Taming Macros

Ryan Culpepper and Matthias Felleisen
ryanc@ccs.neu.edu

Northeastern University
Boston, MA, USA

Abstract. Scheme includes a simple yet powerful macro mechanism. Using macros, programmers can easily extend the language with new kinds of expressions and definitions, thus abstracting over recurring syntactic patterns. As with every other powerful language mechanism, programmers can also easily misuse macros and, to this day, broken macro definitions or macro uses pose complex debugging problems to programmers at all levels of experience.
In this paper, we present a type system for taming Scheme-like macros. Specifically, we formulate a small model that captures the essential properties of Scheme-style macros. For this model, we formulate a novel type system to eliminate some of these problems, prove its soundness, and validate its pragmatic usefulness.

1 The Power of Macro Programming

Over the past 20 years, the Scheme community has developed an expressive and easy-to-use standard macro system [1]. To introduce a new construct via a macro, a programmer simply writes down a rewriting rule consisting of a pair of syntactic patterns [2]. The left-hand side is called a *pattern*; the right-hand side is referred to as a *template*. Collectively the set of rules specifies how the macro expander must translate the surface syntax into core syntax. The macro expander eliminates each instance of the pattern in the program by replacing it with an instance of the template where all pattern variables have been appropriately substituted.

Scheme implementors often use the macro system to equip the language with additional forms of expressions. Programmers use the macro system for similar reasons. Many define small domain-specific notations and then formulate their program in a mix of Scheme and domain-specific expressions [3, 4]. The macro system thus empowers them to follow the old Lisp maxim on problem-solving via language definition, which says that programmers should formulate an embedded programming language for the problem domain and that they should express their solution for the domain in this new language.

Naturally, every powerful programming construct invites misuse. For example, a programmer may pass the wrong number of arguments to a function or attempt to apply an operation on the wrong kind of value (such as destructuring a number as if it were a pair or dividing by zero). These errors may cause

incorrect program results or run-time exceptions. A programmer may misuse macros in similar ways. Misuse of macros may cause the generation of an incorrect program or the triggering of errors during compilation. Given the weak programming environments for macro expansion, debugging macro misuses is extremely difficult compared to other constructs.

With this paper, we attempt to reconcile type checking with powerful syntactic abstractions in the spirit of Scheme. Naturally, the type system reduces the power of the macro system, but we believe that it retains enough for Scheme programmers with ordinary problems. We discuss the one exception to this claim in the conclusion. In the meantime, we briefly introduce macros; illustrate potential misuses; and present our unusual type system and a model for macro expansion so that we can prove type soundness. The penultimate section shows how we can extend our model to accommodate most of Scheme's expressive powers. Finally, we discuss related and future work toward taming macros.

Note: A preliminary version of this paper appeared in the Scheme and Functional Programming Workshop 2003 [5].

2 Why Macros?

Macros enable programmers to abstract over syntactic patterns. The most common uses of macros involve introducing new binding forms, changing the order or context of evaluation, and imposing static constraints on program fragments.

Suppose we wish to write a testing library and want a mechanism to test that an expression raises a specific run-time exception. For example, a programmer should be able to write a test case that looks like this:

```
(assert/exception (/ 1 0) div-by-zero-exception?)
```

The meaning of this expression can be described as follows: Evaluate the test expression (/ 1 0). If an exception is raised during the evaluation, test the exception value with the div-by-zero-exception? predicate. If the predicate holds, return true. Otherwise, if the predicate fails or if no exception was thrown, return false.

In Scheme (or any eager language), a programmer must use a macro to introduce this abstraction. Since the test expression must be evaluated in the context of new exception-handling code, **assert/exception** cannot be defined as a function.

In PLT Scheme [6], the macro might be defined as follows, using **with-handlers** to handle exceptions:

```
(define-syntax assert/exception
  (syntax-rules ()
    ((assert/exception test-expression expected-exn?)
     (with-handlers ((expected-exn? (lambda (exn) true))
                     ((lambda (exn) true) (lambda (exn) false)))
       test-expression
       false))))
```

Note how the first macro argument is placed into an exception handling context constructed from the second macro argument, before it is evaluated.

3 Macros are Too Powerful

Given their purpose, Scheme-style macros suffer from a critical problem. A macro can misapply the language's syntactic constructors, thus creating surface syntax that cannot be parsed into an AST or interpreted. The problem comes in two forms: an error in the macro use and an error in the macro definition.

First, the user of a macro may use it on syntactic forms that the creator of the macro didn't anticipate or intend to allow. Here is an increment macro, which is supposed to function in the context where everything else has its conventional Scheme meaning:

```
(define-syntax incr
  (syntax-rules ()
    ((incr x) (begin (set! x (+ x 1)) x))))
```

While the creator of the macro didn't expect anyone to use the macro with anything but an identifier,[1] the user—perhaps someone used to a different syntax—can misapply it to a vector-dereferencing expression:

```
... (incr (vector-ref a 0)) ...
```

The situation is particularly bad when a macro is imported from a module and the user is not able to understand or even access the macro definition.

Second, consider this macro definition:

```
(define-syntax where
  (syntax-rules (is)
    ((where body lhs is rhs) (let ((rhs lhs)) body))))
```

The intention is to define a **where** macro, which could be used like this:[2]

```
(where (+ x y)
  y is 5)
```

Unfortunately, the right-hand side of the rewriting rule for **where** misuses the rhs pattern variable as a **let**-bound identifier and thus creates an ill-formed expression.

At first glance, the situation is seemingly analogous to that of applying a programmer-defined Scheme function outside of its intended domain or defining a function improperly. In either case, the programmer receives an error message and needs to find the bug. Many Scheme systems offer sophisticated debugging aids for run-time exceptions. In contrast, programmers debugging macros have no such support. For example, in Chez Scheme [7], the misuse of **incr** generates the report that the syntax

```
(set! (incr (vector-ref v 0)) (+ (incr (...)) 1))
```

[1] Scheme's **set!** is only a variable assignment; it cannot mutate vectors, pairs, or other structures.

[2] Or like this in PLT Scheme: ((+ x y) . **where** . y **is** 5).

is invalid; the user of **where** finds out that

```
(let ((5 x)) (+ x 1))
```

is invalid syntax, without any clue of which portion of the program introduced this bug. Even in DrScheme [8], a sophisticated IDE that employs source code tracing and highlighting to provide visual clues, a programmer receives difficult-to-decipher error messages. The misuse of **incr** macro highlights the `vector` dereferencing expression and reports that some **set!** expression is ill-formed, which at least suggests that the error is in the use of **incr**. In contrast, for the use of **where**, DrScheme highlights the `5` and suggests that the **let** expression expects identifiers instead of numbers on the left. This leaves the programmer with at most a hint that the macro definition contains an error.

4 A Model of Scheme Macros

This section defines a small programming language with macros, inspired by Scheme. The macro expansion process is formalized with a rewriting semantics, because we consider this style of semantics the best suited to the syntax of macro definitions. The model omits certain properties of Scheme's macro system, including hygiene [9] and referential transparency [10]; they are not relevant for our purposes. Finally, we specify the goal of type checking in this context.

4.1 Syntax

Figure 1 specifies the syntax of our model programming language. It consists of a core language plus macro definitions and macro applications. More precisely, a program consists of a sequence of macro definitions followed by a sequence of top-level terms, which must be either definitions or expressions. Programs in the surface language are expanded into programs in the core language.

This model eliminates several complications from Scheme's macro system and syntax. In particular, there are no local macros (**let-syntax**), and identifiers are a different lexical class from macro keywords. A **lambda** term contains a single formal parameter. Finally, our model does not support literals or ellipses in macro patterns. We discuss in section 8 how to scale our model to a full-fledged version of Scheme.

We use the metavariables x, y to range over syntax in the language (but not macro definitions), m for macro keywords, and P, G, T for macro patterns, guards, and templates, respectively. We require that a pattern variable appear at most once in a pattern. Frequently we use m and the word "macro" to include primitive syntax (such as **lambda** and **quote**) as well; the meaning is always clear from the context.

4.2 Reduction Semantics for Expansion

Figure 2 specifies macro expansion with a reduction semantics. It consists of two relations and some auxiliary functions that deal with macros.

		Surface	Core
program	::=	*macro-def** *top-level**	*top-level**
top-level	::=	*def*	*def*
	|	*expr*	*expr*
def	::=	(**define** *id expr*)	(**define** *id expr*)
	|	(*macro s-expr*$*$)	
expr	::=	*id*	*id*
	|	*number*	*number*
	|	(*expr expr*)	
	|	(**lambda** (*id*) *expr*)	(**lambda** (*id*) *expr*)
	|	(**quote** *s-expr*)	(**quote** *s-expr*)
	|	(*macro s-expr**)	
	|		(**app** *expr expr*)
macro-def	::=	(**define-syntax** *macro* (**syntax-laws** *type* (*pattern guards s-expr*)*))	
pattern	::=	*pvar* | (*pattern**)	
guards	::=	((*pvar shape*)*)	
tag	::=	unspecified countable set	unspecified countable set
keyword	::=	**lambda**	**lambda**
	|	**define**	**define**
	|	**quote**	**quote**
	|	**define-syntax**	
	|	**syntax-laws**	
macro	::=	disjoint subset of *tag*	disjoint subset of *tag*
$i \in$ *id*	::=	disjoint subset of *tag*	disjoint subset of *tag*
pvar	::=	disjoint subset of *tag*	disjoint subset of *tag*
$x, y \in$ *s-expr*	::=	*keyword*	*keyword*
	|	*macro*	*macro*
	|	*id*	*id*
	|	*pvar*	*pvar*
	|	*number*	*number*
	|	(*s-expr**)	(*s-expr**)

Underlined nonterminals are part of the macro language, not the core language.

Fig. 1. The languages

The first relation $\xrightarrow{Prog}$ defines the expansion of an entire program. Macro definitions are collected into a macro environment ρ. The environment contains all the information from the macro definition, including type annotations. The reduction rules show, however, that the type annotations do not affect expansion.

Each top-level term is expanded in the macro environment ρ using the reduction relation $\xrightarrow{\rho}$ (note that ρ is a parameter of the relation). This reduction relation is defined for numbers, identifiers, applications, the three primitive keywords, and macro applications. The relation is compatible with expansion contexts (E), which allow expansion inside of **lambda**, **define**, and **app** terms but not inside of macro applications. The final result of a successful reduction sequence is a term of the core language (*CoreTerm*).

The recursive structure of expansion contexts (E) shows where expansion occurs, and the matching structure of core terms (*CoreTerm*) shows what expansion may produce in that context. Expansion only occurs in expression and definition contexts. Furthermore, macros must expand into expressions or definitions.

In our language, any parenthesized term in expansion position that is not a special form application must be a procedure application. In our reduction semantics, these are explicitly tagged with **app** after they have been recognized as such. In particular, macro applications cannot expand into keywords and cause expansion of the containing term:

Bad: $((\textbf{id}\ \textbf{id})\ 5) \xrightarrow{\rho} (\textbf{id}\ 5) \xrightarrow{\rho} 5$

is forbidden, just as it is in Scheme.

The single-step macro expansion is described via the auxiliary functions **match** and **transcribe**. If the macro arguments match a pattern, **match** produces a substitution (σ) that maps pattern variables to terms. Then **transcribe** applies the substitution to the template, which produces the replacement term for the macro application. If none of the macro's patterns matches its arguments, the macro is not expanded.

4.3 Stuck Terms

Our goal is to prevent macro expansion from getting stuck. Stuck terms represent those terms for which macro expansion goes awry. Technically, they are terms that are not in the core language and yet they cannot be further expanded. Stuck terms come in three groups.

The first class contains pattern variables, macro keywords, and primitive keywords. These terms have no meaning outside of a macro template or a macro application of the right form. For example, the term **quote** by itself is stuck.

The second class of stuck terms includes all macro applications (and procedure applications) that do not match a clause in ρ or a grammar clause in the core language. For example, (**lambda** x x) is stuck, because **lambda** requires its single formal parameter to be enclosed in parentheses.

The third class of stuck terms are those that have been completely expanded but are not terms of the core language. Thus, (**lambda** (x) (**define** xyz 5)) is stuck because the core grammar does not allow a definition within a **lambda** term.[3]

In a syntax closer to Scheme there would be additional classes of stuck terms. For instance, the formal parameters of **lambda** should be distinct, so (**lambda** (x x) 17) would be illegal. We believe, however, that errors such as this one are analogous to division-by-zero errors, which conventional type systems do not catch. For this reason, we have eliminated this complication from our grammar.

5 Shape Types

The semantics for macro expansion guides the development of the type system in two ways. First, the core language grammar and syntax of macro patterns determines the structure of our types. Second, since expansion happens only in certain contexts, the recursive structure of the type checker must respect these contexts.

5.1 Why Shapes?

Syntactically, our language consists of two essential syntactic categories: expressions and definitions. These are also the only two types into which macro applications can expand. The syntax that macros consume, however, cannot be described in terms of just expressions and definitions. After all, our grammar contains many other kinds of syntactic categories. For example, in the case of **lambda**, the first argument is an identifier within parentheses, which should not be typed as either an expression or a definition. If we wish to design a type language that describes all these intermediate *shapes* of syntax, it must cope with the basic identifiers, expressions, and definitions, and shapes built up by grouping with parentheses. We call these descriptions *shape types*.

Shape types provide a way of describing the terms that macros *and* primitive syntax constructors consume and produce. The type checker therefore treats macro applications and primitive keyword applications identically. In particular, an initial type environment contains types for primitive keywords.

Figure 3 defines the language of shape types. We use the term *type* to refer to only expr and def. The term *shape* refers to simple *type*s and complex, structured shapes. We use t to range over types and s to range over arbitrary shapes.

The success of a single expansion step of a macro application is determined only by the *shape* of its input. The shape type of the macro's input represents a guarantee that if the actual arguments match the shape type, then they also match one of the macro's patterns, and thus expansion does not get stuck in that step.

[3] Scheme allows some internal definitions, though not this one. Our model omits internal definitions to simplify the presentation and the proof.

PROGRAM

$$\frac{md_i = (\textbf{define-syntax}\ m_i\ (\textbf{syntax-laws}\ t_i\ (P_{i,1}G_{i,1}T_{i,1})\cdots)) \qquad i \le m \qquad \rho(m_i) = ([(P_{i,1}, G_{i_,1}, T_{i,1}), \cdots], t_i) \qquad i \le m \qquad x_k \xrightarrow{\rho}{}^{*} x'_k \qquad k \le n \qquad x'_k \in \mathit{CoreTerm} \qquad k \le n}{md_1 \cdots md_m\ x_1 \cdots x_n \xrightarrow{Prog} x'_1 \cdots x'_n}$$

REDUCTION

$$\begin{array}{lll} E & ::= [] \mid (\textbf{lambda}\ (i)\ E) \mid (\textbf{app}\ E\ x) \mid (\textbf{app}\ x\ E) \mid (\textbf{define}\ i\ E) \\ \mathit{CoreExpr} & ::= n \mid i \mid (\textbf{lambda}\ (i)\ \mathit{CoreExpr}) \mid (\textbf{app}\ \mathit{CoreExpr}\ \mathit{CoreExpr}) \mid (\textbf{quote}\ x) \\ \mathit{CoreDef} & ::= (\textbf{define}\ i\ \mathit{CoreExpr}) \\ \mathit{CoreTerm} & ::= \mathit{CoreExpr} \mid \mathit{CoreDef} \end{array}$$

$$\frac{x \xrightarrow{\rho} x'}{E[x] \xrightarrow{\rho} E[x']}$$

$$(x_1\ x_2) \xrightarrow{\rho} (\textbf{app}\ x_1\ x_2) \qquad x_1 \notin \mathit{macro}$$

$$(m\ x_1 \cdots x_n) \xrightarrow{\rho} \textbf{expand}\ \rho m(x_1 \cdots x_n) \qquad \text{if the right-hand side exists}$$

MATCH

$$\begin{array}{rl} \textbf{match}\ \rho m x = & (\sigma, T_k) \\ & \text{where } \rho(m) = ([(P_1, G_1, T_1) \cdots (P_n, G_n, T_n)], t) \\ & \text{and } \textbf{transcribe}\ P_k\sigma = x \text{ for minimal } k \\ \textbf{expand}\ \rho m x = & \textbf{transcribe}\ T\sigma \quad \text{where } (\sigma, T) = \textbf{match}\ \rho m x \\ \textbf{transcribe}\ x\sigma = & x \qquad \text{when } x \in \mathit{datum} \cup \mathit{id} \cup \mathit{keyword} \cup \mathit{macro} \\ \textbf{transcribe}\ p\sigma = & \sigma(p) \qquad \text{when } p \in \textbf{dom}\ (\sigma) \\ \textbf{transcribe}\ (x_1 \cdots x_n)\sigma = & (\ \textbf{transcribe}\ x_1\sigma \cdots \textbf{transcribe}\ x_n\sigma) \end{array}$$

Fig. 2. Reduction semantics

$$\begin{array}{lll} t \in \mathit{type} & ::= \mathsf{expr} \mid \mathsf{def} \\ s \in \mathit{shape} & ::= \mathit{type} \mid \mathsf{ident} \mid \mathsf{any} \\ & \mid (\mathit{shape}^*) \\ & \mid (\textbf{mclauses}\ (\mathit{shape}\ \mathit{shape}) \cdots (\mathit{shape}\ \mathit{shape})) \\ & \mid \mathit{shape} \rightarrow \mathit{type} \end{array}$$

Fig. 3. Shape Types

If all macro templates contain only macro applications of the correct shape, then expansion never becomes stuck. It is impossible to prove that a template has the correct shape if pattern variables be replaced with arbitrary terms, so we allow guards on the macro patterns to constrain the shapes of terms that pattern variables can match. The guards give the macro a more restrictive input shape, and they generate additional information for checking the templates.

In order to check a macro application, there must be a correspondence between terms and shapes. A particular term may be used in many ways, so it must have many potential shapes. Consider the following term:

(**define** (a b) (**let** ((a b)) (a b)))

The subterm (a b) occurs three times, each time used in a different way. The first occurrence is used as the shape (ident ident), the second as the shape (ident expr), and the third as just expr. Put differently, the assignment of shapes to terms is not unique and poses some complex problems.

5.2 Shape Types Guide Recursive Type Checking

Conventional type checkers synthesize the type of a phrase from the types of its pieces. For example, if the type checker encounters a term of the form $(e_1\ e_2)$, it recognizes an application and knows to recursively type check the two subterms. If e_1 is of type $a \to b$ and e_2 is of type a then $(e_1\ e_2)$ is of type b.

The conventional type checker can make this deduction for any $(e_1\ e_2)$ that occurs in an *expression position*. Clearly, no type checker would reject a program containing the string `"never write (car 17)"` on the grounds that `17` is not a list. The conventional type checker operates after the parser has determined the syntactic roles of all program fragments. Thus it knows that in the above string `(car 17)` does not occur in an expression context and does not require type checking.

In our system, type checking happens before macro expansion and thus before parsing. Rather than the parser determining the syntactic roles of program fragments, in our system the shape types of special forms dictate how to recursively type check the arguments to those special forms.

Consider this example:

(**moo** (quo rem) (div x y) (= rem 0))

Without some information about **moo**'s legal input shapes, it is impossible to determine the shape type of (quo rem) in **moo**'s argument, since **moo** may destructure and rearrange its arguments in arbitrary and unknown ways. If the type checker knows, however, that **moo** has shape

((ident ident) expr expr) → expr

then it can recursively check the appropriate arguments to **moo** as expressions. First it makes sure there are three arguments. Then it checks that the first

argument is a group of two identifiers and that the second and third can be validated to be expressions.

The shape associated with the macro is exactly the information necessary to perform type checking without expansion. Even when code has no macro applications and thus parsing could be done without expansion steps, shape types are still powerful enough to describe how to recursively type check uses of primitive keywords like **define** and **lambda**. In other words, the type checker itself does not need special rules for core keywords.

6 The Shape Type System

Type checking an entire program consists of building a macro type environment for the macro definitions and then type checking the macro templates and the subsequent top-level terms. Type checking a term ($\blacktriangleright$) requires auxiliary two auxiliary relations. One relates S-expressions with shapes ($\circ$), and the other defines an ordering on shapes ($\sqsubseteq$).

6.1 Checking A Complete Program

Figure 4 presents the type rules for programs. A program is typable ([Program]) if each top-level term is typable in the initial type environment augmented with the macro type environment from the macro definitions.

The [Environment] rule defines the "respects" relation $\vdash_{\mathcal{R}}$, which ensures that the macro type environment reflects the patterns and annotations of the macro definitions. It checks each macro clause with [Macro Type] for each macro definition as well as the templates (rule [Templates]) using $\Gamma \cup \Gamma_0$ and the guards.

The [Macro Type] rule relates patterns and shapes with the [Guarded] and [Unguarded] rules. The relation $\vdash_{\mathcal{U}} P : u$ indicates that terms with shape u match the pattern, and $\Phi \vdash_{\mathcal{G}} P : s$ says that when a term of shape s matches P, the resulting substitution satisfies Φ. The overlap relation ($\bowtie$) rejects macro definitions with overlapping patterns. Ensuring that the patterns do not overlap is necessary for the type soundness theorem.

6.2 From Terms to Shape Types

Figure 5 introduces the proof rules that relate S-expressions with their shapes. The rules use two kinds of environments: Γ for the macro type environment, mapping *macro* to *shape*, and Φ for the pattern variable environment, mapping *pvar* to *shape*. Top-level terms are type checked in the empty pattern variable environment, and macro templates are checked in the pattern variable environment corresponding to the guards of that clause.

A judgment $\Gamma; \Phi \vdash x \circ s$ means that in the type environments Γ and Φ, term x has shape s and can be used in any position that expects a term of shape s. Two of the rules deserve an explanation:

PROGRAM

$$\frac{md_i = (\textbf{define-syntax}\ m_i\ (\textbf{syntax-laws}\ t_i\ (P_{i,1}G_{i,1}T_{i,1})\cdots)) \quad \rho(m_i) = ([(P_{i,1}, G_{i,1}, T_{i,1}), \cdots], t_i) \quad \Gamma \vdash_{\mathcal{R}} \rho \quad \Gamma \cup \Gamma_0; \Phi \vdash x_k \blacktriangleright t_k \quad t_k \in \mathit{type}}{\vdash_{\mathcal{P}} md_1 \cdots md_n\ x_1 \cdots x_{n'}\ \textbf{correct}}$$

$$\Gamma_0(\textbf{lambda}) = ((\mathsf{ident})\ \mathsf{expr}) \to \mathsf{expr}$$
$$\Gamma_0(\textbf{define}) = (\mathsf{ident}\ \mathsf{expr}) \to \mathsf{def}$$
$$\Gamma_0(\textbf{quote}) = (\mathsf{any}) \to \mathsf{expr}$$
$$\Gamma_0(\textbf{app}) = (\mathsf{expr}\ \mathsf{expr}) \to \mathsf{expr}$$

ENVIRONMENTS

$$\frac{\forall m \in \mathbf{dom}\,(\Gamma) \cap \mathit{macro} : \Gamma(m) \vdash_{\mathcal{R}} \rho(m) \quad \forall m \in \mathbf{dom}\,(\rho) : \Gamma \vdash_{\mathcal{T}} \rho(m)}{\Gamma \vdash_{\mathcal{R}} \rho}$$

MACRO TYPE

$$\frac{\rho(m) = ([(P_0, G_0, T_0), \cdots (P_n, G_n, T_n)], t) \quad \Gamma(m) = (\textbf{mclauses}\ (u_1\ s_1)\cdots(u_n\ s_n)) \to t \quad \forall i \le n : G_i \vdash_{\mathcal{G}} P_i : s_i \quad \forall i \le n : \ \vdash_{\mathcal{U}} P_i : u_i \quad \forall i \ne j : u_i \not\bowtie u_j}{\Gamma(m) \vdash_{\mathcal{R}} \rho(m)}$$

TEMPLATES

$$\frac{\Gamma \cup \Gamma_0; G_i \vdash T_i \blacktriangleright t \qquad \forall i \le n \quad \rho(m) = ([(P_0, G_0, T_0), \cdots (P_n, G_n, T_n)], t)}{\Gamma \vdash_{\mathcal{T}} \rho(m)}$$

GUARDED1

$$\frac{p \in \mathit{pvar} \qquad \Phi(p) = s}{\Phi \vdash_{\mathcal{G}} p : s}$$

GUARDED2

$$\frac{\forall i \le n : \Gamma; \Phi \vdash_{\mathcal{G}} x_i : s_i}{\Phi \vdash_{\mathcal{G}} (x_1 \cdots x_n) : (s_1 \cdots s_n)}$$

UNGUARDED1

$$\frac{p \in \mathit{pvar}}{\vdash_{\mathcal{U}} \mathit{pvar} : \mathsf{any}}$$

UNGUARDED2

$$\frac{\forall i \le n : \vdash_{\mathcal{U}} x_i : u_i}{\vdash_{\mathcal{U}} (x_1 \cdots x_n) : (u_1 \cdots u_n)}$$

$$u \in \mathit{unguarded\ shapes} ::= \mathsf{any} \ \mid \ (u_1 \cdots u_n)$$

OVERLAP1

$$\forall u : \mathsf{any} \bowtie u$$

OVERLAP2

$$\frac{\forall i \le n : u_i \bowtie u_i'}{(u_1 \cdots u_n) \bowtie (u_1' \cdots u_n')}$$

Fig. 4. Program Correctness

$$\textsc{Shape1}\ \frac{x \in \mathit{id}}{\Gamma;\Phi \vdash x \circ \mathsf{ident}} \qquad \textsc{Shape2}\ \frac{x \in \mathit{number}}{\Gamma;\Phi \vdash x \circ \mathsf{expr}} \qquad \textsc{Shape3}\ \frac{x \in \mathit{macro} \cup \mathit{keyword} \quad x \in \mathbf{dom}\,(\Gamma)}{\Gamma;\Phi \vdash x \circ \Gamma(x)} \qquad \textsc{Shape4}\ \frac{x \in \mathit{pvar} \quad x \in \mathbf{dom}\,(\Phi)}{\Gamma;\Phi \vdash x \circ \Phi(x)}$$

$$\textsc{Shape5}\ \frac{x \in \mathit{macro} \cup \mathit{pvar} \cup \mathit{keyword} \quad x \notin \mathbf{dom}\,(\Gamma) \cup \mathbf{dom}\,(\Phi)}{\Gamma;\Phi \vdash x \circ \mathsf{any}} \qquad \textsc{Shape6}\ \frac{\Gamma;\Phi \vdash x \circ s \quad s \sqsubseteq s'}{\Gamma;\Phi \vdash x \circ s'} \qquad \textsc{Shape7}\ \frac{\Gamma;\Phi \vdash x_i \circ s_i \qquad i \leq n}{\Gamma;\Phi \vdash (x_1 \cdots x_n) \circ (s_1 \cdots s_n)}$$

Fig. 5. From Terms to Shapes

[**Shape3**] If a macro keyword has a definition, then it has the shape (an arrow shape type) recorded in the type environment.

[**Shape5**] If a keyword or pattern variable is not bound, it has shape any. It can be used only in positions that place no constraints on the shape; for example, the argument to **quote** takes a term of any shape.

Any S-expression can be given some shape type. In particular, any S-expression can be given the shape type any.

6.3 Ordering Shapes

The judgment $s \sqsubseteq s'$ means that shape s generalizes to shape s'. Put differently, s' reveals fewer details than s. The ordering rules are introduced in fig. 6.

$$\textsc{Order1}\ \mathsf{ident} \sqsubseteq \mathsf{expr} \qquad \textsc{Order2}\ s \sqsubseteq \mathsf{any} \qquad \textsc{Order3}\ \frac{\forall k \leq n : s_k \sqsubseteq s'_k}{(s_1 \cdots s_n) \sqsubseteq (s'_1 \cdots s'_n)} \qquad \textsc{Order4}\ \frac{(s_1 \cdots s_n) \sqsubseteq s}{(s \to t\ s_1 \cdots s_n) \sqsubseteq t}$$

$$\textsc{Order5}\ \frac{}{(\mathsf{expr}\ \mathsf{expr}) \sqsubseteq \mathsf{expr}} \qquad \textsc{Order6}\ \frac{s \sqsubseteq s_k \qquad \forall i \neq j : u_i \not\bowtie u_j}{s \sqsubseteq (\mathbf{mclauses}\ (u_0\ s_0) \cdots (u_k\ s_k) \cdots (u_n\ s_n))} \qquad \textsc{Order7}\ \frac{}{s \sqsubseteq s}$$

$$\textsc{Order8}\ \frac{s \sqsubseteq s' \qquad s' \sqsubseteq s''}{s \sqsubseteq s''}$$

Fig. 6. Ordering Relation on Shapes

The rule [Order4] says that groupings that look like macro applications (and satisfy certain constraints) may be interpreted as the result type of the macro.

[Order5] applies to procedure applications and expr. Analogous rules are included in the actual type checking rules.

The final rule [Order6] governs matching shapes of the arguments to macros against the shapes derived from the clauses in a macro definition. The kth clause of a macro definition yields a pair $(u_k\ s_k)$ in the **mclauses** shape type. The first is the unguarded pattern shape; the second is the guarded pattern shape. By construction, the guarded pattern shape is always below the unguarded pattern shape. A shape is below the shape of the clause if it matches (exactly) one of the guarded shapes. The antecedent that requires macro not to overlap guarantees that it does not match more than one. This allows us to establish a definite correlation between the clause that causes the shape checking to succeed and the clause that matches during expansion.

6.4 Type Checking

The goal of type checking is to ensure that top-level terms have each have a type, not just a shape. Type checking takes place in the context of the type environments $\Gamma;\Phi$, in the same way shape checking does.

A judgment of the form $\Gamma;\Phi \vdash x \blacktriangleright t$, where $t \in \mathit{type}$, means that x has type t in the context described by $\Gamma;\Phi$. A macro template is type checked in a pattern variable environment derived from the guards. The typing rules in fig. 7 have the following meaning:

1. Identifiers and numbers are both expressions.
2. If a pattern variable contains a term of expression or definition shape, then that term is an expression or a definition.
3. A macro application has the macro's declared result type if all of its arguments are of the right shape.
4. Two expressions grouped together constitute a procedure application, which is an expression.

7 Soundness

We prove type soundness for this system via subject reduction [11]. Following preservation and progress theorems, we discuss the untypability of stuck terms.

7.1 Preservation and Progress

The main theorems are conventional theorems about type systems. The Preservation Theorem proves that if a program has a type then the expanded program has the same type. The Progress Theorem shows that a typed term is either a term in the core language or expandable.

While the theorems look familiar, the proofs require different lemmas and techniques. Lemma 1 allows us to switch back and forth between proofs of the $\circ$ relation and $\blacktriangleright$ relation. We use this lemma for inductive proofs where the

$$\textsc{Check1}\ \frac{x \in \mathit{id} \cup \mathit{number}}{\Gamma;\Phi \vdash x \blacktriangleright \mathsf{expr}} \qquad \textsc{Check2a}\ \frac{x \in \mathit{pvar} \qquad \Phi(x) \sqsubseteq \mathsf{expr}}{\Gamma;\Phi \vdash x \blacktriangleright \mathsf{expr}} \qquad \textsc{Check2b}\ \frac{x \in \mathit{pvar} \qquad \Phi(x) \sqsubseteq \mathsf{def}}{\Gamma;\Phi \vdash x \blacktriangleright \mathsf{def}}$$

$$\textsc{Check3a}\ \frac{\begin{array}{c} m \in \mathit{macro} \cup \mathit{keyword} \qquad \Gamma;\Phi \vdash m \circ s \to \mathsf{expr} \\ (s_1 \cdots s_n) \sqsubseteq s \qquad \forall i \le n : \Gamma \vdash x_i \circ s_i \end{array}}{\Gamma;\Phi \vdash (m\ x_1 \cdots x_n) \blacktriangleright \mathsf{expr}}$$

$$\textsc{Check3b}\ \frac{\begin{array}{c} m \in \mathit{macro} \cup \mathit{keyword} \qquad \Gamma;\Phi \vdash m \circ s \to \mathsf{def} \\ (s_1 \cdots s_n) \sqsubseteq s \qquad \forall i \le n : \Gamma \vdash x_i \circ s_i \end{array}}{\Gamma;\Phi \vdash (m\ x_1 \cdots x_n) \blacktriangleright \mathsf{def}} \qquad \textsc{Check4}\ \frac{\Gamma;\Phi \vdash x_i \circ \mathsf{expr}}{\Gamma;\Phi \vdash (x_0\ x_1) \blacktriangleright \mathsf{expr}}$$

Fig. 7. Type Checking

hypotheses involve a different relation than the conclusion. Lemma 2 says that a macro application that type checks always has an expansion, and that the expansion also type checks. Lemma 3 shows that transcriptions of macro templates preserve type if the substitutions respect the template's guards. Lemma 4 guarantees that shape types that do not overlap have no terms in common. This implies that if the unguarded pattern shapes do not overlap, then a term matches a unique pattern.

Theorem 1 (Preservation). *If* $x \xrightarrow{\rho} x'$, $\Gamma \vdash_{\mathcal{R}} \rho$, *and* $\Gamma;\emptyset \vdash x \blacktriangleright t$, *then* $\Gamma;\emptyset \vdash x' \blacktriangleright t$.

Proof. The proof for redexes in the empty evaluation context is by case analysis of the reduction rules. If $x = (x_1\ x_2)$ then we must have $t = \mathsf{expr}$ and x_1 and x_2 both have shape expr, so the term $(\mathbf{app}\ x_1\ x_2)$ also has type expr. The case of macro applications is handled by Lemma 2.

Once we have proved subject reduction for empty contexts, we can use induction on the structure of the context, aided by Lemma 1. □

Theorem 2 (Progress). *If* $\Gamma \vdash_{\mathcal{R}} \rho$, *and* $\Gamma;\emptyset \vdash x \blacktriangleright t$, *then either* $x \in \mathit{CoreTerm}$ *or there is an* x' *such that* $x \xrightarrow{\rho} x'$.

Proof. The proof is by induction on the proof $\Gamma;\emptyset \vdash x \blacktriangleright t$ and case analysis of the last proof step.

[**Check1**] If x is a number or identifier, then x is a core term.
[**Check2**] The pattern variable environment is empty, so this case cannot occur.
[**Check3**] *on primitive syntax:* We prove the claim for **lambda**; **define** is similar, and **quote** is trivial.
Assume $x = (\mathbf{lambda}\ (i)\ y)$. The type of **lambda** ensures that $\Gamma;\emptyset \vdash y \circ \mathsf{expr}$ and Lemma 1 gives us $\Gamma;\emptyset \vdash y \blacktriangleright \mathsf{expr}$. We can apply the induction hypothesis

to get $y \xrightarrow{\rho} y'$ or $y \in \mathit{CoreTerm}$. Then we have either a reduction $x \xrightarrow{\rho}$ (**lambda** (i) y') or (**lambda** (i) y) $\in \mathit{CoreTerm}$.

[Check3] *on macros:* If $x = (m\ x_1 \cdots x_n)$, then Lemma 2 guarantees that the expansion succeeds, so x is a redex.

[Check4] If $x = (x_1\ x_2)$ then we have a reduction $(x_1\ x_2) \xrightarrow{\rho}$ (**app** $x_1\ x_2$). □

Lemma 1. *$\Gamma; \emptyset \vdash x \blacktriangleright t$ iff $\Gamma; \emptyset \vdash x \circ t$.*

Proof. The proof of the forward implication is a simple case analysis of the last proof step of $\Gamma; \emptyset \vdash x \blacktriangleright t$ and a translation into the Shape and Order rules.

The proof of the reverse implication is by induction on the proof $\Gamma; \emptyset \vdash x \circ t$, including both Shape and Order rules. □

Lemma 2. *If $\Gamma; \emptyset \vdash (m\ x_1 \cdots x_n) \blacktriangleright t$ and $\Gamma \vdash_{\mathcal{R}} \rho$ then* ***expand*** *$\rho m(x_1 \cdots x_n)$ exists and $\Gamma; \emptyset \vdash$* ***expand*** *$\rho m(x_1 \cdots x_n) \blacktriangleright t$.*

Proof. We use Lemma 4 to show that $(x_1 \cdots x_n)$ matches a unique pattern. Then since $(x_1 \cdots x_n)$ matches the macro input shape, the resulting substitution σ maps pattern variables to terms with the shapes that satisfy the guard. The macro correctness condition requires that templates type check under the pattern variable environment of their guards. Then by Lemma 3 the transcribed template type checks in an empty pattern environment. □

Lemma 3. *Let σ be a substitution. If $\Gamma; \Phi \vdash x \circ s$ and $\forall p \in$* ***dom*** *$(\Phi) : \Gamma; \emptyset \vdash \sigma(p) \circ \Phi(p)$, then $\Gamma; \emptyset \vdash$* ***transcribe*** *$x\sigma \circ s$.*

Proof. The proof is by straightforward induction on the proof of $\Gamma; \Phi \vdash x \circ s$ and case analysis on the last proof step. □

Lemma 4. *For any fixed Γ, $u_1 \bowtie u_2$ if and only if there exists a term x such that $\Gamma; \emptyset \vdash x : u_1$ and $\Gamma; \emptyset \vdash x : u_2$.*

Proof. The forward direction is by induction on the derivation of $u_1 \bowtie u_2$. The reverse direction is by induction on the structure of x and case analysis of u_1 and u_2.

7.2 Stuck Terms Revisited

Now we can examine the three classes of stuck terms in the shape type system. By inspection we easily see that the type system does not type check stuck terms as either expr or def, meaning that well-typed programs produce proper syntax.

The first class of stuck states consists of pattern variables and keywords. The only shape the type checker assigns to these is any. The second class is the set of macro applications with incorrectly shaped arguments. These cannot be assigned a *type*, because the rules for type checking require correctly shaped arguments. The third class contains expanded terms that are not core terms. These terms violate the shape types of the primitive keywords, so these terms are not typed.

8 Extensions and Pragmatics

Standard Scheme allows programmers to write macros that process arbitrary sequences of inputs, expressed using ellipses in the rewrite rules. For example, the **or** form can be applied to zero or more expressions. Standard Scheme macros can also match on literal data and keywords in patterns. For example, the **cond** form recognizes the keyword **else**.

Extending our model with sequences introduces two kinds of complications. First, a form like **lambda** imposes additional context-sensitive constraints on expressions, which the type system cannot express. In particular, the parameter list may not contain the same identifier twice. As mentioned, we consider such errors analogous to division-by-zero or out-of-bounds array referencing, which type systems in ordinary languages also cannot eliminate. Similarly, various kinds of ellipsis mismatch are beyond the scope of a shape type system. Second, we need to extend the type system to cope with keywords and ellipses in macros.

Keywords in patterns require the extension of our type system. Specifically, it requires the addition of a collection of singleton shape types, each representing one keyword. To preserve the non-overlapping property, these keywords must be a lexical class separate from identifiers and unusable as expressions.

Macros that process sequences also require an extension of the shape language with shapes of the form $(s_r \ldots . s_f)$. The shape $(s_r \ldots . s_f)$ contains all terms of shape s_f as well as all terms of shape $(s_r \,.\, (s_r \ldots . s_f))$. This allows us to express shapes describing a sequence of definitions followed by a nonempty sequence of expressions. We can then use Amadio-Cardelli style recursive subtyping [12] to handle sequence shapes.

To test the pragmatics of our type system, we have reformulated the definitions for the collection of macros dubbed "derived syntax" in R^{5}RS. A reformulation is necessary so that we can annotate the pattern variables with the correct shapes in this extended system. In particular, the ($p0$ p ...) idiom must be replaced when the sequence is not homogeneous such as with the body of a **lambda** expression where all internal definitions must occur before the (nonempty) sequence of expressions. The appendix provides one example from this test and shows that the difference are minor and do not impose an extraordinary amount of work on the programmer.

9 Related Work

Over the past ten years, other language communities have recognized the value of macro systems and have started to explore their use. Ganz, Sabry, and Taha [13] have designed and implemented MacroML, a version of OCaml with macros. Peyton-Jones has investigated the use of macros in conjunction with Haskell [14]. Other researchers have explored macro-like systems for languages with conventional C-style syntax [15–18]. None of the systems achieves the expressive power of Scheme macros.

The system for C presented in [15] introduces quasiquotation for conventional syntax and appears to statically check *most* templates for correctness. No definition of soundness is given, and no proof is attempted.

Maya, a syntactic extension system for Java[18], allows programmer to extend the syntax of Java using grammar extensions and associated transformers ("semantic actions"). Maya uses types and other pattern annotations to control the parsing of macro arguments. The authors claim to catch syntax errors in macro definitions, but the paper does not include proof of correctness.

The MacroML system by Ganz et al is a type-safe macro system based on MetaML. In fact, their type discipline guarantees that ML-style macros generate proper, well-typed ML code. MacroML achieves soundness, however, by severely restricting the expressive power of the macro system, beyond the point of our own constraints.

10 Conclusion

In this paper, we have presented a type system for macros and proved its soundness. We are now exploring the problem of scaling this system up to handle the complexities of advanced Scheme macros.

We are now developing a prototype implementation of the type checker, exploring its practicality for full Scheme. We are also investigating how to overcome the one major restriction of our system that we haven't addressed yet: macro-generating macros. While such abstractions appear to be esoteric at first, they do occur on occasion in real programs and we believe that a full-fledged shape type system must eventual address this issue.

Our shape system also fails to handle programmed macro systems, such as the **syntax-case** system [19]. We are investigating the use of contracts (as described by Findler [20, 21]) for taming programmed macros.

Acknowledgments We are grateful to Robby Findler for helping simplify the reduction semantics, and to Matthew Flatt and the anonymous reviewers for many suggestions and improvements.

References

1. Kelsey, R., Clinger, W., Rees (Editors), J.: Revised^5 report of the algorithmic language Scheme. ACM SIGPLAN Notices **33** (1998) 26–76
2. Kohlbecker, E.E., Wand, M.: Macros-by-example: Deriving syntactic transformations from their specifications. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1987) 77–84
3. Bentley, J.: Programming pearls: little languages. Commun. ACM **29** (1986) 711–721
4. Shivers, O.: A universal scripting framework, or lambda: the ultimate little language. In: Concurrency and Parallelism, Programming, Networking, and Security. Springer Lecture Notes in Computer Science, Springer-Verlag (1996) 254–265

5. Culpepper, R., Felleisen, M.: Well-shaped macros. In: Proceedings of the Fourth Workshop on Scheme and Functional Programming. (2003) 59–68
6. Flatt, M.: PLT MzScheme: Language manual. Technical Report TR97-280, Rice University (1997) http://www.plt-scheme.org/software/mzscheme/.
7. Dybvig, R.K.: Chez Scheme User's Guide. Cadence Research Systems (1998)
8. Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: A programming environment for Scheme. Journal of Functional Programming **12** (2002) 159–182 A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pp. 369–388.
9. Kohlbecker, E.E., Friedman, D.P., Felleisen, M., Duba, B.F.: Hygienic macro expansion. In: ACM Symposium on Lisp and Functional Programming. (1986) 151–161
10. Clinger, W., Rees, J.: Macros that work. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1990) 155–162
11. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Information and Computation (1994) 38–94 First appeared as Technical Report TR160, Rice University, 1991.
12. Amadio, R., Cardelli, L.: Subtyping recursive types. In: ACM Transactions on Programming Languages and Systems. Volume 15. (1993) 575–631
13. Ganz, S.E., Sabry, A., Taha, W.: Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In: International Conference on Functional Programming. (2001) 74–85
14. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In Chakravarty, M.M.T., ed.: ACM SIGPLAN Haskell Workshop 02, ACM Press (2002) 1–16
15. Weise, D., Crew, R.: Programmable syntax macros. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (1993) 156–165
16. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: tools for implementing domain-specific languages. In: Proceedings Fifth International Conference on Software Reuse, Victoria, BC, Canada, IEEE (1998) 143–153
17. Bachrach, J., Playford, K.: The Java syntactic extender (jse). In: Processdings of the Conference on Object-Oriented Programming Systems. (2001) 31–24
18. Baker, J., Hsieh, W.: Maya: Multiple-dispatch syntax extension in java. In: Proc. ACM Conference on Programming Language Design and Implementation. (2002) 270–281
19. Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in Scheme. Lisp and Symbolic Computation **5** (1993) 295–326
20. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ACM SIGPLAN International Conference on Functional Programming. (2002)
21. Findler, R.B.: Behavioral Software Contracts. PhD thesis, Rice University (2002)
22. Cardelli, L., Matthes, F., Abadi, M.: Extensible syntax with lexical scoping. Research Report 121, Digital SRC (1994)
23. Dybvig, R.K.: The Scheme Programming Language. 1 edn. Prentice-Hall (1987)
24. Flatt, M.: Composable and compilable macros: You want it *when?* In: ACM SIGPLAN International Conference on Functional Programming. (2002)
25. Queinnec, C.: Macroexpansion reflective tower. In Kiczales, G., ed.: Proceedings of the Reflection'96 Conference, San Francisco (California, USA) (1996) 93–104

A R5RS Macro Example

The following code is a reformulation of the R^{5}RS macro for **cond** with shape annotations. We use ellipses to write sequence types and **kw:else** for the singleton shape containing the keyword **else**, and likewise for =>. We also use named shape abbreviations.

```
(type expr+ (expr ... expr))
(type cond-clause (union (expr kw:=> expr) expr+))
(type cond-clauses
  (cond-clause ... (union cond-clause (kw:else . expr+))))

(define-syntax cond
  (syntax-laws expr
    ((cond (else . result)
     ([result expr+])
     (begin . result)))
    ((cond (test => result))
     ([test expr] [result expr])
     (let ((temp test))
       (if temp (result temp))))
    ((cond (test => result) . clauses)
     ([test expr] [result expr] [clauses cond-clauses])
     (let ((temp test))
       (if temp
           (result temp)
           (cond . clauses))))
    ((cond (test))
     ([test expr])
     test)
    ((cond (test) . clauses)
     ([test expr] [clauses cond-clauses])
     (let ((temp test))
       (if temp
           temp
           (cond . clauses))))
    ((cond (test . result))
     ([test expr] [result expr+])
     (if test (begin . result)))
    ((cond (test . result) . clauses)
     ([test expr] [result expr+] [clauses cond-clauses])
     (if test
         (begin . result)
         (cond . clauses)))))
```