Well-Shaped Macros

Ryan Culpepper, Matthias Felleisen
Northeastern University
Boston, MA 02115
Email: ryanc@ccs.neu.edu

Abstract

Scheme includes an easy-to-use and powerful macro mechanism for extending the programming language with new expression and definition forms. Using macros, a Scheme programmer can define a new notation for a specific problem domain and can then state algorithms in this language. Thus, Scheme programmers can formulate layers of abstraction whose expressive power greatly surpasses that of ordinary modules.

Unfortunately, Scheme's macros are also too powerful. The problem is that macro definitions extend the parser, a component of a language's environment that is always supposed to terminate and produce predictable results, and that they can thus turn the parser into a chaotic and unpredictable tool.

In this paper, we report on an experiment to tame the power of macros. Specifically, we introduce a system for specifying and restricting the class of shapes that a macro can transform. We dub the revised macro system `well-shaped macros`

1. MACROS ARE USEFUL

Over the past 20 years, the Scheme community has developed an expressive and useful standard macro system [8]. The macro system allows programmers to define a large variety of new expression and definition forms in a safe manner. It 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.

Standard Scheme macros are easy and relatively safe to use. To introduce a macro, a programmer simply writes down a rewriting rule between two syntactic patterns [10], also called `pattern` and `template`. Collectively the rules specify how the macro expander, which is a component of the parser, must translate surface syntax into core Scheme, that is, Scheme without any extensions. Specifically, the (left-hand side) pattern specifies those S-expressions that the expander should eliminate in favor of the (right-hand side) template. Furthermore, the expander is "hygienic" [9] and "referentially transparent" [3], which means that macro expansion automatically respects the lexical scope of the program.

It is a key characteristic of Scheme macros that their uses are indistinguishable from built-in forms. As for built-in and defined functions, a programmer should not, and without context cannot, recognize whether a form is introduced via a macro or exists in core Scheme. Due to this uniformity, (teams of) programmers can build many-tiered towers of abstraction, each using conventional procedural libraries as well as new linguistic mechanisms.[1]

Although the Scheme authors have clearly tamed Lisp's programmed macros and C's string rewriting macros, they have still left the macro sublanguage with as much power as the untyped lambda calculus. In particular, macro expansion can create ill-formed core syntax, and it can diverge. We illustrate this point with some examples in the next section.

The situation suggests that we study ways of taming Scheme macros with a type system.[2] In this paper, we report on the results of one such an experiment. In section 2 we explain how Scheme macros can still perform undesirable computations. In sections 3 and 4, we introduce a modified macro system that allows a Scheme implementation to determine whether macro definitions and programs are syntactically well-formed. In section 5, we compare our work to related work and propose some future research.

2. MACROS ARE TOO POWERFUL

Standard Scheme macros suffer from two problems. On one hand, they can turn the macro expander into an infinite loop. Since the expander is a part of the parser, a programmer can turn the most reliable part of an ordinary programming environment into a useless tool. On the other hand, a macro can misapply Scheme's syntactic constructors, creat-

[1] We readily acknowledge that building such towers poses additional, serious problems for language designers [6, 11], but this topic is beyond the scope of our paper.

[2] If we were to eliminate ellipses and introduce an induction schema, our result would literally reconstruct for macro systems what the type discipline of Church did for the original lambda calculus.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission. Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA. Copyright 2003 Ryan Culpepper, Matthias Felleisen.

ing S-expressions that even Scheme cannot interpret.

Consider the following simple macro definition:

```
(define-syntax diverge
  (syntax-rules () ((_) (diverge))))
```

It introduces a macro that replaces occurrences of (`diverge`) with itself, thus causing the extended parser to diverge. This example of bad behavior is trivial, however, compared to the full power of macros. It is a simple exercise to write a set of macros that simulate a pushdown automaton with two stacks.

While the introduction of unbridled computational power is a problem, we are truly concerned with macros that create ungrammatical Scheme expressions and definitions. Macro definitions can go wrong in two ways. First, the user of a macro may use it on proto-syntactic forms that the creator of the macro didn't anticipate. Consider an increment macro:

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

Furthermore consider the following (ab)use of the macro:

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

Clearly, the creator of the macro didn't expect anyone to use the macro with anything but an identifier, yet the user—perhaps someone used to a different syntax—applied it to a vector-dereferencing expression.

Second, the macro creator may make a mistake and abuse a syntactic construction:

```
(define-syntax where
  (syntax-rules (is)
    ((_ bdy lhs is rhs) (let ([rhs lhs]) bdy))))
```

Here the intention is to define a `where` macro, which could be used like this:

```
(where (+ x 1) y is 5)
```

Unfortunately, the right-hand side of the rewriting rule for `where` abuses 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 applying an erroneous function. In either case, the programmer receives an error message and needs to find the bug. The difference is, however, that many Scheme systems report the location of a safety violation for a run-time error and often allow the programmer to inspect the stack, which provides even more insight. In Chez Scheme [4], for example, the (ab)user of `++` receives the report that the syntax

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

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 [5], a sophisticated IDE that employs source code tracing and high-lighting to provide visual clues, a programmer receives difficult-to-decipher error messages. The (ab)use of `++` macro highlights the `vector` dereference and reports that some `set!` expression is ill-formed, which at least suggests that the error is in the use of `++`. In contrast, for the use of `where`, DrScheme highlights the `5` and suggests that `let` expected an identifier instead. This leaves the programmer with at most a hint that the macro definition contains an error. In this paper, we outline the design and implementation of a tool for catching these kinds of mistakes.

3. CONSTRAINING MACROS BY SHAPES

One way to tame the power of Scheme macros is to provide a type system that discovers errors before an implementation expands macros. In this section, we present such a type system, dubbed a shapesystem. The primary purpose of the type system is to assist macro programmers with the disovery of errors inside of macro definitions, but we also imagine that the users of macro libraries can employ the system to inspect their macro uses.

In the first subsection, we present the language of our model. In the second and third section, we gradually introduce the system of shapes. In the fourth section, we explain why a conventional type checking approach doesn't work. In the last subsection, we sketch the principles of our approach; the actual implementation is described in the next section.

3.1 The language

Figure 1 specifies the programming language of our model: the surface syntax and the core syntax. The surface syntax consists of a core syntax plus macro applications. The core syntax consists of definitions and expressions. The underlined portions of the figure indicate the parts of the language that belong to the surface syntax but not core syntax.

A program is a sequence of macro definitions followed by a sequence of forms in the surface syntax. Macro definitions use `syntax-laws`, a variant of Scheme's `syntax-rules`. More specifically, `syntax-laws` is a version of `syntax-rules` that accommodates shape annotations.

One restrictions of our model is that the set of primitive keywords, macro keywords, identifiers, and pattern variables are assumed to be disjoint subsets of Scheme's set of tags. This eliminates the possibility of `lambda`-bound variables shadowing global macros.

```
program   ::= macro-def* top-level*
top-level ::= def | expr
def       ::= (define id expr) | (macro . s-expr)
expr      ::= id | number | (expr expr*)
            | (lambda (id*) expr)
            | (lambda (id* . id) expr)
            | (quote s-expr) | (macro s-expr)
macro-def ::= (define-syntax macro
                (syntax-laws etype
                  ((_ . pattern) guards s-expr)*))
pattern   ::= pvar | () | (pattern pattern)
            | (pattern ... ())
guards    ::= ((pvar stype)*)
tag       ::= unspecified countable set
keyword   ::= lambda | define | quote
            | define-syntax | syntax-laws | ...
id        ::= subset of tag disjoint from macro
pvar      ::= subset of tag disjoint from macro, id
s-expr    ::= keyword | macro | id | pvar
            | number | () | (s-expr s-expr)
```

$$(x_1.(x_2.\ldots.())) \equiv (x_1 \ldots x_n)$$

Figure 1: Syntax

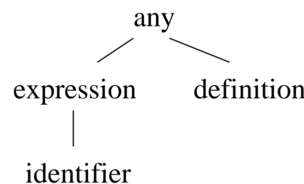

Figure 2: Base types

3.2 Base types and shape types

A close look at the grammar in figure 1 suggests that a macro programmer should know about four base types:

1. expression, which denotes the set of all core Scheme expressions;
2. definition, which denotes the set of all core Scheme definitions;
3. identifier, which denotes the set of all lexical identifiers; and
4. any, which denotes the set of all S-expressions.

The first three correspond to the basic syntactic categories of an ordinary Scheme program. The separation of identifiers from expressions is important so that we can deal with syntactic constructors such as `lambda` and `set!`, which require identifiers in specific positions rather than arbitrary expressions. Scheme's quoting sublanguage also requires the introduction of a distinguished type any so that we can describe the set of all arguments for `quote`.

The four base types are obviously related. Once we classify a tag as identifier, we can also use it as an expression and in a `quote` context. Similarly, a definition can also occur in a quoted context, but it cannot occur in lieu of an expression. Figure 2 summarizes the relationship between the four base types.

At first glance, the collection of base type may suffice to describe the type of a macro. As we know from the Scheme report, a macro's output is always an expression or a definition. This explains the specification of `etype`, which is the collection of range types for macro definitions. Concerning a macro's inputs, however, the Scheme report makes no restrictions. Following the precedence of type theory (for functional programming), we start with the idea that a macro programmer should specify types of the formal parameters, which are the pattern variables in each clause.

Take a look at this example:

```
(define-syntax simple-let
  (syntax-laws expression
    ((_ (var expr) body)
     ((var identifier)
      (expr expression)
      (body expression))
     ((lambda (var) body) expr))))
```

The macro definition introduce a simple form of `let` that binds one identifier to the value of one expression in some second expression. The (left-hand side) pattern therefore includes three pattern variables whose types are specified in the guard of our `syntax-laws` form. Still, it would be misleading to say that `simple-let` has a type like

$$\text{identifier} \quad \text{expression} \quad \text{expression} \rightarrow \text{expression}$$

because that would completely ignore that the macro use must group the `var` and the `expr` components so that they are visually distinct from `body`.

Put more generally, a macro programmer specifies the general shape of a macro's input with two components: the types of the pattern variables and the pattern of grouping parentheses. Since checking the use of macros is about checking the well-formedness of its subexpressions, the types for macros must take these parentheses into account. Based on these observations, we introduce `shape types`, or `shapes` for short, to describe the structure of the terms that macros consume. Shape types include the base types and construct compound types using pair types, the `null` type, case types, and arrow types. The latter two are only useful to describe an entire macro; we do not deal with macros as arguments in this paper.

Using shape types, we can specify the type of `simple-let` as

$$((\text{identifier} \;.\; (\text{expression} \;.\; ())) \;.\; (\text{expression} \;.\; ())) \rightarrow \text{expression}$$

```
etype ::= expression | definition
btype ::= etype | identifier | any
stype ::= btype | () | (stype . stype)
       |  stype → stype | (case stype*)
       |  (stype . . . stype)
```

Figure 3: Types

The choice of pairing as the basic constructor in the model represents our assumption that a macro application is a pair of the macro keyword and some S-expression.

3.3 Sequences

The type language described so far is not rich enough to describe macros like `and` and `let` and primitive syntax like `lambda`. These macros employ ellipses, because the core of Scheme allows programmers to write down arbitrarily long sequences of expressions without intervening visual markers. Since ellipses are an integral part of the pattern and template language of macro definitions, we extend our type language with the sequence shape type constructor.

Ellipses occur in two radically different ways in macros. In patterns, an ellipsis must always end a list. That is, `(a ...)` is a valid pattern, but `(a ... b)` is not. In templates, an ellipsis may be followed by any template. Thus, `(a ... b)` and `(a ... . b)` are both valid templates. To cover both cases, our shape type constructor for sequences handles the general case for templates. Ellipses in patterns are described by sequences whose final part is always `()`. Figure 3 shows the complete grammar of base types and shape types.

Using the full power of shape types, we can write down the types of macros such as `and`:

$$(\text{expression} \ldots . ()) \rightarrow \text{expression}$$

as well as that of core constructs such as `lambda`. The formal parameter list of `lambda` has the shape type

$$(\text{identifier} \ldots . (\texttt{case}\, () \ \text{identifier}))\ .$$

3.4 Structural type checking fails

A traditional type checker recursively traverses an abstract syntax tree and synthesizes the type of the tree from its leaves, using algebraic rules. That is, a type checker descends the tree until it reaches a leaf, for which some external agent (e.g., a type environment or a primitive type judgment) specify the type. For each internal node, it then synthesizes the type from the type of the subtrees.

This context-free traversal approach does not work for shape checking macros and macro uses. Consider the following excerpt from a Scheme program:

```
(add1 x)
```

Since `add1` and `x` are identifiers, one could easily mistake this S-expression for an expression. Suppose, however, that the S-expression appears as the formals part of a `lambda`:

```
(lambda (add1 x) y)
```

Based on this context, we really need to understand the original S-expression as a list of identifiers.

One idea for fixing this problem is traverse the tree and to identify each macro application. Then, it seems possible to check each macro application independently. Put differently, such a type checker would treat each macro application as atomic within the surrounding context and would use traditional type checking locally. Unfortunately, this approach doesn't work either, because macros may dissect their arguments in unforeseen ways. Take a look at the expression

```
(amacro (bmacro x))
```

Assume that `amacro` and `bmacro` are the markers for defined macros of one subexpression each. That is, the S-expression seems to consist of two macro applications. Hence the revised type checker would analyze `(bmacro x)` as a macro application, determining that its type is, say, expression. But take a look at the definition of `amacro`:

```
(define-syntax amacro
  (syntax-laws expression
    [(_ (anything a))
     ((anything any) (a identifier))
     (lambda (a) (quote anything))]))
```

The `amacro` "destroys" the `bmacro` application and instead uses the parts in unexpected ways. More generally, the type checker has thrown away too much information. To determine what to do with the syntax inside of the `amacro`-application, we must use information about `amacro`. Type checking must proceed in a context-sensitive manner.

For a final problem with the conventional type-checking approach, let us examine the `syntax-laws` definition of `let`, with unspecified types for the construct's body:

```
(define-syntax let
  (syntax-laws expression
    [(_ ((lhs rhs) ...) body0 body ...)
     ((lhs identifier)
      (rhs expression)
      (body0 ???)
      (body ???))
     ((lambda (lhs ...) body0 body ...) rhs ...)]))
```

Standard Scheme requires that `let`'s body consists of an arbitrarily long sequence of definitions and expressions, but at least one expression. Using the conventional pattern for `let`, we cannot express this constraint. The pattern `body0 body ...` says only that the sequence has to have at least a first element. We can overcome this problem in our model by using the full power of shape types:

```
(define-syntax let
```

$$\frac{}{\Gamma \vdash \texttt{id} : \textsf{identifier}}\ \textsc{Identifier} \qquad \frac{}{\Gamma \vdash \texttt{number} : \textsf{expression}}\ \textsc{Datum} \qquad \frac{}{\Gamma \vdash \texttt{s-expr} : \textsf{any}}\ \textsc{Any} \qquad \frac{}{\Gamma \vdash () : ()}\ \textsc{Null}$$

$$\frac{\Gamma \vdash x_1 : s_1 \qquad \Gamma \vdash x_2 : s_2}{\Gamma \vdash (x_1\ .\ x_2) : (s_1\ .\ s_2)}\ \textsc{Pair}$$

$$\frac{\Gamma(\textsf{macro}) = s}{\Gamma \vdash \texttt{macro} : s}\ \textsc{Special form} \qquad \frac{\Gamma \vdash x_1 : s_1 \qquad \Gamma \vdash x_2 : s_2}{\Gamma \vdash (x_1 \ldots\ .\ x_2) : (s_1 \ldots\ .\ s_2)}\ \textsc{Sequence} \qquad \frac{\Gamma(\textsf{pvar}) = s}{\Gamma \vdash \texttt{pvar} : s}\ \textsc{Pattern variable}$$

$$\Gamma_0 \vdash \texttt{define} : (\textsf{identifier}\ .\ (\textsf{expression}\ .\ ())) \rightarrow \textsf{definition}$$

$$\Gamma_0 \vdash \texttt{quote} : (\textsf{any}\ .\ ()) \rightarrow \textsf{expression}$$

$$\Gamma_0 \vdash \texttt{lambda} : ((\textsf{identifier} \ldots\ .\ (\texttt{case}\,\textsf{identifier}\ ()))\ .\ (\textsf{expression}\ .\ ())) \rightarrow \textsf{expression}$$

Figure 4: Shape types and programs, initial type environment

```
(syntax-laws ()
  [(_ ((lhs rhs) ...) . body)
   ((lhs identifier)
    (rhs expression)
    (body (definition ...
           expression ...
           expression)))
   ((lambda (lhs ...) body0 body ...) rhs ...)]))
```

This shows that type checking not only must proceed in an unusual context-sensitive manner but that it must also take into account general shapes.

3.5 Relating syntax and shape types

Type checking macros is a matter of verifying that the argument S-expression is below the shape type that describes the domain of the macro. To this end we must specify how patterns, templates, and ordinary top-level expressions give rise to shape types and how types relate to each other.

A macro's domain type is determined by the shape type of the patterns and the shape types of the pattern variables. Specifically, the type of a pattern is the shape type that results from replacing the pattern variables in the pattern with their (guard) types. The type of the entire macro is constructed from the set of patterns' shape types and the result type annotation. Figure 5 formalizes this relationship.

To type-check a top-level expression, our type checker constructs a shape type that describes the structure of the fragment. A pair in the fragment is represented by a pair type, a null by the null type, identifiers by identifier, bound macro keywords by their corresponding arrow types, numbers by expression, and everything else as any. Primitive special forms are treated exactly the same as macros. An initial type environment holds maps every special form to an arrow type.

To type-check a template, the type checker proceeds as for regular top-level expressions, except that it needs to deal with two complications. First, it needs to include pattern variables, which have the types specified in the guards for that clause. Second, it must cope with ellipses, which may appear in various forms and with fewer restrictions than in the pattern.

Figure 4 gives the rules for constructing types for regular program fragments and templates.

4. SHAPE CHECKING

Translating the ideas of the previous section into a working algorithm requires three steps. In this section, we describe these steps, that is, how to check a complete program, what to consider for the subtyping check, and how to implement the check.

4.1 Checking programs

A program shape-checks if its macro definition templates and program body respect the types of its macro applications. The checking of the entire program proceeds in three stages.

First, the type checker builds a type environment from the macro definitions. The type environment maps macro keywords to arrow types. The shape type of a macro is determined by its return type, its patterns, and its guards. The resulting type environment extends the initial type environment with the bindings created via $\vdash_{\mathcal{M}}$.

Second, the type checker verifies that each macro template produces the promised kind of result, assuming it is applied to the specified shapes. For the verification of a template, the global type environment is augmented with the guards in the containing clause.

Third, the type checker verifies that each top-level form in the program is well-shaped. Since a top-level form can be either an expression or a definition, the top-level form is checked against the shape type (`case` expression definition).

The first step has been described earlier. The third is a simple version of the second due to the macro's guards and the template's ellipses. Hence, the type checker turns the template or top-level form into its corresponding shape type and then determines whether the derived shape is a subtype of the expected shape.

$$\frac{\Gamma(\text{pvar}) = s}{\Gamma \vdash_{\mathcal{P}} \text{pvar} : s} \qquad \frac{}{\Gamma \vdash_{\mathcal{P}} () : ()} \qquad \frac{\Gamma \vdash_{\mathcal{P}} x_1 : s_1 \quad \Gamma \vdash_{\mathcal{P}} x_2 : s_2}{\Gamma \vdash_{\mathcal{P}} (x_1 \ . \ x_2) : (s_1 \ . \ s_2)} \qquad \frac{\Gamma \vdash_{\mathcal{P}} x : s}{\Gamma \vdash_{\mathcal{P}} (x \ldots \ . \ ()) : (s \ldots \ . \ ())}$$

$$\frac{G_i \vdash_{\mathcal{P}} P_i : s_i \qquad i \leq n}{\vdash_{\mathcal{M}} (\texttt{define-syntax}\, m\ (\texttt{syntax-laws}\, t\ ((_ \ . \ P_0)\ G_0\ T_0) \cdots (_ \ . \ P_n)\ G_n\ T_n)) \blacktriangleright (\texttt{case}\, s_0 \cdots\ s_n) \rightarrow t}$$

Figure 5: Shape types and macro definitions

Regular application
$(\text{expression} \ . \ (\text{expression} \ldots \ . \ ())) \leq \text{expression}$

Special form application
$$\frac{s' \leq s}{(s \rightarrow t \ . \ s') \leq t}$$

Figure 6: Shape type simplification

The subtype relation for shape types is the natural generalization of the subtype relation on base types to the shape types plus two additional subtyping rules (see figure 6).

4.2 Subtyping

Our algorithm generalizes Amadio and Cardelli's recursive subtyping algorithm using cyclicity tests [1]. It is not a plain structural recursion on the two types. Two issues complicate the algorithm. One arises from the way pair shapes and case shapes interact, and the other from sequence shapes.

It is useful to think of the right hand side of the comparison not as a single type, but as a set of possible choices. The set increases as different possibilities are introduced by case and sequence types. It is not sufficient to check whether the type on the left matches any one type in the set. It may be that the type on the left may be covered only by the combination of multiple types on the right.

The following two inequalities illustrate how `case` and `pair` types interact (a, b, and c are incomparable types):

$$(a \ . \ (\texttt{case}\, b\ c)) \quad \leq \quad (\texttt{case}(a \ . \ b)\ (a \ . \ c))$$
$$((\texttt{case}\, a\ b) \ . \ c) \quad \leq \quad (\texttt{case}(a \ . \ c)\ (b \ . \ c))$$

In the first case, we need to check both the `car` and the `cdr` of the pair on the left. The question is to which type on the right we need to compare them. Clearly, this inequality test should succeed, but if we divide the set and consider the `cdr` of each pair separately, the algorithm fails to verify the inequality, because $(\texttt{case}\, b\ c) \not\leq b$ and $(\texttt{case}\, b\ c) \not\leq c$. The second case is the dual of the first and shows that we cannot split the set when we test the `car` of a pair.

One solution seems to be to check the `car` of the left with the `car`s of all the pairs on the right, and to check the `cdr` of the left with the `cdr`s of all the pairs on the right. Unfortunately, that solution is unsound. It accepts the following bad inequality

$$(a \ . \ a) \leq (\texttt{case}(a \ . \ b)\ (b \ . \ a)) ,$$

because the `car` of the first option would match and the `cdr` of the second.

The correct solution is to match not a set of types on the right, but a set of states, where a state is either $\bullet$ (the initial matching context) or a type with a state as context. The state's context describes the state that is made available to the set of `cdr`s to match if the state's type is matched. The context is only extended when checking pairs. The example above becomes:

$$(a \ . \ a) \leq (\texttt{case}(a \ . \ b)^{\bullet}\ (b \ . \ a)^{\bullet})$$

Checking the `car` of the pair becomes

$$a \leq (\texttt{case}\, a^{b,\bullet}\ b^{a,\bullet})$$

The first state in the set matches and the second fails. So the `cdr` is matched:

$$a \leq b^{\bullet}$$

which fails as required.

The second complication arises because of sequences. When a sequence $(s_r \ldots \ . \ s_u)$ is encountered on either the left or the right, it is unfolded into $(\texttt{case}\, s_u\ (s_r \ . \ (s_r \ldots \ . \ s_u)))$. The algorithm relies on a trace accumulator to detect cycles in checking. The trace keeps track of what inequality checks are currently under consideration. For example, the call to check $(a \ldots \ . \ b) \leq (a \ldots \ . \ b)$ would unfold both sequences, check their base cases, check their `car`s, and then return to checking the same inequality. Since that combination of type and set of states is in the trace accumulator, a cycle has occurred and it is correct to succeed [1].

4.3 Shape checking at work

Recall the macro `++` from Section 2. The following is the macro written in our language:

```
(define-syntax ++
  (syntax-laws expression
    [(_ x)
     ([x identifier])
     (begin (set! x (add1 x)) x)]))
```

This gives `++` the following shape type: [3]

$$(\text{identifier} \ . \ ()) \rightarrow \text{expression}$$

In this context, the macro application

```
(++ (vector-ref v 0))
```

[3] We abbreviate $(\texttt{case}\, s)$ as s.

is invalid, because the shape checking algorithm cannot show that the shapes of ++'s input are below its input shape.

The most specific shape of the input is

((identifier . (identifier . (expression . ()))) . ())

This shape is not bbelow (identifier . ()), the input shape of the macro. Thus the macro application of ++ is flagged as erroneous instead of the set! special form application that it expands into.

Checking is also performed on the templates of a macro definition. In the following definition of where, lhs and rhs are mistakenly swapped in the template:

```
(define-syntax where
  (syntax-laws expression
    [(_ body lhs is rhs)
     ([body expression]
      [lhs identifier]
      [rhs expression])
     (let ([rhs lhs]) body)]))
```

In order to match the shape of the template with expression, the shape checker needs to prove expression $\leq$ identifier, to satisfy the input shape of let. Since this inequality is not true, shape checking fails for the template. The macro definition is therefore rejected, even without any uses of the where macro.

```
;; A TypeSymbol is one of
;; 'identifier, 'expression, 'definition, 'any

;; A Type is
;;   - (make-base-type TypeSymbol)
;;   - (make-null-type)
;;   - (make-pair-type Type Type)
;;   - (make-sequence-type Type Type)
;;   - (make-arrow-type Type Type)
;;   - (make-case-type [Type])
(define-struct base-type (symbol))
(define-struct null-type ())
(define-struct pair-type (car cdr))
(define-struct sequence-type (rep final))
(define-struct arrow-type (domain range))
(define-struct case-type (cases))

;; A State is
;;   - (make-done-state)
;;   - (make-state Type State)
(define-struct done-state ())
(define-struct state (type context))
```

Figure 7: Data definitions

4.4 Implementation

This section presents the algorithm that determines whether one shape type is a subtype of another. We start with the data definitions and follow with the interface procedures. The last part covers those procedures that perform the recursive traversals.

Figure 7 shows the data definitions. We represent shape types as structures and base types as symbols wrapped in a base-type structure.

The main function is the procedure subshape?, which constructs a state from the type on the right hand side of the inequality to be tested and calls check to conduct the actual comparison.

The subshape checking algorithm maintains the invariant that the set of states representing the right hand side never contains a type whose outermost type constructor is sequence or case. Sequences are unfolded to their final type and a pair of their repeated type and the sequence type. The variants of a case type are absorbed into the set of states. The procedure normalize is responsible for maintaining this invariant.

The check procedure consumes a trace, a type, and a list of states. It produces a list of states to be used to check the cdr of a pair, as described above. If check produces a list containing the done state, checking has succeeded. Otherwise, it returns the empty list to indicate failure.

The check procedure always first consults the trace to detect and escape from cycles. If no cycle is found, check calls to check/shape with all the shape types and check/base with only the base types. The union of the results is returned.

The unionmatch macro checks the value of an expression against the pattern of each clause. For each pattern which succeeds, it evaluates the body and returns the union of the results.

The procedure check/shape performs a straightforward case analysis on the composite shape types. A null type matches exactly null types. A pair type matches the car against the cars of all pairs in the state set. The function abstract-car takes the car of all pairs in the set and extends the matching context for each with that pair's cdr. A sequence type matches if both cases of its unfolded representation match. A case type matches if all of its variants match. Since we must return a set of states, we use union and intersection rather than simple or and and.

The procedure check/base verifies subtyping for basic types. Any type is under any. Two base types are compared using the simple subtype relation for base types. An expression can be formed by a pair of an expression and a sequence of expressions, and either expression or definition can be the result of the appropriate special form application.

The procedure check/macro checks a macro application, using the shape types of the macro keyword; normalize maintains the invariant stated above; and abstract-car takes the car of all pair types and extends the context of the resulting states.

```
;; subshape? : Type Type -> boolean
(define (subshape? lhs rhs)
  (ormap done-state?
    (check empty-trace lhs
           (normalize (make-state rhs (make-done-state))))))

;; check : Trace Type [State] -> [State]
(define (check trace lhs states)
  (or (trace-lookup trace lhs states)
      (let [(trace (extend-trace trace lhs states))
            (base-states (filter state/base-type? states))]
        (union*
         (cons
          (check/shape trace lhs states)
          (map (lambda (bstate) (check/base trace lhs bstate))
               bstates))))))
```

Figure 8: Driving procedures

```
;; check/shape : Trace Type [State] -> [State]
(define (check/shape trace lhs states)
  (unionmatch lhs
    [($ null-type)
     (union* (map (lambda (s) (if (null-type? (state-type s)) (succeed s) (fail))) states))]
    [($ pair-type lhs-car lhs-cdr)
     (let [(cdrstates (check trace lhs-car (abstract-car states)))]
       (check trace lhs-cdr cdrstates))]
    [($ sequence-type lhs-rep lhs-final)
     (intersect (check trace lhs-final states)
                (check trace (pair lhs-rep lhs) states))]
    [($ case-type lhs-cases)
     (intersect (map (lambda (lhs-case) (check trace lhs-case states)) lhs-cases))]))

;; check/base : Trace Type State -> [State]
(define (check/shape trace lhs base-state)
  (unionmatch (state-type base-state)
    [($ base-type 'any) (succeed base-state)]
    [($ base-type b)
     (if (and (base-type? lhs) (subtype? lhs (base-type b)))
         (succeed base-state)
         (fail))]
    [($ base-type 'expression)
     (check trace lhs
            (list (make-state
                   (pair-type expression (sequence-type expression (null-type)))
                   (state-context base-state))))]
    [($ base-type (or 'expression 'definition))
     (if (and (pair-type? lhs) (arrow-type? (pair-type-car lhs)))
         (check/macro trace lhs base-state)
         (fail))]))
```

Figure 9: check/shape and check/base

```
;; check/macro : Trace Type State -> [State]
(define (check/macro trace lhs base-state)
  (let [(macro (pair-type-car lhs))
        (argument (pair-type-cdr lhs))]
    (if (base-type-equal? (arrow-type-range macro) (state-type base-state))
        (check trace argument
               (make-state (arrow-type-domain macro) (state-context base-state)))
        (fail))))

;; abstract-car : [State] -> [State]
(define (abstract-car states)
  (union (map abstract-car/1 states)))

;; abstract-car/1 : State -> [State]
(define (abstract-car/1 state)
  (let [(type (state-type))]
    (if (pair-type? type)
        (normalize
         (make-state (pair-type-car type)
                     (make-state (pair-type-cdr type) (state-context state))))
        '())))

;; normalize : State -> [State]
(define (normalize state)
  (map (lambda (type) (make-state type (state-context state)))
       (normalize-type (state-type state))))

;; normalize-type : Type -> [Type]
(define (normalize-type type)
  (cond [(sequence-type? type)
         (cons (pair-type (sequence-type-rep type) type)
               (normalize-type (sequence-type-final type)))]
        [(case-type? type)
         (union (map normalize-type (case-type-cases type)))]
        [else (list type)]))
```

Figure 10: check/macro and auxiliary procedures

4.5 Some first experiences

We used an implementation of the algorithm described to check implementations of the special forms described as derived syntax in R^5RS [8]. The algorithm was extended to handle literals in patterns.

Defining these forms in `syntax-laws` poses two problems. First, we need to reformulate the definitions to be compatible with our system. Second, we need to write down shape types for each pattern variable.

The macro definitions for the derived syntax given in R^5RS were not compatible with our system. For example, the common shape

(definition. . . . (expression. . . . (expression . ())))

cannot be expressed with the idiomatic pattern

```
(body0 body ...)
```

used in R^5RS; there are no suitable annotations. To solve this problem, we rewrite `let` using a "dotted" pattern:

```
(define-syntax let
  (syntax-laws expression
    [(let ((name val) ...) . body)
     ([name identifier] [val expression]
      [body (definition ...
             expression ... expression)])
     ((lambda (name ...) . body) val ...)]))
```

Shape annotations for simple macros are as easy as type annotations in most languages. Complicated macros such as `cond`, however, have extremely verbose annotations due to the complexity of the syntax of `cond` clauses. We are exploring a method of naming or abbreviating shapes.

5. RELATED WORK, LIMITATIONS, AND FUTURE WORK

Cardelli, Matthes, and Abadi [2] study macros as extensible parsing. They superimpose enough discipline so that parser extensions don't violate the lexical structure of the program. They do not consider the question of whether macro applications are well-shaped but instead ensure that the grammar extension produces a well-defined grammar.

Ganz, Sabry, and Taha [7] present MacroML, a version of ML with an extremely simple form of macros. Their macros require the macro user to specify run-time values, syntax, and binding relationships at the place of macro use. In return, they can type check their macros with a tower of type systems. The type checker verifies that MacroML macros expand properly and that the code they produce type checks in ML. Unfortunately, none of their type-checking techniques for macros carry over to Scheme's macro system because of the simplicity of their assumptions.

We have extended our own work so that it applies to a large portion of Scheme's standard macro system. That is, we can rewrite and type check the macros from the Scheme report in the `syntax-law` notation. The expanded version also covers all core forms with two exceptions: `begin` and `quasiquote`. Still, our system imposes several restrictions on the macro writer, including the need for type declarations and the elimination of macro-arguments.

In the near future, we intend to investigate a soundness theorem for macro expansion similar to type soundness for functional languages. Specifically, macro expansion (in our model) should always produce well-formed syntax modulo context-sensitive constraints (e.g., `(lambda (x x) (+ x 1))`) and ellipsis mismatch. The latter is, of course, is analogous to type checking array lookups; the former is probably beyond the scope of a type discipline.

6. REFERENCES

[1] Amadio, R. and L. Cardelli. Subtyping recursive types. In ACM Transactions on Programming Languages and Systems, volume 15, pages 575–631, 1993.

[2] Cardelli, L., F. Matthes and M. Abadi. Extensible syntax with lexical scoping. Research Report 121, Digital SRC, 1994.

[3] Clinger, W. and J. Rees. Macros that work. In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 155–162, 1991.

[4] Dybvig, R. K. The Scheme Programming Language. Prentice-Hall, 1 edition, 1987.

[5] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. Journal of Functional Programming, 12(2):159–182, March 2002. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pp. 369–388.

[6] Flatt, M. Composable and compilable macros: You want it when? In ACM SIGPLAN International Conference on Functional Programming, 2002.

[7] Ganz, S. E., A. Sabry and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In International Conference on Functional Programming, pages 74–85, 2001.

[8] Kelsey, R., W. Clinger and J. Rees (Editors). Revised[5] report of the algorithmic language Scheme. ACM SIGPLAN Notices, 33(9):26–76, 1998.

[9] Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In ACM Symposium on Lisp and Functional Programming, pages 151–161, 1986.

[10] Kohlbecker, E. E. and M. Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 77–84, 1987.

[11] Queinnec, C. Macroexpansion reflective tower. In Kiczales, G., editor, Proceedings of the Reflection'96 Conference, pages 93–104, San Francisco (California, USA), 1996.