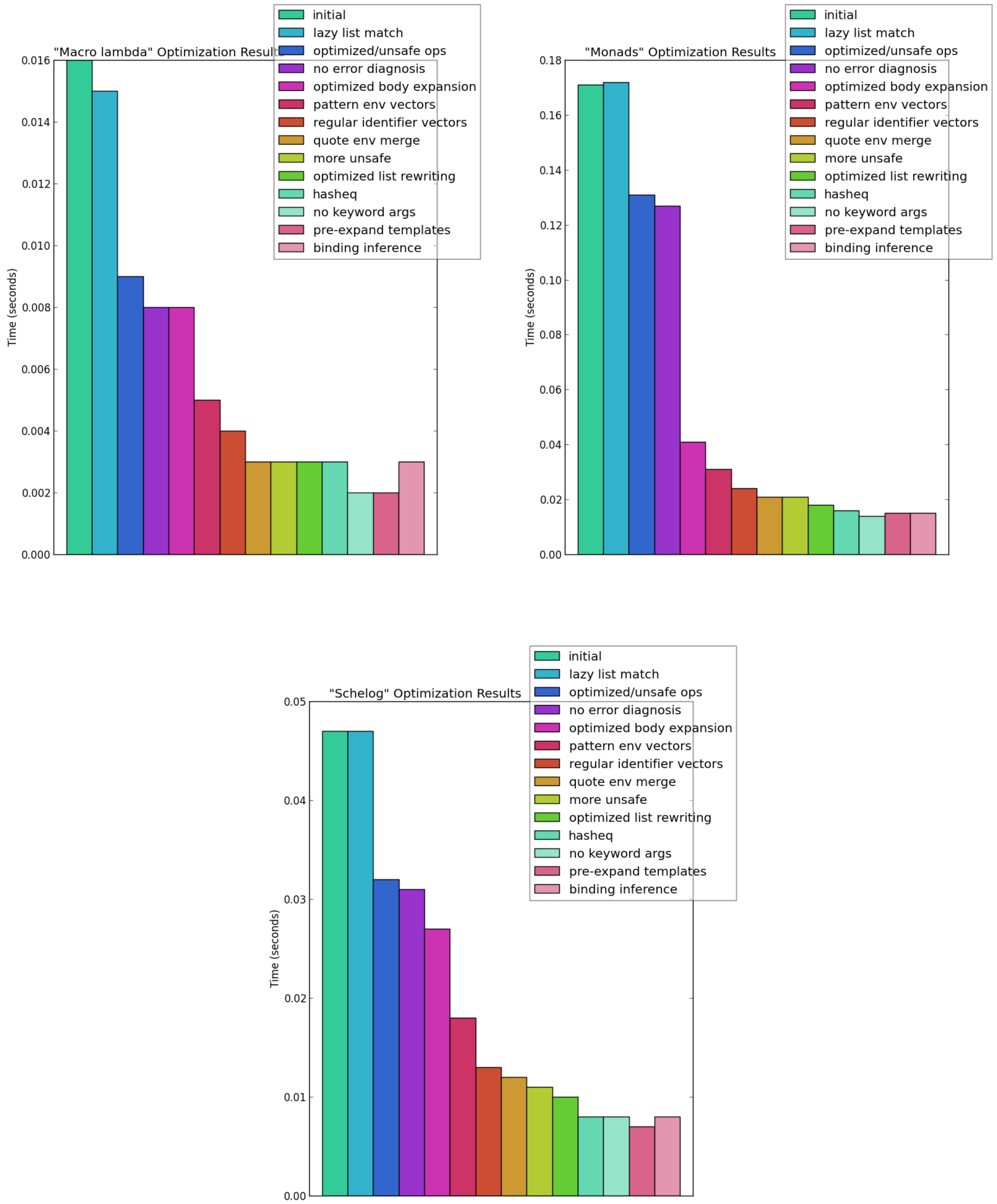

Figure 4.9: Impact of Optimizations on ‘Lightweight’ Benchmarks

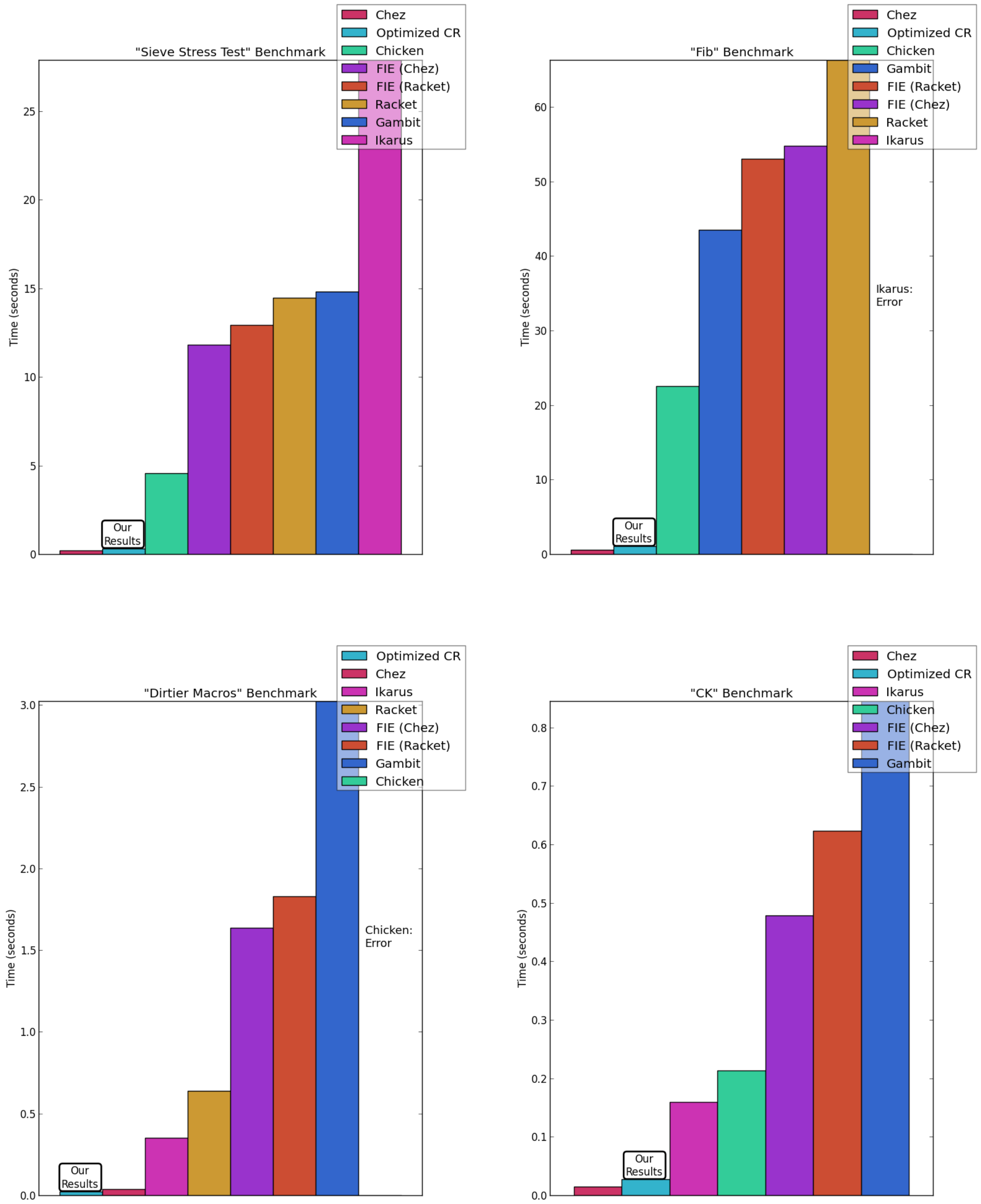

Figure 4.10: Final “Heavyweight” Benchmarks

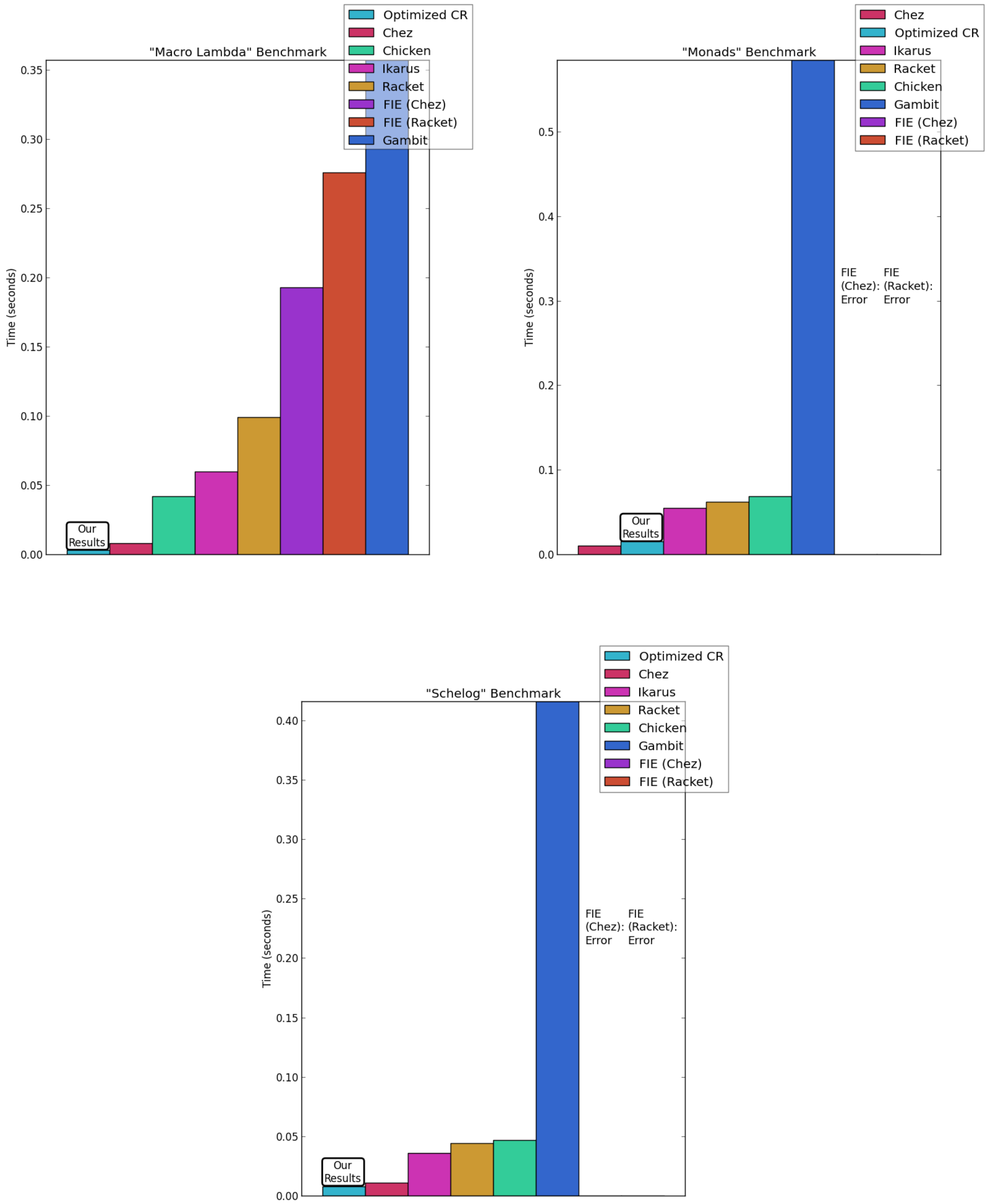

Figure 4.11: Final "Lightweight" Benchmarks

4.7 Summary

Given that compiled C programs are often benchmarked to be several times faster than equivalent programs written in Racket, we consider our optimized expander to successfully show that R5RS macro expanders can be optimized to run in a fraction of the time required by existing expanders. We also believe that it will be easier to experiment with further optimizations in the future, and to reuse portions of the expander to try new expansion algorithms because it is written in a higher level language. The expander itself is designed for reuse with separate components for the explicit renaming expansion algorithm, the R5RS top-level parser, and the R5RS initial top-level environment.

Chapter 5

Conclusions

R5RS macro expansion is complicated by its requirements for hygiene and referential transparency. Performant macro expansion, on the other hand, is important for fast compilation times and responsive semantic feedback from tools such as editors and development environments. Several algorithms have been invented to correctly and efficiently provide hygienic macro expansion for R5RS Scheme. We developed a suite of benchmarks to profile the performance of many existing R5RS macro expanders. We also built our own macro expander as a vehicle to explorer optimized macro expansion.

Our initial implementation of a macro expander revealed that a careful translation of the explicit renaming algorithm to code could in of itself yield a performant macro expander. By compiling **syntax-rules** definitions as hierachical procedures, our first expander, without any performance tuning, performed better than most other benchmarked expanders on several benchmarks. Despite this initial success, our expander trailed far behind the Chez expander and also exhibited poor general parsing performance, as shown in the Schelog and monads benchmarks. Since parsing and macro expansion are so tightly integrated, poor parsing performance can negate speed gained from a faster expander.

We discovered through manual logging and code inspection several important bottlenecks in our implementation. By altering our choice of data structures, fine-tuning our parsing and handling of primitive Scheme forms, and avoiding redundant and unnecessary computations, we improved the performance of our expander almost tenfold, making it competitive with or better than all other tested Scheme implementations in our benchmark suite.

Lastly, we experimented with static optimizations applied directly to macro templates. Our initial foray yielded negative results; our expander's performance did not improve with the optimizations we attempted.

5.1 Current Issues

Our expander currently has the unfair advantage that it is highly customized, or narrow in scope, relative to the some of the other benchmarked expanders which were written to operate as part of a compiler or interpreter. Some or all of these expanders also provide source-object correlation and better error diagnostics. A more fair performance comparison might be derived by integrating our expander with another production-quality interpreter or compiler.

We failed to find any open source R5RS programs larger than a thousand lines and the programs we found that made extensive use of macros were generally smaller than other programs. Larger program examples will provide more insight, both in comparing existing expanders and in deriving useful optimizations.

5.2 Future Work

Our first high-level optimization was to compile pattern and template expressions into hierarchical procedures. For standard library macros, and perhaps other macros, even more perfomant macro procedures might be obtained by developing a system to convert pattern and template expressions directly into Scheme code that can be evaluated, compiled and stored in pre-compiled Racket modules. Such a compilation strategy would remove the overhead of the function pointer indirection incurred through a hierarchy of procedures. This strategy has the potential to provide many opportunities for manual code inlining.

Our initial foray into template pre-expansion yielded a simple prototype that did not improve expansion performance. We believe that more effort could be spent, first in tracking performance regressions to prevent optimization from slowing down macro expansion in the worst case. Then, more effort could be spent developing better algorithms and heuristics, both for deciding when to apply the optimization and for improving the quality of the pre-expansion algorithm when applied. The advantage of template pre-expansion is that it can be applied to any expansion algorithm, not just explicit renaming, as long as its implementation is designed appropriately.

A second possible optimization route for our expander may come from re-implementing it in C to take advantage of manual memory management and more optimized data structures. As an example, there is likely to be a more performant data structure for environments than Racket's functional hashes. Also, the resources for local scopes can be cleaned up as soon as the scope has been fully handled, which cannot be done easily in a garbage-collected langage like Racket.

Finally, based on our research in existing algorithms, it is possible that the mark/substitution algorithm could be optimized to improve the speed of R6RS programs, which make use of a much richer macro system. One possible avenue could be explored by borrowing ideas from Nemerle's expansion algorithm.

5.3 Final Thoughts

One of the major productivity gains of higher level languages comes from fast development feedback cycles. A developer can quickly write code, compile it, and run it to observe its behavior. The sophisticated macro systems of languages like Racket offer for safer, easier-to-use macros at the cost of slower compilation times and less responsive tooling. Our efforts have shown that R5RS macro expansion can be optimized to expand programs in a fraction of the time required by existing expanders. We believe that there is still much that can be done to improve macro expansion performance for R5RS macros, as well as for the more advanced R6RS and Racket macro systems.

References

[1] Alan Bawden and Jonathan Rees. Syntactic Closures. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 86–95, 1988.

[2] William Clinger. Hygienic macros through explicit renaming. *SIGPLAN LISP Pointers*, 4(4):25–28, October 1991. ISSN 1045-3563.

[3] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–162, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8.

[4] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

[5] Marc Feeley. Gambit Scheme, July 2012. URL `http://dynamo.iro.umontreal.ca/~gambit/wiki/index.php/Main_Page`.

[6] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. URL `http://racket-lang.org/tr1`.

[7] Steven E. Ganz. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proceedings of the The International Conference on Functional Programming*, pages 74–85. ACM Press, 2001.

[8] Abdulaziz Ghuloum. Ikarus Scheme User's Guide, 2008. URL `http://cs.indiana.edu/~aghuloum/ikarus/ikarus-scheme-users-guide.pdf`.

[9] Abdulaziz Ghuloum and R. Kent Dybvig. Implicit phasing for R6RS libraries. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 303–314, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2.

[10] Abdulaziz Ghuloum and R. Kent Dybvig. Portable syntax-case, July 2012. URL `http://www.cs.indiana.edu/chezscheme/syntax-case/`.

[11] Richard Kelsey, William Clinger, and Jonathan Rees. Revised 5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998. ISSN 0362-1340.

[12] Oleg Kiselyov. How to write seemingly unhygienic and referentially opaque macros with syntax-rules. In *Scheme Workshop*, 2002.

[13] Oleg Kiselyov. Macros that compose: Systematic macro programming. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 202–217, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44284-7. URL `http://dl.acm.org/citation.cfm?id=645435.652699`.

[14] Oleg Kiselyov. Low- and high-level macro programming in Scheme, July 2012. URL `http://okmij.org/ftp/Scheme/macros.html`.

[15] Edward Kmett. Minimalist Polymorphic scheme-(co)monads, July 2012. URL `https://github.com/ekmett/scheme-monads`.

[16] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 151–161, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4.

[17] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, December 2002. ISSN 0362-1340.

[18] Dorai Sitaram. Programming in Schelog, July 2012. URL `http://www.ccs.neu.edu/home/dorai/schelog/schelog.html`.

[19] Kamil Skalski, Michal Moskal, and Pawel Olszta. Meta-programming in Nemerle, 2004. URL `http://nemerle.org/metaprogramming.pdf`.

[20] Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of LISP. In *Proceedings of The Second ACM SIGPLAN Conference on History of Programming Languages*, pages 231–270, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4.

[21] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211 – 242, 2000.

[22] Andre Van Tonder. SRFI 72: Hygienic macros, July 2012. URL `http://srfi.schemers.org/srfi-72/srfi-72.html`.

[23] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–215, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3.

[24] Felix Winkelmann. Chicken Scheme, July 2012. URL `http://www.call-cc.org`.