Comp 523 Final Report

Milton Rosenbaum
milton.rosenbaum@mail.mcgill.ca

Asha Basu
asha.basu@mail.mcgill.ca

Abstract

We implement a paper (**?**) that lays out a type system for a simplified Scheme-like language with basic macros. Traditional type systems do not guarantee that macro expansion will not get stuck or produce untypable code; they merely catch errors after a macro has expanded, in a fashion uninformative to the user. This paper details our implementation of Culepper and Felleisen's *shape types*, a new framework for reasoning about the types of macros, which ensures that macro evaluation does not get stuck (progress) and returns well-typed code (preservation), as proved in their paper.

Our code, along with some short examples and unit tests, can be found at https://github.com/milrosen/comp523project

1 Introduction

1.1 Motivation & History

Macros are a very useful construct. If a programmer wishes to repeatedly perform some task on a certain syntactic pattern of code, rather than having to manually do so, they can simply code a macro to do it for them in some predetermined fashion. A macro might in general take in syntax of a certain *pattern* – that is, following some designated format (for example, a pair where the first element is itself a pair) and operate on it in some essentially arbitrary fashion; such macros can be defined and then used anywhere in code, and the interpreter will expand any macro it runs across, applying it to the subsequent code as its input format and replacing the two with the result of this application.

While macros in the sense of replacing text with other text in strings may date back further, Lisp-style macros that operate on abstract syntax trees date back to the 1960s, and have been used in many programming languages since then. However, such macros can often go awry. Using the paper's example, a programmer might define a macro meant to increment a number that, given some input, sets it to itself plus one; then, if the macro were applied to something other than a number, the program would return a type error when attempting to add one to something other than a number.

While errors due to misuse are of course a problem with any programming construct, the paper attempts to ameliorate this problem by combining shape checking and parsing. Normally, when such an error occurs, it would not be detected during macro expansion, as macros are not typed; then when detected during evaluation, the programmer will only receive a type error that addition was performed on a non-numerical input, rather than an error relating to what actually went wrong: the macro application. That is, in practice, we do not guarantee anything about the content of our variables, but the shape of the code that defines them.

The paper thus introduces a novel type system for macros themselves and macro expansion, which we implement. Actually typing the entirety of Scheme would be a herculean task, so the authors instead define a simplified, bare-bones version of Scheme with all the features truly essential to macros, and develop a type system for this programming language. We therefore implement this type system, testing both that it does what it intends to do and that it can be implemented at all, making sure that the inference rules are in fact syntax-directed.

1.2 Overview

This paper explains the simplified programming language in (**?**) in the context of our implementation of it. We go over the paper's description of type checking, specifying our algorithmic interpretation of their rules. Before doing so, we go over some preliminaries, including the main aims of this type system. We assign *shapes*, a specification of the syntactic *form* of an expression program rather than the potential values of each variable, to each

part of our program, allowing us to check if macros and keywords alike receive the correct inputs before they are expanded. We then go over each of the paper's judgments one by one.

First, there is the program correctness judgment, which represents the verification that a given program with some macros and expressions will run without type errors. Then there is the 'respects' relation $\vdash_{\mathcal{R}}$, which states that the macro environment – a list of shapes that the type-checker has determined the macros to be – does not contradict the definitions of the macros. We next outline what it means for any one macro's patterns and guards (from which its shape arises) to be respected by the macro environment. To this end, we then detail two judgments: the *unguarded* judgment $\vdash_{\mathcal{U}}$ which details the shape of the unguarded pattern in a macro's definition, and then *guarded* judgment $\vdash_{\mathcal{G}}$ which details the shape of the guards in a macro's definition.

We then go over some more obvious judgments. One important judgment is the *ordering* judgment $\sqsubseteq$, which checks when one shape is a more general version of another, analogous to subtyping. This is actually when the analog of the application rule for shape typing occurs: a macro applied to a shape is simply a subshape of the macro's output type. Another relatively simple judgment is the *overlaps* judgment $\bowtie$, which determines whether two shapes share a possible input. Then there is the *template* judgment $\vdash_{\mathcal{T}}$, which checks that a macro's output type matches its type annotation when its guard conditions are met. Then finally, there are the central judgments of the code: shape-checking ($\blacktriangleright$) and type-checking ($\circ$), which respectively generate the shape of a macro and type-check an expression against a type.

While this is all outlined in the paper, we explain how each judgment was implemented and how they relate to each other. That way, each judgment clearly is relevant to type-checking a macro rather than a series of unmotivated inference rules. We can then conclude the paper by summarizing the overall process. We also detail various challenges with implementing the paper and some possible extensions.

2 Developments

2.1 Preliminaries

The process of shape/type checking fulfills the same role as parsing in other languages. As such, it operates on an entirely unannotated abstract syntax tree. In our implementation, the AST is a list containing either symbols or smaller trees (ie lists). Instead of making judgments based on the runtime contents of our variables, we are interested in proving properties about the shape of the expressions themselves. As such, we only have two primitive types, **expr** and **def**. These correspond to the valid shapes of forms in our toplevel.

In addition to the two primitive types, this paper also introduces complex shapes. These could be lists of other shapes, primitive types, identifiers, or arrows. The arrow shape is similar to a function type, and in the case of a macro, it does represent a transformation from one shape to another. However, arrow shapes are also powerful enough to represent the primitive keywords in our language. As such, we could give the keyword **lambda** the shape **(ident) expr** $\rightarrow$ **expr**, since the use of the keyword is valid if and only if it is followed by an identifier in parentheses and then a valid expression.

In this language, a macro definition is just a specification of the kinds of shapes that they expect to receive. These shapes are specified in clauses. For instance, the **let** macro would have a single clause, with the pattern **let (x y) z**. In addition to a pattern, each clause needs a guard, specifying the shapes of each variable in the pattern. In the case of **let**, the guard would specify that the shape of the pattern is **(ident expr) expr**.

Finally, a macro contains a template that determines the expansion that would take place after the clause is applied. Since our language has **lambda** as a primitive keyword, the template of the **let** macro would look like (**lambda** (**x**) **z**) **y**.

2.2 Implementations of Each Judgment

Correctness The primary judgment in the paper is correctness. Unlike a standard typing judgment, where each rule can be used at any point in a derivation tree, correctness is only ever used once, and its premises are just a list of the other judgments the code must satisfy.

$$
\begin{array}{l}
md_i := \\
(\textbf{define-syntax}\ m_i \\
\quad (\textbf{syntax-laws}\ t_i\ (P_{i,1} G_{i,1} T_{i,1}) \cdots))
\end{array}
$$

$$
\frac{\begin{array}{c}\rho(m_i) = ([(P_{i,j}, G_{i,j}, T_{i,j}), \cdots], t_i) \\ \Gamma \vdash_{\mathcal{R}} \rho \\ \Gamma \cup \Gamma_0; \Phi \vdash x_k \blacktriangleright t_k \quad t_k \in type\end{array}}{\vdash_{\mathcal{P}} md_1 \cdots md_n\ x_1 \cdots x_m\ \textbf{correct}}
$$

As such, it is the entry point into our code. It is called on an abstract syntax tree. It specifies that a correct program begins with some number of macro definitions. Each must be composed of a name m_i, an output type t_i, and then a list of clauses, each comprised of a pattern $P_{i,j}$, guard $G_{i,j}$ and template $T_{i,j}$. Once we have ensured that the code actually follows this format, we assign the list of patterns, guards, and templates to the name of the macro in the expansion environment $\rho(m_i) = \ldots$. Once we have associated the macro with the information needed to perform the eventual expansion, we ensure that it respects ($\vdash_{\mathcal{R}}$) the given environment.

Once the shapes of our macros are generated, they are added to the macro context Γ. The context extended with Γ_0, containing the shapes of keywords, and Φ, containing the shapes of pattern variables. Finally, these extended contexts are used to check that every toplevel form has a definite type. If macro-shape generation, template checking, and toplevel checking is successful, then the entire program is judged correct.

Respects The respects judgment codifies when a macro definition is well-formed, and is also responsible for adding well-defined macros to the context for later checks.

$$
\frac{\begin{array}{c}\forall m \in \textbf{dom}(\Gamma) \cap macro : \Gamma(m) \vdash_{\mathcal{R}} \rho(m) \\ \forall m \in \textbf{dom}(\rho) : \Gamma \vdash_{\mathcal{T}} \rho(m)\end{array}}{\Gamma \vdash_{\mathcal{R}} \rho}
$$

The first premise is responsible for generating the context Γ, which maps macros to their shape types. The second premise checks this context against the template code that our macros generate. This also mirrors the two key ways that a macro definition can be malformed: the patterns overlap, or it generates poorly shaped code.

Macro Type Before we can check whether a macro is used correctly, we must define what it means to use that macro correctly. The macro type rule is responsible for turning the pattern and guard clauses into the shape of the macros.

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

Each macro and keyword are always an arrow shape since they expect code of a certain shape and return code of a different shape. The input shape for each macro will always be **mclauses**. This shape only ever appears as an input to a macro. It will throw an error if it appears anywhere other than the first element of a form.

Each u_i, s_i pair in the shape corresponds to one of the guard clauses and specifies the shape that the clause is expecting to match. As such, the **mclauses** shape acts like a logical or, and is a supershape of any shape which is a subshape of one of its clauses. The guarded ($\vdash_{\mathcal{G}}$) and unguarded ($\vdash_{\mathcal{U}}$) rules are responsible for generating the shapes of each clause. Note that they are both being called on the same pattern, but that only the guarded rule has access to the type specifications in the guards.

After generating both guarded and unguarded shapes for each pattern, the final check ensures that there are no overlaps between any of the unguarded clauses. Since we don't have access to typing information during expansion, we need to be sure that the correct clause can be determined even from an untyped expression.

Guarded These are responsible for generating the shape of each macro. Note how the G_i was on the left of the turnstile in the previous rule, where the Φ is now. Recall that Φ is the context that maps variables to types. The two judgments outline the recursive and base case of a function. In a given pattern we either have a list of forms, $x_1 \ldots x_n$, or a single pattern variable p. In the case of a list, we just recurse on each element of the list and look up single symbols in the $pvar$ context, Φ. This judgment is also responsible for throwing an error if unbound variables appear in the pattern.

$$\text{Guarded1}\quad \frac{p \in pvar \qquad \Phi(p) = s}{\Phi \vdash_{\mathcal{G}} p : s}$$

$$\text{Guarded2}\quad \frac{\forall i \leq n : \Gamma; \Phi \vdash_{\mathcal{G}} x_i : s_i}{\Phi \vdash_{\mathcal{G}} (x_1 \cdots x_n) : (s_1 \cdots s_n)}$$

Unguarded These judgments are identical to the guard clauses, but instead of receiving a context, they map each symbol to the **any** shape.

$$\text{Unguarded1}\quad \frac{p \in pvar}{\vdash_{\mathcal{U}} pvar : \textbf{any}}$$

$$\text{Unguarded2}\quad \frac{\forall i \leq n : \ \vdash_{\mathcal{U}} x_i : u_i}{\vdash_{\mathcal{U}} (x_1 \cdots x_n) : (u_1 \cdots u_n)}$$

Ordering For our shape type system to be useful, we need to have an ordering on shapes. For instance, if a macro expects an **expr** in its first argument, it would be strange if we rejected (**x y**) for being an (**ident ident**), since every application of an identifier to another identifier is also an expression.

This judgment is quite complex and is presented in natural deduction form in the paper. In the interest of time and space, I believe it is easier to write pseudocode directly.

$$\begin{aligned}
_ &\sqsubseteq \textbf{any} && \rightarrow true \\
Type\ t_1 &\sqsubseteq Type\ t_2 && \rightarrow t_1 = t_2 \\
\textbf{ident} &\sqsubseteq \textbf{expr} && \rightarrow true \\
(s_1 \rightarrow t_1)\ s_2 &\sqsubseteq Type\ t_2 && \rightarrow t_1 = t_2\ \&\ s_2 \sqsubseteq s_1 \\
(s_1 \cdots s_n) &\sqsubseteq (s'_1 \cdots s'_n) && \rightarrow \forall i \leq n, s_i \sqsubseteq s'_i \\
(s_1 \cdots s_n) &\sqsubseteq \textbf{expr} && \rightarrow n \geq 2\ \&\ s_i \sqsubseteq \textbf{expr} \\
s &\sqsubseteq \textbf{mclauses} && (s_1, u_1) \cdots (s_n, u_n) \\
& && \rightarrow \exists i \leq n, s \sqsubseteq s_i \\
s_1 &\sqsubseteq s_2 && \rightarrow s_1 = s_2
\end{aligned}$$

As a series of inference rules, the authors of the paper were able to simply add transitivity. However, this would no longer be syntax-directed, since we would effectively need to generate the middle shape from nowhere. However, it is not hard to verify that these rules are transitive.

It's also worth noting that this subshape judgment is by far the most important one in the paper. The arrow type rule, for instance, is where we ensure that the application of a macro or keyword is correct, as this doesn't actually happen in the typing judgment itself. Furthermore, the interaction of the arrow shape rule and the **mclauses** case is why the **mclauses** shape can represent the multiple possible clauses of a macro.

Overlaps In the paper, the overlap rule, $u \bowtie u'$ is presented with its own inference rules. These rules are redundant, since they are identical to $u \sqsubseteq u' \vee u' \sqsubseteq u$, and we already have the unguarded rule which generates the type of a pattern by assigning **any** to each symbol.

Ensuring that no two patterns overlap in a macro imposes a substantial restriction on well-typed macros since **any** always overlaps a list of any length. So, a macro with clauses for (**ident ident**) and **ident** would be forbidden, since (**any any**) $\sqsubseteq$ **any**. As such, it is impossible to create a series of similar clauses arranged from most specific to least specific, a very common design pattern for match-based control flow. However, the proof of progress and preservation requires that matching a clause is uniquely determined by the shape of untyped code.

Templates After building our macro context, we check if the templates themselves are well-shaped. Interestingly, splitting the type checking into generating typing information for each macro and then typing the bodies of each macro means that we gain recursive macros for free, since we add each macro to Γ as soon as we know its clauses are well typed.

Shapes and Types In the paper, the shape and type checker are split into two different judgments, $\Gamma; \Phi \vdash x \blacktriangleright t$, which checks if x has type t, and $\Gamma; \Phi \vdash x \circ s$, which generates the shape of x.

Since we generate the shape of an expression after we have finished creating our context, the function itself is very simple. It simply recurses until it reaches a lone symbol. If that symbol is a string or integer literal it returns the **expr** shape. If the symbol is defined in the context (either Γ or Φ), it returns the shape of that symbol. Finally, if the symbol is neither defined nor a literal, we return **ident**. As such, the expression **lambda** (**x**)(**+ x 1**) will be assigned the shape ((**ident**) **expr** $\rightarrow$ **expr**) (**ident**) (**ident ident expr**).

Unlike the typing judgments presented in class, this judgment never modifies the context, every $\Gamma; \Phi$ pair is the same in the premises and conclusion, it also does not contain a rule for macro application, which is handled in the shape ordering.

After generating a shape for each piece of code, we ensure that each toplevel form in our program has a primitive type and not just a shape. In the paper, this typing judgment is not recursive, in each case, it simply asks for the shape generated by the shaping rule, and then checks if that shape is a subshape of **expr** or **def**. This will be the case unless the code contains an incorrectly applied macro or there is a macro that generates a definition or expression when we were expecting the opposite.

In the lambda example from earlier, we are interested in checking if $\Gamma; \Phi \vdash$ **lambda** (**x**) (**+ x 1**) $\blacktriangleright$ **expr**. First, we check that the first symbol in our expression is a macro or keyword. Since it is, we look up the symbol in our context, and see that $\Gamma($**lambda**$)$ = (**ident**) **expr** $\rightarrow$ **expr**. Next, we check that the return type of our arrow shape is what our $\blacktriangleright$ expects. Then we check that the input shapes match, in this case, that (**ident**) (**ident ident expr**) $\sqsubseteq$ (**ident**) **expr**. Since the two shapes are a list of the same length, and neither are a macro application, we check if the lists are pairwise subshapes of each other. In order, we see that (**ident**) $\sqsubseteq$ (**ident**), and (**ident ident expr**) $\sqsubseteq$ **expr**. The last check is true since the list on the left is longer than two and each element is individually a subshape of **expr**.

2.3 Implementation of Expansion

Once we have a piece of code that type checks, we know that every macro produces code of a valid shape, and that every piece macro is only applied to code that it can work with. As such, if we were to expand our macros, we would never reach a stuck state.

Our overlaps relation ensured that each clause had a unique match without requiring any shape information. We simply reconstruct the unguarded shape of whatever code the macro is being applied to, and find the first guard clause which is a supershape of the unguarded interpretation. From there, we match up the pattern and symbolic expression to generate a substitution, perform that substitution on the template, and then expand the post-substitution template, since our macros may be recursive.

2.4 Challenges

There were two major challenges in implementing this paper. First, the judgments, as presented, give no real indication of their role in computation. Second, the only example of a macro written in the language of the paper uses a significantly more complex shape system. The paper does not explain how the previous judgments needed to be extended to use the advanced typing features that they require.

For each judgment, the process of reconstructing the correct computational interpretation was an exercise in unprincipled guesswork. For instance, we interpreted the shaping judgment as generating the least possible shape of a given piece of code. However, this is not found anywhere in the paper, in fact, the shaping judgment contains a conversion rule based on the subshape relation, which makes it look much more like a traditional derivation tree than it is in our interpretation.

Further confusing this is the fact that the shape ordering is already transitive and contains the rule for applications. Indeed, we never check if a shape on the left of the circle is syntactically equal to another shape, only that it is a sub-shape of the type or macro argument that we are checking. As such, both rules are defined with transitivity independently.

Similar challenges to these popped up in almost every step of our implementation. We found it difficult to determine whether rules added variables to a context or checked membership in the context. Usually, this is disambiguated by writing $\Gamma, x : A \vdash M : C$ when x is added, and $\Gamma(x) = A$ when x is being checked, however, the first format never appears in the paper, and both additions and checks were seemingly written in the later format.

In addition to a lack of clarity on the role of each rule, the order of operations was also very difficult to determine. For instance, it seems reasonable to assume that type and shape checking should occur before expansion, and, indeed, the result of the correctness judgment is a program that contains macro definitions, and so must not have been expanded. However, the authors of the paper define a keyword, **app**, which can only appear after expansion, and indicates that an s-expr has been tagged as a procedure application rather than a macro application. Inexplicably, this keyword, which should never appear during type checking, is added to the initial context in which the program is checked.

Furthermore, the correctness judgment includes $\Gamma \cup \Gamma_0; \Phi \vdash x_k \blacktriangleright t_k \quad t_k \in type$ as a premise. The Φ represents a context mapping pattern variables to shapes. Since $pvars$ should never appear outside of the template of a macro, it is unclear how to interpret this. Nevertheless, Φ appears. In other

situations where a context should be empty, the authors use a $\cdot$ or just a space, so, it seems to be the case that there should be a non-empty pattern variable context for our toplevel forms, how this context should be defined is left undetermined by the paper, further compounding our confusion on the role of each rule.

The second batch of challenges crop up when their paper is compared to their example, the **cond** macro. It turns an expression to be evaluated and a list of (premise, result) pairs and transforms it into a nested if expression which will return the first expression whose predicate is true. To define this macro in a type-safe way, the authors introduce three extensions to their shape type system. First, the ability to define keywords – shape types that are inhabited by a single symbol – like **else** or **=>**. Second, union shapes, which should be supershapes of any element in their union. Finally, they introduce arbitrary-length lists which should match either **nil** or (**cons** s (**list** s)).

While it is easy to check when a union shape is a subshape of a non-union shape, it is much more difficult to check that a set of macro clauses exhaust the union shape, especially considering that union shapes may contain the list shape or other union shapes. However, since the authors apply macros directly to union shapes in their code, this is a must if we want to have preservation and progress.

This is much more complicated than any of the cases presented in the paper, and no guidance is presented on how it could be achieved. Further, operating on lists of arbitrary length is essential for defining key lisp macros, such as **(let* [x (expr) ... z (expr)] in expr)**, which binds any number of identifiers to the result of evaluating the expression on their right. It should only ever shape-check on applications to lists of **ident (expr)** pairs. This means that there would need to be at least two clauses in the definition of let, one which matches **nil** case of the list, and one which matches the **cons** case. These two clauses would be defined separately since they match on different shapes. In the body of the **cons** case, we would need to apply the macro on the shorter list. For this application to be correct, we would need some way of checking that the macro clauses cover both the **nil** and **cons** cases of the list shape.

However, as defined, an application of a macro is valid iff exactly one of the cases matches the given input shape. Obviously, there is no one clause in the let macro that could match both the **nil** and **cons** case, since they have different shapes, meaning that the judgment would need to be extended somehow. This extension would be complicated, since it would need to determine when macro clauses exhaust an arbitrary shape, which may contain deeply nested unions and lists. The paper does not indicate what is involved in implementing these essential extensions.

3 Conclusion

In the paper, Culpepper and Felleisen discuss many extensions that could be implemented, chiefly including sequences of arbitrary length, without providing any concrete way to do so. Thus the type system they detail is a long way from being able to fully match Scheme macros, even though they claim that, in practice, most Scheme code can be written under the simplifying constraints that this type-checker assumes. Thus the most obvious avenue of future work would be to figure out how to implement these extensions. Additionally, their paper omits hygiene and referential transparency; implementing those would also be a useful extension that would make this type-system closer to be useful for Scheme as it is actually used.

Furthermore, this is not the only line of research into type systems for macros. While this specific type system has not been built on very much in the literature, there are many diverse ways of attempting to type-check macros, some by the author of this paper himself. Ideally, future research into this topic would also attempt to implement some other such proposals, perhaps with an attempt to capture more functionality than is in this paper's simplified programming language.

To recap, this paper implements an already-existing type system for macros outlined in (**?**). To do so, it first goes through what it is the paper is trying to do, outlining the broad idea. It then explains the paper's various judgments in its type system, and explains what they mean and how they are implemented in the corresponding code. Afterward, it ties them together under one unified explanation for what this type-checking algorithm is fundamentally doing, and concludes with an outline of some of the challenges faces and possible extensions.