A Theory of Typed Hygienic Macros

A dissertation presented
by

David Herman

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts
May, 2010

Dedication

In memory of Waldo and Bubba

1991–2008 and 2001–2009

Rest in peace, little friends.

Acknowledgments

Working with Mitch Wand has been a rare privilege. I am grateful for all he has taught me and for the opportunity to work for a man I admire.

A great deal of my education took place in the extraordinary Programming Research Laboratory on the third floor of West Village H. In particular, I find it hard to imagine ever finishing this work without the perspective, insight, and friendship of Richard Cobbe, Ryan Culpepper, Felix Klock, Sam Tobin-Hochstadt, and David Van Horn. I gratefully acknowledge as well my PRL colleagues and friends Dan Brown, John Clements, Carl Eastlund, Vassilis Koutavas, Greg Pettyjohn, Aaron Turon, and Dimitris Vardoulakis. And I thank the professors who built and fostered the PRL research culture.

For their mentorship, I thank Olivier Danvy, Matthias Felleisen, Cormac Flanagan, Shriram Krishnamurthi, and the members of my thesis committee: William Clinger, Oleg Kiselyov, and Olin Shivers.

Brendan Eich and the Mozilla Corporation generously supported my research for years. I especially appreciate their patience in these final months. I also thank Larry Finkelstein and Agnes Chan for their support.

I thank my family for their endless love and encouragement.

And to Lisa: thank you for being there all along. I love you with all my heart.

Abstract

We present the λ_m-calculus, a semantics for a language of hygienic macros with a non-trivial theory. Unlike Scheme, where programs must be macro-expanded to be analyzed, our semantics admits reasoning about programs *as they appear to programmers*. Our contributions include a semantics of hygienic macro expansion, a formal definition of α-equivalence that is independent of expansion, and a proof that expansion preserves α-equivalence. The key technical component of our language is a type system similar to Culpepper and Felleisen's "shape types," but with the novel contribution of *binding signature types,* which specify the bindings and scope of a macro's arguments.

Preface

This dissertation investigates the formal semantics of hygienic macros and presents the λ_m-calculus, a model of hygienic macro expansion. The model does not describe a novel macro language, but is rather intended to shed light on the behavior of hygienic macro systems in the tradition of the Scheme programming language [14, 40, 61].

What this work does introduce is a novel logical system for reasoning about the behavior of macros. This system allows us to express and validate formal properties of hygienic expansion. The aim of the dissertation is to provide firmer theoretical foundations for characterizing hygiene, which might inform the design of future macro systems.

Background material

This work brings together technical material from several different areas of programming languages research. It would be impractical to attempt to provide an adequate background for all of these topics within one dissertation. The reader may find it useful to have at least a moderate level of familiarity with the following topics:

- hygienic macros and **syntax-rules**

  The formalism of the λ_m-calculus concerns the semantics of hygienic macros. The reader should at least be familiar with the basic concepts of macros and macro expansion. A deep understanding of hygienic expansion is not required, but some familiarity with at least one Scheme system is helpful.

All the macros of this dissertation are written in the style of Scheme's **syntax-rules** form. For an excellent introduction to Scheme macros, including macros written with **syntax-rules**, see Chapter 8 of Dybvig's *The Scheme Programming Language* [22].

- operational semantics

  The reader should be comfortable with operational models of programming languages, particularly small-step operational semantics. The λ_m-calculus is expressed as a reduction semantics with evaluation contexts [23], although familiarity with any operational techniques should probably be sufficient to follow most of the material. Part I of *Semantics Engineering with PLT Redex* [24] contains an introduction to this topic.

- basic type theory

  A key element of the development of the λ_m-calculus is a sound static type system. The reader should be comfortable with types as proof systems. Pierce's textbook *Types and Programming Languages* [51] is a good introductory resource. Chapter 21 is particularly helpful for understanding the recursive types of the λ_m type system.

The Scheme family of languages

The Scheme community often describes Scheme as a *family* of programming languages. As a standard, Scheme has gone through six revisions. And in practice, there are numerous, dramatically differing programming systems that can all plausibly lay claim to the name "Scheme."

For our purposes, the distinctions between these systems are generally irrelevant. This dissertation aims to model a small core semantics of hygienic macros, written in the so-called “high level” style of **syntax-rules**. Most macro systems designed for Scheme either support just the macros expressible with **syntax-rules** or provide more general systems that can nonetheless express **syntax-rules** as a derived form. Our goal is not to model a wide spectrum of macro language features, but rather to identify a subset that is just expressive enough to illustrate the challenges of specifying the properties of hygienic expansion.

Notational conventions

Throughout this dissertation we use the notation $\overline{x}$ to represent sequences. Depending on the context, sequences may be considered as a shorthand for whitespace-separated lists:

$$\overline{x_i}^{1..n} \stackrel{\text{def}}{=} x_1\ x_2\ \cdots\ x_n$$

or for comma-separate lists:

$$\overline{s_i}^{1..n} \stackrel{\text{def}}{=} x_1, x_2, \cdots, x_n$$

or for “cons-lists”:

$$\overline{x_i}^{1..n} \stackrel{\text{def}}{=} x_n :: \cdots :: x_2 :: x_1 :: \varepsilon$$

We represent the empty sequence with the special symbol ε. We typically elide the bound n where it can be inferred.

When representing Scheme syntax, we use bolded parentheses to represent a pair:

$$\mathbf{(}\mathit{term}_1\ \mathbf{.}\ \mathit{term}_2\mathbf{)}$$

This helps distinguish the semantically significant parentheses of Scheme syntax from the disambiguiating parentheses of traditional mathematical notation. We use the Lisp tradition of representing nested sequences of pair terms with the shorthand S-expression sequence notation:

$$\mathbf{(}\overline{\mathit{term}_i}\mathbf{)} \stackrel{\text{def}}{=} \mathbf{(}\mathit{term}_1\ \mathbf{.}\ \mathbf{(}\mathit{term}_2\ \mathbf{.}\ \mathbf{(}\cdots\ \mathbf{.}\ \mathbf{(}\mathit{term}_n\ \mathbf{.}\ \mathbf{()))))}$$

We use the special symbol ι to represent the identity function.

We represent finite tables as sets of pairs $x \mapsto y$. The notation $S(x)$ denotes table lookup; $S[x \mapsto y]$ denotes functional update.

For a partial function f, we write $f(x) \Downarrow$ to denote that $f(x)$ is defined.

Contents

List of Figures

CHAPTER 1

Hygienic Macro Expansion

Hygienic macro expansion is one of the crown jewels of Scheme, but to this day nobody understands just exactly what it is.

This dissertation demonstrates that hygienic macro expansion can be given a precise definition, with useful formal properties, by explicitly specifying the shape and binding structure of macros. In due course we shall understand better what these specifications, definitions, and properties look like. For now, we begin with an introduction to hygienic macro expansion by examples. We discuss where informal intuitions fail us, illustrating why a precise definition of hygiene has been so elusive.

1.1 The power of syntactic abstraction

The Lisp family is unique in the power of its tools for *syntactic abstraction*. The surface representation of programs as *S-expressions*, a simple and regular notation for trees of symbolic data, makes it convenient to manipulate program fragments as data structures. But the true power of syntactic abstraction comes from *macros*: compile-time language extensions defined as lexically embedded syntax transformations. With macros, Lisp and Scheme programmers synthesize new syntactic constructs that encapsulate common language idioms that are otherwise hard or impossible to abstract.

Macros provide an unusual level of extensibility in programming languages. Lisp and Scheme programmers put macros to use in breathtaking ways:

- domain-specific languages

  Macros facilitate the design philosophy of "little languages" [8]: creating a general solution to a class of programming patterns by encoding them as a custom programming language—often embedded within a general-purpose host language [35]. While in most programming languages, little languages are implemented with interpreters, macros allow embedded domain-specific languages to be compiled.

- extensible compilers

  Macros provide language support for extending the compilation toolchain without the usual mess of additional build machinery. For example, the `parser-tools` library of Owens et al. [48] provides a complete suite of parsing tools, with compile-time generation of LALR parsing tables, as a *library* rather than a stand-alone program. The engineering benefit for the client is clear: they get the benefit of compile-time code generation with none of the cost of complicated build processes.

- tiered language architecture

  Scheme's extensibility complements the parsimony of its core. Scheme combines macros with very general native programming constructs such as **lambda** and *call/cc* [61]. As a result, constructs that would ordinarily require native language support can be relegated to libraries. This layered approach leads simpler and more modular semantics, as well as simpler and more modular language implementations.

- declarative data structure initializers

  Whereas most programming languages can only support a fixed set of syntaxes for literal data, Scheme programmers can invent new literal syntax with complicated initialization protocols abstracted away by macros.

- custom static forms

  Traditional programming languages typically provide a fixed set of "second-class" language forms, such as declarations. Macros make it possible to abstract over these forms in ways that are often impossible to achieve otherwise.

- custom control-flow operators

  Because macros dispatch at compile-time, they can rearrange expressions to modify their control flow. This makes it possible to abstract over the flow of control without resorting to explicit higher-order constructs like **lambda**—in other words, to synthesize derived control-flow operators.

- custom binding forms

  Similarly, macros allow the synthesis of derived binding forms other than **lambda**.

The most distinctive characteristic of Scheme macros is *hygiene*. Sadly, hygienic macros have long resisted a concise, formal definition. Our incomplete understanding of hygienic macros makes them difficult to explain succinctly and accurately. Let us begin, then, by describing the problems with traditional macro expansion that hygienic expansion was invented to address.

1.2 Naïve macro expansion

The semantics of naïve or "unhygienic" macro expansion is easy to understand. Programs are simply represented as S-expressions. At every macro call, the S-expression of the macro call is replaced by the S-expression on the right-hand side of the macro definition. With pattern-matching macros, such as those written with `#define` in the C preprocessor [36] or **syntax-rules** in Scheme, variables in the macro's pattern are replaced by their corresponding subterms in the macro application.

For example, consider a simple macro for swapping two variables:

Example 1

```
(define-syntax swap!
  (syntax-rules ()
    [(swap! x y)
     (let ([z x])
       (set! x y)
       (set! y z))]))
```

When a use of **swap!** occurs in a program:

Example 2

```
(let ([a 1]
      [b 2])
  (swap! a b)
  (cons a b))
```

it is replaced by the right-hand side of the definition of **swap!**:

Example 3

```
(let ([a 1]
      [b 2])
  (let ([z a])
    (set! a y)
    (set! b z))
  (cons a b))
```

Notice that the macro pattern variables x and y are replaced by their corresponding S-expressions in the macro call, in this case the symbols a and b, respectively. The result of evaluating this program is (2 . 1).

1.2.1 Unintended capture: introduced bindings

Unfortunately, this semantic simplicity comes at a cost: macros written in an unhygienic system can result in unintended variable capture. For example, things start going wrong when an unlucky client happens to use **swap!** on a variable named z:

Example 4

```
(let ([q 1]
      [z 2])
  (swap! q z)
  (cons q z))
```

If we again inspect the results of naïve expansion, we see that the binding of z introduced by the definition of **swap!** captures the binding of z in the client code:

Example 5

```
(let ([q 1]
      [z 2])
  (let ([z q]) ; capture!
    (set! q z)
    (set! z z))
  (cons q z))
```

The program produces a completely different result: (1 . 2).

1.2.2 Unintended capture: introduced references

The above example demonstrates that macros that *introduce* bindings into a program during expansion, i.e., bindings that are internal to the implementation rather than externally specified by the client, may accidentally capture references in client code. A dual form of unintended capture—and a subtler one—occurs when macros introduce variable references that might be captured by client code in the context of the macro application.

Consider a macro that creates a simple alias to another binding:

Example 6

```
(define-syntax first
  (syntax-rules ()
    [(first e)
     (car e)]))
```

With naïve macro expansion, the behavior of **first** is sensitive to the *context* in which it is used, not simply its arguments. If the call site rebinds *car*:

Example 7

```
(let ([car 1])
  (first ls))
```

	Variable	**Syntactic keyword**
Introduced binding	captures client reference	captures client reference
Introduced reference	captured by client context	captured by client context

Figure 1.1: Various forms of capture with unhygienic macros.

then the naïve expansion of **first** produces a reference not to the original binding of *car* at the macro definition site, but rather the new binding from the use site:

Example 8

```
(let ([car 1])
  (car ls))
```

Note that this second class of bugs is particularly difficult for macro-writers to guard against. While many macro systems provide facilities for explicitly generating fresh names for introduced bindings, there is very little an implementer can do to protect a macro's introduced references against its contexts of use.

1.2.3 Capture of syntactic keywords

In Scheme, syntactic keywords are scoped just like variables, regardless of whether they are user-defined macros or pre-defined primitives like **lambda**, **if**, and **quote**. This means that the same problems of capture arise when macros introduce bindings or references to syntactic keywords.

Figure 1.1 shows a table that summarizes the different forms of capture that occur with unhygienic macro expansion: macro-introduced bindings to variables or syntactic keywords (e.g. via **let-syntax**) may capture references in subterms provided by the client; macro-introduced references to variables or syntactic keywords may be captured by the context provided by the client.

1.2.4 The trouble with unhygienic macro expansion

Unhygienic macro expansion identifies programs with their representation as trees. But because of variable scope, programs in fact have additional graph structure, with edges between variable bindings and their references. Since these edges are not explicitly represented in S-expressions, maintaining their integrity during macro expansion becomes the responsibility of programmers. Put differently, the tree representation of a program has unenforced representation invariants that are the responsibility of programmers to maintain.

Worse, these invariants require collaboration between macro definitions and clients, not just exposing implementation details of macros but in fact requiring clients to be aware of them. Specifically, the clients of a macro in an unhygienic setting must be aware of any introduced bindings or references in the implementation of the macro in order to avoid unintended capture. Thus unhygienic macros fail as syntactic abstractions and do not scale well beyond small programs.

1.3 Hygienic macro expansion

Languages with hygienic macro expansion automatically avoid these name collisions by renaming variable bindings during the expansion process. For example, the use of the **swap!** macro in Example 4 expands in Scheme to (roughly):

Example 9

```
(let ([q 1]
      [z 2])
  (let ([z1 q]) ; renamed!
    (set! q z)
    (set! z z1))
  (cons q z))
```

Moreover, hygienic macro expansion ensures that bindings in the client program do not inadvertently capture introduced references by renaming *all* bindings during the expansion process. For example, the program in Example 7 expands to:

Example 10

```
(let ([car$_1$ 1]) ; renamed!
  (car ls))
```

1.4 What is hygienic macro expansion?

Though the motivations are clear enough, hygienic macro expansion has so far resisted a precise, formal specification. At the heart of the problem is identifying what is meant by “scope” in a language with extensible syntax.

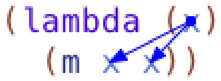
1.4.1 Bindings and references

The motivation for hygienic macro expansion presented in Section 1.2 appeals to intuitions about bindings and references in Scheme programs. However, due to the presence of macros, the syntactic role of an identifier is not always predictable. Consider a program fragment applying an unknown macro **m**:

Example 11

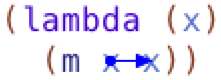
```
(lambda (x)
  (m x x))
```

Without knowing the definition of **m**, we might assume the two inner occurrences of *x* to refer to the outer **lambda**-bound variable:



```
(lambda (x)
  (m x x))
```

Alternatively, **m** might be a binding form similar to **lambda**:

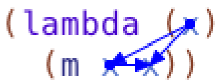


```
(lambda (x)
  (m x x))
```

Because Scheme macros are computationally complete, it is not generally possible to predict the syntactic roles of identifiers before expansion terminates.

1.4.2 Exotic identifiers

In fact, Scheme macros make it possible to define even more exotic binding structures. It is not hard to come up with a definition of **m** that exhibits *both* of the above binding structures simultaneously:



```
(lambda (x)
  (m x x))
```

This works by duplicating the arguments to **m** and placing them in different contexts:

```
(define-syntax m                                      Example 12
  (syntax-rules ()
    [(m a e)
     (begin
       (set! a e)
       (lambda (a) e))]))
```

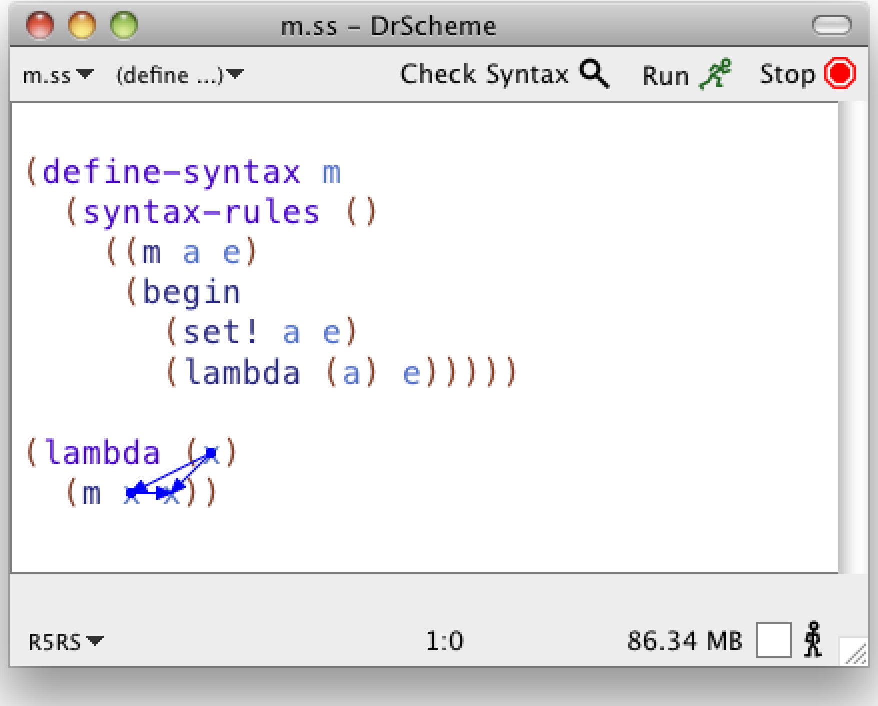
These kinds of identifiers are particularly troublesome to reason about. For one thing, they do not admit the usual freedom of α-conversion. Before expansion, renaming the λ-binding of x requires renaming the inner binding, and vice versa, in spite of the fact that these two bindings can be independently renamed after expansion.

Note that these exotic identifiers are only the by-product of user-defined forms that duplicate identifiers into distinct syntactic contexts. The primitive forms of Scheme exhibit regular, well-defined lexical scoping behavior. By implication, no exotic identifiers remain once a program is fully expanded.

1.4.3 Post-expansion reasoning

For all of these reasons—the inability to reason statically about macro expansion and the presence of exotic identifiers before expansion—a natural approach is to reason about macros by appeal to the results of expansion. Indeed, this is how Scheme actually works: an evaluator or compiler must first completely expand programs before doing analysis, optimization, compilation, or evaluation.[1]

Scheme admits a wide variety of useful tools by performing all reasoning on fully expanded programs. For example, by tracking the provenance of Scheme syntax (primarily by keeping a record of source location information), the DrScheme interactive development environment (IDE) can present the user with a scope-aware view of source programs by fully expanding the program and relating the resulting binding structure to the source syntax:



```
(define-syntax m
  (syntax-rules ()
    ((m a e)
     (begin
       (set! a e)
       (lambda (a) e)))))

(lambda (x)
  (m x x))
```

1.4.4 A circularity

Thus the current state of the art involves revealing the scope of programs by fully expanding them first. As a strategy for formally defining hygienic macro expansion, however, this approach has a fatal flaw. To wit:

[1] The presence of *eval* complicates this picture somewhat, but the approach of expansion before evaluation remains essentially the same.

- To characterize hygienic macro expansion, we need to understand a program's scope.

- To understand a program's scope, we need to know the results of hygienic macro expansion.

What is lacking, then, is a *specification* of the correctness of macro expansion that is *independent of its algorithmic definition*.

1.5 Contributions

In this dissertation, we present the λ_m-calculus, a model of a subset of Scheme macros which comes equipped with a logic for reasoning about the binding structure of programs with macros. The contributions of this dissertation are:

- a formal characterization of hygiene;

- a definition of α-equivalence for programs with macros that is independent of any expansion algorithm;

- a semantic specification of hygienic macro expansion; and

- a novel construct of *binding signature types*.

The remaining chapters of this dissertation proceed as follows.

Chapter 2

We introduce the notion of binding specifications and explain how explicitly specifying the binding structure of macros allows us to reason formally about hygiene.

Chapter 3

We make these intuitions precise with a formal definition of binding signature types. We also present a subtyping judgment on binding signature types with a decidable subtyping algorithm.

Chapter 4

We present the core syntax, semantics, and operations of the λ_m-calculus, including parsing, α-equivalence, and hygienic macro expansion.

Chapter 5

We define the type rules and other well-formedness that ensure that λ_m-calculus programs adhere to their declared specifications.

Chapter 6

We present the mathematical validation of the λ_m-calculus, including type soundness, confluence, and hygiene.

Chapter 7

We discuss the expressiveness of the λ_m-calculus as a programming language. We demonstrate that it is capable of expressing most of the macros of the $\mathrm{R}^5\mathrm{RS}$ [40] standard library, but also discuss some of the limitations and need for future work.

Chapter 8

We conclude with related and future work.

CHAPTER 2

Understanding Hygiene

A specification of hygienic macro expansion must be independent of any specific expansion algorithm. That is, to understand what it means for an expansion algorithm to be hygienic, we require a definition of bindings and references that does not rely on inspecting the results of expansion.

This chapter presents a high-level introduction to the formal framework of this dissertation, which provides an approach to specifying the correctness of hygienic macro expansion in a well-defined manner. The framework hinges on *binding specifications*, which make the binding structure of user-defined macros explicit. The result is a notion of α-equivalence that is independent of the macro expansion algorithm, which in turn provides a correctness criterion for hygiene.

2.1 Hygienic expansion preserves α-equivalence

Consider Examples 2 and 4 from Section 1.2. The failure of unhygienic expansion arises from the expectation of the client that these two programs should be interchangeable. In particular, programmers informally expect that the two programs are α-equivalent, i.e., different only in the particular choice of bound variable names.

In other words, programmers expect the following to hold:

```
(let ([a 1]                 (let ([q 1]
      [b 2])                      [z 2])
  (swap! a b)        =α       (swap! q z)
  (cons a b))                 (cons q z))
```

Naïve macro expansion produces distinct results for these two programs:

```
(let ([a 1]                 (let ([q 1]
      [b 2])                      [z 2])
  (let ([z a])                (let ([z q])
    (set! a b)       ≠α         (set! q z)
    (set! b z))                 (set! z z))
  (cons a b))                 (cons q z))
```

But with hygienic macro expansion, the two source programs expand to equivalent programs:

```
(let ([a 1]                 (let ([q 1]
      [b 2])                      [z 2])
  (let ([z1 a])               (let ([z1 a])
    (set! a b)       =α         (set! q z)
    (set! b z1))                (set! z z1))
  (cons a b))                 (cons q z))
```

This suggests a correctness criterion for hygienic macro expansion: α-equivalent source programs should expand to α-equivalent target programs. Presented as a diagram:

$$\begin{tikzcd}
pgm_1 \arrow[r, dash, "=_\alpha"] \arrow[d, mapsto, two heads] & pgm'_1 \arrow[d, mapsto, two heads] \\
pgm_2 \arrow[r, dash, dashed, "=_\alpha"'] & pgm'_2
\end{tikzcd}$$

What remains is to make the notion of α-equivalence precise for source programs, i.e. Scheme programs with macros.

2.2 Breaking the cycle with interfaces

As we saw in Chapter 1, the difficulty in understanding the scope of Scheme programs comes from the fact that macros extend the syntax of Scheme with new and arbitrarily complicated binding structures. So how can we understand the scope of Scheme programs without expanding them first?

2.2.1 Macro interfaces are binding specifications

In fact, Scheme programmers regularly use macros without inspecting the results of expansion. The reason they are able to do so is that well-specified macros are typically provided with documentation describing their input grammar and binding structure.

Consider a simple **for** loop macro that binds a single variable:

Example 13

```
(define-syntax for
  (syntax-rules ()
    [(for (x e1) e2)
     (for-each (lambda (x) e2) e1)]))
```

A use of the **for** macro might look like:

Example 14

```
(for (i '(1 2 3 4 5))
  (display i))
```

It is customary to document macros with a "schematic" presentation of their syntax along with information about their scoping behavior. For example, a typical style of documenting the **for** macro might look something like this:

> **Syntax:** (**for** (*identifier expression*) *expression*) :: *expression*
>
> Evaluates the first *expression* to obtain a list and repeatedly evaluates the second *expression* with the *identifier* bound to successive elements of the list.

Much of this information can be formalized. In this work we provide a framework for expressing macro specifications concisely and formally.

2.2.2 Macros with explicit interfaces

Informally, programmers can refer to the documentation to deduce the correctness of transformations such as:

$$\begin{array}{lcl} (\textbf{for}\ (\textit{i}\ \text{'(1 2 3 4 5))} & & (\textbf{for}\ (\textit{j}\ \text{'(1 2 3 4 5))} \\ \quad (\textit{display i})) & =_\alpha & \quad (\textit{display j})) \end{array}$$

The key insight of this dissertation is that by annotating *all* macro definitions with interfaces describing their grammar and binding structure, we can reason *formally* about the binding structure of Scheme programs, and without first macro-expanding. More technically, explicit annotations provide us with enough information to obtain a formal definition of α-equivalence of pre-expansion Scheme programs.

2.2.3 A note on applying the theory

Before taking a closer look at macro interfaces, let us take a moment to discuss several potential applications of this theory.

2.2.3.1 Programming system

Most directly, the model presented in this dissertation lends itself to the design of a programming system with macros that are explicitly typed and checked. Such a system may be more restrictive than typical Scheme implementations, in that it would reject macros that an unrestricted Scheme would not. Nevertheless, the language would have a clear and well-defined notion of hygiene and α-equivalence and would provide programmers with stronger guarantees than Scheme.

2.2.3.2 Proof system

This system could also be applied as a framework for reasoning about a subset of Scheme macros in the context of Scheme itself. In future work, we intend to explore the interaction between macros with explicit interfaces and unannotated macros.

2.2.3.3 Programming methodology

The formal model presented here also suggests a way of thinking about well-behaved Scheme macros. Even in the absence of formal proofs, it sheds light on the kind of information that should be provided in documentation for Scheme macros and suggests what kinds of macros are harder to reason about. Some preliminary thoughts on this subject were presented in Herman and Van Horn [33].

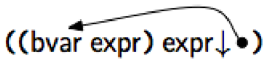
2.3 Shape and binding specifications

Prior work by Culpepper and Felleisen [17] demonstrated that the syntactic *shape* of macros like **for** can be represented as a type:

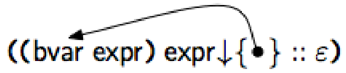
```
((var expr) expr)
```

This type is not dissimilar from the schematic presentation given above. Note that the type only describes the shape of the macro argument (i.e., the *cdr* of the form), rather than the entire macro application form including the macro keyword.

In order to represent the binding structure of **for**, we must also indicate that the identifier is bound in the second subexpression:



More precisely, the lexical environment of the second subexpression is extended with a frame containing a binding for the identifier. Assuming a list-of-frames (or “ribcage”) representation for environments, we can specify the binding structure by describing the *extension* to the lexical environment of the second subexpression:



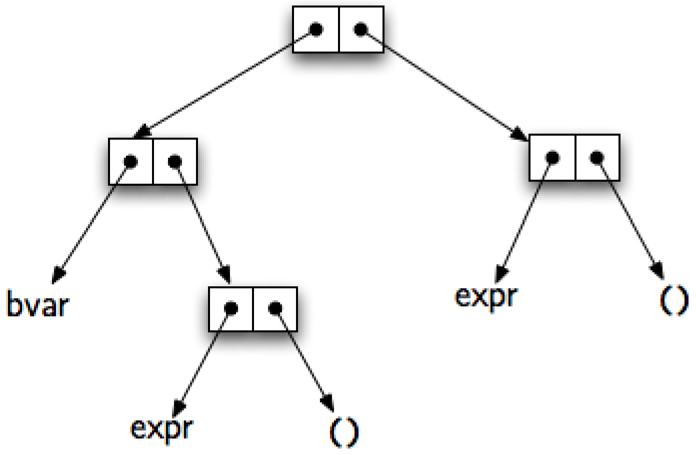
(To avoid confusion between S-expressions representing syntax and lists representing environments, we use a distinct notation for the latter, with ε for the empty list and – :: – for adjunction.) Notice that the frame containing the bound identifier is a singleton, whose only element “points to” the binding occurrence.

2.3.1 Tree addresses

What remains to formalize is a precise way to identify distinct identifiers—or collections of identifiers—in a type, i.e., the ad-hoc arrows in the above pseudo-notation. For this we employ *tree addresses* [30], which allow us to identify the bindings defined at a particular location within a macro’s argument.

Definition 2.3.1. *A tree address ℓ is an element of $\mathbb{D}^*$, where $\mathbb{D} = \{\mathsf{A}, \mathsf{D}\}$.*

Consider the tree structure of ((bvar expr) expr):



In this shape, the address of the identifier is AA, and the addresses of the first and second subexpressions are ADA and AD, respectively. Note that addresses compose right-to-left, analogous to conventional Scheme operation names such as *caar*, *cadar*, and *cadr*. Thus we might represent the type expected by the **for** macro as

$$\texttt{((}\mathsf{bvar}\ \mathsf{expr}\texttt{)}\ \mathsf{expr}{\downarrow}(\{\mathsf{AA}\} :: \varepsilon)\texttt{)}$$

2.3.2 Collections of bindings

Macros often bind multiple bindings simultaneously. For example, the type of the **lambda** primitive might be expressed as

$$\texttt{(}formals\ \mathsf{expr}{\downarrow}(\mathsf{A} :: \varepsilon)\texttt{)} \rightarrow \mathsf{expr}$$

given an appropriate definition of the $formals$ type.

In order for a tree location to denote a *collection* of bindings rather than a single binding, it must be possible for a given tree location to specify the bindings defined within its structure. This is the motivation behind *export types*. For example, the recursive type $formals$ might be represented as:

$$\mu A.\cup\{\texttt{()}{\uparrow}\{\}, \texttt{(}\mathsf{bvar}\ \texttt{.}\ A\texttt{)}{\uparrow}(\{\mathsf{A}\} \cup \mathsf{D})\}$$

This type can be read as specifying that a formals list is either an empty list, denoting an empty set of bindings, or a pair of a bound variable and a formals list, denoting the bindings within the remainder of the list along with the additional bound variable. (Note the typographic distinction between the recursive type variable A and the tree address A.) In each variant of the union type, the upwards-facing arrow specifies the collection of bindings defined within its corresponding structure and “exported” for use in another scope.

2.4 Binding signature types as attribute grammars

As a metaphor for understanding binding signature types, it can be helpful to consider how we might implement a compiler or evaluator for a language with a fixed syntax using *attribute grammars* [42, 43]. Consider a simple Scheme-like language with multiary **lambda**, application and variable references. A simple grammar for this language might look roughly like the following:

Example 15

$$
\begin{array}{lcl}
\mathit{expr} & ::= & (\textbf{lambda}\ \mathit{formals}\ \mathit{expr}) \\
 & \mid & (\mathit{expr}\ .\ \mathit{actuals}) \\
 & \mid & \mathit{var} \\
 & & \\
\mathit{formals} & ::= & (\mathit{var}\ .\ \mathit{formals}) \\
 & \mid & () \\
 & & \\
\mathit{actuals} & ::= & (\mathit{expr}\ .\ \mathit{actuals}) \\
 & \mid & ()
\end{array}
$$

Attribute grammars extend classic BNF-style grammars by attaching semantic information to the nodes in the parse tree. This additional information is attached to productions in the grammar by attribute expressions, or *semantic actions*, as they are commonly known in popular parser generators such as yacc [37] and bison [21]. Attributes are classified into two categories:

- *synthesized attributes* are determined by the attributes of child nodes;
- *inherited attributes* are determined by the attributes of a node's parent.

We can annotate the parse tree with information about the language's scoping rules:

Example 16

$$\begin{array}{lcl}
\mathit{expr} & ::= & (\textbf{lambda}\ \mathrm{xs}{:}\mathit{formals}\ \mathrm{e}{:}\mathit{expr}) \\
 & & \quad \hookrightarrow \mathrm{e}.\mathit{imports} = \mathrm{xs}.\mathit{exports} :: \varepsilon \\
 & | & (\mathit{expr}\ .\ \mathit{actuals}) \\
 & | & \mathit{var} \\
 & & \\
\mathit{formals} & ::= & (\mathrm{x}{:}\mathit{var}\ .\ \mathrm{xs}{:}\mathit{formals}) \\
 & & \quad \hookrightarrow \mathrm{this}.\mathit{exports} = \{\mathrm{x}.\mathit{name}\} \cup \mathrm{xs}.\mathit{exports} \\
 & | & () \\
 & & \quad \hookrightarrow \mathrm{this}.\mathit{exports} = \{\} \\
 & & \\
\mathit{actuals} & ::= & (\mathit{expr}\ .\ \mathit{actuals}) \\
 & | & ()
\end{array}$$

In this attribute grammar, the **lambda** form annotates its body expression with an attribute indicating that it “imports” the variables bound in the *formals* list—that is, it brings the variables in scope by extending the environment. The *formals* list identifies its bound variables by storing them in a synthesized attribute called $\mathit{exports}$.

Binding signature types can be thought of as a modularized attribute grammar for Scheme: instead of a single, monolithic grammar, the language starts with a base grammar and is extended locally by the introduction of macros. The types include both the syntactic information of a grammar production and the scope information of a semantic action. Specifically, the downwards-arrow types, which extend the current environment, correspond to the inherited $\mathit{imports}$ attribute in the above example, and the upwards-arrow types correspond to the $\mathit{exports}$ attribute.

Attribute grammars have a rich and complex family of computational models; in our case, the set of attributes is fixed and small, so the model for evaluating semantic actions is relatively simple. (As we shall see, the language of binding signatures is restricted to prevent complex dependencies, thereby simplifying the evaluation model.) Nevertheless, the analogy to attribute grammars can prove useful in understanding the various components of the formalism described in following chapters.

CHAPTER 3

Binding Signature Types

At the heart of the λ_m-calculus is the novel construct of *binding signature types*, which specify both the syntax and binding structure of macros. This chapter introduces these types and some of their operations.

3.1 Syntax as trees

In Chapter 1, we characterized programs as tree-like but with additional graph structure due to variable bindings and scope. Scheme macros present syntax as tree-shaped data, while tracking additional metadata about bindings and renamings to implement hygiene.

3.1.1 Starting from S-expressions

Despite this additional metadata, Scheme syntax contains a core structure that is similar to S-expressions. That is, syntax trees take the rough form:

$$tree ::= atom \mid \texttt{()} \mid \texttt{(}tree \ \texttt{.} \ tree\texttt{)}$$

for a given notion of $atom$. As in Lisp and Scheme, we use the notation

$$\texttt{(}tree_1 \ tree_2 \ \cdots \ tree_n\texttt{)} \overset{\text{def}}{=} \texttt{(}tree_1 \ \texttt{. (}tree_2 \ \texttt{. (} \cdots \texttt{. (}tree_n \ \texttt{. ()))))}$$

as a convenient shorthand for n-ary sequences. It is important to understand that this sequence notation is purely a notational convenience *in the* *surface presentation of the model*; within the model itself, syntax sequences are nothing more than nested pairs.

3.1.2 Tree addresses

Recall from Definition 2.3.1 that a tree address ℓ is an element of $\mathbb{D}^*$, where $\mathbb{D}$ consists of the directives A and D, respectively denoting left (*car*) and right (*cdr*) projection of syntax pairs. Presented as a grammar:

$$\ell ::= \epsilon \mid \mathsf{A}\ell \mid \mathsf{D}\ell$$

We represent address concatenation as juxtaposition:

$$\begin{aligned} \epsilon\ell &= \ell \\ (\mathsf{A}\ell)\ell' &= \mathsf{A}(\ell\ell') \\ (\mathsf{D}\ell)\ell' &= \mathsf{D}(\ell\ell') \end{aligned}$$

Address projection is defined inductively:

$$\begin{aligned} \mathit{tree}.\epsilon &= \mathit{tree} \\ \mathit{tree}.\mathsf{A}\ell &= \mathit{tree}_1 \quad \text{if } \mathit{tree}.\ell = \texttt{(}\mathit{tree}_1 \texttt{ . } \mathit{tree}_2\texttt{)} \\ \mathit{tree}.\mathsf{D}\ell &= \mathit{tree}_2 \quad \text{if } \mathit{tree}.\ell = \texttt{(}\mathit{tree}_1 \texttt{ . } \mathit{tree}_2\texttt{)} \end{aligned}$$

The prefix and proper-prefix relations are respectively defined by:

$$\begin{aligned} \ell \preceq \ell' &\overset{\text{def}}{\iff} \ell\ell'' = \ell' \text{ for some } \ell'' \\ \ell \prec \ell' &\overset{\text{def}}{\iff} \ell \preceq \ell' \wedge \ell \neq \ell' \end{aligned}$$

3.2 Types to classify forms

In Scheme tradition, the loosely-defined word *form* is sometimes used to mean a single cohesive program fragment such as a function application, literal expression, variable reference, definition, or macro application. Forms can serve different syntactic roles, as in the following fragment of a memoization library:

Example 17

```
(define-syntax memo-lambda
  (syntax-rules ()
    [(memo-lambda (x) e)
     (let ([memo-table (make-memo-table)])
       (lambda (x)
         (unmemo! memo-table x (lambda () e))))]))

(define-syntax define/memo
  (syntax-rules ()
    [(define/memo (f x) e)
     (define f (memo-lambda (x) e))]))
```

In this example, the **memo-lambda** macro defines an *expression form*:[1] it can be applied in any context where Scheme expressions are allowed. The **define/memo** macro defines a *definition form* and can only be applied in a Scheme definition context.

These syntactic roles give rise to a notion of *form types*, which classify Scheme forms. The ground types of Scheme include expr, the type of expression forms, and defn, the type of definition forms:

$$o ::= \mathsf{expr} \mid \mathsf{defn}$$

However, in the present work, we focus only on the ground type expr and leave definitions for future work. See Chapter 8 for a discussion of definitions and the defn type.

Variable references can also serve distinct syntactic roles. For example, in the expression

Example 18

```
(memo-lambda (n)
  (sqrt (abs n)))
```

there are variable references to *sqrt*, *abs*, and *n*, but the use of **memo-lambda** is itself another kind of variable reference. So we have traditional

[1] Somewhat confusingly, traditional usage overloads the word “form” to refer both to nodes in the syntax tree and syntax operators. Perhaps a good way to reconcile these distinct usages is to think of “the **memo-lambda** form” as a sort of metonymic stand-in for “the class of forms defined by the **memo-lambda** operator.”

$$
\begin{array}{lcl}
\tau & ::= & \mathsf{expr} \mid \sigma \rightarrow \mathsf{expr} \\
\sigma & ::= & \tau \mid \mathsf{bvar} \mid \mathsf{data} \mid \mathtt{()} \mid \mathtt{(}\sigma \ \mathtt{.}\ \sigma\mathtt{)} \mid \sigma{\downarrow}\beta \mid \sigma{\uparrow}\beta \mid \mu A.\sigma \mid A \mid \cup\{\overline{\sigma_i}\} \\
 & & \\
\delta & ::= & \textsc{none} \mid \textsc{var} \mid \textsc{rib} \mid \textsc{env} \\
 & & \\
\beta & ::= & \rho \mid \gamma \\
\gamma & ::= & \varepsilon \mid \rho :: \gamma \mid \ell @ \gamma \\
\rho & ::= & \{\overline{\ell : \delta}\}
\end{array}
$$

Figure 3.1: Types and binding signatures.

variables like *n*, which can be used as expressions, and macro variables like **memo-lambda**, which can be used as syntactic operators. Macro types take the general form $\sigma \rightarrow \mathsf{expr}$, which can be read as "the type of macros that take arguments of *syntax type* σ and produce expression forms." Next we consider the meaning of syntax types σ.

3.3 Types to interpret syntax

Figure 3.1 presents the grammar of types for the λ_m system, including form types τ and syntax types σ. Where form types classify forms, syntax types provide an *interpretation* for the syntax expected by a macro. Alternatively, as described in Section 2.4, we can view syntax types as analogous to attribute grammars: they describe both the syntax (shape) and semantics (binding structure) of macro arguments.

Let us consider each of the variants of syntax types in turn. For a node in a syntax tree to have a form type τ, it may be any form of the specified type, with no constraints placed on its syntactic representation. The macro must treat the node as opaque; it cannot deconstruct, rearrange, or even inspect the syntactic representation of the node.[2] The bvar variant is the type of binding occurrences of variables. The type data describes literal data

[2] This is analogous to the *generativity* property described in Ganz et al. [27]

suitable for use with the **quote** primitive. The types () and (σ_1 . σ_2) describe surface syntax that by itself serves no semantic purpose.

The next two variants, *binding signature types*, are of key importance. An *import type* $\sigma{\downarrow}\beta$ describes a syntax node in the scope of a *binding signature* β, i.e., that "imports" the bindings of β into its environment. An *export type* $\sigma{\uparrow}\beta$ attaches a collection of bindings described by signature β to its syntax node; these bindings are then available for other binding signatures to add to their own collections of bindings. We return to the details of binding signatures in the next section.

The next two variants add equi-recursive types to the model. The recursive type constructor $\mu A.\sigma$ binds a type variable A, drawn from a countably infinite universe of type variable names. We assume each type variable A to have an associated bindings type δ (see Section 3.4). The binary relation $A : \delta$ relates a type variable to its bindings type.

The ad-hoc union type constructor $\cup\{\overline{\sigma_i}\}$ consists of a non-empty sequence of variants.[3] The combination of recursive types and union types makes it possible to express inductively defined tree grammars with alternate variants.

3.4 Binding signatures

Figure 3.1 also presents the definition of binding signatures β. There are four categories δ of collections of bindings: none at all; a single identifier; a frame (or "rib") of identifiers all bound at the same lexical level; or an ordered list of frames (or "ribcage"). A binding signature β may be either an *environment signature* γ, representing a ribcage, or a *rib signature* ρ describing a single rib. The environment signature ε represents the empty ribcage. The environment signature $\rho :: \gamma$ places the rib ρ as the innermost

[3] Syntactically, the variants are ordered, although the order turns out to be essentially irrelevant. The order serves as a minor convenience in defining several of the operations of the λ_m-calculus.

frame in front of the environment signature γ. The environment signature $\ell@\gamma$ uses the tree address to refer to another ribcage exported at the syntax node at address ℓ, concatenating those bindings in front of the environment signature γ. A rib signature consists of a set of tree addresses referring to bound variables or exports at the addressed syntax nodes. We return to the δ-annotation in Section 5.1.

Ribcage concatenation is denoted γ, γ' and is straightforward to define:

$$\begin{array}{lcl} \varepsilon, \gamma' & = & \gamma' \\ (\rho :: \gamma), \gamma' & = & \rho :: (\gamma, \gamma') \\ (\ell@\gamma), \gamma' & = & \ell@(\gamma, \gamma') \end{array}$$

3.5 Addressing regions

Tree addresses refer to nodes in a syntax tree relative to a root position (at address ϵ), and binding signatures occur within the context of syntax types. So addresses and signatures are interpreted relative to the structure of the type in which they appear. In particular, several type constructs introduce new *addressing regions*, which restart addresses at ϵ. These are recursive types $\mu A.\sigma$, union types $\cup\{\overline{\sigma_i}\}$, and macro types $\sigma \to \textsf{expr}$. Addresses within these types are interpreted relative to the root of their subtree.

Let us consider the type of the **lambda** primitive as an example. To begin with, the type of the formals list is:

Example 19

$$\mathit{formals} \stackrel{\text{def}}{=} \mu A.\cup\{\texttt{()}\!\uparrow\!\emptyset, \texttt{(bvar . }A\texttt{)}\!\uparrow\!\{\textsf{A}\!:\!\textsc{var}, \textsf{D}\!:\!\textsc{rib}\}\}$$

That is, a formals list is either:

- empty, in which case it exports an empty rib of bindings;

- or a pair of a bound identifier and a formals list, in which case it exports the rib of the sub-list along with the additional identifier.

Note that the addressing is relative to the root of syntax trees matched by the union type. The type of the **lambda** primitive uses the above definition:

Example 20

$$\texttt{(}\mathit{formals}\ \mathsf{expr}{\downarrow}\{\mathsf{A}{:}\mathrm{RIB}\} :: \varepsilon\texttt{)} \to \mathsf{expr}$$

Here the body expression imports the rib defined by the formals list by addressing it relative to the root of the whole tree matched by **lambda**. Note that, because of the presence of the union-type constructor, the $formals$ type can be placed directly into another type unchanged—without, for example, offsetting its tree addresses relative to some new base address.

Now, there are restrictions on the range of addressable locations within a given region. For example, a type cannot import from an address within a nested union type; otherwise it would be possible to refer to potentially non-existent sub-trees, as in the following example:

Example 21

$$\texttt{(}\mathsf{expr}{\downarrow}\{\mathsf{AD}{:}\mathrm{VAR}\} :: \varepsilon\ \texttt{.}\ \cup\{\texttt{()}, \texttt{(}\mathsf{bvar}\texttt{)}\}\texttt{)}$$

This type cannot be legal, since its grammar matches syntax trees such as $\texttt{(}x\ \texttt{.}\ \texttt{())}$. The tree address AD (i.e., *cadr*) clearly does not correspond to a node in this tree.

The region structure of types and the validity of tree address references within binding signatures are aspects of the syntactic well-formedness of types. We examine these criteria in detail in Section 5.2.1.

3.6 Subtyping

The presence of recursive types with ad-hoc unions gives rise to a natural notion of subtyping. Consider the type of a macro that takes a sequence of expressions in its argument:

Example 22

$$(\mu A.\cup\{\texttt{()}, \texttt{(}\mathsf{expr}\ \texttt{.}\ A\texttt{)}\}) \to \mathsf{expr}$$

It is naturally valid to apply this macro to an empty sequence `()`. Conceptually, the actual argument `()` can be interpreted to have the type `()`, which is a subtype of the expected type:

$$\dfrac{\dfrac{\texttt{()} <: \texttt{()}}{\texttt{()} <: \cup\{\texttt{()}, \mu A.\cup\{\texttt{()}, A\}\}}}{\texttt{()} <: \mu A.\cup\{\texttt{()}, A\}}$$

There are well-known algorithms for deciding subtyping with equirecursive types. Our approach follows Gapeyev et al [28]. The presence of binding signatures complicates the algorithm, but the structure is similar:

1. The subtyping relation is defined coinductively, i.e., as the greatest fixed point of a set of inference rules.

2. The decision algorithm works backwards from the goal, searching the graph of assertions reachable by inversion of the inference rules. Because the relation is coinductive, the derivation may in general be infinite. We restrict the grammar of types to ensure that the derivation is always regular and can be expressed as a finite graph.

3. To ensure termination, assertions that can lead to cycles in the graph are cached and visited at most once.

4. A pair of types is in the relation iff no inconsistent assertion is found after searching the graph exhaustively.

3.6.1 Contractiveness

Many of the operations on types involve unfolding a recursive type definition $\mu A.\sigma$ to obtain $\sigma[\mu A.\sigma/A]$, which might be a larger type. We impose the simple restriction that all recursive types must be *contractive*: there must appear at least one pair type between a recursive type variable and its binding. This restriction disallows pathological types such as $\mu A.A$ or $\mu A.(A \to \textsf{expr})$, and ensures that the complete unfolding of any type is a regular tree, i.e., a tree with only a finite number of distinct subtrees.

3.6.2 Import normalization

The syntax of import types $\sigma{\downarrow}\beta$ is conveniently expressive, but because of its generality, it may admit multiple equivalent representations of a type. For example, importing a signature in a compound syntax type such as

Example 23

$$\mathsf{(expr\ .\ expr)}{\downarrow}\beta$$

could be simplified by distributing the signature to the subcomponents:

Example 24

$$\mathsf{(expr}{\downarrow}\beta\ \mathsf{.\ expr}{\downarrow}\beta\mathsf{)}$$

When comparing types, the distinction between these two should be irrelevant; intuitively, a pair of expressions collectively in the scope of β is the same as a pair of expressions each individually in the scope of β.

To eliminate this surface distinction, the subtyping relation makes use of an *import normalization* operation $\sigma \Downarrow \beta$, which "pushes imports inwards." A syntax type in *import normal form*, written $\hat{\sigma}$, matches the following grammar:

$$\hat{\sigma} \quad ::= \quad \tau{\downarrow}\gamma \mid (\mu A.\sigma){\downarrow}\gamma \mid (\cup\{\overline{\sigma_i}\}){\downarrow}\gamma \mid \mathsf{bvar} \mid \mathsf{data} \mid \mathsf{()} \mid \mathsf{(}\hat{\sigma}\ \mathsf{.}\ \hat{\sigma}\mathsf{)} \mid \hat{\sigma}{\uparrow}\beta$$

In import normal form, imported binding signatures in the outermost region can only occur on form types, recursive types, or union types.

Import normalization is defined in Figure 3.2. Types are normalized relative to an import environment γ. For form types τ, normalization attaches the import environment directly to the form type. In the next three cases, the imports are dropped since binding occurrences of identifiers, literal (quoted) data, and the empty sequence $\mathsf{()}$ are unaffected by imports. The rules for

$$
\begin{array}{rcl}
\tau \Downarrow \gamma & = & \tau{\downarrow}\gamma \\
\mathsf{bvar} \Downarrow \gamma & = & \mathsf{bvar} \\
\mathsf{data} \Downarrow \gamma & = & \mathsf{data} \\
\mathtt{()} \Downarrow \gamma & = & \mathtt{()} \\
\mathtt{(}\sigma_1\ \mathtt{.}\ \sigma_2\mathtt{)} \Downarrow \gamma & = & \mathtt{(}\sigma_1 \Downarrow \gamma\ \mathtt{.}\ \sigma_2 \Downarrow \gamma\mathtt{)} \\
\sigma{\uparrow}\beta \Downarrow \gamma & = & \sigma \Downarrow \gamma{\uparrow}\beta \\
\sigma{\downarrow}\beta \Downarrow \gamma & = & \sigma \Downarrow (\beta, \gamma) \\
(\mu A.\sigma) \Downarrow \gamma & = & (\mu A.\sigma){\downarrow}\gamma \\
(\cup\{\overline{\sigma_i}\}) \Downarrow \gamma & = & (\cup\{\overline{\sigma_i}\}){\downarrow}\gamma
\end{array}
$$

Figure 3.2: Import normalization.

pair types and export types are structural. The rule for import types accumulates the imports and concatenates them to the front of the import argument. Finally, recursive types and union types directly attach the accumulated imports, as with form types τ.

Note that imported binding signatures never cross region boundaries, since this would change their meaning. Instead, normalization cooperates with the definition of subtyping (see Section 3.6.4) by suspending imported signatures at region boundaries, where the subtyping judgment picks them up and continues normalization.

Proposition 3.6.1 (Associativity). $(\sigma \Downarrow \gamma) \Downarrow \gamma' = \sigma \Downarrow (\gamma, \gamma')$

Proof. Straightforward induction on the definition of import normalization. □

3.6.3 Region displacement

A typical approach to subtyping union types is to compare their variants pointwise. For example, we might wish to conclude that $\sigma <: \cup\{\overline{\sigma_i'}\}$ if $\sigma <: \sigma_i'$ for one of the types σ_i' in the union. However, due to binding signatures, naïvely extracting types from within region delimiters like $\cup$ is not semantics-preserving. This would lead to incorrect conclusions such as:

$$(\mathsf{expr}\ .\ (\mathsf{bvar}\ .\ \mathsf{expr}{\downarrow}\{\mathsf{A}{:}\textsc{var}\})) <: (\mathsf{expr}\ .\ \cup\{(), (\mathsf{bvar}\ .\ \mathsf{expr}{\downarrow}\{\mathsf{A}{:}\textsc{var}\})\})$$

The supposed subtype is nonsense, because the binding signature refers to tree location A, which is no longer a binding occurrence of an identifier but instead an expression. Instead, the correct conclusion should be:

$$(\mathsf{expr}\ .\ (\mathsf{bvar}\ .\ \mathsf{expr}{\downarrow}\{\mathsf{AD}{:}\textsc{var}\})) <: (\mathsf{expr}\ .\ \cup\{(), (\mathsf{bvar}\ .\ \mathsf{expr}{\downarrow}\{\mathsf{A}{:}\textsc{var}\})\})$$

In other words, we must account for the fact that, in the subtype, the address of bvar is relative not to the root of the union type but its container.

In order to extract types from region delimiters safely, we must keep track of the current location relative to the nearest enclosing region delimiter. When we extract types from nested regions, we offset their tree addresses by the current location. The binary *region displacement* operator $\ll$ is given in Figure 3.3. Relocating a signature β to address ℓ, written $\beta \ll \ell$, adds ℓ as a suffix to all tree addresses occurring within β. Relocating a type σ to address ℓ, written $\sigma \ll \ell$, relocates all signatures found within the top-level region of σ, i.e., outside of any region delimiters.

The subtyping judgment must keep track of the current address in order to shift addresses appropriately when extracting syntax types from their regions. This gives us a judgment of the form $\ell \vdash \hat{\sigma} <: \hat{\sigma}'$, with the top-level definition of subtyping:

$$\sigma <: \sigma' \overset{\text{def}}{=} \epsilon \vdash \sigma \Downarrow \varepsilon <: \sigma' \Downarrow \varepsilon$$

Proposition 3.6.2 (Distributivity). $(\sigma \Downarrow \gamma) \ll \ell = (\sigma \ll \ell) \Downarrow (\gamma \ll \ell)$

Proof. By induction on the definition of import normalization. □

3.6.4 Subtyping judgment

The subtyping judgment $\ell \vdash \hat{\sigma} <: \hat{\sigma}'$ is defined as the greatest fixed point of the rules given in Figure 3.4; in other words, the rules are interpreted coinductively.

$$
\begin{array}{rcl}
\{\overline{\ell'_i : \delta_i}\} \ll \ell & = & \{\overline{\ell'_i \ell : \delta_i}\} \\
\varepsilon \ll \ell & = & \varepsilon \\
\rho :: \gamma \ll \ell & = & (\rho \ll \ell) :: (\gamma \ll \ell) \\
\ell' @ \gamma \ll \ell & = & \ell' \ell @ (\gamma \ll \ell) \\
 & & \\
\tau \ll \ell & = & \tau \\
\mathsf{bvar} \ll \ell & = & \mathsf{bvar} \\
\mathsf{data} \ll \ell & = & \mathsf{data} \\
\texttt{()} \ll \ell & = & \texttt{()} \\
\texttt{(}\sigma_1 \;\texttt{.}\; \sigma_2\texttt{)} \ll \ell & = & \texttt{(}(\sigma_1 \ll \ell) \;\texttt{.}\; (\sigma_2 \ll \ell)\texttt{)} \\
\sigma{\downarrow}\beta \ll \ell & = & (\sigma \ll \ell){\downarrow}(\beta \ll \ell) \\
\sigma{\uparrow}\beta \ll \ell & = & (\sigma \ll \ell){\uparrow}(\beta \ll \ell) \\
\mu A.\sigma \ll \ell & = & \mu A.\sigma \\
A \ll \ell & = & A \\
\cup\{\overline{\sigma_i}\} \ll \ell & = & \cup\{\overline{\sigma_i}\}
\end{array}
$$

Figure 3.3: Region displacement.

At form types, the import signatures γ must agree. In the case of macro types, the types to the left of the arrow are compared contravariantly. Base types bvar, data, and `()` are subtypes of themselves. Pair types are compared structurally and update the tree address ℓ to account for the new tree context. Export types must have equal export signatures and compare their types covariantly.

The rule for comparing two recursive types is carefully designed to ensure decidability. Specifically, two recursive types are only unfolded and compared if their imports are the same. A more permissive definition might continue importing the signatures to see if they end up the same at the leaves of the unfolded type. Indeed, in the next two cases, where only one of the two types being compared is recursive, the recursive type is unfolded and the imports pushed inwards. However, the case of two recursive types is the one case that can lead to infinite proof trees. By unfolding the types in the initial tree address ϵ and with empty imports, we guarantee that there are a finite number of distinct pairs of recursive type bodies in the potentially infinite proof tree. This leads to a computable decision procedure that caches

$$\boxed{\ell \vdash \hat{\sigma} <: \hat{\sigma}}$$

$$\frac{}{\ell \vdash \mathsf{expr}{\downarrow}\gamma <: \mathsf{expr}{\downarrow}\gamma} \qquad \frac{\sigma' <: \sigma}{\ell \vdash (\sigma \to \mathsf{expr}){\downarrow}\gamma <: (\sigma' \to \mathsf{expr}){\downarrow}\gamma}$$

$$\frac{}{\ell \vdash \mathsf{bvar} <: \mathsf{bvar}} \qquad \frac{}{\ell \vdash \mathsf{data} <: \mathsf{data}}$$

$$\frac{}{\ell \vdash \mathtt{()} <: \mathtt{()}} \qquad \frac{\mathsf{A}\ell \vdash \hat{\sigma}_1 <: \hat{\sigma}'_1 \qquad \mathsf{D}\ell \vdash \hat{\sigma}_2 <: \hat{\sigma}'_2}{\ell \vdash \mathtt{(}\hat{\sigma}_1 \ \mathtt{.}\ \hat{\sigma}_2\mathtt{)} <: \mathtt{(}\hat{\sigma}'_1 \ \mathtt{.}\ \hat{\sigma}'_2\mathtt{)}}$$

$$\frac{\ell \vdash \hat{\sigma} <: \hat{\sigma}'}{\ell \vdash \hat{\sigma}{\uparrow}\beta <: \hat{\sigma}'{\uparrow}\beta} \qquad \frac{\epsilon \vdash \sigma[\mu A.\sigma/A] \Downarrow \varepsilon <: \sigma'[\mu A'.\sigma'/A'] \Downarrow \varepsilon}{\ell \vdash (\mu A.\sigma){\downarrow}\gamma <: (\mu A'.\sigma'){\downarrow}\gamma}$$

$$\frac{\begin{array}{c}\hat{\sigma}' \neq (\mu \_ . \_){\downarrow}\_ \\ \ell \vdash (\sigma[\mu A.\sigma/A] \ll \ell) \Downarrow \gamma <: \hat{\sigma}'\end{array}}{\ell \vdash (\mu A.\sigma){\downarrow}\gamma <: \hat{\sigma}'} \qquad \frac{\begin{array}{c}\hat{\sigma} \neq (\mu \_ . \_){\downarrow}\_ \\ \ell \vdash \hat{\sigma} <: (\sigma'[\mu A'.\sigma'/A'] \ll \ell) \Downarrow \gamma'\end{array}}{\ell \vdash \hat{\sigma} <: (\mu A'.\sigma'){\downarrow}\gamma'}$$

$$\frac{\begin{array}{c}\hat{\sigma}' \neq (\mu \_ . \_){\downarrow}\_ \\ \forall i.\ell \vdash (\sigma_i \ll \ell) \Downarrow \gamma <: \hat{\sigma}'\end{array}}{\ell \vdash (\cup\{\overline{\sigma_i}\}){\downarrow}\gamma <: \hat{\sigma}'} \qquad \frac{\begin{array}{c}\hat{\sigma} \neq (\mu \_ . \_){\downarrow}\_, (\cup\ \_){\downarrow}\_ \\ \exists i.\ell \vdash \hat{\sigma} <: (\sigma'_i \ll \ell) \Downarrow \gamma\end{array}}{\ell \vdash \hat{\sigma} <: (\cup\{\overline{\sigma'_i}\}){\downarrow}\gamma}$$

Figure 3.4: The subtyping judgment.

the bodies of recursive types to prune out infinite paths in the proof tree (see Section 3.6.5).

Comparing a recursive type to a non-recursive type involves three steps:

1. Unfold the recursive type.

2. Adjust the addresses in the unfolded type by the current address ℓ.

3. Normalize the resulting type, importing the suspended signature γ.

The rules for comparing union types proceed similarly. In the first case, a union type is a valid subtype if all of its variants are subtypes of the right-hand type; in the second, a union type is a valid supertype if at least one of its variants is a supertype of the left-hand type.

Lemma 3.6.3. *If* $\ell \vdash \hat{\sigma} <: \hat{\sigma}'$ *then* $\ell\ell_0 \vdash (\hat{\sigma} \ll \ell_0) <: (\hat{\sigma}' \ll \ell_0)$.

Proof. By simultaneous induction on $\hat{\sigma} \ll \ell_0$ and $\hat{\sigma}' \ll \ell_0$. □

Lemma 3.6.4. *If* $\ell \vdash \hat{\sigma} <: \hat{\sigma}'$ *then* $\ell \vdash \hat{\sigma} \Downarrow \gamma <: \hat{\sigma}' \Downarrow \gamma$.

Proof. By simultaneous induction on $\hat{\sigma} \Downarrow \gamma$ and $\hat{\sigma}' \Downarrow \gamma$. □

Theorem 3.6.5. *If* $\ell \vdash \hat{\sigma} <: \hat{\sigma}'$ *and* $\ell \vdash \hat{\sigma}' <: \hat{\sigma}''$ *then* $\ell \vdash \hat{\sigma} <: \hat{\sigma}''$.

Proof. By coinduction. We show the case where only $\hat{\sigma}$ is non-recursive:

$$
\begin{array}{ll}
 & \ell \vdash \hat{\sigma} <: \mu A'.\sigma'{\downarrow}\gamma' \wedge \ell \vdash \mu A'.\sigma'{\downarrow}\gamma' <: \mu A''.\sigma''{\downarrow}\gamma' \\
\Longrightarrow & \{\text{inversion}\} \\
 & \ell \vdash \hat{\sigma} <: \mu A'.\sigma'{\downarrow}\gamma' \wedge \epsilon \vdash \sigma'[\mu A'.\sigma'/A'] \Downarrow \varepsilon <: \sigma''[\mu A''.\sigma''/A''] \Downarrow \varepsilon \\
\Longrightarrow & \{\text{inversion}\} \\
 & \ell \vdash \hat{\sigma} <: (\sigma'[\mu A'.\sigma'/A'] \ll \ell) \Downarrow \gamma' \\
\wedge & \epsilon \vdash \sigma'[\mu A'.\sigma'/A'] \Downarrow \varepsilon <: \sigma''[\mu A''.\sigma''/A''] \Downarrow \varepsilon \\
\Longrightarrow & \{\text{Lemma 3.6.3}\} \\
 & \ell \vdash \hat{\sigma} <: (\sigma'[\mu A'.\sigma'/A'] \ll \ell) \Downarrow \gamma' \\
\wedge & \ell \vdash (\sigma'[\mu A'.\sigma'/A'] \Downarrow \varepsilon) \ll \ell <: (\sigma''[\mu A''.\sigma''/A''] \Downarrow \varepsilon) \ll \ell \\
\Longrightarrow & \{\text{Proposition 3.6.2 (Distributivity)}\} \\
 & \ell \vdash \hat{\sigma} <: (\sigma'[\mu A'.\sigma'/A'] \ll \ell) \Downarrow \gamma' \\
\wedge & \ell \vdash (\sigma'[\mu A'.\sigma'/A'] \ll \ell) \Downarrow \varepsilon <: (\sigma''[\mu A''.\sigma''/A''] \ll \ell) \Downarrow \varepsilon \\
\Longrightarrow & \{\text{Lemma 3.6.4}\} \\
 & \ell \vdash \hat{\sigma} <: (\sigma'[\mu A'.\sigma'/A'] \ll \ell) \Downarrow \gamma' \\
\wedge & \ell \vdash (\sigma'[\mu A'.\sigma'/A'] \ll \ell) \Downarrow \varepsilon \Downarrow \gamma' <: (\sigma''[\mu A''.\sigma''/A''] \ll \ell) \Downarrow \varepsilon \Downarrow \gamma' \\
\Longrightarrow & \{\text{Proposition 3.6.1 (Associativity)}\} \\
 & \ell \vdash \hat{\sigma} <: (\sigma'[\mu A'.\sigma'/A'] \ll \ell) \Downarrow \gamma' \\
\wedge & \ell \vdash (\sigma'[\mu A'.\sigma'/A'] \ll \ell) \Downarrow \gamma' <: (\sigma''[\mu A''.\sigma''/A''] \ll \ell) \Downarrow \gamma' \\
\Longrightarrow & \{\text{coinduction hypothesis}\} \\
 & \ell \vdash \hat{\sigma} <: (\sigma''[\mu A''.\sigma''/A''] \ll \ell) \Downarrow \gamma' \\
\Longrightarrow & \{\text{subtyping rule}\} \\
 & \ell \vdash \hat{\sigma} <: \mu A''.\sigma''
\end{array}
$$

The case where $\hat{\sigma}$ and $\hat{\sigma}'$ are union types (but not $\hat{\sigma}''$) is similar. The remaining cases are straightforward. □

Corollary. *The binary relation* $<:$ *is transitive.*

3.6.5 Subtyping algorithm

The subtyping algorithm follows the standard algorithm of Gapeyev et al, but is adapted to accommodate union types and binding signatures. The SUBTYPE(σ, σ') function normalizes the types and calls FALSIFY with the empty tree address and an empty cache. The latter function attempts to falsify the assertion by searching for inconsistent assertions reachable by inversion of the inference rules.

The FALSIFY(ℓ, $\hat{\sigma}$, $\hat{\sigma}'$, C) function searches the graph of assertions reachable by inversion from $\ell \vdash \hat{\sigma} <: \hat{\sigma}'$, looking for inconsistent assertions. The cache A contains pairs of recursive types that have already been visited, to prevent looping on cycles in the graph. The functions FALSIFY-ANY(ℓ, $\overline{\hat{\sigma}_i}$, $\hat{\sigma}'$, A) and FALSIFY-ALL(ℓ, $\hat{\sigma}$, $\overline{\hat{\sigma}'_i}$, A) search the graph from the respective assertions $\ell \vdash \cup\{\overline{\hat{\sigma}_i}\} <: \hat{\sigma}'$ and $\ell \vdash \hat{\sigma} <: \cup\{\overline{\hat{\sigma}'_i}\}$.

Theorem 3.6.6. *Given contractive σ and σ', the algorithm* SUBTYPE*(σ, σ') terminates.*

Proof. The unfolding of a contractive syntax type is a regular tree, which has a finite number of distinct subtrees—in particular, a finite number of distinct μ-type subtrees. In the cases where FALSIFY recurs with a type that is not a syntactic subtree of one of its inputs, it only uses the operations $\sigma \Downarrow \gamma$ and $\sigma \ll \ell$, neither of which affects nested μ-types. Thus even in those cases, the set of distinct μ-type subtrees remains unchanged. Consequently, the lexicographic order of the number of pairs of μ-type subtrees of $\hat{\sigma}$ and $\hat{\sigma}'$ that are not in C (more significant) and the syntactic size of $\hat{\sigma}$ and $\hat{\sigma}'$ (less significant) forms a well-founded induction measure for FALSIFY. □

Algorithm 1 The subtyping algorithm

```
function $\textsc{Subtype}(\sigma, \sigma')$
    $\textsc{Falsify}(\epsilon, \sigma \Downarrow \varepsilon, \sigma' \Downarrow \varepsilon, \emptyset) \neq \text{Succ}$

function $\textsc{Falsify}(\ell, \hat{\sigma}, \hat{\sigma}', C)$
    match $(\hat{\sigma}, \hat{\sigma}')$ with
        $\mid (\mathsf{expr}{\downarrow}\gamma, \mathsf{expr}{\downarrow}\gamma) \Rightarrow \text{Fail } C$
        $\mid ((\sigma_0 \rightarrow \mathsf{expr}){\downarrow}\gamma, (\sigma'_0 \rightarrow \mathsf{expr}){\downarrow}\gamma) \Rightarrow$
            if $\textsc{Subtype}(\sigma'_0, \sigma_0)$ then
                Fail $C$
            else
                Succ
        $\mid (\mathsf{bvar}, \mathsf{bvar}) \Rightarrow \text{Fail } C$
        $\mid (\mathsf{data}, \mathsf{data}) \Rightarrow \text{Fail } C$
        $\mid (\texttt{()}, \texttt{()}) \Rightarrow \text{Fail } C$
        $\mid (\texttt{(}\hat{\sigma}_1 \texttt{ . } \hat{\sigma}_2\texttt{)}, \texttt{(}\hat{\sigma}'_1 \texttt{ . } \hat{\sigma}'_2\texttt{)}) \Rightarrow$
            match $\textsc{Falsify}(\mathsf{A}\ell, \hat{\sigma}_1, \hat{\sigma}'_1, C)$ with
                $\mid \text{Succ} \Rightarrow \text{Succ}$
                $\mid \text{Fail } C' \Rightarrow \textsc{Falsify}(\mathsf{D}\ell, \hat{\sigma}_2, \hat{\sigma}'_2, C')$
        $\mid (\hat{\sigma}_0{\uparrow}\gamma, \hat{\sigma}'_0{\uparrow}\gamma) \Rightarrow \textsc{Falsify}(\hat{\sigma}_0, \hat{\sigma}'_0)$
        $\mid ((\mu A.\sigma_0){\downarrow}\gamma, (\mu A'.\sigma'_0){\downarrow}\gamma) \Rightarrow$
            if $(\mu A.\sigma_0, \mu A'.\sigma'_0) \in C$ then
                Fail $C$
            else
                $\textsc{Falsify}(\epsilon, \sigma_0 \Downarrow \varepsilon, \sigma'_0 \Downarrow \varepsilon, \{(\mu A.\sigma_0, \mu A'.\sigma'_0)\} \cup C)$
        $\mid ((\mu A.\sigma_0){\downarrow}\gamma, \_) \Rightarrow \textsc{Falsify}(\ell, (\sigma_0[\mu A.\sigma_0/A] \ll \ell) \Downarrow \gamma, \sigma', C)$
        $\mid (\_, (\mu A'.\sigma'_0){\downarrow}\gamma) \Rightarrow \textsc{Falsify}(\ell, \sigma, (\sigma'_0[\mu A'.\sigma'_0/A'] \ll \ell) \Downarrow \gamma, C)$
        $\mid ((\cup\{\overline{\sigma_i}\}){\downarrow}\gamma, \_) \Rightarrow \textsc{Falsify-Any}(\ell, \overline{\sigma_i \Downarrow \gamma}, \sigma', C)$
        $\mid (\_, (\cup\{\overline{\sigma'_i}\}){\downarrow}\gamma) \Rightarrow \textsc{Falsify-All}(\ell, \sigma, \overline{\sigma'_i \Downarrow \gamma}, C)$
        $\mid \_ \Rightarrow \text{Succ}$
```

Algorithm 2 The subtyping algorithm (cont'd)

```
function FALSIFY-ANY($\ell$, $\overline{\hat{\sigma}_i}$, $\hat{\sigma}'$, $C$)
    match $\overline{\hat{\sigma}_i}$ with
        | $\varepsilon \Rightarrow$ Fail $C$
        | $\hat{\sigma}_1 :: \overline{\hat{\sigma}_j} \Rightarrow$
            match FALSIFY($\ell$, $\hat{\sigma}_1$, $\hat{\sigma}'$, $C$) with
                | Succ $\Rightarrow$ Succ
                | Fail $C' \Rightarrow$ FALSIFY-ANY($\ell$, $\overline{\hat{\sigma}_j}$, $\hat{\sigma}'$, $C'$)

function FALSIFY-ALL($\ell$, $\hat{\sigma}$, $\overline{\hat{\sigma}'_i}$, $C$)
    match $\overline{\hat{\sigma}'_i}$ with
        | $\varepsilon \Rightarrow$ Succ
        | $\hat{\sigma}'_1 :: \overline{\hat{\sigma}'_j} \Rightarrow$
            match FALSIFY($\ell$, $\hat{\sigma}$, $\hat{\sigma}'_1$, $C$) with
                | Succ $\Rightarrow$ FALSIFY-ALL($\ell$, $\hat{\sigma}$, $\overline{\hat{\sigma}'_j}$, $C$)
                | Fail $C' \Rightarrow$ Fail $C'$
```

CHAPTER 4

A Model of Typed Hygienic Macros

This chapter presents the syntax and semantics of λ_m, a model of typed hygienic macros.

4.1 A system view of macros

From an end-to-end view, evaluating a Scheme program is the process of transforming a symbolic tree src with the generic grammar:

$$src ::= sym \mid \texttt{()} \mid \texttt{(}src\ \texttt{.}\ src\texttt{)}$$

into an expression $expr$ of a known, fixed grammar:

$$expr \quad ::= \quad x \mid \texttt{(lambda (}\overline{x}\texttt{)}\ expr\texttt{)} \mid \texttt{(apply}\ \overline{expr}\texttt{)} \mid \texttt{(quote}\ src\texttt{)}$$

and then evaluating the expression in the usual way for any functional programming language. Although the surface language treats a compound node with any non-syntax operator as a function application, we imagine here that the internal language understood by the compiler requires an explicit apply operator to make the syntax more regular.

Because of the computational power of Scheme macros, it is not possible for a compiler to understand the internal structure of a source tree src before macro expansion. Traditional macro expansion instead gradually reveals the abstract syntax tree of a program by expanding from the outside in until no uses of macros remain [55]. Figure 4.1 presents an abstract view of the traditional workflow of Scheme systems.

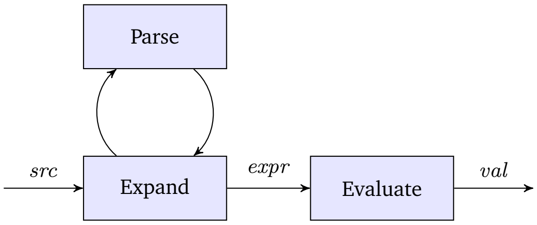


Figure 4.1: The traditional workflow of Scheme implementations.

In the λ_m model, types provide enough information to determine program structure without expanding macros. An input program is an *annotated S-expression*, where the syntactic structure of every subform is manifest. The result is the system view of Figure 4.2.

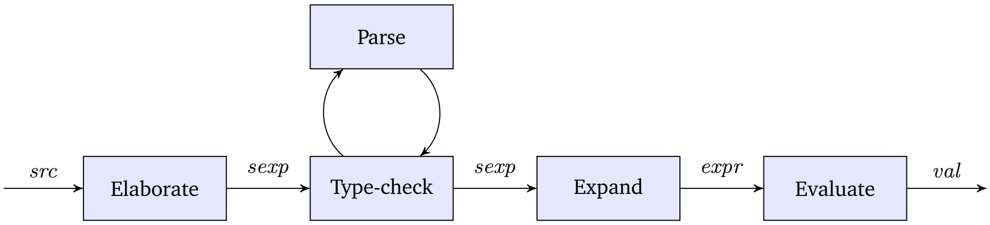


Figure 4.2: The workflow of λ_m.

In practice, it is sufficient for programmers to annotate only macro definitions; a process of *elaboration* can then transform an input program src to a fully annotated S-expression $sexp$. We return to the elaboration process in Chapter 7.

4.2 Annotated programs

The syntax of λ_m appears in Figure 4.3. Variables are drawn from the set $Variable = \mathbb{B} \uplus \mathbb{P}$, built from disjoint, countably infinite universes of atoms:

- *Base variables* $x, y \in \mathbb{B}$ model ordinary program variables, as bound by lambda or letrec-syntax, for example. Base variables may range over runtime values or macros.

$$
\begin{array}{lll}
\mathit{var} & ::= & x \mid a \\
\mathit{form} & ::= & \mathit{var} \mid \texttt{(}\mathsf{letrec\text{-}syntax}\ \texttt{((}x\ m\texttt{))}\ \mathit{form}\texttt{)} \mid \texttt{(}\mathit{mexp}\ \texttt{.}\ \mathit{sexp}\,\texttt{:}\,\sigma\texttt{)} \\
\mathit{mexp} & ::= & \mathit{prim} \mid \mathit{var} \mid m \\
\mathit{prim} & ::= & \mathsf{lambda} \mid \mathsf{apply} \mid \mathsf{quote} \\
m & ::= & \texttt{(}\mathsf{syntax\text{-}rules}\ \sigma \to \mathsf{expr}\ \texttt{(}\overline{\texttt{(}p\,\texttt{:}\,\hat{\sigma}\ \mathit{form}\texttt{)}}\texttt{))} \\
p & ::= & a \mid \texttt{()} \mid \texttt{(}p\ \texttt{.}\ p\texttt{)} \\
\mathit{sexp} & ::= & \mathit{form} \mid m \mid \mathit{prim} \mid \mathit{sym} \mid \texttt{()} \mid \texttt{(}\mathit{sexp}\ \texttt{.}\ \mathit{sexp}\texttt{)} \\
\mathit{data} & ::= & \mathit{sym} \mid a \mid \texttt{()} \mid \texttt{(}\mathit{data}\ \texttt{.}\ \mathit{data}\texttt{)}
\end{array}
$$

Figure 4.3: The syntax of λ_m.

- *Pattern variables* $a, b \in \mathbb{P}$ are bound by macro patterns and range over compile-time syntax.

A form is either a variable reference, a macro definition, or an application of a syntactic operator mexp to an annotated S-expression sexp. In the latter case, the application is explicitly annotated with the syntax type σ of the S-expression. This annotation provides the information needed to determine the syntax and binding structure of the argument. Syntactic operators mexp are either primitive operators (lambda, quote, or the explicit function application operator apply), macro references, or macros m. Macros contain a type annotation and a sequence of type-annotated patterns and templates. Patterns are trees of pattern variables. S-expressions are trees whose leaves can be forms, macros, primitives, or literal symbols sym. Finally, the category data is a subset of sexp which represents literal data; this is used to type-check operators such as quote that consume trees of literal data.

We define two useful type constants $\mathit{formals}$ and $\mathit{actuals}$ and equip the primitive operators with types:

$$
\begin{array}{rcl}
\mathit{formals} & \overset{\text{def}}{=} & \mu A.\cup\{\texttt{()}{\uparrow}\{\}, \texttt{(}\mathsf{bvar}\ \texttt{.}\ A\texttt{)}{\uparrow}\{\mathsf{A}\,{:}\,\textsc{var}, \mathsf{D}\,{:}\,\textsc{rib}\}\} \\
\mathit{actuals} & \overset{\text{def}}{=} & \mu A.\cup\{\texttt{()}, \texttt{(}\mathsf{expr}\ \texttt{.}\ A\texttt{)}\} \\
\mathsf{lambda} & : & \texttt{(}\mathit{formals}\ \mathsf{expr}{\downarrow}\{\mathsf{A}\,{:}\,\textsc{rib}\} :: \varepsilon\texttt{)} \to \mathsf{expr} \\
\mathsf{apply} & : & \texttt{(}\mathsf{expr}\ \texttt{.}\ \mathit{actuals}\texttt{)} \to \mathsf{expr} \\
\mathsf{quote} & : & \texttt{(}\mathsf{data}\texttt{)} \to \mathsf{expr}
\end{array}
$$

4.2.1 Type-directed induction

In Section 3.6.1, we described the use of contractiveness to ensure termination of subtyping even for definitions involving the unfolding of recursive types. Similarly, we exploit contractiveness for *type-directed* operations on syntax to ensure that our definitions do not admit infinite unfoldings. We define a well-founded *type-directed induction* measure on syntax trees. For a contractive syntax type σ and syntax tree $tree$, the measure is defined as the lexicographic ordering of the structural induction measure of $tree$ (more significant) and the following measure, defined inductively on the representation of σ:

$$
\begin{aligned}
&|\mathsf{expr}| = |\mathsf{data}| = |\mathsf{bvar}| = |\mathtt{()}| = |\sigma \to \mathsf{expr}| = 1\\
&|\mathtt{(}\sigma_1 \ \mathtt{.}\ \sigma_2\mathtt{)}| = 0\\
&|\sigma{\downarrow}\beta| = |\sigma{\uparrow}\beta| = |\sigma| + 1\\
&|\mu A.\sigma| = |\sigma| + 1\\
&|\cup\{\overline{\sigma_i}\}| = \max_i |\sigma_i|
\end{aligned}
$$

Lemma 4.2.1. *For all contractive $\mu A.\sigma$, $|\mu A.\sigma| > |\sigma[\mu A.\sigma/A]|$.*

Proof. By induction on the representation of σ. Since A an only appear under a pair type and $|\mathtt{(}\sigma_1[\mu A.\sigma/A] \ \mathtt{.}\ \sigma_2[\mu A.\sigma/A]\mathtt{)}| = 0$ for any $\mathtt{(}\sigma_1 \ \mathtt{.}\ \sigma_2\mathtt{)}$ within σ, the substitution does not change the size of σ. □

4.2.2 Parsing syntax trees

Given an S-expression $sexp$ and its annotated type σ, the process of determining the syntax and binding structure of $sexp$ is called "parsing," by analogy to parsing a program with an attribute grammar. The result of parsing is a *binding table*, which maps each node in the tree, identified by address, to the bindings exported by (defined within) that node. Binding tables Σ are defined by the grammar

$$
\begin{aligned}
\Sigma \quad &::= \quad \{\overline{\ell \mapsto attr}\}\\
attr \quad &::= \quad \beta \mid \mathrm{B}
\end{aligned}
$$

Each binding attribute is either a binding signature β, which may contain unresolved references to the exports of other nodes in the parse tree, or a collection of fully resolved bindings B (see Section 4.2.3).

The parsing operation is defined in Figure 4.4. The partial function $\mathcal{P}$ is defined by type-directed induction on a syntax type σ and an S-expression $sexp$, producing a fully-resolved binding table Σ. In the terminology of attribute grammars, we can consider the syntax type σ as an attribute grammar, the S-expression as the parse input, and the binding table as the solution to the attribute grammar. In the binding table, each tree address ℓ uniquely identifies a node in the parse tree.[1] This top-level function $\mathcal{P}$ delegates to a $parse$ function to parse the current addressing region, with an additional argument representing the current address in that region. The result is a table which may contain unresolved binding signatures β. These signatures are then resolved with the $resolve$ function, described in Section 4.2.3.

We now describe each rule of the $parse$ function in turn. When the syntax tree is a pattern variable a, the variable itself is the binding, unless the syntax type σ has the binding type NONE (i.e., does not produce exports). The binding type δ is determined by the well-formedness judgment on types (see Chapter 5). A base variable x can be the binding for the syntax type bvar. The types expr, data, $\sigma \rightarrow$ expr and () never export bindings. Pairs can be parsed by pair types by combining the results of parsing the left and right sides, with the current address producing no exports. Import signatures are irrelevant to the parsing process and are discarded. Export types recur and set the bindings of the current address to the exported signature β. Recursive types and union types introduce new addressing regions, so they both recur in a fresh root address to produce new subtables; recursive types unfold the type, and union types use the first subtable that successfully parses.

[1] The unique locations serve the same role as a node's object identity or memory location in a parse tree produced by yacc or bison.

$$
\begin{array}{lcl}
\mathcal{P}(\sigma, \mathit{sexp}) & = & \{\overline{\ell \mapsto \mathit{resolve}(\Sigma, \Sigma(\ell))}\} \\
\quad \text{where } \Sigma = \mathit{parse}(\sigma, \mathit{sexp}, \epsilon) & & \\
 & & \\
\mathit{parse}(\sigma, a, \ell) & = & \{\ell \mapsto \mathcal{T}(a, \delta)\} \\
\quad \text{where } \ell \vdash_{\uparrow} \sigma : \{\ell \mapsto \delta\} & & \\
\mathit{parse}(\mathsf{bvar}, x, \ell) & = & \{\ell \mapsto x\,{:}\,\mathsf{expr}\} \\
\mathit{parse}(\mathsf{expr}, \mathit{form}, \ell) & = & \{\ell \mapsto \bullet\} \\
\mathit{parse}(\mathsf{data}, \mathit{data}, \ell) & = & \{\ell \mapsto \bullet\} \\
\mathit{parse}(\sigma \rightarrow \mathsf{expr}, \mathit{mexp}, \ell) & = & \{\ell \mapsto \bullet\} \\
\mathit{parse}(\texttt{()}, \texttt{()}, \ell) & = & \{\ell \mapsto \bullet\} \\
\mathit{parse}(\texttt{(}\sigma_1 \texttt{ . } \sigma_2\texttt{)}, \texttt{(}\mathit{sexp}_1 \texttt{ . } \mathit{sexp}_2\texttt{)}, \ell) & = & (\Sigma_1 \cup \Sigma_2)[\ell \mapsto \bullet] \\
\quad \text{where } \Sigma_1 = \mathit{parse}(\sigma_1, \mathit{sexp}_1, \ell\mathsf{A}) & & \\
\quad \text{and } \Sigma_2 = \mathit{parse}(\sigma_2, \mathit{sexp}_2, \ell\mathsf{D}) & & \\
\mathit{parse}(\sigma{\downarrow}\beta, \mathit{sexp}, \ell) & = & \mathit{parse}(\sigma, \mathit{sexp}, \ell) \\
\mathit{parse}(\sigma{\uparrow}\beta, \mathit{sexp}, \ell) & = & \Sigma[\ell \mapsto \beta] \\
\quad \text{where } \Sigma = \mathit{parse}(\sigma, \mathit{sexp}, \ell) & & \\
\quad \text{and } \ell \notin \mathit{dom}(\Sigma) \vee \Sigma(\ell) = \bullet & & \\
\mathit{parse}(\mu A.\sigma, \mathit{sexp}, \ell) & = & \{\ell \mapsto \Sigma(\epsilon)\} \\
\quad \text{where } \Sigma = \mathcal{P}(\sigma[\mu A.\sigma/A], \mathit{sexp}) & & \\
\mathit{parse}(\cup\{\overline{\sigma_i}\}, \mathit{sexp}, \ell) & = & \{\ell \mapsto \Sigma(\epsilon)\} \\
\quad \text{where } \Sigma = \mathcal{P}(\sigma_i, \mathit{sexp}) \text{ for smallest } i \text{ s.t. } \mathcal{P}(\sigma_i, \mathit{sexp}) \Downarrow & & \\
 & & \\
\mathcal{T}(a, \textsc{none}) & = & \bullet \\
\mathcal{T}(a, \textsc{var}) & = & a\,{:}\,\textsc{var} \\
\mathcal{T}(a, \textsc{rib}) & = & \{a\,{:}\,\textsc{rib}\} \\
\mathcal{T}(a, \textsc{env}) & = & a@\varepsilon
\end{array}
$$

Figure 4.4: S-expression parsing.

4.2.3 Computing environment structure

Once a binding table has been constructed, a syntax node's exported variable or binding signature can be found by looking it up in the table. Of course, if the table entry is a binding signature β, the signature may refer to other nodes' addresses. So after producing a binding table, the parsing process resolves all node addresses in binding signatures to produce fully resolved bindings for each node. Put differently, binding signatures are *expressions* (akin to the expressions of an attribute grammar) that must be recursively evaluated in the context of a given binding table.

Binding signature resolution produces fragments of environment structure: variables, ribs, or sequences of ribs. This environment structure plays a key role in many of the structures and operations of λ_m. Most notably, environments track program context for type checking, which we return to in Chapter 5. But insofar as environments detail the binding structure of nodes in a syntax tree, they are also used for many scope-related operations, such as computing free and bound variables.

Environments Γ are composed of ribs P and pattern variables a:

$$\Gamma ::= \varepsilon \mid \mathrm{P} :: \Gamma \mid a@\Gamma$$

Pattern variables stand in for binding structure that comes from macro arguments. For example, in the **let**∗ macro:

Example 25

```
;; clauses =
;;    μA.∪{()↑ε,((bvar expr) . A↓{AA:VAR})↑D@{AA:VAR} :: ε}

;; (clauses expr↓A@ε) → expr
(define-syntax let∗
  (syntax-rules ()
    [(let∗ () e) e]
    [(let∗ ((a1 e1) . rest) e2)
     (let ([a1 e1])
       (let∗ rest e2))]))
```

the pattern variable *rest* encapsulates a sequence of environment ribs that will be determined when the macro is applied. In order to type-check the use of *e2* in the macro template, the environment is extended with the pattern variable *rest*, indicating that all variables bound within the clauses ranged over by *rest*—whatever they may be—will be in scope for the body expression *e2*.

Environment ribs are unordered mappings from variables to types:

$$\begin{array}{rcl} \mathrm{P} & ::= & \{\overline{\mathrm{V}}\} \\ \mathrm{V} & ::= & x : \tau \mid a : \delta \end{array}$$

A rib in a well-formed Scheme program binds each variable only once, so each variable in a rib is unique. Note that base variables x map to form types like expr or $\sigma \rightarrow$ expr, whereas pattern variables a map to types of bindings such as VAR or RIB.

Pattern variables can appear in ribs in two different ways. A pattern variable that ranges over individual identifiers has type VAR. As an example, the variable x in the definition of **for** from Example 13 of Section 2.2.1 would appear in its rib with binding type VAR. A pattern variable that ranges over ribs is assigned the type RIB. Consider an implementation of the **let** macro:

Example 26

```
;; clauses =
;;    μA.∪{()↑∅, ((bvar expr) . A)↑{AA:VAR, D:RIB}}

;; (clauses expr↓{A:RIB}) → expr
(define-syntax let
  (syntax-rules ()
    [(let cs e)
     (unzip-let () () cs e)]))

;; (formals actuals clauses expr↓{A:RIB, ADD:RIB}) → expr
(define-syntax unzip-let
  (syntax-rules ()
    [(unzip-let as es ((a1 e1) . rest) e)
     (unzip-let (a1 . as) (e1 . es) rest e)]
    [(unzip-let as es () e)
     ((lambda as e) . es)]))
```

This implementation uses a macro **unzip-let** to separate the the bound variables and initialization expressions from binding clauses and binds the variables in a single rib with **lambda**. In the recursive call to **unzip-let**, the pattern variables *a1*, *as*, and *rest* are combined in a single rib:

$$\{\mathit{a1} : \mathrm{VAR}, \mathit{as} : \mathrm{RIB}, \mathit{rest} : \mathrm{RIB}\}$$

A collection of fully resolved bindings B can then be any such fragment of environment, or $\bullet$ for no bindings at all:

$$\mathrm{B} ::= \bullet \mid \mathrm{V} \mid \mathrm{P} \mid \Gamma$$

The binding resolution function $\mathit{resolve}$ takes a binding table Σ and either a binding attribute attr or tree address ℓ to produce a fully resolved B. We

$$
\begin{array}{lcl}
\mathit{resolve}(\Sigma, \ell) & = & \mathit{resolve}(\Sigma|_{\{\ell\}(\prec)}, \mathit{attr}) \\
\quad \text{if } \Sigma(\ell) = \mathit{attr} & & \\
\mathit{resolve}(\Sigma, \mathrm{B}) & = & \mathrm{B} \\
\mathit{resolve}(\Sigma, \{\ell : \textsc{var}\} \cup \rho) & = & \{\Sigma(\ell)\} \uplus \mathit{resolve}(\Sigma, \rho) \\
\mathit{resolve}(\Sigma, \{\ell : \textsc{rib}\} \cup \rho) & = & \mathit{resolve}(\Sigma, \ell) \uplus \mathit{resolve}(\Sigma, \rho) \\
\mathit{resolve}(\Sigma, \rho :: \gamma) & = & \mathit{resolve}(\Sigma, \rho) :: \mathit{resolve}(\Sigma, \gamma) \\
\mathit{resolve}(\Sigma, \ell @ \gamma) & = & \mathit{resolve}(\Sigma, \ell), \mathit{resolve}(\Sigma, \gamma)
\end{array}
$$

Figure 4.5: Resolution of bindings.

sometimes use a curried form:

$$
\begin{array}{lcl}
\Sigma^{\mathcal{R}}(\mathit{attr}) & = & \mathit{resolve}(\Sigma, \mathit{attr}) \\
\Sigma^{\mathcal{R}}(\ell) & = & \mathit{resolve}(\Sigma, \ell)
\end{array}
$$

The resolution function is defined in Figure 4.5. Resolving a tree address ℓ recursively resolves the attribute found at that address in the table $\Sigma|_{\{\ell\}(\prec)}$, i.e. the table Σ restricted to the domain of addresses strictly prefixed by ℓ. This captures the notion that a binding signature may only refer to addresses of strict sub-trees. In Chapter 5, we return to this point and describe well-formedness criteria for syntax types to ensure signatures only refer to valid addresses. The next three rules show that variable bindings, ribs, and environments are already fully resolved. Rib resolution recursively resolves the components of a rib and combines the results with the disjoint map-union operation $\uplus$. Similarly, environment resolution recursively resolves the components of an environment signature and recombines them with the corresponding environment constructors.

Theorem 4.2.2. *For any attr and Σ, $\mathit{resolve}(\Sigma, \mathit{attr})$ terminates.*

Proof. In the first rule, the attribute expression may grow but the binding table necessarily shrinks; the table $\Sigma|_{\{\ell\}(\prec)}$ is smaller than Σ because it at least removes ℓ from the domain. In all other rules, the attributes in recursive calls decrease in size. Thus the lexicographic ordering of binding tables (more significant) and attributes (less significant) forms a well-founded induction measure for the algorithm. □

4.2.3.1 Duplicate variables

The definition of resolution ensures that ribs never contain duplicate variables by means of a disjoint map-union operation. This means that all operations that depend on resolution—including parsing and macro expansion—could fail in the presence of duplicate variables. However, in Chapter 5, we demonstrate the type rules that ensure that well-typed programs never contain (or expand into programs containing) duplicate variables in ribs. This ensures that, for well-typed programs, all operations on S-expressions are defined. Consequently an invariant of our type system is that macro expansion never gets "stuck" (i.e., faults) due to duplicate-variable errors. We return to this point in Chapter 6.

4.3 Binding and alpha-equivalence

The ability to parse macro applications without expanding them opens up source programs to all manner of analysis. Most directly, we can perform the usual operations of lexically scoped programming languages such as computing the free and bound variables of a term. Moreover, we can rename bound variables and their corresponding references (perform α-conversions) and compare programs for syntactic equivalence up to variable names (α-equivalence).

4.3.1 Free and bound variables

The free base variables of a term are computed with the fv function defined in Figure 4.6. On a $form$ or $mexp$, the operation is unary; for annotated S-expressions $sexp$, the operation requires two additional arguments: a syntax

$$
\begin{array}{lcl}
\mathit{fv}(x) & = & \{x\} \\
\mathit{fv}(a) & = & \emptyset \\
\mathit{fv}(\textsf{(letrec-syntax ((}x\ m\textsf{))}\ \mathit{form}\textsf{)}) & = & (\mathit{fv}(m) \cup \mathit{fv}(\mathit{form})) - \{x\} \\
\mathit{fv}(\textsf{(}\mathit{mexp}\ \textsf{.}\ \mathit{sexp}\,\textsf{:}\,\sigma\textsf{)}) & = & \mathit{fv}(\mathit{mexp}) \cup \mathit{fv}(\mathit{sexp})^{\sigma}_{\mathcal{P}(\sigma,\mathit{sexp})} \\
 & & \\
\mathit{fv}(\mathit{prim}) & = & \emptyset \\
\mathit{fv}(\textsf{(syntax-rules}\ \tau\ \textsf{(}\overline{\textsf{(}p_i\,\textsf{:}\,\hat{\sigma}_i\ \mathit{form}_i\textsf{)}}\textsf{))}) & = & \bigcup_i \mathit{fv}(\mathit{form}_i) \\
 & & \\
\mathit{fv}(a)^{\sigma}_{\Sigma} & = & \emptyset \\
\mathit{fv}(\mathit{form})^{\textsf{expr}}_{\Sigma} & = & \mathit{fv}(\mathit{form}) \\
\mathit{fv}(\mathit{mexp})^{\sigma\rightarrow\textsf{expr}}_{\Sigma} & = & \mathit{fv}(\mathit{mexp}) \\
\mathit{fv}(\mathit{var})^{\textsf{bvar}}_{\Sigma} & = & \emptyset \\
\mathit{fv}(\mathit{data})^{\textsf{data}}_{\Sigma} & = & \emptyset \\
\mathit{fv}(\textsf{()})^{\textsf{()}}_{\Sigma} & = & \emptyset \\
\mathit{fv}(\textsf{(}\mathit{sexp}_1\ \textsf{.}\ \mathit{sexp}_2\textsf{)})^{\textsf{(}\sigma_1\ \textsf{.}\ \sigma_2\textsf{)}}_{\Sigma} & = & \mathit{fv}(\mathit{sexp}_1)^{\sigma_1}_{\Sigma} \cup \mathit{fv}(\mathit{sexp}_2)^{\sigma_2}_{\Sigma} \\
\mathit{fv}(\mathit{sexp})^{\sigma\downarrow\beta}_{\Sigma} & = & \mathit{fv}(\mathit{sexp})^{\sigma}_{\Sigma} - \{\overline{x}\} \\
\quad \text{where } \{\overline{x}, \overline{a}\} = \mathit{dom}(\mathit{resolve}(\Sigma, \beta)) & & \\
\mathit{fv}(\mathit{sexp})^{\sigma\uparrow\beta}_{\Sigma} & = & \mathit{fv}(\mathit{sexp})^{\sigma}_{\Sigma} \\
\mathit{fv}(\mathit{sexp})^{\mu A.\sigma}_{\Sigma} & = & \mathit{fv}(\mathit{sexp})^{\sigma[\mu A.\sigma/A]}_{\mathcal{P}(\sigma[\mu A.\sigma/A],\mathit{sexp})} \\
\mathit{fv}(\mathit{sexp})^{\cup\{\overline{\sigma_i}\}}_{\Sigma} & = & \mathit{fv}(\mathit{sexp})^{\sigma_i}_{\mathcal{P}(\sigma_1,\mathit{sexp})} \\
\quad \text{for smallest } i \text{ s.t. } \mathcal{P}(\sigma_1, \mathit{sexp}) \Downarrow & &
\end{array}
$$

Figure 4.6: Free base variables of a term.

type σ and a binding table Σ. This is of course precisely the information provided by parsing, which allows the operation to traverse the otherwise unstructured syntax tree.

For forms and macro expressions, the definition is straightforward: a base variable x defines a singleton set of free variables; the free variables of a macro definition form are found by taking the free variables of the subterms and removing the bound macro variable; a macro's free variables occur in its right-hand sides. The macro application form is the interesting case: the free variables are found by taking the free variables of the operator and operand; the latter's free variables are computed using the multiary fv operator, using the annotated syntax type σ, the bindings table Σ computed by parsing the argument S-expression, and the root tree address ϵ.

$$
\begin{array}{lcl}
bv(x) & = & \emptyset \\
bv(a) & = & \emptyset \\
bv(\texttt{(letrec-syntax ((}x\ m\texttt{))}\ form\texttt{)}) & = & bv(m) \cup bv(form) \cup \{x\} \\
bv(\texttt{(}mexp\ \texttt{.}\ sexp\,\texttt{:}\,\sigma\texttt{)}) & = & bv(mexp) \cup bv(sexp)^{\sigma}_{\mathcal{P}(\sigma, sexp)} \\
\\
bv(prim) & = & \emptyset \\
bv(\texttt{(syntax-rules}\ \tau\ \texttt{(}\overline{\texttt{(}p_i\,\texttt{:}\,\hat{\sigma}_i\ form_i\texttt{)}}\texttt{))}) & = & \bigcup_i bv(form_i) \\
\\
bv(a)^{\sigma}_{\Sigma} & = & \emptyset \\
bv(form)^{\mathsf{expr}}_{\Sigma} & = & bv(form) \\
bv(mexp)^{\sigma \to \mathsf{expr}}_{\Sigma} & = & bv(mexp) \\
bv(x)^{\mathsf{bvar}}_{\Sigma} & = & \{x\} \\
bv(data)^{\mathsf{data}}_{\Sigma} & = & \emptyset \\
bv(\texttt{()})^{\texttt{()}}_{\Sigma} & = & \emptyset \\
bv(\texttt{(}sexp_1\ \texttt{.}\ sexp_2\texttt{)})^{\texttt{(}\sigma_1\ \texttt{.}\ \sigma_2\texttt{)}}_{\Sigma} & = & bv(sexp_1)^{\sigma_1}_{\Sigma} \cup bv(sexp_2)^{\sigma_2}_{\Sigma} \\
bv(sexp)^{\sigma \Downarrow \beta}_{\Sigma} & = & bv(sexp)^{\sigma}_{\Sigma} \cup \{\overline{x}\} \\
\quad \text{where } \{\overline{x}, \overline{a}\} = dom(resolve(\Sigma, \beta)) & & \\
bv(sexp)^{\sigma \Uparrow \beta}_{\Sigma} & = & bv(sexp)^{\sigma}_{\Sigma} \cup \{\overline{x}\} \\
\quad \text{where } \{\overline{x}, \overline{a}\} = dom(resolve(\Sigma, \beta)) & & \\
bv(sexp)^{\mu A.\sigma}_{\Sigma} & = & bv(sexp)^{\sigma[\mu A.\sigma/A]}_{\mathcal{P}(\sigma[\mu A.\sigma/A], sexp)} \\
bv(sexp)^{\cup\{\overline{\sigma_i}\}}_{\Sigma;\ell} & = & bv(sexp)^{\sigma_i}_{\mathcal{P}(\sigma_i, sexp)} \\
\quad \text{for smallest } i \text{ s.t. } \mathcal{P}(\sigma_i, sexp) \Downarrow & &
\end{array}
$$

Figure 4.7: Bound base variables of a term.

For annotated S-expressions, the definition of fv follows the structure of the annotated type σ. Pattern variables contain no base variables and so have empty free variable sets. At type expr, a form's free variables are computed using the unary fv operation; likewise for a macro expression at macro type. At type bvar, a variable is in binding position; thus it contains no free variables, even if it is itself a base variable. Quoted data contains no free variables, nor does the nil syntax node (). The free variables of pairs are found by structural recursion, updating the tree address accordingly. The most interesting case is that of import types: the binding signature β determines the variables being brought into scope. These can be computed by evaluating the binding signature with the binding table Σ. Thus the free

$$\frac{\ell \in bp(\sigma) \qquad \ell \neq \epsilon}{\ell \in bp(\sigma{\uparrow}\beta)} \qquad \frac{\ell \in bp(\sigma)}{\ell \in bp(\sigma{\downarrow}\beta)}$$

$$\frac{\ell \in bp(\sigma_1)}{\ell\mathsf{A} \in bp(\mathtt{(}\sigma_1 \mathbin{\mathtt{.}} \sigma_2\mathtt{)})} \qquad \frac{\ell \in bp(\sigma_2)}{\ell\mathsf{D} \in bp(\mathtt{(}\sigma_1 \mathbin{\mathtt{.}} \sigma_2\mathtt{)})}$$

$$\frac{\begin{array}{c}\ell \neq \epsilon \\ \ell \in bp(\sigma[\mu A.\sigma/A])\end{array}}{\ell \in bp(\mu A.\sigma)} \qquad \frac{\begin{array}{c}\ell \neq \epsilon \\ \exists i.\ell \in bp(\sigma_i)\end{array}}{\ell \in bp(\cup\{\overline{\sigma_i}\})} \qquad \frac{}{\epsilon \in bp(\mathsf{bvar})}$$

Figure 4.8: Binding positions of a syntax type.

variables of the term are the free variables found recursively except for those that are bound here. Export types do not affect the variables currently in scope, so the binding signature is ignored. Finally, recursive types and union types recur with the nested sub-table found at the current address.

The operation for determining the bound variables bv of a term, presented in Figure 4.7, is defined similarly. We describe here only the different cases. A free reference to a base variable x is of course not bound; by contrast, at type bvar a base variable is included in the result of bv. A macro definition adds its bound macro variable rather than removing it from the result set. Both import types and export types specify bound variables, so in both cases their base variables are included in the result set.

4.3.2 Binding positions

We can also use syntax types to determine the *binding positions* of a syntax tree, i.e., the addresses of nodes that provide exported bindings. Given a syntax type σ and an address ℓ, we can determine whether ℓ is in the set of binding positions $bp(\sigma)$ using the definition of Figure 4.8. Note that the definition is type-directed; since types are contractive and the address ℓ is made smaller at pair types, the definition is well-founded. The definition searches through the type tree for the type at position ℓ; if that type is bvar

$$
\begin{array}{lll}
x\{z/x\} & = z & \\
y\{z/x\} & = y & \text{if } x \neq y \\
a\{z/x\} & = a & \\
\mathit{sym}\{z/x\} & = \mathit{sym} & \\
\mathit{prim}\{z/x\} & = \mathit{prim} & \\
\multicolumn{3}{l}{(\texttt{syntax-rules}\ \tau\ (\overline{(p_i : \hat{\sigma}_i\ \mathit{form}_i)}))\{z/x\}} \\
\multicolumn{3}{l}{\quad = (\texttt{syntax-rules}\ \tau\ (\overline{(p_i : \hat{\sigma}_i\ \mathit{form}_i\{z/x\})}))} \\
\multicolumn{3}{l}{(\texttt{letrec-syntax}\ ((x'\ m))\ \mathit{form})\{z/x\}} \\
\multicolumn{3}{l}{\quad = (\texttt{letrec-syntax}\ ((x'\{z/x\}\ m\{z/x\}))\ \mathit{form}\{z/x\})} \\
\multicolumn{3}{l}{(\mathit{mexp}\ .\ \mathit{sexp} : \sigma)\{z/x\}} \\
\multicolumn{3}{l}{\quad = (\mathit{mexp}\{z/x\}\ .\ \mathit{sexp}\{z/x\} : \sigma)} \\
()\{z/x\} & = () & \\
\multicolumn{3}{l}{(\mathit{sexp}_1\ .\ \mathit{sexp}_2)\{z/x\}} \\
\multicolumn{3}{l}{\quad = (\mathit{sexp}_1\{z/x\}\ .\ \mathit{sexp}_2\{z/x\})}
\end{array}
$$

Figure 4.9: Uniform variable substitution.

then ℓ is indeed a binding position.

4.3.3 Alpha-equivalence

Because parsing S-expressions makes it possible to understand the binding structure of programs, we can not only recognize free and bound variables, but also compare terms for equality up to different choices of variable names. That is, using the parsing and attribute evaluation algorithms, it is possible to define a sound α-equivalence relation for λ_m. Figure 4.11 provides such a definition.

The definition of α-equivalence is built on top of the *uniform substitution* operation $\mathit{sexp}\{z/x\}$, which is defined purely structurally on terms, i.e., with no knowledge of the shape or binding structure of macros. Uniform substitution is defined in Figure 4.9. The α-equivalence relation also makes use of a "freshness" relation:

$$z \# \mathit{sexp} \stackrel{\text{def}}{=} z \notin \mathit{supp}(\mathit{sexp})$$

where the *support* of an S-expression $\mathit{supp}(\mathit{sexp})$ is given by the definition in

$$
\begin{array}{lcl}
\mathit{supp}(x) & = & \{x\} \\
\mathit{supp}(a) & = & \emptyset \\
\mathit{supp}(\mathit{sym}) & = & \emptyset \\
\mathit{supp}(\mathit{prim}) & = & \emptyset \\
\mathit{supp}(\texttt{(}\textsf{syntax-rules}\ \tau\ \texttt{(}\overline{\texttt{(}p_i\,\texttt{:}\,\hat{\sigma}_i\ \mathit{form}_i\texttt{)}}\texttt{))}) & = & \bigcup_i \mathit{supp}(\mathit{form}_i) \\
\mathit{supp}(\texttt{(}\textsf{letrec-syntax}\ \texttt{((}x\ m\texttt{))}\ \mathit{form}\texttt{)}) & & \\
\quad = \{x\} \cup \mathit{supp}(m) \cup \mathit{supp}(\mathit{form}) & & \\
\mathit{supp}(\texttt{()}) & = & \emptyset \\
\mathit{supp}(\texttt{(}\mathit{sexp}_1\ \texttt{.}\ \mathit{sexp}_2\texttt{)}) & = & \mathit{supp}(\mathit{sexp}_1) \cup \mathit{supp}(\mathit{sexp}_2)
\end{array}
$$

Figure 4.10: The support of a term.

Figure 4.10. We also use the following shorthands:

$$
\begin{array}{rcl}
\overline{z_i}\#\mathit{sexp} & \stackrel{\text{def}}{=} & \forall i.z_i\#\mathit{sexp} \wedge \forall i \neq j.z_i \neq z_j \\
\overline{z_i}\#\overline{\mathit{sexp}_j} & \stackrel{\text{def}}{=} & \forall j.\overline{z_i}\#\mathit{sexp}_j
\end{array}
$$

Let us examine the rules in detail. Rules [A-VAR], [A-PVAR], [A-SYM], and [A-PRIM] state that free variables, symbols, and primitives must be identical to be α-equivalent. Rule [A-MACDEF] compares letrec-syntax forms by unifying the names of their bindings: given a set of fresh names $\overline{z_i}$, two macro-definition forms are α-equivalence if their macro bindings and body expressions are α-equivalent after substituting the fresh variables $\overline{z_i}$ for their respective variable bindings.

The rule [A-MACAPP] is central. To compare two macro applications for α-equivalence, we must compare their operators and operands. The operators are simply compared inductively. The operands, however, may bind variables in arbitrary ways. Unifying the bindings of two S-expressions proceeds in two steps:

1. Freshen binding occurrences of base variables bound by this form.

2. Convert all corresponding references to their fresh names.

The first step involves enumerating addresses $\overline{\ell_i}$ of binding positions in the form and selecting the base variables $\overline{x_i}, \overline{x'_i}$ bound at those locations in the

[A-VAR] $$\frac{}{x =_\alpha x}$$ [A-PVAR] $$\frac{}{a =_\alpha a}$$ [A-SYM] $$\frac{}{\mathit{sym} =_\alpha \mathit{sym}}$$ [A-PRIM] $$\frac{}{\mathit{prim} =_\alpha \mathit{prim}}$$

[A-MACDEF]

$$\frac{\begin{array}{c} z\#\mathit{form}, m, \mathit{form}', m' \\ m\{z/x\} =_\alpha m'\{z/x'\} \qquad \mathit{form}\{z/x\} =_\alpha \mathit{form}'\{z/x'\} \end{array}}{\texttt{(letrec-syntax ((}x\ m\texttt{))}\ \mathit{form}\texttt{)} =_\alpha \texttt{(letrec-syntax ((}x'\ m'\texttt{))}\ \mathit{form}'\texttt{)}}$$

[A-MACAPP]

$$\frac{\begin{array}{l} \{\overline{\ell_i \mapsto x_i}\} = \mathit{bindings}(\sigma, \mathit{sexp}) \qquad \{\overline{\ell_i \mapsto x_i'}\} = \mathit{bindings}(\sigma, \mathit{sexp}') \\ \overline{z_i}\#\mathit{sexp}, \mathit{sexp}' \qquad \mathit{sexp}_1 = \mathit{sexp}[\overline{\ell_i \mapsto z_i}] \qquad \mathit{sexp}_2 = \mathit{sexp}'[\overline{\ell_i \mapsto z_i}] \\ \mathit{mexp} =_\alpha \mathit{mexp}' \qquad \mathit{sexp}_1\{\overline{z_i/x_i}\}^{\sigma}_{\mathcal{P}(\mathit{sexp}_1,\sigma)} =_\alpha \mathit{sexp}_2\{\overline{z_i/x_i'}\}^{\sigma}_{\mathcal{P}(\mathit{sexp}_2,\sigma)} \end{array}}{\texttt{(}\mathit{mexp}\ \texttt{.}\ \mathit{sexp}\texttt{:}\sigma\texttt{)} =_\alpha \texttt{(}\mathit{mexp}'\ \texttt{.}\ \mathit{sexp}'\texttt{:}\sigma\texttt{)}}$$

[A-MACRO]

$$\frac{\forall i.\mathit{form}_i =_\alpha \mathit{form}'_i}{\texttt{(syntax-rules}\ \tau\ \texttt{(}\overline{\texttt{(}p_i\texttt{:}\hat{\sigma}_i\ \mathit{form}_i\texttt{)}}\texttt{))} =_\alpha \texttt{(syntax-rules}\ \tau\ \texttt{(}\overline{\texttt{(}p_i\texttt{:}\hat{\sigma}_i\ \mathit{form}'_i\texttt{)}}\texttt{))}}$$

[A-NULL] $$\frac{}{\texttt{()} =_\alpha \texttt{()}}$$ [A-CONS] $$\frac{\mathit{sexp}_1 =_\alpha \mathit{sexp}_2 \qquad \mathit{sexp}_1' =_\alpha \mathit{sexp}_2'}{\texttt{(}\mathit{sexp}_1\ \texttt{.}\ \mathit{sexp}_1'\texttt{)} =_\alpha \texttt{(}\mathit{sexp}_2\ \texttt{.}\ \mathit{sexp}_2'\texttt{)}}$$

Figure 4.11: The α-equivalence relation for λ_m.

respective S-expressions:

$$\mathit{bindings}(\sigma, \mathit{sexp}) \overset{\text{def}}{=} \{\ell \mapsto \mathit{sexp}.\ell \mid \ell \in \mathit{bp}(\sigma), \mathit{sexp}.\ell \in \mathbb{B}\}$$

The modified S-expressions $\mathit{sexp}_1, \mathit{sexp}_2$ are formed by replacing these binding occurrences with fresh variables $\overline{z_i}$. The second step performs a *type-directed α-conversion*, which requires parsing the S-expressions with their syntax type in order to traverse their subterms. The definition of type-directed conversion—given in the next section—is subtle, but the intuition is reasonably straightforward: each fresh variable z_i is substituted for its corresponding base variable x_i or x_i' in all subterms where the variable is in scope.

The rule [A-MACRO] compares macros by comparing their right-hand side expressions. Note that this relation does not take into account changes of pattern variable; we assume here that choices of pattern variable are inflexible. It is possible to define a more general notion of α-equivalence that allows for renaming of pattern variables, but for our hygienic semantics this more restrictive relation suffices. Finally, [A-NULL] compares the nil syntax node for equality, and [A-CONS] compares pairs structurally.

4.3.3.1 Type-directed alpha-conversion

The [A-MACAPP] rule of α-equivalence relies on a type-directed α-conversion operator for S-expressions. The operation, written $sexp\{\overline{z_i/x_i}\}^{\sigma}_{\Sigma}$, substitutes variables z_i for x_i where they are in scope within an S-expression $sexp$ of arbitrary shape; the operation uses a syntax type σ, binding table Σ and address ℓ to traverse the syntax tree.

It is worth taking a moment to examine the assumptions we make in the definitions of these operations. First, we wish to ensure that every z is uniquely chosen, i.e., if $z_i = z_j$ then $i = j$. Second, let us assume that every rib in a variable renaming is free of duplicate variable names. For example, the operation is undefined on the ill-formed expression

(**lambda** (*x x*) *x*)

Example 27

Fortunately, as we explained in Section 4.2.3.1 above, parsing always produces well-formed bindings tables, which never contain duplicate variable bindings in a single rib.

Now let us turn to the definition in Figure 4.12. Pattern variables are never affected by renaming base variables. At type bvar, base variables are binding occurrences and therefore never α-converted. For import types, we first perform any inner α-conversions (since inner bindings potentially shadow outer bindings), and then perform a *bindings-directed α-conversion*, defined in the next section, based on evaluating the imported binding signature β. Exported binding signatures do not affect the current scope and are ignored. Pairs are converted by structural recursion, updating the tree ad-

$$\begin{array}{rcl}
a\{\overline{z/x}\}^{\sigma}_{\Sigma} & = & a \\
y\{\overline{z/x}\}^{\mathsf{bvar}}_{\Sigma} & = & y \\
sexp\{\overline{z/x}\}^{\sigma\downarrow\beta}_{\Sigma} & = & sexp\{\overline{z/x}\}^{\sigma}_{\Sigma}\{\overline{z/x}\}^{resolve(\Sigma,\beta)} \\
sexp\{\overline{z/x}\}^{\sigma\uparrow\beta}_{\Sigma} & = & sexp\{\overline{z/x}\}^{\sigma}_{\Sigma} \\
\texttt{(}sexp_1\ \texttt{.}\ sexp_2\texttt{)}\{\overline{z/x}\}^{\texttt{(}\sigma_1\ \texttt{.}\ \sigma_2\texttt{)}}_{\Sigma} & = & \texttt{(}sexp_1\{\overline{z/x}\}^{\sigma_1}_{\Sigma}\ \texttt{.}\ sexp_2\{\overline{z/x}\}^{\sigma_2}_{\Sigma}\texttt{)} \\
sexp\{\overline{z/x}\}^{\mu A.\sigma}_{\Sigma} & = & sexp\{\overline{z/x}\}^{\sigma[\mu A.\sigma/A]}_{\mathcal{P}(\sigma[\mu A.\sigma/A],sexp)} \\
sexp\{\overline{z/x}\}^{\cup\{\overline{\sigma_i}\}}_{\Sigma} & = & sexp\{\overline{z/x}\}^{\sigma_i}_{\mathcal{P}(\sigma_i,sexp)} \\
 & \multicolumn{2}{l}{\text{for smallest } i \text{ s.t. } \mathcal{P}(\sigma_i, sexp)\Downarrow} \\
sexp\{\overline{z/x}\}^{\sigma}_{\Sigma} & = & sexp \\
 & \multicolumn{2}{l}{\text{otherwise}}
\end{array}$$

Figure 4.12: Type-directed α-renaming.

$$\begin{array}{lcll}
\{\overline{z/x}\}^{\varepsilon} & = & \iota & \\
\{\overline{z/x}\}^{a@\Gamma} & = & \{\overline{z/x}\}^{a}\{\overline{z/x}\}^{\Gamma} & \\
\{\overline{z/x}\}^{\mathrm{P}::\Gamma} & = & \{\overline{z/x}\}^{\mathrm{P}}\{\overline{z/x}\}^{\Gamma} & \\
\{\overline{z/x}\}^{\{\overline{\mathrm{V}_i}\}} & = & \overline{\{\overline{z/x}\}^{\mathrm{V}_i}} & \\
\{\overline{z/x}\}^{z_i{:}\_} & = & \{z_i/x_i\} & \\
\{\overline{z/x}\}^{var{:}\_} & = & \iota & var \notin \{\overline{z}\}
\end{array}$$

Figure 4.13: Bindings-directed α-renaming.

dress accordingly. As usual, recursive and union types select out their nested sub-table by the current tree address and recur. In other cases (for example, the types expr, data, and ()), α-conversion leaves the term unchanged.

4.3.3.2 Bindings-directed alpha-renaming

Bindings-directed renaming performs a set of variable renamings at syntax nodes where the current environment is extended with new bindings. The operation is defined by a set of substitutions $\{\overline{z/x}\}$ and an environment fragment defining the new bindings. The definition is given in Figure 4.13. With an empty environment fragment ε, the operation is the identity ι. At compound environment fragments $a@\Gamma$ or $\mathrm{P} :: \Gamma$, the operation is defined structurally, with renamings guided by the inner environment structure taking precedent over outer structure. Rib-directed renamings are composed of renamings directed by their component variables. Thanks to our initial assumptions, we know that each distinct variable x maps to a unique fresh variable z, so the order of these renamings is insignificant. Finally, at a single variable, if the variable happens to be one of the substituting variables z_i, the operation is the corresponding substitution $\{z_i/x_i\}$; otherwise the operation is the identity.

Since these definitions are rather technical, let us walk through a small example. Comparing the expression:

Example 28

(lambda $(x_1\ x_2\ x_3)\ x_2)$

to the expression:

Example 29

(lambda $(y_1\ y_2\ y_3)\ y_2)$

for α-equivalence requires unifying their bound variables. The [A-MACAPP] rule extracts bindings for the first expression:

$$\{\mathsf{AA} \mapsto x_1, \mathsf{ADA} \mapsto x_2, \mathsf{ADDA} \mapsto x_3\}$$

and for the second:

$$\{\mathsf{AA} \mapsto y_1, \mathsf{ADA} \mapsto y_2, \mathsf{ADDA} \mapsto y_3\}$$

Picking fresh bindings to unify the two, we obtain:

$$sexp_1 = \texttt{((}z_1\ z_2\ z_3\texttt{)}\ x_2\texttt{)}$$

and for the second expression:

$$sexp_2 = \texttt{((}z_1\ z_2\ z_3\texttt{)}\ y_2\texttt{)}$$

Note how these intermediate syntax trees have fresh binding occurrences but the references still use the original names. Since binding tables are oblivious to variable references, both of these terms produce the same bindings table:

$$\begin{aligned}
& \mathcal{P}(\textbf{(}\mathit{formals}\ \mathsf{expr}{\downarrow}\mathsf{A}\,{:}\,\textsc{rib}\textbf{)}, \mathit{sexp}_1) \\
= \;& \mathcal{P}(\textbf{(}\mathit{formals}\ \mathsf{expr}{\downarrow}\mathsf{A}\,{:}\,\textsc{rib}\textbf{)}, \mathit{sexp}_2) \\
= \;& \mathsf{A} \mapsto \{z_1 : \mathsf{expr}, z_2 : \mathsf{expr}, z_3 : \mathsf{expr}\}
\end{aligned}$$

Let Σ be this bindings table. Now we perform a type-directed α-conversion on sexp_1:

$$\begin{aligned}
& \textbf{((}z_1\ z_2\ z_3\textbf{)}\ x_2\textbf{)}\{\overline{z_i/x_i}\}_{\Sigma}^{\textbf{(}\mathit{actuals}\ \mathsf{expr}\downarrow\mathsf{A}::\varepsilon\textbf{)}} \\
= \;& \textbf{((}z_1\ z_2\ z_3\textbf{)}\{\overline{z_i/x_i}\}_{\Sigma}^{\mathit{actuals}}\ (x_2\{\overline{z_i/x_i}\}_{\Sigma}^{\mathsf{expr}\downarrow\mathsf{A}::\varepsilon})\textbf{)} \\
= \;& \textbf{((}z_1\ z_2\ z_3\textbf{)}\ (x_2\{\overline{z_i/x_i}\}_{\Sigma}^{\mathsf{expr}\downarrow\mathsf{A}::\varepsilon})\textbf{)} \\
= \;& \textbf{((}z_1\ z_2\ z_3\textbf{)}\ (x_2\{\overline{z_i/x_i}\}_{\Sigma}^{\mathsf{expr}}\{\overline{z_i/x_i}\}_{\Sigma}^{\mathsf{A}::\varepsilon})\textbf{)} \\
= \;& \textbf{((}z_1\ z_2\ z_3\textbf{)}\ (x_2\{\overline{z_i/x_i}\}_{\Sigma}^{\mathit{resolve}(\Sigma,\mathsf{A}::\varepsilon)})\textbf{)} \\
= \;& \textbf{((}z_1\ z_2\ z_3\textbf{)}\ (x_2\{\overline{z_i/x_i}\}_{\Sigma}^{\{\overline{z_i:\mathsf{expr}}\}::\varepsilon})\textbf{)} \\
= \;& \textbf{((}z_1\ z_2\ z_3\textbf{)}\ (x_2\{\overline{z_i/x_i}\})\textbf{)} \\
= \;& \textbf{((}z_1\ z_2\ z_3\textbf{)}\ z_2\textbf{)}
\end{aligned}$$

A similar process produces the same result for sexp_2.

4.4 Hygienic macro expansion

It is folklore that hygienic macros respect α-equivalence, yet without a way to describe the scope of programs other than expanding all macros, no definition of α-equivalence has ever been available to make this notion precise. Armed with a formal and precise definition, then, we can define a semantics for hygienic macro expansion and prove formally the guarantees it provides.

4.4.1 Expansion contexts

To begin with, we define a set of *expansion contexts*, in which expansion may occur. These include *form contexts* F and *S-expression contexts* S:

$$
\begin{array}{rcl}
F & ::= & [\,] \\
 & | & \texttt{(}\mathsf{letrec\text{-}syntax}\ \texttt{((}x\ m\texttt{))}\ F\texttt{)} \\
 & | & \texttt{(}mexp\ \texttt{.}\ S\,\texttt{:}\,\sigma\texttt{)} \\
 & & \\
S & ::= & F \mid \texttt{(}S\ \texttt{.}\ sexp\texttt{)} \mid \texttt{(}sexp\ \texttt{.}\ S\texttt{)}
\end{array}
$$

Note that we do not allow expansion to occur on the right-hand side of the clauses of a macro m. While we might expect this to be a reasonable position to "substitute equals for equals," it introduces subtleties with free pattern variables that complicate the proofs in the next chapter. (In real macro expanders, expansion does not occur in such contexts anyway, so this restriction does not ignore any behavior from actual practice.[2]) However, this notion of contexts does allow for expansion to occur within sub-expressions that real expanders cannot discover. For example, in the program fragment:

Example 30

```
(let ([x (or (f 1) (g #f))])
  (lambda (y) x))
```

the inner **or** expression cannot be macro-expanded without first expanding the **let** expression, since Scheme expanders have no way of predicting the behavior of the **let** macro. With an explicitly typed **let** macro, however, we can parse its arguments and discover the inner expansion positions within its subterms, all without expanding.

4.4.2 Hygienic expansion semantics

Figure 4.14 presents the rules for a small-step substitution semantics of hygienic expansion. Rule subst interprets macro definitions by substituting

[2] An analogy from the call-by-value λ-calculus is β-reduction under a binder.

$$\texttt{(letrec-syntax ((}x\ m\texttt{))}\ \mathit{form}\texttt{)} \longmapsto_{\mathsf{subst}} \texttt{(letrec-syntax ((}x\ m\texttt{))}\ \mathit{form}[m/x]\texttt{)}$$
where $x \in \mathit{fv}(\mathit{form})$
and $\mathit{bv}(\mathit{form}) \cap \mathit{fv}(m) = \emptyset$

$$\texttt{(letrec-syntax ((}x\ m\texttt{))}\ \mathit{form}\texttt{)} \longmapsto_{\mathsf{return}} \mathit{form}$$
where $x \notin \mathit{fv}(\mathit{form})$

$$\texttt{((syntax-rules}\ \tau\ \texttt{(}\overline{\texttt{(}p_i\,\texttt{:}\,\hat{\sigma}_i\ \mathit{form}_i\texttt{)}}\texttt{))}\ \texttt{.}\ \mathit{sexp}\,\texttt{:}\,\sigma\texttt{)} \longmapsto_{\mathsf{trans}} \mu(\mathit{form}_i)$$
where $\mu = \mathit{match}(p_i, \mathit{sexp})$ for smallest i s.t. $\mathit{match}(p_i, \mathit{sexp}) \Downarrow$
and $\mathit{bv}(\mathit{sexp})^{\hat{\sigma}_i}_{\mathcal{P}(\hat{\sigma}_i, \mathit{sexp})} \cap \mathit{fv}(\mathit{form}_i) = \emptyset$
and $\mathit{bv}(\mathit{form}_i) \# \mathit{sexp}$

Figure 4.14: The expansion semantics of λ_m.

them in for their bound variable. As a simple way of modeling the recursive binding structure, we leave the letrec-syntax definitions in place, and only remove them if and when no free references remain in the body form, using rule return. Finally, rule trans performs a macro transcription step: the first pattern p_i that matches sexp produces a substitution μ, which is used to instantiate the right-hand side of the clause form_i.

4.4.2.1 Substitution and transcription

The substitution relation $\longmapsto_{\mathsf{subst}}$ uses a scope-respecting macro substitution operation $\mathit{form}[m/x]$, which is defined in Figure 4.15. The definition might appear daunting, but in fact it follows the same structure as other scope-respecting operations such as fv and bv. As usual, at macro applications, the operation parses the macro argument using the annotated type and proceeds with the syntax type σ and binding table Σ to guide the operation through the S-expression tree. Note that at import types, if the variable x being substituted is shadowed, then no substitution occurs. However, whenever substitution reaches a free reference to x, it replaces x with the macro m.

Macro transcription is far simpler, since it does not require any understanding of program structure. The pattern matching operation match, de-

$$
\begin{array}{lcl}
a[m/x] & = & a \\
x[m/x] & = & m \\
y[m/x] & = & y \\
\quad \text{if } x \neq y & & \\
\multicolumn{3}{l}{(\mathsf{letrec\text{-}syntax}\ ((x\ m'))\ \mathit{form})[m/x]} \\
\multicolumn{3}{l}{\qquad = (\mathsf{letrec\text{-}syntax}\ ((x\ m'))\ \mathit{form})} \\
\multicolumn{3}{l}{(\mathsf{letrec\text{-}syntax}\ ((x'\ m'))\ \mathit{form})[m/x]} \\
\multicolumn{3}{l}{\qquad = (\mathsf{letrec\text{-}syntax}\ ((x'\ m'[m/x]))\ \mathit{form}[m/x])} \\
\quad \text{if } x \neq x' & & \\
(\mathit{mexp}\ .\ \mathit{sexp} : \sigma)[m/x] & = & (\mathit{mexp}[m/x]\ .\ \mathit{sexp}[m/x]^{\sigma}_{\Sigma} : \sigma) \\
\quad \text{where } \Sigma = \mathcal{P}(\sigma, \mathit{sexp}) & &
\end{array}
$$

$$
\begin{array}{lcl}
a[m/x]^{\sigma}_{\Sigma} & = & a \\
\mathit{var}[m/x]^{\mathsf{bvar}}_{\Sigma} & = & \mathit{var} \\
\mathit{form}[m/x]^{\mathsf{expr}}_{\Sigma} & = & \mathit{form}[m/x] \\
\mathit{mexp}[m/x]^{\sigma \rightarrow \mathsf{expr}}_{\Sigma} & = & \mathit{mexp}[m/x] \\
()[m/x]^{()}_{\Sigma} & = & () \\
(\mathit{sexp}_1\ .\ \mathit{sexp}_2)[m/x]^{(\sigma_1\ .\ \sigma_2)}_{\Sigma} & = & (\mathit{sexp}_1[m/x]^{\sigma_1}_{\Sigma}\ .\ \mathit{sexp}_2[m/x]^{\sigma_2}_{\Sigma}) \\
\mathit{sexp}[m/x]^{\sigma\downarrow\beta}_{\Sigma} & = & \mathit{sexp} \\
\quad \text{if } x \in \mathit{dom}(\mathit{resolve}(\Sigma, \beta)) & & \\
\mathit{sexp}[m/x]^{\sigma\downarrow\beta}_{\Sigma} & = & \mathit{sexp}[m/x]^{\sigma}_{\Sigma} \\
\quad \text{if } x \notin \mathit{dom}(\mathit{resolve}(\Sigma, \beta)) & & \\
\mathit{sexp}[m/x]^{\sigma\uparrow\beta}_{\Sigma} & = & \mathit{sexp}[m/x]^{\sigma}_{\Sigma} \\
\mathit{sexp}[m/x]^{\mu A.\sigma}_{\Sigma} & = & \mathit{sexp}[m/x]^{\sigma[\mu A.\sigma/A]}_{\mathcal{P}(\sigma[\mu A.\sigma/A],\mathit{sexp})} \\
\mathit{sexp}[m/x]^{\cup\{\overline{\sigma_i}\}}_{\Sigma;\ell} & = & \mathit{sexp}[m/x]^{\sigma_i}_{\mathcal{P}(\sigma_i,\mathit{sexp})} \\
\quad \text{for smallest } i \text{ s.t. } \mathcal{P}(\sigma_i, \mathit{sexp}) \Downarrow & &
\end{array}
$$

Figure 4.15: Macro definition substitution.

fined in Figure 4.16, maps pattern variables to nodes in sexp by address, forming a substitution μ. Applying the substitution to the template form on the right-hand side of the macro clause is a simple matter of searching the form for occurrences of pattern variables and replacing them with their associated S-expression nodes. Application of μ is lifted to apply to all S-expressions; the definition is given in the same figure.

$$match(p, sexp) \overset{\text{def}}{=} \{a \mapsto sexp.\ell \mid a = p.\ell\}$$

$$\begin{array}{lcl}
\mu(x) & = & x \\
\mu(\texttt{(}mexp\ \texttt{.}\ sexp\,\texttt{:}\,\sigma\texttt{)}) & = & \texttt{(}\mu(mexp)\ \texttt{.}\ \mu(sexp)\,\texttt{:}\,\sigma\texttt{)} \\
\mu(prim) & = & prim \\
\mu(m) & = & m \\
\mu(sym) & = & sym \\
\mu(\texttt{()}) & = & \texttt{()} \\
\mu(\texttt{(}sexp_1\ \texttt{.}\ sexp_2\texttt{)}) & = & \texttt{(}\mu(sexp_1)\ \texttt{.}\ \mu(sexp_2)\texttt{)}
\end{array}$$

Figure 4.16: Macro transcription.

4.4.2.2 Hygiene side conditions

Both the substitution rule $\longmapsto_{\mathsf{subst}}$ and transcription rule $\longmapsto_{\mathsf{trans}}$ have side conditions that we have not yet addressed. The substitution rule's side condition mandates that free variables occurring in the macro being substituted cannot conflict with bound variables in the body expression. This ensures that references in a macro retain their meaning when the macro is placed in a new context. This notion is sometimes (rather confusingly) referred to in macro literature as *referential transparency*. It is also reminiscent of capture-avoiding substitution in the λ-calculus.

The side condition on transcription ensures that bound variables occurring the template of a macro clause are "private" to the macro by disallowing them to conflict with free variables in the macro argument. This makes it impossible for a macro definition to capture references in the macro application site.

Taken together, these rules constitute the *hygiene conditions* for macro expansion.

4.4.2.3 Hygienic expansion relation

Finally, we introduce an explicit notion of an α-renaming step:

$$sexp \longmapsto_{\alpha} sexp' \overset{\text{def}}{=} sexp =_{\alpha} sexp'$$

This rule allows an expander to choose an α-equivalent program term nondeterministically.

Thus we arrive at the full definition of hygienic macro expansion:[3]

$$\begin{aligned} \longmapsto_{\varepsilon} &\overset{\text{def}}{=} \longmapsto_{\alpha}; S[\longmapsto_{\mathsf{subst}} \cup \longmapsto_{\mathsf{return}} \cup \longmapsto_{\mathsf{trans}}] \\ \longmapsto\!\!\!\!\twoheadrightarrow_{\varepsilon} &\overset{\text{def}}{=} \longmapsto_{\varepsilon}^{*} \end{aligned}$$

In words: a hygienic expansion sequence is any number of macro substitutions, returns, or transcription steps occurring within an expansion context, each preceded by an α-conversion to ensure the hygiene conditions are met. By preceding each step with an α-conversion, we lift the definition of expansion to α-equivalence classes of programs.

[3] We overload $S[\text{—}]$ to denote the compatible closure of a binary $sexp$ relation.

CHAPTER 5

Well-Formedness

The formal guarantees provided by explicit binding specifications can only be maintained if those specifications are respected by macro definitions and their clients. In this chapter we present the type system of λ_m, which ensures statically that expansion never violates the specifications of macros.

5.1 Type checking

Recall from Chapter 4 the definition of bindings and environments:

$$\begin{array}{lcl} \mathrm{B} & ::= & \mathrm{V} \mid \mathrm{P} \mid \Gamma \\ \Gamma & ::= & \varepsilon \mid \mathrm{P} :: \Gamma \mid a@\Gamma \\ \mathrm{P} & ::= & \{\overline{\mathrm{V}_i}\} \\ \mathrm{V} & ::= & x : \tau \mid a : \delta \end{array}$$

These environments provide the information needed for type checking to determine the syntactic types of bound variables. Type checking involves an additional *pattern environment* Π:

$$\Pi ::= \bullet \mid p \mathbin{\boldsymbol{:}} \hat{\sigma}$$

This environment tracks the pattern variables bound in the right-hand side of a macro definition clause. Given a pattern environment Π, we can determine the type of a pattern variable a, as shown in Figure 5.1.

$$
\begin{array}{lcll}
\Pi(a) & = & \mathit{ptype}(p, \hat{\sigma}, a) & \text{where } \Pi = p \mathbin{:} \hat{\sigma} \\
\\
\mathit{ptype}(a, \hat{\sigma}, a) & = & \hat{\sigma} & \\
\mathit{ptype}(p, \sigma{\uparrow}\beta, a) & = & \mathit{ptype}(p, \sigma, a) & \text{if } p \notin \mathbb{P} \\
\mathit{ptype}(\texttt{(}p_1 \;\texttt{.}\; p_2\texttt{)}, \texttt{(}\hat{\sigma}_1 \;\texttt{.}\; \hat{\sigma}_2\texttt{)}, a) & = & \mathit{ptype}(p_i, \hat{\sigma}_i, a) & \text{for smallest } i \in 1, 2
\end{array}
$$

Figure 5.1: Pattern environment lookup.

A well-formed pattern environment has a type for every pattern variable:

$$\mathit{wf}(p \mathbin{:} \hat{\sigma}) \overset{\text{def}}{=} \mathit{wf}(p) \wedge \forall a \in \mathit{dom}(p).\mathit{ptype}(p, \hat{\sigma}, a) \text{ is defined}$$

5.1.1 Two dimensions of scope

The two environments Γ and Π parallel the two dimensions of scope in Scheme. The traditional dimension corresponds to the scope of variables in the host program. For example, in the expression

(**lambda** (*foo*) $expr$) Example 31

the variable *foo* may be referred to within $expr$. The second dimension corresponds to the right-hand side of a macro definition, where the current pattern variables are in scope and stand in for syntax nodes in the macro argument. It is important to understand that pattern variables are scoped in *both* dimensions. For example, within a macro clause binding pattern variable a with type bvar, we can use a throughout the right-hand side of the clause because it is bound in Π:

(**lambda** (a) $expr$) Example 32

But within $expr$, we can also use a as a variable *reference*, because, having bound a with **lambda**, it now occurs in Γ.

5.1.2 Inverting the pattern environment

Well-formed patterns bind their variables linearly:

$$wf(p) \overset{\text{def}}{=} \forall \ell, \ell'.(p.\ell = p.\ell') \implies (\ell = \ell')$$

Consequently each pattern p implicitly defines a bijection between pattern variables and tree addresses. In particular, we can invert a pattern to determine the address of a pattern variable in the current macro pattern:

$$p^{-1}(a) = \ell \text{ s.t. } p.\ell = a$$

We overload this notation for arbitrary collections of bindings B in the natural way, as well as lifting the notation to pattern environments:

$$\Pi^{-1}(\mathrm{B}) \overset{\text{def}}{=} p^{-1}(\mathrm{B}) \text{ where } \Pi = p \mathbin{\boldsymbol{:}} \hat{\sigma}$$

Inversion is used in type-checking pattern variable references to ensure the current base environment binds the expected pattern variables appropriately. For example, macros can document that one pattern variable occurs in the scope of another:

$$(\mathsf{bvar}\ \mathsf{expr}{\downarrow}\{\mathsf{A}{:}\mathsf{VAR}\}) \to \mathsf{expr}$$

An implementation that places a pattern variable in the position of the expr expression but without binding the variable in the bvar position must be rejected:

Example 33

```
;; (bvar expr↓{A:VAR}) → expr
(define-syntax bad
  (syntax-rules ()
    [(bad a e)
     e]))
```

Otherwise, the types would lead to inconsistent conclusions about scope:

$$\begin{array}{l}(\textbf{lambda}\ (x)\\ \quad(\textbf{bad}\ x\ x))\end{array} \quad =_\alpha \quad \begin{array}{l}(\textbf{lambda}\ (x)\\ \quad(\textbf{bad}\ y\ y))\end{array} \quad \longmapsto^*_\varepsilon \quad (\textbf{lambda}\ (x)\ y)$$

$$\begin{array}{lcll}
\varepsilon|_{\mathbb{P}} & = & \varepsilon & \\
(\mathrm{P} :: \Gamma)|_{\mathbb{P}} & = & \Gamma|_{\mathbb{P}} & \text{if } \mathrm{P}|_{\mathbb{P}} = \emptyset \\
(\mathrm{P} :: \Gamma)|_{\mathbb{P}} & = & \mathrm{P}|_{\mathbb{P}} :: \Gamma|_{\mathbb{P}} & \text{if } \mathrm{P}|_{\mathbb{P}} \neq \emptyset \\
(a@\Gamma)|_{\mathbb{P}} & = & a@(\Gamma|_{\mathbb{P}}) & \\
\{\overline{x\!:\!\tau}, \overline{a\!:\!\delta}\}|_{\mathbb{P}} & = & \{\overline{a\!:\!\delta}\} &
\end{array}$$

Figure 5.2: Projecting pattern variables from the base environment.

Moreover, type checking must ensure that the entire structure of pattern variables bound in the base environment exactly matches the structure documented in the macro type. We do this by extracting the pattern variable portion of a base environment, notated $\Gamma|_{\mathbb{P}}$, and inverting the resulting environment to recover a binding signature that can be compared to the macro's documented signature.

Figure 5.2 presents the definition of $\mathrm{B}|_{\mathbb{P}}$. Most of the cases are structural. The one subtlety is the case where a rib contains no pattern variables; since an empty rib does not affect the documented binding structure of a macro, type checking is made more robust by ignoring it.

5.1.3 The base environment stack

The expansion semantics given in Chapter 4 uses substitution to specify macro definitions. In other words, the approach of our model is a small-step substitution semantics, where macros can only be applied when their definition occurs inline at the point of use:

$$\texttt{((}\mathsf{syntax\text{-}rules}\ \sigma \rightarrow \mathsf{expr}\ \cdots\texttt{)}\ \texttt{.}\ \mathit{sexp}\,\texttt{:}\,\sigma\texttt{)}$$

This approach has the consequence that macros may get substituted into the bodies of other macros. However, we do not allow nested macros to refer to pattern variables of outer macros, as this would not model any part of the actual behavior of Scheme macros. So each macro body is type-checked in a fresh pattern environment, to ensure that no such references are possible.

Changing the pattern environment is not sufficient to implement pattern inversion correctly: when we invert the base environment to determine which pattern variables in scope where, we are only interested on those pattern variables from the innermost macro. So instead of maintaining a single base environment, type checking uses a stack of base environments. Pattern inversion is performed on only the top of the base environment stack, and each time the type checker enters the body of a macro, it pushes a fresh base environment.

5.1.4 Checking form types

Figure 5.3 presents the rules for the form type checking judgment. Let us consider the rules in detail. Rule [F-MACDEF] checks a macro-definition form. Macro definitions in λ_m are not allowed to nest within macro bodies, so the rule only applies when the pattern environment is $\bullet$. The rule extends the base environment Γ by associating the macro name x with the type of the macro m. Since the definition may be recursive, both the macro m and the body expression $form$ are type-checked in the extended environment.

Rule [F-MACAPP] checks macro applications. The macro expression $mexp$ is checked to have an arrow type, and the annotated argument type σ is checked to be a subtype of its domain σ'. The argument expression $sexp$ is parsed with σ to produce a bindings table Σ, which is then used to type-check $sexp$ at σ.

The rule also requires Σ to be well-formed. Specifically, no rib should contain two pattern variables that might be expanded into duplicate variable bindings or overlapping sets of variable bindings. Figure 5.5 presents the judgment $\Pi \vdash \mathrm{B}$ **ok** for well-formed bindings and $\Pi \vdash \Sigma$ **ok** for well-formed binding tables. The latter judgment checks that all pattern variables occurring together in a single rib can be proven not to alias one another. This works by comparing the adjacency matrix of pattern variables that oc-

$$\boxed{\overline{\Gamma}; \Pi \vdash \mathit{form} : \mathsf{expr}}$$

[F-MacDef]

$$\frac{\begin{array}{c}\Gamma' = \{x : \tau\} :: \Gamma \\ \Gamma' :: \overline{\Gamma}; \bullet \vdash m : \tau_i \qquad \Gamma' :: \overline{\Gamma}; \bullet \vdash \mathit{form} : \mathsf{expr}\end{array}}{\Gamma :: \overline{\Gamma}; \bullet \vdash \texttt{(}\mathsf{letrec\text{-}syntax}\ \texttt{((}x\ m\texttt{))}\ \mathit{form}\texttt{)} : \mathsf{expr}}$$

[F-MacApp]

$$\frac{\begin{array}{c}\sigma : \delta \qquad \sigma <: \sigma' \qquad \Sigma = \mathcal{P}(\sigma, \mathit{sexp}) \qquad \Pi \vdash \Sigma\ \mathbf{ok} \\ \overline{\Gamma}; \Pi \vdash \mathit{mexp} : \sigma' \rightarrow \mathsf{expr} \qquad \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp} : \sigma\end{array}}{\overline{\Gamma}; \Pi \vdash \texttt{(}\mathit{mexp}\ \texttt{.}\ \mathit{sexp}\,\mathbf{:}\,\sigma\texttt{)} : \mathsf{expr}}$$

[F-Var]

$$\frac{\overline{\Gamma}(x) = \mathsf{expr}}{\overline{\Gamma}; \Pi \vdash x : \mathsf{expr}}$$

[F-PExpr]

$$\frac{\begin{array}{c}\Pi^{-1}(\Gamma|_{\mathbb{P}}) = \gamma \qquad a \notin \mathit{dom}(\overline{\Gamma}) \\ \Pi(a) <: \mathsf{expr}{\downarrow}\gamma\end{array}}{\Gamma :: \overline{\Gamma}; \Pi \vdash a : \mathsf{expr}}$$

[F-PBVar]

$$\frac{\begin{array}{c}\Pi^{-1}(\Gamma|_{\mathbb{P}}) = \rho :: \gamma \qquad \rho(\Pi^{-1}(a)) = \textsc{var} \\ \Pi(a) <: \mathsf{bvar}\end{array}}{\Gamma :: \overline{\Gamma}; \Pi \vdash a : \mathsf{expr}}$$

$$\boxed{\overline{\Gamma}; \Pi \vdash \mathit{mexp} : \sigma \rightarrow \mathsf{expr}}$$

[M-Var]

$$\frac{\overline{\Gamma}(x) = \sigma \rightarrow \mathsf{expr}}{\overline{\Gamma}; \Pi \vdash x : \sigma \rightarrow \mathsf{expr}}$$

[M-PMac]

$$\frac{\begin{array}{c}\Pi^{-1}(\Gamma|_{\mathbb{P}}) = \gamma \qquad a \notin \mathit{dom}(\overline{\Gamma}) \\ \Pi(a) \Downarrow \varepsilon = (\sigma \rightarrow \mathsf{expr}){\downarrow}\gamma\end{array}}{\Gamma :: \overline{\Gamma}; \Pi \vdash a : \sigma \rightarrow \mathsf{expr}}$$

[M-Macro]

$$\frac{\begin{array}{c}\sigma : \delta \qquad \cup\{\overline{\hat{\sigma}_i}\} : \delta \qquad \sigma <: \cup\{\overline{\hat{\sigma}_i}\} \\ \forall i. \mathit{wf}(p_i\,\mathbf{:}\,\hat{\sigma}_i) \wedge \mathit{wf}(p_i) \wedge \varepsilon :: \overline{\Gamma}; p_i\,\mathbf{:}\,\hat{\sigma}_i \vdash \mathit{form}_i : \mathsf{expr}\end{array}}{\overline{\Gamma}; \Pi \vdash \texttt{(}\mathsf{syntax\text{-}rules}\ \sigma \rightarrow \mathsf{expr}\ \texttt{(}\overline{\texttt{(}p_i\,\mathbf{:}\,\hat{\sigma}_i\ \mathit{form}_i\texttt{)}}\texttt{))} : \sigma \rightarrow \mathsf{expr}}$$

[M-Prim]

$$\frac{\mathit{prim} : \sigma \rightarrow \mathsf{expr}}{\overline{\Gamma}; \Pi \vdash \mathit{prim} : \sigma \rightarrow \mathsf{expr}}$$

Figure 5.3: The form and macro type checking judgments.

$$\boxed{\overline{\Gamma};\Pi;\Sigma \vdash \mathit{sexp} : \sigma}$$

[S-PVar]

$$\frac{\Pi^{-1}(\Gamma|_{\mathbb{P}}) = \gamma \qquad \Pi(a) <: \sigma{\downarrow}\gamma}{\Gamma :: \overline{\Gamma};\Pi;\Sigma \vdash a : \sigma}$$

[S-BVar]

$$\frac{}{\overline{\Gamma};\Pi;\Sigma \vdash x : \mathsf{bvar}}$$

[S-Data]

$$\frac{\mathit{data} \notin \mathbb{P} \qquad \forall \ell, a.(\mathit{data}.\ell = a) \Longrightarrow \Pi(a) <: \mathsf{data}}{\overline{\Gamma};\Pi;\Sigma \vdash \mathit{data} : \mathsf{data}}$$

[S-Expr]

$$\frac{\begin{array}{c}\mathit{form} \notin \mathbb{P}\\ \overline{\Gamma};\Pi \vdash \mathit{form} : \mathsf{expr}\end{array}}{\overline{\Gamma};\Pi;\Sigma \vdash \mathit{form} : \mathsf{expr}}$$

[S-MExp]

$$\frac{\begin{array}{c}\mathit{mexp} \notin \mathbb{P}\\ \overline{\Gamma};\Pi \vdash \mathit{mexp} : \sigma' \to \mathsf{expr} \qquad \sigma <: \sigma'\end{array}}{\overline{\Gamma};\Pi;\Sigma \vdash \mathit{mexp} : \sigma \to \mathsf{expr}}$$

[S-Import]

$$\frac{\begin{array}{c}\mathit{sexp} \notin \mathbb{P} \qquad \mathrm{B} = \mathit{resolve}(\Sigma, \beta) \qquad \Pi \vdash \mathrm{B}\ \mathbf{ok}\\ (\mathrm{B}, \Gamma) :: \overline{\Gamma};\Pi;\Sigma \vdash \mathit{sexp} : \sigma\end{array}}{\Gamma :: \overline{\Gamma};\Pi;\Sigma \vdash \mathit{sexp} : \sigma{\downarrow}\beta}$$

[S-Export]

$$\frac{\begin{array}{c}\mathit{sexp} \notin \mathbb{P}\\ \overline{\Gamma};\Pi;\Sigma \vdash \mathit{sexp} : \sigma\end{array}}{\overline{\Gamma};\Pi;\Sigma \vdash \mathit{sexp} : \sigma{\uparrow}\beta}$$

[S-Null]

$$\frac{}{\overline{\Gamma};\Pi;\Sigma \vdash \texttt{()} : \texttt{()}}$$

[S-Cons]

$$\frac{\overline{\Gamma};\Pi;\Sigma \vdash \mathit{sexp}_1 : \sigma_1 \qquad \overline{\Gamma};\Pi;\Sigma \vdash \mathit{sexp}_2 : \sigma_2}{\overline{\Gamma};\Pi;\Sigma \vdash \texttt{(}\mathit{sexp}_1\ \texttt{.}\ \mathit{sexp}_2\texttt{)} : \texttt{(}\sigma_1\ \texttt{.}\ \sigma_2\texttt{)}}$$

[S-Rec]

$$\frac{\begin{array}{c}\mathit{sexp} \notin \mathbb{P} \qquad \sigma' = \sigma[\mu A.\sigma/A]\\ \Sigma' = \mathcal{P}(\sigma', \mathit{sexp}) \qquad \Pi \vdash \Sigma'\ \mathbf{ok}\\ \overline{\Gamma};\Pi;\Sigma' \vdash \mathit{sexp} : \sigma'\end{array}}{\overline{\Gamma};\Pi;\Sigma \vdash \mathit{sexp} : \mu A.\sigma}$$

[S-Union]

$$\frac{\begin{array}{c}\mathit{sexp} \notin \mathbb{P}\\ \Sigma' = \mathcal{P}(\sigma_i, \mathit{sexp}) \qquad \Pi \vdash \Sigma'\ \mathbf{ok}\\ \overline{\Gamma};\Pi;\Sigma' \vdash \mathit{sexp} : \sigma_i\end{array}}{\overline{\Gamma};\Pi;\Sigma \vdash \mathit{sexp} : \cup\{\overline{\sigma_i}\}}$$

Figure 5.4: The syntax type checking judgment.

$$\frac{\mathrm{B} \cap \mathbb{P} = \emptyset}{\bullet \vdash \mathrm{B}\ \mathbf{ok}} \qquad \frac{\mathit{adj}(\mathrm{B}) \subseteq \{\{p.\ell, p.\ell'\} \mid \{\ell, \ell'\} \in \mathit{adj}(\hat{\sigma})\}}{p\,\mathbf{:}\,\hat{\sigma} \vdash \mathrm{B}\ \mathbf{ok}}$$

$$\frac{\forall \mathrm{B} \in \mathit{rng}(\Sigma).\Pi \vdash \mathrm{B}\ \mathbf{ok}}{\Pi \vdash \Sigma\ \mathbf{ok}}$$

Figure 5.5: Well-formed bindings.

cur together in a rib:

$$\mathit{adj}(\mathrm{B}) \stackrel{\text{def}}{=} \{\{a, b\} \mid \mathrm{P} \in \mathrm{B} \wedge a \neq b \in \mathit{dom}(\mathrm{P})\}$$

to the adjacency matrix implicitly provided by the macro's documented type:

$$\mathit{adj}(\sigma) \stackrel{\text{def}}{=} \{\{\ell, \ell'\} \mid \rho \in \sigma \wedge \ell \neq \ell' \in \mathit{dom}(\rho)\}$$

As long as a and b occur together in a rib in the macro rule's specified type $\hat{\sigma}$, the type system will prevent any use of the macro from instantiating a and b with aliases. This way we can safely place them together within a single rib in the macro implementation.

The next three rules specify type checking of variables. With rule [F-VAR], base variables are checked by simply looking them up in the base environment. Pattern variables are subtler, however. For pattern variables annotated with type expr, possibly in the scope of some additional environment structure, the rule [F-PEXPR] uses pattern inversion to ensure that the expected environment structure matches the actual environment structure in Γ. Finally, rule [F-PBVAR] checks a use of a bvar pattern variable as a reference. We discuss this rule in the next section.

5.1.5 The aliasing problem

The design of the λ_m type system led to the discovery of the following peculiarity of Scheme macros. Consider the following macro:

Example 34

```
;; (bvar bvar) → expr
(define-syntax K
  (syntax-rules ()
    [(K a b)
     (lambda (a)
       (lambda (b)
         a))]))
```

We might expect that any application of **K** would produce an expression equivalent to (**lambda** (*x*) (**lambda** (*y*) *x*)). Not so: the expression (**K** *x x*) uses the same variable for both *a* and *b* and expands to:

```
(lambda (x)                  (lambda (x)
  (lambda (x) x))     ≠α       (lambda (y) x))
```

This means that the binding structure of **K** is dependent on the actual choice of names given to its arguments. This dependency defeats the ability to determine binding structure statically.

This *aliasing problem* is addressed in the type rule [F-PBVar], which imposes a restriction to prevent such ambiguities. A pattern variable a of type bvar may be used as a variable reference, but only if it meets the following conditions:

1. a is in scope in the base environment; and
2. there are no other pattern variables in scope in the base environment that might shadow a.

This is called the *shadow restriction*, and is specified by requiring the addresses of a to appear in the first rib of the inverted environment. While there may be other pattern variables in the same rib, the syntax type system described below ensures that none of those variables may alias a.

The shadow restriction prohibits macros like **K** from being written. If this restriction seems draconian, consider that **K** can easily be rewritten:

Example 35

```
;; (bvar bvar) → expr
(define-syntax K′
  (syntax-rules ()
    [(K′ a b)
     (lambda (a)
       (let ([tmp a])
         (lambda (b) tmp)))]))
```

Note that in Scheme, this macro always exhibits the intended behavior, in that both (**K′** *x y*) and even (**K′** *x x*) expand into an expression equivalent to (**lambda** (*x*) (**lambda** (*y*) *x*)).

5.1.6 Checking macro types

Next we examine the rules for checking the types of macro expressions *mexp*, also in Figure 5.3. Rules [M-Var] and [M-PMac] parallel rules [F-Var] and [F-PExpr] for checking variable references to macros.[1] Rule [M-Macro] type-checks user-defined macros. To ensure completeness of pattern matching, we require the annotated type σ to be a subtype of the union of all pattern types; we could relax this restriction at the cost of expansion-time match errors. We could also eliminate unused patterns by requiring the union to be a subtype of the annotated type σ. We then check each rule in the macro by using the pattern and its type as the current pattern environment and pushing a new, empty base environment onto the stack. Finally, rule [M-Prim] returns the fixed type of a primitive.

5.1.7 Checking syntax types

Figure 5.4 presents the rules for checking an S-expression against a syntax type σ. Unlike forms and macro expressions, an S-expression may have any number of different syntax types; in other words, the rules are defined by type-directed induction. In an implementation, checking forms and macro

[1] For the equality comparison in rule [M-PMac], we simplify degenerate forms in the type such as unused recursive type variables and union types with a single element.

expressions produces a type as an output; syntax type checking instead takes the expected type σ as an input. The syntax type checking judgment uses an additional context argument: the bindings table Σ, which was obtained by parsing the argument S-expression with the annotated type σ in rule [F-MacApp].

Rule [S-PVar] is analogous to rules [F-PExpr] and [M-PMac] and checks a pattern variable by ensuring its specified imports are in the base environment. Rule [S-BVar] allows any base variable to be a binding occurrence. Rule [S-Data] checks purely symbolic data (as in the argument to **quote**) by ensuring that any nested pattern variables have type data. Rules [S-Expr] and [S-MExp] respectively indicate form and macro expression positions, and delegate to their respective type judgments.

The next two rules deal with binding signature types. Rule [S-Import] checks an import type by extending the base environment with the new bindings and recurring. Even though Σ comprises only well-formed environment fragments, we must check that the environment extension B is well-formed, since resolving β may form new ribs with aliased bindings. Rule [S-Export] does not require additional checks, since the table Σ already incorporates β in its structure as a result of parsing.

The remaining rules are essentially structural. Both rule [S-Null] and rule [S-Cons] use the syntax type to parse S-expression structure. The last two rules unfold the next addressing region in the syntax type and continue parsing the S-expression before recurring. Again, parsing forms a new binding table Σ', which must be well-formed.

5.2 Well-formed types

Several points in the type-checking judgments rely on types themselves being well-formed. Indeed, because syntax types and binding signatures constitute a little programming language in their own right (with parsing as their operational semantics), we specify a separate, meta-level type system for ensuring the well-formedness of syntax types.

5.2.1 Well-formed syntax types

The well-formedness rules for syntax types mimic the structure of parsing, as defined in Chapter 4. In particular, evaluating syntax types is a two-phase process:

1. parse the syntax node with the syntax type, producing a table of bindings Σ; and

2. use Σ to resolve binding signatures.

The definition of well-formedness, provided in Figure 5.6, is similarly separated into two phases. The first judgment is *export well-formedness*:

$$\ell \vdash_{\uparrow} \sigma : \Upsilon$$

This judgment provides an abstract table Υ mapping syntax nodes by address to binding types δ. This judgment corresponds to the parsing phase and the construction of a bindings table Σ. The second judgment, *import well-formedness*, ensures that all import types contain well-formed binding signatures according to the abstract table Υ constructed by the first judgment:

$$\Upsilon \vdash_{\downarrow} \sigma \ \mathbf{ok}$$

The well-formedness judgment also parallels the interleaving of the parsing process. Each time parsing crosses the boundary of an addressing region, the evaluation recurs. Similarly, the well-formedness judgment recurs at addressing region boundaries.

$$\boxed{\sigma : \delta}$$

$$\frac{\epsilon \vdash_{\uparrow} \sigma : \Upsilon \qquad \Upsilon(\epsilon) = \delta \qquad \Upsilon \vdash_{\downarrow} \sigma\ \mathbf{ok}}{\sigma : \delta}$$

$$\boxed{\ell \vdash_{\uparrow} \sigma : \Upsilon}$$

$$\frac{}{\ell \vdash_{\uparrow} \mathsf{expr} : \{\ell \mapsto \textsc{none}\}} \qquad \frac{}{\ell \vdash_{\uparrow} \mathsf{data} : \{\ell \mapsto \textsc{none}\}}$$

$$\frac{\sigma : \delta}{\ell \vdash_{\uparrow} \sigma \to \mathsf{expr} : \{\ell \mapsto \textsc{none}\}} \qquad \frac{}{\ell \vdash_{\uparrow} \mathsf{bvar} : \{\ell \mapsto \textsc{var}\}}$$

$$\frac{\begin{array}{c}\ell \notin dom(\Upsilon) \vee \Upsilon(\ell) = \textsc{none} \\ \ell \vdash_{\uparrow} \sigma : \Upsilon \qquad \Upsilon|_{\{\ell\}(\prec)} \vdash \beta : \delta\end{array}}{\ell \vdash_{\uparrow} \sigma{\uparrow}\beta : \Upsilon[\ell \mapsto \delta]} \qquad \frac{\ell \vdash_{\uparrow} \sigma : \Upsilon}{\ell \vdash_{\uparrow} \sigma{\downarrow}\beta : \Upsilon}$$

$$\frac{}{\ell \vdash_{\uparrow} \mathbf{()} : \{\ell \mapsto \textsc{none}\}} \qquad \frac{\ell\mathsf{A} \vdash_{\uparrow} \sigma_1 : \Upsilon_1 \qquad \ell\mathsf{D} \vdash_{\uparrow} \sigma_2 : \Upsilon_2}{\ell \vdash_{\uparrow} \mathbf{(}\sigma_1\ \mathbf{.}\ \sigma_2\mathbf{)} : (\Upsilon_1 \cup \Upsilon_2)[\ell \mapsto \textsc{none}]}$$

$$\frac{A : \delta \qquad \sigma : \delta}{\ell \vdash_{\uparrow} \mu A.\sigma : \{\ell \mapsto \delta\}} \qquad \frac{A : \delta}{\ell \vdash_{\uparrow} A : \{\ell \mapsto \delta\}}$$

$$\frac{\forall i \neq j.\sigma_i \not\bowtie \sigma_j \qquad \forall i.\sigma_i : \delta}{\ell \vdash_{\uparrow} \cup\{\overline{\sigma_i}\} : \{\ell \mapsto \delta\}}$$

$$\boxed{\Upsilon \vdash_{\downarrow} \sigma\ \mathbf{ok}}$$

$$\frac{}{\Upsilon \vdash_{\downarrow} \mathsf{expr}\ \mathbf{ok}} \qquad \frac{}{\Upsilon \vdash_{\downarrow} \mathsf{data}\ \mathbf{ok}} \qquad \frac{}{\Upsilon \vdash_{\downarrow} \sigma \to \mathsf{expr}\ \mathbf{ok}} \qquad \frac{}{\Upsilon \vdash_{\downarrow} \mathsf{bvar}\ \mathbf{ok}}$$

$$\frac{\Upsilon \vdash_{\downarrow} \sigma\ \mathbf{ok}}{\Upsilon \vdash_{\downarrow} \sigma{\uparrow}\beta\ \mathbf{ok}} \qquad \frac{\Upsilon \vdash \beta : \textsc{env} \qquad \Upsilon \vdash_{\downarrow} \sigma\ \mathbf{ok}}{\Upsilon \vdash_{\downarrow} \sigma{\downarrow}\beta\ \mathbf{ok}}$$

$$\frac{}{\Upsilon \vdash_{\downarrow} \mathbf{()}\ \mathbf{ok}} \qquad \frac{\Upsilon \vdash_{\downarrow} \sigma_1\ \mathbf{ok} \qquad \Upsilon \vdash_{\downarrow} \sigma_2\ \mathbf{ok}}{\Upsilon \vdash_{\downarrow} \mathbf{(}\sigma_1\ \mathbf{.}\ \sigma_2\mathbf{)}\ \mathbf{ok}}$$

$$\frac{}{\Upsilon \vdash_{\downarrow} \mu A.\sigma\ \mathbf{ok}} \qquad \frac{}{\Upsilon \vdash_{\downarrow} A\ \mathbf{ok}} \qquad \frac{}{\Upsilon \vdash_{\downarrow} \cup\{\overline{\sigma_i}\}\ \mathbf{ok}}$$

Figure 5.6: Well-formed types.

$$\boxed{\Upsilon \vdash \beta : \delta}$$

$$\frac{\forall i.\Upsilon(\ell_i) = \text{VAR} \qquad \forall j.\Upsilon(\ell_j) = \text{RIB}}{\Upsilon \vdash \{\overline{\ell_i : \text{VAR}}, \overline{\ell_j : \text{RIB}}\} : \text{RIB}} \qquad \frac{}{\Upsilon \vdash \varepsilon : \text{ENV}}$$

$$\frac{\Upsilon \vdash \rho : \text{RIB} \qquad \Upsilon \vdash \gamma : \text{ENV}}{\Upsilon \vdash \rho :: \gamma : \text{ENV}} \qquad \frac{\Upsilon(\ell) = \text{ENV} \qquad \Upsilon \vdash \gamma : \text{ENV}}{\Upsilon \vdash \ell @ \gamma : \text{ENV}}$$

Figure 5.7: Well-formed signatures.

5.2.2 Well-formed signatures

The rules for well-formed signatures are given in Figure 5.7. These fairly straightforward rules simply ensure that references to syntax nodes respect their binding types found in the abstract bindings table Υ.

5.2.3 Shapes

Well-formed union types are disallowed from having syntactically overlapping disjuncts. To specify this property, it is useful to describe an abstraction of syntax types. We define syntactic *shapes*:

$$\square, \triangle ::= \top \mid \texttt{()} \mid \texttt{(}\square\ \texttt{.}\ \triangle\texttt{)} \mid \mu A.\square \mid A \mid \cup\{\overline{\square}\}$$

Shapes eliminate form types, macro types, and binding specifications, and focus instead on just the tree structure of a syntax type. We restrict shapes to be contractive in the analogous fashion to syntax types. Given a syntax type σ, we can compute its shape inductively, as shown in Figure 5.8.

These definitions allow us to define the shape-overlap relation, $\sigma \bowtie \sigma'$, shown in Figure 5.9. The shape $\top$ overlaps with all shapes. The null shape `()` overlaps with itself. Pair shapes overlap structurally. A union shape overlaps with another shape if any of its disjuncts overlaps with the other. A recursive shape overlaps with another shape if its unfolding overlaps.

$$
\begin{array}{lcl}
\mathit{shape}(\sigma{\uparrow}\beta) & = & \mathit{shape}(\sigma) \\
\mathit{shape}(\sigma{\downarrow}\beta) & = & \mathit{shape}(\sigma) \\
\mathit{shape}(\mu A.\sigma) & = & \mu A.\mathit{shape}(\sigma) \\
\mathit{shape}(A) & = & A \\
\mathit{shape}(\cup\{\overline{\sigma_i}\}) & = & \cup\{\overline{\mathit{shape}(\sigma_i)}\} \\
\mathit{shape}(\mathbf{()}) & = & \mathbf{()} \\
\mathit{shape}(\mathbf{(}\sigma_1 \;.\; \sigma_2\mathbf{)}) & = & \mathbf{(}\mathit{shape}(\sigma_1) \;.\; \mathit{shape}(\sigma_2)\mathbf{)} \\
\mathit{shape}(\mathsf{bvar}) & = & \top \\
\mathit{shape}(\mathsf{expr}) & = & \top \\
\mathit{shape}(\sigma \to \mathsf{expr}) & = & \top \\
\mathit{shape}(\mathsf{data}) & = & \top
\end{array}
$$

Figure 5.8: Computing the shape of a syntax type.

$$\boxed{\sigma \bowtie \sigma'}$$

$$\frac{\mathit{shape}(\sigma) \bowtie \mathit{shape}(\sigma')}{\sigma \bowtie \sigma'}$$

$$\boxed{\Box \bowtie \triangle}$$

$$\frac{}{\top \bowtie \Box} \qquad \frac{}{\Box \bowtie \top} \qquad \frac{}{\mathbf{()} \bowtie \mathbf{()}}$$

$$\frac{\Box_1 \bowtie \triangle_1}{\mathbf{(}\Box_1 \;.\; \Box_2\mathbf{)} \bowtie \mathbf{(}\triangle_1 \;.\; \triangle_2\mathbf{)}} \qquad \frac{\Box_2 \bowtie \triangle_2}{\mathbf{(}\Box_1 \;.\; \Box_2\mathbf{)} \bowtie \mathbf{(}\triangle_1 \;.\; \triangle_2\mathbf{)}}$$

$$\frac{\exists i.\Box_i \bowtie \triangle}{\cup\{\overline{\Box_i}\} \bowtie \triangle} \qquad \frac{\exists i.\Box \bowtie \Box_i}{\Box \bowtie \cup\{\overline{\Box_i}\}} \qquad \frac{\Box[\mu A.\Box/A] \bowtie \triangle}{\mu A.\Box \bowtie \triangle} \qquad \frac{\Box \bowtie \triangle[\mu A.\triangle/A]}{\Box \bowtie \mu A.\triangle}$$

Figure 5.9: Shape overlap.

Shapes bear a strong resemblance to the *shape types* of Culpepper and Felleisen [17]—unsuprisingly, as shape types formed the inspiration for this work! Indeed, syntax types σ can be seen as shape types with the addition of binding signatures in order to specify scope and binding.

CHAPTER 6

Properties of Typed Hygienic Macros

This chapter presents the key correctness properties of the λ_m model. Naturally, the language enjoys standard properties such as type soundness (for both parsing and expansion). More subtly, there are several properties that are typically left unstated for standard calculi, but which nevertheless become non-trivial in the presence of macros. These include type preservation under α-conversion, as well as the guaranteed α-convertibility of all programs—a prerequisite of progress for hygienic expansion. Finally, the key correctness criterion of the λ_m model—and one of the central contributions of this thesis—is a formal characterization of hygiene, construed as *preservation of α-equivalence* and proved as a corollary of confluence.

6.1 Soundness of parsing

Recall from Chapters 3 and 5 that binding signatures (and consequently syntax types) are classifed by "binding types," i.e., the types of bindings defined within and exported by a node in a parse tree:

$$\delta ::= \textsc{var} \mid \textsc{rib} \mid \textsc{env} \mid \textsc{none}$$

Parsing generates a table mapping syntax nodes to bindings, which are classified by binding types:

$$\frac{}{\bullet : \textsc{none}} \qquad \frac{}{\mathrm{V} : \textsc{var}} \qquad \frac{}{\mathrm{P} : \textsc{rib}} \qquad \frac{}{\Gamma : \textsc{env}}$$

$$\boxed{\Sigma : \Upsilon} \qquad \boxed{\Upsilon \vdash \mathrm{B} : \delta}$$

$$\frac{\forall \ell \in dom(\Upsilon).\Upsilon|_{\{\ell\}(\prec)} \vdash \Sigma(\ell) : \Upsilon(\ell)}{\Sigma : \Upsilon} \qquad \frac{\mathrm{B} : \delta}{\Upsilon \vdash \mathrm{B} : \delta}$$

Figure 6.1: Well-typed bindings.

Parsing and resolution satisfy the simple invariant that they respect the binding types predicted by a binding type environment Υ. A binding table is well-typed with respect to a binding type environment if it satisfies the property given in Figure 6.1. That is, well-typed binding tables map addresses to bindings of the appropriate type.

Theorem 6.1.1 (Resolution soundness). *Let* $\Sigma : \Upsilon$. *Then the following properties hold:*

1. *If* $\Upsilon \vdash attr : \delta$ *then* $\Upsilon \vdash resolve(\Sigma, attr) : \delta$.

2. *If* $\Upsilon(\ell) = \delta$ *then* $\Upsilon \vdash resolve(\Sigma, \ell) : \delta$

Proof. By induction on the definition of $resolve$. □

Corollary. *If* $\Sigma : \Upsilon$ *then* $\{\overline{\ell \mapsto resolve(\Sigma, \Sigma(\ell))}\} : \Upsilon$.

The parsing process respects binding types, and so always produces well-typed bindings tables. The proof relies on the above theorem as well as the following lemma, which is required for unfolding recursive types:

Lemma 6.1.2 (Type substitution). *If* $\ell \vdash_{\uparrow} \sigma : \Upsilon$ *and* $A : \delta$ *and* $\sigma_0 : \delta$ *then* $\ell \vdash_{\uparrow} \sigma[\mu A.\sigma_0/A] : \Upsilon$.

Proof. By induction on σ. We consider the case where $\sigma = A$; the remaining cases are straightforward. We have $\sigma[\mu A.\sigma_0/A] = \mu A.\sigma_0$ and $\Upsilon = \{\ell \mapsto \delta\}$. By assumption $\sigma_0 : \delta$ so $\ell \vdash_{\uparrow} \mu A.\sigma_0 : \{\ell \mapsto \delta\}$. □

Theorem 6.1.3 (Parsing soundness)**.** *If* $\ell \vdash_{\uparrow} \sigma : \Upsilon$ *and* $\Sigma = parse(\sigma, sexp, \ell)$ *then* $\Sigma : \Upsilon$.

Proof. By induction on the definition of $parse$. □

6.2 Freshness

The next property we prove is one that is typically left unstated, but is nevertheless non-trivial in a system with macros: the guaranteed existence of fresh variables that can be used to α-convert a term. This property is crucial for the progress of hygienic macro expansion, because it means that expansion can never get stuck for lack of fresh variables.

The theorem relies on a rather technical lemma, which guarantees that a type-directed α-conversion which is defined when a term is *parseable,* i.e., can be successfully parsed at a given type σ.

Lemma 6.2.1. *Let* $sexp$ *be parseable at* σ*, where* $\forall i.\ell_i \in bp(\sigma)$ *and* $x_i = sexp.\ell_i$*. If* $\overline{z_i} \# sexp$ *and* $\sigma : \delta$ *then*

$$sexp[\overline{\ell_i \mapsto z_i}]\{\overline{z_i/x_i}\}^{\sigma}_{\mathcal{P}(\sigma, sexp[\overline{\ell_i \mapsto z_i}])}$$

is defined.

Proof. By nested inductions on the definition of parsing. The full proof is provided in Appendix A. □

Theorem 6.2.2. *If* $\overline{\Gamma}; \Pi \vdash sexp : \sigma$ *or* $\overline{\Gamma}; \Pi; \Sigma \vdash sexp : \sigma$ *then there exists an S-expression* $sexp' =_{\alpha} sexp$ *such that* $x \# sexp'$.

Proof. By induction on the type derivation. At each macro application, choose a fresh set of variable bindings $\overline{z_i}$ that do not occur in $supp(sexp)$ and α-convert. Lemma 6.2.1 ensures that the α-conversion succeeds. □

$$\frac{\forall \ell \in dom(\Upsilon).\Upsilon(\ell) = \textsc{none} \vee \Upsilon(\ell) = \Upsilon(\ell')}{\Upsilon \sqsubseteq \Upsilon'}$$

$$\frac{\epsilon \vdash_{\uparrow} \sigma_0 : \Upsilon_0 \qquad \ell \vdash_{\uparrow} \sigma : \Upsilon \qquad \Upsilon \sqsubseteq \Upsilon_0 \qquad \Upsilon \vdash_{\downarrow} \sigma \ \mathbf{ok}}{\ell \vdash \sigma : \Upsilon \sqsubseteq \sigma_0 : \Upsilon_0}$$

Figure 6.2: Generalized well-formedness.

6.3 Alpha-equivalence

The fact that α-conversion does not affect the type (or typability) of a program is rarely proved for typical languages. But given that α-conversion is an explicit part of hygienic expansion—namely, via the $\longmapsto_{\alpha}$ rule—and that scope is the essential component of type-checking in λ_m, we must take extra care to prove this property.

The theorem relies on a technical lemma to show that a type-directed α-conversion preserves the type of an S-expression. The statement of the lemma makes use of a generalization of the well-formedness relation on types, given in Figure 6.2. The judgment $\ell \vdash \sigma : \Upsilon \sqsubseteq \sigma_0 : \Upsilon_0$ generalizes the judgment $\sigma : \delta$ for inductive proofs by tracking the various well-formedness judgments on a sub-component σ of a syntax type σ_0. Note that for any well-formed type $\sigma : \delta$, there is by definition a binding-type environment Υ such that $\epsilon \vdash \sigma : \Upsilon \sqsubseteq \sigma : \Upsilon$.

Lemma 6.3.1 (Type-directed α-conversion)**.** *If the following properties hold:*

- $\{\overline{\ell_i \mapsto x_i}\} \subseteq bindings(\sigma, sexp)$
- $\Sigma = \mathcal{P}(\sigma_0, sexp)$
- $\Sigma' = \Sigma[\overline{\ell_i \mapsto z_i}]$ *where* $\overline{z_i}\#sexp$
- $\Sigma : \Upsilon_0$
- $\ell \vdash \sigma : \Upsilon \sqsubseteq \sigma_0 : \Upsilon_0$

then:

$$\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp}.\ell : \sigma \iff \overline{\Gamma}; \Pi; \Sigma' \vdash \mathit{sexp}[\overline{\ell_i \mapsto z_i}].\ell\{\overline{z_i/x_i}\}^{\sigma}_{\Sigma'} : \sigma$$

Proof. By induction on σ. The proof relies on two additional lemmas and is presented in Appendix A. □

Theorem 6.3.2. *Let $\mathit{sexp} =_\alpha \mathit{sexp}'$. Then the following properties hold:*

1. $\overline{\Gamma}; \Pi \vdash \mathit{sexp} : \tau \Longrightarrow \overline{\Gamma}; \Pi \vdash \mathit{sexp}' : \tau$
2. $\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp} : \sigma \Longrightarrow \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp}' : \sigma$

Proof. By induction on the type derivation. We consider the case of macro applications here. Let $\Sigma = \mathcal{P}(\sigma, \mathit{sexp})$ and $\Sigma' = \mathcal{P}(\sigma, \mathit{sexp}')$; by the definition of α-equivalence, we have $\Sigma'' = \Sigma[\overline{\ell_i \mapsto z_i}] = \Sigma'[\overline{\ell_i \mapsto z_i}]$.

$$\begin{array}{ll}
 & \overline{\Gamma}; \Pi \vdash \texttt{(}\mathit{mexp}\ \texttt{.}\ \mathit{sexp}\,\texttt{:}\,\sigma\texttt{)} : \textsf{expr} \\
\iff & \{\text{inversion of Rule [F-MacApp]}\} \\
 & \overline{\Gamma}; \Pi \vdash \mathit{mexp} : \sigma_0 \to \textsf{expr} \land \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp} : \sigma \\
\iff & \{\text{Lemma 6.3.1}\} \\
 & \overline{\Gamma}; \Pi \vdash \mathit{mexp} : \sigma_0 \to \textsf{expr} \land \overline{\Gamma}; \Pi; \Sigma'' \vdash \mathit{sexp}[\overline{\ell_i \mapsto z_i}]\{\overline{z_i/x_i}\}^{\sigma}_{\Sigma''} : \sigma \\
\iff & \{\text{induction hypothesis}\} \\
 & \overline{\Gamma}; \Pi \vdash \mathit{mexp}' : \sigma_0 \to \textsf{expr} \land \overline{\Gamma}; \Pi; \Sigma'' \vdash \mathit{sexp}'[\overline{\ell_i \mapsto z_i}]\{\overline{z_i/x_i'}\}^{\sigma}_{\Sigma''} : \sigma \\
\iff & \{\text{Lemma 6.3.1}\} \\
 & \overline{\Gamma}; \Pi \vdash \mathit{mexp}' : \sigma_0 \to \textsf{expr} \land \overline{\Gamma}; \Pi; \Sigma' \vdash \mathit{sexp}' : \sigma \\
\iff & \{\text{Rule [F-MacApp]}\} \\
 & \overline{\Gamma}; \Pi \vdash \texttt{(}\mathit{mexp}'\ \texttt{.}\ \mathit{sexp}'\,\texttt{:}\,\sigma\texttt{)} : \textsf{expr}
\end{array}$$

The remaining cases are straightforward. □

6.4 Subsumption

The type-checking rules for forms and macro expressions are invariant in their result type: they always produce expr for forms or the most specific arrow type for macro expressions. An S-expression may be typeable at any number of types, however.[1] A key property of syntax type checking is *sub-*

[1] Put differently, we can consider the type as an *input* to the syntax type checking judgment and an *output* of the other two judgments.

sumption: if type-checking succeeds for a type σ it succeeds for any supertype of σ.

Since subtyping relies on import normalization and region displacement, the proof of subsumption depends on the following two lemmas:

Lemma 6.4.1 (Import normalization)**.**

$$\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp} : \sigma{\downarrow}\beta \iff \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp} : \sigma \Downarrow \beta$$

Proof. By induction on the definition of $\sigma \Downarrow \beta$. □

Lemma 6.4.2 (Region displacement)**.**

$$\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp} : \sigma \iff \overline{\Gamma}; \Pi; \Sigma \ll \ell \vdash \mathit{sexp} : \sigma \ll \ell$$

Proof. By induction on the definition of $\sigma \ll \ell$. □

Lemma 6.4.3. *If* $\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp}.\ell : \hat{\sigma}$ *and* $\ell \vdash \hat{\sigma} <: \hat{\sigma}'$ *then* $\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp}.\ell : \hat{\sigma}'$.

Proof. By coinduction. We consider the case where σ is a recursive type here. Let $\Sigma' = \mathcal{P}(\sigma[\mu A.\sigma/A], \mathit{sexp}.\ell.\epsilon)$ and $\Sigma'' = \Sigma \cup (\Sigma' \ll \ell)$. Note that $\mathit{dom}(\Sigma) \cap \mathit{dom}(\Sigma' \ll \ell) = \emptyset$.

$$\begin{array}{ll}
 & \Gamma :: \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp}.\ell : (\mu A.\sigma){\downarrow}\gamma \\
\Longrightarrow & \{\text{inversion of Rule [}\textsc{S-Import}\text{]}\} \\
 & (\mathit{resolve}(\Sigma, \gamma), \Gamma) :: \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp}.\ell : \mu A.\sigma \\
\Longrightarrow & \{\text{inversion of Rule [}\textsc{S-Rec}\text{]}\} \\
 & (\mathit{resolve}(\Sigma, \gamma), \Gamma) :: \overline{\Gamma}; \Pi; \Sigma' \vdash \mathit{sexp}.\ell.\epsilon : \sigma[\mu A.\sigma/A] \\
\Longrightarrow & \{\text{Lemma 6.4.2}\} \\
 & (\mathit{resolve}(\Sigma, \gamma), \Gamma) :: \overline{\Gamma}; \Pi; \Sigma' \ll \ell \vdash \mathit{sexp}.\ell.\epsilon : \sigma[\mu A.\sigma/A] \ll \ell \\
\Longrightarrow & \{\mathit{dom}(\Sigma) \cap \mathit{dom}(\Sigma' \ll \ell) = \emptyset\} \\
 & (\mathit{resolve}(\Sigma'', \gamma), \Gamma) :: \overline{\Gamma}; \Pi; \Sigma'' \vdash \mathit{sexp}.\ell.\epsilon : \sigma[\mu A.\sigma/A] \ll \ell \\
\Longrightarrow & \{\text{Lemma 6.4.1}\} \\
 & \Gamma :: \overline{\Gamma}; \Pi; \Sigma'' \vdash \mathit{sexp}.\ell : (\sigma[\mu A.\sigma/A] \ll \ell) \Downarrow \gamma \\
\Longrightarrow & \{\text{coinduction hypothesis}\} \\
 & \Gamma :: \overline{\Gamma}; \Pi; \Sigma'' \vdash \mathit{sexp}.\ell : \hat{\sigma}' \\
\Longrightarrow & \{\mathit{dom}(\Sigma) \cap \mathit{dom}(\Sigma' \ll \ell) = \emptyset\} \\
 & \Gamma :: \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp}.\ell : \hat{\sigma}'
\end{array}$$

The case of union types is similar; the other cases are straightforward. □

Theorem 6.4.4 (Subsumption). *If* $\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp} : \sigma$ *and* $\sigma <: \sigma'$ *then* $\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp} : \sigma'$.

Proof. Follows immediately from Lemma 6.4.3. □

6.5 Type soundness

For the well-formedness invariants to be meaningful, they must accurately describe the behavior of macro expansion. Fortunately, the type system of λ_m is sound; the invariants of form and syntax types are maintained throughout expansion. In particular, the system we have described is specified tightly enough that *no* expansion-time errors occur. In practice, many expansion-time errors are reasonable to allow, such as pattern-matching failure or failure of computational effects in procedural macros (see Chapters 7 and 8).

We prove type soundness with the standard syntactic approach of Wright and Felleisen [72]. Many of the lemmas and theorems about well-typed programs involve two parallel propositions: one for forms and macro expressions, and one for S-expressions. For example, we may have a judgment $\overline{\Gamma}; \Pi \vdash \mathit{form} : \mathsf{expr}$ on a form, a judgment $\overline{\Gamma}; \Pi \vdash \mathit{mexp} : \sigma \rightarrow \mathsf{expr}$ on a macro expression, or a judgment $\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp} : \sigma$ on an S-expression. Similarly, we may sometimes refer to an overloaded operation such as $\mathit{fv}(\mathit{form})$ or $\mathit{fv}(\mathit{sexp})^{\sigma}_{\Sigma}$ that may or may not require a syntax type and bindings table. To keep the presentation concise without introducing additional generalizations, we use the simple shorthand $\overline{\Gamma}; \Pi \colorbox{lightgray}{$; \Sigma$} \vdash \mathit{sexp} : \sigma$, $\mathit{fv}(\mathit{sexp})\colorbox{lightgray}{$^{\sigma}_{\Sigma}$}$, etc. to describe all cases at once.

6.5.1 Type preservation

The proof that expansion preserves types relies on an auxiliary notion of merging type environment stacks, given in Figure 6.3. Merging two stacks $\overline{\Gamma} \odot \overline{\Gamma}'$ combines the two innermost environments, i.e., the rightmost environment of $\overline{\Gamma}$ and the leftmost environment of $\overline{\Gamma}'$.

$$\begin{array}{rcl}
\overline{\Gamma} \odot \varepsilon & = & \overline{\Gamma} \\
\varepsilon \odot \overline{\Gamma} & = & \overline{\Gamma} \\
(\overline{\Gamma}, \Gamma :: \varepsilon) \odot (\Gamma' :: \overline{\Gamma}') & = & \overline{\Gamma}, (\Gamma, \Gamma') :: \overline{\Gamma}'
\end{array}$$

Figure 6.3: Environment pasting.

6.5.1.1 Expansion contexts and decomposition

The first two lemmas are standard; they allow us to shift focus from a top-level program to the hole of an expansion context where an individual expansion rule applies.

Lemma 6.5.1 (Decomposition)**.** *Let $form \notin Variable$. If:*

$$\overline{\Gamma}_0; \bullet\, ; \Sigma \vdash S[form] : \sigma$$

then

$$\overline{\Gamma} \odot \overline{\Gamma}_0; \Pi \vdash form : \mathsf{expr}$$

for some $\overline{\Gamma}$.

Proof. By induction on the expansion context S. □

Lemma 6.5.2. *Let $form$, $form' \notin Variable$. If the following propositions hold:*

- $\overline{\Gamma}_0; \bullet\, ; \Sigma \vdash S[form] : \sigma$
- $\overline{\Gamma} \odot \overline{\Gamma}_0; \bullet \vdash form : \mathsf{expr}$
- $\overline{\Gamma} \odot \overline{\Gamma}_0; \bullet \vdash form' : \mathsf{expr}$

then $\overline{\Gamma}_0; \bullet\, ; \Sigma \vdash S[form'] : \sigma$.

Proof. By induction on the expansion context S. □

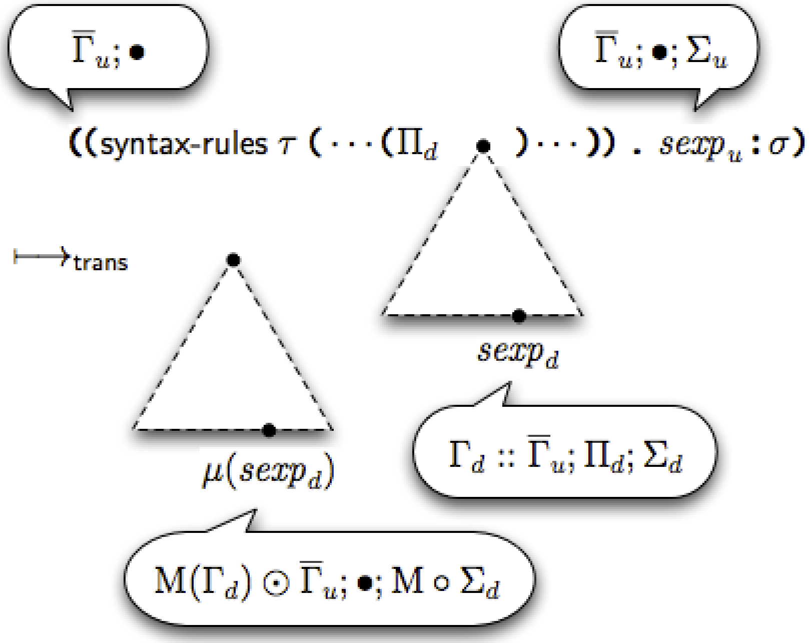

$\overline{\Gamma}_u; \bullet$
$\overline{\Gamma}_u; \bullet; \Sigma_u$
((syntax-rules τ ($\cdots$(Π_d $\bullet$)$\cdots$)) . $sexp_u$: σ)
$\longmapsto_{\mathsf{trans}}$
$sexp_d$
$\mu(sexp_d)$
$\Gamma_d :: \overline{\Gamma}_u; \Pi_d; \Sigma_d$
$\mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d$

Figure 6.4: Term and environment structure of macro transcription.

6.5.1.2 Macro substitution and the $\longmapsto_{\mathsf{subst}}$ rule

The rule $\longmapsto_{\mathsf{subst}}$ performs a macro substitution on the body of a letrec-syntax form. The following lemma ensures that the result of macro substitution is well-typed.

Lemma 6.5.3 (Macro substitution). *Let* $\overline{\Gamma} \odot \overline{\Gamma}_0; \Pi\,; \Sigma \vdash sexp : \sigma$ *such that* $(\overline{\Gamma} \odot \overline{\Gamma}_0)(x) = \sigma \to \mathsf{expr}$ *and* $\overline{\Gamma}_0; \bullet \vdash m : \sigma' \to \mathsf{expr} <: \sigma \to \mathsf{expr}$. *Then:*

$$\overline{\Gamma} \odot \overline{\Gamma}_0; \Pi\,; \Sigma \vdash sexp[m/x]_{\substack{\sigma \\ \Sigma}} : \sigma$$

Proof. By induction on the type derivation, using subsumption. □

6.5.1.3 Macro transcription and the $\longmapsto_{\mathsf{trans}}$ rule

Macro transcription is the heart of expansion and central to the soundness proof. Figure 6.4 illustrates a macro transcription step. The figure shows the application of a macro where pattern p is the first to match the argument

$$\overline{\Gamma}_u; \bullet; \Sigma_u \vdash \mu : \Pi_d \overset{\text{def}}{=} \forall a \in dom(\mu).\overline{\Gamma}_u; \bullet; \Sigma_u^{\mathcal{R}} \circ \Pi_d^{-1} \circ \Sigma_d \vdash \mu(a) : \Pi_d(a)$$

Figure 6.5: Well-typed pattern match.

$sexp_u$ ("u" for "use site"). The macro template is depicted as a tree containing a sub-tree $sexp_d$ ("d" for "definition site"). Pattern matching against $sexp_u$ forms a match μ, which is applied as a substitution on the definition tree, producing a result containing the sub-tree $\mu(sexp_d)$.

Notice the type environments associated with various terms in the figure. We see that the initial macro application term is typeable in the environments $\overline{\Gamma}_u; \bullet$. (Recall that we never expand inside of macro definitions, so any expansion site necessarily has an empty pattern environment). Thus the argument $sexp_u$ is similarly typeable in a context $\overline{\Gamma}_u; \bullet; \Sigma_u$. The macro template, moreover, is typeable in an extension of that environment: type checking pushes a frame onto the environment and replaces the use-site pattern environment with a new definition-site environment Π_d. Now, for the template tree to contain a nested S-expression $sexp_d$, it must occur as part of a macro application in the template. Within this context, we label the base environment Γ_d. Since the S-expression occurs as part of a macro application, there must also be a bindings table Σ_d.

Finally, after performing the transcription, the substituted template body replaces the entire application. Subtly, this means that the definition environment frame Γ_d—with pattern variables from p replaced by their binding structure in Σ_u—is *merged* with the first frame of $\overline{\Gamma}_u$ via the $\odot$ operator. The reason: environment frames correspond to entering the body of a macro, a boundary which the transcription step eliminates. The environments are transformed with a mapping $\mathrm{M} = \Sigma_u^{\mathcal{R}} \circ \Pi_d^{-1}$, which maps addresses from the macro pattern to their corresponding binding structure in the actual arguments.

Our first lemma ensures that the result of matching the pattern p against the use site argument $sexp_u$ produces an appropriate pattern match μ. For the subsequent transcription to be well-typed, μ must map pattern variables to appropriately typed fragments of $sexp_u$. Figure 6.5 defines a well-typed pattern match: each pattern variable a maps to an S-expression fragment that can be checked at the expected type of a in the environment of the transcribed term. Let use examine this environment carefully. The base environment is just that of the use site, $\overline{\Gamma}_u$, because any additional variables bound by the macro are expressed by a's syntax type (as import types). Again, the macro environment is $\bullet$ because our system disallows expansion inside of macro definitions. The bindings table is constructed by remapping any pattern variables in the definition-site Σ_d with their corresponding bindings in Σ_u (using the inversion of $penv_d$ to find each pattern variable's tree address in the pattern/use site argument).

Lemma 6.5.4 (Match)**.** *Let* $\Pi_d = p : \hat{\sigma}$. *If* $wf(\Pi_d)$ *and* $\overline{\Gamma}_u; \bullet; \Sigma_u \vdash sexp_u : \hat{\sigma}$ *then* $\mu = match(p, sexp)$ *exists and*

$$\overline{\Gamma}_u; \bullet; \Sigma_u \vdash \mu : \Pi_d$$

Proof. By induction on $ptype(p, \hat{\sigma}, a)$ and $sexp$ for each $a \in dom(p)$. □

Lemma 6.5.5. *Let* $\mathrm{M} = \Sigma_u^{\mathcal{R}} \circ \Pi_d^{-1}$. *If* $\overline{\Gamma}_u; \bullet; \Sigma_u \vdash \mu : \Pi_d$ *and* $\sigma : \delta$ *then*

$$\mathrm{M} \circ \mathcal{P}(\sigma, sexp) = \mathcal{P}(\sigma, \mu(sexp))$$

Proof. By induction on the parsing algorithm. □

The following lemma states that resolving a binding signature with a well-formed bindings table produces another well-formed bindings table. Specifically, resolution never encounters ribs with duplicate variables.

Lemma 6.5.6 (Unique names)**.** *Let* $\mathrm{M} = \Sigma_u^{\mathcal{R}} \circ \Pi_d^{-1}$. *If* $\Pi_d \vdash resolve(\Sigma_d, \beta)$ **ok** *and* $\bullet \vdash \Sigma_u$ **ok** *then* $\bullet \vdash resolve(\mathrm{M} \circ \Sigma_d, \beta)$ **ok**.

Proof. By induction on the resolution algorithm. □

The transcription lemma states that transcription results in a well-typed term, given a well-typed pattern substitution, a well-typed template, and several hygiene conditions. The conditions ensure that the template environment contains only pattern variables from the macro environment and fresh base variables, that free variables in the template are not captured by the substituted S-expressions, and that base variables bound in the template are fresh.

Lemma 6.5.7 (Transcription). *Let* $\mathrm{M} = \Sigma_u^{\mathcal{R}} \circ \Pi_d^{-1}$ *and* $\bullet \vdash \Sigma_u$ **ok**. *Given the following hygiene conditions:*

- $dom(\Gamma_d) \cap \mathbb{P} \subseteq dom(\Pi_d)$
- $dom(\Gamma_d) \cap \mathbb{B} \# \mu$
- $fv(sexp)^{\sigma}_{\Sigma_d} \cap \bigcup_a bv(\mu(a))^{\Pi_d(a)}_{\Sigma_u} = \emptyset$
- $bv(sexp)^{\sigma}_{\Sigma_d} \# \mu$

and a well-typed match:

$$\overline{\Gamma}_u; \bullet; \Sigma_u \vdash \mu : \Pi_d$$

then a well-typed macro template:

$$\Gamma_d :: \overline{\Gamma}_u; \Pi_d \,; \Sigma_d \vdash sexp : \sigma$$

leads to a well-typed transcription:

$$\mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet \,; \mathrm{M} \circ \Sigma_d \vdash \mu(sexp) : \sigma$$

Proof. By induction on the type derivation. We present several of the more interesting cases here; the full proof appears in Appendix A.

The case of type rule [F-MACAPP] is reasonably mechanical:

$$
\begin{array}{ll}
 & \Gamma_d :: \overline{\Gamma}_u; \Pi_d \vdash \mathtt{(}\mathit{mexp}\ \mathtt{.}\ \mathit{sexp}\,\mathtt{:}\,\sigma\mathtt{)} : \mathsf{expr} \\
\Longrightarrow & \{\text{inversion of Rule [F-MacApp]}\} \\
 & \Gamma_d :: \overline{\Gamma}_u; \Pi_d; \mathcal{P}(\sigma, \mathit{sexp}) \vdash \mathit{sexp} : \sigma \\
\wedge & \Gamma_d :: \overline{\Gamma}_u; \Pi_d \vdash \mathit{mexp} : \sigma' \to \mathsf{expr} \\
\Longrightarrow & \{\text{induction hypothesis}\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \mathcal{P}(\sigma, \mathit{sexp}) \vdash \mu(\mathit{sexp}) : \sigma \\
\wedge & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet \vdash \mu(\mathit{mexp}) : \sigma' \to \mathsf{expr} \\
\Longrightarrow & \{\text{Lemma 6.5.5}\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathcal{P}(\sigma, \mu(\mathit{sexp})) \vdash \mu(\mathit{sexp}) : \sigma \\
\wedge & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet \vdash \mu(\mathit{mexp}) : \sigma' \to \mathsf{expr} \\
\Longrightarrow & \{\text{Rule [F-MacApp]}\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet \vdash \mathtt{(}\mu(\mathit{mexp})\ \mathtt{.}\ \mu(\mathit{sexp})\,\mathtt{:}\,\sigma\mathtt{)} : \mathsf{expr} \\
\Longrightarrow & \{\text{definition of } \mu(-)\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet \vdash \mu(\mathtt{(}\mathit{mexp}\ \mathtt{.}\ \mathit{sexp}\,\mathtt{:}\,\sigma\mathtt{)}) : \mathsf{expr}
\end{array}
$$

The case of type rule [S-Import] is again mechanical but requires ensuring that duplicate variables do not appear when extending the base environment:

$$
\begin{array}{ll}
 & \Gamma_d :: \overline{\Gamma}_u; \Pi_d; \Sigma_d \vdash \mathit{sexp} : \sigma{\downarrow}\beta \\
\Longrightarrow & \{\text{inversion of [S-Import]}\} \\
 & \Pi_d \vdash \mathit{resolve}(\Sigma_d, \beta)\ \mathbf{ok} \\
\wedge & (\mathit{resolve}(\Sigma_d, \beta), \Gamma_d) :: \overline{\Gamma}_u; \Pi_d; \Sigma_d \vdash \mathit{sexp} : \sigma \\
\Longrightarrow & \{\text{induction hypothesis}\} \\
 & \Pi_d \vdash \mathit{resolve}(\Sigma_d, \beta)\ \mathbf{ok} \\
\wedge & \mathrm{M}(\mathit{resolve}(\Sigma_d, \beta), \Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d \vdash \mathit{sexp} : \sigma \\
\Longrightarrow & \{\text{distributivity}\} \\
 & \Pi_d \vdash \mathit{resolve}(\Sigma_d, \beta)\ \mathbf{ok} \\
\wedge & (\mathrm{M}(\mathit{resolve}(\Sigma_d, \beta)), \mathrm{M}(\Gamma_d)) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d \vdash \mathit{sexp} : \sigma \\
\Longrightarrow & \{\text{distributivity}\} \\
 & \Pi_d \vdash \mathit{resolve}(\Sigma_d, \beta)\ \mathbf{ok} \\
\wedge & (\mathit{resolve}(\mathrm{M} \circ \Sigma_d, \beta), \mathrm{M}(\Gamma_d)) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d \vdash \mathit{sexp} : \sigma \\
\Longrightarrow & \{\text{Lemma 6.5.6}\} \\
 & \bullet \vdash \mathit{resolve}(\mathrm{M} \circ \Sigma_d, \beta)\ \mathbf{ok} \\
\wedge & (\mathit{resolve}(\mathrm{M} \circ \Sigma_d, \beta), \mathrm{M}(\Gamma_d)) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d \vdash \mathit{sexp} : \sigma \\
\Longrightarrow & \{\text{Rule [S-Import]}\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d \vdash \mathit{sexp} : \sigma{\downarrow}\beta
\end{array}
$$

The case of type rule [F-PBVar] requires careful manipulation of type environments. Note that at variable nodes, $\mathit{sexp}.\ell = \mu(p.\ell)$.

$$
\begin{array}{ll}
 & \Gamma_d :: \overline{\Gamma}_u; \Pi_d; \Sigma_d \vdash a : \mathsf{expr} \\
\Longrightarrow & \{\text{Rule [F-PBVar]}\} \\
 & \Pi_d^{-1}(\Gamma_d|_{\mathbb{P}}) = \{\Pi_d^{-1}(a) : \mathsf{VAR}, \cdots\} :: \gamma \\
\wedge & \Pi_d(a) <: \mathsf{bvar} \\
\Longrightarrow & \{\text{assumption}\} \\
 & \Pi_d^{-1}(\Gamma_d|_{\mathbb{P}}) = \{\Pi_d^{-1}(a) : \mathsf{VAR}, \cdots\} :: \gamma \\
\wedge & \overline{\Gamma}_u; \bullet; \Sigma_u \vdash \mu(a) : \Pi_d(a) <: \mathsf{bvar} \\
\Longrightarrow & \{\text{subsumption}\} \\
 & \Pi_d^{-1}(\Gamma_d|_{\mathbb{P}}) = \{\Pi_d^{-1}(a) : \mathsf{VAR}, \cdots\} :: \gamma \\
\wedge & \overline{\Gamma}_u; \bullet; \Sigma_u \vdash \mu(a) : \mathsf{bvar} \\
\Longrightarrow & \{\text{inspection of Rules [S-PVar], [S-BVar]}\} \\
 & \Pi_d^{-1}(\Gamma_d|_{\mathbb{P}}) = \{\Pi_d^{-1}(a) : \mathsf{VAR}, \cdots\} :: \gamma \\
\wedge & \mu(a) \in \mathit{Variable} \\
\Longrightarrow & \{\mathit{rng}(\mu) \cap \mathbb{P} = \emptyset\} \\
 & \Pi_d^{-1}(\Gamma_d|_{\mathbb{P}}) = \{\Pi_d^{-1}(a) : \mathsf{VAR}, \cdots\} :: \gamma \\
\wedge & \mu(a) \in \mathbb{B} \\
\Longrightarrow & \{\text{definition of } \mathit{resolve}\} \\
 & \mathrm{M}(\Gamma_d|_{\mathbb{P}}) = \{\mathrm{M}(a) : \mathsf{expr}, \cdots\} :: \Gamma \\
\wedge & \mu(a) \in \mathbb{B} \\
\Longrightarrow & \{\Sigma_u(\ell) = \mathit{sexp}.\ell = \mu(p.\ell)\} \\
 & \mathrm{M}(\Gamma_d|_{\mathbb{P}}) = \{\mu(a) : \mathsf{expr}, \cdots\} :: \Gamma \\
\wedge & \mu(a) \in \mathbb{B} \\
\Longrightarrow & \{\text{definition of } \mathit{resolve}\} \\
 & \mathrm{M}(\Gamma_d) = \Gamma', \{\mu(a) : \mathsf{expr}, \cdots\} :: \Gamma \\
\wedge & \mu(a) \in \mathbb{B} \\
\Longrightarrow & \{\mathit{dom}(\Gamma_d) \cap \mathbb{B} \# \mu\} \\
 & \mathrm{M}(\Gamma_d)(\mu(a)) = \mathsf{expr} \\
\Longrightarrow & \{\text{Rule [F-Expr]}\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet \vdash \mu(a) : \mathsf{expr} \\
\Longrightarrow & \{\text{Rule [S-Expr]}\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d \vdash \mu(a) : \mathsf{expr}
\end{array}
$$

□

6.5.1.4 Preservation

With the above lemmas in place, type preservation is easily proved.

Lemma 6.5.8 (Preservation). *If form : expr and $\mathit{form} \longmapsto_{\varepsilon} \mathit{form}'$ then form' :* expr.

Proof. By Theorem 6.3.2, the $\longmapsto_{\alpha}$ step preserves type. Lemma 6.5.1 allows us to focus in on the redex and 6.5.2 to plug the result back into the expansion context. By Lemma 6.5.3, the $\longmapsto_{\mathsf{subst}}$ rule preserves the type of the redex; as does the $\longmapsto_{\mathsf{trans}}$ rule, thanks to Lemmas 6.5.4 and 6.5.7. The $\longmapsto_{\mathsf{return}}$ rule preserves the type of the redex, by inversion of [F-MACDEF]. □

6.5.2 Progress

Progress ensures that hygienic expansion never gets stuck with a type error. In fact, in the λ_m system, there are no expansion-time errors at all, so if expansion terminates, it terminates with a valid expression in the core language.

Definition 6.5.9 (Pre-redex)**.** *A form is a* ***pre-redex*** *if it belongs to the following grammar:*

$$\begin{array}{rcl} r & ::= & \mathtt{((}\mathsf{syntax\text{-}rules}\ \tau\ \mathtt{(}\overline{p_i\,\mathbf{:}\,\hat{\sigma}_i\ \mathit{form}_i}\mathtt{))}\ \ .\ \ \mathit{sexp}\,\mathbf{:}\,\sigma\mathtt{)} \\ & | & \mathtt{(}\mathsf{letrec\text{-}syntax}\ \mathtt{(}\overline{\mathtt{(}x_i\ m_i\mathtt{)}}\mathtt{)}\ \mathit{form}\mathtt{)} \end{array}$$

Definition 6.5.10 (Fully expanded form)**.** *A form is* ***fully expanded*** *if there do not exist F, r such that $\mathit{form} = F[r]$.*

Lemma 6.5.11 (Progress)**.** *If form : expr then either form is fully expanded or $\mathit{form} \longmapsto_{\varepsilon} \mathit{form}'$ for some form'.*

Proof. Let $\mathit{form} = F[r]$. By decomposition, r is well-typed. By cases on r:

- Case $r = \mathtt{(}\mathsf{letrec\text{-}syntax}\ \mathtt{((}x\ \mathit{form}\mathtt{))}\ \mathit{form}\mathtt{)}$: If $x \notin \mathit{fv}(\mathit{form})$, then r is a redex and rule $\longmapsto_{\mathsf{return}}$ applies. Otherwise, $r \longmapsto_{\mathsf{subst}} r'$ only if the hygiene condition holds:

  $$\forall i.\mathit{bv}(\mathit{form}) \cap \mathit{fv}(m) = \emptyset$$

  Because of Theorem 6.2.2, we can choose an α-equivalent term r'' such that this condition holds, and hence $r \longmapsto_{\alpha} r'' \longmapsto_{\mathsf{subst}} r'$.

- Case $r = \mathtt{((}\mathsf{syntax\text{-}rules}\ \sigma' \to \tau\ \mathtt{(}\overline{\mathtt{(}p_i\,\mathbf{:}\,\hat{\sigma}_i\ \mathit{form}_i\mathtt{)}}\mathtt{))}\ \ .\ \ \mathit{sexp}\,\mathbf{:}\,\sigma\mathtt{)}$: Because r is well-typed, we have $\sigma <: \sigma'$ and $\sigma' <: \cup\{\overline{\hat{\sigma}_i}\}$. Thus by transitivity and the definition of subtyping, $\sigma <: \hat{\sigma}_i$ for some i. So by the match lemma, $match(p_i, sexp)$ exists. Now, because none of the types $\hat{\sigma}_i$ have overlapping shapes, $sexp$ can only match one of the patterns in the macro, so by choosing the first pattern that matches, as dictated by the $\longmapsto_{\mathsf{trans}}$ expansion rule, we necessarily choose p_i.

Again, Theorem 6.2.2 allows us to fulfill the hygiene conditions by choosing an α-equivalent r'' with fresh bindings in $sexp$ and $form_i$. Type rule [F-MACAPP] ensures that parsing the expression produces a well-formed bindings table; in other words, expansion does not fail due to duplicate variable names.

□

6.5.3 Soundness

Recall the definition of *core expressions* from Chapter 4:

$$expr \quad ::= \quad x \mid (\mathsf{lambda}\ (\overline{x})\ expr) \mid (\mathsf{apply}\ \overline{expr}) \mid (\mathsf{quote}\ src)$$

Theorem 6.5.12 (Core expressions)**.** *If $form$: expr and $form$ is fully expanded then $form$ is in bijection with a core expression $expr$.*

Proof. By induction on the type derivation and inspection of the types of the primitives lambda, apply, and quote. □

Theorem 6.5.12 allows us to interchange fully expanded forms with their equivalent core expressions. In particular, we write $form \longmapsto\!\!\!\!\twoheadrightarrow_{\varepsilon} expr$ where $form \longmapsto\!\!\!\!\twoheadrightarrow_{\varepsilon} form'$ for some fully expanded $form' \cong expr$. We also write $form \Uparrow_{\varepsilon}$ to mean $\forall form' : form \longmapsto\!\!\!\!\twoheadrightarrow_{\varepsilon} form'.form' \longmapsto_{\varepsilon}$.

Theorem 6.5.13 (Type soundness)**.** *If $form$: expr then either $form \Uparrow_{\varepsilon}$ or $form \longmapsto\!\!\!\!\twoheadrightarrow_{\varepsilon} expr$.*

Proof. By induction on the length of the reduction sequence, using preservation (Lemma 6.5.8) and progress (Lemma 6.5.11). □

6.6 Confluence

Traditional macro expansion algorithms fix their expansion order to work from the outside-in. That is, the outermost macro application is always the first expanded. Since traditional macros are free to inspect, duplicate, remove, and modify their arguments without restriction, macros could observe any change in the expansion order.

In practice, however, programmers expect certain equivalences to hold, where expansion order should be irrelevant. For example, two nested macro applications should be interleavable if one occurs in an expression position. Consider a use of the **let** macro with a nested use of the **swap!** macro from Chapter 1:

Example 36

```
(let ([x 1]
      [y 2])
  (swap! x y)
  x)
```

Traditional expansion starts with the outer **let** application:

Example 37

```
((lambda (x y)
   (swap! x y)
   x)
 1 2)
```

But given the behavior of **let**, it ought to be harmless to expand the use of **swap!** first instead:

Example 38

```
(let ([x 1]
      [y 2])
  (let ([z x])
    (set! x y)
    (set! y z))
  x)
```

In fact, this *is* a legal choice of expansion order in λ_m, where macros are guaranteed to respect the integrity of their sub-expressions. The definition of expansion contexts leaves the choice of expansion order non-deterministic. The next theorem demonstrates proves that for any non-deterministic expansion choice, expansion can always eventually return to some common term (up to α-equivalence).

The proof of confluence follows the presentation in Barendregt [5].

Lemma 6.6.1 (Substitution). *Let* $\overline{\Gamma};\Pi\,;\Sigma \vdash \mathit{sexp} : \sigma$. *If* $x \neq y$, $x \notin \mathit{fv}(m_2)$ *and* $\mathit{bv}(\mathit{sexp})^{\sigma}_{\Sigma} \cap (\mathit{fv}(m_1) \cup \mathit{fv}(m_2)) = \emptyset$ *then*

$$\mathit{sexp}[m_1/x]^{\sigma}_{\Sigma}[m_2/y]^{\sigma}_{\Sigma} = \mathit{sexp}[m_2/y]^{\sigma}_{\Sigma}[m_1[m_2/y]/x]^{\sigma}_{\Sigma}$$

Proof. By induction on the type derivation.

- Case $\mathit{sexp} = x$: $x[m_1/x][m_2/y] = m_1[m_2/y] = x[m_2/y][m_1[m_2/y]/x]$.

- Case $\mathit{sexp} = y$: We have $y[m_1/x][m_2/y] = y[m_2/y] = m_2$. Moreover, $m_2 = m_2[m_1[m_2/y]/x]$ since $x \notin \mathit{fv}(m_2)$. So $m_2 = m_2[m_1[m_2/y]/x] = y[m_2/y][m_1[m_2/y]/x]$.

- Case $\mathit{sexp} = w \neq x, y$: $w[m_1/x][m_2/y] = w = w[m_2/y][m_1[m_2/y]/x]$.

- Case $\mathit{sexp} =$ (letrec-syntax ((w m)) form): We must consider three cases:

  - Subcase: $w = x$:

$$\begin{aligned} & \mathit{sexp}[m_1/x][m_2/y] \\ = \; & \mathit{sexp}[m_2/y] \\ = \; & \text{(letrec-syntax ((}w\ m[m_2/y]\text{))}\ \mathit{form}[m_2/y]\text{)} \\ = \; & \text{(letrec-syntax ((}w\ m[m_2/y]\text{))}\ \mathit{form}[m_2/y]\text{)}[m_1[m_2/y]/x] \\ = \; & \mathit{sexp}[m_2/y][m_1[m_2/y]/x] \end{aligned}$$

- Subcase: $w = y$:

$$\begin{aligned}
& \mathit{sexp}[m_1/x][m_2/y] \\
= \; & (\textsf{letrec-syntax}\ ((w\ m[m_1/x]))\ \mathit{form}[m_1/x]) \\
= \; & (\textsf{letrec-syntax}\ ((w\ m[m_1[m_2/y]/x]))\ \mathit{form}[m_1[m_2/y]/x]) \\
= \; & \mathit{sexp}[m_1[m_2/y]/x] \\
= \; & \mathit{sexp}[m_2/y][m_1[m_2/y]/x]
\end{aligned}$$

- Subcase: $w \neq x, y$: straightforward application of the induction hypothesis.

The remaining cases are mostly straightforward, with binding forms treated similarly to the last case above. □

Following Barendregt, we define an extended language where redexes may be marked:

$$\mathit{form} ::= \ldots \mid r$$

We adapt all the operations of λ_m to marked terms in the obvious way. Additionally, we define an operation φ that reduces all marked redexes. The definition is given in Figure 6.6.

Lemma 6.6.2. *Let* $\overline{\Gamma}; \Pi\,; \Sigma \vdash \mathit{sexp} : \sigma$.

$$\begin{tikzcd}
\mathit{sexp} \arrow[r, dashed, maps to, two heads, "\varepsilon"] \arrow[d, "|\,|"'] & \mathit{sexp}' \arrow[d, dashed, "|\,|"] \\
|\mathit{sexp}| \arrow[r, maps to, two heads, "\varepsilon"'] & |\mathit{sexp}'|
\end{tikzcd}$$

Proof. Each expansion step in the unmarked sequence is matched by an analogous step in the sequence $\mathit{sexp} \longmapsto\!\!\!\!\twoheadrightarrow_{\varepsilon} \mathit{sexp}'$, with the only difference being that some redexes may be marked. □

Lemma 6.6.3. *Let* $\overline{\Gamma}; \Pi\,; \Sigma \vdash \mathit{sexp} : \sigma$. *If* $\mathit{bv}(\mathit{sexp})^{\sigma}_{\Sigma} \cap \mathit{fv}(m) = \emptyset$ *then*

$$\varphi(\mathit{sexp}[m/x]^{\sigma}_{\Sigma}) = \varphi(\mathit{sexp})[m/x]^{\sigma}_{\Sigma}$$

$$
\begin{array}{ll}
\varphi(var) & = var \\
\varphi(\texttt{(letrec-syntax ((}x\ m\texttt{))}\ form\texttt{)}) & = \texttt{(letrec-syntax ((}x\ m\texttt{))}\ \varphi(form)\texttt{)} \\
\varphi(\texttt{(}mexp\ \texttt{.}\ sexp\,\texttt{:}\,\sigma\texttt{)}) & = \texttt{(}\varphi(mexp)\ \texttt{.}\ \varphi(sexp)\,\texttt{:}\,\sigma\texttt{)} \\
\varphi(\ \texttt{(letrec-syntax ((}x\ m\texttt{))}\ form\texttt{)}\ ) & = \texttt{(letrec-syntax ((}x\ m\texttt{))}\ \varphi(form)[m/x]\texttt{)} \\
\qquad \text{if } x \in fv(form) & \\
\quad \text{and } bv(form) \cap fv(m) = \emptyset & \\
\varphi(\ \texttt{(letrec-syntax ((}x\ m\texttt{))}\ form\texttt{)}\ ) & = \varphi(form) \\
\quad \text{if } x \notin fv(form) & \\
\varphi(\ \texttt{((syntax-rules}\ \tau\ \texttt{(}\overline{\texttt{(}p_i\,\texttt{:}\,\hat{\sigma}_i\ form_i\texttt{)}}\texttt{))}\ \texttt{.}\ sexp\,\texttt{:}\,\sigma\texttt{)}\ ) & = \mu(form_i) \\
\qquad \text{if } \mu = match(p_i, \varphi(sexp)) & \\
\quad \text{and } bv(sexp)^{\hat{\sigma}_i}_{\mathcal{P}(\hat{\sigma}_i, sexp)} \cap fv(form_i) = \emptyset & \\
\quad \text{and } bv(form_i) \# sexp & \\
\varphi(m) & = m \\
\varphi(\texttt{()}) & = \texttt{()} \\
\varphi(\texttt{(}sexp_1\ \texttt{.}\ sexp_2\texttt{)}) & = \texttt{(}\varphi(sexp_1)\ \texttt{.}\ \varphi(sexp_2)\texttt{)}
\end{array}
$$

Figure 6.6: Reducing marked redexes.

Proof. By induction on the type derivation. □

Lemma 6.6.4. *Let* $\overline{\Gamma}; \Pi; \Sigma \vdash sexp : \hat{\sigma}$ *and* $\overline{\Gamma}; \bullet \vdash form : \mathsf{expr}$. *If the hygiene conditions* $bv(sexp)^{\hat{\sigma}}_{\mathcal{P}(\hat{\sigma}, sexp)} \cap fv(form) = \emptyset$ *and* $bv(form) \# sexp$ *hold, then*

$$match(p, \varphi(sexp))(form) = \varphi(match(p, sexp)(form))$$

Proof. By induction on the type derivation of $form$ (using a generalized induction hypothesis). □

Lemma 6.6.5. *Let* $\overline{\Gamma}; \Pi; \Sigma \vdash sexp : \sigma$.

$$\begin{tikzcd}
sexp \arrow[r, mapsto, "\varepsilon"] \arrow[d, "\varphi"'] & sexp' \arrow[d, "\varphi"] \\
\varphi(sexp) \arrow[r, mapsto, dashed, two heads, "\varepsilon"'] & \varphi(sexp')
\end{tikzcd}$$

Proof. By induction on the type derivation, using Lemmas 6.6.3 and 6.6.4.

□

Lemma 6.6.6. *Let* $\overline{\Gamma}; \Pi\, ; \Sigma \vdash sexp : \sigma$.

$$\begin{tikzcd}
sexp \arrow[r, maps to, two heads, "\varepsilon"] \arrow[d, "\varphi"'] & sexp' \arrow[d, "\varphi"] \\
\varphi(sexp) \arrow[r, maps to, two heads, dashed, "\varepsilon"'] & \varphi(sexp')
\end{tikzcd}$$

Proof. Straightforward induction on the length of the expansion sequence, using Lemma 6.6.5. □

Lemma 6.6.7. *Let* $\overline{\Gamma}; \Pi\, ; \Sigma \vdash sexp : \sigma$.

$$\begin{tikzcd}
 & sexp \arrow[dl, "|\,|"'] \arrow[dr, "\varphi"] & \\
|sexp| \arrow[rr, maps to, two heads, dashed, "\varepsilon"'] & & \varphi(sexp)
\end{tikzcd}$$

Proof. Straightforward induction on the type derivation. □

Lemma 6.6.8 (Strip Lemma)**.**

$$\begin{tikzcd}
 & sexp \arrow[dl, maps to, "\varepsilon"'] \arrow[ddrr, maps to, two heads, "\varepsilon"] & & \\
sexp_1 \arrow[ddrr, maps to, two heads, dashed, "\varepsilon"'] & & & \\
 & & & sexp_2 \arrow[dl, maps to, two heads, dashed, "\varepsilon"] \\
 & & sexp_3 &
\end{tikzcd}$$

Proof. Construct a term $sexp'$ by marking the redex of $sexp \longmapsto_{\varepsilon} sexp_1$. Then $|sexp'| = s$ and $\varphi(sexp') = sexp_1$. Construct the following diagram using

Lemmas 6.6.2, 6.6.6, and 6.6.7.

$$\begin{tikzcd}
& \mathit{sexp} \arrow[drrr, maps to, two heads, "\varepsilon"] \arrow[dddl, maps to, two heads, "\varepsilon"'] & & & & \\
& & & & \mathit{sexp}_2 \arrow[dddl, dashed, maps to, two heads, "\varepsilon"] & \\
& & \mathit{sexp}' \arrow[uul, "="] \arrow[dll, "\varphi"] \arrow[drrr, dashed, maps to, two heads, "\varepsilon"] & & & \\
\mathit{sexp}_1 \arrow[drrr, dashed, maps to, two heads, "\varepsilon"'] & & & & & \mathit{sexp}'_2 \arrow[uul, dashed, "="'] \arrow[dll, dashed, "\varphi"] \\
& & & \mathit{sexp}_3 & &
\end{tikzcd}$$

□

Theorem 6.6.9 (Confluence). *Let* $\mathit{form} : \mathsf{expr}$ *and* $\mathit{form}' : \mathsf{expr}$.

$$\begin{tikzcd}
& \mathit{form} =_{\alpha} \mathit{form}' \arrow[dl, maps to, two heads, "\varepsilon"'] \arrow[dr, maps to, two heads, "\varepsilon"] & \\
\mathit{form}_1 \arrow[dr, dashed, maps to, two heads, "\varepsilon"'] & & \mathit{form'}_1 \arrow[dl, dashed, maps to, two heads, "\varepsilon"] \\
& \mathit{form}_2 =_{\alpha} \mathit{form'}_2 &
\end{tikzcd}$$

Proof. Induction on the length of the expansion sequence $\mathit{form} \longmapsto\!\!\!\twoheadrightarrow_{\varepsilon} \mathit{form}_1$, using the Strip Lemma. □

6.7 Hygiene

The central correctness result of this thesis, *hygiene*, follows as a direct consequence of confluence.

Theorem 6.7.1 (Hygiene). *Let* $\mathit{form} : \mathsf{expr}$ *and* $\mathit{form}' : \mathsf{expr}$. *If* $\mathit{form} =_{\alpha} \mathit{form}'$ *and* $\mathit{form} \longmapsto\!\!\!\twoheadrightarrow_{\varepsilon} \mathit{expr}$ *and* $\mathit{form}' \longmapsto\!\!\!\twoheadrightarrow_{\varepsilon} \mathit{expr}'$ *then* $\mathit{expr} =_{\alpha} \mathit{expr}'$.

Proof. Since the expansion relation $\longmapsto_{\varepsilon}$ is defined on α-equivalence classes and confluence guarantees unique normal forms, expr and expr' must be in the same α-equivalence class. □

CHAPTER 7

Expressiveness and Limitations

This chapter discusses the expressiveness of the λ_m system, including examples from the Scheme standard library, and its limitations.

7.1 Useful extensions

It was helpful to keep the λ_m model as small as possible for theoretical investigation. We can now relax some of the restrictions and consider a few additions in order to explore the expressiveness of the system.

7.1.1 Front end

Recall from Chapter 4 that the end-to-end view of a macro expansion system begins with an uninterpreted symbolic tree:

$$\mathit{src} ::= \mathit{sym} \mid \texttt{()} \mid \texttt{(}\mathit{src}\ \texttt{.}\ \mathit{src}\texttt{)}$$

We can transform a source program src to a fully-annotated form through a type-directed elaboration process.

The elaboration process is presented in Figures 7.1 and 7.2. Similar to the type judgments, the elaboration judgments maintain type environments, which in this context are used primarily to determine how to interpret symbols in the input source.

The elaboration process assumes that each universe of variables $\mathbb{B}$ and $\mathbb{P}$ is in bijection to the symbols. In particular, there is a map $\lfloor - \rfloor : \mathit{Variable} \to \mathit{Symbol}$ which is injective in $\mathbb{B}$ and $\mathbb{P}$ separately, but not in $\mathit{Variable} = \mathbb{B} \cup \mathbb{P}$. That is, every symbol sym has exactly one preimage $x \in \mathbb{B}$ and exactly one preimage $a \in \mathbb{P}$. This naturally leads to an ambiguity during elaboration: what to do if a symbol is bound in both environments $\overline{\Gamma}$ and Π. Following Scheme, we let the pattern environment take precedent: if a symbol is bound in Π then it is a pattern variable; if it is bound only in $\overline{\Gamma}$ it is a base variable; otherwise it can only be a quoted symbol.

We also assume a representation of types as source with an unspecified elaboration judgment:

$$\vdash \mathit{src} \hookrightarrow \sigma$$

Any faithful representation of syntax types suffices for our purposes.

The judgment $\overline{\Gamma}; \Pi \vdash \mathit{src} \hookrightarrow \mathit{form}$ elaborates input source to a form. The rule for macro definitions recognizes a use of $\mathit{letrec\text{-}syntax}$ only when the identifier has not been shadowed. Variable references are elaborated using an auxiliary judgment defined below. Macro applications are elaborated with the help of an operation bind, defined by:

$$\begin{aligned}
\mathit{bind}(\sigma, \mathit{src}, \Pi) &\stackrel{\text{def}}{=} \mathit{src}[\ell \mapsto \mathit{binding}(\mathit{src}.\ell, \Pi) \mid \ell \in \mathit{dom}(\mathit{src}) \cap \mathit{bp}(\sigma)] \\
\mathit{binding}(\mathit{sym}, \Pi) &\stackrel{\text{def}}{=} \begin{cases} a & \text{if } a \in \mathit{dom}(\Pi) \wedge \lfloor a \rfloor = \mathit{sym} \\ x & \text{if } a \notin \mathit{dom}(\Pi) \wedge \lfloor x \rfloor = \mathit{sym} \end{cases}
\end{aligned}$$

This operation essentially elaborates all binding occurrences of variables so that parsing can obtain an accurate binding table Σ. Function applications are recognized as application of anything other than a syntax operator.

The judgment $\overline{\Gamma}; \Pi \vdash \mathit{src} \hookrightarrow m$ elaborates input source to a macro. We elaborate the annotated type and each clause. Clauses are elaborated by taking the input pattern p_i and specializing the annotated type σ to obtain the type $\hat{\sigma}_i$.

The specialization judgment $\sigma \Downarrow p$ specializes the annotated type σ of a macro to the specific subtype matched by a single pattern p of the macro.

$$\boxed{\overline{\Gamma}; \Pi \vdash \mathit{src} \hookrightarrow \mathit{form}}$$

$$\begin{array}{c}
\lfloor \mathit{letrec\text{-}syntax} \rfloor = \mathit{sym}_0 \qquad \mathit{letrec\text{-}syntax} \notin \mathit{dom}(\Gamma :: \overline{\Gamma}) \\
\lfloor x \rfloor = \mathit{sym} \qquad \Gamma' = \{x : \tau\} :: \Gamma \qquad \Gamma' :: \overline{\Gamma}; \bullet \vdash \mathit{src} \hookrightarrow m \qquad m : \tau \\
\Gamma' :: \overline{\Gamma}; \bullet \vdash \mathit{src}_0 \hookrightarrow \mathit{form} \\
\hline
\Gamma :: \overline{\Gamma}; \bullet \vdash \texttt{(}\mathit{sym}_0\ \texttt{((}\mathit{sym}\ \mathit{src}\texttt{))}\ \mathit{src}_0\texttt{)} \hookrightarrow \texttt{(}\mathsf{letrec\text{-}syntax}\ \texttt{((}x\ m\texttt{))}\ \mathit{form}\texttt{)}
\end{array}$$

$$\begin{array}{c}
\overline{\Gamma}; \Pi \vdash \mathit{sym} \hookrightarrow \mathit{var} : \sigma \\
\hline
\overline{\Gamma}; \Pi \vdash \mathit{sym} \hookrightarrow \mathit{var}
\end{array}
\qquad
\begin{array}{c}
\overline{\Gamma}; \Pi \vdash \mathit{sym} \hookrightarrow \mathit{var} : \sigma \to \mathsf{expr} \\
\overline{\Gamma}; \Pi; \mathcal{P}(\sigma, \mathit{bind}(\mathit{src}, \Pi)) \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \sigma \\
\hline
\overline{\Gamma}; \Pi \vdash \texttt{(}\mathit{sym}\ \texttt{.}\ \mathit{src}\texttt{)} \hookrightarrow \texttt{(}\mathit{var}\ \texttt{.}\ \mathit{sexp}\,\texttt{:}\,\sigma\texttt{)}
\end{array}$$

$$\begin{array}{c}
\mathit{src}_1 \notin \mathit{Symbol} \vee \overline{\Gamma}; \Pi \vdash \mathit{sym} \not\hookrightarrow \sigma \to \mathsf{expr} \\
\overline{\Gamma}; \Pi; \mathcal{P}(\mathit{actuals}, \texttt{(}\mathit{src}_1\ \texttt{.}\ \mathit{src}_2\texttt{)}) \vdash \texttt{(}\mathit{src}_1\ \texttt{.}\ \mathit{src}_2\texttt{)} \hookrightarrow \mathit{sexp} : \mathit{actuals} \\
\hline
\overline{\Gamma}; \Pi \vdash \texttt{(}\mathit{src}_1\ \texttt{.}\ \mathit{src}_2\texttt{)} \hookrightarrow \texttt{(}\mathsf{apply}\ \texttt{.}\ \mathit{sexp}\,\texttt{:}\,\mathit{actuals}\texttt{)}
\end{array}$$

$$\boxed{\overline{\Gamma}; \Pi \vdash \mathit{src} \hookrightarrow m}$$

$$\begin{array}{c}
\lfloor \mathit{syntax\text{-}rules} \rfloor = \mathit{sym}_0 \qquad \mathit{syntax\text{-}rules} \notin \mathit{dom}(\overline{\Gamma}) \qquad \vdash \mathit{src} \hookrightarrow \sigma \to \mathsf{expr} \\
\forall i. \lfloor p_i \rfloor = \mathit{src}_i \wedge \sigma \Downarrow p_i = \hat{\sigma}_i \wedge (\varepsilon :: \overline{\Gamma}); p_i\,\texttt{:}\,\hat{\sigma}_i \vdash \mathit{src'}_i \hookrightarrow \mathit{form}_i \\
\hline\hline
\overline{\Gamma}; \Pi \vdash \texttt{(}\mathit{sym}_0\ \mathit{sym}\ \texttt{(}\overline{\texttt{(}\mathit{src}_i\ \mathit{src}'_i\texttt{)}}\texttt{))} \hookrightarrow \texttt{(}\mathsf{syntax\text{-}rules}\ \sigma \to \mathsf{expr}\ \texttt{(}\overline{\texttt{(}p_i\,\texttt{:}\,\hat{\sigma}_i\ \mathit{form}_i\texttt{)}}\texttt{))}
\end{array}$$

$$\boxed{\sigma \Downarrow p}$$

$$\begin{array}{rcll}
\sigma \Downarrow a & = & \sigma \Downarrow \varepsilon & \\
\texttt{()} \Downarrow \texttt{()} & = & \texttt{()} & \\
\texttt{(}\sigma_1\ \texttt{.}\ \sigma_2\texttt{)} \Downarrow \texttt{(}p_1\ \texttt{.}\ p_2\texttt{)} & = & \texttt{((}\sigma_1 \Downarrow p_1\texttt{)}\ \texttt{.}\ \texttt{(}\sigma_2 \Downarrow p_2\texttt{))} & \\
(\sigma{\uparrow}\beta) \Downarrow p & = & (\sigma \Downarrow p){\uparrow}\beta & p \notin \mathbb{P} \\
(\sigma{\downarrow}\beta) \Downarrow p & = & (\sigma \Downarrow p) \Downarrow \beta & p \notin \mathbb{P} \\
(\mu A.\sigma) \Downarrow p & = & (\sigma[\mu A.\sigma/A]) \Downarrow p & p \notin \mathbb{P} \\
(\cup\{\overline{\sigma_i}\}) \Downarrow p & = & \sigma_i \Downarrow p & p \notin \mathbb{P} \\
\text{for smallest } i & & &
\end{array}$$

Figure 7.1: Elaboration of source programs.

$$\boxed{\overline{\Gamma}; \Pi \vdash \mathit{sym} \hookrightarrow \mathit{var} : \sigma}$$

$$\frac{\begin{array}{c} \mathit{sym} \notin \lfloor \mathit{dom}(\Pi) \rfloor \\ \overline{\Gamma}(x) = \tau \qquad \lfloor x \rfloor = \mathit{sym} \end{array}}{\overline{\Gamma}; \Pi \vdash \mathit{sym} \hookrightarrow x : \tau} \qquad \frac{\Pi(a) = \sigma \qquad \lfloor a \rfloor = \mathit{sym}}{\overline{\Gamma}; \Pi \vdash \mathit{sym} \hookrightarrow a : \sigma}$$

$$\boxed{\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \sigma}$$

$$\frac{\overline{\Gamma}; \Pi \vdash \mathit{sym} \hookrightarrow \mathit{var}}{\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sym} \hookrightarrow \mathit{var} : \sigma} \qquad \frac{\mathit{sym} \notin \lfloor \mathit{dom}(\Pi) \rfloor \qquad \lfloor x \rfloor = \mathit{sym}}{\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sym} \hookrightarrow x : \mathsf{bvar}}$$

$$\frac{\overline{\Gamma}; \Pi \vdash \mathit{src} \hookrightarrow \mathit{form}}{\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src} \hookrightarrow \mathit{form} : \mathsf{expr}} \qquad \frac{\mathit{data} = \mathit{src}[a/\lfloor a \rfloor \mid a \in \mathit{dom}(\Pi)]}{\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src} \hookrightarrow \mathit{data} : \mathsf{data}}$$

$$\frac{}{\overline{\Gamma}; \Pi; \Sigma \vdash \texttt{()} \hookrightarrow \texttt{()} : \texttt{()}}$$

$$\frac{\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src}_1 \hookrightarrow \mathit{sexp}_1 : \sigma_1 \qquad \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src}_2 \hookrightarrow \mathit{sexp}_2 : \sigma_2}{\overline{\Gamma}; \Pi; \Sigma \vdash \texttt{(}\mathit{src}_1 \texttt{ . } \mathit{src}_2\texttt{)} \hookrightarrow \texttt{(}\mathit{sexp}_1 \texttt{ . } \mathit{sexp}_2\texttt{)} : \texttt{(}\sigma_1 \texttt{ . } \sigma_2\texttt{)}}$$

$$\frac{\begin{array}{c} \mathit{src} \notin \lfloor \mathit{dom}(\Pi) \rfloor \qquad \mathrm{B} = \mathit{resolve}(\Sigma, \beta) \\ (\mathrm{B}, \Gamma) :: \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \sigma \end{array}}{\Gamma :: \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \sigma{\downarrow}\beta} \qquad \frac{\begin{array}{c} \mathit{src} \notin \lfloor \mathit{dom}(\Pi) \rfloor \\ \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \sigma \end{array}}{\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \sigma{\uparrow}\beta}$$

$$\frac{\begin{array}{c} \mathit{src} \notin \lfloor \mathit{dom}(\Pi) \rfloor \qquad \sigma' = \sigma[\mu A.\sigma/A] \\ \overline{\Gamma}; \Pi; \mathcal{P}(\sigma', \mathit{bind}(\mathit{src}, \Pi)) \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \sigma' \end{array}}{\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \mu A.\sigma}$$

$$\frac{\begin{array}{c} \mathit{src} \notin \lfloor \mathit{dom}(\Pi) \rfloor \\ \overline{\Gamma}; \Pi; \mathcal{P}(\sigma_i, \mathit{bind}(\mathit{src}, \Pi)) \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \sigma_i \end{array}}{\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \cup\{\overline{\sigma_i}\}}$$

Figure 7.2: Elaboration of source programs (continued).

Specialization behaves much like parsing, matching the pattern against the annotated type. The operation halts on reaching a pattern variable, returning just the normalized type.

Continuing in Figure 7.2, the judgment $\overline{\Gamma}; \Pi \vdash \mathit{sym} \hookrightarrow \mathit{var} : \sigma$ elaborates a symbol to obtain a variable and its type. If the symbol is bound in Π, it is interpreted as a pattern variable; otherwise if the symbol is bound in $\overline{\Gamma}$ it is treated as a base variable.

The judgment $\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{src} \hookrightarrow \mathit{sexp} : \sigma$ fairly closely parallels the type checking judgment for S-expressions. Symbols are elaborated in the usual way. At type expr, we return to the form-elaboration rules. At type data, we elaborate pattern variables but leave all other symbols unchanged. Empty sequence and pairs and straightforward. Import types make use of the bindings table Σ to extend the environment and recur; export types are unnecessary for elaboration and are dropped. As usual, recursive types and unions require parsing to generate a binding table before recurring.

Because of the similar structure of the elaboration and type-checking processes, an implementation could fuse the two phases into one. Since elaboration is essentially orthogonal to the core model, we have found it preferable to separate the two in our study of λ_m.

7.1.2 Modifications to the core system

In the type system of Chapter 5, we required all macro patterns to be complete for their documented type. Although this helped simplify the theoretical presentation, in practice this requirement is too restrictive. Many macros make certain well-formedness assumptions that are checked during expansion. For example, macros that take two parallel sequences often assume that the sequences are of the same length, and simply let expansion fail otherwise. In a real system, we would likely remove completeness checking, or possibly offer the completeness check as an optional warning.

Next, we could add some basic primitives to the language, including constants such as boolean, number, character, and string literals (all of type expr) and a primitive

$$\mathsf{if} : \cup\{\mathtt{(}\mathsf{expr}\ \mathsf{expr}\mathtt{)}, \mathtt{(}\mathsf{expr}\ \mathsf{expr}\ \mathsf{expr}\mathtt{)}\} \to \mathsf{expr}$$

We can also add a form type ref $<:$ expr corresponding to variable references, and extend subsumption to apply to form types as well.[1] This allows us to add a primitive

$$\mathsf{set!} : \mathtt{(}\mathsf{ref}\ \mathsf{expr}\mathtt{)} \to \mathsf{expr}$$

Other straightforward extensions include multi-ary macro definitions (allowing letrec-syntax to bind any number of macros simultaneously), let-syntax and let*-syntax forms, and a top-level define-syntax form. Proper treatment of the latter form opens up some more tricky questions with regards to definition forms (see Section 7.3.2).

7.2 Standard Scheme macros

With these extensions, most of the core functionality of the standard Scheme macros are expressible in the λ_m system. In this section, we briefly discuss well-typed implementations of the macros of the R^{5}RS standard library.

Several of the simpler macros can be implemented quite easily. The **begin** macro, which takes a non-empty sequence of expressions:

$$actuals^{+} \overset{\text{def}}{=} \mu A.\cup\{\mathtt{(}\mathsf{expr}\mathtt{)}, \mathtt{(}\mathsf{expr}\ \mathtt{.}\ A\mathtt{)}\}$$

and produces an expression, is unproblematic to implement:

[1] Note that there is no relationship between the types ref and bvar, despite the fact that they are both inhabited by identifiers.

Example 39

```
(define-syntax begin
  (syntax-rules $actuals^{+} \rightarrow \mathsf{expr}$
    [(begin e) e]
    [(begin e . es)
     (let ([tmp e])
       (begin . es))]))
```

Another straightforward implementation is the **delay** macro, which produces a "promise" (a data structure encapsulating a lazy computation):

Example 40

```
(define-syntax delay
  (syntax-rules $(\mathsf{expr}) \rightarrow \mathsf{expr}$
    [(delay e)
     (make-promise (lambda () e))]))
```

Of the trilogy of local binding forms, the easiest to write is **let**$*$. Its type, however is a little less obvious. The **let**$*$ macro consumes a sequence of clauses:

$$let{*}\text{-}clauses \overset{\text{def}}{=}$$

$$\mu A.\cup\{\texttt{()}\uparrow\varepsilon, \texttt{((}\mathsf{bvar\ expr}\texttt{) .}\ A\downarrow\{\mathsf{AA}:\mathsf{VAR}\} :: \varepsilon\texttt{)}\uparrow\mathsf{D}@\{\mathsf{AA}:\mathsf{VAR}\} :: \varepsilon\}$$

Each clause consists of a bound variable and an expression, where the remaining clauses are in the scope of the bound variable. The entire sequence then exports the bound variables. The type of **let**$*$ is:

$$\texttt{(}let{*}\text{-}clauses\ \mathsf{expr}\downarrow\mathsf{A}@\varepsilon\texttt{)} \rightarrow \mathsf{expr}$$

which takes the bindings defined in the clauses and imports them into the scope of the body expression.

The definition is as easy as can be:

Example 41

```
(define-syntax let*
  (syntax-rules $(let{*}\text{-}clauses\ \mathsf{expr}\downarrow\mathsf{A}@\varepsilon) \rightarrow \mathsf{expr}$
   [(let* () body) body]
   [(let* ((x e) . cs) body)
    (let ([x e])
      (let* cs body))]))
```

Scheme's **letrec** form is tricker to implement, although this has at least as much to do with the subtle semantics of **letrec** and the limitations of **syntax-rules** as a macro language as it does with the types. The type of **letrec**'s binding clauses is similar to that of the **let**$*$ clauses:

$$\begin{aligned}&\mathit{letrec\text{-}clauses} \stackrel{\text{def}}{=}\\ &\qquad \mu A.\cup\{\texttt{()}\!\uparrow\!\{\}, \texttt{((bvar expr) . } A\texttt{)}\!\uparrow\!\{\mathsf{AA}\!:\!\textsc{var}, \mathsf{D}\!:\!\textsc{rib}\}\}\end{aligned}$$

With **letrec**, the clauses export a single rib of bindings; the **letrec** form then imports these bindings both into the rib itself and the body expression:

$$\texttt{(}\mathit{letrec\text{-}clauses}\ \texttt{expr)}\!\downarrow\!\{\mathsf{A}\!:\!\textsc{rib}\} :: \varepsilon \to \texttt{expr}$$

The subtlety in the semantics of **letrec** is that it requires *all* the right-hand side initializer expressions to be evaluated before assigning the results to the bound variables. So we must take care to generate code that evaluates these steps in the proper order. Our implementation proceeds in several steps. The first step is to "unzip" the binding/initializer pairs into separate sequences using an auxiliary macro *letrec/unzip*. This macro takes three accumulators: the unzipped bindings, a sequence of #f literals, and the unzipped initializer expressions. It also consumes the clauses and body of the **letrec** expression. In the base case, *letrec/unzip* binds the variables to #f and uses a second auxiliary macro, *begin-set!*, to evaluate the intializer expressions and then assign their results to the bound variables.

The *begin-set!* macro takes the bound variables, initializer expressions and body expression, evaluates each expression, and binds its result to a temporary variable. The macro recurs by adding an assignment to the body from the temporary to the variable *x*. Because the base case places the body expression at the very end of this sequence, all of the assignments end up occurring after the evaluation of the initializer expressions.

The type of *begin-set!* requires the type of a sequence of references:

$$\mathit{refs} \stackrel{\text{def}}{=} \mu A.\cup\{\texttt{()}, \texttt{(ref . } A\texttt{)}\}$$

Example 42

```
(define-syntax letrec
  (syntax-rules (letrec-clauses expr)↓{A:RIB} :: ε → expr
    [(letrec cs body)
     (letrec/unzip () () () cs body)]))

(define-syntax letrec/unzip
  (syntax-rules
    (formals actuals actuals letrec-clauses expr↓{A:RIB, ADDD:RIB} :: ε)
       → expr
    [(letrec/unzip xs is es () body)
     ((lambda xs
         (begin-set! xs es body))
      . is)]
    [(letrec/unzip xs is es ((x e) . cs) body)
     (letrec/unzip (x . xs) (#f . is) (e . es) cs body)]))

(define-syntax begin-set!
  (syntax-rules (refs actuals expr) → expr
    [(begin-set! () () body) body]
    [(begin-set! (x . xs) (e . es) body)
     ((lambda (tmp)
         (begin-set! xs es (begin (set! x tmp) body)))
      e)]))
```

Next we tackle the **let** macro. The primary complication is handling the overloaded "named **let**" syntax, which provides a convenient form for defining and immediately applying a recursive function. This is not particularly hard to handle from the perspective of typing, since it simply requires a union type:

$$\begin{aligned}
\mathit{named\text{-}let} &\overset{\text{def}}{=} (\mathsf{bvar}\ \mathit{clauses}\ \mathsf{expr}{\downarrow}\{\mathsf{A}:\textsc{var}\} :: \{\mathsf{AD}:\textsc{rib}\} :: \varepsilon) \\
\mathit{std\text{-}let} &\overset{\text{def}}{=} (\mathit{clauses}\ \mathsf{expr}{\downarrow}\{\mathsf{A}:\textsc{rib}\}) \\
\mathit{clauses} &\overset{\text{def}}{=} \mu A.\cup\{()\uparrow\{\}, ((\mathsf{bvar}\ \mathsf{expr})\ .\ A)\uparrow\{\mathsf{AA}:\textsc{var}, \mathsf{D}:\textsc{rib}\}\}
\end{aligned}$$

The type of **let** is then composed of the union of the two overloaded forms:

$$\cup\{\mathit{named\text{-}let}, \mathit{std\text{-}let}\} \to \mathsf{expr}$$

Notice that the union type is well-formed, since the shapes do not overlap.

```
                                                          Example 43
(define-syntax let
  (syntax-rules ∪{named-let, std-let} → expr
    [(let cs body)
     (let/unzip () () clauses body)]
    [(let f inits body)
     (rec/unzip f () () inits body)]))

(define-syntax let/unzip
  (syntax-rules
    (formals actuals clauses expr↓{A:RIB, ADD:RIB} :: ε) → expr
    [(let/unzip xs es () body)
     ((lambda xs body) . es)]
    [(let/unzip xs es ((x e) . cs) body)
     (let/unzip (x . xs) (e . es) cs body)]))

(define-syntax rec/unzip
  (syntax-rules
    (bvar formals actuals clauses expr↓{A:VAR} :: {AD:RIB, ADDD:RIB} :: ε)
      → expr
    [(rec/unzip f xs es () body)
     (letrec ([f (lambda xs body)])
       (f . es))]
    [(rec/unzip f xs es ((x e) . inits) body)
     (rec/unzip f (x . xs) (e . es) inits body)]))
```

The implementation of **let** uses the shape to dispatch to one of two helper macros. Just as with **letrec**, the helper macros unzip the binding/initializer pairs into separate sequences. The **let/unzip** helper macro places the unzipped bindings in the formals list of a **lambda** and initializers in the actuals list. The **rec/unzip** helper macro places the bindings in a **lambda**, binds the recursive variable *f* in a **letrec**, and applies *f* to the initializer expressions.

The last macro we demonstrate is **cond**. We implement only two of the four clause types, since the other two depend on distinguished syntactic literals (see Section 7.3.1). A **cond** clause in this implementation takes one of two forms:[2]

$$\mathit{cond\text{-}clause} \overset{\text{def}}{=} \cup\{\texttt{(expr)}, \texttt{(expr . } \mathit{actuals}^{+}\texttt{)}\}$$

[2] Note that this definition of *cond-clause* is in fact equivalent to *actuals*$^{+}$, but we separate the two types for emphasis.

The implementation tries each clause in turn with the auxiliary macro *cond1*. If all clauses fail, it defaults to (**if** #f #f), an expression which returns Scheme's distinguished "void" value. The *cond1* macro accepts either a single expression, which is intended to be both the test expression and the result on success; or in the second case, a test expression followed by any number of result expressions, evaluated with **begin**.

Example 44

```
(define-syntax cond
  (syntax-rules μA.∪{(cond-clause), (cond-clause . A)} → expr
    [(cond clause)
     (cond1 clause (if #f #f))]
    [(cond clause . clauses)
     (cond1 clause (cond . clauses))]))

(define-syntax cond1
  (syntax-rules (cond-clause expr) → expr
    [(cond1 (test) alt)
     (let ([tmp test])
       (if tmp tmp alt))]
    [(cond1 (test . result) alt)
     (if test (begin . result) alt)]))
```

7.3 Limitations

Having demonstrated that the core of the R^{5}RS macros are expressible, we can see that the type system of λ_m supports a promising kernel of the standard idioms of programming with **syntax-rules**. However, there are certain conspicuous omissions. We discuss the most immediate limitations here; a fuller discussion of future work can be found in Chapter 8.

7.3.1 Syntactic literals

Our implementation of **cond** was missing two important cases: the optional final **else** clause, and clauses making use of the => identifier to bind the result of the test expression. Both of these cases make use of the ability in **syntax-rules** macros to identify a list of syntactic "literals," which play the role of syntactic constants in a grammar.

There are several examples in the literature of the use of literals to expose binding information in ways that would likely defeat the invariants of the λ_m system [33, 41, 49]. For this reason, we have so far omitted them from our investigation. However, the known examples all make use of a particular interaction between the literals list and lexically nested macros, which we have also excluded. In fact, even with the addition of both constructs, it may be possible to provide some amount of support for syntactic literals in a conservative way that is nevertheless consistent with the existing semantics of Scheme macros.

7.3.2 Definitions

As we presented them, most of the macro definitions at the beginning of this chapter are incomplete: they do not support internal definitions. Indeed, definitions are the other inherent form type in Scheme syntax. By adding the type defn to our model, we should be able to support a much richer set of Scheme constructs.

Definitions introduce a few complexities into the semantics of macros. First, because the exports of a definition are imported by their *context* (i.e., the block that contains it), we must adapt the binding signature types to accommodate exports that are communicated between a macro and its container, not just its sub-terms. Second, the division in a block between the initial sequence of definitions and the subsequent sequence of expressions is not explicitly marked, and can even change during expansion by the process of expanding internal macros definitions. Types should be helpful to reign in the unpredictable behavior of blocks, but this will require further research.

7.3.3 Lexically nested macros

Finally, our model does not allow for lexically nested, or "macro-defining" macros. We have in fact begun investigation into the behavior of macro-defining macros, and found that the semantics of Scheme macros leads to very unsatisfying formal models on this score. As it turns out, macro-defining macros betray some of the worst aspects of the legacy of Lisp's raw S-expressions: because the semantics of expansion is defined as simple textual substitution, without regard for binding structure, macro-defining macros tend to expose the data representation of syntax employed by particular macro expansion algorithms. It is therefore quite difficult to present at once a useful theory that nonetheless remains faithful to the behavior of Scheme.

We have thus far chosen to restrict our attention to a subset of the semantics of Scheme macros, in order to gain a better understanding of the theory underlying Scheme. In the future, however, our research into formal foundations of hygienic macros may lead to the design of alternative systems. We might even dare to hope that principled research could help simplify and demystify macro-defining macros, which are to date considered one of the more advanced techniques in macrology.

CHAPTER 8

Discussion

In this dissertation, we have presented λ_m, a model of hygienic macro expansion that validates the claim that it is possible for formulate a precise definition of hygienic expansion by making the intended binding structure of macros explicit. We have demonstrated that hygiene can be characterized as the preservation of α-equivalence and proved the property holds for the λ_m language. We have also demonstrated that with some minor extensions, this theoretical model can express many of the macros of the R^{5}RS Scheme standard—an encouraging initial sign that with further development, our type system might eventually form the basis for practical language design.

8.1 Related work

The Scheme community has a long history of proposals for macro systems that attempt, with varying degrees of automation, to address problems with variable capture. These include a number of hygienic macro expansion systems [44, 13, 15, 68, 55]. Hygienic expansion algorithms involve intricate representations of identifiers; expansion must generally perform provisional variable renamings, since it does not discover the actual syntactic roles of identifiers until quite late in the expansion process. By contrast, our system provides all of this information up front, making the specification of expansion relatively simple. (Instead, the complexity of scope and binding structure shows up in the λ_m type system.) In some sense the work on *syntactic closures* [7] shares a similar motivation to our work, namely that macro implementors know the binding structure of their macros and should be able to make this structure explicit. The so-called “first-class macros” of Bawden [6] also involve some level of checked documentation for macros, although the types of λ_m are able to describe richer properties. Another work that shares a similar spirit is Shivers's `loop` macro [58], which renders scope explicit. That work treats control dominance as the salient property of scope, and focuses on ensuring control-flow invariants from within a language of loop sub-forms.

The work of Culpepper and Felleisen on *shape types* [17] is closest in nature and in lineage to ours. Shape types provide enough information to check the syntactic structure of macro arguments, but not enough to track binding structure. Our work began with the question: “how much could be gained by adding binding information to shape types?”

Macros and similar compile-time meta-programming facilities have been added to a number of other programming languages outside the Lisp family [20, 56, 71, 3, 4, 11, 2, 19, 60, 59, 66, 26]. A few of these systems attempt to address issues of hygiene. Several research programs outside of the Scheme tradition have attempted a principled approach to issues of scope and binding. The proposed language MacroML [27] attempts to deal with variable scope by limiting the contexts in which binding structures can be introduced–specifically by making ML's `let` into an extensible form. This leads to a much less expressive macro system, since it does not admit the construction of wholly new syntaxes for binding forms.

The *notational definitions* of Griffin [31], later adapted by Taha and Johann in their work on *staged notational definitions* [64], are quite similar in motivation to our work. Notational definitions allow flexible syntactic extension, and require the extensions to be defined in a style based on higher-order abstract syntax [50], which represents synthesized binding forms with existing binding forms in the meta-language. Thus rather than representing binding structure as an external property of macros, this structure is implicit in the implementation. Their approach also requires macros to be defined externally to a program, which loses the flexibility of Scheme's lexically scoped macros. The system of extensible syntax of Cardelli et al. [12] similarly allows for custom notation while preserving lexical scoping, but without the power of locally defined macros.

The binding specifications of λ_m relate to a large body of prior work in theory and language design. Pottier [54] traces the concept of binding specifications back to Plotkin [52], Talcott's *binding structures* [65], Honsell's *nominal algebras* [34], and Urban et al.'s *nominal signatures* [67]. Each of these frameworks provides a specification language for describing binding structure of programs in an object language. Shinwell's FreshO'Caml [57] and Pottier's Cαml provide more expressive meta-programming constructs with type systems that can describe the binding structure of abstract syntax. But unlike λ_m, these languages operate strictly on abstract syntax. By contrast, the types of λ_m incorporate syntactic structure, which acommodates the more reflective nature of "macros as embedded meta-programs." *Context calculi* [32] allow for manipulation of programs with open terms, facilitating some elements of meta-programming; again, these do not deal directly with macros, though there may be interesting connections worth exploring.

Several authors have investigated models for describing the behavior of Scheme macros as well as designing advanced macro systems [46, 29]. Bove and Arbilla [10] describe a calculus of macro expansion based on de Bruijn indices [18] that attempts to model hygiene in a formal way. Their work also identifies confluence as an important property of hygienic macros. However, similar to the work on notational definitions, their work makes the simplifying assumption that macro definitions are known at the outset of a program. Their work does not include an analog of our binding signatures, which provide a formal notation for user-specified binding structure.

8.2 Future work

There are quite a few interesting areas to explore beyond the simple model presented in this work. Let us consider some of the open questions by topic.

8.2.1 Expressiveness

The λ_m model is still not expressive enough for practical programming. In addition to the open questions detailed in Chapter 7, we would like to explore more expressive pattern-matching languages such as Kohlbecker and Wand's "macro-by-example" ellipsis patterns [45] as well as Culpepper's extremely expressive `syntax-parse` pattern language [16].

Another important question is whether type systems like this one can be adapted to programmatic macros, which implement their macro templates in a more conventional programming language (typically the same as the base language). This would likely require integrating the type system with a conventional type language, probably using constructs from typed multistage programming [63, 62, 47].

8.2.2 Type system extensions

An alternative approach to static, pre-expansion type checking would be an expansion-time checking system. This would ideally still allow reasoning about the binding structure of macros, and would guarantee a kind of partial correctness:

> *If a macro returns without an expansion-time error, then its expansion is guaranteed to respect its documented syntax type.*

Since macros can be passed to other macros as arguments, such an approach would likely make use of ideas from prior work on contracts for higher-order languages [25].

Another open question is whether this type system can be adapted to support a typed base language with the guarantee that well-typed input programs always expand to well-typed expanded programs, as is frequently the case in multi-staged programming languages. This would ensure that derived language constructs are always smoothly integrated with the core language, without resulting in surprising and impenetrable type errors involving the results of expansion.

It would also be useful to investigate the addition of more powerful type system constructs. Polymorphic types would be a useful addition, since generic macros appear to be common, such as macros that accept a "continuation" macro argument. Since duplicating arguments is the source of many bugs such as the duplication of run-time effects, linear types [70, 69, 53, 9, 1] could be a useful construct for ensuring that input expressions appear exactly once in the results of expansion.

8.2.3 Applications

We have demonstrated the applicability of binding signatures to typed hygienic macros. It may be the case that these constructs have applications beyond Scheme. For example, the theorem prover ACL2 [39, 38] makes extensive use of macros, yet reasons only about the results of expansion. It could be helpful to users to be able to provide more robust syntactic abstractions that can integrate into the theorem prover without depending on the results of expansion. Furthermore, the constructs of binding signatures and binding signature types may have additional applications to other forms of meta-programming.

Bibliography

[1] A. Ahmed, M. Fluet, and G. Morrisett. **L**^{3}: A linear language with locations. Technical Report TR-24-04, Harvard University, 2004.

[2] Jonathan Bachrach and Keith Playford. D-Expressions: Lisp power, Dylan style, 1999. `http://people.csail.mit.edu/jrb/Projects/dexprs.pdf`.

[3] Jonthan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *OOPSLA '01: Proceedings of the 16th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 31–42, New York, NY, USA, 2001. ACM.

[4] J. Baker and W. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 270–281, June 2002.

[5] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

[6] Alan Bawden. First-class macros have types. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–141, New York, NY, USA, 2000. ACM.

[7] Alan Bawden and Jonathan Rees. Syntactic closures. In *LISP and Functional Programming*, pages 86–95, 1988.

[8] Jon Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[9] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *CSL '94: Proceedings of the 8th International Workshop on Computer Science Logic*, number 933 in Lecture Notes in Computer Science, pages 121–135, Heidelberg, 1995. Springer-Verlag.

[10] Ana Bove and Laura Arbilla. A confluent calculus of macro expansion and evaluation. In *LISP and Functional Programming*, pages 278–287. ACM Press, June 1992.

[11] C. Brabrand, M. Schwartzbach, and M. Vanggaard. The metafront system: Extensible parsing and transformation. In *LDTA '03: Proceedings of the 3rd ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications*, April 2003.

[12] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible syntax with lexical scoping. Technical Report SRC-RR-121, DEC Systems Research Center, February 1994.

[13] William Clinger and Jonathan Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–162, 1991.

[14] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. Technical report, 1991.

[15] William D. Clinger. Hygienic macros through explicit renaming. *Lisp Pointers*, (4):25–28, December 1991.

[16] Ryan Culpepper. *Refining Syntactic Sugar: Tools for Supporting Macro Development*. PhD thesis, Northeastern University, April 2010.

[17] Ryan Culpepper and Matthias Felleisen. Taming macros. In *GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, pages 153–165, October 2004.

[18] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[19] Rodrigo B. de Oliveira. The Boo programming language. Online, 2007. `http://boo.codehaus.org/`.

[20] Daniel de Rauglaudre. Camlp4 reference manual. Online, September 2003. `http://pauillac.inria.fr/caml/camlp4/manual/`.

[21] Akim Demaille, Joel E. Denny, and Paul Eggert. *Bison 2.4.1*. Free Software Foundation, 2009. `http://www.gnu.org/software/bison/manual/html_node/index.html`.

[22] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, 4th edition, 2009.

[23] Matthias Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[24] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[25] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, pages 48–59, October 2002.

[26] Fabien Fleutot. Man Metalua. Online reference manual, April 2007. `http://metalua.luaforge.net/metalua-manual.html`.

[27] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *International Conference on Functional Programming*, pages 74–85. ACM Press, 2001.

[28] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2002.

[29] Martin Gasbichler. *Fully-parameterized, first-class modules with hygienic macros*. PhD thesis, University of Tübingen, August 2006.

[30] Saul Gorn. Explicit definitions and linguistic dominoes. In *Systems and Computer Science, Proceedings of the Conference held at University of Western Ontario*, pages 77–115, 1967.

[31] Timothy Griffin. Notational definition—a formal account. In *LICS '88: Proceedings of the 3rd Symposium on Logic in Computer Science*, pages 372–383, 1988.

[32] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, 266(1-2):249–272, 2001.

[33] David Herman and David Van Horn. A few principles of macro design. In *Proceedings of the 2008 Workshop on Scheme and Functional Programming*, pages 89–93, September 2008.

[34] Furio Honsell, Marino Miculan, and Ivan Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, pages 963–978, London, UK, 2001. Springer-Verlag.

[35] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, page 196, 1996.

[36] ISO. The ANSI C standard (C99). Technical report, ISO/IEC, 2005. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf.

[37] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. http://dinosaur.compilertools.net/yacc/.

[38] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[39] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: an Approach*. Kluwer Academic Publishers, 2000.

[40] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[41] Oleg Kiselyov. How to write seemingly unhygienic and referentially opaque macros with syntax-rules. In *Proceedings of the 2002 Workshop on Scheme and Functional Programming*, pages 77–88, 2002.

[42] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[43] Donald E. Knuth. Examples of formal semantics. In E. Engeler, editor, *Symp. on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 212–235. Springer-Verlag, New York–Heidelberg–Berlin, 1971.

[44] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LISP and Functional Programming*, pages 151–161, 1986.

[45] Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *POPL '87: Proceedings of the 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–84, 1987.

[46] Shriram Krishnamurthi. *Linguistic Reuse*. PhD thesis, Rice University, May 2001.

[47] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming*, pages 193–207, 1999.

[48] Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin McMullan. Lexer and parser generators in scheme. In *Proceedings of the 2004 Workshop on Scheme and Functional Programming*, 2004.

[49] Al* Petrofsky. How to write seemingly unhygienic macros using syntax-rules. Online newsgroup posting, November 2001. `http://groups.google.com/group/comp.lang.scheme/msg/5438d13dae4b9f71`.

[50] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM.

[51] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[52] Gordon Plotkin. An illative theory of relations. In Robin Cooper, Kuniaki Mukai, and John Perry, editors, *Situation Theory and its Applications*, volume 1 of *CSLI Lecture Notes*, pages 133–146. Stanford University, 1990.

[53] Gordon Plotkin. Type theory and recursion. In *LICS '93: Proceedings of the 8th Symposium on Logic in Computer Science*, page 374, 1993.

[54] François Pottier. An overview of alphaCaml. In *ML ’05: Proceedings of the 2005 ACM SIGPLAN Workshop on ML*, 2005.

[55] Robert Hieb R. Kent Dybvig and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[56] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *Haskell ’02: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 1–16, 2002.

[57] Mark R. Shinwell. Fresh O’Caml: nominal abstract syntax for the masses. In *ML ’05: Proceedings of the 2005 ACM SIGPLAN Workshop on ML*, 2005.

[58] Olin Shivers. The anatomy of a loop: a story of scope and control. In *ICFP ’05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 2–14, 2005.

[59] Kamil Skalski. Syntax-extending and type-reflecting macros in an object-oriented language. Master’s thesis, University of Wrocław, 2005. `http://nazgul.omega.pl/macros.pdf`.

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

[61] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (Editors). Revised^6 report on the algorithmic language Scheme, 2007.

[62] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[63] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. *Lecture Notes in Computer Science*, 1443, 1998.

[64] Walid Taha and Patricia Johann. Staged notational definitions. In *GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, pages 97–116, 2003.

[65] Carolyn Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112(1):99–143, 1993.

[66] Laurence Tratt. Compile-time meta-programming in a dynamically typed OO language. In *DLS '05: Proceedings of the 2005 Symposium on Dynamic Languages*, pages 49–63, New York, NY, USA, 2005. ACM.

[67] Christian Urban, Andrew Pitts, and Murdoch Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.

[68] André van Tonder. SRFI 72: Hygienic macros. Online, September 2005. `http://srfi.schemers.org/srfi-72/srfi-72.html`.

[69] P. Wadler. Linear types can change the world. In *Programming Concepts and Methods*, pages 347–359, Amsterdam, 1990. North Holland.

[70] D. Walker. Substructural type systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1, pages 3–44. Cambridge, 2005.

[71] Daniel Weise and Roger F. Crew. Programmable syntax macros. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, 1993.

[72] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Appendix A

Additional Proofs

This appendix expands on the material in Chapter 6 with additional lemmas and details of proofs.

A.1 Freshness

Let us define a partial inverse to the region displacement operation:

$$\ell\ell_0 \gg \ell_0 \overset{\text{def}}{=} \ell$$

and lift (and totalize) this definition to bound variable maps:

$$\begin{gathered}\{\overline{\ell_i \mapsto x_i}, \overline{\ell_j \mapsto x_j}\} \gg \ell \overset{\text{def}}{=} \{\overline{\ell_i \gg \ell \mapsto x_i}\} \\ \text{where } \forall i.\ell_i \gg \ell \text{ is defined} \\ \text{and } \forall j.\ell_j \gg \ell \text{ is undefined}\end{gathered}$$

Lemma A.1.1. *If the following propositions hold:*

- $\{\overline{\ell_i \mapsto x_i}\} = bindings(\sigma_0, sexp)$
- $\{\overline{\ell'_j \mapsto x_j}\} = bindings(\sigma, sexp.\ell) = \{\overline{\ell_i \mapsto x_i}\} \gg \ell$ *where* $\{\bar{j}\} \subseteq \{\bar{i}\}$
- $\overline{z_i} \# sexp$
- $\Sigma_0 = parse(\sigma_0, sexp, \epsilon)$ *is defined*
- $\Sigma_1 = parse(\sigma, sexp.\ell, \ell) \sqsubseteq \Sigma_0$ *is defined*

- $\forall \ell' \in \mathit{dom}(\Sigma_0).\mathit{resolve}(\Sigma_0, \Sigma_0(\ell'))$ *is defined*

then the following conclusions hold:

- $\Sigma_2 = \mathit{parse}(\sigma, \mathit{sexp}[\overline{\ell_i \mapsto z_i}].\ell, \ell)$ *is defined; and*

- $\forall \ell' \in \mathit{dom}(\Sigma_3).\mathit{resolve}(\Sigma_3, \Sigma_3(\ell'))$

where $\Sigma_3 = \Sigma_0[\overline{\ell'_j \mapsto z_j}]$.

Proof. By induction on $\mathit{parse}(\sigma, \mathit{sexp}.\ell, \ell)$.

Let us consider the case where $\Sigma_1 = \mathit{parse}(\mu A.\sigma, \mathit{sexp}.\ell, \ell)$. By definition:

- $\Sigma_1 = \{\ell \mapsto \mathcal{P}(\sigma[\mu A.\sigma/A], \mathit{sexp}.\ell, \epsilon)(\epsilon)\}$; and

- $\Sigma'_0 = \mathit{parse}(\sigma[\mu A.\sigma/A], \mathit{sexp}.\ell.\epsilon, \epsilon)$ is defined; and

- $\Sigma'_1 = \Sigma'_0 = \Sigma'_0 \sqsubseteq \Sigma_0$ is defined; and

- $\forall \ell' \in \mathit{dom}(\Sigma'_0).\mathit{resolve}(\Sigma'_0, \Sigma'_0(\ell'))$ is defined.

Note that $\mathit{bindings}(\mu A.\sigma, \mathit{sexp}.\ell) = \mathit{bindings}(\sigma[\mu A.\sigma/A], \mathit{sexp}.\ell) = \{\overline{\ell'_j \mapsto x_j}\}$. By the induction hypothesis, $\Sigma'_2 = \mathit{parse}(\sigma[\mu A.\sigma/A], \mathit{sexp}.\ell[\overline{\ell'_j \mapsto z_j}].\epsilon, \epsilon)$ is defined and $\forall \ell' = \mathit{dom}(\Sigma'_3).\mathit{resolve}(\Sigma'_3, \Sigma'_3(\ell'))$ is defined, where

$$\Sigma'_3 = \Sigma'_0[\overline{\ell'_j \mapsto z_j}] = \Sigma'_2$$

Now, $\mathit{sexp}.\ell[\overline{\ell'_j \mapsto z_j}].\epsilon = \mathit{sexp}[\overline{\ell_i \mapsto z_i}].\ell$, so $\mathcal{P}(\sigma[\mu A.\sigma/A], \mathit{sexp}[\overline{\ell_i \mapsto z_i}].\ell)$ is defined and fully resolved. Thus $\Sigma_2(\ell)$ is defined and fully resolved. □

Corollary. *If* $\{\overline{\ell_i \mapsto x_i}\} = \mathit{bindings}(\sigma, \mathit{sexp})$ *and* $\mathcal{P}(\sigma, \mathit{sexp})$ *is defined and* $\overline{z_i}\#\mathit{sexp}$ *then* $\mathcal{P}(\sigma, \mathit{sexp}[\overline{\ell_i \mapsto z_i}])$ *is defined.*

Lemma A.1.2. *Let* $\{\overline{\ell_i \mapsto z_i}\} = \mathit{bindings}(\sigma_0, \mathit{sexp})$ *where* $\forall i \neq j.z_i \neq z_j$. *If* $\Sigma_0 = \mathit{parse}(\sigma_0, \mathit{sexp}[\overline{\ell_i \mapsto z_i}], \epsilon)$ *and* $\Sigma = \mathit{parse}(\sigma, \mathit{sexp}[\overline{\ell_i \mapsto z_i}].\ell, \ell) \sqsubseteq \Sigma_0$ *are defined and* $\forall \ell' \in \mathit{dom}(\Sigma_0).\mathit{resolve}(\Sigma_0, \Sigma_0(\ell'))$ *is defined, then* $\mathit{sexp}\{\overline{z_i/x_i}\}^{\sigma}_{\Sigma_1}$ *is defined, where* $\Sigma_1 = \{\overline{\ell' \mapsto \mathit{resolve}(\Sigma_0, \Sigma_0(\ell'))}\}$.

Proof. By induction on $parse(\sigma, sexp[\overline{\ell_i \mapsto z_i}].\ell, \ell))$.

Consider the case of recursive types:

$$\begin{array}{rcl} \Sigma & = & parse(\mu A.\sigma, sexp[\overline{\ell_i \mapsto z_i}].\ell, \ell) \\ & = & \{\ell \mapsto \mathcal{P}(\sigma[\mu A.\sigma/A], sexp[\overline{\ell_i \mapsto z_i}].\ell)\} \\ & = & \{\ell \mapsto \mathcal{P}(\sigma[\mu A.\sigma/A], sexp.\ell[\overline{\ell_j \mapsto z_j}].\epsilon)\} \end{array}$$

where $\{\overline{\ell_j \mapsto z_j}\} = \{\overline{\ell_i \mapsto z_i}\} \gg \ell$. From this we conclude that

$$\Sigma_0' = parse(\sigma[\mu A.\sigma/A], sexp.\ell[\overline{\ell_j \mapsto z_j}].\epsilon, \epsilon)$$

is defined and $\Sigma' = \Sigma_0' \sqsubseteq \Sigma_0'$ is defined and $\forall \ell' \in dom(\Sigma_0').resolve(\Sigma_0', \Sigma_0'(\ell'))$ is defined. Thus by the induction hypothesis:

$$\begin{array}{rl} & sexp.\ell[\overline{\ell_j \mapsto z_j}].\epsilon\{\overline{z_j/x_j}\}^{\sigma[\mu A.\sigma/A]}_{\Sigma_1'} \text{ is defined} \\ = & sexp[\overline{\ell_i \mapsto z_i}].\ell\{\overline{z_j/x_j}\}^{\sigma[\mu A.\sigma/A]}_{\Sigma_1} \\ = & sexp[\overline{\ell_i \mapsto z_i}].\ell\{\overline{z_j/x_j}\}^{\mu A.\sigma}_{\Sigma_1} \\ = & sexp[\overline{\ell_i \mapsto z_i}].\ell\{\overline{z_i/x_i}\}^{\mu A.\sigma}_{\Sigma_1} \end{array}$$

The last step is due to the fact that $\forall k \in \bar{i} - \bar{j}.z_k \notin bindings(\mu A.\sigma, sexp)$. □

Corollary. *If* $\{\overline{\ell_i \mapsto z_i}\} = bindings(\sigma, sexp)$ *where* $\forall i \neq j.z_i \neq z_j$ *and* $\Sigma = \mathcal{P}(\sigma, sexp)$ *is defined, then* $sexp\{\overline{z_i/x_i}\}^{\sigma}_{\Sigma}$ *is defined.*

Lemma (6.2.1)**.** *Let* $sexp$ *be parseable at* σ *and* $\{\overline{\ell_i \mapsto x_i}\} = bindings(\sigma, sexp)$. *If* $\overline{z_i}\#sexp$ *then*

$$sexp\{\overline{\ell_i \mapsto z_i}\}\{\overline{z_i/x_i}\}^{\sigma}_{\mathcal{P}\sigma sexp\{\overline{\ell_i \mapsto z_i}\}}$$

is defined.

Proof. Follows directly from the preceding two lemmas. □

A.2 Alpha-conversion preserves type

Lemma A.2.1 (Base variable renaming)**.** *Let* $x \in dom(\overline{\Gamma})$. *Then:*

$$\overline{\Gamma} \odot \overline{\Gamma}_0; \Pi\, ; \Sigma \vdash sexp : \sigma \iff \overline{\Gamma}\{z/x\} \odot \overline{\Gamma}_0; \Pi\, ; \{z/x\} \circ \Sigma \vdash sexp\{z/x\} : \sigma$$

Proof. By induction on the type derivation.

As a warm-up, let us consider the case of [F-MacDef], which has a nice fixed binding structure.

$$\begin{array}{ll}
 & (\Gamma :: \overline{\Gamma}) \odot \overline{\Gamma}_0; \bullet \vdash \textsf{(letrec-syntax (}y\ m\textsf{)}\ \mathit{form}\textsf{)} : \textsf{expr} \\
\Longleftrightarrow & \{\text{inversion of Rule [F-MacDef]}\} \\
 & ((\{y\!:\!\tau\} :: \Gamma) :: \overline{\Gamma}) \odot \overline{\Gamma}_0; \bullet \vdash \mathit{form} : \textsf{expr} \\
\wedge & ((\{y\!:\!\tau\} :: \Gamma) :: \overline{\Gamma}) \odot \overline{\Gamma}_0; \bullet \vdash m : \tau \\
\Longleftrightarrow & \{\text{induction hypothesis}\} \\
 & ((\{y\!:\!\tau\} :: \Gamma) :: \overline{\Gamma})\{z/x\} \odot \overline{\Gamma}_0; \bullet \vdash \mathit{form}\{z/x\} : \textsf{expr} \\
\wedge & ((\{y\!:\!\tau\} :: \Gamma) :: \overline{\Gamma})\{z/x\} \odot \overline{\Gamma}_0; \bullet \vdash m\{z/x\} : \tau \\
\Longleftrightarrow & \{\text{definition of renaming}\} \\
 & ((\{y\{z/x\}\!:\!\tau\} :: \Gamma\{z/x\}) :: \overline{\Gamma}\{z/x\}) \odot \overline{\Gamma}_0; \bullet \vdash \mathit{form}\{z/x\} : \textsf{expr} \\
\wedge & ((\{y\{z/x\}\!:\!\tau\} :: \Gamma\{z/x\}) :: \overline{\Gamma}\{z/x\}) \odot \overline{\Gamma}_0; \bullet \vdash m\{z/x\} : \tau \\
\Longleftrightarrow & \{\text{Rule [F-MacDef]}\} \\
 & (\Gamma :: \overline{\Gamma})\{z/x\} \odot \overline{\Gamma}_0; \bullet \vdash \textsf{(letrec-syntax (}y\ m\textsf{)}\ \mathit{form}\textsf{)}\{z/x\} : \textsf{expr}
\end{array}$$

A somewhat more interesting case is Rule [S-Import]:

$$\begin{array}{ll}
 & (\Gamma :: \overline{\Gamma}) \odot \overline{\Gamma}_0; \Pi; \Sigma \vdash \mathit{sexp} : \sigma{\downarrow}\beta \\
\Longleftrightarrow & \{\text{inversion of Rule [S-Import]}\} \\
 & ((\mathit{resolve}(\Sigma, \beta), \Gamma) :: \overline{\Gamma}) \odot \overline{\Gamma}_0; \Pi; \Sigma \vdash \mathit{sexp} : \sigma \\
\Longleftrightarrow & \{\text{induction hypothesis}\} \\
 & ((\mathit{resolve}(\Sigma, \beta), \Gamma) :: \overline{\Gamma})\{z/x\} \odot \overline{\Gamma}_0; \Pi; \{z/x\} \circ \Sigma \vdash \mathit{sexp}\{z/x\} : \sigma \\
\Longleftrightarrow & \{\text{distributivity}\} \\
 & ((\mathit{resolve}(\{z/x\} \circ \Sigma, \beta), \Gamma\{z/x\}) :: \overline{\Gamma}\{z/x\}) \odot \overline{\Gamma}_0 \cdots \\
\Longleftrightarrow & \{\text{Rule [S-Import]}\} \\
 & (\Gamma :: \overline{\Gamma})\{z/x\} \odot \overline{\Gamma}_0; \Pi; \{z/x\} \circ \Sigma \vdash \mathit{sexp}\{z/x\} : \sigma{\downarrow}\beta
\end{array}$$

The remainder of the proof is straightforward. □

For convenience, let ς range over sets of variable substitutions from a single rib, i.e.:

$$\varsigma ::= \{\overline{y_i/x_i}\} \text{ where } \forall i \neq i.y_i \neq y_j \wedge x_i \neq x_j$$

Lemma A.2.2 (Bindings-directed substitutions). *Let $\Sigma = \mathcal{P}(\sigma, \mathit{sexp})$ and $\Sigma' = \mathcal{P}(\sigma, \mathit{sexp}[\overline{\ell_i \mapsto z_i}])$, where $\{\overline{\ell_k \mapsto x_i}\} = \mathit{bindings}(\sigma, \mathit{sexp})$ and $\overline{z_i}\#\mathit{sexp}$. If $\mathrm{B} = \mathit{resolve}(\Sigma, \mathit{attr}) = \overline{\mathrm{P}_j}$ and $\mathrm{B}' = \mathit{resolve}(\Sigma', \mathit{attr}')$, where $\mathit{attr} = \mathit{attr}' = \beta$ or $\mathit{attr} = \Sigma(\ell)$ and $\mathit{attr}' = \Sigma'(\ell)$, then the following hold:*

1. $\{\overline{z_i/x_i}\}^{\mathrm{B}'} = \overline{\varsigma_j}$
2. $\mathrm{B}' = \overline{\mathrm{P}_j\varsigma_j}$
3. $\forall j.\varsigma_j = \{\overline{z_k/x_k}\}$ *for some permuted subset* $\{\overline{k}\} \subseteq \{\bar{i}\}$
4. $\forall i.x_i \notin \mathit{dom}(\mathrm{B}')$

Proof. By induction on the definition of $\mathit{resolve}(\Sigma', \mathit{attr}')$ and $\mathcal{P}(\sigma, \mathit{sexp})$. In the case of a rib $\mathit{attr} = \mathit{attr}' = \rho$, the rib signature contains some number of VAR members and some number of RIB members. The former must all resolve to distinct $\overline{x_k}$ or pattern variables, and the latter form an inductive case. The other cases are applications of the induction hypothesis. □

Lemma A.2.3 (Bindings-directed α-conversion). *Let* $\Sigma = \mathcal{P}(\sigma, \mathit{sexp})$ *and* $\Sigma' = \mathcal{P}(\sigma, \mathit{sexp}[\overline{\ell_i \mapsto z_i}])$ *where* $\{\overline{\ell_i \mapsto x_i}\} = \mathit{bindings}(\sigma, \mathit{sexp})$ *and* $\overline{z_i}\#\mathit{sexp}$. *Then*

$$\begin{aligned} & (\mathit{resolve}(\Sigma, \beta), \Gamma_0) :: \overline{\Gamma}; \Pi; \Sigma' \vdash \mathit{sexp}.\ell : \sigma \\ \iff \; & (\mathit{resolve}(\Sigma', \beta), \Gamma_0) :: \overline{\Gamma}; \Pi; \Sigma' \vdash \mathit{sexp}\{\overline{z_i/x_i}\}^{\mathit{resolve}(\Sigma,\beta)}.\ell : \sigma \end{aligned}$$

Proof. By Lemma A.2.2, we perform some number of rib substitutions, all the while enforcing the invariant that a growing prefix of the environment contains no occurrences of the bound base variables $\overline{x_i}$. At each rib, we perform some number of variable renamings, using Lemma A.2.1. □

Lemma (Type-directed α-conversion, 6.3.1). *If the following properties hold:*

- $\{\overline{\ell_i \mapsto x_i}\} = \mathit{bindings}(\sigma_0, \mathit{sexp})$
- $\Sigma = \mathcal{P}(\sigma_0, \mathit{sexp})$
- $\Sigma' = \Sigma[\overline{\ell_i \mapsto z_i}]$ *where* $\overline{z_i}\#\mathit{sexp}$
- $\Sigma : \Upsilon_0$
- $\ell \vdash \sigma : \Upsilon \sqsubseteq \sigma_0 : \Upsilon_0$

then:

$$\overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp}.\ell : \sigma \iff \overline{\Gamma}; \Pi; \Sigma' \vdash \mathit{sexp}[\overline{\ell_i \mapsto z_i}].\ell\{\overline{z_i/x_i}\}^{\sigma}_{\Sigma'} : \sigma$$

Proof. By induction on the type derivation. Let us consider the most interesting case, Rule [S-Import]:

$$\begin{aligned}
& \Gamma :: \overline{\Gamma}; \Pi; \Sigma \vdash \mathit{sexp}.\ell : \sigma{\downarrow}\beta \\
\iff & \{\text{inversion of Rule [S-Import]}\} \\
& (\mathit{resolve}(\Sigma, \beta), \Gamma) :: \overline{\Gamma} \odot \overline{\Gamma}_0; \Pi; \Sigma \vdash \mathit{sexp}.\ell : \sigma \\
\iff & \{\text{induction hypothesis}\} \\
& (\mathit{resolve}(\Sigma, \beta), \Gamma) :: \overline{\Gamma} \odot \overline{\Gamma}_0; \Pi; \Sigma' \vdash \mathit{sexp}[\overline{\ell_i \mapsto z_i}].\ell\{\overline{z_i/x_i}\}^{\sigma}_{\Sigma'} : \sigma \\
\iff & \{\text{Lemma A.2.3}\} \\
& (\mathit{resolve}(\Sigma', \beta), \Gamma) :: \overline{\Gamma} \odot \overline{\Gamma}_0; \Pi; \Sigma' \vdash \\
& \qquad \mathit{sexp}[\overline{\ell_i \mapsto z_i}].\ell\{\overline{z_i/x_i}\}^{\sigma}_{\Sigma'}\{\overline{z_i/x_i}\}^{\mathit{resolve}(\Sigma', \beta)} : \sigma \\
\iff & \{\text{definition of } \mathit{resolve}\} \\
& (\mathit{resolve}(\Sigma', \beta), \Gamma) :: \overline{\Gamma} \odot \overline{\Gamma}_0; \Pi; \Sigma' \vdash \mathit{sexp}[\overline{\ell_i \mapsto z_i}].\ell\{\overline{z_i/x_i}\}^{\sigma\downarrow\beta}_{\Sigma'} : \sigma \\
\iff & \{\text{Rule [S-Import]}\} \\
& \Gamma :: \overline{\Gamma} \odot \overline{\Gamma}_0; \Pi; \Sigma' \vdash \mathit{sexp}[\overline{\ell_i \mapsto z_i}].\ell\{\overline{z_i/x_i}\}^{\sigma\downarrow\beta}_{\Sigma'} : \sigma{\downarrow}\beta
\end{aligned}$$

The remaining cases are straightforward. □

A.3 Type soundness

Lemma (Transcription, 6.5.7)**.** *Let* $\mathrm{M} = \Sigma_u^{\mathcal{R}} \circ \Pi_d^{-1}$ *and* $\bullet \vdash \Sigma_u$ **ok**. *Given the following hygiene conditions:*

- $\mathit{dom}(\Gamma_d) \cap \mathbb{P} \subseteq \mathit{dom}(\Pi_d)$
- $\mathit{dom}(\Gamma_d) \cap \mathbb{B} \# \mu$
- $\mathit{fv}(\mathit{sexp})^{\sigma}_{\Sigma_d} \cap \bigcup_a \mathit{bv}(\mu(a))^{\Pi_d(a)}_{\Sigma_u} = \emptyset$
- $\mathit{bv}(\mathit{sexp})^{\sigma}_{\Sigma_d} \# \mu$

and a well-typed match:

$$\overline{\Gamma}_u; \bullet; \Sigma_u \vdash \mu : \Pi_d$$

then a well-typed macro template:

$$\Gamma_d :: \overline{\Gamma}_u; \Pi_d ; \Sigma_d \vdash \mathit{sexp} : \sigma$$

leads to a well-typed transcription:

$$\mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet\, ; \mathrm{M} \circ \Sigma_d \vdash \mu(sexp) : \sigma$$

Proof. By induction on the type derivation.

- Case [F-MacDef]: impossible, since $\Pi_d \neq \bullet$.
- Case [F-MacApp]: (already presented)
- Case [F-Var]: By assumption, $dom(\Gamma_d) \cap \mathbb{B} \# \mu$, so:

$$(\mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u)(x) = (\Gamma_d \odot \overline{\Gamma}_u)(x)$$

- Case [F-PExpr]:

$$
\begin{array}{ll}
 & \Gamma_d :: \overline{\Gamma}_u; \Pi_d; \Sigma_d \vdash a : \mathsf{expr} \\
\Longrightarrow & \{\text{Rule [F-P}\textsc{Expr}]\} \\
 & \Pi_d(a) <: \mathsf{expr}{\downarrow}\Pi_d^{-1}(\Gamma_d|_{\mathbb{P}}) \\
\Longrightarrow & \{\text{assumption}\} \\
 & \overline{\Gamma}_u; \bullet; \Sigma_u \vdash \mu(a) : \Pi_d(a) <: \mathsf{expr}{\downarrow}\Pi_d^{-1}(\Gamma_d|_{\mathbb{P}}) \\
\Longrightarrow & \{\text{subsumption}\} \\
 & \overline{\Gamma}_u; \bullet; \Sigma_u \vdash \mu(a) : \mathsf{expr}{\downarrow}\Pi_d^{-1}(\Gamma_d|_{\mathbb{P}}) \\
\Longrightarrow & \{\text{inversion of Rule [S-}\textsc{Import}]\} \\
 & \mathrm{M}(\Gamma_d|_{\mathbb{P}}) \odot \overline{\Gamma}_u; \bullet; \Sigma_u \vdash \mu(a) : \mathsf{expr} \\
\Longrightarrow & \{\text{inversion of Rule [S-}\textsc{Expr}]\} \\
 & \mathrm{M}(\Gamma_d|_{\mathbb{P}}) \odot \overline{\Gamma}_u; \bullet \vdash \mu(a) : \mathsf{expr} \\
\Longrightarrow & \{dom(\Gamma_d) \cap \mathbb{B} \# \mu\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet \vdash \mu(a) : \mathsf{expr} \\
\Longrightarrow & \{\text{Rule [F-}\textsc{Expr}]\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d \vdash \mu(a) : \mathsf{expr}
\end{array}
$$

- Case [F-PBVar]: (already presented)
- Case [M-Var]: similar to [F-Var].
- Case [M-Prim]: trivial.
- Case [M-Macro]: straightforward induction.

- Case [S-PVAR]:

$$\begin{array}{rl}
 & \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d \vdash \mu(a) : \Pi_d(a) <: \sigma{\downarrow}\Pi_d^{-1}(\Gamma_d|_{\mathbb{P}}) \\
\Longrightarrow & \{\text{subsumption}\} \\
 & \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d \vdash \mu(a) : \sigma{\downarrow}\Pi_d^{-1}(\Gamma_d|_{\mathbb{P}}) \\
\Longrightarrow & \{\text{inversion of [S-IMPORT]}\} \\
 & \Sigma_u^{\mathcal{R}}(\Pi_d^{-1}(\Gamma_d|_{\mathbb{P}})) \odot \overline{\Gamma}_u; \Pi_u; \Sigma'_d \vdash \mu(a) : \sigma \\
\Longrightarrow & \{dom(\Gamma_d) \cap \mathbb{B} \# \mu\} \\
 & \Gamma'_d; \Pi_u; \Sigma'_d \vdash \mu(a) : \sigma
\end{array}$$

- Case [S-EXPR]: straightforward induction.

- Case [S-IMPORT]: (already presented)

- Case [S-REC]:

$$\begin{array}{rl}
 & \Gamma_d :: \overline{\Gamma}_u; \Pi_d; \Sigma_d \vdash sexp : \mu A.\sigma \\
\Longrightarrow & \{\text{inversion of Rule [S-REC]}\} \\
 & \Gamma_d :: \overline{\Gamma}_u; \Pi_d; \mathcal{P}(\sigma', sexp) \vdash sexp : \sigma' \\
\wedge & \sigma' = \sigma[\mu A.\sigma/A] \\
\wedge & \Pi_d \vdash \mathcal{P}(\sigma', sexp)\ \mathbf{ok} \\
\Longrightarrow & \{\text{induction hypothesis}\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \mathcal{P}(\sigma', sexp) \vdash \mu(sexp) : \sigma' \\
\wedge & \sigma' = \sigma[\mu A.\sigma/A] \\
\wedge & \Pi_d \vdash \mathcal{P}(\sigma', sexp)\ \mathbf{ok} \\
\Longrightarrow & \{\text{Lemma 6.5.6}\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \mathcal{P}(\sigma', sexp) \vdash \mu(sexp) : \sigma' \\
\wedge & \sigma' = \sigma[\mu A.\sigma/A] \\
\wedge & \bullet \vdash \mathrm{M} \circ \mathcal{P}(\sigma', sexp)\ \mathbf{ok} \\
\Longrightarrow & \{\text{Lemma 6.5.5}\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathcal{P}(\sigma', \mu(sexp)) \vdash \mu(sexp) : \sigma' \\
\wedge & \sigma' = \sigma[\mu A.\sigma/A] \\
\wedge & \bullet \vdash \mathcal{P}(\sigma', \mu(sexp))\ \mathbf{ok} \\
\Longrightarrow & \{\text{Rule [S-REC]}\} \\
 & \mathrm{M}(\Gamma_d) \odot \overline{\Gamma}_u; \bullet; \mathrm{M} \circ \Sigma_d \vdash sexp : \mu A.\sigma
\end{array}$$

- Case [S-UNION]: similar.

□

Index of Formal Notation

Index of Formal Notation

D	right tree address projection	18
$\Sigma^{\mathcal{R}}$	resolution of bindings (curried)	48
$\overline{\Gamma}; \Pi \vdash \mathit{form} : \mathsf{expr}$	form type judgment	72
$\overline{\Gamma}; \Pi \vdash \mathit{mexp} : \sigma \to \mathsf{expr}$	macro type judgment	72
Υ	abstract bindings table	78, 79
$\Upsilon_1 \sqsubseteq \Upsilon_2$	abstract bindings table approximation	86
$\mathbb{D}^*$	universe of tree addresses	18
$=_\alpha$	α-equivalence	56
$\ell @ \gamma$	append-environment signature	26
attr	binding attribute	44
$\mathit{bindings}(\sigma, \mathit{sexp})$	base variable bindings	55
$\mathit{bp}(\sigma)$	binding positions	53
bv	bound base variables	52
$\mathit{bv}(\mathit{sexp})^\sigma_\Sigma$	type-directed bound base variables	52
bvar	variable binding type	26
$\rho :: \gamma$	pair	xi, 26
data	quoted data type	26
δ	binding type	26
$\mathbb{D}$	universe of tree address projections	18
$\lfloor - \rfloor$	variable name extraction	105
ℓ	tree address	18
Γ	environment	47
ENV	environment bindings type	26
γ	environment binding signature	26
$\sigma{\uparrow}\beta$	export type	26
expr	expression type	26
fv	free base variables	51
$\mathit{fv}(\mathit{sexp})^\sigma_\Sigma$	type-directed free base variables	51
$\gg$	partial inverse of $\ll$	133
$\hat{\sigma}$	import normal form	31

Σ	bindings table	44
$\sigma{\downarrow}\beta$	import type	26
ι	identity function	xi
$\Upsilon \vdash_{\downarrow} \sigma\ \mathbf{ok}$	import well-formedness	79
$\ell \vdash \sigma : \Upsilon \sqsubseteq \sigma_0 : \Upsilon_0$	generalized well-formedness	86
$\Pi \vdash \mathrm{B}\ \mathbf{ok}$	well-formed bindings	74
$\Pi \vdash \Sigma\ \mathbf{ok}$	well-formed bindings table	74
$\ell \vdash \hat{\sigma}_1 <: \hat{\sigma}_2$	subtyping judgment	35
$\Upsilon \vdash \mathrm{B} : \delta$	well-typed bindings	84
$\Upsilon \vdash \beta : \delta$	well-formed binding signature	80
$\ell \vdash_{\uparrow} \sigma : \Upsilon$	export well-formedness	79
$\ll$	region displacement	34
$\mathit{Variable}$	universe of variables	42
$\mathit{actuals}$	input type of apply	43
$\mathit{adj}(\mathrm{B})$	pattern variable adjacency matrix	71
$\mathit{adj}(\sigma)$	tree address adjacency matrix	71
$\mathit{binding}(\mathit{sym}, \Pi)$	pattern variable elaboration	106
$\mathit{bind}(\sigma, \mathit{src}, \Pi)$	partial S-expression elaboration	106
data	quoted data	43
expr	fully expanded expression	41
$\mathit{formals}$	input type of lambda	43
form	form	43
$\mathit{form} \longmapsto\!\!\!\!\twoheadrightarrow_{\varepsilon} \mathit{expr}$	complete expansion	98
$\mathit{match}(p, \mathit{sexp})$	macro pattern matching	64
mexp	macro expression	43
prim	primitive syntax operator	43
sexp	parsed S-expression	43
$\mathit{sexp}[m/x]$	macro substitution	63
$\mathit{sexp}[m/x]^{\sigma}_{\Sigma}$	type-directed macro substitution	63
$\mathit{sexp}\{\overline{z/x}\}^{\sigma}_{\Sigma}$	type-directed α-renaming	58

$sexp\{y/x\}$	uniform variable substitution	54
src	unparsed source S-expression	41
$supp$	support	55
sym	quoted symbol	43
$tree$	syntax tree	23
$tree.\ell$	tree address projection	24
var	variable	43
μ	macro pattern substitution	64
ε	empty sequence	xi, 26
NONE	null binding type	26
$\bowtie$	shape overlap relation	80
$\texttt{(}mexp\ \texttt{.}\ sexp\,\texttt{:}\,\sigma\texttt{)}$	macro application	43
$\texttt{(}tree\ \texttt{.}\ tree\texttt{)}$	syntax pair	23
$parse(\sigma, sexp, \ell)$	S-expression parsing	46
$penv$	macro pattern environment	67
$penv(a)$	pattern environment lookup	68
$penv^{-1}$	pattern environment inversion	69
$\preceq$	address prefix relation	24
$\prec$	strict address prefix relation	24
apply	primitive application operator	43
lambda	primitive binding operator	43
letrec-syntax	primitive macro binding operator	43
quote	primitive quotation operator	43
syntax-rules	primitive macro operator	43
$ptype(p, \hat{\sigma}, a)$	pattern type lookup	68
$\mathbb{P}$	universe of pattern variables	42
$\mu A.\sigma$	recursive type	26
$resolve(\Sigma, attr)$	resolution of bindings	49
P	rib	47
RIB	rib bindings type	26

ρ	rib binding signature	26
$(\overline{\mathit{term}})$	S-expression sequence	xi
$(\overline{\mathit{tree}})$	syntax list	23
$()$	null syntax tree	23
$\Box, \triangle$	syntactic shape	80
$\mathit{shape}(\sigma)$	shape extraction	80
β	binding signature	26
σ	syntax type	26
$\odot$	environment stack pasting relation	89
r	marked redex	101
$\longmapsto_{\varepsilon}$	macro expansion relation	65
$\longmapsto_{\alpha}$	α-conversion expansion relation	65
$\longmapsto_{\mathsf{return}}$	macro return expansion relation	62
$\longmapsto\!\!\!\!\twoheadrightarrow_{\varepsilon}$	multi-step macro expansion relation	65
$\longmapsto_{\mathsf{subst}}$	macro substitution expansion relation	62
$\longmapsto_{\mathsf{trans}}$	macro transcription expansion relation	62
$<:$	subtyping relation	33
τ	form type	26
$\mathcal{T}(a, \delta)$	pattern variable binding	46
$\top$	top shape	80
$\cup\{\overline{\sigma}\}$	union type	26
V	variable binding	47
$\textsc{var}$	variable binding type	26
ς	rib variable substitutions	136
$\mathbb{B}$	universe of base variables	42
$\overline{x}$	sequence	xi
$\mathit{wf}(\mathit{penv})$	well-formed pattern environment	67
$\mathit{wf}(p)$	well-formed pattern	69
φ	marked-redex reduction	101
$\mathrm{B} : \delta$	bindings type	83

$\Sigma : \Upsilon$	well-typed bindings table	84
$\sigma : \delta$	well-formed type	79
$\{\ell\}(\prec)$	set of prefixed addresses	49
$\{\overline{z/x}\}^{\mathrm{B}}$	bindings-directed α-renaming	58
a, b	pattern variable	43
$f(x) \Downarrow$	f is defined at x	xi
m	macro	43
p	macro pattern	43
p^{-1}	pattern inversion	69
r	pre-redex	97
$x \mapsto y$	mapping	xi
x, y	base variable	42