Fully-parameterized, First-class Modules with Hygienic Macros

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Josef Martin Gasbichler
aus Biberach/Riß

Tübingen
2006

Tag der mündlichen Qualifikation:	15. 02. 2006
Dekan:	Prof. Dr. Michael Diehl
1. Berichterstatter:	Prof. Dr. Herbert Klaeren
2. Berichterstatter:	Prof. Dr. Peter Thiemann (Universität Freiburg)

Abstract

It is possible to define a formal semantics for configuration, elaboration, linking, and evaluation of fully-parameterized first-class modules with hygienic macros, independent compilation, and code sharing. This dissertation defines such a semantics making use of explicit substitution to formalize hygienic expansion and linking. In the module system, interfaces define the static semantics of modules and include the definitions of exported macros. This enables full parameterization and independent compilation of modules even in the presence of macros. Thus modules are truly exchangeable components of the program. The basis for the module system is an operational semantics for hygienic macro expansion—computational macros as well as rewriting-based macros. The macro semantics provides deep insight into the nature of hygienic macro expansion through the use of explicit substitutions instead of conventional renaming techniques. The semantics also includes the formal description of Macro Scheme, the meta-language used for evaluating computational macros.

Zusammenfassung

Es ist möglich, eine formale Semantik anzugeben, welche die Phasen Konfiguration, syntaktische Analyse mit Makroexpansion, Linken und Auswertung für ein vollparametrisiertes Modulsystem mit Modulen als Werten erster Klasse, unabhängiger Übersetzung und Code-Sharing beschreibt. Diese Dissertation beschreibt eine solche Semantik. Dabei formalisieren explizite Substitutionen die hygienische Makroexpansion und das Linken. Im Modulsystem beschreiben Schnittstellen die statische Semantik von Modulen und enthalten die Definitionen der exportierten Makros. Dies ermöglicht volle Parametrisierung und unabhängige Übersetzung sogar in Kombination mit Makros. Module sind damit echte austauschbare Komponenten eines Programms. Die Grundlage für das Modulsystem bildet eine operationelle Semantik für hygienische Makroexpansion die berechnende Makros ebenso beschreibt wie regelbasierte Makros. Durch die Verwendung expliziter Substitutionen anstelle konventioneller Umbenennung gibt die Semantik für Makroexpansion tiefe Einblicke in das Wesen hygienischer Makroexpansion. Die Semantik beschreibt außerdem Makro Scheme, die Metasprache für die berechnenden Makros.

Contents

List of Figures

Acknowledgments

I'm an apprentice of Dr. S^2. I did it. I did it all, by myself. Well, almost.

First of all, I thank Prof. Herbert Klaeren, my advisor. He allowed me great latitude in my choice of research activities, provided a pleasant environment, agreed with this dissertation project, and urged me to finish it when it was about time.

I am also indebted to Prof. Peter Thiemann, my co-advisor, for providing profound comments on the entire text. His remarks helped me to see many things from a broader point of view. The dissertation really benefited a lot from him.

Mike Sperber introduced me to world of function programming and guided me on my initial steps into research and teaching. Mike pointed me to Richard Kelsey's work on module systems from which this project started. Later, he read through various drafts of this dissertation, provided numerous corrections and suggestions and marked the places where more illustration was necessary. I am deeply grateful for the time and effort he spent sharing his extensive knowledge with me.

My colleague Holger Gast was always available for discussing various aspects of the semantics. Holger also wrote TEX macros to split the generated rules automatically into several lines if they exceed the page width. I am still trying to understand this code.

I really appreciate the time and work the authors of PLT redex put into developing this tool. PLT redex has been a great help.

Finally, I thank my wife Christl for her love, encouragement, and comfort. "Moritz auch."

Chapter 1

Introduction

It is possible to define a formal semantics for configuration, elaboration, linking, and evaluation of fully-parameterized first-class modules with hygienic macros, independent compilation, and code sharing.

Module systems are an essential part of modern programming languages. They serve various purposes, among them name spaces and units of compilation. Perhaps most importantly, module systems are means of abstraction: Interfaces specify abstractions and modules provide implementations. To fulfill these roles, a module system must provide a number of features:

Full parameterization enables the programmer to choose and exchange the imports of a module at any time during development without modifying the module definition or relying on external programs to assemble the program. With full parameterization it is also possible to link several instantiations of the same module against different providers within the same program.

First-class modules extend the program at run time, define local name spaces, and enable programmable linking. In addition, if first-class modules and top-level modules resemble each other, the semantics for first-class modules can explain the run-time linking, evaluation, and access of top-level modules.

Independent compilation permits the development of modules even if imported modules are not available yet. Thus they ease the management of large software projects.

Code sharing reduces the size of object code by pooling the code of modules with identical static semantics.

Syntactic abstractions are the realm of hygienic macros, a well-established and reliable vehicle for extending the syntax of a language. Unfortunately, existing systems that combine modules and macros break the abstractions procured by modules. In these systems, the export description of a module contains the names of exported macros only, hence the compiler needs to access the bodies of the imported modules to retrieve the macro definitions. Thus, to support macros, such systems sacrifice independent compilation and the ability to parameterize over modules providing macros. This dissertation shows how to regain these features by moving the declaration of exported macros to the interfaces of modules. In the resulting module system, modules are again exchangeable components of a program but with full support for hygienic macros.

Taken together, fully-parameterized modules, first-class modules and macro declarations within interfaces are essential to make modules exchangeable components of a program. This dissertation provides the semantics of a module system and a hygienic macro expander with these features. For implementors and programmers alike, the semantics defines the behavior of expansion, linking, and evaluation in a precise manner.

1.1 Combining Modules and Macros

This section gives an example that demonstrates the combination of fully-parameterized modules and macros. The example deals with the implementation of the control operators `shift` and `reset` [DF90]. The operators define *composable continuations*, where `reset` limits the extend of such a continuation, and `shift` reifies the current continuation up to the next enclosing `reset` as a function. Originally, `shift` and `reset` were implemented using a CPS transformation and hence within the compiler. In [Fil94] Filinski shows how to implement `shift` and `reset` using the `call-with-current-continuation` operator and assignment. Danvy translated Filinski's SML implementation to Scheme and contributed it to the Scheme 48 distribution [KR95].

To ease the usage, Danvy implemented `shift` and `reset` as macros:

```
(define-syntax reset
  (syntax-rules ()
    ((reset e) (*reset (lambda () e)))))

(define-syntax shift
  (syntax-rules ()
    ((shift k e) (*shift (lambda (k) e)))))
```

The macros `reset` and `shift` exhibit two typical uses of macros: `reset` puts its argument under a nullary λ-abstraction (called a *thunk*) to defer evaluation, and `shift` establishes a new binding form to bind the composable continuation to the variable `k` in the expression `e`. The actual implementation of the control operators is left to `*reset` and `*shift`, which are ordinary procedures.

To package the implementation of `shift` and `reset` into a module, it is necessary to define the interface of such a module. `Shift` and `reset` are macros and, as mentioned above, in our system interfaces contain the definitions of exported macros. Thus the following code defines the interface `shift-reset-interface` that exports the two macros:

```
(define-interface shift-reset-interface
  (export
    (define-syntax shift
      (syntax-rules ()
        ((shift k e) (*shift (lambda (k) e)))))

    (define-syntax reset
      (syntax-rules ()
        ((reset e) (*reset (lambda () e)))))))
```

A module implementing this interface must provide definitions of the `shift*` and `reset*` procedures. The following `define-module` form defines such an implementation:

```
(define-module filinski-shift-reset shift-reset-interface
  (open scheme-interface)

  (begin
    (define (*shift f)
      (call-with-current-continuation
        (lambda (k)
          (*abort (lambda ()
                    (f (lambda (v)
                         (reset (k v)))))))))

    (define (*reset thunk) ...)

    ...)
```

In the module definition, `filinski-shift-reset` is the name of the module, `shift-reset-interface` the name of the export interface, the `open` clause describes the module's imports (here `scheme-interface`, a built-in interface with the definitions from R^{5}RS), and the `begin` clause contains the actual code by Danvy. This code is only sketched here and it is sufficient to see that it contains definitions of `*shift` and `*reset`.

Modules are fully-parameterized and hence the imports of a module are interfaces. A module using the implementation of `shift` and `reset` only lists the above interface in its `open` clause. A hypothetical module definition for a partial evaluator using `shift` and `reset` could thus look like this:

```
(define-module partial-evaluator partial-evaluator-interface
  (open scheme-interface
        shift-reset-interface)
  (begin
    ... reset ... shift ...))
```

Here, the module name is `partial-evaluator`, its export interface is some `partial-evaluator-interface`, the imports are `scheme-interface` and `shift-reset-interface`, and the code contains applications of the `shift` and `reset` macros.

To evaluate such a module, it is necessary to link it against actual implementations of the imported interface. The decision which modules to use is up to the linker, possibly guided by annotations from in the configuration language.

Filinski's implementation of `shift` and `reset` is faster than the implementation using a CPS transformation. However, it is still not optimal because it relies on `call-with-current-continuation`, which needs to reify the *entire* continuation of an expression. As an improvement, Sperber together with the author present a *direct* implementation of the control operators [GS02]. The implementation relies on the representation of the continuation as linked stack frames and uses *slices* of the stack to represent composable continuations. This approach yields significant overall performance gains for typical applications of `shift` and `reset` such as partial evaluators. The interface of the enhanced version is still `shift-reset-interface`:

```
(define-module direct-shift-reset shift-reset-interface
  (open scheme-interface
        vm-primitives-interface)

  (begin
    (define (*shift f)
      (let* ((slice (create-stack-slice))
             (c (lambda vs
                  (copy-slice-to-stack slice)
                  (apply values vs))))
        (f c)))

    (define (*reset thunk)
      ...)))
```

This implementation of `*shift` and `*reset` differs significantly from the one in `filinski-shift-reset` as the procedures now manipulate stack frames. In the sketched example, `vm-primitives-interface` describes the exports of the virtual machine that provides direct access to the stack such as `create-stack-slice`.

As both modules, `filinski-shift-reset` and `direct-shift-reset`, implement the same interface, it is possible to link `partial-evaluator` against both of them. However, it is not necessary to re-compile `partial-evaluator` for it—even though the modules export macros—because the interface contains the macro definitions. Even run-time linking is possible for such modules. Further, the implementor of `partial-evaluator` module need not be aware of the

various implementations of the `shift-reset-interface`. She can implement and compile the module without knowledge of the implementations. As the module system is fully-parameterized, it is not even necessary to take any precautions for making the import exchangeable: A fully-parameterized module specifies all imports by their interfaces only and thus the corresponding modules can be exchanged by the linker.

This dissertation defines the semantics for a module system as presented in this section and explains all aspects of such a powerful system formally.

1.2 Representation of Identifiers

The semantics uses an advanced representation for identifiers based on de Bruijn indices. Like de Bruijn indices, our representation describes the λ-abstraction of a locally bound variable using a natural number counting the binders that separate a variable occurrence and its binder. The representation is an essential means for describing hygienic macro expansion. As an example for the importance of a powerful representation of identifiers, consider this (simplified) version of the `or` macro from R^{5}RS:

```
(define-syntax or
  (syntax-rules ()
    ((or a b)
     (let ((temp a))
       (if temp temp b)))))
```

The macro accepts two arguments, `a` and `b`, and expands into an expression that evaluates `a`, binds the result to the variable `temp` and uses `if` to return the value of `a` if it is not false and evaluates and returns `b` otherwise. Now consider this expression using `or`:

```
(lambda (temp)
  (or (+ 34 8) temp))
```

Hygiene requires the variable `temp` in the second argument of `or` to refer to the surrounding λ-abstraction, independent of the bindings introduced in the output of `or`. Hence the following naive expansion violates hygiene:

```
(lambda (temp)
  (let ((temp (+ 34 8)))
    (if temp temp temp)))
```

Here, the third operand of `if` does not refer to the λ-abstraction but instead to the `let`—a violation of hygiene. A common technique to avoid this problem, is to introduce new, uniquely generated names for the bound identifiers. However, this also affects the list of bound names of a λ-abstraction and renaming is not purely functional as some kind of state is necessary to guarantee generation of unique names. If we attach de Bruijn indices [Bru72] to variable occurrences and use the indices to describe the binding place of a variables (thus neglecting the name of the variable), the expansion of the above expression yields:

```
(lambda (temp)
  (let ((temp (+ 34 8)))
    (if temp^0 temp^0 temp^1)))
```

Now the third argument of `or` is bound by the λ-abstraction as required by hygiene. No renaming or name generation is necessary because the de Bruijn index uniquely determines the binding place of an identifier.

To cover identifier occurrences in module bodies and support advanced hygienic macro systems, this dissertation extends de Bruijn indices to other dimensions:

- For identifiers imported by modules, an *import path* denotes a path to the exporting module. The import path mentions only the interfaces that were imported to access the identifier, hence it is compatible with independent compilation.
- A *set of marks* describes the macro expansion that introduced an identifier. The set of marks is essential during macro expansion to maintain hygiene but it is irrelevant during evaluation.
- A *position* distinguishes variables bound by the same λ-abstraction. However, dealing with positions is straightforward and hence omitted in the formal semantics for the sake of simplicity.
- The *name* of the identifier helps to compare possibly unbound identifiers, which is important to describe syntactic extensions with literal identifiers.

The resulting representation of identifiers is powerful enough to support hygienic macro expansion and also to support modules exporting and importing macros from other modules through interfaces.

1.3 Explicit Substitutions

The semantics in this dissertation strongly build on *explicit substitutions* [ACCL91] and this section briefly summarizes the use of explicit substitutions within this dissertation.

An explicit substitution replaces a variable by a term but, unlike a meta-substitution, the explicit substitution is itself part of a term. Then special reduction rules propagate the explicit substitution from the top of a term to the base terms. If such a base term is a variable, the rules reduce the substitution to the substituting term or discard the substitution if it is not applicable. This process is called *elimination* and the special rules are called *elimination rules*. The semantics in this dissertation uses explicit substitutions for two different purposes. First, during the evaluation of expressions and during linking of modules explicit substitutions replace identifiers by values. These explicit substitutions are called *evaluation substitutions*. Second, for parsing and macro expansion, five different kinds of explicit substitutions replace source code symbols, identifiers, and meta-variables by syntactic terms. The dissertation refers to these explicit substitutions altogether as *expansion substitutions*.

The motivation for using evaluation substitutions is that they model evaluation on a real machine more closely than meta-substitutions. In particular, a meta-substitution applied to a term immediately creates a new term by replacing in the original term all occurrences of the variable that the substitution replaces. If a meta-substitution models function application, this means that the code of the function's body will be replaced by a copy of the code in which the bound variable has been replaced by the argument. This model bears no resemblance to real implementations where some form of environment maps variables to values and the code itself remains unchanged throughout evaluation. An explicit substitution models such an environment by attaching the binding information to a term without modifying the term otherwise. Only if the substitution arrives during evaluation and elimination at a variable, the term from the substitution replaces the variable by another term. This corresponds to a lookup of the variable in the environment.

In addition, explicit substitutions can also describe linking of modules in strong resemblance to real implementations. The reason why explicit substitutions are more suitable than meta-substitutions is analogous to function applications: Linking connects the imported variables in the body of a module to values exported by other modules. If a meta-substitution models linking, it replaces the code of the module body by a copy, where the imported variables have been replaced by values. An explicit substitution modeling linking, however, will leave the module body intact. Hence, this dissertation uses explicit substitution to model function application and more importantly to model linking of modules. The semantics for linking incrementally builds from the values of imported modules an explicit substitution as the environment for the

module body. Evaluation and elimination move the substitution to the imported variables which can then evaluate to the respective values. The semantics can then form the foundation of an implementation of a powerful module system.

The motivation for using expansion substitutions is the possibility to attach these explicit substitutions to terms without eliminating them directly. This is important for hygienic macro expansion because there expansion and parsing are interwoven and therefore the macro expander cannot work on abstract syntax because the parser has not generated it yet. Instead, the macro expander and the parser generate explicit substitutions to record their results and eliminate the substitutions only gradually as parsing and expansion advance. With this technique, the substitutions work only on the abstract syntax and hence it is always possible to comprehend the correctness of the expansion process. The "lazy" propagation of results and the representation of the expanded terms as a mixture of abstract and concrete syntax are essential ingredients to describe hygienic macro expansion and they make the descriptions in this dissertation a real semantics for hygienic macro expansion, rather than a bare algorithm.

1.4 Structure of the Dissertation

This dissertation has the following structure: Chapter 2 introduces fully-parameterized module systems and the concept of including macro declarations in interfaces. Chapter 3 first defines λ_v^n, a variant of the λ-calculus that builds on Λ^n, a language with explicit substitutions and de Bruijn indices. Afterwards, the chapter presents the semantics of a hygienic macro expander and two macro transformer facilities. The first transformer implements computational macros and uses the language Macro Scheme for evaluation of transformer procedures. The second transformer is a slightly simplified variant of `syntax-rules`. Chapter 4 defines $\Lambda_n^{\text{Module}}$, a language with fully-parameterized modules, again based on a formal semantics. It also shows the combination of modules and macros that yields independent compilation. Chapter 5 describes the prototype implementations of the macro expander and the module systems, including the way I have generated the formulas in this dissertation. Chapter 6 reviews related work and Chapter 7 summarizes the results. Finally, Appendix A includes the mathematical notation of the dissertation.

Chapter 2

Fully-Parameterized Module Systems

For modularity, interfaces and modules play central roles: An interface is a specification of an abstraction, while a module is an implementation of a specification. Such an implementation may require or *import* other modules to accomplish its task. Again, interfaces specify abstractions over these required implementations [LB88]. Consequently, modules are replaceable components of a program and interfaces specify the junctions where replacement may take place. A separate resolution phase, called *linking*, maps the required interfaces to imported modules. The programmer may have to provide additional information to instruct the linker which module it should use to implement a required interface.

Conventional module systems typically support only a weaker form of modularity where modules import other modules directly instead of abstracting over them by means of interfaces. Some of the conventional module systems—most notably the systems of the languages from the ML family—support as a variant modules that abstract over a single imported module. These modules are then said to be *parameterized* and are also called *functors* in the ML terminology. The programmer usually performs linking explicitly by providing the implementation of the interface.

The *units* system from PLT Scheme [FF98] is a module system where every module is abstracts over all the modules it imports. In analogy to the terminology above, units are *fully-parameterized*. Again, the linker for a fully-parameterized module needs information about which module should be used to satisfy the imported interface. However, in the setup of a fully-parameterized module system, it is a tedious task for the programmer to provide this specification: as all imports are given as interfaces, a declaration for every import relation is required.

As observed by Kelsey [Kel97], most of these link declarations can easily be inferred automatically in realistic programs simply because there is often only one module implementing a specific interface. Only when several modules implement the same interface does the linker need the aid of the programmer. Wiesenmaier, for his Master's thesis [Wie00], implemented a prototype of Kelsey's fully-parameterized module system where link declarations are inferred automatically whenever possible.

Parameterization is important to support the concept of a module as a replaceable component of a program: A module provides an interface to the outside world and has to be exchangeable with any other module providing the same interface. A module system with full parameterization and automatic linking permits the programmer to replace every module within a program at little cost.

2.1 Fully-Parameterized Module Systems and Scheme

For a language with a powerful macro facility such as Scheme, it is desirable to export and import macros as well as regular run-time bindings. However, listing the macro name in the interface

is not enough to enable independent compilation: Macro expansion is usually interleaved with parsing and therefore takes place in the first phase of a compiler. For compilation to succeed, the compiler needs access to the definition of the imported macros. Scheme implementations usually solve this by sacrificing independent compilation and falling back to *separate* compilation: During macro expansion the compiler looks up the definitions of the imported macros in the source code of the imported modules [Fla04, KR02, Dyb98].

For a module system with fully-parameterized modules, the imports are only interfaces and the source code of the corresponding imported modules is only known after linking. Wiesenmaier therefore includes a static linking phase that resolves the parameterization. However, this renders full parameterization an exclusively configuration-phase facility and hides much of the power and expressiveness of a fully-parameterized module system: Independent compilation and programmer-controlled linking of higher-order modules are no longer possible.

This dissertation chooses a different alternative that supports independent compilation in the presence of macros: The interfaces do not only contain the name but also the definition of exported macros. This makes it possible to macro expand the source code of a module by only looking at the interfaces it imports. While this approach looks unfamiliar at first glance, it is necessary to support independent compilation in the presence of macros. Note, that it also subsumes the conventional approach: The macro definitions in the interfaces can be derived automatically from the source code after a preprocessing phase.

The next section informally introduces fully-parameterized modules by a series of examples. Section 2.3 presents the terminology used in this dissertation, Section 2.4 gives an initial overview of the various phases involved during evaluation of a program with modules. Section 2.5 recapitulates Kelsey's work and adds some features missing in his draft. Wiesenmaier's work is the topic of Section 2.6. Section 2.7 presents the configuration language for our module system with fully-parameterized modules and Section 2.8 defines the semantics of this language by transforming it into Kelsey's abstract data-types.

2.2 The Programmer's Point of View

A Scheme programmer writes the source of the modules and declarations of interfaces, modules, and programs. The language of the source code is plain Scheme and the code may reference identifiers defined outside the source code. Section 2.7 formally presents the syntax for the language in which the programmer defines interfaces, modules, and programs—this section gives an informal overview.

The task of an interface is to enumerate the identifiers exported by a module. The keyword `define-interface` introduces such a definition by giving it a name and an interface declaration. Usually, the declaration is an `export` form which in turn contains the actual list of identifiers to be exported. For example, the following form defines an interface named `foo`, which exports the variables `a` and `b`:

```
(define-interface foo-interface
  (export a b))
```

A module contains a piece of source code, along with a module name, an export interface, and a list of imports. All modules are fully-parameterized, therefore the programmer lists the names of interfaces as imports of a module. The keyword `define-module` defines a module. For example, to define a module `foo` with export interface `foo-interface`, imported interfaces `scheme-interface` and `bar-interface`, and source file `foo.scm`, this declaration is necessary:

```
(define-module foo foo-interface
  (open scheme-interface
        bar-interface)
  (files foo))
```

The definition starts with the module name and the name of the export interface. Afterwards, various clauses may follow. Usually this is just an `open` clause which lists the name of the imported interfaces and a `file` clause which references a file containing the source code is present. The interface `scheme-interface` and a module `scheme` with this interface are built into the system. This module `scheme` provides the definitions of the Scheme standard.

Here, the argument of `files` it name `foo` which stands for the file `foo.scm` in the same directory.

To complete the example, the following forms add definitions for the interface `bar-interface` with variables `c` and `d` and for the module `bar` with export interface `bar-interface`, import `scheme-interface`, and source file `bar.scm`:

```
(define-interface bar-interface
  (export c d))

(define-module bar bar-interface
  (open scheme-interface)
  (files bar))
```

Finally, the programmer needs to define a program which links the imports of the modules with other modules providing these imports. The form `define-program` declares such a definition. Like the other forms, its first argument is the name to be defined. Afterwards, program clauses may follow. For the example, a `modules` clause which lists the names of the modules that are part of the program is sufficient:

```
(define-program p1
  (modules foo bar scheme))
```

This defines a program `p1` with modules `foo` and `bar`. This program does not contain explicit links between the imports and their implementation. Instead, the system uses an algorithm for linking. That algorithm links the imports of the modules to another module if there is only one module in the program that provides the imported interface. Otherwise, the algorithms does not link the import. In the example above, each import can be linked by the linking algorithm. But if the following module `bar2` is added to the program, the algorithm fails:

```
(define-module bar2 bar-interface
   (open scheme-interface)
   (files bar2))

(define-program p2
   (modules foo bar bar2 scheme))
```

The program `p2` contains two modules which export or *implement* the `bar-interface`. Consequently, the linking algorithm cannot link the import `bar-interface` of the module `foo` as it is not uniquely determined.. To fully link `p2`, the programmer has to add a `link` clause to the program that defines, which module should be linked to the `bar-interface` of `foo`:

```
(define-program p2
   (modules foo bar bar2 scheme)
   (link (foo bar-interface bar2)))
```

Here, `p2` chooses `bar2` as the provider of `bar-interface` for module `foo`.

The `define-program` form also supports a `programs` clause that imports other programs into the program. In this case, the input programs correspond to "libraries," that is, related pieces of software components. The new program includes the modules of the input programs and combines their link environments with the link environment of the importing program. However, to enable this combination, the link environments must be consistent: For each module contained in both programs, the link environments must link each import to the same implementation, or at least

one of the two must lack a link for the import. The arguments of `programs` is the list of programs that serve as input. In the following example, the programs `lexer-program` and `parser-program` add different links for the import `hash-table-interface` of the module `symbol-table`. Hence their combination in the program `compiler-program` is an error, as the link environments are not consistent:

```
(define-module hash-table hash-table-interface
   (open scheme-interface)
   (files hash-table))

(define-module vector-hash-table hash-table-interface
   (open scheme-interface)
   (files vector-hash-table))

(define-module symbol-table symbol-table-interface
   (open scheme-interface
         hash-table-interface)
   (files symbol-table))

(define-module parser parser-interface
   (open scheme-interface
         symbol-table-interface)
   (files parser))

(define-program parser-program
   (modules parser symbol-table hash-table)
   (link (symbol-table hash-table-interface hash-table)))

(define-module lexer lexer-interface
   (open scheme-interface
         symbol-table-interface))

(define-program lexer-program
   (modules lexer symbol-table vector-hash-table)
   (link (symbol-table hash-table-interface vector-hash-table)))

(define-module driver driver-interface
   (open scheme-interface
         parser-interface
         lexer-interface)
   (files driver))

(define-program compiler-program
   (modules driver)
   (programs parser-program lexer-program)) ;; ERROR!
```

In such a case, the programmer can resolve the conflict in one of two ways: either she modifies the `link` clause of one of the programs to use the same implementation in both programs or she creates a new implementation from the existing module `symbol-table`. For the latter case, a variant of `define-module` exists:

```
(define-module lexer-symbol-table (copy-module symbol-table))
```

This form creates a copy of a module and assigns a new name to the copy. Now the programmer can use the copy within the `lexer-program` and the link environment for `compiler-program` will be consistent:

```
(define-program lexer-program
   (modules lexer-symbol-table symbol-table vector-hash-table)
   (link (lexer-symbol-table hash-table-interface vector-hash-table)))
```

The combination of `copy-module` and `link` is quite common for modules that are only used internally.

So far, the modules only imported variables from other modules. However, it is also possible for a module to import and export macro definitions. The definitions for these macros must be listed explicitly in the interface. This unique feature of our system enables independent compilation in the presence of macros. The macro definition appears as a `define-syntax` form in the interface; it is also visible in the body of the exporting module. For example, the following interface defines a macro `delay` which wraps its argument into a thunk (a procedure with no arguments) before applying the procedure `make-promise` to it:

```
(define-interface promises-interface
  (export
    (open scheme-interface)
    (define-syntax delay
      (syntax-rules ()
        ((delay form) (make-promise (lambda () form)))))
    force))
```

The `open` clause of the interface provides bindings for the free identifiers in the output of the macro. If these imports do not bind an identifier, the identifier is assumed to be a variable (not a keyword) and the module that provides the interface must supply a definition for this variable. We call this strategy "defaults to provider." In the example above, `scheme-interface` provides a binding for `lambda` but not for `make-promise`. A module with `promises-interface` as its export interface must provide a definition for `make-promise` and, of course, for `force`. Here is a simple implementation of such a module:

```
(define-module simple-promises promises-interface
   (open scheme-interface)
   (begin
     (define (make-promise thunk) thunk)
     (define (force promise) (promise))))
```

The "defaults to provider" strategy sets the binding place of the free identifiers in the output of an imported macro to the module that provides the corresponding interface. This is convenient and powerful because the writer of the interface does not need to specify the actual set of identifiers in the output of a macro. However, sometimes identifiers within macro definitions are assumed to be unbound. In particular, the identifiers used as literal identifiers of `syntax-rules` transformers are often unbound. But with the "defaults to provider" strategy, there are no unbound identifiers: identifiers in the output of an exported macro, which are not imported via an `open` clause, are assumed to be bound by the providing module. Consider the following example:[1]

```
(define-interface if-t-e-interface
  (export
  (open scheme-interface)
  (define-syntax if-t-e
    (syntax-rules (then else)
       ((if-t-e test then cons else alt)
        (if test cons alt))))))

(define-module main main-interface
```

[1]The example uses `begin`, another module clause that permits the programmer to write the source code within the module declaration. We use it here to keep the presentation compact.

```
  (open scheme-interface
        if-t-e-interface)
  (begin
    (if-t-e 42 then 1 else 0))) ;; syntax error!
```

Here the macro call in the body of `main` would not match the macro definition of `if-t-e` because the "defaults to provider" strategy binds `then` and `else` to the provider of `if-t-e-interface` but in the body of `main` these identifiers are unbound. To remedy this problem, interfaces may contain an additional `free` clause that lists identifiers that are unbound in the providing module. Using this clause, the interface definition above can be written as:

```
(define-interface if-t-e-interface
  (export
    (open scheme-interface)
    (free then else)
    (define-syntax if-t-e
      (syntax-rules (then else)
         ((if-t-e test then cons else alt)
          (if test cons alt))))))
```

Now the macro call `(if-t-e 42 then 1 else 0)` matches the macro definition because `then` and `else` are unbound in the macro definition and in the macro call.

So far, the modules in the examples have always been on the top-level. However, modules can also occur as expressions. In this case, the primitive `make-module` creates a module. Its syntax is

- `(make-module` *exp-interface* `(open` *imp-interface ...*`)` *module-def ...*`)` → *module*

In this prototype, *exp-interface* is the name of the export interface, after `open` the list of imported interfaces follows, and *module-def ...* is a series of local definitions that make up the body of the module. It is important to note that interfaces are not first-class values. Instead, they are only accessible during elaboration time, just like macro transformers. Furthermore, there is no form to define an interface within a module. Instead, the special module clause `import-from-config` imports an interface name from the configuration level:

```
(define-interface an-a-interface
   (export a))

(define-interface dupper-interface
   (export dup&upcase))

(define-module dupper dupper-interface
   (open scheme-interface
         higher-order-modules-interface)
   (import-form-config bla-interface)
   (begin
      (define (dup&upcase str)
          (let ((strstr (string-append str str)))
             (make-module an-a-interface
                 (open string-morphisms-interface
                       unicode-tools-interface)
                 (define a (string-map char-upcase strstr)))))))
```

The code defines a procedure `dup&upcase` which accepts as its parameter a string, binds a local variable `strstr` to the duplicated string and returns a module, which defines its exported variable `a` as `strstr` with all letters upper-case. To access the `char-upcase` procedure, the local module imports the interface `unicode-tools-interface`; to access the `string-map` procedure, it imports

`string-morphisms-interface`. The module has access to the bindings of the surrounding scope, hence it can access `strstr`.

The return value of `dup&upcase` is an unlinked module. To access `a`, it is first necessary to link and evaluate the module. For linking, the module `higher-order-modules` provides the procedures `link` and `link-all`. Analogously to the program clauses with the same name, `link` satisfies imported interfaces with implementations. Like the program clause, this procedure operates on an importing module, an interface, and an exporting module. The `link-all` procedure corresponds to the automatic linker and tries to satisfy the imported interfaces of its first argument with the remaining arguments. The procedure `eval-package` evaluates the module[2]. Like the `make-module` primitive, the module `higher-order-modules` provides these procedures via the interface `higher-order-modules-interface`.

```
(define-interface unicode-modules-interface
   (export unicode-tools
           string-morphisms
           ...))

(define-module main main-interface
   (open scheme-interface
         higher-order-modules-interface
         unicode-modules-interface
         dupper-interface)
   (begin
      (define mod (dup&upcase "hom"))
      (define structure (link-all mod unicode-tools string-morphisms))
      (define package (eval-package structure))
      (define homhom-upcase (package-binding package 'a))))
```

The procedure `package-binding` selects the top-level binding of a evaluated module by its name. That is, the last definition binds `fpmfpm-upcase` to `"HOMHOM"`.

The above example shows one possible usage scenario for higher-order modules: A module can import definitions into a local scope without affecting the surrounding scope. Another important application of first-class modules is the loading of a module at run-time, enabling "plug-ins". Examples of software products that load plug-ins to extend functionality include Gimp, Photoshop, Mozilla, and Eclipse. There even exist standard interfaces for plug-ins that are used by multiple applications, such as TWAIN [TWA00] or LADSPA [LAD05].

As our system does not distinguish top-level modules and internal modules semantically, programmers can also turn a top-level module into a first-class value and use it as a plug-in. This eases program development as the programmer can test a module as an ordinary top-level module and use the same module later as a plug-in.

2.3 Terminology

There is no uniform terminology in the literature on module systems. Most notably, the definition of the term "module" itself is not tied down. This section presents our definition of the most important terms and relates them to other common taxonomies. The terminology used here is inspired by but not identical to the terminology of the Scheme 48 module system by Jonathan Rees [Ree94]. From Cardelli [Car97], we borrow the term "fragment".

The notion of an *identifier* is usually defined by the core language. Here, we use the terminology that for languages with macros, an identifier is either a variable or a keyword, and for languages without macros, an identifier is always a variable. The identifier might contain information about the place where it is defined (so-called *qualified* or *long* identifiers). We will not consider long identifiers here as they fundamentally complicate macro expansion [Blu97].

[2] A package is a fully linked module with state.

A language description usually contains a definition of the *static semantics* of the language. The static semantics determines for each language construct a set of properties that can be derived without evaluation. These properties typically include information about the binding place of an identifier for languages with lexical binding, the type of an expression for statically typed languages, and the macro transformer for a keyword in languages with macros.

The core component of any modular component of a program is a *source code fragment* or *fragment* for short. Source code is what the programmer expects to be compiled and evaluated. A fragment may be not self-contained. It may refer to identifiers not defined in the fragment itself: these are the *free* identifiers of the fragment. In a language with macros, these identifiers might even be keywords of the language. Consequently, in such a language it is not possible to construct the syntax tree for a source-code fragment standing by itself, let alone to compile the fragment.

An *interface* is an enumeration of identifiers along with their static semantics. For a language with macros, the static semantics of a keyword (a macro identifier) is the definition of the macro along with the static semantics of the identifiers referenced by the macro. For a statically typed language, the static semantics of a variable is its type. Otherwise, the static semantics is usually just the description of its binding place.

A *module definition* relates a fragment to information about the free identifiers of the fragment and to an interface that defines the export information.

A *unit* contains a source-code fragment and enough additional information to derive the static semantics of the fragment. For the free identifiers of the fragment, a set of *imported interfaces* provides this additional information. For most languages this specification permits some form of compilation of the source code. The free identifiers of the fragment become *imported identifiers* of the unit as their static semantics describes their binding place.

A *module* consists of a unit and an export interface. A module serves as a reference for import statements of other modules within link environments. Several modules may share the same unit and therefore compiled code. A module is a static unit, it does not have state.

A *link environment* defines for each module, which other modules provide the free identifiers of the module. A link environment might be incomplete in the sense that for some modules, it does not define all modules providing the free identifiers of these modules.

The process of deriving the link environment is called *static linking* because it resolves identifiers and happens before program evaluation.

A *structure* is a module with a link environment that describes which other structures provide the free identifiers of the module's unit.

A *package* is a structure with a run-time environment called the *template* that contains values of the identifiers imported from other packages and for the identifiers defined in the package itself. These values might be set to some undefined value and evaluation can change these values. Therefore, the values of the identifiers constitute the state of the package. Modules and structures do not have state.

Target code is a representation of the source-code fragment suitable to be executed by a machine. It contains instructions to load values from the template.

Using a source-code fragment as the code component of several different modules makes it possible to parameterize over the static semantics of the identifiers. This enables the programmer to use the source code in different contexts. For example, the source code of the implementation of the operational semantics of this dissertation has been used as the input for a rewriting system and to derive the LaTeX code of this dissertation. The definition of elemental keywords such as `define` differs within these two modules: While the operational semantics uses the standard binding for `define`, the definition of `define` during the generation of the LaTeX evaluates the right-hand side of the definition to a string and writes it into a file, which is later included into the hand-written part of the dissertation.

2.4 Phase Overview

With the introduction of modules, evaluation of a program is defined in terms of several phases that turn the source code fragments into modules, structures, and packages and evaluate the packages. This section describes these phases and lists their inputs and outputs.

Configuration This phase processes module and interface definitions and creates modules, units and a link environment. Configuration creates modules that share a common unit if the module definitions reference the same fragment and if the imported interfaces of the modules are the same. The input of the configuration phase is a set of module and interface definitions and a specification for linking. The output is a link environment, a set of modules, and a set of units.

Elaboration This phase derives the static semantics of a source code fragment. For a language with macros, elaboration performs macro expansion. For a language with a static type system, it performs type checking. Input to elaboration is a unit, its output is the static semantics of the fragment.

Static linking Static linking creates a link environment, which determines the other modules that provide the free identifiers of the module. Its input are a set of modules, the module to be linked, and a link environment. The output of the phase is the link environment.

Compilation This phase creates target code from the static semantics of a fragment. It also provides the layout of the template.

Dynamic linking This phase turns a module into a package. It creates the template of the package and fills the template with values for all imported identifiers. The values of the identifiers of the package are set to an unspecific value. The input of the phase is the module to be linked, the link environment, and a set of packages that provide the imports.

Evaluation This phase corresponds to the evaluation proper of the language. It defines the values for the definitions of the package in the template.

Figure 2.1 shows a graphical representation of the phases and the entities. The dashed lines clarify the numerical relationships between the entities:[3]

- Several module definitions may reference the same source-code fragment.
- Several modules may share the same unit. In this case, configuration has detected that the imports and the fragments of the modules are the same.
- Several units may refer to the same fragment. This case results from module whose definition refer to the same fragment but whose imports differ.
- For each module, there is exactly one structure because static linking creates a structure for each module.
- Multiple structures may reference the same unit because for each structure there is one module but several modules may share a unit.
- As the template constitutes the state of the package, there is one template for each package.
- Multiple packages may share the same target code because of the many-to-one relationship between structures and target code.

[3]The occurrences of `N` in the figure do not refer to some variable but simply mean “many” as opposed to “one.”

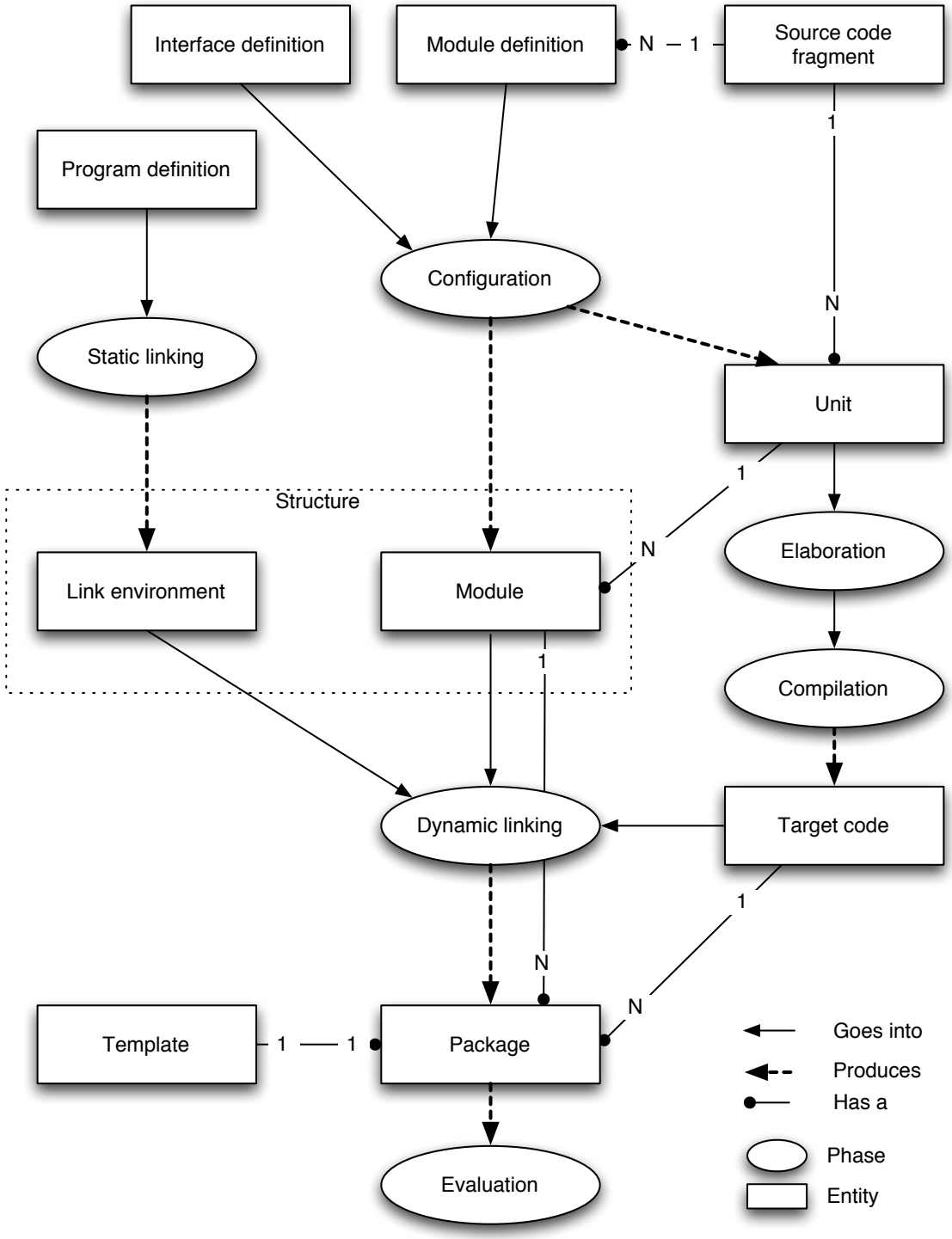

Figure 2.1: Overview of phases and entities

2.5 The Missing Link Revisited

This section presents the core of Kelsey's description of a fully-parameterized module system as presented in the unpublished draft [Kel97] along with two extensions necessary for our system with hygienic macros and independent compilation: interfaces may import other interfaces to provide static information for identifiers inserted by macros, and modules contain units instead of the code as in Kelsey's system to allow a set of modules to share compiled code. We also omit some features not necessary for our presentation: modules with multiple export interfaces, importing several modules with the same interface, and some sanity checks that cannot be performed in the presence of macros.

Interfaces are partial functions from names to static information and carry a unique identifier. The static information of a variable is either the symbol variable or the symbol free depending on the variable being bound or unbound. The static information of a keyword is a transformer. The static information of an imported identifier is the static information of the identifier as defined by the exporting interface paired with the name of the exporting interface.[4]

$$
\begin{array}{rcl}
n & \in & \text{Name} \\
N & \in & \mathcal{P}(\text{Names}) \\
\iota & \in & \text{Interface-Uid} \\
\mathcal{I} & \in & \mathcal{P}(\text{Interface}) \\
i & \in & \text{Interface} = \text{Interface-Uid} \times \text{Static-Env} \\
tf & \in & \text{Transformer} \\
si & \in & \text{Static} = \{\text{variable}, \text{free}\} \cup \text{Transformer} \cup \langle \text{Static}, \text{Interface} \rangle \\
se & \in & \text{Static-Env} = \text{Name} \xrightarrow{\text{fin}} \text{Static}
\end{array}
$$

For a static environment se, $dom(se)$ is the domain of se and $defined(se)$ is the set of identifiers in $dom(se)$ that is not mapped to free. For an interface i, $interfse(i)$ is the static environment of the interface:

$$interfse(\langle \iota, se \rangle) = se$$

A unit consists of a code fragment and a static environment which must provide the static semantics of all free variables of the fragment:

$$
\begin{array}{rcl}
code & \in & \text{Code} \\
u & \in & \text{Unit} = \text{Code} \times \text{Static-Env}
\end{array}
$$

A module combines a unit with imported interfaces and an exported interface. A module also has a unique identifier:

$$
\begin{array}{rcl}
\mu & \in & \text{Module-Uid} \\
M & \in & \mathcal{P}(\text{Module}) \\
m & \in & \text{Module} = \text{Module-Uid} \times \mathcal{P}(\text{Interface})_{imports} \times \text{Interface} \times \text{Unit}
\end{array}
$$

A program consists of a set of modules, an initialization sequence, and a link environment:

$$
\begin{array}{rcl}
p & \in & \text{Program} = \mathcal{P}(\text{Module}) \times \text{Init-Order} \times \text{Link-Env} \\
le & \in & \text{Link-Env} = \langle \text{Module}_{importing}, \text{Interface} \rangle \xrightarrow{\text{fin}} \text{Module}_{exporting} \\
io & \in & \text{Init-Order} = \mathcal{P}(\text{Module} \times \text{Module})
\end{array}
$$

[4] The set of interfaces is assumed to be finite and the interface import graph is assumed to be acyclic.

The constructors for interfaces and units are straightforward:

$$\begin{aligned}&\textit{make-interface} : \text{Static-Env} \to \text{Interface}\\&\textit{make-interface}(se) = \langle \textit{new-interface-uid}(), se\rangle\end{aligned}$$

$$\begin{aligned}&\textit{make-unit} : \text{Code} \times \text{Static-Env} \to \text{Unit}\\&\textit{make-unit}(code, se) = \langle code, se\rangle\end{aligned}$$

The domain of the static environment of a unit has to include all free identifiers of the code fragment.[5]

The constructor for modules combines a unit with imported and exported interfaces and attaches a unique identifier. The static environment of the unit must match the union of the static environments of the imported interfaces:

$$\begin{aligned}&\textit{make-module}{:}\mathcal{P}(\text{Interface}) \times \text{Interface} \times \text{Code} \times \text{Static-Env} \to \text{Module}\\&\textit{make-module}(\mathcal{I}_{imports}, i_{exports}, \langle code, se\rangle) =\\&\quad\begin{cases}\langle \textit{new-module-uid}(), \mathcal{I}_{imports}, i_{export}, \langle code, se\rangle\rangle & \text{if } se = \bigcup\limits_{i \in i_{imports}} \textit{interfse}(i)\\ \textit{error} & \text{otherwise}\end{cases}\end{aligned}$$

Two small functions on modules turn out to be useful:

$$\begin{aligned}&\textit{imports} : \text{Module} \to \mathcal{P}(\text{Interface})\\&\textit{imports}(\langle \mu, \mathcal{I}_{imports}, i_{export}, u\rangle) = \mathcal{I}_{imports}\end{aligned}$$

$$\begin{aligned}&\textit{export} : \text{Module} \to \text{Interface}\\&\textit{export}(\langle \mu, \mathcal{I}_{imports}, i_{export}, u\rangle) = i_{export}\end{aligned}$$

Two constructors for programs exist. The first turns a module into a program with with an empty link environment and initialization sequence:

$$\begin{aligned}&\textit{make-program} : \text{Module} \to \text{Program}\\&\textit{make-program}(m) = \langle \{m\}, \emptyset, \emptyset\rangle\end{aligned}$$

The second constructor combines two programs into a new program. The link environments and the initial ordering of the two programs must be consistent. For the link environment, this means that module-import pairs appearing in both environments, must be mapped to the same exporting module. For the initial ordering, consistence means that the union of the two orderings must be *acyclic*, i.e. its transitive closure has to be irreflexive.

$$\begin{aligned}&\textit{make-program} : \text{Program} \times \text{Program} \to \text{Program}\\&\textit{make-program}(\langle M_0, io_0, le_0\rangle, \langle M_1, io_1, le_1\rangle) =\\&\quad\begin{cases}\langle M_0 \cup M_1, io, le\rangle & \text{if } \forall \langle m, i\rangle \in \textit{dom}(le_0) \cap \textit{dom}(le_1) : le_0(\langle m, i\rangle) = le_1(\langle m, i\rangle)\\ & \quad \wedge\ io \text{ is acyclic}\\ \textit{error} & \text{otherwise}\end{cases}\end{aligned}$$

where

$$\begin{aligned}io &= io_0 \cup io_1\\ le(\langle m, i\rangle) &= \begin{cases}le_0(\langle m, i\rangle) & \text{if } \langle m, i\rangle \in \textit{dom}(le_0)\\ le_1(\langle m, i\rangle) & \text{otherwise}\end{cases}\end{aligned}$$

[5] However, the set of free identifiers cannot be determined without macro expansion. Hence the constructor of unit cannot check that the static environment is appropriate for the code fragment.

The function *link* adds a single link to a program and returns a new program if the arguments are consistent with the existing link environment of the program:

$$\textit{link} : \text{Program} \times \text{Module} \times \text{Interface} \times \text{Module} \to \text{Program}$$

$$\textit{link}(\langle M, io, le\rangle, m_{exporting}, i, m_{importing}) = \begin{cases} \langle M, io, le'\rangle & \text{if } m_{exporting} \in M \\ & \quad \wedge\ m_{importing} \in M \\ & \quad \wedge\ i \in \textit{imports}(m_{importing}) \\ & \quad \wedge\ i = \textit{export}(m_{exporting}) \\ & \quad \wedge\ \langle m_{importing}, i\rangle \notin \textit{dom}(le) \\ \textit{error} & \text{otherwise} \end{cases}$$

where

$$le'(\langle m', i'\rangle) = \begin{cases} m_{exporting} & \text{if } m' = m_{importing} \wedge\ i' = i \\ le(\langle m', i'\rangle) & \text{otherwise} \end{cases}$$

The function *link-all* implements the automatic linker that extends the link environment of a program by all links for imported interfaces that have only one implementation in the program:

$$\textit{link-all} : \text{Program} \to \text{Program}$$

$$\textit{link-all}(\langle M, io, le\rangle) = \begin{cases} \textit{link-all}(\langle M, io, le'\rangle) & \text{if } \exists m \in M : \exists i \in \textit{imports}(m) : \\ & \langle m, i\rangle \notin \textit{dom}(()le) \wedge \exists! m_e \in M : i = \textit{export}(m_e) \\ & \text{where} \\ & le'(\langle m', i'\rangle) = \begin{cases} m_e & \text{if } m' = m \wedge i = i' \\ le(m', i') & \text{otherwise} \end{cases} \\ \langle M, io, le\rangle \end{cases}$$

The function *init-order* extends the initial ordering of a program by a new pair of modules:

$$\textit{init-order} : \text{Program} \times \text{Module} \times \text{Module} \to \text{Program}$$

$$\textit{init-order}(\langle M, io, le\rangle, m_{before}, m_{after}) = \begin{cases} \langle M, io', le\rangle & \text{if } io' \text{ is acyclic} \\ \textit{error} & \text{otherwise} \end{cases}$$

$$\text{where } io' = io \cup \{\langle m_{before}, m_{after}\rangle\}$$

The function *resolve* looks up a name in the link environment:

$$qn \in \text{Qualified-Name} = \text{Module-Uid} \times \text{Name}$$
$$\textit{resolve} : \text{Program} \to \text{Name} \to qn \text{ or } \textit{error}$$

A nice property of Kelsey's module system is that its definition contains no formal propositions about the language and hence a set of abstract data types suffices for the definition. This gives the system a broad range of applications in the design of a language: The system can be used only during configuration and return a list of fully linked modules as in a traditional module system, or the system can be used during separate compilation to determine the imported modules, or—if independent compilation is possible—the system only describes how to link a set of already compiled modules into a library or a complete program. As already sketched in Section 2.2, we take the latter approach and Sections 2.7 and 2.8 formally define the configuration language and how it translates to the abstract data types. Our dynamic linker uses the link environment of a program to combine compiled (actually macro-expanded) modules into a complete program. The dynamic linker does not explicitly call the resolve function but instead perceives the link environment as a ternary relation that describes for each importing modules, which modules satisfy the imported interfaces. That is, it does not use *resolve* to find the binding place of a single variable but instead resolves all variables listed in an interface at once. But this is of course just a technical variation that does not change the binding place of imported variables.

2.6 Transformation with Code Sharing

To compile the source code of a module, it is necessary to resolve the references to imported identifiers. As a naive approach, this could be done by a source code transformation which replaces the names defined at the top-level of the modules by generated, globally unique names. However, such a transformation would have a severe drawback: it would emit code for every module within the program. This is a waste of space as it means that modules built on the same unit do not share code. In his master's thesis Wiesenmaier developed a transformation which preserves sharing of code among modules with the same unit. The transformation works by generating a procedure for each unit. The procedure expects a vector of cells as its only argument. The vector corresponds to the imports of the unit and the cells contain the values of the imported variables. The transformation maintains a mapping from imported names to indices into the vector for each unit. It uses this mapping to convert occurrences of global variables into references in the vector. A module arises from a unit by applying the procedure to a vector filled with cells containing the imports.

2.7 Configuration Language

For the configuration phase it is necessary to define a language that includes module definitions, interface definitions, and linking specifications. The module definitions contain source code fragments and imported interfaces. The interface definitions contain the identifiers along with their static semantics. The linking specification describes how the configuration phase should determine for each module the set of imported modules. This can be done explicitly, corresponding to the *link* operation from Section 2.5 or implicitly by relying some algorithm with in the configuration phase. The latter variant corresponds to the *link-all* operation from Section 2.5.

In his diploma thesis, Wiesenmaier also defined a configuration language that builds on the abstract data types from Section 2.5. Wiesenmaier's language is in turn inspired by the configuration language of Scheme 48. We use Wiesenmaier's language here but extend it by macros in interfaces. For the sake of simplicity, we omit tags and renaming on import as present in Wiesenmaier's and in Kelsey's work. However, I am aware that both features are important in realistic programs: importing with tags enables a module to require the same interface several times and satisfy it with different implementations, and renaming helps to avoid name clashes between imported (and internal) identifiers. Including these features later is straightforward for the configuration language as shown by Wiesenmaier. The main additions to Kelsey's original specification are macro declarations, import clauses, and unbound declarations within interfaces.

Figure 2.2 contains the grammar of the language. A configuration is a series of definitions. A definition may either concern an interface, a module, or a program. Interface definitions include a name and a description of the interface. The description itself is either an `export` form or a `compound-interface` form. The latter simply combines several interfaces referenced by name into one interface description. A list of export declarations follows the `export` form. An export declaration is either the name of an exported variable, or the macro definition of an exported keyword, or an `open` form that lists the names of interfaces providing bindings for the identifiers inserted by the macros, or a `free` form that declares the unbound identifiers in the output of the exported macros.

A module definition first determines the name of the module and the name of the export interface. Afterwards, a list of module clauses follows. Within these clauses, an `open` form defines the imported interfaces of the module, possibly with a tag. The `begin` clause contains the source code as a sequence of definitions whereas a `files` clause references source code contained in a file.

A program definition contains the name of the program followed by a series of program clauses. The `modules` clause lists the names of the modules that make up the program. The `program` clause includes other programs. The `link` clause adds user-defined link commands to the program. Within a link command, the first name is the name of the importing module, the second name is the interface, and the third name is the module to be imported. The `init-order` clause sets up a

partial order among the modules by listing sequences of module names which should be initialized in the given order.

⟨*configuration*⟩ ⟶ ⟨*definition*⟩*
⟨*definition*⟩ ⟶ ⟨*interface-def*⟩ | ⟨*module-def*⟩ | ⟨*program-def*⟩
⟨*interface-def*⟩ ⟶ `(define-interface` ⟨*name*⟩ ⟨*interface*⟩`)`
⟨*interface*⟩ ⟶ `(export` ⟨*export-decl*⟩*`)` | `(compound-interface` ⟨*name*⟩*`)`
⟨*export-decl*⟩ ⟶ ⟨*name*⟩ | ⟨*macro-def*⟩ | `(open` ⟨*name*⟩*`)` | `(free` ⟨*name*⟩*`)`
⟨*macro-def*⟩ ⟶ `(define-syntax` ⟨*name*⟩ ⟨*transformer*⟩`)`
⟨*module-def*⟩ ⟶ `(define-module` ⟨*name*⟩ ⟨*name*⟩ ⟨*module-clause*⟩*`)`
| `(define-module` ⟨*name*⟩ `(copy-module` ⟨*name*⟩`))`
⟨*module-clause*⟩ ⟶ `(open` ⟨*name*⟩*`)`
| `(begin` ⟨*scheme-definition*⟩*`)`
| `(files` ⟨*pathname*⟩`)`
⟨*pathname*⟩ ⟶ ⟨*name*⟩ | ⟨*string*⟩
⟨*program-def*⟩ ⟶ `(define-program` ⟨*name*⟩ ⟨*program-clause*⟩*`)`
⟨*program-clause*⟩ ⟶ `(modules` ⟨*name*⟩*`)`
| `(programs` ⟨*name*⟩*`)`
| `(link` ⟨*link*⟩*`)`
| `(init-order {(`⟨*name*⟩*`)}*)`
⟨*link*⟩ ⟶ `(`⟨*name*⟩ ⟨*name*⟩ ⟨*name*⟩`)`
⟨*name*⟩ ⟶ Name

Figure 2.2: The configuration language

2.8 Semantics of the Configuration Language

This section defines the semantics of the configuration language from the previous section by transforming programs written in the configuration language into the data types from Section 2.5. The semantics is then an environment function mapping the names of the configuration language to the data types.

Usually, a semantics of a language describes the meaning of abstract syntax terms rather than concrete syntax. This simplifies the description of the semantics because abstract syntax is less verbose than concrete syntax. The configuration language however is so simple that defining an abstract syntax would not pay off. Instead, we simply define for each kind of definition a subset as *normal forms* for the definitions and use these normal forms in the description of the semantics.

There are two normal forms for interface definitions. The first specifies an order among the declarations and permits only one `open` and one `free` clause. The second form limits the arguments of the `compound-interface` form to two:

`(define-interface` ⟨*name*⟩
  `(export (open` ⟨*name*⟩*`)`
          `(free` ⟨*name*⟩*`)`
          ⟨*macro-def*⟩*
          ⟨*name*⟩*`))`
`(define-interface` ⟨*name*⟩ `(compound-interface` ⟨*name*⟩ ⟨*name*⟩`))`

A general `define-interface` form can be transformed into the first normal form by merging the arguments of all `open` clauses and sorting the macro definitions and exports as required. The second form results from the general form with `compound-interface` by introducing a series of interface definitions with fresh names and letting each of the new interfaces combine two interfaces.

The normal forms for module definitions requires either exactly one `open` and one `begin` clause or a `copy-module` clause:

`(define-module` ⟨*name*⟩ ⟨*name*⟩ `(open` ⟨*name*⟩*`)` `(begin` ⟨*scheme-definition*⟩*`))`

`(define-module` ⟨*name*⟩ `(copy-module` ⟨*name*⟩`))`

A `file` clause in the general form is transformed into a `begin` clause by inserting the code from the file into the `begin` clause. Again, several `open` clauses can be merged into one clause.

For programs, the normal form permits exactly one clause of each kind and prescribes a certain order among the clauses. Furthermore, each clause contains only a single argument:

```
(define-program ⟨name⟩
  (programs ⟨name⟩)
  (link ⟨link⟩)
  (init-order (⟨name⟩ ⟨name⟩ ⟨name⟩))
  (modules ⟨name⟩))
```

Figure 2.3 contains the semantics of the normal forms. The semantics is a function that takes as its argument a definition and a *configuration environment* and returns an augmented configuration environment. A configuration maps names to data types:

$$\text{CEnv} : \text{Name} \to \{\text{Interface}, \text{Module}, \text{Program}\}$$

The semantics of a configuration uses the semantics of definitions to construct from an initially empty configuration environment a configuration environment mapping all the defined names to data types. As the semantics of a definition depends on the value of the configuration environment for the names referenced in the definition, the definitions within a configuration must be ordered such that each definition appears before its first reference. Figure 2.4 defines a strict partial order $\sqsubset$ that fulfills this requirement.

The semantics of a well-formed configuration ⟨*configuration*⟩ sorted by $\sqsubset$ is then defined as

$$[\![\langle \textit{configuration} \rangle]\!]\sigma_{\text{empty}}$$

where σ_{empty} is an empty configuration environment and the semantics function $[\![]\!]$ for configurations is defined as:

$$\begin{aligned} [\![]\!] &: \langle \textit{configuration} \rangle \times \text{CEnv} \to \text{CEnv} \\ [\![\epsilon]\!]\sigma &= \sigma \\ [\![\langle \textit{definition} \rangle \langle \textit{definition} \rangle^*]\!]\sigma &= [\![\langle \textit{definition} \rangle^*]\!]([\![\langle \textit{definition} \rangle]\!]_{\text{def}}\sigma) \end{aligned}$$

Extending the semantics to the general cases prescribed by the grammar is straightforward. For the `define-module` and `define-interface` definitions, this has already been sketched above. For the `define-program` clauses the contents of clauses that appear several times can be merged. Then the semantics of the resulting form can be explained by extending the current semantics as follows:

- If the `link` clause of a program forms contains a series of links, this can be explained by adding calls to the *link* function at the place of the current single call to *link*.
- If an argument of an `init-order` clause contains more than two modules, for every pair of modules a call to *init-order* needs to be added. A `init-order` clause with multiple arguments simply adds calls to *init-order* for one argument after the other.

$[\![]\!]_{\text{def}} : \langle \textit{definition} \rangle \times \text{CEnv} \to \text{CEnv}$

$$[\![\texttt{(define-interface } N_1 \texttt{ (export (open } N_2 \ \ldots N_j\texttt{)(free } n_1 \ldots n_m\texttt{)}$$
$$\texttt{(define-syntax } k_1 \ tf_1\texttt{)} \ldots \texttt{(define-syntax } k_t \ tf_t\texttt{) } v_1 \ldots v_l\texttt{))}]\!]_{\text{def}}\sigma =$$
$$\begin{cases} \sigma[N_1 \mapsto \textit{make-interface}(se)] & \text{if } \bigcap_{o=2}^{n} \textit{defined}(\textit{interfse}(i_o)) = \emptyset \\ & \text{and}(\bigcup_{o=2}^{n} \textit{defined}(\textit{interfse}(i_o))) \cap \\ & \quad \{n_1, \ldots, n_m, v_1, \ldots, v_l, k_1, \ldots, k_t\} = \emptyset \\ \textit{error} & \text{otherwise} \end{cases}$$

where

$$\sigma(N_2) = i_2, \ldots, \sigma(N_j) = i_j$$

$$se(n) = \begin{cases} \text{variable} & \text{if } n \in \{v_1, \ldots, v_l\} \\ tf_i & \text{if } n = k_i, 1 \leq i \leq t \\ \text{free} & \text{if } n \in \{n_1, \ldots, n_m\} \\ \langle se'(n), i \rangle & \text{if } \exists i \in \{i_1, \ldots, i_j\} : se' = \textit{interfse}(i), n \in \textit{dom}(se') \end{cases}$$

$$[\![\texttt{(define-interface } N_1 \texttt{ (compound-interface } N_2 \ N_3\texttt{))}]\!]\sigma =$$
$$\begin{cases} \sigma[N_1 \mapsto \textit{make-interface}(se)] & \text{if } \forall n \in \textit{dom}(se_2) \cap \textit{dom}(se_3) : se_2(n) = se_3(n) \\ \textit{error} & \text{otherwise} \end{cases}$$

where

$$\sigma(N_2) = i_2$$
$$\sigma(N_3) = i_3$$
$$se(n) = \begin{cases} se_2(n) & \text{if } se_2 = \textit{interfse}(i_2), n \in \textit{dom}(se_2) \\ se_3(n) & \text{if } se_3 = \textit{interfse}(i_3), n \in \textit{dom}(se_3) \setminus \textit{dom}(se_2) \end{cases}$$

$$[\![\texttt{(define-module } N_m \ N_{\texttt{i-exp}} \texttt{ (open } N_{\texttt{i-impl}} \ \ldots N_{\texttt{i-impj}}\texttt{) (begin } \textit{code}\texttt{))}]\!]\sigma =$$
$$\sigma[N_m \mapsto \textit{make-module}(\{i_1 \ldots i_j\}, i_{\text{exp}}, \textit{make-unit}(\textit{code}, se))]$$

where

$$\sigma(N_{\text{i-imp1}}) = i_1, \ldots, \sigma(N_{\text{i-impj}}) = i_j$$
$$\sigma(N_{\text{i-exp}}) = i_{\text{exp}}$$
$$se(n) = \langle se'(n), i \rangle \text{ if } \exists i \in \{i_1, \ldots, i_j\} : se' = \textit{interfse}(i), n \in \textit{dom}(se')$$

$$[\![\texttt{(define-module } N_1 \texttt{ (copy-module } N_2\texttt{))}]\!]\sigma = \sigma[N_1 \mapsto \textit{make-module}(is, i, u)]$$

where

$$\sigma(N_2) = \langle id_1, is, i, u \rangle$$

$$[\![\texttt{(define-program } N_{p1} \texttt{ (program } N_{p2}\texttt{) (link (}N_{ml1} N_i N_{ml2}\texttt{))}$$
$$\texttt{(init-order (}N_{mio1} N_{mio2}\texttt{)) (modules } N_m\texttt{))}]\!]\sigma =$$
$$\sigma[N_{p1} \mapsto \textit{init-order}(\textit{link-all}(\textit{link}(\textit{make-program}(p_2, \textit{make-program}(m)), m_{l1}, i, m_{l2})), m_{io1}, m_{io2})]$$

where

$$\sigma(N_{p2}) = p_2, \sigma(N_{ml1}) = m_{l1}, \sigma(N_i) = i,$$
$$\sigma(N_{ml2}) = m_{l2}, \sigma(N_{mio1}) = m_{io1}, \sigma(N_{mio2}) = m_{io2}, \sigma(N_m) = m$$

Figure 2.3: Semantics of the configuration language

(define-interface n ⟨*interface*⟩) ⊏
(define-interface ⟨*name*⟩ (export ...(open ...n ...) ...))

(define-interface n ⟨*interface*⟩) ⊏
(define-interface ⟨*name*⟩ (compound-interface n ⟨*name*⟩))

(define-interface ⟨$name_1$⟩ ⟨*interface*⟩) ⊏
(define-interface ⟨*name*⟩ (compound-interface ⟨$name_2$⟩ n))

(define-interface n ⟨*interface*⟩) ⊏
(define-module ⟨*name*⟩ n ...)

(define-interface n ⟨*interface*⟩) ⊏
(define-module ⟨*name*⟩ ⟨*name*⟩ ...(open ...n...) (begin ...))

(define-module n ...) ⊏
(define-module ⟨*name*⟩ (copy-module n))

(define-interface n ⟨*interface*⟩) ⊏
(define-program ⟨*name*⟩ ...(link (⟨*name*⟩ n ⟨*name*⟩)) ...)

(define-module n ...) ⊏
(define-program ⟨*name*⟩ ...(link (n ⟨*name*⟩ ⟨*name*⟩)) ...)

(define-module n ...) ⊏
(define-program ⟨*name*⟩ ...(link (⟨*name*⟩ ⟨*name*⟩ n)) ...)

(define-module n ...) ⊏
(define-program ⟨*name*⟩ ...(init-order n ⟨*name*⟩ ⟨*name*⟩) ...)

(define-module n ...) ⊏
(define-program ⟨*name*⟩ ...(init-order ⟨*name*⟩ n ⟨*name*⟩) ...)

(define-module n ...) ⊏
(define-program ⟨*name*⟩ ...(init-order ⟨*name*⟩ ⟨*name*⟩ n) ...)

(define-module n ...) ⊏
(define-program ⟨*name*⟩ ...(modules n) ...)

(define-program n ...) ⊏
(define-program ⟨*name*⟩ (programs n) ...)

Figure 2.4: Partial order among definition clauses

Chapter 3

A Semantics for Hygienic Macros

This chapter defines a formal semantics for hygienic macros in a language with s-expression based syntax. The semantics includes a parser and a core macro expander plus two transformers. The first transformer features computational macros and is close but not identical to the `syntax-case` system [DHB92, Dyb96, Dyb92]. The second transformer is very close to Scheme's rewriting-like `syntax-rules` facility, omitting only ellipsis patterns. The definition of both systems builds on explicit substitutions [ACCL91] as pioneered by Bove and Arbilla [BA92].

Usually, reduction systems for programming languages are defined on a set of terms corresponding to a representation of the abstract syntax. The concrete syntax of the language does not matter. In addition, authors often assume Barendregt's variable convention [Bar84]. The convention implies that bound variables are always distinct from free variables. These two simplifications cannot be used if the language supports macros because macro expansion interleaves with parsing while creating the abstract syntax tree and (hygienic) macro expansion needs to preserve the correct binding relations, *especially* for variables whose names are not unique.

To specify a semantics for macro expansion, this work starts from the concrete syntax of the language. The first step of the semantics specifies the relation between concrete and abstract syntax by a set of reductions. These reductions parse the concrete syntax, accumulate macro definitions, expand macro applications, and finally produce abstract syntax. The representation of a variable in the abstract syntax is not simply the name of the variable but contains additional information, called a *level*, that is reminiscent of de Bruijn indices and records the binding place of the variable. This representation is the key ingredient for the description of hygienic macro expansion. The semantics of evaluation of the language differs from an ordinary call-by-value semantics in exactly this point, that is, it relates abstract syntax terms defined over identifiers with levels.

The rest of this chapter is organized as follows: Section 3.1 reviews the basic properties of hygienic macro expansion, Section 3.2 presents the ordinary call-by-value lambda calculus, Section 3.3 presents the abstract syntax over identifiers with levels and its call-by-value evaluation, Sections 3.4 and 3.5 describe the transformation from concrete to abstract syntax by reductions for parsing and macro expansion. Section 3.6 defines a macro expander for computational macros à la `syntax-case`. Sections 3.7 and 3.8 define parser and macro expander for the extended language. Section 3.9 gives a semantics for Scheme's `syntax-rules` facility, Section 3.10 describe the features missing from full Scheme. Section 3.11 compares the macro expander with previous work.

3.1 Hygienic Macros

A macro is a user-defined source-to-source transformation performed by the compiler. A hygienic macro system prevents the user from writing macros that inadvertently capture variables from the input or that insert variables that are inadvertently being captured by surrounding code. In macro systems with local binding constructs for keywords, hygiene also prevents these syntactic binding

constructs from inadvertent capturing of keywords. Hygiene is important for macros because it gives the macro writer complete freedom over the choice of variable names and ensures that none of the macros can interfere with the bindings in a program. Hygiene is therefore essential for using macros in large programs because it hides the use of names internal to the macro.

The literature characterizes hygienic macros by stating requirements the macros must fulfill. The first definition goes back to Kohlbecker et. al. [KFFD86]:

> **Hygiene Condition for Macro Expansion** Generated identifiers that become binding instances in the completely expanded program must only bind identifiers that are generated at the same transcription step.

Here the term "transcription step" is defined as the one-step expansion of a macro.

Kohlbecker's definition does not cover the relationship between identifiers inserted by the macro and binding instances in the original input. Clinger and Rees [CR90] improve this by formulating the following:

> **Hygiene Condition**
>
> 1. It is impossible to write a high-level macro that inserts a binding that can capture references other than those inserted by the macro.
> 2. It is impossible to write a high-level macro that inserts a reference that can be captured by bindings other than those inserted by the macro.

We refer to two conditions by Clinger and Rees as the *First and Second Hygiene Condition* respectively in the subsequent text.

Clinger and Rees also define a condition for local macros that is summarized in [DHB92] as:

> Local macros are *referentially transparent* in the sense that free identifiers appearing in the output of a local macro are scoped where the macro definition appears.

The following examples illustrate the hygienic macro facility in Scheme. The `define-syntax` construct defines a new, global binding for a macro:

- `(define-syntax` *keyword transformer*`)` syntax

In R^{5}RS, the only *transformer* form is a `syntax-rules` expression, which enables the programmer to write macros with a high-level, rewriting-system-like language. Existing Scheme implementations often provide additional facilities that can be used in place of the *transformer* form.

A `syntax-rules` expression consists of a list of literals and a list of rules:

- `(syntax-rules (`*literal* `...)  ((`*pattern template*`) ...))`

Here, *pattern* is an s-expression that is matched against the macro call. For the first pattern that matches the call, the corresponding *template* form replaces the call with all variables in the pattern replaced by the corresponding input forms. A pattern is a variable if it is an identifier and if it is not one of the *literal* identifiers. If a pattern is not a variable, it has to match the input exactly. As an example, consider the following slightly simplified implementation of the `cond` macro from R^{5}RS:

```
(define-syntax cond
  (syntax-rules (else)
     ((cond ((else expr))) expr)
     ((cond ((test rhs))) (if test rhs))
     ((cond ((test rhs) clause ...)) (if test rhs (cond (clause ...))))))
```

This macro lists `else` as a literal identifier. Therefore, in the pattern of the first rule, the identifier `else` has to appear in the input as well, whereas `expr` is a pattern variable. In the second rule, `test` and `rhs` are both variables and if the pattern matches, the macro expander replaces them by the corresponding input forms in the template `(if test rhs)`. This final rule contains an ellipsis pattern: `clause ...` matches arbitrary many input forms.

For the demonstration of hygiene, consider the implementation of the `or` syntax from R^{5}RS:

```
(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or e1 e2 ...) (let ((temp e1))
                      (if temp temp (or e2 ...))))))
```

The macro consists of two rules: The first rule explains `or` with no arguments and rewrites to `#f`. The template of the second rule binds the value of the first argument to a temporary variable, tests its boolean value, returns if it was not false, and rewrites `or` with the remaining arguments otherwise. The use of a temporary variable is necessary to prevent duplicate evaluation of the expression `e1`. Hygiene ensures that the variable the macro binds cannot capture free occurrences of the same identifier within the input forms, in this case `e2 ...`. That is, in the program

```
(define temp 23)
(or (= 1 2) temp)
```

the identifier `temp` always refers to the global variable and the program evaluates to 23. Without hygiene, the second argument of `or` would be substituted literally into the template, yielding

```
(define temp 23)
(let ((temp (= 1 2)))
   (let ((temp temp))
      (if temp
          temp
          #f)))
```

which evaluates to `#f`.

Besides preventing macros from capturing free variables, hygiene also ensures that the variables used by the macros are not captured. As an example, the `receive` macro provides a nicer interface to Scheme's multiple return values facility [Sto99]:

```
(define-syntax receive
  (syntax-rules ()
    ((receive formals expression body ...)
     (call-with-values (lambda () expression)
                       (lambda formals body ...)))))
```

In the expression

```
(let ((call-with-values 23))
   (receive (a b c) (values 1 2 3)
      (+ a b c)))
```

the macro `receive` inserts a reference to the variable `call-with-values` within the body of a `let` that binds this variable. However, hygiene ensures that the inserted reference to `call-with-values` still refers to the top-level binding from R^{5}RS.

To demonstrate hygiene for keywords, we need Scheme's local binding constructs for macros:

- `(let-syntax ((`*keyword transformer*`) ...)` *body*`)` syntax
- `(letrec-syntax ((`*keyword transformer*`) ...)` *body*`)` syntax

Both special forms bind the keywords to the respective transformers within *body*. In addition, `letrec-syntax` also binds the keywords recursively within the transformers, just as `define-syntax` does.

Now we can demonstrate how hygiene prevents macros from capturing free keywords:

```
(define-syntax foo
  (syntax-rules ()
```

```
    ((foo x)
      (let-syntax
         ((bar (syntax-rules ()
                  ((bar) 12))))
        (list x (bar))))))
(define bar 23)
(foo bar)
```

The macro `foo` expands into the definition of another macro `bar`. Within the body of the `let-syntax` binder, the template contains the pattern variable `x`. The macro application `(foo bar)` binds this pattern variable to `bar`. Hygiene ensures that the `let-syntax` does not capture the global variable `bar`.

Finally, consider the macro `thunkify` that wraps an expression into a thunk (a `lambda` expression with no arguments) to prevent immediate evaluation:

```
(define-syntax thunkify
  (syntax-rules ()
     ((thunkify expr) (lambda () expr))))
```

Here hygiene ensures that the keyword `lambda` in the template always refers to the `lambda` special form, independent of the bindings that surround the macro call. That is the expression

```
(let ((lambda 42))
   ((thunkify 17)))
```

evaluates to 17. Without hygiene this program would expand to

```
(let ((lambda 42))
  ((42 () 17)))
```

which is a syntax error.

Macro expansion needs to treat the `quote` form specially. The operand of `quote` is an s-expression and quote transforms the s-expression into a value whose external representation is the input s-expression. For example

```
(quote 52)
```

evaluates to `52` (the number fifty-two) since the s-expression `52` (the digit "5" followed by the digit "2") is the external representation of the number fifty-two. `Quote` comes in handy for writing literal expressions for lists. For example the expression

```
(quote (1 a))
```

evaluates to the list of the number one and the symbol with name `a`. The form `'datum` is an abbreviation for `(quote datum)`:

```
'(1 . 2)
```

is therefore an expression that evaluates to the pair of one and two.

If `quote` appears in the template of a macro, forms that look like code suddenly becomes data:

```
(define-syntax foo
  (syntax-rules ()
     ((foo (a (b) c)) (quote b))))
(foo (lambda (x) (+ x x)))
```

evaluates to the symbol `x`. Even more strange things can happen if quote is passed as an argument to a macro:

```
(define-syntax swap
   (syntax-rules ()
      ((swap a b) (b a))))
(swap a quote)
```

which evaluates to the symbol `b` even though the template of `swap` looks as if it would produce a procedure application. This example demonstrates that there are macro definitions where it is impossible to derive any kind of information about the output of the macro by analyzing only the macro definition. Consequently, macro expansion is strictly necessary to derive the syntax tree of a program.

3.2 The λ_v Calculus

Before we formally describe expansion for hygienic macros as presented in the previous section, we first define the semantics of the core language. For this semantics do not start from scratch but show that it is equal to the well-known call-by-value λ calculus. This section recapitulates the basic properties of the call-by-value λ calculus, called λ_v-calculus, and introduces the basic formal tools. The presentation is heavily based on [FH92] but the notation deviates in one important aspect: we use "lhs $\rightarrow_R$ rhs (RuleName)" to introduce a notion of reduction (called *rule*) and implicitly union all rules containing $\rightarrow_R$ into one notions of reduction, which we refer to as $\rightarrow_R$. Furthermore, in our notation $\twoheadrightarrow_R$ denotes the reduction, that is, the compatible closure of $\rightarrow_R$ with respect to the expression grammar.

Figure 3.1 contains the language Λ of the λ_v calculus. It comprises values, variables *Vars*, and function applications. Values (Values) in turn comprise constants *Consts* and λ-abstractions The sets of free variables, $FV(M)$, and bound variables, $BV(N)$ of an expression $M$are defined

Syntactic domains:

a	$\in$ *Const*
x	$\in$ *Vars*
M, N	$\in \Lambda$

Abstract syntax:

M	$::= V \mid x \mid (MM)$
V	$::= a \mid \lambda\, x.M$

Figure 3.1: The language Λ

as usual, with λ being the only binding construct:

$$\begin{aligned} FV(a) &= \emptyset & BV(a) &= \emptyset \\ FV(x) &= \{x\} & BV(x) &= \emptyset \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} & BV(\lambda x.M) &= BV(M) \cup \{x\} \\ FV(MN) &= FV(M) \cup FV(N) & BV(MN) &= FV(M) \cup FV(N) \end{aligned}$$

We do *not* adopt Barendregt's *variable convention*[Bar84] that ensures that the bound variables of a term are always disjoint from the free variables. This complicates the subsequent definition of substitution but makes it easier to relate meta-level substitution and explicit substitution. We do however regard terms that differ only in the names of bound variables as equal.

Substitution on Λ terms, written $M[N/x]$, replaces all free occurrences of the variable x in the

term M by the term N:

$$\begin{aligned} a[N/x] &= a \\ x[N/x] &= N \\ y[N/x] &= y \text{ iff } x \neq y \\ (\lambda y.M)[N/x] &= (\lambda z.M[z/y][N/x]) \text{ where } z \text{ is } \textit{fresh} \\ (L\ M)[N/x] &= (L[N/x]\ M[N/x]) \end{aligned}$$

Another syntactic operation performed on the meta-level is the concept of a *term context*, an expression with a hole ("[ ]") at the place of a subexpression. If C is a term context, then filling the hole of C, written $C[M]$, stands for the result of putting the expression Minto the hole of the context C. Unlike substitution, filling the hole of a context may capture free variables. For Λ, the set of contexts is defined as:

$$C \qquad ::= [\,] \mid (M\ C) \mid (C\ M) \mid (\lambda\ x.C)$$

The set *ClosedValues* is the set of values that have no free variables:

$$\textit{ClosedValues} = \{\, V \in \textit{Values} \mid FV(V) = \emptyset\}$$

The precise set of constants is left unspecified; we only assume a partial function δ defined as:

$$\begin{gathered} b \in \textit{BaseConst} \subset \textit{Const} \\ f \in \textit{FunConst} \subset \textit{Const} \\ \delta : \textit{FunConst} \times \textit{BaseConst} \to \textit{ClosedValues} \end{gathered}$$

that assigns meaning to application of functional constants to closed values.

A calculus is an equational theory over a term language. The λ_v calculus is an equational theory over Λ. The two basic relations, called *notions of reductions*, of the calculus are:

$$f\ a \to_\delta \delta(f, a) \qquad (\delta)$$
$$(\lambda x.M)\, V \to_{\beta_v} M[V/x] \qquad (\beta_v)$$

Definition 3.1 (λ_v). *Taken together, they form the notion of reduction $\to_v$:*

$$\to_v = \to_\delta \cup \to_{\beta_v}$$

The one-step reduction $\twoheadrightarrow_v$ is the compatible closure of $\to_v$:

$$M \twoheadrightarrow_v M' \text{ if } M \equiv C[M_1], M' \equiv C[M_2], M_1 \to_v M_2$$

The reduction $\twoheadrightarrow_v^*$ is the reflexive, transitive closure of $\twoheadrightarrow_v$. $=_v$ is the smallest equivalence relation generated by $\twoheadrightarrow_v$. If $M =_v N$, we write $\lambda_v \vdash M = N$.

□

Besides the equational theory, another important aspect of the λ_v-calculus concerns the algorithmic evaluation of expressions. Using evaluation contexts we can define a canonical sequence of reduction steps that reduce an expression to its value.

Definition 3.2 (Evaluation contexts). *The following grammar defines the set of evaluation contexts:*

$$C_v \qquad ::= [\,] \mid (V\ C_v) \mid (C_v\ M)$$

□

A reduction sequence that takes only place within evaluation contexts *standard-reduces* an expression. The evaluation context thereby uniquely splits the expression into a context and the leftmost-innermost redex outside the body of a λ-expression. We can therefore define a standard reduction function:

Definition 3.3 (Standard reduction function). *The standard reduction function reduces M to M', written $M \mapsto_v M'$, if for some evaluation context C_v, $M \equiv C_v[M_1], M' \equiv C_v[M_2]$, $M_1 \to_v M_2$. Let $\mapsto_v^*$ denote the transitive closure of $\mapsto_v$.*

□

The semantics of a program (a closed expression) P can now be defined using the standard reduction function:

Definition 3.4. *Evaluation function*

$$eval_v(P) \stackrel{Def}{\equiv} V \text{ iff } P \mapsto_v^* V$$

3.3 The $\lambda_v^{n,(\!(\ )\!)}$ Calculus

This section presents a variant of the classical call-by-value λ-calculus based on the language Λ^n that uses indexed variables and explicit substitutions. Λ^n is our starting point and the subsequent sections and chapters extend it to incorporate two hygienic macro facilities and a fully-parameterized module system. Indexed variables are the key ingredient to preserve hygiene without relying on renaming. Explicit substitutions makes it possible to make the linking phase explicit.

The abstract syntax uses identifiers with labels to represent variables. The label of an identifier is a natural number and indicates the number of binders between the occurrence of the variable and the abstraction that binds the variable. This representation of identifiers is reminiscent of the de-Bruijn notation [Bru72]. Unlike the de Bruijn notation, identifiers in Λ^n still carry a name along with the level. Keeping this name both improves readability and, more importantly, enables hygienic macro expansion. Keeping the name is important because the final syntactic role of an identifier is only known after macro expansion. Until then it is unknown whether the identifier will become a variable (with the name being "superfluous" in the sense of de Bruijn), or whether the identifier will be used as symbolic data because it appears in the argument of a `quote` form. In the latter case the binding information is irrelevant, only the name matters.

Figure 3.2 contains the abstract syntax for Λ^n. Expressions encompass identifiers with levels, written as x^n, values, which are constants and λ-abstractions, and applications, written $(@\ e\,e)$[1]. Furthermore, explicit substitutions extend the set of expressions—we call them "evaluation substitutions" to distinguish them from other explicit substitutions that will be introduced later. They are written as $e(\!(t)\!)$ where t is a pair $\langle x^n, v\rangle$. Such a pair corresponds to the substitution of a variable x^n by the value v, where the value v must be closed.

> **Remark**: To simplify matters, evaluation substitutions are restricted to substitute closed values only: If we would allow values to contain free variables, the substitution would need to adjust the levels of these variables whenever the substitution enters the scope of a λ-abstraction. However, in programming languages the evaluation of a free (unbound) variable is an error. Certainly, the calculus would be stronger if the substitutions were more powerful but such a calculus is not the main objective of this dissertation.

Names are superfluous in Λ^n because the level of a variable already represents the binding information. Hence we identify terms that are equal modulo the names of the bound variables.

[1] To draw the distinction between abstract and concrete syntax, the abstract syntax uses @ to mark procedure applications.

Syntactic Domains:

a	$\in \mathit{Const}$
n	$\in \mathbb{N}$
x	$\in \mathit{Vars}$
x^n	$\in \mathit{Ids}$
e	$\in \Lambda^n$
v	$\in \mathit{Values}^n$
t	$\in \mathit{Ids} \times \mathit{Values}$

Abstract syntax:

$$
\begin{array}{ll}
e & ::= x^n \mid v \mid (@\ e\,e) \mid e\langle\!(t)\!\rangle \\
v & ::= a \mid (\lambda\ x.e) \\
t & ::= \langle x^n, v\rangle
\end{array}
$$

Figure 3.2: Abstract syntax of Λ^n

However, we want to rule out terms where the names and the levels do not correspond, for example, $\lambda x.y^0$ or $\lambda x.\lambda y.x^0y^0$. To that end, we define a predicate *WellFormed* that determines whether a term is well-formed or not:

$$
\begin{array}{lcl}
\mathit{WellFormed} : \Lambda^n \to \mathit{Boolean} & & \\
\mathit{WellFormed}(e) & \Leftrightarrow & \mathit{WellFormed'}(e, ()) \\
\mathit{WellFormed'} : \Lambda^n \times \mathit{Vars}^* \to \mathit{Boolean} & & \\
\mathit{WellFormed'}((@e_1e_2), s) & \Leftrightarrow & \mathit{WellFormed'}(e_1, s) \wedge \mathit{WellFormed'}(e_2, s) \\
\mathit{WellFormed'}((\lambda x.e), s) & \Leftrightarrow & \mathit{WellFormed'}(e, x : s) \\
\mathit{Wellformed'}(x^i, s) & \Leftrightarrow & s_i = x \\
\mathit{Wellformed'}(e\langle\!(\langle v, x^n\rangle)\!\rangle, s) & \Leftrightarrow & \mathit{WellFormed'}(e, s) \wedge \mathit{WellFormed}(v)
\end{array}
$$

The predicate *WellFormed*(e) uses the helper predicate *WellFormed'* to ensure that the names of the bound variables in e match the names in the corresponding λ. To that end, *WellFormed'* receives as additional argument a sequence of names that corresponds to the variables bound by the λ-abstractions in the context of the expression. The sequence is ordered from the inside to the outside and is initially empty. For each λ-abstraction, *WellFormed'* adds the bound name to the sequence and for an occurrence of a variable with level i, the rule verifies that the name of the variable matches the ith element of the sequence, that is, the name bound by the λ-abstraction i binders away. The values within an evaluation substitutions must be closed, hence the predicate *WellFormed* is applicable.

From now on, we will only consider *WellFormed* expressions.

The elimination of the evaluation substitution is the subject of the $\to_{(\!(\,)\!)}$ notion of reduction defined in Figure 3.3. Rule (SEvalSubstId) applies an evaluation substitution to an identifier that matches the identifier of the substitution. The result is the value from the substitution. If the identifier does not match the identifier from the substitution, rule (SEvalSubstIdOther) drops the substitution. The same happens in rule (SEvalSubstConst), which describes the elimination of an evaluation substitution applied to a constant. For applications, rule (SEvalSubstApp) propagates the substitution to the operator and the operand. For λ-abstractions, rule (SEvalSubstLam) pushes the evaluation substitution to the body but increments the level of the identifier by one as it moves one binder further away from its own binder.

The contexts for $\to_{(\!(\,)\!)}$ are defined as:

$$
\begin{array}{ll}
C_{(\!(\,)\!)} & ::= C'_{(\!(\,)\!)} \mid (@\ C_{(\!(\,)\!)}e) \mid (@\ eC_{(\!(\,)\!)}) \mid (\lambda\ x.\ C_{(\!(\,)\!)}) \\
C'_{(\!(\,)\!)} & ::= [\,] \mid C'_{(\!(\,)\!)}\langle\!(t)\!\rangle
\end{array}
$$

They give rise to a one-step reduction $\twoheadrightarrow_{(\!(\,)\!)}$ for the elimination of evaluation substitutions: $e \twoheadrightarrow_{(\!(\,)\!)} e'$, iff for some context $C_{(\!(\,)\!)}$, $e \equiv C_{(\!(\,)\!)}[e_1]$, $e' \equiv C_{(\!(\,)\!)}[e_2]$, $e_1 \to_{(\!(\,)\!)} e_2$. From this, we can derive an equational theory: $e_1 =_{(\!(\,)\!)} e_2$ if $e_1 \twoheadrightarrow^*_{(\!(\,)\!)} e_2$.

$$
\begin{array}{rll}
\mathit{ESExpr} \ni e_{\text{evs}} ::= e\mathbb{(}t\mathbb{)} & & \\
\qquad \rightarrow_{\mathbb{(}\,\mathbb{)}} \subseteq \mathit{ESExpr} \times \mathit{Expressions} & & \\
x^n\mathbb{(}\langle x^n, v\rangle\mathbb{)} \rightarrow_{\mathbb{(}\,\mathbb{)}} v_1 & & \text{(SEvalSubstId)} \\
y^m\mathbb{(}\langle x^n, v\rangle\mathbb{)} \rightarrow_{\mathbb{(}\,\mathbb{)}} y^m \text{ iff } y^m \neq x^n & & \text{(SEvalSubstIdOther)} \\
a\mathbb{(}t\mathbb{)} \rightarrow_{\mathbb{(}\,\mathbb{)}} a & & \text{(SEvalSubstConst)} \\
(@\ e_1\ e_2)\mathbb{(}t\mathbb{)} \rightarrow_{\mathbb{(}\,\mathbb{)}} (@\ e_1\mathbb{(}t\mathbb{)}\ e_2\mathbb{(}t\mathbb{)}) & & \text{(SEvalSubstApp)} \\
(\lambda y.e)\mathbb{(}\langle x^n, v\rangle\mathbb{)} \rightarrow_{\mathbb{(}\,\mathbb{)}} (\lambda y.e\mathbb{(}\langle x^{n+1}, v\rangle\mathbb{)}) & & \text{(SEvalSubstLam)}
\end{array}
$$

Figure 3.3: Elimination of $\mathbb{(}\,\mathbb{)}$

Proposition 3.5 (terminating). $\rightarrow_{\mathbb{(}\,\mathbb{)}}$ *is terminating.*

Proof. From [Ros96] we borrow a function $s : \Lambda^n \mapsto \mathbb{N}$, s.t. $s(e_1) > s(e_2)$ if $e_1 \twoheadrightarrow_{\mathbb{(}\,\mathbb{)}} e_2$:

$$
\begin{aligned}
s(x^n) &= 1 \\
s(n) &= 1 \\
s(@e_1\ e_2) &= s(e_1) + s(e_2) + 1 \\
s(\lambda x.e) &= s(e) + 1 \\
s(e\mathbb{(}\langle x^n, v\rangle\mathbb{)}) &= s(e) \times (s(v) + 1)
\end{aligned}
$$

□

Proposition 3.6 (local confluence). $\rightarrow_{\mathbb{(}\,\mathbb{)}}$ *is locally confluent*

Proof. There are no critical pairs. □

Lemma 3.7 (Confluence of $\rightarrow_{\mathbb{(}\,\mathbb{)}}$). *The reduction $\rightarrow_{\mathbb{(}\,\mathbb{)}}$ is confluent.*

Proof. Follows from Proposition 3.5 and 3.6 by Newman's lemma [BN98, New42]. □

Proposition 3.8 (unique normal form). $\rightarrow_{\mathbb{(}\,\mathbb{)}}$ *has unique normal forms*

Proof. Follows from confluence (Lemma 3.7). □

Proposition 3.9 (Normal forms of $\rightarrow_{\mathbb{(}\,\mathbb{)}}$). *The normal forms $\underline{e}$ of $\rightarrow_{\mathbb{(}\,\mathbb{)}}$ are the terms without evaluation substitutions:*

$\underline{e} \in \underline{\Lambda^n} \subset \Lambda^n$
$\underline{v} \in \underline{\mathit{Values}^n} \subset \mathit{Values}^n$

$$
\begin{array}{ll}
\underline{e} & ::= x^n \mid \underline{v} \mid (@\ \underline{e}\ \underline{e}) \\
\underline{v} & ::= a \mid (\lambda x.\underline{e})
\end{array}
$$

Proof. $\twoheadrightarrow_{\mathbb{(}\,\mathbb{)}}$ is applicable to any term that contains an evaluation substitution and moves the substitution into the subterms or eliminates it for the base cases. □

For the normal forms, we can define a set of contexts $\underline{C^n}$ analogous to C:

$$\underline{C^n} \quad ::= [\,] \mid (@\ \underline{C^n}\ \underline{e}) \mid (@\ \underline{e}\ \underline{C^n}) \mid (\lambda x.\underline{C^n})$$

A second set of contexts C^n will be used later to specify evaluation. These contexts do not place the hole under λ-abstractions or under evaluation substitutions:

$$C^n \quad ::= [\,] \mid (@\ C^n\ e) \mid (@\ e\ C^n)$$

Both restrictions are necessary to ensure well-formedness of reduced terms. Like the limitation of evaluation substitutions to substitute only closed values, this is a profound deviation from other calculi with explicit substitutions [ACCL91]. Additional operators to manipulate the levels of identifiers are required to permit such restrictions. The reason for the deviation is that we are mainly interested in a semantics that describes actual implementations, not in a powerful calculus.

The function to determine the free variables of an expression e, $FV^n(e)$, returns the names of the variables, not the variables themselves. The function FV^n relies on a helper function $FV^{n\prime}$ that takes an additional level argument indicating the minimum level of unbound identifiers:

$$\begin{aligned}
FV^n &: \Lambda^n \to \mathcal{P}(\mathit{Vars}) \\
FV^n(e) &= FV^{n\prime}(e, 0) \\
FV^{n\prime} &: \Lambda^n \times \mathbb{N} \to \mathcal{P}(\mathit{Vars}) \\
FV^{n\prime}(a, m) &= \emptyset \\
FV^{n\prime}(x^n, m) &= \{x\} \text{ if } n \geq m \\
FV^{n\prime}(x^n, m) &= \emptyset \text{ if } n < m \\
FV^{n\prime}((\lambda x.e), m) &= FV^{n\prime}(e, m+1) \\
FV^{n\prime}((@e_1\ e_2), m) &= FV^{n\prime}(e_1, m) \cup FV^{n\prime}(e_2, m) \\
FV^{n\prime}(e\mathbb{C}\langle x^n, v\rangle\mathbb{D}, m) &= FV^{n\prime}(e', m) \text{ where } e\mathbb{C}\langle x^n, v\rangle\mathbb{D} \to_{\mathbb{C}\mathbb{D}} e'
\end{aligned}$$

For the bound variables, the function $BV^n(e)$ likewise returns a set of variable names:

$$\begin{aligned}
BV^n &: \Lambda^n \to \mathcal{P}(\mathit{Vars}) \\
BV^n(a, m) &= \emptyset \\
BV^n(x^n) &= \emptyset \\
BV^n((\lambda x).e) &= BV^n(e) \cup \{x\} \\
BV^n((@e_1\ e_2)) &= BV^n(e_1) \cup BV^n(e_2)
\end{aligned}$$

The basic notion of reduction introduces an expression with an evaluation substitution:

$$(@\ (\lambda x.e)\ v) \to_{\beta_v^n} e\mathbb{C}\langle x^0, v\rangle\mathbb{D} \text{ iff } FV^n(v) = \emptyset \qquad (\beta_v^n)$$

Unlike the classical λ_v-calculus, we assume the argument to be closed as our evaluation substitutions may only substitute closed values.

Together with δ, β_v^n forms the notion of reduction $\to_v^n$:

$$\to_v^n = \delta \cup \beta_v^n$$

The *one-step reduction* $\twoheadrightarrow_v^n$ is the compatible closure of $\to_v^n$:

$$e \twoheadrightarrow_v^n e' \text{ if } e \equiv C^n[e_1], e' \equiv C^n[e_2], e_1 \to e_2$$

Definition 3.10 ($\lambda_v^{n,\mathbb{C}\mathbb{D}}$). *Let $\twoheadrightarrow_v^{n,\mathbb{C}\mathbb{D}} = \twoheadrightarrow_v^n \cup \twoheadrightarrow_{\mathbb{C}\mathbb{D}}$ and $\twoheadrightarrow_v^{n,\mathbb{C}\mathbb{D}}$ its reflexive, transitive closure. $=_v^{n,\mathbb{C}\mathbb{D}}$ is the smallest equivalence relation generated by $\twoheadrightarrow_v^{n,\mathbb{C}\mathbb{D}}$. If $e_1 =_v^{n,\mathbb{C}\mathbb{D}} e_2$, we write $\lambda_v^{n,\mathbb{C}\mathbb{D}} \vdash e_1 = e_2$.*

□

We now establish the fact that the $\lambda_v^{n,\mathbb{C}\mathbb{D}}$ calculus is a conservative extension of (a variant of) the λ_v calculus. The proof follows the strategy of [ACCL91, Ros96], which in turn use Hardin's interpretation method. It is simplified by the fact that reduction does not take place within explicit substitutions.

To relate the $\lambda_v^{n,\mathbb{C}\mathbb{D}}$ calculus with levels with the ordinary λ_v calculus, we first need a translation that produces Λ^n terms from ordinary Λ terms. Curien [Cur96] defines a translation for de Bruijn's indices, which we can easily adapt to our notation with names. We first define the functions $\mathit{Outer}V$ and $\underline{\mathit{Outer}V}$ that derive for a context the sequence of bound variables from the hole to the root:

Definition 3.11 (*OuterV*).

$$\begin{aligned} \mathit{OuterV} &: C \to \mathit{Vars}^* & \underline{\mathit{OuterV}} &: \underline{C^n} \to \mathit{Vars}^* \\ \mathit{OuterV}([\,]) &= \epsilon & \underline{\mathit{OuterV}}([\,]) &= \epsilon \\ \mathit{OuterV}(\lambda x.C) &= \mathit{OuterV}(C) : x & \underline{\mathit{OuterV}}(\lambda x.\underline{C^n}) &= \underline{\mathit{OuterV}}(\underline{C^n}) : x \\ \mathit{OuterV}(C\ M) &= \mathit{OuterV}(M) & \underline{\mathit{OuterV}}(\underline{C^n}\ M) &= \underline{\mathit{OuterV}}(M) \\ \mathit{OuterV}(M\ C) &= \mathit{OuterV}(M) & \underline{\mathit{OuterV}}(M\ \underline{C^n}) &= \underline{\mathit{OuterV}}(M) \end{aligned}$$

□

Now the translation of every subterm of a Λ term can be defined by dividing the term into a context and the subterm:

Definition 3.12 (De Bruijn's translation). *For $N \equiv C[M] \in \Lambda$ s.t. $FV(N) = \emptyset$, we define the de Bruijn translation of M as $\phi(M, \mathit{OuterV}(C))$ where ϕ is defined as:*

$$\begin{aligned} \phi &: \Lambda \times \mathit{Vars}^* \to \Lambda^n \\ \phi(a, s) &= a \\ \phi(x, (x_1, \ldots, x_n)) &= x^i \text{ if } i \in \{1, \ldots, n\} \text{ is minimum s.t. } x = x_i \\ \phi((\lambda x.M), (x_1, \ldots, x_n)) &= (\lambda x.\phi(M, (x, x_1, \ldots, x_n))) \\ \phi((M\ N), (x_1, \ldots, x_n)) &= (@\ \phi(M, (x_1, \ldots, x_n))\ \phi(N, (x_1, \ldots, x_n))) \end{aligned}$$

A translation in the opposite direction must introduce fresh names for every variable because the names in Λ^n do not account for shadowing. For example, the translation of the Λ^n-term $\lambda x.\lambda x.x^1$ to the Λ term $\lambda x.\lambda x.x$ is obviously flawed. Hence, the translation function τ chooses a fresh variable name for every bound variable. Consequently, it needs to remember the new names using a sequence of names that reflects the bound variables of the surrounding terms. Translation can only happen for terms not containing explicit substitutions.

Definition 3.13 (Reverse de Bruijn translation). *For $\underline{e'} \equiv \underline{C^n}[\underline{e}] \in \underline{\Lambda^n}$ s.t. $FV^n(\underline{e'}) = \emptyset$, we define the reverse de Bruijn translation of $\underline{e}$ as $\tau(\underline{e}, \underline{\mathit{OuterV}}(\underline{C^n}))$, where τ is defined as:*

$$\begin{aligned} \tau &: \Lambda^n \times \mathit{Vars}^* \to \Lambda \\ \tau(a, s) &= a \\ \tau(x^i, (x_1, \ldots, x_n)) &= x_i \\ \tau((\lambda x.\underline{e}), s) &= \lambda z.\tau(\underline{e}, z : s) \text{ where } z \text{ fresh} \\ \tau((@\underline{e_1}\ \underline{e_2}), s) &= (\tau(\underline{e_1}, s)\ \tau(\underline{e_2}, s)) \end{aligned}$$

and no name occurs twice in $\underline{\mathit{OuterV}}(\underline{C^n})$.

Lemma 3.14. *ϕ is a bijection from Λ to Λ^n:*[2]

1. *For $\underline{e'} \equiv \underline{C^n}[\underline{e}] \in \underline{\Lambda^n}$ s.t. $FV^n(\underline{e'}) = \emptyset$: $\phi(\tau(\underline{e}, \underline{\mathit{OuterV}}(\underline{C^n})), \underline{\mathit{OuterV}}(\underline{C^n})) = \underline{e}$.*
2. *For $N \equiv C[M] \in \Lambda$ s.t. $FV(N) = \emptyset$: $\tau(\phi(M, \mathit{OuterV}(C)), \mathit{OuterV}(C)) = M$.*

Proof.

1. Induction on the structure of $\underline{e}$:
   - $\underline{e} = x^i : \phi(\tau(x^i, s), s) = \phi(x_i) = x^i$, where $i \in s = \{1, \ldots, n\}$ is minimum s.t. $x = x_i$
   - $\underline{e} = (@\underline{e_1}\ \underline{e_2})$: Obvious from the induction hypothesis.

[2] Note that we identify terms that are equal modulo the names of the bound variables.

- $\underline{e} = (\lambda x.\tilde{e}) : \phi(\tau((\lambda x.\tilde{e}), s), s) = \phi((\lambda z.\tau(\tilde{e}, z : s)), s) = (\lambda z.\phi(\tau(\tilde{e}, z : s), z : s)$. Now using $\underline{e'} \equiv \tilde{C}[\tilde{e}], \tilde{C} \equiv C[\lambda z.[\ ]]$, we can apply the induction hypothesis and obtain $(\lambda z.e')$.

2. Analogous to the first case.

□

Corollary 3.15. *τ is a bijection from Λ^n to Λ.*

The next lemma shows that the result of applying ϕ or τ to a value is independent of the context:

Lemma 3.16 (ϕ for closed values). *1. $\forall s, s' : \phi(V, s) = \phi(V, s')$ if $FV(V) = \emptyset$*

2. $\forall s, s' : \phi(v, s) = \rho(v, s')$ if $FV(v) = \emptyset$

Proof. The only interesting case is $V = \lambda x.M, FV(M) = \{x\}$. Without loss of generality, $M = x$. But then $\phi(\lambda x.x, s) = \lambda x.\phi(x, x : s) = \lambda x.x^0 = \lambda x.\phi(x, x : s') = \phi(\lambda x.x, s')$. □

Now we show that ϕ translates a term subject to a meta-substitution into a term subject to an evaluation substitution.

Lemma 3.17 (Equivalence of substitutions for ϕ). *For $N \equiv C[M[V/x]] \in \Lambda$ s.t. $FV(N) = \emptyset$:*

$$\phi(M[V/x], (x_0, \ldots, x_n)) =_{\mathbb{C}\mathbb{D}} \phi(M, (x_0, \ldots, x_n))\mathbb{C}\langle x^i, \phi(V, (x_0, \ldots, x_n))\rangle\mathbb{D}$$

where $FV(V) = \emptyset$, $OuterV(C) = (x_0, \ldots, x_n)$, $i =$ minimum s.t. $x_i = x$.

Proof. Induction on the structure of M:

- $M = x$:
  Left-hand side:
  $$\phi(x[V/x], s) = \phi(V, s)$$
  Right-hand side:
  $$\phi(x, s)\mathbb{C}\langle x^i, \phi(V, s)\rangle\mathbb{D} = x^i\mathbb{C}\langle x^i, \phi(V, s)\rangle\mathbb{D} \overset{EvalSubstId}{=} \phi(V, s)$$

- $M = y, y \neq x$:
  Left-hand side:
  $$\phi(y[V/x], s) = \phi(y, s) = y^j$$
  Right-hand side:
  $$\phi(y, s)\mathbb{C}\langle x^i, \phi(V, s)\rangle\mathbb{D} = y^j\mathbb{C}\langle x^i, \phi(V, s)\rangle\mathbb{D} \overset{\text{EvalSubstIdOther}}{=} y^j$$

- $M = (M_1\ M_2)$:
  Left-hand side:
  $$\begin{array}{rcl} \phi((M_1\ M_2)[V/x], s) & = & (\phi(M_1[V/x], s)\ \phi(M_2[V/x], s)) \\ & \overset{\text{IH}}{=} & (\phi(M_1, s)\mathbb{C}\langle x^i, \phi(V, s)\rangle\mathbb{D}\ \phi(M_2, s)\mathbb{C}\langle x^i, \phi(V, s)\rangle\mathbb{D}) \end{array}$$
  Right-hand side:
  $$\begin{array}{rcl} \phi((M_1\ M_2), s)\mathbb{C}\langle x^i, \phi(V, s)\rangle\mathbb{D} & = & (\phi(M_1, s)\ \phi(M_2, s))\mathbb{C}\langle x^i, \phi(V, s)\rangle\mathbb{D} \\ & \overset{EvalSubstApp}{=} & (\phi(M_1, s)\mathbb{C}\langle x^i, \phi(V, s)\rangle\mathbb{D} \\ & & \phi(M_2, s)\mathbb{C}\langle x^i, \phi(V, s)\rangle\mathbb{D}) \end{array}$$

- $M = (\lambda y.K)$:
  Without loss of generality, $x \neq y$ (otherwise rename y).
  Left-hand side:

$$\begin{array}{rcl} \phi((\lambda y.K)[V/x], s) & \overset{FV(V)=\emptyset}{=} & \phi((\lambda y.K[V/x]), s) \\ & = & \lambda y.\phi(K[V/x], y:s) \\ & \overset{\text{IH}}{=} & \lambda y.\phi(K, y:s)(\!|\langle x^{i+1}, \phi(V, x:s)\rangle|\!) \end{array}$$

  Right-hand side:

$$\begin{array}{rcl} \phi(\lambda y.K, s)(\!|\langle x^i, \phi(V,s)\rangle|\!) & = & (\lambda y.\phi(K, y:s))(\!|\langle x^i, \phi(V,s)\rangle|\!) \\ & \overset{\text{EvalSubstLam}}{=} & \lambda y.\phi(K, y:s)(\!|\langle x^{i+1}, \phi(V,s)\rangle|\!) \end{array}$$

  By Lemma 3.16, $\phi(V, s) = \phi(V, x:s)$.

□

A similar result as in the previous lemma holds for the translation of a Λ^n term subject to an evaluation substitution into a Λ term via ρ:

Lemma 3.18 (Equivalence of substitutions for τ). *For $e' \equiv \underline{C^n}[\underline{e}(\!|\langle x^i, \underline{v}\rangle|\!)] \in \Lambda^n$ s.t. $FV(e) = \emptyset$:*

$$\tau(\underline{e}(\!|\langle x^i, \underline{v}\rangle|\!), (x_0, \ldots, x_n)) = \tau(\underline{e}, (x_0, \ldots, x_n))[\tau(\underline{v}, (x_0, \ldots, x_n))/x]$$

where $FV(v) = \emptyset$, $\underline{OuterV(C^n)} = (x_0, \ldots, x_n)$, i = minimum s.t. $x_i = x$.

Proof. Analogous to the proof of Lemma 3.17. □

As $\rightarrow_{\beta_v^n}$ accepts only closed values as operands, we can only state equivalence of λ_v^n with a λ_v calculus where the $\rightarrow_\beta$ reduction has the same limitation. We call this reduction $\rightarrow_{\beta_{v-\text{closed}}}$.

Lemma 3.19 (Simulation for $\rightarrow_{\beta_{v\text{-closed}}}$). *For $N \equiv ((\lambda x.M)\, V) \in \Lambda$ s.t. $FV(N) = \emptyset$, $V \in$ Values, $FV(V) = \emptyset$: If $((\lambda x.M)\ V) \rightarrow_{\beta_{v\text{-closed}}} L$, then $\phi(((\lambda x.M)\ V), \epsilon) \rightarrow_{\beta_v^n} e$, s.t. $e =_{(\!|\,|\!)} \phi(L, x:\epsilon)$*

Proof.

$$\begin{array}{rcl} \phi(L, x:\epsilon) & = & \phi(M[V/x], x:\epsilon) \\ & \overset{\text{Lemma 3.17}}{=_{(\!|\,|\!)}} & \phi(M, x:\epsilon)(\!|\langle x^0, \phi(V, \epsilon)\rangle|\!) \end{array}$$

$$\begin{array}{rcl} \phi(((\lambda x.M)\ V), \epsilon) & = & (\phi(\lambda x.M, \epsilon)\ \phi(V, \epsilon)) \\ & = & ((\lambda x.\phi(M, x:\epsilon))\ \phi(V, \epsilon)) \\ & \rightarrow_{\beta_v^n} & \phi(M, x:\epsilon)(\!|\langle x^0, \phi(V, x:\epsilon)\rangle|\!) \\ & \overset{\text{Lemma 3.16}}{=} & \phi(M, x:\epsilon)(\!|\langle x^0, \phi(V, x)\rangle|\!) \end{array}$$

□

As the contexts of Λ_n do not include holes under λ-abstractions, we further need to limit the contexts of the λ_v calculus accordingly to C_v. We call the resulting one-step reduction $\rightarrow\!\!\triangleright_{v\text{-closed}}$.

Lemma 3.20 (Simulation for $\rightarrow\!\!\triangleright_{\beta_{v\text{-closed}}}$). *For $M, N \in \Lambda, FV(M) = \emptyset$, $V \in$ Values, $FV(V) = \emptyset$: If $M \rightarrow\!\!\triangleright_{\beta_{v\text{-closed}}} N$, then $\phi(M, \epsilon) \rightarrow\!\!\triangleright_{v^n} e$, s.t. $e =_{(\!|\,|\!)} \phi(N, x:\epsilon)$.*

Proof. The translated term does not contain evaluation substitutions, hence we can choose to reduce the same redex and use Lemma 3.19. The lemma applies since C_v does not include λ-abstractions and hence $\rightarrow$ is never applied under a λ (and thus ϕ's second argument is always ϵ). □

Next, we prove that reductions in λ_v^n yield the same results as λ_v reductions.

Lemma 3.21 (Simulation for $\rightarrow_{\beta_v^n}$). *For $e' = (@(\lambda x.e)\ v) \in \Lambda^n$ and $v \in Values^n$: if $(@(\lambda x.e)\ v) \rightarrow_{\beta_v^n} e'$ then $\tau(\underline{(@(\lambda x.e)\ v)}, \epsilon) \rightarrow_{\beta_v\text{-}closed} N$ s.t. $N = \tau(\underline{e'}, z : \epsilon)$.*

Proof.
Left-hand side:

$$(@(\lambda x.e)\ v) \quad \rightarrow_{\beta_v^n} \quad e\langle\!(\langle x^0, v\rangle)\!\rangle = e'$$

Right-hand side:

$$\begin{array}{rll}
\tau(\underline{(@(\lambda x.e)\ v)}, \epsilon) & = & \tau((@\underline{(\lambda x.e)}\ \underline{v}), \epsilon) \\
& = & (\tau(\underline{(\lambda x.e)}, \epsilon)\ \tau(\underline{v}, \epsilon)) \\
& = & (\tau((\lambda x.\underline{e}), \epsilon)\ \tau(\underline{v}, \epsilon)) \\
& = & ((\lambda z.\tau(\underline{e}, z : \epsilon))\ \tau(\underline{v}, \epsilon))\ z \text{ fresh} \\
& \rightarrow_{\beta_v\text{-closed}} & \tau(\underline{e}, z : \epsilon)[\tau(\underline{v}, \epsilon)/z] \\
& \overset{Lemma\ 3.16}{=} & \tau(\underline{e}, z : \epsilon)[\tau(\underline{v}, z : \epsilon)/z] \\
& \overset{Lemma\ 3.18}{=} & \tau(\underline{e}\langle\!(\langle x^0, \underline{v}\rangle)\!\rangle, z : \epsilon)
\end{array}$$

□

Lemma 3.22 (Simulation for $\rightarrow\!\!\triangleright_{\beta_v^n}$). *For $e, e' \in \Lambda^n$ and $FV^n(e) = \emptyset$: if $e \rightarrow\!\!\triangleright_{\beta_v^n} e'$ then $\tau(\underline{e}) \rightarrow\!\!\triangleright_{\beta_v\text{-}closed} N$ s.t. $N = \tau(e', \epsilon)$.*

Proof. As reductions do not take place under explicit substitutions, we can chose to reduce the same redex and use Lemma 3.21. □

Theorem 3.23 (Equivalence). *For $M, N \in \Lambda$, $FV(M) = \emptyset$:*

1. *If $\phi(M, \epsilon) \rightarrow\!\!\triangleright_v^{n,(\!(\,)\!)*} e$ then $M \rightarrow\!\!\triangleright_v^* N$, s.t. $\phi(N, \epsilon) =_{(\!(\,)\!)} e$.*
2. *If $M \rightarrow\!\!\triangleright_{\beta_{v\text{-}closed}}^* N$, then $\phi(M, \epsilon) \rightarrow\!\!\triangleright_{\beta_v^n}^* e$, s.t. $e =_{(\!(\,)\!)} \phi(N, x : \epsilon)$.*

Proof. 1. Induction on the length of the $\rightarrow\!\!\triangleright_{\beta_{v\text{-closed}}}$ reduction, use Lemma 3.20 in each step.

2. Induction on the length of the reduction. In each step, either $\rightarrow\!\!\triangleright_{\beta_v^n}$ or $\rightarrow\!\!\triangleright_{(\!(\,)\!)}$ reduces the term. In the first case, Lemma 3.21 relates the two terms, in the second case, the confluence of $\rightarrow\!\!\triangleright_{(\!(\,)\!)}$ does.

□

Corollary 3.24. *The reduction $\rightarrow\!\!\triangleright_v^{n,(\!(\,)\!)}$ is confluent.*

Proof. Follows from Hardin's interpretation method using $\rightarrow\!\!\triangleright_{(\!(\,)\!)}$ as projection. □

Proposition 3.25. *The reduction $\rightarrow\!\!\triangleright_v^{n,(\!(\,)\!)}$ preserves strong normalization.*

Proof. Reductions do not take place within explicit substitutions, hence no additional β-redexes emerge. □

3.4 Parsing Scheme Without Macros

The previous section added identifiers with levels to the classical lambda calculus to include a tool for tracking the binding place of a variable without using renaming techniques. However, the identifiers in usual programming languages are merely symbols. It is the job of the compiler to relate the variables to their binding places using the rules of lexical scoping. To mimic this in Λ^n, we define an s-expression-based source language with ordinary symbol-based variables and define a parsing reduction to translate it to Λ^n. The next section adds macro expansion to the source language.

As for most s-expression-based languages, the parser is straightforward: either the s-expression is an atom or the first element of a parenthesized expression determines the syntactic role of the expression. The parser then recursively descends into the sub-expressions. To map identifiers from the source code to identifiers with levels, the parser generates explicit substitutions that replace symbols by identifiers with levels. The parser pushes these substitutions inwards and maintains the level of the identifier. Whenever the explicit substitution enters a λ-abstraction, the parser must increment the level of the identifier. This usage of explicit substitutions differs completely from the technique in the previous section where the semantics uses explicit substitutions to relate the evaluation in the calculus to actual implementations. The primary motivation for using explicit substitutions in the parser is the possibility to obtain a concise description of hygienic macro expansion. The macro expander will use its own set of explicit substitutions. Because parsing and macro expansion must be interleaved the parser uses explicit substitutions as well.

The source language terms are s-expressions, which consist of constants, symbols, and parenthesized forms. Figure 3.4 contains the definition of this language.

Syntactic Domains:
$n \in Const$
$\mathtt{x} \in Symbols$
$se \in S\text{-}Expressions$
Concrete Syntax:

$se ::= n \mid \mathtt{x} \mid (se\ \ldots)$

Figure 3.4: Concrete Syntax based on s-expressions

The parser transforms an s-expression into a term of Λ^n by descending into sub-expressions. Just before such a descent, the input of the parser is a mixture of concrete and abstract syntax: the parser has already parsed the context and generated the abstract syntax for it but the sub-expressions are still concrete syntax. For example, for the s-expression `(lambda x (x x))`, the parser creates in its first step the term $(\lambda x.(\mathtt{x}\ \mathtt{x}))$, which is an abstract-syntax λ-abstraction with a concrete syntax body. In the next step, the parser will descend into the body and turn it into an abstract-syntax procedure application. Figure 3.5 describes the set *MixtureTerms* that describes this mixture of abstract and concrete syntax called *mixture syntax*. The mixture syntax uses the term ${}^{ks}x^n$ to represent an identifier. The macro expander uses the additional index *ks* to track the occurrences of identifiers. This index will be explained below in Section 3.5, for now this index can be safely ignored and we will often omit it. Translating the result of the parser to Λ^n works by dropping the index *ks* from all identifiers.

As a minor extension of Λ^n, applications in the mixture syntax are not limited to one argument expressions but allow an arbitrary number of arguments. Section 3.6 uses these multi-argument applications to describe evaluation of a new binding construct with multiple parameters. However, λ-abstractions itself remain unary in the mixture syntax and in Λ^n.

To keep track of the binding place of identifiers, the parser also adds *parsing substitutions* to the mixture. Parsing substitutions are written as $(\!|r|\!)$ and they extend the set of terms. That is, parsing substitutions are the explicit substitutions that replace symbols by identifiers with labels as sketched in the introduction of this section. A parsing substitution replaces a source-code

identifier (a symbol) by a labeled identifier, and is written as ${}^{ks}x^{n}/\mathtt{x}$. S-expressions with parsing substitutions are called *lexical s-expressions* and receive an extra domain *Lex-S-Expressions* to make the evolving presentation more precise (Without *Lex-S-Expressions*, every s-expressions with a parsing substitution is a member of set *MixtureTerms*). The variable *ses* ranges over lexical s-expressions. The parser uses the set *Specials* containing two identifiers λ and $\star$. These identifiers are members of *Vars* but cannot be present in the source code, i.e.

$$\mathit{Specials} \cap \mathit{Symbols} = \emptyset$$

The parser uses the special identifier λ to keep track of uses of the `lambda` keyword. The $\star$ identifier assigns unbound variables the correct level for global identifiers as it matches any symbol. The parser achieves this tracking by initially applying the two parsing substitutions $(\!|\lambda^{0}/\mathtt{lambda}|\!)(\!|\star/\star^{0}|\!)$ to its input term.

Syntactic Domains:

c	$\in \mathit{MixtureTerms}$
ks	$\in \mathcal{P}(\mathbb{N})$
ses	$\in \mathit{Lex\text{-}S\text{-}Expressions}$
	$\mathit{Specials} = \{\lambda, \star\}$
x	$\in \mathit{Vars} = \mathit{Symbols} \cup \mathit{Specials}$
n	$\in \mathbb{N}$
${}^{ks}x^{n}$	$\in \mathit{Ids}$
r	$\in \mathit{ParsingSubsts}$

Mixture syntax for parsing:

r	$::= {}^{ks}x^{n}/\mathtt{x}$
ses	$::= se \mid ses(\!|r|\!)$
c	$::= se \mid a \mid {}^{ks}x^{n} \mid (\lambda\ x.c) \mid (@\ cc\ldots) \mid c(\!|r|\!)$

Figure 3.5: Mixture of abstract and concrete syntax

With the concrete syntax and the mixture syntax in place, the definition of the parser is possible. There are two tasks to perform. First, the parser turns concrete syntax into mixture syntax until it reaches the base expressions and the generated mixture syntax consists of abstract-syntax forms only. For every λ-abstraction, the parser generates a parsing substitution that replaces the symbol—representing the bound variable—by an identifier whose level refers to the λ-abstraction. For a parsing substitution to take effect, it needs to be *eliminated*, which means that the parser pushes the substitutions inwards the generated mixture syntax until the substitution meets a symbol and turns it into an identifier. This elimination of parsing substitutions is the second task to perform during parsing. There is one reduction for each task:

- The parsing reduction $\rightarrow_{\mathrm{Parse}}$ turns concrete syntax into mixture syntax.
- The reduction $\rightarrow_{(\!|\,|\!)}$ eliminates parsing substitutions.

The macro expander and the macro transformer introduce other explicit substitutions that serve similar purposes as the parsing substitutions and the parser and the macro expander need to eliminate these substitutions analogously to the parsing substitution. To avoid redefining the parser and the expander during the introduction of the other substitutions, the parser and the expander use the reduction $\rightarrow_{\mathrm{El}}$ for the elimination of all explicit substitutions. The reduction $\rightarrow_{\mathrm{El}}$ is defined as the union of the individual elimination reductions. For now, it eliminates only parsing substitutions:

$$\rightarrow_{\mathrm{El}} = \rightarrow_{(\!|\,|\!)}$$

Later we will redefine $\rightarrow_{\mathrm{El}}$ to contain the other substitutions as well.

Next, it is also necessary to specify where and when the elimination takes place. Elimination should not happen if the term subject to the substitution is a compound s-expression as this

$$\rightarrow_{\text{Parse}} \subseteq \textit{Lex-S-Expressions} \times \textit{MixtureTerms}$$

$$c \rightarrow_{\text{Parse}} c' \text{ iff } c \neq c' \text{ where } c \mapsto^*_{\text{El}}= c' \qquad \text{(ElSubstSym)}$$

$$P[((se_{1_1}\ se_{1_2}\ \ldots)se_2\ \ldots)] \rightarrow_{\text{Parse}} (@\ P[(se_{1_1} se_{1_2} \ldots)]\ P[se_2]\ldots) \qquad \text{(NestedAppSimple)}$$

$$P[(\texttt{x}\ \texttt{y}\ se)] \rightarrow_{\text{Parse}} P[(\lambda \texttt{y}.se(\!|^{\emptyset}\texttt{y}^0/\texttt{y}|\!))] \text{ iff } P[\texttt{x}] \mapsto^*_{\text{El}} \lambda^n \qquad \text{(LambdaIdSimple)}$$

$$P[(\texttt{x}\ se_2\ \ldots)] \rightarrow_{\text{Parse}} P[(@\ \texttt{x}\ se_2\ldots)] \text{ iff } x \notin \textit{Specials} \text{ where } P[\texttt{x}] \mapsto^*_{\text{El}}= {}^{ks}x^n \qquad \text{(IdAppSimple)}$$

Figure 3.6: Parsing reduction without macros

s-expression has not been parsed and its syntactic role is therefore not known. As an example, consider the term $(\texttt{lambda x x})(\!|\lambda^0/\texttt{lambda}|\!)(\!|x^2/\texttt{x}|\!)$. If elimination would push the inner substitution to the s-expression, the result would be $(\lambda^0\ \texttt{x x})(\!|x^2/\texttt{x}|\!)$: the substitution replaces the symbol `lambda` by the special identifier λ^0 but does not affect the two symbols `x`. If we ignore for a moment that this term is not a member of *MixtureTerms*, we may continue with elimination. That is, we will again eliminate a parsing substitution applied to a compound s-expression. This yields the term $(\lambda^0\ x^2\ x^2)$. However the level of the identifier x^2 in the body is wrong, as x is a local variable with level 0 in the body of a λ-abstraction. However, the fact that $(\lambda^0\ x^2\ x^2)$ is a λ-abstraction is only known after parsing. We could let the parser fix the level of the identifiers in the body as it turns the s-expression into an abstract-syntax term but it is also possible to avoid the generation of the spurious identifiers in the first place by not eliminating parsing substitutions applied to compound s-expressions. Instead, the parsing substitutions stick at compound s-expressions until the parser turns the s-expression into abstract syntax. Then, however, all accumulated substitutions have to be eliminated. The innermost substitution arrived first at the term and hence it is natural to eliminate it first. The context *EL* selects this innermost substitution from a cascade of parsing substitutions:

$$EL \quad ::= EL(\!|t|\!) \mid [\,]$$

Using this context, we define the standard elimination reduction

$$c \mapsto_{\text{El}} c' \text{ iff } c = EL[c_1], c' = EL[c_2] \text{ for some context } EL \wedge c_1 \rightarrow_{\text{El}} c_2$$

and let $\mapsto^*_{\text{El}}$ denote the reflexive, transitive closure of $\mapsto_{\text{El}}$. As we add new elimination reductions to $\rightarrow_{\text{El}}$, we will also extend *EL* accordingly to ensure that it still selects the innermost substitution.

Next, we define the parsing reduction $\rightarrow_{\text{Parse}}$ and afterwards $\rightarrow_{(\!|\,|\!)}$ for the elimination of parsing substitutions.

The parsing reduction from Figure 3.6 translates from concrete syntax to the abstract syntax from Section 3.3. The description of the parsing reduction uses *parsing contexts* defined as[3]:

$$P \quad ::= P(\!|r|\!) \mid [\,]$$

Parsing contexts enable the parser to "peek" under the outer parsing substitutions. This is necessary as we do not eliminate parsing substitutions applied to compound s-expressions. For example the term $(\texttt{x 3})(\!|x^0/\texttt{x}|\!)(\!|y^0/\texttt{y}|\!)$ can be written as $P[\texttt{x 3}]$ where P is the parsing context $[\,](\!|x^0/\texttt{x}|\!)(\!|y^0/\texttt{y}|\!)$. The encoding $P[\texttt{x 3}]$ makes it obvious that the term is a compound expression. The parsing context contains information on the meaning of the symbol `x`.

The parsing reduction works as follows: The rule (ElSubstSym) eliminates a parsing substitution, if possible. To that end, it uses the $\mapsto^*_{\text{El}}$ reduction. The parser does not contain a rule for symbols as the elimination of parsing substitutions turns them into identifiers. (Remember that the parsing substitution $(\!|\star/\star^n|\!)$ matches any symbol.) Also, the concrete syntax and the abstract

[3]The context P is identical to the context *EL* but the two serve different purposes and hence receive separate definitions.

syntax use the same representation for constants so the parser does not have to handle them and the remaining rules cover compound s-expressions surrounded by parentheses.

The first element of a compound s-expression determines its syntactic role. If it is another compound s-expression, the outer s-expression must be an procedure application. The rule (NestedAppSimple) covers this case. Otherwise, the first element has to be an identifier, which is represented by a symbol in the concrete syntax. The rules (LambdaIdSimple) and (IdAppSimple) use the elimination for explicit substitutions $\rightarrow_{\mathrm{El}}$ to turn the symbol into an identifier and recognize the correct syntax of the compound expression. In the rule (IdAppSimple) this identifier is a variable, hence the whole expression is a procedure application. Note that this rule turns a symbol, written `x`, into an identifier, written ${}^{ks}x^{n}$, where x is the name of the identifier. If the identifier at the head is the special identifier λ^n it corresponds to the syntactic keyword `lambda`. The rule (LambdaIdSimple) treats this case.[4] Here, the parser needs to generate a fresh parsing substitution to replace the symbol from the parameter position by an identifier with level 0.

To perform the second task of the parser—the elimination of parsing substitutions— a definition of the reduction $\rightarrow_{(\!|\,|\!)}$ is necessary. The elimination reduction $\rightarrow_{\mathrm{El}}$ is based on $\rightarrow_{(\!|\,|\!)}$ and we have just seen that the rule (ElSubstSym) uses this reduction to eliminate parsing substitutions applied to terms, and the rules (LambdaIdSimple) and (IdAppSimple) use it to turn the symbol in the head of a compound s-expression into an identifier. Figure 3.7 contains the rules of the $\rightarrow_{(\!|\,|\!)}$ reduction.

$$
\begin{aligned}
&\mathit{ParseSubstTerms} \ni c_{\mathrm{PS}} ::= c(\!|r|\!) \\
&\qquad\qquad \rightarrow_{(\!|\,|\!)} \subseteq \mathit{ParseSubstTerms} \times \mathit{MixtureTerms} \\
&\mathtt{x}(\!|{}^{ks}\star^{n}/\star|\!) \rightarrow_{(\!|\,|\!)} {}^{ks}\mathtt{x}^{n} && \text{(ParseSubstStar)} \\
&\mathtt{x}(\!|id/\mathtt{x}|\!) \rightarrow_{(\!|\,|\!)} id && \text{(ParseSubstSym)} \\
&a(\!|r|\!) \rightarrow_{(\!|\,|\!)} a && \text{(ParseSubstConst)} \\
&{}^{ks}x^{n}(\!|r|\!) \rightarrow_{(\!|\,|\!)} {}^{ks}x^{n} && \text{(ParseSubstId)} \\
&\mathtt{x}(\!|id/\mathtt{y}|\!) \rightarrow_{(\!|\,|\!)} \mathtt{x} \text{ iff } \mathtt{x} \neq \mathtt{y} \wedge \mathtt{y} \neq \star && \text{(ParseSubstSymOther)} \\
&(\lambda z.ses)(\!|{}^{ks}w^{n}/\mathtt{y}|\!) \rightarrow_{(\!|\,|\!)} (\lambda z.ses(\!|{}^{ks}w^{n+1}/\mathtt{y}|\!)) && \text{(ParseSubstLamId)} \\
&(@\ ses_1\ ses_2 \ldots)(\!|r|\!) \rightarrow_{(\!|\,|\!)} (@\ ses_1(\!|r|\!)\ ses_2(\!|r|\!) \ldots) && \text{(ParseSubstApp)}
\end{aligned}
$$

Figure 3.7: Reduction $\rightarrow_{(\!|\,|\!)}$ without macros

The elimination reduction works on the mixture syntax: The parsing reduction generates terms for λ-abstractions and applications but the representation of variables is turned from symbols to identifiers as the elimination of parsing substitutions proceeds. The rules of the elimination reduction behave as follows:

- Rule (ParseSubstStar) applies the substitution for unbound variables. It uses the level associated with the special identifier $\star$ to create an identifier from the symbol.
- Rule (ParseSubstSym) applies the parsing substitution to turn a symbol into an identifier.
- Rule (ParseSubstConst) drops the substitution applied to a constant.
- Dropping the substitution also happens in rule (ParseSubstId) if the substitution is applied to an identifier and in rule (ParseSubstSymOther) if the substitution is applied to a non-matching symbol.

[4] Unlike Scheme, our concrete syntax supports only unary `lambda` forms and the parameter is not parenthesized. Procedure application nevertheless allows an arbitrary number of arguments to support primitives with more than one argument.

- Rule (ParseSubstLamId) moves an explicit substitution into the body of an abstraction. This means that the identifier of the substitution is one level further away from its binding place; consequently the rule increments the level index of the identifier. Note that this preserves lexical scoping because the parser already generated a substitution binding the parameter of the abstraction when it generated the abstract syntax term for the abstraction in rule (LambdaIdSimple).
- For applications, rule (ParseSubstApp) simply needs to propagate the substitution to the operator and the operands.

For the parsing reduction, we define expansion/parsing contexts EP to uniquely identify the left most outer most source term that the parser needs to replace by abstract syntax next:

$$EP \quad ::= (@\ e \ldots EP c \ldots) \mid (\lambda\ x.EP) \mid [\,]$$

The expansion/parsing contexts give rise to a standard parsing reduction: $c_1 \mapsto_{\text{Parse}} c_2$, if for some expansion/parsing context EP, $c_1 \equiv EP[ses], c_2 \equiv EP[c], ses \to_{\text{Parse}} c$.

We define $\mapsto^*_{\text{Parse}}$ as the reflexive, transitive closure of $\mapsto_{\text{Parse}}$ and derive a function *parse*:

$$\begin{aligned} &\mathit{parse} : \textit{S-Expressions} \to \Lambda^n \\ &\mathit{parse}(se) = e \text{ iff } se(\!|\lambda^0/\texttt{lambda}|\!)(\!|\star^0/\star|\!) \mapsto^*_{\text{Parse}} e. \end{aligned}$$

The *parse* function maps an input s-expression M to a parsed expression e. The function annotates the input s-expression by two initial explicit substitutions for parsing:

- $(\!|\lambda/\texttt{lambda}|\!)$ binds the keyword `lambda` to the special symbol λ. The parser later uses this binding to recognize uses of the keyword.
- $(\!|\star/\star^0|\!)$ creates an explicit substitution that is used by the parser to assign the correct level to free identifiers.

Proposition 3.26. *The parse relation is a function.*

Proof. Follows immediately from the fact that the expansion/parsing contexts uniquely divide each source term into a context and a redex. □

From now on, we assume for all standard reductions that they reduce to a single term. The contexts will always be defined accordingly and the left-hand sides of the rules never overlap.

We conclude this section with an example of the parsing reduction. Assume *parse* is applied to the term `(lambda x (x y))`. Then the reduction proceeds as follows, where the tags on the right indicate the rule that applies and lists multiple rules if the parsing reduction invokes the elimination of parsing substitutions:

$$\begin{aligned}
&\texttt{(lambda x (x y))}(\!|\lambda^0/\texttt{lambda}|\!)(\!|\star/\star^0|\!) \to && \text{((LambdaIdSimple))}\\
&(\lambda x.\texttt{(x y)}(\!|x^0/\texttt{x}|\!))(\!|\lambda^0/\texttt{lambda}|\!)(\!|\star/\star^0|\!) \to \\
&\qquad \text{((ElSubstSym),(ParseSubstLamId),(ParseSubstLamId))}\\
&(\lambda x.\texttt{(x y)}(\!|x^0/\texttt{x}|\!)(\!|\lambda^1/\texttt{lambda}|\!)(\!|\star/\star^1|\!)) \to && \text{((IdAppSimple))}\\
&(\lambda x.((@\texttt{x y})(\!|x^0/\texttt{x}|\!)(\!|\lambda^1/\texttt{lambda}|\!)(\!|\star/\star^1|\!)))) \to \\
&\qquad \text{((ElSubstSym),(ParseSubstApp),(ParseSubstApp))}\\
&(\lambda x.(@(\texttt{x}(\!|x^0/\texttt{x}|\!)(\!|\lambda^1/\texttt{lambda}|\!)(\!|\star/\star^1|\!))(\texttt{y}(\!|x^0/\texttt{x}|\!)(\!|\lambda^1/\texttt{lambda}|\!)(\!|\star/\star^1|\!)))) \to \\
&\qquad \text{((ElSubstSym),(ParseSubstSym),(ParseSubstId))}\\
&(\lambda x.(@x^0(\texttt{y}(\!|x^0/\texttt{x}|\!)(\!|\lambda^1/\texttt{lambda}|\!)(\!|\star/\star^1|\!)))) \to \\
&\qquad \text{((ElSubstSym),(ParseSubstSymOther),(ParseSubstStar))}\\
&\qquad (\lambda x.(@x^0 y^1))
\end{aligned}$$

In the result, x^0 is a local variable bound by the surrounding λ whereas y^1 is unbound. The level of the initial parsing substitution $(\!|\star/\star^0|\!)$ has been incremented as the substitution enters the scope of the λ, hence `y` receives the correct level 1.

3.5 The Core Macro Expander

Having a basic parser for Λ^n we can now turn to the central point of this chapter: a formal description of hygienic macro expansion. We introduce the language Λ_M, an extension of Λ^n that encompasses macro expansion. The presentation splits macro expansion into two parts:

- The core macro expander collects macro definitions and moves them into the scope of λ-abstractions while maintaining hygiene.
- For the macro expansion proper, specifications for each kind of macro transformer extend the core macro expander. Such a specification must provide rules to parse the definition of the transformer, elimination rules for the abstract-syntax representation of the transformer, and, of course, an extension of the core macro expander that handles macro applications where the keyword is bound to a transformer of the respective kind.

This section only defines the core macro expander. Section 3.6 contains the specification for a computational macro transformer and Section 3.9 contains the specification for the `syntax-rules` transformer from R^{5}RS.

Adding macros to the language first requires a binding construct for macros and a syntactic form for macro applications. An identifier that is bound to a macro is commonly called a *keyword*. In the language Scheme three different binding constructs for keywords exist: `define-syntax` introduces a global, recursive definition for a keyword, `let-syntax` binds a keyword within a local scope and `letrec-syntax` binds a keyword locally and recursively. As the latter construct can be used to simulate the others, we choose to focus on it. However, to keep the presentation reasonably compact, we do not handle (mutually recursive) binding of several keywords. Instead we only outline how this feature would be handled.

In our restricted version of `letrec-syntax` the first argument is the identifier that is bound as the keyword, followed by a transformer and the body. The transformer may be a `syntax-rules` clause from R^{5}RS or some other form. The actual specification of the transformer form will be given in a later section. The binding of keywords follows the rules of lexical binding and variable bindings and keyword bindings may shadow each other. The abstract syntax does not distinguish between variables and keywords, but uses the same representation as before. In the new mixture syntax the level of an identifier represents the number of λ-abstractions *and* `letrec-syntax` binders between the occurrence of the identifier and the corresponding binding place. The parser needs to respect several small restrictions not explained so far, hence the parsing rules for `letrec-syntax` need to be delayed until Section 3.7. For now it suffices to know that the parser first records the source code of the transformer, then propagates the parsing substitutions—and other kinds of substitutions to be introduced shortly—to it, and finally parses the transformer. Consequently, the mixture syntax contains two variants to represent `letrec-syntax` forms, one where the transformer is a lexical s-expression and one where the transformer has been parsed:

$$c \quad ::= \ldots \mid (\textit{letrec-syn } x \textit{ ses ses}) \mid (\textit{letrec-syn } x \textit{ tf ses})$$

Following Scheme and Lisp tradition, we do not introduce a special concrete syntax for macro applications but let parenthesized expressions with the first form being a keyword represent a macro application. The mixture syntax represents macro applications through the term $\langle {}^{ks}x^n \, ses \rangle$. Section 3.7 describes how the parser recognizes a macro application and constructs the mixture syntax for it.

The macro expander needs to collect the keyword bound by `letrec-syntax` and remember this information during the expansion of the `letrec-syntax` body. In addition, the expander ensures that the identifiers the macro inserts refer to the same binding place as in the macro definition. This is vital to maintain hygiene. To that end, the expander attaches a mapping from keywords to transformers, called the *set of transformer bindings*, to the terms to be expanded. Whenever expansion enters the scope of a lexical or syntactic binding construct, it applies a shift operator to the set of transformer bindings. The shift operator increments the level of all free identifiers within

the definitions by one to reflect the fact that there is now one more binding operator between the occurrence of the identifier and its binding place.

Macro expansion also removes `letrec-syntax` forms as it records its definition in the set of transformer bindings. Removing `letrec-syntax` means removing a binding construct. Therefore the levels of the variables in the body of the removed `letrec-syntax` need to be decremented by one. An unshift operator performs this task. Both the shift and the unshift operator receive a level argument to protect local variables. The level of the operators describes the minimum level of identifiers that are affected by the operators. This protects identifiers that are bound locally relative to the introduction of the operator. Whenever the operator enters the scope of a binding construct, the level of the operator is incremented by one to protect the variables bound by the binding construct. The shift operator extends the set of terms: the term $c\uparrow^n$ increments within the term c the levels of all identifiers whose level is greater or equal to n by one. For example, in the term $(@x^0(\lambda y.(@y^0 z^1)))$ the variable y^0 is bound locally but the variables x^0 and z^1 are free. If the macro expander wants to increase the levels of all free variables by one, it applies the shift operator to the term: $(@x^0(\lambda y.(@y^0 z^1)))\uparrow^0$. The expander initially sets the level of the operator to zero as then the operator affects all free variables according to the definition of the shift operator. As the shift operator moves into the outer application, it increases the level of the variable x^0: $(@x^1(\lambda y.(@y^0 z^1))\uparrow^0)$. The level of the operator in turn increases by one as it moves into the λ abstraction: $(@x^1(\lambda y.(@y^0 z^1)\uparrow^1))$. Now the operator does not affect the (local) variable y^0 as its level is lower than the level of the operator. It does however increase the level of z: $(@x^1(\lambda y.(@y^0 z^2)))$.

In the expanded expression, the level of an identifier uniquely determines the binder of the identifier. However, the level is not powerful enough to distinguish during expansion identifier occurrences within different macro applications. As an example, consider the following code snippet:

```
(define x 42)

(define-syntax cmp
  (syntax-rules ()
     ((cmp a b)  (lambda (a) b))))

(define-syntax insx1
  (syntax-rules ()
     ((insx1 u) (cmp u x))))

(define-syntax insx2
  (syntax-rules ()
     ((insx2) (insx1 x))))

(insx2)
```

The macro application `(insx2)` should expand to the term $(\lambda x.x^1)$, that is, the identifier `x` from `insx2` should not capture the identifier `x` from `insx1`. We now take a closer look on how the expander may obtain this result. The macro application `(insx2)` leads to an application of the macro `cmp` with the arguments `x` and `x`. To perform the latter macro application, the `syntax-rules` transformer needs to decide whether the arguments are allowed to capture each other because the template (the right-hand side of the rule) is a λ-abstraction where the bound variable is the first argument and the body is the second argument of the macro. As we will see later, macro expansion records the level of the identifiers at the macro definition and manipulates it to ensure that the level refers to the original binding place as the transformer inserts the identifiers. In the example, there is no manipulation to perform as there are no bindings interleaving the macro definitions and the macro application. Hence the `x` identifiers both have a level of zero as the transformer for `cmp` processes the macro application in question. This means both identifiers still refer to the top-level definition `(define x 42)`. To the transformer for `cmp` these identifiers

are now indistinguishable because they both have the same level and there is no other information attached to them. Hence the transformer decides that they may capture each other and expands the macro application into $\lambda x.x^0$. However, this is a violation of the Hygiene Condition for Macro Expansion. Recall its definition:

> **Hygiene Condition for Macro Expansion** Generated identifiers that become binding instances in the completely expanded program must only bind identifiers that are generated at the same transcription step.

In the example above, in the terminology of the Hygiene Condition, the `(insx1 x)` form "generates" an identifier `x` and so does `(cmp u x)` in the next step. The Hygiene Condition specifies that these two generated identifiers must not bind each other as they are generated in different transcription steps. However the expansion of `(cmp u x)` uses the first generated identifier as a binding variable and—as the transformer cannot distinguish it from the second generated identifier—binds the second generated identifier with this binding variable. The transformer cannot learn from the level of the two identifiers that they were introduced during different transcription steps: the first while expanding `(insx2)` to `(insx1 x)` and the second while expanding this term to `(cmp u x)`. The information the transformer needs to know to satisfy the Hygiene Condition is that the two identifiers have been generated by different transcription steps. The example above suggests that it is possible to derive this information from the macro definition because the `x` identifiers appear in different macro definitions and hence at different places in the source code. Alas, this is generally not the case because the same macro can be invoked multiple times. As an example, consider the following macro:

```
(define-syntax gen-ys
  (syntax-rules ()
    ((gen-as (id1 id2 id3 ...) ())
     (cmp id1 id2))
    ((gen-as (id ...) (rest1 rest2 ...))
     (gen-as (y id ...) (rest2 ...)))))
```

The macro maintains two list of identifiers. In the first, it accumulates "generated" identifiers and the second list contains an arbitrary form for each identifier to be generated. Once the second list is empty, `gen-ys` expands to a macro application of `cmp` with the first two generated identifiers as arguments. Hygiene requires the macro application `(gen-ys () (1 2))` to expand to $(\lambda y.y^1)$ even though both occurrences of `y` have been introduced by the very same line of code because different transcription steps introduced the identifiers. Consequently, it is necessary to annotate the generated identifiers with information about the transcription step during macro expansion. To that end, we adapt the technique of *marking* from `syntax-case` [DHB92]. For each macro application, the expander provides the transformer with a fresh mark. The transformer propagates the mark to the inserted identifiers where it sticks. The mark then describes the macro application that introduced (or generated) the identifier. Subsequent transformers, which receive the marked identifiers in their argument, use the mark in addition to the level to distinguish identifiers. If a macro expands into the definition of another macro, the generated identifiers of the generated macro are marked twice: once as the output of the macro that expands into the macro definition and then again at every macro application of the generated macro. Of course, the process of macros generating other macro definitions can be continued ad infinitum. Hence, to support repeated marking of identifiers, we record not a single mark but a set of marks with every identifier. Each mark in the set refers to a macro application that generated the identifier. As marks, we use natural numbers.[5] With this representation, the expander only needs to maintain a counter, the *current mark*, that it initializes with 0 and that it increments at every transcription step to acquire a fresh mark. A mark k attached to an identifier means that the identifier has been introduced at the kth transcription step during the macro expansion process. If the current mark

[5] Instead of natural numbers, any infinite set could be used to represent marks. In this case, the expander would need to remember the marks used for the previous macro applications.

is k', it also means that the introducing transcription step happened before k'-k transcription steps. We extend the representation of identifiers to include a set of marks. An identifier with name x, level n that has been marked by the marks $k_1, \ldots, k_m$ is written as ${}^{\{k_1,\ldots,k_m\}}x^n$. The core macro expander also provides a means for the transformer to perform the marking. A mark operation extends the set of terms just like the parsing substitutions and the shift operator. The mark operation receives as its arguments the term to be marked and the mark to use. We write $c\sigma^k$ for marking a term c with mark k.

In the initial example `insx2` on page 45, the first macro application is `(insx2)` and in its output the identifiers `insx` and `x` are marked with the first mark, say 1. The second macro application is `(insx1 x)` and in its output the transformer marks the generated `x` with 2, the next mark. Then the transformer for `cmp` realizes that its two arguments are generated identifiers and that they have been generated by different transcription steps because they have different mark sets. Hence the transformer must ensure that the λ-abstraction the transformer produces does not capture the second argument. To that end, it increments the level of the second argument by one thus letting the identifier refer to the global binding instead of the generated λ-abstraction. The result is then the term $\lambda x.{}^{\{1\}}x^1$. Once terms have been completely parsed, the marks no longer matter. They are only a means for the transformer to distinguish generated identifiers while it decides whether one identifier may capture another. The transformer still uses the level of the identifier to indicate the binder. Hence the identifier representation within Λ_n does not change: we can simply drop the marks after macro expansion.

Parsing substitutions, the shift operator, and the mark operator serve similar tasks: they record binding information of identifiers. The subsequent text refers to them in similar contexts. Hence, we introduce the term *expansion substitution* to denote the union of the three. However, the unshift operator is not an expansion substitution as it only performs post-processing on the output of the macro expander.

Syntactic Domains:

c	$\in$ *MixtureTerms*
e	$\in$ *Expressions*
k	$\in \mathbb{N}$
d	$\in$ *Definitions*
$\underline{d}$	$\in$ *NormalizedDefinitions* $\subset$ *Definitions*
ses	$\in$ *Lex-S-Expressions*
tf	$\in$ *Transformers*
$\underline{tf}$	$\in Transformers Norm \subset$ *Transformers*

Mixture syntax:

$\underline{d}$	$::= \epsilon \mid ({}^{ks}x^n \mapsto \underline{tf} :: \underline{d})$
d	$::= \epsilon \mid ({}^{ks}x^n \mapsto \overline{tf} :: d) \mid d\uparrow$
ses	$::= se \mid ses(\!\lceil r \rceil\!) \mid ses\uparrow^n \mid ses\sigma^k$
e	$::= a \mid x^n \mid (\lambda\ x.e) \mid (@\ e e \ldots)$
c	$::= se \mid {}^{ks}x^n \mid (\lambda\ x.c) \mid (@\ c\ c \ldots) \mid \langle\ {}^{ks}x^n\ ses\rangle \mid (letrec\text{-}syn\ x\ sesses) \mid$ $(letrec\text{-}syn\ x\ tf ses) \mid d, k \vdash c \mid c(\!\lceil r \rceil\!) \mid c\uparrow^n \mid c\sigma^k \mid e\downarrow^n$

Figure 3.8: Mixture syntax for expansion

Figure 3.8 contains the mixture syntax used during expansion. It is an extension of the mixture syntax used by the parser as defined in Figure 3.5. During expansion of a term c, the expander associates a definition set d and a current mark k with the term. This is written as $d, k \vdash c$. A definition set, d, is either empty (ϵ) or a mapping from keywords to transformers (tf), or a definition set subject to a shift operator. A *normalized definition set*, $\underline{d}$, lacks the shift operator and contains only normalized transformers ($\underline{tf}$). The set of terms now includes macro applications, written $\langle\ {}^{ks}x^n\ ses\rangle$, and the two variants of *letrec-syn* as explained above. In addition, it includes

$$
\begin{aligned}
\mathit{DefSetTerms} \ni c_{\mathrm{DS}} &::= d, k \vdash c \\
&\rightarrow_{\mathrm{Expand}} \subseteq \mathit{DefSetTerms} \times \mathit{MixtureTerms} \\
d, k \vdash c &\rightarrow_{\mathrm{Expand}} \underline{d}, k \vdash c \text{ iff } d \notin \mathit{NormalizedDefinitions} && \text{(ExpandNormDef)} \\
&\qquad \text{where } d \mapsto^*_{\text{Def-El-Tf}} \underline{d} \\
\underline{d}, k \vdash c &\rightarrow_{\mathrm{Expand}} \underline{d}, k \vdash c' \text{ iff } c' \neq c \text{ where } c \mapsto^*_{\mathrm{El}} = c' && \text{(ExpandElim)} \\
\underline{d}, k \vdash {}^{ks}x^n &\rightarrow_{\mathrm{Expand}} {}^{ks}x^n && \text{(ExpandId)} \\
\underline{d}, k \vdash a &\rightarrow_{\mathrm{Expand}} a && \text{(ExpandConst)} \\
\underline{d}, k \vdash (\lambda x.c) &\rightarrow_{\mathrm{Expand}} (\lambda x.(\underline{d} \uparrow), k \vdash c) && \text{(ExpandLam)} \\
\underline{d}, k \vdash (@\ c \ldots) &\rightarrow_{\mathrm{Expand}} (@\ (\underline{d}, k \vdash c) \ldots) && \text{(ExpandApp)} \\
\underline{d}, k \vdash (\mathit{letrec\text{-}syn}\ x\ \mathit{ses}_{tf}\ \mathit{ses}) &\rightarrow_{\mathrm{Expand}} \underline{d}, k \vdash (\mathit{letrec\text{-}syn}\ x\ \mathit{tf}\ \mathit{ses}) && \text{(ExpandParseTransformer)} \\
&\qquad \text{where } \mathit{ses}_{tf} \mapsto^*_{\mathrm{PT}} \mathit{tf} \\
\underline{d}, k \vdash (\mathit{letrec\text{-}syn}\ x\ \mathit{tf}\ \mathit{ses}) &\rightarrow_{\mathrm{Expand}} (((\emptyset x^0 \mapsto \mathit{tf}) :: (\underline{d} \uparrow)), k \vdash \mathit{ses}\downarrow^1) && \text{(ExpandLetSyn)} \\
&\rightarrow_{\mathrm{PT}} \subseteq \mathit{Lex\text{-}S\text{-}Expressions} \rightarrow \mathit{Transformers}
\end{aligned}
$$

Figure 3.9: Reduction rules for the core macro expander

the shift operator, written $\uparrow^n$, the mark operator, written ${\male}^k$, and the unshift operator, written as $\downarrow^n$, where n indicates the level of the operator that protects local variables from unshifting. Lexical s-expressions now also contain the shift operator and the mark operator, hence there is a new definition for *ses*. Below we will introduce the reduction $\rightarrow_{\mathrm{Def}}$ to normalize definitions, the reduction $\rightarrow_{\uparrow}$ to eliminate shift operators, the reduction $\rightarrow_{\male}$ to eliminate mark operators, and the reduction $\rightarrow_{\downarrow}$ to eliminate the unshift operator. The elimination reductions are responsible to move the expansion substitutions from the top of the terms to the identifiers where they affect the level and the set of marks. Hence the elimination reduction $\rightarrow_{\mathrm{El}}$ from Section 3.4 receives its first redefinition and so do the elimination contexts *EL*:

$$\rightarrow_{\mathrm{El}} = \rightarrow_{\uparrow} \cup \rightarrow_{\male} \cup \rightarrow_{(\!|\emptyset|\!)}$$

$$\mathit{EL} \quad ::= \mathit{EL}(\!|t|\!) \mid \mathit{EL}\uparrow^n \mid \mathit{EL}{\male}^k \mid [\,]$$

The parser from the previous section needs extensions to handle the terms generated by the macro expander and to produce *letrec-syn* forms and macro applications. Section 3.7 contains the augmented parser.

Figure 3.9 contains the rules for the expander. The expander collects macro definitions and propagates them through the terms. However, the figure does not include a rule that describes macro application as the treatment of macro applications depends on the transformer used to define the macro. Consequently, also none of the rules increments the current mark or introduces a macro operator as this happens after a macro application only. Sections 3.6.3 and 3.9.1 will augment the expander with rules for macro applications.

These rules take place within expansion/parsing contexts *EP*from the previous section extended by a rule for the $\downarrow$ operator:

$$\mathit{EP} \quad ::= [\,] \mid (@\ e \ldots \mathit{EP}\, c \ldots) \mid (\lambda x.\mathit{EP}) \mid \mathit{EP}\downarrow^n$$

- Rule (ExpandNormDef) uses $\mapsto_{\text{Def-El-Tf}}$ which normalizes definitions. This reduction is the union of the reduction $\rightarrow_{\mathrm{Def}}$ from Figure 3.10 and the normalization of transformers,

$\rightarrow_{\text{El-Tf}}$ which is part of the specification of a transformer. All other rules require normalized definitions.

- In rule (ExpandElim) the reduction $\rightarrow_{El}$ as defined above removes the expansion substitutions from the top of the term. This enables the other rules to see the top-most syntactic form. This rule generalizes the rule (ElSubstSym) from the parser (Figure 3.6) from parsing substitutions to all expansion substitutions.
- Rules (ExpandId) and (ExpandConst) constitute the base cases of the reduction. They simply drop the definitions if the term is a number or an identifier.
- Rule (ExpandLam) performs expansion of a λ-abstraction. This works by pushing the definitions inside to the body of the abstraction and applying the shift operator to the definitions. The latter is necessary as the set of transformer bindings is then in the scope of the λ-abstraction and hence its identifiers are one level further away from their binders.
- For a function application, rule (ExpandApp) spreads the definitions to the operator and the operands. It can use the same mark for all subforms of the applications as the subforms cannot interact with each other; macro expansion of the subforms happens independently.
- The rule (ExpandParseTransformer) parses the transformer argument of a `letrec-syntax` once all expansion substitutions have been moved to the transformer. The rule uses the reduction $\rightarrow_{\text{PT}}$ to perform the actual parsing. This reduction is part of the specification of a transformer, the end of the figure only lists the domain of the reduction.
- Rule (ExpandLetSyn) describes expansion of a `letrec-syntax` form. It incorporates the new binding of the keyword x in the definitions and shifts the identifiers in the old definitions. Then the rule associates the resulting definitions with the body *ses* and applies the unshift operator to the result. The unshift operator is necessary as the rule removes also removes the *letrec-syn* form and thereby a binder. Consequently, the identifiers in the body move one level closer to their binders. The level of the operator is initialized with 1 because this is the minimum level of free variables after the binding construct disappears.

Each specification of a transformer contains rules of the form

$$\underline{d}, k \vdash \langle {}^{ks}x^{n}\, ses\rangle \rightarrow_{\text{Expand}} \underline{d}, k+1 \vdash ses'$$

will handle macro applications by transforming a macro application $\langle {}^{ks}x^{n}\, ses\rangle$ into a lexical s-expression, where $\underline{d}$ contains the definition of the accordant transformer. These rules use the mark k to mark the generated identifiers of the transformer and increment k by one afterwards.

Figure 3.10 shows the reduction of definitions required for rule (ExpandNormDef). The first rule (DefShiftEpsilon) trivially explains shifting of an empty set of transformer bindings. The next rule (DefShiftDefs) promotes the shift operator to the transformer of the first definition and to the rest of the definitions. The initial level of the shift operator for the transformer is 0 as the level of all (free) variables needs to be incremented. In addition to shifting the transformer, the level of the keyword has to be incremented, too: the keywords present in the transformer and also the keywords that have already been bound move one level further away from their binding place with every `letrec-syntax`.

The reduction $\mapsto_{\text{Def}}$ is the extension of $\rightarrow_{\text{Def}}$ to contexts D, which are defined as

$$D \quad ::= [\,] \mid ({}^{ks}x^{n} \mapsto \underline{tf} :: D) \mid D\uparrow$$

$\mapsto^{*}_{\text{Def}}$ denotes the reflexive, transitive closure of $\mapsto_{\text{Def}}$, defined as:

$$d \mapsto_{\text{Def}} d' \text{ iff } d = D[d_1], d' = D[d_2], d_1 \rightarrow_{\text{Def}} d_2$$

Furthermore, normalization of transformers within the set of transformer bindings is necessary for two reasons:

$$\begin{aligned}
\mathit{ShiftedDefinitions} \ni d_{\text{Shifted}} &::= d\uparrow \\
&\rightarrow_{\text{Def}} \subseteq \mathit{ShiftedDefinitions} \times \mathit{Definitions} \\
(\epsilon\uparrow) &\rightarrow_{\text{Def}} \epsilon && \text{(DefShiftEpsilon)} \\
(((^{ks}x^{n} \mapsto tf) :: \underline{d})\uparrow) &\rightarrow_{\text{Def}} ((^{ks}x^{n+1} \mapsto (tf\uparrow^{0})) :: (\underline{d}\uparrow)) && \text{(DefShiftDefs)}
\end{aligned}$$

Figure 3.10: Reduction for definitions

1. Rule (ExpandLetSyn) stores a non-normalized transformer in the set of transformer bindings.[6]

2. Rule (DefShiftDefs) propagates the shift operator to transformers,

We assume a reduction $\rightarrow_{\text{El-Tf}} \subseteq \mathit{Transformers} \times \mathit{Transformers}$ to perform this normalization. The specification of the transformers in the next sections will include its definition. Here, we place this reduction in contexts DTF, defined as:

$$DTF \quad ::= (^{ks}x^{n} \mapsto EL :: d) \mid (\underline{d} :: DTF) \mid DTF\uparrow$$

The context DTF uses the context elimination context EL to place the reduction at the innermost expansion substitution. Using DTF we can now define the standard reduction $\mapsto_{\text{El-Tf}}$ of the reduction $\mapsto_{\text{El-Tf}}$:

$$d \mapsto_{\text{El-Tf}} d' \text{ iff } d = DTF[tf_1], d' = DTF[tf_2], tf_1 \rightarrow_{\text{El-Tf}} tf_2$$

The context D places the hole only after normalized definitions while the context DTF places the hole in the first non-normalized transformer. Hence even their combination divides the set of transformer bindings uniquely into a context and a redex and we can combine the reduction relations $\mapsto^{*}_{\text{Def}}$ and $\mapsto^{*}_{\text{El-Tf}}$ to define $\rightarrow_{\text{Def-El-Tf}}$:

$$\mapsto_{\text{Def-El-Tf}} = \mapsto_{\text{Def}} \cup \mapsto_{\text{El-Tf}}$$

In addition to the reduction $\mapsto_{\text{Def-El-Tf}}$, the macro expander in Figure 3.9 also depends on the reduction $\rightarrow_{\text{El}}$. This reduction eliminates expansion substitutions. So far, expansion substitutions encompass parsing substitutions, the shift and the mark operator. Figure 3.7 already presented the elimination of parsing substitutions. To complete the definition of $\rightarrow_{\text{El}}$, Figure 3.11 contains the elimination rules for the shift operator and Figure 3.12 contains the elimination rules for the mark operator.

In Figure 3.11, the rule (ShiftConst) simply drops the operator with constants. The rules (ShiftId) and (ShiftLocalId) explain shifting for identifiers. Both rules first compare the level associated with the identifier with the level of the shift operator. If the level of the identifier is smaller than the level of the operator, the variable is a local identifier from the point of view of the shift operator. In this case, the rule (ShiftLocalId) matches and does not alter the level, otherwise (ShiftId) increments the level. Accordingly, rule (ShiftLam) increments the level of the shift operator as it applies it to the body of the λ-abstraction. For the `letrec-syn` form, the rule (ShiftLetSyn) increments the level of the shift operator both for the transformer and for the body as `letrec-syn` is a recursive binding construct. Finally, the rule (ShiftApp) simply needs to distribute the operator to the operand and the operands of the function application. There is no rule for macro applications because the parser in Section 3.7 will take care of this case.

Figure 3.12 contains the elimination rules for the mark operator. The operator propagates unchanged to the components of the various forms. The only interesting case is rule (MarkId)

$$
\begin{aligned}
\mathit{ShiftedTerms} \ni c_{\text{shifted}} &::= c{\uparrow^n} \\
&\rightarrow_{\uparrow} \subseteq \mathit{ShiftedTerms} \times \mathit{MixtureTerms} \\
(a{\uparrow^n}) &\rightarrow_{\uparrow} a && \text{(ShiftConst)} \\
({}^{ks}x^n{\uparrow^m}) &\rightarrow_{\uparrow} {}^{ks}x^{n+1} \text{ iff } n \geq m && \text{(ShiftId)} \\
({}^{ks}x^n{\uparrow^m}) &\rightarrow_{\uparrow} {}^{ks}x^n \text{ iff } n < m && \text{(ShiftLocalId)} \\
((\lambda x.ses){\uparrow^n}) &\rightarrow_{\uparrow} (\lambda x.(ses{\uparrow^{n+1}})) && \text{(ShiftLam)} \\
((\textit{letrec-syn } x \ ses_{tf} \ ses){\uparrow^n}) &\rightarrow_{\uparrow} (\textit{letrec-syn } x \ (ses_{tf}{\uparrow^{n+1}}) \ (ses{\uparrow^{n+1}})) && \text{(ShiftLetSyn)} \\
((@ \ ses_1 \ ses_2 \ldots){\uparrow^n}) &\rightarrow_{\uparrow} (@ \ (ses_1{\uparrow^n}) \ (ses_2{\uparrow^n}) \ldots) && \text{(ShiftApp)}
\end{aligned}
$$

Figure 3.11: Elimination of the $\uparrow$ operator

$$
\begin{aligned}
\mathit{MarkedTerms} \ni c_{\text{marked}} &::= c{\male^k} \\
&\rightarrow_{\male} \subseteq \mathit{MarkedTerms} \times \mathit{MixtureTerms} \\
a{\male^n} &\rightarrow_{\male} a && \text{(MarkConst)} \\
{}^{ks}x^n{\male^m} &\rightarrow_{\male} {}^{ks\cup\{m\}}x^n && \text{(MarkId)} \\
(\lambda x.ses){\male^n} &\rightarrow_{\male} (\lambda x.ses{\male^n}) && \text{(MarkLam)} \\
(\textit{letrec-syn } x \ ses_{tf} \ ses){\male^n} &\rightarrow_{\male} (\textit{letrec-syn } x \ ses_{tf}{\male^n} \ ses{\male^n}) && \text{(MarkLetSyn)} \\
(@ \ ses_1 \ ses_2 \ldots){\male^n} &\rightarrow_{\male} (@ \ ses_1{\male^n} \ ses_2{\male^n} \ldots) && \text{(MarkApp)}
\end{aligned}
$$

Figure 3.12: Elimination of the $\male$ operator

that describes the elimination of the mark operator for identifiers. The rule adds the mark to the set of marks of the identifier.

The unshift operator is the final operator left. Unlike expansion substitutions, it is not applied to unexpanded terms but merely serves as a post-processor on expressions to decrement the level of variables after the removal of a *letrec-syn* form. Figure 3.13 contains the rules for its elimination. The rule (UnshiftConst) drop the operator for constants. For identifiers, rule (UnshiftId) compares the level of the identifier with the level of the operator. The identifier's level is left unchanged if it is smaller than or equal to the level of the operator because in this case the identifier is bound local relative to the point where the operator has been inserted to reflect the removal of a `letrec-syntax`. Therefore no modification is necessary as the `letrec-syntax` was not a binder between the occurrence and the binding place of the identifier. Otherwise the rule decrements the level of the identifier. The rule (UnshiftLam) moves the unshift operator to the body of a λ-abstraction. It increments the level of the operator because the operator enters the scope of a binding construct and the minimum level of identifiers that are free with respect to the operator is higher by one. Finally, rule (UnshiftApp) simply propagates the unshift operator unchanged to operator and operand positions of an application.

$$
\begin{aligned}
&\mathit{UnshiftedExprs} \ni e_{\text{unshifted}} ::= e{\downarrow^n} \\
&\qquad\qquad \rightarrow_{\downarrow} \subseteq \mathit{UnshiftedExprs} \times \mathit{MixtureTerms} \\
&(a{\downarrow^n}) \rightarrow_{\downarrow} a && \text{(UnshiftConst)} \\
&({}^{ks}x^n{\downarrow^m}) \rightarrow_{\downarrow} {}^{ks}x^{n-1} \text{ iff } n \geq m && \text{(UnshiftId)} \\
&({}^{ks}x^n{\downarrow^m}) \rightarrow_{\downarrow} {}^{ks}x^n \text{ iff } n < m && \text{(UnshiftIdLocal)} \\
&((\lambda x.e){\downarrow^n}) \rightarrow_{\downarrow} (\lambda x.(e{\downarrow^{n+1}})) && \text{(UnshiftLam)} \\
&((@\ e_1\ e_2 \ldots){\downarrow^n}) \rightarrow_{\downarrow} (@\ (e_1{\downarrow^n})\ (e_2{\downarrow^n}) \ldots) && \text{(UnshiftApp)}
\end{aligned}
$$

Figure 3.13: Elimination of the $\downarrow$ operator

We can now define the standard expansion reduction, written $c \mapsto_{\text{Expand}} c'$, iff for some parsing context EP, $c \equiv EP[c_1], c' \equiv EP[c_2], c_1 \rightarrow_{\text{Expand}} c_2$). We let $\mapsto^*_{\text{Expand}}$ denote its reflexive, transitive closure. Likewise we define the standard unshift reduction, written $c \mapsto_{\downarrow} c'$, if for some parsing context EP, $c \equiv EP[e_1], c' \equiv EP[e_2], e_1 \rightarrow_{\downarrow} e_2$) and let again $\mapsto^*_{\downarrow}$ denote the reflexive, transitive closure.

At this point the description of the core macro expander is complete but for a full macro expansion facility we still need a parser that handles `letrec-syntax` and also the description of the actual transformers. A definition of a transformer needs to provide three definitions:

1. Reduction rules for macro applications, which extend $\rightarrow_{\text{Expand}}$.
2. A definition of the parsing reduction for transformers, $\rightarrow_{\text{PT}}$, which is used by the rule (ExpandParseTransformer).
3. A definition of the $\rightarrow_{\text{El-Tf}}$ reduction to eliminate expansion substitutions for transformers.

Macro expansion is then defined as the union of parsing, expansion, unshifting:

$$
\begin{aligned}
&\mathit{expand} : \mathit{S\text{-}Expressions} \rightarrow \mathit{Expressions} \\
&\mathit{expand}(se) = e \text{ iff } se(\!|{}^{\emptyset}\lambda^0/\texttt{lambda}|\!)(\!|{}^{\emptyset}\star^0/\star|\!) \mapsto^*_{\text{ParseExpandUnshift}} e
\end{aligned}
$$

where

$$\mapsto_{\text{ParseExpandUnshift}} = \mapsto_{\text{Parse}} \cup \mapsto_{\text{Expand}} \cup \mapsto_{\downarrow}$$

[6] Alternatively, we could normalize the transformers before storing them in the set of transformer bindings. However, this means mingling the unrelated reductions for expansion and normalization of transformers.

3.5.1 Time Complexity of the Macro Expander

The time complexity of a macro expansion algorithm is an important issue for a language like Scheme where all but a few syntactic forms are implemented as macros, adding new macro definitions is very common, and programs contain many macro applications. The original algorithm for hygienic macro expansion by Kohlbecker et. al. [KFFD86] runs in time quadratic in the size of the macro arguments: after each macro expansion step, it first traverses the complete output and adds time stamps to the symbols, and afterwards proceeds by expanding the output. For macros expanding into applications of other macros, the algorithm thus traverses macro arguments repeatedly on each transcription step. Clinger and Rees [CR91] and later Dybvig et. al. [DHB92] show that lazy processing of the output brings the complexity of the original algorithm down to linear time. Our macro expander also uses linear time because the expansion substitutions move inwards only as expansion proceeds. The specification formally describes how this happens and at which places it is necessary to resolve the substitutions to access the identifiers.

3.6 Computational Macro Transformers

This section introduces macros where the transformer is a Scheme procedure that computes the output of the macro. These *computational macros* are popular in languages of the Lisp family: the syntax of expressions and the external representation of data in Lisp are both based on s-expressions, and therefore a macro can construct code by assembling an s-expression whose external representation equals the syntax of the code. The fundamental problem with computational macros in Lisp is their lack of hygiene. In the output of the macro, ordinary symbols represent variables and therefore symbols with equal names become the same variables regardless of their origin.

A number of proposals aim to add hygiene to computational macros. They either provide the macro programmer with tools to create unique symbols [Cli91] or introduce a new data type that represents the code during the evaluation of the transformer procedure and records lexical information [DHB92]. The latter approach can work automatically without annotations by the programmer and is hence less error-prone. We therefore use it for our computational macros as well. Unlike existing systems such as `syntax-case`, we do not use renaming to preserve hygiene but keep track of the variable's binding places via explicit substitutions as pioneered by Bove and Arbilla [BA92]. This also enables a true native representation of a code data type that does not interact with variable bindings of the language computing the code.

The keyword `es-transformer` (for "explicit substitutions transformer") introduces a transformer for computational macros. The single argument of the transformer is a *transformer procedure*, a Scheme expression that should evaluate to a procedure, which accepts the macro expression as its single argument:

- `(es-transformer` *expression*`)`

The evaluation of the transformer procedure takes place in a default environment similar to R^5RS augmented by several special constructs and procedures that will be introduced next. We refer to the resulting language as *Macro Scheme*. As this language computes code of another language, it is really a *meta-language*.

The transformer procedure must return a *syntactic object* . It can construct this object using the special form `syntax`:

- `(syntax` *s-expression*`)` → *syntactic object*

The argument of `syntax` is an s-expression that corresponds to the code the macro generates. Unlike the s-expressions in Lisp macros, syntactic objects support hygiene by recording the lexical context of the macro definition. If the macro expander transforms a syntactic object to code, it resolves free variables in the lexical context of the macro definition just as required by the second Hygiene Condition. The macro expression that the macro application passes to the transformer

procedure is also a syntactic object. It records the lexical context of the macro application thus satisfying the first Hygiene Condition.

As an example for the use of `syntax`, consider the following macro `inc!`, which expands into an expression that increases the value of the global variable `counter` by one:

```
(define counter 0)
(define-syntax inc!
   (es-transformer
       (lambda (mcall)
          (syntax (set! counter (+ counter 1))))))
```

The evaluation of the expression

```
(let ((counter 12))
    (inc!))
```

increments the value of the global variable `counter` despite the local `let`-binding: The syntactic object generated by `syntax` recorded the global binding of the variable `counter` during the macro definition and therefore the `let`-binding did not capture the inserted variable.

As the syntactic objects encapsulate s-expressions, it is also possible to de-construct them according to the underlying s-expression form. Syntactic objects represent code hence the s-expressions in questions are mostly either atoms or pairs. Atoms cannot be deconstructed but for pairs the two accessors `syntax-car` and `syntax-cdr` exist:

- `(syntax-car` *syntactic-object*`)` → *syntactic-object*
- `(syntax-cdr` *syntactic-object*`)` → *syntactic-object*

Analogous to their counterparts on ordinary pairs, they select the first and second component of a syntactic object that encapsulates a pair. For example, the identity macro—a macro that simply returns its single argument—can be written as:

```
(define-syntax identity
   (es-transformer
       (lambda (mcall)
           (syntax-car (syntax-cdr mcall)))))
```

Macros often de-construct their arguments and combine the resulting objects with code templates to produce the output. To that end, `es-transformer` supports *meta-variables*. Meta-variables can only occur within syntactic objects and the evaluation of the transformer procedure replaces them by other syntactic objects while maintaining hygiene. The binding construct for meta-variables is the special form `syntax-lambda`. It evaluates to a procedure that binds meta-variables within its body[7].

- `(syntax-lambda` *x ... expression*`)` → *procedure*

The arguments of the procedure must be syntactic objects and the `syntax-lambda` will bind the meta-variables to these syntactic objects within the syntactic objects of *expression*.

As an example, consider the `es-transformer` version of the `thunkify` macro from Section 3.1:

```
(define-syntax thunkify
   (es-transformer
       (lambda (mcall)
           ((syntax-lambda arg (syntax (lambda () arg)))
            (syntax-cadr mcall)))))
```
[8]

[7] We do not group the arguments of `syntax-lambda` by a pair of parentheses to simplify parsing of `syntax-lambda` forms

[8] `(syntax-cadr x)` is defined as `(syntax-car (syntax-cdr x))`.

The application of the `syntax-lambda` form binds the argument of the macro call to the meta-variable `arg` within the body, which is a `syntax` expression that references the meta-variable. The substitution of meta-variables happens hygienically: Bindings in the host syntactic object cannot capture variables that are bound to the syntactic objects replacing the meta-variables. Therefore, in the following example the `let` binding of the variable `temp` does not bind the argument of the macro and the final expression evaluates to 34:

```
(define-syntax foo
    (es-transformer
        (lambda (mcall)
             ((syntax-lambda arg
                 (syntax (let ((temp 12)) arg)))
              (syntax-cadr mcall)))))
(define temp 34)
(foo temp)
```

It is important to note that the namespace of `syntax-lambda` is disjoint from the namespace of a `lambda` abstraction in Macro Scheme. While the latter binds variables that appear in expression position within its body, `syntax-lambda` binds variables that appear within syntactic objects within its body. Hence, the following code expands into `42`:

```
(define-syntax foo
    (es-transformer
        (lambda (mcall)
             ((syntax-lambda arg
                 (let ((arg 4711))
                    (syntax arg)))
              (syntax-cadr mcall)))))
(foo 42)
```

The `let` binding within the transformer procedure does not interfere with the binding of the meta-variable `arg`. Likewise, variables in Macro Scheme are not affected by `syntax-lambda`:

```
(define-syntax constant42
    (es-transformer
        (lambda (mcall)
            (let ((arg (syntax 42)))
               ((syntax-lambda arg
                    arg)
                (syntax-cadr mcall))))))
```

Every macro application of `constant42` expands to `42`, because the `arg` is bound by the `let` within the transformer procedure.

Using `syntax-lambda`, it is however possible, to insert meta-variables into Macro Scheme. This may happen if a macro expands into the definition of another macro:

```
(define-syntax genm
   (es-transformer
       (lambda (mcall)
           ((syntax-lambda proc
              (syntax (letrec-syntax
                         ((m (es-transformer
                                (lambda (mcall2)
                                    (proc (syntax 12))))))
                           (m))))
            (syntax-cadr mcall)))))
```

Here, the macro `genm` generates a local definition of the macro `m`. The transformer procedure of `m` contains the meta-variable `proc`. A macro application of `genm` will return a syntactic object that the expander turns into a local macro definition and contains a binding for the meta-variable `proc`. Next, as the expander processes the definition of `m`, it needs to expand the transformer procedure with the lexical context recorded in the syntactic object returned from `genm`. The definition of `m`'s transformer procedure depends on the argument of `genm`. For example the macro application

```
(genm (lambda (x) x))
```

will expand to `12` as `proc` is the identity procedure. The procedure could also discard its argument and return a different syntactic object. Accordingly,

```
(genm (lambda (x) (syntax 34)))
```

will expand to `34`. If however the argument of `genm` contains free identifiers as in

```
(define foo 12)
(genm foo)
```

expanding the macro application yields an *out of context* error, as there is no binding for the identifier during the evaluation of the transformer procedure: The binding for `foo` introduced by the `define` form will come to exist only after macro expansion and hence after the evaluation of the transformer procedure.

Following the principles of hygiene, using a meta-variable as a binding variable updates—in the scope of the binder—occurrences of this variable in the code substituting other meta-variables to refer to the new binder. Consider the following example:

```
(define-syntax simple-let
  (es-transformer
     (lambda (mcall)
         ((syntax-lambda var arg body
             (syntax ((lambda (var) body) arg)))
          (syntax-cadr mcall)
          (syntax-caddr mcall)
          (syntax-cadddr mcall)))))

(simple-let x (* 2 2) (+ x 23))
```

The meta-variable `var` is used as a binding variable and the macro expander updates occurrences of the corresponding variable (`x` in the sample macro application) to refer to the inserted `lambda` within the code substituting `body`, i.e. `(+ x 23)` in the sample macro application. Therefore the sample macro application expands to $(@(\lambda x.(@ +^1 x^0 23))(@ *^0 22))$.

This mechanism also affects meta-variables bound by different `syntax-lambda` forms. That is, the macro definition above can also be written as:

```
(define-syntax simple-let
  (es-transformer
     (lambda (mcall)
         ((syntax-lambda var arg
             ((syntax-lambda body
                 (syntax ((lambda (var) body) arg)))
              (syntax-cadddr mcall)))
          (syntax-cadr mcall)
          (syntax-caddr mcall)))))
```

Here, the meta-variable `body` is introduced by a different `syntax-lambda` than the meta-variable `var` but using `var` as a binding meta-variable affects the code substituting `body`.

3.6.1 The Semantics of Macro Scheme

The previous section introduced Macro Scheme as the extension of Scheme by the special forms `syntax`, `syntax-lambda` and the primitives `syntax-car` and `syntax-cdr`. This section presents the semantics for these constructs, first informally by deriving the semantics from the examples above, then formally by embedding the constructs into a semantics for the evaluation of Macro Scheme. This evaluation builds on the $\lambda_v^{n,\mathbb{C}\,\mathbb{D}}$-calculus from Section 3.3. Section 3.6.3 uses this semantics to describe the expansion of a macro application where the keyword is bound to an `es-transformer`.

The task of the `syntax` expression is to capture the lexical context of the macro definition as part of a syntactic object. Section 3.4 already introduced a vehicle to record syntactic information, namely parsing substitutions. The extension of the parser to `letrec-syntax` forms in Section 3.7 propagates these parsing substitutions to the transformer. It is the task of the transformer to use this parsing substitution. For the case of the `es-transformer`, the key idea for the explanation of `syntax` is to pass the parsing substitution collected by the expander through the evaluation of Macro Scheme until a `syntax` form is encountered. Then, `syntax` can build a syntactic object from the s-expression and the parsing substitution. That is, the parsing substitution captures the lexical context of the macro definition, and a syntactic object contains the s-expression wrapped into parsing substitutions.

A second vehicle for recording lexical information is the shift operator from Section 3.5. It prevents free identifiers from transformers from being captured by local bindings, thus ensuring hygiene for macro applications in local environments. The expander propagates the shift operator to transformers as well. In addition, and analogous to the parsing substitution, passing the shift operator to `syntax` forms during the evaluation of Macro Scheme, and letting the `syntax` form attach the shift operator to its argument, yields hygienic macro expansion for the `syntax` form of the `es-transformer`. That is, a syntactic object also needs to record applications of the shift operator and therefore—in extension of the definition above—contains an s-expression wrapped into parsing substitutions *and* applied to shift operators. Once the transformer procedure returns a syntactic object, the parsing and expansion algorithms can turn it into abstract syntax just like they do with the plain source code.

The argument of the transformer procedure, the syntactic object representing the macro form, is just the representation of the macro application form during expansion, an s-expression subject to parsing substitutions and shift operators. In addition, the expander applies the mark operator to this lexical s-expression. Consequently the contents of a syntactic object may also contain lexical s-expressions subject to the mark operator.

Before we turn to the formal definition of the evaluation of Macro Scheme and show how the expansion substitutions propagate through the evaluation to the `syntax` forms, we sketch the definition of `syntax-lambda`. `Syntax-lambda` introduces bindings for meta-variables within syntactic objects and its application replaces meta-variables by syntactic objects. A new kind of explicit substitutions, called *meta-substitutions*, represent this replacement. During the evaluation of a transformer procedure there is one meta-substitution, called the *current meta-substitution*, and an application of a `syntax-lambda` form extends the current meta-substitution to map the meta-variables to the arguments. Meta-variables only occur within syntactic objects. Therefore, evaluation of the body of the `syntax-lambda` needs to pass the meta-substitution unchanged "inwards" until it meets the syntactic objects. There, evaluation must not resolve the meta-substitution directly, because this might break hygiene. If, for example, the s-expression of some syntactic object A is `(lambda (x) e)`, where `e` is a meta-variable, and a meta-substitution wants to replace `e` by `x`, hygiene requires that the inserted `x` must not be captured by the λ-abstraction. However, whether `(lambda (x) e)` represents a λ-abstraction or not depends on the parsing substitutions, and the shift and mark operators recorded by A. Only parsing can answer this question. Hence, the meta-substitution cannot be resolved immediately. Instead, the evaluation of Macro Scheme records meta-substitutions in the syntactic object—a further extension of the definition of syntactic objects—, and the parser and the macro expander eliminate them as they construct abstract syntax from the output of the transformer.

The idea to use explicit substitutions to represent the replacement of meta-variables by input forms is due to Arbilla and Bove [BA92]. We will highlight two of their considerations that show how the elimination of meta-substitutions preserves hygiene.[9]

First, the expander must not pass meta-substitution unchanged to the body of λ-abstractions as the λ can capture identifiers that the meta-substitution will insert. As an example, consider the following code:

```
(let ((x 23))
  (letrec-syntax ((insert (es-transformer
                           (lambda (mcall)
                              ((syntax-lambda e
                                  (syntax (lambda (x) e)))
                               (syntax-cadr mcall))))))
    (insert x)))
```

For the macro application `(insert x)`, the expander creates a meta-substitution that replaces the meta-variable `e` by the identifier x^0. However, if the expander would pass this meta-substitution unchanged to the body of the term `(lambda (x) e)`, the resulting λ would capture the identifier, which would break hygiene. To prevent this, all (free) identifiers in the right-hand sides of the meta-substitution need to have their level incremented. To that end, the expander shifts the meta-substitution before applying it to the body of the abstraction. Then the result of expanding the example above becomes:

```
(let ((x 23))
  (lambda (x) x¹))
```

as required by hygiene.

The second consideration concerns the case where a meta-variable occurs as the binding variable of a λ-abstraction. Then, the meta-substitution needs to be modified because the variable that replaces the meta-variable receives a level of zero in the body of the abstraction and this new level must be reflected in the entire meta-substitution when it is passed to the body. As an example, consider the `let-simple` macro:

```
(define-syntax let-simple
  (es-transformer
    (lambda (mcall)
        ((syntax-lambda var arg body (syntax ((lambda (var) body) arg)))
         (syntax-cadr mcall)
         (syntax-caddr mcall)
         (syntax-cadddr mcall)))))
(lambda (x) (lambda (y) (let-simple x 1 (+ x 1))))
```

The substitution generated for the macro application initially replaces the meta-variable `var` by `x` and the meta-variable `body` by `(+ x 1)`. In both cases, the meta-substitution also contains a parsing substitution that replaces `x` by x^1. During the elimination of this substitution, the macro expander realizes that the meta-variable `var` is used as the binding variable of the abstraction `(lambda (var) body)`. For the body `body` it therefore has to modify the substitution to replace `var` and `body` by terms that map `x` to x^0 instead of x^1. While modifying the substitution to replace `var` by x^0 is easy, modifying the term that replaces `body` is harder: as this term has not been parsed, we do not know whether it contains a new binder for x^1. The macro expander therefore uses a new kind of explicit substitution, called *identifier substitution*. An identifier substitution replaces one identifier by another (in our case x^1 by x^0)[10]. If a meta-variable occurs as a binding variable, the expander adds an identifier substitution to the meta-substitution from where it

[9] However, Section 3.11 shows that the calculus of Arbilla and Bove does not implement all aspects of hygiene correctly.

[10] As we will see in Section 3.9.2, it is even necessary to replace an identifier by a meta-variable.

propagates to the terms that replace meta-variables within the meta-substitution. The expander eliminates the identifier substitutions as it expands the corresponding term. The elimination of identifier substitutions will require parsed terms and respect binding forms, that is, it will shift identifiers as it moves into the scope of binding constructs.

We extend the term *expansion substitution* to encompass meta-substitutions and identifier substitutions in addition to parsing substitutions and the shift and mark operators.

It remains to be shown how the parser recognizes meta-variables. The parameters of a `syntax-lambda` are visible within the syntactic objects of its scope. Hence a parsing substitution that replaces the names of the formal parameters by meta-variables is all that is required. The evaluation of `syntax-lambda` creates such a parsing substitution for the body.

We now turn to the evaluation of Macro Scheme, which we specify as an extension of the calculus from Section 3.3. In Section 3.8 we also define macro expansion and parsing for Macro Scheme.

The abstract syntax from Figure 3.2 needs several extensions to represent the terms of Macro Scheme:

- Terms to represent `syntax` and `syntax-lambda` expressions. The abstract syntax uses the term $\lfloor ses \rfloor$ to represent the former and the term $\Lambda(\mathtt{x}\ldots).e$ for the latter.
- Parsing substitutions, the shift operator, the mark operator, meta-substitutions, and identifier substitutions further augment the set of expressions.

From the mixture syntax in Figure 3.8, the abstract syntax can inherit the representation of parsing substitutions and the shift and mark operators. Meta-substitutions and identifier substitutions require new representations. However, not only the abstract syntax of Macro Scheme but also the mixture syntax needs to represent them as these terms occur as the result of Macro Scheme evaluation and the parser and expander need to eliminate them.

Figure 3.14 contains the extended abstract syntax for Macro Scheme and the extensions of the mixture syntax shared by the abstract syntax. The latter consists of a re-definition of the domain *Lex-S-Expressions*, which describes s-expressions subject to expansion substitutions. The re-definition adds meta-substitutions $\wr\int$ and identifier substitutions [ ]. A meta-substitution replaces meta-variables that range over the set *MetaVars*, which is assumed to be disjoint from *Vars*. Meta-variables are written in a sans-serif font (a). A meta-substitution s is then a sequence of pairs, $c/\mathtt{a}$, where the term c substitutes the meta-variable a. ϵ denotes the empty sequence. A meta-substitution subject to the shift operator is written as $s\uparrow$, a meta-substitution subject to an identifier substitution is written as $s[u]$, where u either denotes an identifier replacing another identifier, ${}^{ks}x^n/{}^{ks'}y^m$, or a meta-variable replacing an identifier, written as $\mathtt{a}/{}^{ks}x^n$. The latter is introduced here but not needed until the description of `syntax-rules` in Section 3.9.

The set of expressions *Expressions* in the abstract syntax includes identifiers x^n, values v, applications $(@\ e e \ldots)$, and evaluation substitutions $\wr\int$ as before. In addition, the term $(\Lambda\ (x\ \ldots).e)$ represents a `syntax-lambda` term with variables $(x\ \ldots)$ and body e. Finally, an expression subject to one of the five expansion substitutions—the parsing substitution $\Downarrow$, the shift operator $\uparrow$, the mark operator $\male$, the meta-substitution $\wr\int$, and the identifier substitution []—is also an expression.

In addition to constants and λ-abstractions, the set of values now includes *syntactic closures*[11] and syntactic objects. A syntactic closure, written $EvS[ExS[\Lambda(\mathtt{x}\ldots).e]\wr s\int]$, is the value of a `syntax-lambda` expression. Its rather complex form results from the fact that evaluation cannot propagate the expansion substitutions, the current meta-substitution, and the evaluation substitution to the body of the Λ-form. Instead the substitutions accumulate at the Λ-form. The description of evaluation will detail this process, here we only present the structure of the syntactic closure: Innermost, there is a series of expansion substitutions, represented by the context *ExS*.[12] On top of this, there is one meta-substitution, surrounded by a series of evaluation substitutions, represented by the context *EvS*. Besides syntactic closures, syntactic objects, written

[11] There is no direct relation between our syntactic closures and the ones invented by Bawden and Rees [BR88]

[12] The contexts *EL* and *P* will turn out to be identical to *ExS*.

e $\in$ *Expressions*
v $\in$ *Values*
se $\in$ *S-Expressions*
a, b $\in$ *MetaVars*
s $\in$ *MetaSubsts*
$\underline{s}$ $\in$ *NormalizedMetaSubsts*
u $\in$ *IdentifierSubsts*
id $\in$ *Vars* $\cup$ *MetaVars*
syn-clos $\in$ *SyntacticClosures*
$\lfloor ses \rfloor$ $\in$ *SyntacticObjects*
ses $\in$ *Lex-S-Expressions*

Abstract syntax:

e ::= $x^n \mid v \mid (@\ e e \ldots) \mid (\Lambda\ (x \ \ldots).e) \mid e\mathbb{C}t\mathbb{D} \mid e(\!|r|\!) \mid e\uparrow^n \mid e\sigma^k \mid e\wr s \int \mid e[u]$
v ::= $a \mid (\lambda\ x.e) \mid$ *syn-clos* $\mid \lfloor ses \rfloor$
syn-clos ::= $EvS[ExS[\Lambda(\mathsf{x}\ldots).e]\wr s \int]$

Extensions for the Mixture syntax:

ses ::= $se \mid ses(\!|r|\!) \mid ses\uparrow^n \mid ses\sigma^k \mid ses\wr s \int \mid ses[u]$
$\underline{s}$ ::= $\phi \mid c/\mathsf{a},\underline{s}$
s ::= $\phi \mid c/\mathsf{a},s \mid s\uparrow \mid s[u]$
u ::= $id/^{ks}x^n$

Contexts:

EvS ::= $[\,] \mid EvS\mathbb{C}t\mathbb{D}$
ExS ::= $[\,] \mid ExS(\!|r|\!) \mid ExS\uparrow^n \mid ExS\sigma^k \mid ExS\wr s \int \mid ExS[u]$

Figure 3.14: Abstract syntax for evaluating Macro Scheme

$\lfloor ses \rfloor$, enlarge the set of values. The s-expression inside a syntactic object is subject to expansion substitutions, i.e. a lexical s-expression *ses*.

We now turn to the reduction rules for the evaluation of Macro Scheme. We need to define the reduction of ordinary applications, the reduction of applications of `syntax-lambda` forms, the elimination of evaluation substitution expressions, the elimination of expansion substitution expressions, and the applications of primitives.

As for the normal λ calculus, evaluation takes place in evaluation contexts. The definition of evaluation contexts does not change, which means evaluation does not take place under evaluation or expansion substitutions:

$$C_v^n \quad ::= [\,] \mid (@\ v \ldots C_v^n e \ldots)$$

The rule for applications does not change either, that is we can re-use the rule β_v^n from Section 3.3.

Figure 3.15 contains the rule (SyntaxBeta) to reduce `syntax-lambda` applications. As described before, the operands have to be syntactic objects. The rule uses the contexts *EvS*and *ExS*to split the syntactic closure into evaluation substitution, the current meta-substitution, expansion substitutions, and the `syntax-lambda` expression. (SyntaxBeta) reduces the Λ expression to its body, generates parsing substitutions to bind the meta-variables in the body, extends the current meta-substitution by entries for the arguments, and applies it to the body. Afterwards, the rule propagates the context *EvS*to the augmented body.

$$\rightarrow_{SyntaxBeta} \subseteq \mathit{Expressions} \times \mathit{Expressions}$$

$$(@\ (\mathit{EvS}[\mathit{ExS}[(\Lambda(\mathtt{x}\ldots).e)]\wr s\wr])\ \lfloor ses \rfloor \ldots) \rightarrow_{(SyntaxBeta)} \mathit{EvS}[\mathit{ExS}[e(\!|\mathtt{x}/\mathtt{x}|\!)\ldots]\wr ses/\mathtt{x},\ldots,s\wr] \qquad \text{(SyntaxBeta)}$$

Figure 3.15: Application of `syntax-lambda`

Next, evaluation needs to eliminate the explicit substitutions. We distinguish between the elimination of evaluation substitutions and the elimination of the expansion substitutions.

For the evaluation substitutions, Section 3.3 already contains the rules for ordinary expressions (see Figure 3.3). An evaluation reduction applied to a syntactic closure cannot be reduced but sticks to the syntactic closure. Elimination of evaluation substitutions applied to syntactic objects is also easy: As a syntactic objects contains no Macro Scheme variables, the evaluation substitution has no effect and the rule (EvalSubstSyn) drops it:

$$\mathit{ESExpr} \ni e_{\text{evs}} ::= e(\!|t|\!)$$

$$\rightarrow_{(\!|\,|\!)} \subseteq \mathit{ESExpr} \times \mathit{Expressions}$$

$$\lfloor ses \rfloor (\!|t|\!) \rightarrow_{(\!|\,|\!)} \lfloor ses \rfloor \qquad \text{(EvalSubstSyn)}$$

The elimination of evaluation substitutions takes place in elimination contexts C_{elim}, which are defined like evaluation contexts but select the innermost evaluation substitution via the helper context *AS*:

$$C_{\text{elim}} \quad ::= (@\ v \ldots C_{\text{elim}} e \ldots) \mid AS$$
$$AS \quad ::= [\,] \mid AS(\!|t|\!) \mid AS(\!|r|\!) \mid AS\uparrow^n \mid AS\sigma^k \mid AS\wr s\wr \mid AS[u]$$

Using the elimination contexts C_{elim}, we can define the standard elimination reduction for evaluation substitutions as $e \mapsto_{(\!|\,|\!)} e'$, iff for some elimination context C_{elim}, $e \equiv C_{\text{elim}}[e_1]$, $e' \equiv C_{\text{elim}}[e_2]$, $e_1 \rightarrow_{(\!|\,|\!)} e_2$.

Evaluation of expansion substitutions is next. Evaluation needs to propagate the expansion substitutions to the syntactic objects. There, the substitution applies to the lexical s-expression of the syntactic object. Again, evaluation cannot reduce explicit substitutions applied to syntactic closures. As these two considerations apply to all expansion substitutions, we abstract over them using a context and use one set of rules for all of them. The abstracting context is *ES*:

$$ES \quad ::= [\,]\langle\!|r|\!\rangle \mid [\,]\uparrow^n \mid [\,]\sigma^{\nearrow k} \mid [\,]\wr s \int \mid [\,][u]$$

Figure 3.16 contains the respective rules. For constants, rule (EvalESConst) drops the substitution. The substitutions do not affect identifiers, hence rule (EvalESId) likewise drops them. For applications, rule (EvalESApp) distributes the substitution to operator and operands. For λ-abstractions, rule (EvalESLam) pushes substitution inside to the body. No shifting is necessary as the substitutions do not contain variables. Finally, if a substitution encounters a syntactic object, rule (EvalESSyn) records the substitution by applying it to the lexical s-expression of the syntactic object. As said above, there is no rule for `syntax-lambda` which means that substitutions applied to such terms will simply stay and thereby build up syntactic closures.

$$\begin{aligned} \textit{ExpandSExpr} \ni e_{\text{exs}} ::= e \mid e_{\text{exs}}\langle\!|r|\!\rangle \mid e_{\text{exs}}\uparrow^n \mid e_{\text{exs}}\sigma^{\nearrow k} \mid e_{\text{exs}}\wr s \int \mid e_{\text{exs}}[u] \\ \rightarrow_{\text{ES}} \subseteq \textit{ExpandSExpr} \times \textit{Expressions} \end{aligned}$$

$$ES[a] \rightarrow_{\text{ES}} a \qquad \text{(EvalESConst)}$$

$$ES[{}^{ks}x^n] \rightarrow_{\text{ES}} {}^{ks}x^n \qquad \text{(EvalESId)}$$

$$ES[(@e_1 \ldots)] \rightarrow_{\text{ES}} (@\ ES[e_1] \ldots) \qquad \text{(EvalESApp)}$$

$$ES[(\lambda x.e)] \rightarrow_{\text{ES}} (\lambda x.ES[e]) \qquad \text{(EvalESLam)}$$

$$ES[\lfloor ses \rfloor] \rightarrow_{\text{ES}} \lfloor ES[ses] \rfloor \qquad \text{(EvalESSyn)}$$

Figure 3.16: Evaluation for the explicit substitutions for parsing and expansion

Again a standard reduction, $e \mapsto_{\text{ES}} e'$, is defined: iff for some elimination context C_{elim}, $e \equiv C_{\text{elim}}[e_1]$, $e' \equiv C_{\text{elim}}[e_2]$, then $e_1 \rightarrow_{\text{ES}} e_2$.

A description of the primitives `syntax-car` and `syntax-cdr` will now complete the description of Macro Scheme. We take these two primitives as examples for functions related to syntactic objects that encapsulate s-expressions. The definition of other selectors and predicates for s-expression objects such as `syntax-cadr`, `syntax-null?` and `syntax-equal?` is completely analogous.

The two primitives accept a syntactic object as their single argument and require that this syntactic object encapsulates a pair or a meta-variable that is bound to a syntactic object representing a pair. To support the second possibility, it is necessary that the primitives resolve meta-substitutions. However, this process stops once the outermost pair has been revealed. This means that the contents may become code and hence the parser is required to determine the syntactic roles of the components. The following definition uses the helper reduction $\mapsto_{\text{patout}}$ to resolve the lexical s-expression within the syntactic object into a form where the outermost pair becomes visible. The reduction pushes the substitutions that surround this pair inside to the two elements of the pair thus enabling the normal selectors for pairs to access the first or second element. The actual definition of $\mapsto_{\text{patout}}$ will be given later in Section 3.9.3, for now it suffices to know that it fulfills the requirements just sketched.

Figure 3.17 contains the definitions of the primitives `syntax-car` and `syntax-cdr`. The rule (EvalSyntaxCar) reduces the lexical s-expression of the syntactic object using $\mapsto_{\text{patout}}$ to reveal the pair and uses the primitive *car* to select the first element. It returns this element as a syntactic object. The rule (EvalSyntaxCdr) works analogously but returns the second element using *cdr*.

To shorten the presentation, these rules do not check that the variables `syntax-car` and `syntax-cdr` are indeed the top-level identifiers we intend to define but simply assume that no intermediate bindings for these names exist. Section 3.4 already demonstrated what a hygienic check using special identifiers would look like and the chapter about module systems will add a more convenient way to formulate references to identifiers from certain name-spaces.

Now all rules are in place to define the evaluation of Macro Scheme expressions.

$$\mapsto_{\text{patout}} \subseteq \textit{Lex-S-Expressions} \times \textit{Lex-S-Expressions}$$
$$\rightarrow_{\delta} \subseteq \textit{Expressions} \times \textit{Expressions}$$
$$(@\ \texttt{syntax-car}^n\ \lfloor ses \rfloor) \rightarrow_{\delta} \lfloor car(ses') \rfloor \text{ where } ses \mapsto^*_{\text{patout}} ses' \qquad \text{(EvalSyntaxCar)}$$
$$(@\ \texttt{syntax-cdr}^n\ \lfloor ses \rfloor) \rightarrow_{\delta} \lfloor cdr(ses') \rfloor \text{ where } ses \mapsto^*_{\text{patout}} ses' \qquad \text{(EvalSyntaxCdr)}$$

Figure 3.17: Evaluation for primitives

Together δ, β_v^n and (SyntaxBeta) form the notion of reduction $\rightarrow_v^{n,SyntaxBeta}$:

$$\rightarrow_v^{n,SyntaxBeta} = \delta \cup \beta_v^n \cup (SyntaxBeta)$$

The standard reduction function $\mapsto_v^{n,SyntaxBeta}$ places $\rightarrow_v^{n,SyntaxBeta}$ within an evaluation context:

$$e \mapsto_v^{n,SyntaxBeta} e' \text{ if } p \rightarrow_v^{n,SyntaxBeta} q,\ e \equiv C_v^n[p],\ e' \equiv C_v^n[q],\ p, q \in \Lambda_M^n$$

Next, we combine expression evaluation with elimination of evaluation substitutions and expansion substitutions:

$$\mapsto_v^{MS} = \mapsto_v^{n,SyntaxBeta} \cup \mapsto_{\text{ES}} \cup \mapsto_{(\!|\,|\!)}$$

This yields the definition of the evaluation function for Macro Scheme:

$$eval_v^{MS}(p) \stackrel{Def}{\equiv} v \text{ iff } p \mapsto_v^{MS*} v$$

The next section adds some more primitives to the reduction $\rightarrow_\delta$. Then, finally, Section 3.6.3 defines macro applications for `es-transformer`. It will use $eval_v^{MS}$ to evaluate the argument of `es-transformer` to a closure, generate, for a macro application, an application of this closure on the arguments of the macro application, and use $eval_v^{MS}$ to evaluate this application. The core macro expander then processes the resulting syntactic object and produces the expanded expression.

3.6.2 Additional Primitives

In addition to the s-expression related primitives, a few other primitives are necessary to write powerful macros. Most notably, primitives to inspect the lexical context of syntactic objects are missing. `Syntax-case` [DHB92] defines two primitives for comparing identifiers with regard to their binding places:

- `free-identifier=?` determines whether two identifiers are equal if they appear in the output of a macro and the output does not bind them.
- `bound-identifier=?` determines whether two identifiers are equal if they appear in the output of a macro and the output binds them.

The first primitive is often used to simulate the matching against literal identifiers with a `syntax-rules` transformer. For example the following macro definition expands to the number one if its argument is `one` and to the number two otherwise:

```
(define-syntax onetwo
   (es-transformer
      (lambda (mcall)
          (if (free-identifier=? (syntax-cadr mcall)
                                 (syntax one))
              (syntax 1)
              (syntax 2)))))
```

For macro applications `(onetwo one)` where `one` is unbound, this macro expands to 1 as `one` is also unbound at the macro definition. If however `one` is a local variable, the same macro application expands to 2 as then the argument is a local variable whereas the `one` from `(syntax one)` is still unbound and hence the two identifiers are not equal if they appear in the output of a macro. That is, `(lambda (one) (onetwo one))` expands to `(lambda (one) 2)`.

The `bound-identifier=?` primitive can be used to detect duplicate entries within a list of bound variables. For example the macro `let23-42` binds two variables to 23 and 42:

```
(define-syntax let23-42
  (es-transformer
     (lambda (mcall)
         (if (bound-identifier=? (syntax-cadr mcall)
                                 (syntax-caddr mcall))
             (error "duplicate bound identifier")
             ((syntax-lambda v1 v2 body
                  (syntax ((lambda (v1 v2) body) 23 42)))
              (syntax-cadr mcall)
              (syntax-caddr mcall))))))
```

The macro application `(let23-42 a a 65)` yields an error but if one of the identifiers has been generated by a macro, the identifiers are no longer `bound-identifier=?`:

```
(define-syntax call-let23-42
   (es-transformer
      (lambda (mcall)
          ((syntax-lambda v
              (syntax (let23-42 v a (+ v a))))
           (syntax-cadr mcall)))))
(call-let23-42 a)
```

If in this example `let23-42` had applied the `free-identifier=?` to the arguments, the primitive would have returned `#t` because both identifiers refer to the same enclosing binding. In this case, they are unbound.

Using the explicit substitutions attached to syntactic object, it is possible to specify the identifier equality checks in the implementation of the primitives. `Free-identifier=?` needs to resolve all pending substitutions as shown in Figure 3.18, rules (EvalFreeIdEqTrue) and (EvalFreeIdEqFalse), and then ignore the marks when comparing the results. The function *free-id=?* defined in the same figure performs this comparison. To support comparison of unbound identifiers as well, the function compares the name as well as the level. For bound identifiers, comparing the name is superfluous but within a macro application all unbound identifiers have the same level and there it is necessary to account the name to decide equality.

Checking whether two identifiers A and B are equal if they appear as bound variables in the output boils down to the question whether it is possible to capture A if B is used as a bound variable. (The vice versa question will always return the same answer, so regarding one case suffices). As required by the Hygiene Condition for Macro Expansion, this is only the case if both identifiers are either introduced by a macro application in the same transcription step or if both identifiers appear in the input to the macro application. This means that the level and the marks must be equal after the expansion substitutions have been resolved. The Rules (EvalBoundIdTrue) and (EvalBoundIdFalse) can therefore directly compare the result of substitution elimination.

Sometimes it is desirable to write non-hygienic macros. To that end, the primitive `datum->syntax-object` turns a lexical s-expression into a syntactic object. The lexical context of the syntactic object is not simply empty. Instead, the primitive accepts another syntactic object and uses its lexical context as the context for the new object. Rule (EvalDatumToSyntaxObject) reduces applications of this primitive.

$$\rightarrow_\delta \subseteq \mathit{Expressions} \times \mathit{Expressions} \tag{3.1}$$

$$(@\ \texttt{free-identifier=?}^n\ \lfloor ses_1 \rfloor\ \lfloor ses_2 \rfloor) \rightarrow_\delta 1 \text{ iff } \mathit{free\text{-}id{=}?}({}^{ks}x^n, {}^{ks'}y^m) \tag{EvalFreeIdEqTrue}$$

$$\text{where } ses_1 \mapsto^*_{\text{patout}} {}^{ks}x^n \text{ and } ses_2 \mapsto^*_{\text{patout}} {}^{ks'}y^m$$

$$(@\ \texttt{free-identifier=?}^n\ \lfloor ses_1 \rfloor\ \lfloor ses_2 \rfloor) \rightarrow_\delta 0 \text{ iff } \neg \mathit{free\text{-}id{=}?}({}^{ks}x^n, {}^{ks'}y^m) \tag{EvalFreeIdEqFalse}$$

$$\text{where } ses_1 \mapsto^*_{\text{patout}} {}^{ks}x^n \text{ and } ses_2 \mapsto^*_{\text{patout}} {}^{ks'}y^m$$

$$(@\ \texttt{bound-identifier=?}^n\ \lfloor ses_1 \rfloor\ \lfloor ses_2 \rfloor) \rightarrow_\delta 1 \text{ iff } {}^{ks}x^n = {}^{ks'}y^m \tag{EvalBoundIdTrue}$$

$$\text{where } ses_1 \mapsto^*_{\text{patout}} {}^{ks}x^n \text{ and } ses_2 \mapsto^*_{\text{patout}} {}^{ks'}y^m$$

$$(@\ \texttt{bound-identifier=?}^n\ \lfloor ses_1 \rfloor\ \lfloor ses_2 \rfloor) \rightarrow_\delta 0 \text{ iff } {}^{ks}x^n \neq {}^{ks'}y^m \tag{EvalBoundIdFalse}$$

$$\text{where } ses_1 \mapsto^*_{\text{patout}} {}^{ks}x^n \text{ and } ses_2 \mapsto^*_{\text{patout}} {}^{ks'}y^m$$

$$(@\ \texttt{datum->syntax-object}^n\ \lfloor \mathit{ExS}[se_1] \rfloor\ (\texttt{'}\,se_2)) \rightarrow_\delta \lfloor \mathit{ExS}[se_2] \rfloor \tag{EvalDatumToSyntaxObject}$$

$$\mathit{free\text{-}id{=}?}({}^{ks}x^n, {}^{ks'}y^m) = \begin{cases} \mathit{true} \text{ if } x = y \wedge n = m \\ \mathit{false} \text{ otherwise} \end{cases}$$

Figure 3.18: Evaluation for additional primitives

While the paper on `syntax-case` [DHB92] proposes `datum->syntax-object` as a general tool for importing arbitrary code into a certain lexical context, the PLT manual [Fla04] also shows the limits of the primitive:

> [...] Another reason to use syntax-case is to implement "non-hygienic" macros that introduce capturing identifiers:
>
> ```
> (define-syntax (if-it stx)
>   (syntax-case stx ()
>     [(src-if-it test then else)
>      (syntax-case (datum->syntax-object (syntax src-if-it) 'it) ()
>        [it (syntax (let ([it test]) (if it then else)))])])))
> (if-it (memq 'b '(a b c)) it 'nope) ; => '(b c)
> ```
>
> [...]
> Macros that expand to non-hygienic macros rarely work as intended. For example:
>
> ```
> (define-syntax (cond-it stx)
>   (syntax-case stx ()
>     [(_ (test body) . rest)
>      (syntax (if-it test body (cond-it . rest)))]
>     [(_) (syntax (void))]))
> (cond-it [(memq 'b '(a b c)) it] [#t 'nope]) ; => undefined variable it
> ```
>
> The problem is that `cond-it` introduces `if-it` (hygienically), so `cond-it` effectively introduces `it` (hygienically), which doesn't bind `it` in the source use of `cond-it`. In general, the solution is to avoid macros that expand to uses of non-hygienic macros.[13]

[13] In this particular case, Shriram Krishnamurthi points out changing `if-it` to use `(datum->syntax-object (syntax test) 'it)` solves the problem in a sensible way.

3.6.3 Expanding Macro Applications Using `es-transformer`

The last two sections defined the evaluation semantics of Macro Scheme. There are two places where the evaluation of Macro Scheme expressions is necessary. First, the transformer procedure argument of `es-transformer` must be evaluated to a closure before storing it in the set of transformer bindings. This happens while parsing the transformer using the reduction $\rightarrow_{\mathrm{PT}}$ in rule (ExpandParseTransformer). The reduction $\rightarrow_{\mathrm{PT}}$ for `es-transformer` will be described at the end of this section. Second and most important, for macro applications where the keyword is defined using `es-transformer`, $eval_v^{MS}$ evaluates the application of this closure to the macro application form. More specifically, a macro application extracts the transformer closure from the set of transformer bindings, constructs a syntactic object holding the macro application form and an application of the closure to this syntactic object. It uses the function $eval_v^{MS}$ to perform this application. The result is a syntactic object representing the output of the macro application. Parser and expander need to turn this object into abstract syntax as described in Sections 3.4 and 3.5. In this process, the reduction $\rightarrow_{\mathrm{El}}$—used in the rule (ExpandElim) of the expander (Figure 3.9)—now needs to eliminate meta-substitutions and identifier substitutions as well. This section presents the respective rules and the accordant extension of $\rightarrow_{\mathrm{El}}$.

To formulate macro application using `es-transformer`, it is first necessary to augment the terms of the mixture syntax by transformers for `es-transformer`, meta-substitutions, meta-variables, and identifier substitutions as introduced during the presentation of Macro Scheme (see Figure 3.14). Figure 3.19 contains the new mixture syntax. It is an extension of Figure 3.8.

Mixture syntax:

$$
\begin{array}{lll}
tf & ::= & (\textit{es-transf}\ e) \mid tf(\!|r|\!) \mid tf\uparrow^n \mid tf\sigma^k \mid tf\lceil s\rfloor \mid tf[u] \\
\underline{tf} & ::= & (\textit{es-transf}\ e) \\
\underline{c} & ::= & se \mid {}^{ks}x^n \mid \mathtt{a} \mid (\lambda\ x.c) \mid (\lambda\ \mathtt{a}.c) \mid (@\ \mathrm{c}\ \mathrm{c}\ \ldots) \mid \langle\ {}^{ks}x^n\ ses\rangle \mid \\
 & & (\textit{letrec-syn}\ idsesses) \mid (\textit{letrec-syn}\ idtfses) \mid \\
 & & d,k \vdash c \mid c(\!|r|\!) \mid c\uparrow^n \mid c\sigma^k \mid c\lceil s\rfloor \mid c[u] \mid e\downarrow^n
\end{array}
$$

Figure 3.19: Mixture syntax for expanding `es-transformer` macros

There, the sets of transformers and normalized transformers have been left unspecified. Here, we introduce them based on the term (*es-transf* e), which represents an `es-transformer`. The argument e represents the evaluated transformer closure possibly subject to expansion substitutions that the expander adds during expansion. Furthermore, a non-normalized transformer may be subject to expansion substitutions. The set of terms also includes λ-abstractions where the bound variable is a meta-variable. Such terms occur during the expansion of terms where the macro uses a meta-variable as a binding variable. The parser must not resolve this variable at the time it creates the λ expression because the corresponding meta-substitution must be modified to reflect the fact that a meta-variable is used as a binding variable and an identifier substitution needs to be created. (See the sample macro `simple-let` from Section 3.6.1) Instead the parser creates a λ expression with a meta-variable as its binding variable and only the elimination of the corresponding meta-substitution resolves the meta-variable and modifies the meta-substitution accordingly. This will be shown below in the elimination rules for the meta-substitution; an extended version of the parser which creates λ-abstractions with meta-variables is included in Section 3.7. For the same reasons, *letrec-syn* terms with meta-variables need to be included.[14]

With the extended mixture syntax in place we can now turn to macro applications using `es-transformer` macros. Figure 3.20 contains the rule (ExpandESTapp), which describes the application of `es-transformer` macros. The rule extends the core macro expander from Figure 3.9. The rule searches for such a transformer using the helper function D also defined in Figure 3.20.

[14] If a macro generates another macro, it is possible for the generated macro to contain meta-variables of the generating macro as binding meta-variables of `syntax-lambda`. We do not cover this case here but will explain it in the context of `syntax-rules` transformers.

$$\begin{aligned}
&\mathit{DefSetTerms} \ni c_{\mathrm{DS}} ::= d \vdash c \\
&\qquad \rightarrow_{\mathrm{Expand}} \subseteq \mathit{DefSetTerms} \times \mathit{MixtureTerms} \\
&\underline{d}, k \vdash \langle {}^{ks}x^n \mathit{ses} \rangle \rightarrow_{\mathrm{Expand}} \underline{d}, k+1 \vdash \mathit{ses} \text{ iff } D({}^{ks}x^n, \underline{d}) = (\mathit{es\text{-}transf}\ e) \qquad \text{(ExpandESTapp)} \\
&\qquad\qquad \text{where } \lfloor \mathit{ses} \rfloor = \mathit{eval}_v^{MS}((@\ (e \text{♂}^k \lceil \phi \rceil)\ \lfloor \mathit{ses} \rfloor))
\end{aligned}$$

$$\begin{aligned}
D : (\mathit{Ids} \times \mathit{NormalizedDefinitions}) &\rightarrow \mathit{Transformers} \\
D({}^{ks}x^n, ({}^{ks}x^n \mapsto \mathit{tf} :: \underline{d})) &= \mathit{tf} \\
D({}^{ks}x^n, ({}^{ks'}y^m \mapsto \mathit{tf} :: \underline{d})) &= D({}^{ks}x^n, \underline{d}) \text{ if } x \neq y \vee n \neq m
\end{aligned}$$

Figure 3.20: Expansion for `es-transformer` forms

The function D simply walks along the set of transformer bindings and compares the keyword from the macro application with the bound keywords. If the name of the keyword and the level matches, it returns the associated transformer. It is undefined on empty sets and does not consult the mark sets of the identifiers because it only needs to know if the keywords are bound by the same binder. (ExpandESTapp) now turns the macro application form into a syntactic object by applying the $\lfloor \rfloor$ constructor, marks the transformer closure with the current mark, builds an application of the marked closure to this syntactic object, and uses the function eval_v^{MS} to evaluate the call. The rule requires the value to be a syntactic object and the rule reduces to the content of the syntactic object. Next, the parser and macro expander process this term but the rule increments the current mark by one. This entails that the terms generated by the next macro application receive a different mark.

This completes the description of the expansion of macro applications using `es-transformer`. The only loose end is the redefinition of the $\rightarrow_{\mathrm{El}}$, which eliminates the expansion substitutions. This function now—in addition to the parsing substitution, the shift operator, and the mark operator—needs to handle meta-substitutions and identifiers substitutions as produced by Macro Scheme. The description of these elimination reductions comes next by first describing the elimination rules for meta-substitutions and then the elimination rules for identifier substitutions. Afterwards, we add elimination rules for the shift operator and the mark operator that handle terms involving meta-variables (meta-variables themselves, λ-abstractions and *letrec-syn*-terms with meta-variables as binding variables). This is necessary because we introduced meta-variables after the initial definitions of the elimination for these operators in Section 3.5. Finally, we re-define $\rightarrow_{\mathrm{El}}$ to include these rules as well.

Figure 3.21 contains the elimination rules for meta-substitutions. These rules capture the spirit of hygienic macro expansion with meta-variables: They apply the shift operator to the substitutions before moving the substitutions into the scope of a binding construct to avoid capturing of identifiers in the substituting terms, and if a binding construct binds a meta-variable, the rules update the entire meta-substitution to reflect the new binding.

(*NormalizedMetaSubsts*) This rule immediately normalizes meta-substitutions; all other rules assume normalized meta-substitutions. This makes the elimination process deterministic. The corresponding elimination reduction $\rightarrow_{\mathrm{SMS}}$ is defined in the lower part of the figure and will be explained after this listing.

(MetaSubstConst) and (MetaSubstId) These two rules drop the meta-substitutions as the operands are not within the domain of the substitution.

(MetaSubstMV) For a meta-variable that is part of the substitution, the variable needs to be replaced by the term c.

(MetaSubstMVEmpty) An empty meta-substitution has no effect.

(MetaSubstMVOther) If the first pair does not match the meta variable, the search continues in the rest of the sequence.

(MetaSubstLam) Hygiene requires us to increment the levels of all free variables when propagating the substitution to the body of the λ-abstraction to avoid unintended capturing. Therefore, the shift operator is applied to the substitution as it is moved inside to the body.

(MetaSubstLamMVId) Here the bound variable of a λ abstraction is a meta-variable and the substitution contains a replacement for this meta-variable, hence the binding variable of the λ form is established. Hygiene requires that the whole substitution is modified to reflect this introduction of a binding. Otherwise the context of the macro application could capture the free identifiers of the substitution. To obtain the binding variable, the rule applies the meta-substitution on the meta-variable x and resolves this and all pending substitutions—including parsing substitutions, and shift and mark operators—using the reduction $\rightarrow_{\mathrm{El}}$, which yields the variable ${}^{ks}y^{n}$. The reduction $\rightarrow_{\mathrm{El}}$ will be defined at the end of this section as the union of the corresponding elimination reductions. The variable ${}^{ks}y^{n}$ must be replaced by the variable ${}^{ks}y^{0}$ within the substitution as the λ-abstraction now binds this variable. Apart from that, our rule needs to increment the levels of all other identifiers in the meta-substitution as in the previous case. Taken together, the substitution for the body is extended by a new pair, receives a shift operator and afterwards an identifier substitution mapping the then-incremented version of the bound variable to a variable with index zero.

(MetaSubstLamMVMV) If the meta-variable is not in the domain of the substitution, no replacement takes place and the rule applies the shifted substitution to the body.

(MetaSubstLetSyn) This case is similar to the rule (MetaSubstLam): the rule shifts the substitution before applying it to the body and to the transformer.

(MetaSubstLetSynMVId) Analogously to rule (MetaSubstLamMVId), the bound identifier of a `letrec-syntax` may be a meta-variable that the substitution maps to a variable. Therefore, the rule needs to update the substitution to reflect the new binding.

(MetaSubstLetSynMVMV) This rule covers the case where the substitution does not affect the bound meta-variable. Only shifting the substitution is required.

(MetaSubstApp) For applications, the elimination works by spreading the substitution unchanged to operator and operand.

The lower part of Figure 3.21 contains the reduction $\rightarrow_{\mathrm{SMS}}$ for normalization of meta-substitutions. The rules (SMSShiftEmpty) and (SMSSubstEmpty) cover the trivial cases in which the substitution is empty (ϕ). The rule (SMSShiftSubst) explains the shift operator for substitutions. It simply propagates the operator to the lexical s-expression and to the rest of the substitution. The elimination of identifier substitutions for meta-substitutions is covered by the rule (SMSSubstSubstId). Again, all that needs to be done is the propagation to the lexical s-expression and to the rest.

The reduction takes place in contexts *SMS*:

$$SMS \quad ::= [\,] \mid \mathsf{a}/c, SMS \mid SMS[id/{}^{ks}x^{n}] \mid SMS\uparrow$$

That is, the standard reduction used in rule (*NormalizedMetaSubsts*) is defined as $SMS[s] \mapsto_{\mathrm{SMS}} SMS[s']$ if $s \rightarrow_{\mathrm{SMS}} s'$ for some context *SMS*.

The identifier substitution needs elimination rules, too. Figure 3.22 contains them. The rule (IdSubstId) replaces a variable by some other identifier. If the level or marks do not match, rule (IdSubstOther) drops the substitution. The rules (IdSubstMV) and (IdSubstConst) drop substitutions for the other base cases. For λ-abstractions two rules are necessary according to the two different types of identifiers that replace the identifiers. If the identifier is a normal

$$\begin{array}{rl}
\mathit{MetaSTerms} \ni c_{\mathrm{MS}} ::= & c\wr s\int \\
\multicolumn{2}{r}{\to_{\wr\int} \subseteq \mathit{MetaSTerms} \times \mathit{MixtureTerms}} \\
\multicolumn{2}{r}{ses\wr s\int \to_{\wr\int} ses\wr s'\int \text{ where } s \mapsto^*_{\mathrm{SMS}} s' \text{ iff } s \notin \mathit{NormalizedMetaSubsts}} \\
\multicolumn{2}{r}{\text{(MetaSubstNorm)}} \\
a\wr s_!\int \to_{\wr\int} a & \text{(MetaSubstConst)} \\
{}^{ks}x^n\wr s_!\int \to_{\wr\int} {}^{ks}x^n & \text{(MetaSubstId)} \\
\mathsf{a}\wr ses/\mathsf{a}, s_!\int \to_{\wr\int} ses & \text{(MetaSubstMV)} \\
\mathsf{a}\wr\phi\int \to_{\wr\int} \mathsf{a} & \text{(MetaSubstMVEmpty)} \\
\mathsf{a}\wr ses/\mathsf{b}, s_!\int \to_{\wr\int} \mathsf{a}\wr s_!\int \text{ iff } \mathsf{a} \neq \mathsf{b} & \text{(MetaSubstMVOther)} \\
(\lambda x.ses)\wr s_!\int \to_{\wr\int} (\lambda x.ses\wr(s_! \uparrow)\int) & \text{(MetaSubstLam)} \\
(\lambda \mathsf{x}.ses)\wr s_!\int \to_{\wr\int} (\lambda y.ses\wr((s_! \uparrow)[{}^{ks}y^0/{}^{ks}y^{n+1}])\int) \text{ iff } \mathsf{x}\wr s_!\int \mapsto^*_{\mathrm{El}} {}^{ks}y^n & \\
\multicolumn{2}{r}{\text{(MetaSubstLamMVId)}} \\
(\lambda \mathsf{x}.ses)\wr s_!\int \to_{\wr\int} (\lambda \mathsf{x}.ses\wr(s_! \uparrow)\int) \text{ iff } \mathsf{x}\wr s_!\int \mapsto^*_{\mathrm{El}} \mathsf{x} & \text{(MetaSubstLamMVMV)} \\
(\mathit{letrec\text{-}syn}\ x\ ses_{tf}\ ses)\wr s_!\int \to_{\wr\int} (\mathit{letrec\text{-}syn}\ x\ ses_{tf}\wr(s_! \uparrow)\int\ ses\wr(s_! \uparrow)\int) & \text{(MetaSubstLetSyn)} \\
(\mathit{letrec\text{-}syn}\ \mathsf{x}\ ses_{tf}\ ses)\wr s_!\int \to_{\wr\int} (\mathit{letrec\text{-}syn}\ y\ ses_{tf}\wr s'\int\ ses\wr s'\int) & \text{(MetaSubstLetSynMVId)} \\
\multicolumn{2}{r}{\text{iff } \mathsf{x}\wr s_!\int \mapsto^*_{\mathrm{El}} {}^{ks}y^n \text{ where } s' = ((s_! \uparrow)[{}^{ks}y^0/{}^{ks}y^{n+1}])} \\
(\mathit{letrec\text{-}syn}\ \mathsf{x}\ ses_{tf}\ ses)\wr s_!\int \to_{\wr\int} & \text{(MetaSubstLetSynMVMV)} \\
\multicolumn{2}{r}{(\mathit{letrec\text{-}syn}\ \mathsf{x}\ ses_{tf}\wr(s_! \uparrow)\int\ ses\wr(s_! \uparrow)\int) \text{ iff } \mathsf{x}\wr s_!\int \mapsto^*_{\mathrm{El}} \mathsf{x}} \\
(@\ ses_1\ ses_2 \ldots)\wr s_!\int \to_{\wr\int} (@\ ses_1\wr s_!\int\ ses_2\wr s_!\int \ldots) & \text{(MetaSubstApp)}
\end{array}$$

$$\begin{array}{rl}
\mathit{ShiftedMetaSubst} \ni s_{\mathrm{shifted}} ::= & s\uparrow \\
\multicolumn{2}{r}{\to_{\mathrm{SMS}} \subseteq \mathit{ShiftedMetaSubst} \times \mathit{MetaSubsts}} \\
(\phi \uparrow) \to_{\mathrm{SMS}} \phi & \text{(SMSShiftEmpty)} \\
(ses/\mathsf{a}, s_! \uparrow) \to_{\mathrm{SMS}} (ses\uparrow^0)/\mathsf{a}, (s_! \uparrow) & \text{(SMSShiftSubst)} \\
(\phi[id/{}^{ks}x^n]) \to_{\mathrm{SMS}} \phi & \text{(SMSSubstEmpty)} \\
(ses/\mathsf{a}, s_![id/{}^{ks}x^n]) \to_{\mathrm{SMS}} ses[id/{}^{ks}x^n]/\mathsf{a}, (s_![id/{}^{ks}x^n]) & \text{(SMSSubstSubstId)}
\end{array}$$

Figure 3.21: Elimination of $\wr\int$ and normalization of meta-substitutions

identifier, the rule (IdSubstIdLam) increments the levels of both variables in the substitution to avoid capturing. For meta-variables, the rule (IdSubstMVLam) only needs to increment the levels of the replaced identifiers. For applications, the rule (IdSubstApp) propagates the substitution to operator and operands. Two variants are also necessary for `letrec-syntax`. If the substitution replaces the identifier by another identifier, the rule (IdSubstVarLetSyn) increments the level of both identifiers before applying the substitution to transformer and body. For substitutions that replace identifiers by meta-variables, rule (IdSubstMVLetSyn) increments the level of the replaced identifiers in the substitution propagated to the transformer and the body.

$$
\begin{aligned}
&\mathit{IdSubstTerms} \ni c_{\text{ID}} ::= c[u] \\
&\qquad\qquad \rightarrow_{[]} \subseteq \mathit{IdSubstTerms} \times \mathit{MixtureTerms} \\
&{}^{ks}x^n[id/{}^{ks}x^n] \rightarrow_{[]} id && \text{(IdSubstId)} \\
&{}^{ks}x^n[id/{}^{ks'}y^m] \rightarrow_{[]} {}^{ks}x^n \text{ iff } {}^{ks}x^n \neq {}^{ks'}y^m && \text{(IdSubstOther)} \\
&\mathsf{a}[u] \rightarrow_{[]} \mathsf{a} && \text{(IdSubstMV)} \\
&a[u] \rightarrow_{[]} a && \text{(IdSubstConst)} \\
&(\lambda z.ses)[{}^{ks_2}w^m/{}^{ks_1}y^n] \rightarrow_{[]} (\lambda z.ses[{}^{ks_2}w^{m+1}/{}^{ks_1}y^{n+1}]) && \text{(IdSubstIdLam)} \\
&(\lambda z.ses)[\mathsf{a}/{}^{ks}x^n] \rightarrow_{[]} (\lambda z.ses[\mathsf{a}/{}^{ks}x^{n+1}]) && \text{(IdSubstMVLam)} \\
&(@\ ses_1\ ses_2 \ldots)[u] \rightarrow_{[]} (@\ ses_1[u]\ ses_2[u] \ldots) && \text{(IdSubstApp)} \\
&(\mathit{letrec\text{-}syn}\ z\ ses_{tf}\ ses)[{}^{ks_2}y^m/{}^{ks_1}x^n] \rightarrow_{[]} (\mathit{letrec\text{-}syn}\ z\ ses_{tf}[u]\ ses[u]) && \text{(IdSubstVarLetSyn)} \\
&\qquad \text{where } u = {}^{ks_2}y^{m+1}/{}^{ks_1}x^{n+1} \\
&(\mathit{letrec\text{-}syn}\ z\ ses_{tf}\ ses)[\mathsf{a}/{}^{ks}x^n] \rightarrow_{[]} (\mathit{letrec\text{-}syn}\ z\ ses_{tf}[u']\ ses[u']) && \text{(IdSubstMVLetSyn)} \\
&\qquad \text{where } u' = \mathsf{a}/{}^{ks}x^{n+1}
\end{aligned}
$$

Figure 3.22: Elimination of identifier substitutions

Next, we present the elimination rules for the shift operator and the mark operator applied to terms involving meta-variables. These rules extend the reductions from Figure 3.11 and Figure 3.12 respectively. Figure 3.23 contains the new rules for both operators. Rules (ShiftLamMV) and (ShiftLetSynMV) are identical to the variants with ordinary variables. The rule (ShiftMV) drops a shift operator applied to a meta-variable. Likewise unchanged (MarkLamMV) and (MarkLetSynMV) propagate the marks to the transformer and the body and (MarkMV) drops the mark operator.

Now the definitions of all expansion substitutions are in place and the re-definition of the elimination reduction $\rightarrow_{\text{El}}$ is possible. The reduction $\rightarrow_{\text{El}}$ now reduces meta-substitutions and identifier substitutions in addition to parsing substitutions, and the shift and mark operators:

$$\rightarrow_{\text{El}} = \rightarrow_{\uparrow} \cup \rightarrow_{\sigma} \cup \rightarrow_{(\!|} \cup \rightarrow_{\wr\int} \cup \rightarrow_{[]}$$

Its contexts needs to be extended accordingly:

$$EL ::= EL(\!|t|\!) \mid EL\uparrow^n \mid EL\sigma^n \mid EL\wr s \int \mid EL[u] \mid [\,]$$

At this point the expansion reduction $\rightarrow_{\text{Expand}}$ contains a rule to expand macro applications using `es-transformer`, namely (ExpandESTapp). Its rule (ExpandNormDef) can rely on the reduction $\rightarrow_{\text{El}}$ to eliminate all expansion substitutions resulting from this macro application.

As required by the core macro expander at the end of Section 3.5, a transformer specification also has to provide a reduction to eliminate expansion substitutions for transformers ($\rightarrow_{\text{El-Tf}}$) and a reduction for parsing transformers ($\rightarrow_{\text{PT}}$). These reductions come next.

Figure 3.24 contains the definition of $\rightarrow_{\text{El-Tf}}$ for `es-transformer` that is needed to extend $\mapsto_{\text{Def-El-Tf}}$ from page 50. The rules simply pass the expansion substitutions to the transformer

$$\begin{aligned}
&\mathit{ShiftedTerms} \ni c_{\text{shifted}} ::= c{\uparrow^n} \\
&\qquad\qquad \rightarrow_{\uparrow} \subseteq \mathit{ShiftedTerms} \times \mathit{MixtureTerms} \\
&((\lambda \mathsf{a}.\mathit{ses}){\uparrow^n}) \rightarrow_{\uparrow} (\lambda \mathsf{a}.(\mathit{ses}{\uparrow^{n+1}})) && \text{(ShiftLamMV)} \\
&((\mathit{letrec\text{-}syn}\ \mathsf{a}\ \mathit{ses}_{tf}\ \mathit{ses}){\uparrow^n}) \rightarrow_{\uparrow} (\mathit{letrec\text{-}syn}\ \mathsf{a}\ (\mathit{ses}_{tf}{\uparrow^{n+1}})\ (\mathit{ses}{\uparrow^{n+1}})) && \text{(ShiftLetSynMV)} \\
&(\mathsf{a}{\uparrow^n}) \rightarrow_{\uparrow} \mathsf{a} && \text{(ShiftMV)}
\end{aligned}$$

$$\begin{aligned}
&\mathit{MarkedTerms} \ni c_{\text{marked}} ::= c{\male^k} \\
&\qquad\qquad \rightarrow_{\male} \subseteq \mathit{MarkedTerms} \times \mathit{MixtureTerms} \\
&(\lambda \mathsf{a}.\mathit{ses}){\male^n} \rightarrow_{\male} (\lambda \mathsf{a}.\mathit{ses}{\male^n}) && \text{(MarkLamMV)} \\
&(\mathit{letrec\text{-}syn}\ \mathsf{a}\ \mathit{ses}_{tf}\ \mathit{ses}){\male^n} \rightarrow_{\male} (\mathit{letrec\text{-}syn}\ \mathsf{a}\ \mathit{ses}_{tf}{\male^n}\ \mathit{ses}{\male^n}) && \text{(MarkLetSynMV)} \\
&\mathsf{a}{\male^n} \rightarrow_{\male} \mathsf{a} && \text{(MarkMV)}
\end{aligned}$$

Figure 3.23: Elimination of $\uparrow$ and $\male$ for terms containing meta-variables

procedure where they accumulate until the evaluation of the procedure. This implements the propagation of the lexical information surrounding the macro definition to the syntactic objects appearing in the macro output.

$$\begin{aligned}
&\rightarrow_{\text{El-Tf}} \subseteq \mathit{Transformers} \times \mathit{Transformers} \\
&(\mathit{es\text{-}transf}\ e)(\!|r|\!) \rightarrow_{\text{El-Tf}} (\mathit{es\text{-}transf}\ e(\!|r|\!)) && \text{(ParseSubstEST)} \\
&(\mathit{es\text{-}transf}\ e)[u] \rightarrow_{\text{El-Tf}} (\mathit{es\text{-}transf}\ e[u]) && \text{(IdSubstEST)} \\
&(\mathit{es\text{-}transf}\ e)\lfloor s \rfloor \rightarrow_{\text{El-Tf}} (\mathit{es\text{-}transf}\ e\lfloor s \rfloor) && \text{(MetaSubstEST)} \\
&((\mathit{es\text{-}transf}\ e){\uparrow^n}) \rightarrow_{\text{El-Tf}} (\mathit{es\text{-}transf}\ (e{\uparrow^n})) && \text{(ShiftEST)} \\
&(\mathit{es\text{-}transf}\ e){\male^n} \rightarrow_{\text{El-Tf}} (\mathit{es\text{-}transf}\ e{\male^n}) && \text{(MarkEST)}
\end{aligned}$$

$$P[(\texttt{es-transformer}\ \mathit{se})] \rightarrow_{\text{PT}} (\mathit{es\text{-}transf}\ \mathit{eval}_v^{MS}(P[\mathit{expand'}(P[\mathit{se}])])) \qquad \text{(ESTransformer)}$$

$$\begin{aligned}
&\mathit{expand'} : \mathit{S\text{-}Expressions} \rightarrow \mathit{Expressions} \\
&\mathit{expand'}(\mathit{se}) = e \text{ iff } \mathit{se} \mapsto^*_{\text{ExpandExpandUnshift}} e \wedge FV(e) = \emptyset \wedge \mathit{WellFormed}(e)
\end{aligned}$$

Figure 3.24: Elimination and parsing for `es-transformer`

Extending the reduction for parsing transformer is next. We first give a brief review of the requirements: A macro definition using `es-transformer` needs to expand and evaluate the transformer procedure. Expanding the procedure is necessary because Macro Scheme supports macros just like ordinary Scheme. As Macro Scheme contains a few constructs (`syntax` and `syntax-lambda`) not part of ordinary Scheme, it is necessary to extend the macro expander to cover these constructs as well. This is the subject of Section 3.8. The evaluation of the transformer procedure yields a closure, which the expander stores in the set of macro definitions.

The (ESTransformer) rule, also shown in Figure 3.24, satisfies these requirements. It extends the reduction $\rightarrow_{\text{PT}}$ for parsing transformers. The rule splits the transformer into a parsing context and the source code. The parsing context contains the lexical context of the macro definition.

The expansion function *expand'* creates the abstract syntax for the transformer procedure. It is a variant of the function *expand* from the end of Section 3.5 that does not bind the special identifiers and the identifier $\star$. In addition, *expand'* checks, whether the resulting expression is closed and well-formed. If this is not the case but identifiers refer to an outer scope, this corresponds to an "out of context" error. In the rule (ESTransformer), the a argument of *expand'* is the source code of the transformer procedure wrapped into the parsing context of the macro definition. Applying the parsing context to the source code propagates the lexical information of the macro definition to the Macro Scheme source code. This propagation of object level bindings to the meta-level enables macros that expand into macro definitions to use meta-variables in the source code of the transformer procedure. After expansion, the evaluation function $eval_v^{MS}$ from Section 3.6.1 turns the code into a closure. Before, the rule again applies the parsing context to the code, this time to provide the bindings for the special identifiers. The rule (ExpandLetSyn) subsequently adds this transformer in the set of macro definitions and continues with the expansion of the body.

Rule (ExpandParseTransformer) from Figure 3.9 can now use the transitive closure of the standard reduction, $\mapsto_{\mathrm{PT}}$, to parse the transformer argument of a *letrec-syn* form.

3.7 Parsing Scheme with Macros

Section 3.5 added macros to the mixture syntax from Section 3.4. The additions comprise the *letrec-syn* binding construct for macros and macro applications. This section extends the parser from Section 3.4 to generate these terms from the concrete syntax. In the concrete syntax, macros are written as parenthesized s-expressions just as procedure applications. The parser distinguishes the two by consulting the set of transformer bindings: If the first element of a parenthesized s-expression is a symbol and the set of definitions contains a binding for the identifier corresponding to that symbol through the parsing context, the s-expression is a macro application. Otherwise it is a procedure application (unless the identifier is one of the special identifiers). The binding construct for macros is also a parenthesized s-expression where the first element is the special identifier £. Initially the parser binds the symbol `letrec-syntax` to this special identifier. It is followed by a transformer expression that we leave unspecified in the parser. Section 3.5 already contains the rule (ExpandParseTransformer) which uses the reduction $\rightarrow_{\mathrm{PT}}$ to parse the transformer. The final form of `letrec-syntax` is the body where the macro is visible.

Figure 3.26 shows the parser as a reduction system. It builds on the same ideas as the parser in Section 3.4 but the introduction of meta-variables requires a significant extension: The variable bound by a λ-abstraction may be an ordinary identifier or to a meta-variable. In the first case, the λ-abstraction introduces a fresh binding as before. However, if the bound variable is a meta-variable, the parser defers the binding of the variable until the corresponding meta-substitution arrives at the λ-abstraction. The rule (MetaSubstLamMVMV) will then update the substitution to reflect the binding occurrence. The same considerations apply to keywords bound by `letrec-syntax`.

The parser works on the mixture syntax as defined in Figure 3.19. The only extension concerns parsing substitutions: In addition to identifiers, a parsing substitution now may also replace a symbol by a meta-variable. Figure 3.25 contains the new definition of the parsing substitutions. The new parser also needs to keep track of the `letrec-syntax` keyword. As for `lambda`, a parsing substitution with a special symbol aids the parser: initially the substitution $(\!|{}^{\emptyset}\pounds^{0}/\texttt{letrec-syntax}|\!)$ is applied to the source term. Apart from the inclusion of the set of transformer bindings, rule (NestedApp) is the same as (NestedAppSimple). However, (NestedApp) receives a sibling, rule

Extension to Mixture syntax:	
r	$::= {}^{ks}x^{n}/\mathtt{x} \mid \mathtt{a}/\mathtt{x}$

Figure 3.25: Extension of the mixture syntax for parsing

$$\textit{DefLexicalS-Expressions} \ni c_{\text{DefLexSE}} ::= \underline{d} \vdash \textit{ses}$$

$$\rightarrow_{\text{Parse}} \subseteq \textit{DefLexicalS-Expressions} \times \textit{MixtureTerms}$$

$$\underline{d}, k \vdash P[((se_{1_1}\ se_{1_2}\ \ldots)se_2\ \ldots)] \rightarrow_{\text{Parse}} \underline{d}, k \vdash (@\ P[(se_{1_1} se_{1_2} \ldots)]\ P[se_2]\ldots) \quad \text{(NestedApp)}$$

$$\underline{d}, k \vdash P[(\mathtt{x}\ se_2\ \ldots)] \rightarrow_{\text{Parse}} \underline{d}, k \vdash P[(@\ \mathtt{x}\ se_2 \ldots)] \text{ where } P[\mathtt{x}] \mapsto^*_{\text{El}} = \textit{ses} \quad \text{(MVApp)}$$

$$\underline{d}, k \vdash P[(\mathtt{x}\ se_2\ \ldots)] \rightarrow_{\text{Parse}} \underline{d}, k \vdash P[(@\ \mathtt{x}\ se_2 \ldots)] \text{ iff } \neg D({}^{ks}x^n, \underline{d}) \wedge x \notin \textit{Specials} \quad \text{(IdApp)}$$

$$\text{where } P[\mathtt{x}] \mapsto^*_{\text{El}} = {}^{ks}x^n$$

$$\underline{d}, k \vdash P[(\mathtt{x}\ se_2\ \ldots)] \rightarrow_{\text{Parse}} \underline{d}, k \vdash \langle {}^{ks}x^n P[(\mathtt{x}\ se_2\ \ldots)]\rangle \text{ iff } D({}^{ks}x^n, \underline{d}) \quad \text{(MacApp)}$$

$$\text{where } P[\mathtt{x}] \mapsto^*_{\text{El}} = {}^{ks}x^n$$

$$\underline{d}, k \vdash P[(\mathtt{x}\ \mathtt{y}\ se)] \rightarrow_{\text{Parse}} \underline{d}, k \vdash P[(\lambda \mathtt{y}.se(\!|{}^{\emptyset}\mathtt{y}^0/\mathtt{y}|\!))] \quad \text{(LambdaId)}$$

$$\text{iff } P[\mathtt{y}] \mapsto^*_{\text{SymRes}} {}^{ks}y^m \wedge P[\mathtt{x}] \mapsto^*_{\text{El}} \lambda^n$$

$$\underline{d}, k \vdash P[(\mathtt{x}\ \mathtt{y}\ se)] \rightarrow_{\text{Parse}} \underline{d}, k \vdash P[(\lambda \mathtt{b}.se)] \text{ iff } P[\mathtt{y}] \mapsto^*_{\text{SymRes}} \mathtt{b} \wedge P[\mathtt{x}] \mapsto^*_{\text{El}} \lambda^n \quad \text{(LambdaMV)}$$

$$\underline{d}, k \vdash P[(\mathtt{x}\ \mathtt{y}\ se_{\text{tf}}\ se_{\text{body}})] \rightarrow_{\text{Parse}} \quad \text{(LetSynId)}$$

$$\underline{d}, k \vdash P[(\textit{letrec-syn}\ \mathtt{y}\ se_{\text{tf}}(\!|{}^{ks}\mathtt{y}^0/\mathtt{y}|\!)\ se_{\text{body}}(\!|{}^{ks}\mathtt{y}^0/\mathtt{y}|\!))]$$

$$\text{iff } P[\mathtt{y}] \mapsto^*_{\text{SymRes}} {}^{ks}y^n \wedge P[\mathtt{x}] \mapsto^*_{\text{El}} \pounds^n$$

$$\underline{d}, k \vdash P[(\mathtt{x}\ \mathtt{y}\ se_{\text{tf}}\ se_{\text{body}})] \rightarrow_{\text{Parse}} \underline{d}, k \vdash P[(\textit{letrec-syn}\ \mathtt{b}\ se_{\text{tf}}\ se_{\text{body}})] \quad \text{(LetSynMV)}$$

$$\text{iff } P[\mathtt{y}] \mapsto^*_{\text{SymRes}} \mathtt{b} \wedge P[\mathtt{x}] \mapsto^*_{\text{El}} \pounds^n$$

$${}^{ks}x^n\wr s\int \rightarrow_{\wr\int_{\text{SymRes}}} {}^{ks}x^n \quad \text{(SymResSubstMetaId)}$$

$$\mathtt{a}\wr s\int \rightarrow_{\wr\int_{\text{SymRes}}} \mathtt{a} \quad \text{(SymResSubstMetaMV)}$$

$$\rightarrow_{\text{SymRes}} = \rightarrow_{(\!|\,|\!)} \cup \rightarrow_{\uparrow} \cup \rightarrow_{\circlearrowright} \cup \rightarrow_{[]} \cup \rightarrow_{\wr\int_{\text{SymRes}}}$$

Figure 3.26: Parsing reduction $\rightarrow_{\text{Parse}}$ and symbol resolution reduction $\rightarrow_{\text{SymRes}}$

(MVApp), which covers the cases where the first element of a parenthesized expression is a meta-variable that reduces to a compound expression. If the first form within a parenthesized expression is an identifier, the rules (IdApp) and (MacApp) call the function D to search in the set of transformer bindings whether the identifier is a keyword or not. If it is a keyword, (MacApp) produces a macro application, otherwise (IdApp) generates a procedure application. The abstract syntax form of macro applications contains the syntactic variable followed by arguments of the macro wrapped in the surrounding parsing context P.

For the binding constructs λ and `letrec-syntax`, the parser must not use directly the symbol found at the variable/keyword position as the bound identifier: The symbol may stand for a meta-variable and thus for a different identifier. Therefore, the parser uses the $\rightarrow_{\text{SymRes}}$ reduction also defined in Figure 3.26 to resolve the symbol. This reduction works like the normal elimination reduction $\rightarrow_{EL}$ but does not eliminate meta-substitutions. Instead rule (SymResSubstMetaMV) drops a meta-substitution applied to a meta-variable. Depending on whether the identifier resulting from the elimination using $\rightarrow_{\text{SymRes}}$ is a meta-variable or not, the rules generate parsing substitutions for body of the λ-abstraction. The (LambdaId) and (LambdaMV) rules cover the two outcomes of the $\rightarrow_{\text{SymRes}}$ reduction for λ-abstractions. (LambdaId) is applicable, if the symbol reduces to an ordinary identifier and generates a parsing substitution for the body just as its predecessor (LambdaIdSimple). Note, that (LambdaId) uses the original symbol from the input to generate the new identifier. Using the outcome of the $\rightarrow_{\text{SymRes}}$ reduction would be incorrect as it can contain special identifiers like λ^n. If the symbol at the binding position resolves to a meta-variable, rule (LambdaMV) places the meta-variable at the binding position of the λ but generates no parsing substitution. Later, the rule (MetaSubstLamMVId) will modify a meta-substitution that replaces the meta-variable of the λ-abstraction.

The new rules (LetSynId) and (LetSynMV) parse the definition of a local macro. Both rules use the $\rightarrow_{\text{SymRes}}$ reduction as just described and accordingly only the rule (LetSynId) generates a parsing substitution. Again the rule uses the verbatim symbol in the input to build the bound identifier. As `letrec-syntax` is a recursive binding facility, the rule binds the keyword not only in the body but in the transformer as well. For both cases, it generates the same parsing substitution that replaces the symbol by a keyword.

To complete the definition of the parser, the set of parsing contexts needs to be augmented to allow the parser to peek into all expansion substitutions:

$$P \quad ::= P(\!|r|\!) \mid P\uparrow^n \mid P\sigma^n \mid P\wr s\int \mid P[u]$$

The new parser applies parsing substitutions to the complete set of mixture terms. Consequently, the reduction $\rightarrow_{(\!|\,|\!)}$, which eliminates parsing substitutions, needs to be extended, too. Figure 3.27 contains the new definition of $\rightarrow_{(\!|\,|\!)}$, which deals with all terms of the mixture syntax. The rules (ParseSubstStar), (ParseSubstSym), (ParseSubstConst), (ParseSubstId), (ParseSubstSymOther), (ParseSubstLamId) and (ParseSubstApp) remain unchanged. For `letrec-syntax`, (ParseSubstVarLetSyn) increments the level of the replacing identifier by one in the substitution passed to transformer and body. (ParseSubstMV) drops parsing substitutions applied to meta-variables. Finally, rules (ParseSubstMVLam) and (ParseSubstMVLetSyn) pass the substitutions unchanged to the bodies and the transformer if the substitution replaces a symbol by a meta-variable. In this case, there are no levels to be incremented and the rules to eliminate meta-substitutions will take care of the proper scoping. The new definition of $\rightarrow_{(\!|\,|\!)}$ is assumed to be part of $\rightarrow_{\text{El}}$, the elimination reduction for expansion substitutions.

Again, the parsing reduction defines a function *parse*. This time, *parse* also needs to keep track of the `letrec-syntax` keyword. To that end, it replaces the symbol `letrec-syntax` by the special symbol £:

$$\mathit{parse}(se) = e \text{ iff } se(\!|^{\emptyset}\lambda^0/\texttt{lambda}|\!)(\!|^{\emptyset}£^0/\texttt{letrec-syntax}|\!)(\!|^{\emptyset}\star^0/\star|\!) \mapsto^*_{\text{Parse}} e$$

$$
\begin{array}{rl}
\mathit{ParseSubstTerms} \ni c_{\mathrm{PS}} ::= c(\!|r|\!) & \\
\qquad \rightarrow_{(\!|\,|\!)} \subseteq \mathit{ParseSubstTerms} \times \mathit{MixtureTerms} & \\
\mathtt{x}(\!|{}^{ks}\star^{n}/\star|\!) \rightarrow_{(\!|\,|\!)} {}^{ks}\mathtt{x}^{n} & \text{(ParseSubstStar)} \\
\mathtt{x}(\!|\mathit{id}/\mathtt{x}|\!) \rightarrow_{(\!|\,|\!)} \mathit{id} & \text{(ParseSubstSym)} \\
a(\!|r|\!) \rightarrow_{(\!|\,|\!)} a & \text{(ParseSubstConst)} \\
{}^{ks}x^{n}(\!|r|\!) \rightarrow_{(\!|\,|\!)} {}^{ks}x^{n} & \text{(ParseSubstId)} \\
\mathtt{x}(\!|\mathit{id}/\mathtt{y}|\!) \rightarrow_{(\!|\,|\!)} \mathtt{x} \text{ iff } \mathtt{x} \neq \mathtt{y} \wedge \mathtt{y} \neq \star & \text{(ParseSubstSymOther)} \\
(\lambda z.ses)(\!|{}^{ks}w^{n}/\mathtt{y}|\!) \rightarrow_{(\!|\,|\!)} (\lambda z.ses(\!|{}^{ks}w^{n+1}/\mathtt{y}|\!)) & \text{(ParseSubstLamId)} \\
(@\ ses_1\ ses_2 \ldots)(\!|r|\!) \rightarrow_{(\!|\,|\!)} (@\ ses_1(\!|r|\!)\ ses_2(\!|r|\!) \ldots) & \text{(ParseSubstApp)} \\
(\mathit{letrec\text{-}syn}\ z\ ses_{tf}\ ses)(\!|{}^{ks}x^{n}/\mathtt{y}|\!) \rightarrow_{(\!|\,|\!)} & \text{(ParseSubstVarLetSyn)} \\
\qquad (\mathit{letrec\text{-}syn}\ z\ ses_{tf}(\!|{}^{ks}x^{n+1}/\mathtt{y}|\!)\ ses(\!|{}^{ks}x^{n+1}/\mathtt{y}|\!)) & \\
\mathtt{a}(\!|r|\!) \rightarrow_{(\!|\,|\!)} \mathtt{a} & \text{(ParseSubstMV)} \\
(\lambda z.ses)(\!|\mathtt{a}/\mathtt{y}|\!) \rightarrow_{(\!|\,|\!)} (\lambda z.ses(\!|\mathtt{a}/\mathtt{y}|\!)) & \text{(ParseSubstMVLam)} \\
(\mathit{letrec\text{-}syn}\ z\ ses_{tf}\ ses)(\!|\mathtt{a}/\mathtt{y}|\!) \rightarrow_{(\!|\,|\!)} (\mathit{letrec\text{-}syn}\ z\ ses_{tf}(\!|\mathtt{a}/\mathtt{y}|\!)\ ses(\!|\mathtt{a}/\mathtt{y}|\!)) & \text{(ParseSubstMVLetSyn)}
\end{array}
$$

Figure 3.27: Elimination of $(\!|\,|\!)$ substitutions

3.8 Parsing and Macro Expansion for Macro Scheme

Expressions written in Macro Scheme may contain macro definitions and macro applications. The macros facility for Macro Scheme is the same as for ordinary Scheme. This builds up a tower of macro expansion languages. However, Macro Scheme contains some constructs, namely `syntax` and `syntax-lambda`, that are not part of ordinary Scheme. Consequently, parsing and macro expansion for Macro Scheme needs to cover these constructs, too. This is the topic of this section.

Extending the parser is straightforward. For the sake of simplicity, we leave out a check to ensure that the keywords are bound to the predefined denotations. Figure 3.28 contains the two rules.

$$
\begin{array}{rl}
\underline{d}, k \vdash P[\texttt{(x } se\texttt{)}] \rightarrow_{\mathrm{Parse}} \underline{d}, k \vdash P[\lfloor se \rfloor] \text{ iff } P[\mathtt{x}] \mapsto^{*}_{\mathrm{El}} {}^{ks}\mathtt{syntax}^{n} & \text{(ParseSyntax)} \\
\underline{d}, k \vdash P[\texttt{(x y} \ldots se_{\mathrm{body}}\texttt{)}] \rightarrow_{\mathrm{Parse}} \underline{d}, k \vdash P[(\Lambda(\mathtt{y} \ldots).se)] & \text{(ParseSyntaxLambda)} \\
\qquad \text{iff } P[\mathtt{x}] \mapsto^{*}_{\mathrm{El}} {}^{ks}\mathtt{syntax\text{-}lambda}^{n} &
\end{array}
$$

Figure 3.28: Parsing `syntax` and `syntax-lambda`

Figure 3.29 extends the core macro expander from Figure 3.9 with the rules for `syntax` and `syntax-lambda`. Expansion does not affect a `syntax` expression because with regard to macro expansion the syntactic object is just data where no macro expansion takes place. Hence rule (ExpandSyntax) only drops the set of transformer bindings and reduces to the unchanged `syntax` form. Likewise, for `syntax-lambda`, rule (ExpandSynLam) only has to propagate the set of transformer bindings to the body. This rule does not shift the definitions because `syntax-lambda` does not affect identifiers of Macro Scheme.

Expansion must now also take place in the body of a `syntax-lambda` form, hence we extend the set of expansion contexts from Section 3.5 by a new rule:

$$EP ::= \ldots \mid (\Lambda\ (x \ldots).\ EP)$$

$$\underline{d}, k \vdash \lfloor se \rfloor \rightarrow_{\text{Expand}} \lfloor se \rfloor \qquad \text{(ExpandSyntax)}$$
$$\underline{d}, k \vdash (\Lambda(\mathtt{x}\ldots).ses) \rightarrow_{\text{Expand}} (\Lambda(\mathtt{x}\ldots).\underline{d}, k \vdash ses) \qquad \text{(ExpandSynLam)}$$

Figure 3.29: Expanding Macro Scheme special forms

Next, we need to extend the elimination of expansion substitutions and the unshift operator to `syntax` and `syntax-lambda` forms. For the parsing substitution, we need to extend the rules from Figure 3.6, for the shift, mark and unshift operators, it is the rules from Figures 3.11, 3.12, and 3.13, and for the meta- and identifier substitutions, the rules from Figures 3.21 and 3.22. Figure 3.30 contains the new rules, which all follow the same pattern: The substitutions do not affect `syntax` forms and propagate unchanged to the body of `syntax-lambda` forms. The reason why the expansion substitutions do not affect the forms even though both are related to macro expansion is that *expansion* for Macro Scheme (our current topic) affects the source terms of Macro Scheme but the `syntax` and `syntax-lambda` forms describe macro expansion during the *evaluation* of Macro Scheme. That is, the expansion substitutions of Macro Scheme take place one phase before the macro expansion of ordinary Scheme and just like variable bindings in Macro Scheme do not affect `syntax` forms, meta-variable bindings during the expansion of Macro Scheme do not affect these forms.

$$\lfloor ses \rfloor (\!|r|\!) \rightarrow_{(\!|\,|\!)} \lfloor ses \rfloor \qquad \text{(ParseSubstSyn)}$$
$$(\Lambda(\mathtt{x}\ldots).ses)(\!|r|\!) \rightarrow_{(\!|\,|\!)} (\Lambda(\mathtt{x}\ldots).ses(\!|r|\!)) \qquad \text{(ParseSubstSynLam)}$$
$$(\lfloor ses \rfloor \uparrow^n) \rightarrow_{\uparrow} \lfloor ses \rfloor \qquad \text{(ShiftSyn)}$$
$$((\Lambda(\mathtt{x}\ldots).ses)\uparrow^n) \rightarrow_{\uparrow} (\Lambda(\mathtt{x}\ldots).(ses\uparrow^n)) \qquad \text{(ShiftSynLam)}$$
$$\lfloor ses \rfloor \mathord{\male}^n \rightarrow_{\male} \lfloor ses \rfloor \qquad \text{(MarkSyn)}$$
$$(\Lambda(\mathtt{x}\ldots).ses)\mathord{\male}^n \rightarrow_{\male} (\Lambda(\mathtt{x}\ldots).ses\mathord{\male}^n) \qquad \text{(MarkSynLam)}$$
$$\lfloor ses \rfloor \wr s \int \rightarrow_{\wr\int} \lfloor ses \rfloor \qquad \text{(MetaSubstSyn)}$$
$$(\Lambda(\mathtt{x}\ldots).ses)\wr s \int \rightarrow_{\wr\int} (\Lambda(\mathtt{x}\ldots).ses\wr s \int) \qquad \text{(MetaSubstSynLam)}$$
$$\lfloor ses \rfloor [u] \rightarrow_{[]} \lfloor ses \rfloor \qquad \text{(IdSubstSyn)}$$
$$(\Lambda(\mathtt{x}\ldots).ses)[u] \rightarrow_{[]} (\Lambda(\mathtt{x}\ldots).ses[u]) \qquad \text{(IdSubstSynLam)}$$
$$(\lfloor ses \rfloor \downarrow^n) \rightarrow_{\downarrow} \lfloor ses \rfloor \qquad \text{(UnshiftSyn)}$$
$$((\Lambda(\mathtt{x}\ldots).e)\downarrow^n) \rightarrow_{\downarrow} (\Lambda(\mathtt{x}\ldots).(e\downarrow^n)) \qquad \text{(UnshiftSynLam)}$$

Figure 3.30: Elimination of the expansion substitutions and the $\downarrow$ operator for `syntax` and `syntax-lambda`

3.8.1 Scoping Issues Between Object And Meta-Language

Using explicit substitutions for macro expansion raises a number of questions about the relationship between the scoping in the meta and the object language. The big advantage of using explicit substitutions instead of renaming is that we can chose what to do. The following enumeration lists the cases we have identified, explains how the semantics from the previous section resolve the issue, and shows how the well-established `syntax-case` facility operates in each case.

Three guidelines have directed the design of our semantics:

- The syntactic objects represent data during the evaluation of the meta-language, hence there should be no interaction between bindings in the meta language and the variables within the syntactic objects.

- The evaluation of meta-language and object language is temporally separate, hence there should be no interaction between bindings in the object language and variables in the meta-language.
- Macros expanding to definitions of other macros should be fully supported.

The `syntax-case` implementations do not provide an equivalent to our `syntax-lambda` form but only the full `syntax-case` form which also performs pattern-matching. However, the formal model of `syntax-case` provides a *plambda* expression that serves the same purposes as our `syntax-lambda`[DHB92]. Our remarks about `syntax-case` refer to this construct.

1. Should a local variable binding in the meta language affect the argument of `syntax`? For example in

```
(lambda (a)
  (let-syntax ((baz (es-transformer
                       (lambda (mcall)
                           (let ((a 12))
                              (syntax a))))))
     (baz)))
```

   should the `a` in `(syntax a)` refer to the binding of `lambda` or to the binding of the `let` and yield an "out-of-context" error?

   `Syntax-case` yields an "out-of-context" error[DHB92].

   In our model, `a` is bound by the object-level `lambda` because the meta-level binding does not affect variables within syntactic objects.

   Emulating the behavior of `syntax-case` is also possible: the elimination rules for the expansion substitutions and the unshift operator would need to be adapted for the $\lfloor\ \rfloor$ cases. More specifically, the rules (ParseSubstSyn), (ShiftSyn), (MetaSubstSyn), (IdSubstSyn), and (UnshiftSyn) from Figure 3.30 would pass the substitutions to the lexical s-expression of the syntactic object instead of dropping them.

2. Should `syntax-lambda` affect normal bindings in the meta-language? E.g. in

```
(let-syntax ((m (es-transformer
                  (lambda (mcall)
                    ...
                    (syntax-lambda syntax-car
                       (syntax-car (syntax (1 2))))
                    ...))))
   ...)
```

   should the `syntax-car` in `(syntax-car (syntax (1 2)))` refer to the `syntax-lambda` and yield an "illegal use of syntax" error or should it refer to the default primitive?

   In our model, the variable `syntax-car` is bound to the primitive because the scope of `syntax-lambda` covers only (meta-)variables within syntactic objects. `Syntax-lambda` generates the meta substitution during the evaluation of the transformer but the expression in question is parsed during the definition of the transformer.

   `Syntax-case` signals an "illegal use of syntax" error.

   Emulating `syntax-case` is also possible: the rule for parsing `syntax-lambda`, (ParseSyntaxLambda) in Figure 3.28, would need to generate the parsing substitution for the meta-variables that is now part of (SyntaxBeta).

3. Should a local variable binding in the object language affect normal bindings in the meta-language? E.g. in

```
(let ((syntax-car 1))
   (let-syntax ((m (es-transformer
                     (lambda (mcall) (syntax-car
                                       (syntax-cdr mcall))))))
      (m)))
```

should the `syntax-car` in `(syntax-car (syntax-cdr stx))` refer to the `let` and yield an "out-of-context" error or should `syntax-car` be bound to the default primitive?

Our model and `syntax-case` yield an "out-of-context" error. In our model, the reason is that rule (ESTransformer) applies the expansion context of the macro definition to the source s-expression of the transformer procedure before applying *expand'* to it (see Figure 3.24). The expansion context contains the lexical information surrounding the macro definition. The motivation for propagating this information is that if a macro expands into the definition on another macro, it is necessary to propagate the expansion substitutions to the transformer procedure in case its source contains meta-variables.

If the bindings of the object language should not affect bindings in the meta-language, the parsing context could be removed. Then the full expansion function *expand* should be used as it also contains the parsing substitutions to bind the special identifiers and unbound identifiers.

4. Should `syntax-lambda` bind meta-variables that occur within generated transformer procedures? E.g. in

```
(let ((a 23))
  (let-syntax ((m (es-transformer
                    (lambda (mcall)
                      ((syntax-lambda x
                         (syntax
                            (let-syntax ((n (es-transformer
                                              (lambda (mcall2)
                                                 (if x
                                                     (syntax 1)
                                                     (syntax 2))))))
                                  (n))))
                       (syntax-cadr mcall))))))
        (m a)))
```

should the `x` in `(if x ...)` be replaced by the argument of `m` or should `x` be unbound?

Our model replaces `x`, because the meta-variable occurs within a syntactic object. `Syntax-case` also replaces `x`.

As can be seen in the example, replacing might introduce object language variable occurrences into the meta-language. Binding these occurrences requires pushing parsing and meta substitutions to the procedure of `es-transformer` during parsing/expansion. The rule (ES-Transformer) accomplishes this by applying the parsing context to code of the transformer procedure before calling *expand'*.

5. Should the identifiers inserted by a macro all be `bound-identifier=?`? E.g. should

```
(letrec-syntax
    ((m (es-transformer
```

```
          (lambda (mcall)
            ((syntax-lambda x
               (syntax
                 (let ((temp 23)) x)))
             (syntax temp))))))
  (m))
```

expand into `(let ((temp 23)) temp)` or should the `temp` in the body of the `let` be unbound?

In our model, the lexical information is propagated to the `syntax`-terms from where macro expansion continues. `Syntax-lambda` inserts the meta-variables hygienically into the syntax objects, hence `temp` would be unbound.

`Syntax-case` uses the same renaming function for the macro definition, hence identical names become equal identifiers and the example expands into `(let ((temp 23)) temp)`.

6. Should the use of meta-variables as binding variables affect identifier occurrences within code that substitutes meta-variables bound by different `syntax-lambda`? That is, should `syntax-lambda` extend the same (current) meta-substitution or should each `syntax-lambda` form introduce its own meta-substitution?

   In our model there is a special current meta-substitution and each `syntax-lambda` extends it. This is compatible with `syntax-case` where the renaming is global.

   If the `syntax-lambda` forms should not affect each other, we would abandon the current meta-substitution and let each `syntax-lambda` introduce its own meta-substitution, valid within its body.

3.9 Semantics of `syntax-rules`

This section defines a formal semantics for the `syntax-rules` transformer from R^{5}RS. Unlike the `es-transformer` facility from Section 3.6, `syntax-rules` macros are not based on a full programming language but use pattern matching and rewriting to transform the macro application. Consequently, it is less powerful but also easier to understand. To reflect this simplicity in the semantics, we do not translate `syntax-rules` macros into `es-transformer` macros but define the semantics directly using a small pattern matcher and the expansion substitutions introduced above. The resulting system is easy to understand and very precise. To estimate the mental overhead involved in a semantics based on translating `syntax-rules` to a computational macro facility, consider the scoping issues from the previous section: none of these issues arise in the simple model of `syntax-rules`. This semantics is the first formal semantics of `syntax-rules`.

A `syntax-rules` form consists of a list of identifiers called *literals* and a list of rules. Each rule has a pattern as its left-hand side and a template as its right-hand side. If the macro expander detects a macro application and the corresponding keyword is bound to a `syntax-rules` transformer, the expander tries to match the arguments of the macro application against the patterns of the rules in the order the rules appeared in the `syntax-rules` form. During matching, a pattern identifier that is listed as a literal of the `syntax-rules` form matches the input only if the input is also an identifier and if both identifiers refer to the same lexical binding. In addition, two literal identifiers match if they are both unbound. A pattern identifier that is not a literal counts as a pattern variable and matches any input. For the first matching rule, the macro expander outputs the rule's template with the pattern variables in the matching pattern replaced by the corresponding input forms of the macro application. If no rule matches, the macro expander signals a syntax error.

As for the `letrec-syntax` form, we again simplify the syntax for `syntax-rules` to shorten the presentation. Our variant of `syntax-rules` contains only one rule whose pattern and template follow the list of literals. The expander matches the macro application form against the pattern

and returns the template in case of a match. If the macro application does not match the pattern, the expander signals a syntax error. To support more rules—as in the standard `syntax-rules` form—the expander would try to match the other rules before signaling an error. In addition our variant of `syntax-rules` contains no ellipsis patterns. Section 3.10 sketches how the semantics would incorporate them.

The macro expander uses meta-substitutions to represent the replacement of pattern variables by input forms. To expand the application of a macro, the expander first constructs a meta-substitution that maps the meta-variables to the corresponding parts of the arguments of the macro application. Then the expander replaces the macro application by the template and applies the generated meta-substitution to the template. During the subsequent expansion process, the parser will resolve the meta-variables and thereby replace the meta-variables by the input of the former macro application.

If a macro expands into the definition of a macro and the expansion uses a meta-variable to construct a binding form for meta-variables, it is necessary to modify the meta-substitution to reflect this new binding. Consider the following example:

```
(define-syntax gensr
  (syntax-rules ()
     ((gensr pat v)
      (letrec-syntax ((m (syntax-rules ()
                           ((m pat) v))))
          (m (12 34))))))
(gensr (x y) x)
```

Here, `gensr` generates a syntax binding and uses the meta-variable `pat` to specify the pattern of the generated macro. This means that the identifiers that replace `pat` become meta-variables. In the example above, this concerns the identifiers `x` and `y`. This change must be propagated to the code that replaces the meta-variable `v`. Hence, during expansion of the pattern `(m pat)`, the meta-substitution must be updated by an identifier substitution that replaces identifiers by meta-variables.

A minor peculiarity concerns the treatment of the keyword during pattern matching. R^{5}RS contains this statement:

> The keyword at the beginning of the pattern in a syntax rule is not involved in the matching and is not considered a pattern variable or literal identifier.
>
> *Rationale:* The scope of the keyword is determined by the expression or syntax definition that binds it to the associated macro transformer. If the keyword were a pattern variable or literal identifier, then the template that follows the pattern would be within its scope regardless of whether the keyword were bound by `let-syntax` or by `letrec-syntax`.

We meet this specification by removing the keyword from the pattern during parsing and involving only the macro's arguments in the matching phase. Unfortunately, removing the keyword from the pattern is slightly tricky for macros that are generated by other macros. In this case, the pattern of the generated macro may be a pattern variable of the generating macro:

```
(let-syntax ((foo (syntax-rules ()
                    ((foo m pat)
                      (let-syntax m (syntax-rules ()
                                      (pat 23))
                            (m (1 2)))))))
   (foo bar (bar (x y))))
```

During the parsing of the inner `let-syntax` definition, the pattern of the macro is just `pat` and removing the keyword is not possible until the elimination of the corresponding meta-substitution. Therefore, the parser removes the keyword only if the pattern is a list of identifiers. If it is a meta-variable, the elimination of the meta-substitution is responsible for removing the keyword.

$$
\begin{array}{lll}
\underline{\mathtt{a}}, \underline{\mathtt{b}} & \in & BindingMVars \\
pat, & \in & Patterns \\
\underline{pat} & \in & NormalizedPatterns \\
pato & \in & OutsideNormalizedPatterns \\
litl & \in & LiteralList \subset Patterns \\
\underline{litl} & \in & NormalizedLiteralList \subset NormalizedPatterns \\
\\
r & ::= & {}^{ks}x^{n}/\mathtt{x} \mid \mathtt{a}/\mathtt{x} \mid \underline{\mathtt{a}}\ /\mathtt{x} \\
tf & ::= & (es\text{-}transf\ e) \mid (syn\text{-}rules\ pat\ litl\ ses) \mid tf(\!|r|\!) \mid tf\uparrow^{n} \mid tf\male^{k} \mid tf\lbag s\rbag \mid tf[u] \\
pat & ::= & n \mid {}^{ks}x^{n} \mid \mathtt{a} \mid \underline{\mathtt{a}} \mid (pat\ .\ \ pat) \mid pat(\!|r|\!) \mid pat\uparrow^{n} \mid pat\male^{k} \mid pat\lbag s\rbag \mid pat[u] \\
\underline{pat} & ::= & a \mid {}^{ks}x^{n} \mid \mathtt{a} \mid \underline{\mathtt{a}} \mid (\underline{pat}\ .\ \ \underline{pat}) \\
pato & ::= & a \mid {}^{ks}x^{n} \mid \mathtt{a} \mid \underline{\mathtt{a}} \mid (pat\ .\ \ pat) \\
l & ::= & \mathtt{x} \mid \mathtt{a} \\
litl & ::= & (l\ldots) \mid \mathtt{a} \mid litl(\!|r|\!) \mid litl\uparrow^{n} \mid litl\male^{k} \mid litl\lbag s\rbag \mid litl[u] \\
\underline{litl} & ::= & ({}^{ks}x^{n}\ \ldots)
\end{array}
$$

Figure 3.31: Mixture syntax for expanding `syntax-rules`

3.9.1 Macro Expansion for `syntax-rules`

We now turn to the formal definition of macro expansion for `syntax-rules` transformers. This section presents the reduction rules that extend the mixture syntax and the core macro expander from Section 3.5 to cover macro applications with `syntax-rules` transformers. The next two sections describe the elimination of the expansion substitutions for `syntax-rules` transformers and patterns.

The mixture syntax needs two extensions to cover `syntax-rules`:

- a representation for `syntax-rules` transformers
- a grammar for representing patterns and literals.

Figure 3.31 contains the new definitions. It extends the definitions from Figure 3.19, which in turn extends Figure 3.8. A `syntax-rules` transformer is written as (*syn-rules pat litl ses*), where *pat* is the pattern, *litl* the list of literals, and *ses* the template.

A pattern *pat* is either a constant, written n, or an identifier, written as usual ${}^{ks}x^{n}$, or a pair of two patterns, written `(`*pat*`.  `*pat*`)`. The set of constants is assumed to contain the empty list (`()`). Furthermore, patterns contain meta-variables that can play two different roles: Either the meta-variable represents a bound variable of the pattern, or the meta-variable occurs in the pattern because the pattern was itself generated by a macro application and the meta-variable will be replaced by some other term later. As an example for the second case, consider this example:

```
(let-syntax
    ((gen-bar (syntax-rules ()
                ((gen-bar a)
                 (let-syntax ((bar (syntax-rules ()
                                     ((bar a) 23))))
                   (bar 1))))))
  (gen-bar 1))
```

The variable `a` is a bound variable in the pattern `(gen-bar a)`. However, in the macro application `(gen-bar 1)`, `a` in the pattern `(bar a)`, is not a bound variable but a meta-variable. The corresponding meta-substitution will replace it by `1`. To distinguish these two kinds of meta-variable occurrences within patterns, we refer to meta-variables that are bound variables of the pattern as *binding meta-variables* and write them as $\underline{\mathtt{a}}$. Meta-variables that occur within patterns because the pattern is the output of an outer macro application remain unchanged, i.e. are still written as

`a`. In the example above, the `a` within the pattern `(gen-bar a)` is a binding meta-variable, while its occurrence within `(bar a)` is an ordinary meta-variable. Of course, a meta-variable may be substituted by a binding meta-variable. In the example above, a macro application `(gen-bar c)` would produce this case and during macro expansion `c` would become a binding meta-variable. Binding meta-variables emerge from symbols within the pattern of a `syntax-rules` form through extended parsing substitutions that replace symbols by binding meta-variables. The new definition of parsing substitutions is also shown in Figure 3.31.

To continue with the definitions of patterns in the abstract syntax, expansion substitutions can also be applied to patterns. Patterns without expansion substitutions are called *normalized* and are written as $\underline{pat}$. Section 3.9.3 contains the definition of the reduction $\mapsto_{\text{patnorm}}$that turns patterns into normalized patterns. Sometimes it is necessary to access the first or second component of a pair pattern. This is only possible if no expansion substitutions wrap the pair. The domain *OutsideNormalizedPatterns* describes the patterns without expansion substitutions at the outermost level. Section 3.9.3 presents a reduction $\mapsto_{\text{patout}}$, which resolves outer expansion substitutions and thus produces elements of *OutsideNormalizedPatterns*.

The list of literals of a `syntax-rules` transformer is either a sequence of identifiers or a meta-variable that is to be replaced by a sequence of identifiers later during parsing. We do not introduce a new domain for literals but perceive them as a subset of patterns. This simplifies the forthcoming presentation.

Section 3.9.4 introduces the accompanying concrete syntax and the extensions for the parser. The parser is responsible for generating meta-variables both within the expression terms and within patterns. The latter requires it to consult the list of literals.

Figure 3.32 contains two rules to handle macro applications with `syntax-rules` transformers. The rules use the helper function D from Figure 3.20 to retrieve the macro definition from the set of transformer bindings. Next, the rules need to check whether the arguments of the macro application matches the pattern from the definition. To that end, both rules call the function *match* from Figure 3.33, which takes as its input a normalized pattern and a lexical s-expression and checks whether the s-expression matches the pattern. It does not need to take the list of literals into account: the parser resolves literals as it parses the pattern and inserts the corresponding identifiers into the pattern.

$$\underline{d}, k \vdash \langle {}^{ks}x^{n}\, ses\rangle \rightarrow_{\text{Expand}} \underline{d}, k+1 \vdash ses_{rhs}\sigma^{\nearrow k}\wr \mathit{gen\text{-}subst}(\underline{pat}, ses, \phi)\int \qquad \text{(ExpandMappSR)}$$
$$\text{iff } D({}^{ks}x^{n}, \underline{d}) = (\mathit{syn\text{-}rules}\ \underline{pat}\ ()\ ses_{rhs}) \wedge \mathit{match}(\underline{pat}, ses)$$
$$\underline{d}, k \vdash \langle {}^{ks}x^{n}\, ses\rangle \rightarrow_{\text{Expand}} \text{syntax error} \qquad \text{(ExpandMappSRFail)}$$
$$\text{iff } D({}^{ks}x^{n}, \underline{d}) = (\mathit{syn\text{-}rules}\ \underline{pat}\ ()\ ses_{rhs}) \wedge \neg \mathit{match}(\underline{pat}, ses)$$

Figure 3.32: Expansion for `syntax-rules` macros

In case of a mismatch, rule (ExpandMappSRFail) reduces to a syntax error. Otherwise, rule (ExpandMappSR) calls the function *gen-subst* to generate the meta-substitution that maps meta-variables to arguments of the macro application. The right-hand side of the rule (the template) may generate identifiers, hence the rule applies the mark operator with the current mark to it. Finally, the rule augments the marked template with the meta-substitution and returns the resulting form. The rule increments the current mark by one to provide a fresh mark for subsequent macro applications.

Figure 3.33 contains the definitions of *match* and *gen-subst* and their helper functions. As pattern matching for `syntax-rules` should not take the keyword into account, *match* first removes it from the lexical s-expression that represents the macro application. To that end, it uses the reduction $\mapsto_{\text{patout}}$to resolve expansion substitutions on the top of the lexical s-expression and calls its helper function *match*′ with the `cdr` of the resulting pair. Section 3.9.3 presents the definition of $\mapsto_{\text{patout}}$. *match*′ performs the pattern matching proper. It dispatches according to the type of

the pattern. For constant patterns, it resolves all pending substitutions on the input form and compares the result with the pattern. For resolving the substitutions, *match′* uses the reduction $\mapsto_{\text{patnorm}}$that eliminates expansion substitutions for patterns. Its definition is also in Section 3.9.3. If the pattern is an identifier, *match′* also resolves the expansion substitutions within the input using $\mapsto_{\text{patnorm}}$. The pattern matches the input if both are equal according to *free-id=?*, that is, if name and level match. The marks do not need to match because R^{5}RS says (Section 4.3.2):

> A subform in the input matches a literal identifier if and only if it is an identifier and either both its occurrence in the macro expression and its occurrence in the macro definition have the same lexical binding, or the two identifiers are equal and both have no lexical binding.

Hence, only the binding place is crucial, not the identifier occurrence. Indeed, the marks of the pattern and the input usually differ as the pattern is part of the transformer and is therefore marked at macro application. In the next case of *match′*, binding meta-variables match anything. If the pattern is a pair, *match′* uses the reduction $\mapsto_{\text{patout}}$to push the substitutions inwards the input form and thus to reveal whether the input is a pair or not. The generation of the meta-substitution with the function *gen-subst* can assume that the input matches the pattern. Just like *match*, it first strips the keyword. Then it calls its helper function *gen-subst′* to generate the substitution. *gen-subst′* just needs to traverse the pattern and the s-expression and append a new pair to the substitution whenever it comes across a binding meta-variable.

3.9.2 Elimination for `syntax-rules` Forms

Analogously to the elimination for the `es-transformer` in Figure 3.24, this section presents the elimination rules for the parsing substitution, the meta-substitution, the shift operator, the mark operator and the identifier substitution applied to `syntax-rules` transformers. The elimination is necessary because the elimination rules for *letrec-syn*—(ParseSubstVarLetSyn) and (ParseSubstMVLetSyn) in Figure 3.27, (ShiftLetSyn) in Figure 3.11, (MarkLetSyn) in Figure 3.12, (MetaSubstLetSyn) and friends in Figure 3.21, and (IdSubstVarLetSyn) and (IdSubstMVLetSyn) in Figure 3.22—propagate the expansion substitutions to the transformer and rule (ExpandNormDef) from Figure 3.9 eliminates them using the reduction $\rightarrow_{\text{Def-El-Tf}}$, which in turn relies on the reduction $\rightarrow_{\text{El-Tf}}$ to eliminate the expansion reductions for transformers. Therefore, the specification of a transformer needs to include a definition of $\rightarrow_{\text{El-Tf}}$ for the introduced transformer terms.

The elimination of the parsing substitution for `syntax-rules` forms inserts the binding meta-variables into the pattern and generates the corresponding parsing substitution that replaces symbols by meta-variables in the template. Two circumstances heavily complicate the specification of these rules:

1. The list of literals determines which symbols in the source pattern are binding meta-variables and which are literal identifiers. This feature is specific to the `syntax-rules` facility and requires several special cases during parsing.

2. Meta-variables from surrounding macro applications can occur within the pattern and within or as the list of literals as shown in the example macro `gen-bar` from Section 3.9.1. Analogously to λ-abstractions with meta-variables as bound variables, the parser needs to resolve these meta-variables to see the names of the actual meta-variables being bound.

Figure 3.34 contains the rules for elimination of the the parsing substitution as reduction $\rightarrow_{\text{SR}}$. The rules consult the list of literals to decide whether the symbol which the substitution is replacing must become a literal identifier or a binding pattern variable within the pattern. Also, whenever a parsing substitution recognizes a symbol as a literal, it removes the symbol from the list of literals: in this case the corresponding identifier has been identified and the substitution will replace the symbol by this identifier within the pattern. Subsequent parsing substitutions may also replace this symbol but the one that comes first has been generated by the innermost binder

$$\mapsto_{\text{patout}} \subseteq \mathit{Patterns} \times \mathit{OutsideNormalizedPatterns}$$
$$\mapsto_{\text{patnorm}} \subseteq \mathit{Patterns} \times \mathit{NormalizedPatterns}$$

$$\begin{aligned}
\mathit{match} : (\mathit{NormalizedPatterns} \times \mathit{MixtureTerms}) &\to \mathit{Boolean} \\
\mathit{match}(\underline{\mathit{pat}}, \mathit{ses}) &= \mathit{match}'(\underline{\mathit{pat}}, \mathit{ses}_r) \\
\mathit{match}' : (\mathit{NormalizedPatterns} \times \mathit{MixtureTerms}) &\to \mathit{Boolean} \\
\mathit{match}'(a, \mathit{ses}) &= \mathit{true} \text{ iff } \mathit{ses} \mapsto^*_{\text{patnorm}} a \\
\mathit{match}'({}^{ks}x^n, \mathit{ses}) &= \mathit{true} \\
&\quad \text{iff } \mathit{ses} \mapsto^*_{\text{patnorm}} {}^{ks'}y^m \wedge \mathit{free\text{-}id{=}?}({}^{ks}x^n, {}^{ks'}y^m) \\
\mathit{match}'(\underline{\mathtt{a}}, \mathit{ses}) &= \mathit{true} \\
\mathit{match}'((\mathit{ses}_l \ \mathtt{.}\ \mathit{ses}_l), \mathit{ses}) &= \mathit{match}'(\mathit{ses}_l, \mathit{ses}_1) \wedge \mathit{match}'(\mathit{ses}_l, \mathit{ses}_2) \\
&\quad \text{iff } \mathit{ses} \mapsto^*_{\text{patout}} (\mathit{ses}_1 \ \mathtt{.}\ \mathit{ses}_2) \\
\mathit{match}'(\underline{\mathit{pat}}, \mathit{ses}) &= \mathit{false} \text{ otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathit{gen\text{-}subst} :& (\mathit{NormalizedPatterns} \times \mathit{MixtureTerms} \times \mathit{MetaSubsts}) \to \\
& \mathit{MetaSubsts} \\
\mathit{gen\text{-}subst}(\underline{\mathit{pat}}, \mathit{ses}, s) &= \mathit{gen\text{-}subst}'(\underline{\mathit{pat}}, \mathit{ses}_{\text{cdr}}, s) \text{ where } \mathit{ses} \mapsto^*_{\text{patout}} (\mathit{ses}_{\text{car}} \,\mathtt{.}\, \mathit{ses}_{\text{cdr}}) \\
\mathit{gen\text{-}subst}' :& (\mathit{NormalizedPatterns} \times \mathit{MixtureTerms} \times \mathit{MetaSubsts}) \to \\
& \mathit{MetaSubsts} \\
\mathit{gen\text{-}subst}'(a, \mathit{ses}, s) &= s \\
\mathit{gen\text{-}subst}'({}^{ks}x^n, \mathit{ses}, s) &= s \\
\mathit{gen\text{-}subst}'(\underline{\mathtt{a}}, \mathit{ses}, s) &= \mathit{ses}/\mathtt{a}, s \\
\mathit{gen\text{-}subst}'((\mathit{ses}_l \ \mathtt{.}\ \mathit{ses}_r), \mathit{ses}, s) &= \mathit{gen\text{-}subst}'(\mathit{ses}_l, \mathit{ses}_1, \mathit{gen\text{-}subst}'(\mathit{ses}_r, \mathit{ses}_2, s)) \\
& \quad \text{where } \mathit{ses} \mapsto^*_{\text{patout}} (\mathit{ses}_1 \ \mathtt{.}\ \mathit{ses}_2)
\end{aligned}$$

Figure 3.33: Helper functions for macro expansion

and is therefore the correct one according to the rules of lexical scoping. If the ⋆-substitution arrives at the `syntax-rules` form, all remaining symbols in the list of literals denote unbound identifiers and occurrences of these symbols in the pattern must be replaced by identifiers that receive the level of the ⋆-substitution whereas symbols that do not appear in the list of literals become (binding) meta-variables. Here is a description of the rules that implement these ideas:

ParseSubstSRSymLit If the symbol of the substitution is a literal, the substitution is passed to the pattern unchanged. There, the substitution will turn the symbol into an identifier to be matched with the input literally. The rule also passes the substitution unchanged to the template. The symbol is removed from the list of literals to prevent the ⋆-substitution from recognizing the symbol as a literal once more.

ParseSubstSRSymNoOccur If the symbol does not occur in the pattern and in the literal list, the rule passes the substitution unchanged to the template because then the `syntax-rules` form does not bind the symbol. The helper function *in-pat?* tests whether a symbol occurs within a normalized pattern. It is defined at the end of the figure.

ParseSubstSRMVLitNoStar Here, the literal list is a meta-variable which means that no literals are specified verbatim and the rule may push the substitution unchanged to the pattern and the template. The elimination of meta-substitutions will later resolve the meta-variable and adapt the pattern according to the list that replaces the meta-variable.

ParseSubstSRIdMVB A symbol `x` that occurs in the pattern but does not occur in the literal list becomes a binding meta-variable $\underline{x}$ in the pattern and a meta-variable `x` in the template. That is, the names $\underline{x}$ and `x` emerge from the name of the symbol.

ParseSubstSRMV A substitution that replaces a symbol by a meta-variable can be passed unchanged to pattern, literals and template. The corresponding meta-substitution will take care of building the pattern and binding meta-variables.

ParseSubstSRStarLitList If the parsing substitution for unbound identifiers reaches a `syntax-rules` term, all remaining symbols in the pattern become (binding) meta-variables provided they do not occur in the list of literals. This is analogous to rule (ParseSubstSRIdMVB). Symbols that are members of the list of literals, become literal identifiers as demonstrated by rule (ParseSubstSRSymLit). The function *GenPatStar* generates a new pattern and a sequence of parsing substitutions for the template. The definition of this function is shown in the same figure. It takes as argument the normalized pattern and the literal list, the term to be augmented by the meta-substitution and the level of the ⋆-identifier. It then walks along the pattern and if it encounters a symbol that is a member of the literal list, it returns as pattern an identifier generated from the level of the ⋆-symbol and binds the symbol to the identifier in the s-expression. If it encounters a symbol that is not in the literal list or the list of literals is a meta-variable, it returns as pattern a binding meta-variable and binds the symbol to a meta-variable in the s-expression to a meta-variable. The function *remove-symbols* afterwards deletes all symbols from the list of literals as the ⋆-substitution has also removed them from the pattern.

ParseSubstSRStarMVLit This rule also covers the case that the ⋆-substitution arrives at a `syntax-rules` term, but this time the list of literals is a meta-variable. Just as (ParseSubstSRStarLitList), the rule calls *GenPatStar* to generate the pattern and the parsing substitutions for the template. The only difference is that the rule does not need to remove symbols from the literal list.

The rules above take over most of the work for parsing `syntax-rules` forms, the elimination of parsing substitutions within patterns requires only a couple of trivial rules that will be shown later in Section 3.9.3.

As we have seen above, meta-variables can occur as and within patterns and literal lists. Therefore, the elimination of meta-substitutions needs to be extended, too. However, the `syntax-`

$$\begin{aligned} \mathit{ParseSubstSR} \ni \mathit{tf}_{\mathrm{PS}} &::= (\mathit{syn\text{-}rules}\ \mathit{pat}\ \mathit{litl}\ \mathit{ses})(\!|r|\!) \\ &\to_{(\!|\,|\!)_{\mathrm{SR}}} \subseteq \mathit{ParseSubstSR} \times \mathit{Transformers} \end{aligned}$$

$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}\ (l_1 \ldots \mathtt{x}\ l_m \ldots)\ \mathit{ses}_{rhs})(\!|{}^{ks}x^n/\mathtt{x}|\!) \to_{(\!|\,|\!)_{\mathrm{SR}}} \quad \text{(ParseSubstSRSymLit)}$$
$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}(\!|{}^{ks}x^n/\mathtt{x}|\!)\ (l_1 \ldots l_m\ \ldots)\ \mathit{ses}_{rhs}(\!|{}^{ks}x^n/\mathtt{x}|\!))$$

$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}\ \underline{\mathit{litl}}\ \mathit{ses}_{rhs})(\!|{}^{ks}x^n/\mathtt{x}|\!) \to_{(\!|\,|\!)_{\mathrm{SR}}} \quad \text{(ParseSubstSRSymNoOccur)}$$
$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}\ \underline{\mathit{litl}}\ \mathit{ses}_{rhs}(\!|{}^{ks}x^n/\mathtt{x}|\!))$$
$$\text{iff } \neg \mathit{in\text{-}pat?}(\mathtt{x}, \underline{\mathit{litl}}) \wedge \neg \mathit{in\text{-}pat?}(\mathtt{x}, \underline{\mathit{pat}}) \wedge \mathtt{x} \neq \star$$

$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}\ \mathtt{a}\ \mathit{ses}_{rhs})(\!|{}^{ks}x^n/\mathtt{x}|\!) \to_{(\!|\,|\!)_{\mathrm{SR}}} \quad \text{(ParseSubstSRMVLitNoStar)}$$
$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}(\!|{}^{ks}x^n/\mathtt{x}|\!)\ \mathtt{a}\ \mathit{ses}_{rhs}(\!|{}^{ks}x^n/\mathtt{x}|\!)) \text{ iff } \mathtt{x} \neq \star$$

$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}\ \underline{\mathit{litl}}\ \mathit{ses}_{rhs})(\!|{}^{ks}x^n/\mathtt{x}|\!) \to_{(\!|\,|\!)_{\mathrm{SR}}} \quad \text{(ParseSubstSRIdMVB)}$$
$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}(\!|\mathtt{x}/\mathtt{x}|\!)\ \underline{\mathit{litl}}\ \mathit{ses}_{rhs}(\!|\mathtt{x}/\mathtt{x}|\!))$$
$$\text{iff } \neg \mathit{in\text{-}pat?}(\mathtt{x}, \underline{\mathit{litl}}) \wedge \mathit{in\text{-}pat?}(\mathtt{x}, \underline{\mathit{pat}})$$

$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}\ \underline{\mathit{litl}}\ \mathit{ses}_{rhs})(\!|\mathtt{a}/\mathtt{x}|\!) \to_{(\!|\,|\!)_{\mathrm{SR}}} \quad \text{(ParseSubstSRMV)}$$
$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}(\!|\mathtt{a}/\mathtt{x}|\!)\ \underline{\mathit{litl}}(\!|\mathtt{a}/\mathtt{x}|\!)\ \mathit{ses}_{rhs}(\!|\mathtt{a}/\mathtt{x}|\!))$$

$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}\ (l\ \ldots)\ \mathit{ses}_{rhs})(\!|{}^{ks}x^n/\star|\!) \to_{(\!|\,|\!)_{\mathrm{SR}}} \quad \text{(ParseSubstSRStarLitList)}$$
$$(\mathit{syn\text{-}rules}\ \mathit{pat}\ \mathit{remove\text{-}symbols}((l \ldots))\ \mathit{se}_{\mathrm{rhs}}(\!|{}^{ks}x^n/\star|\!))$$
$$\text{where } \langle \mathit{pat}, \mathit{se}_{\mathrm{rhs}} \rangle = \mathit{GenPatStar}(\underline{\mathit{pat}}, (l \ldots), \mathit{ses}_{rhs}, n)$$

$$(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}\ \mathtt{a}\ \mathit{ses}_{rhs})(\!|{}^{ks}x^n/\star|\!) \to_{(\!|\,|\!)_{\mathrm{SR}}} \quad \text{(ParseSubstSRStarMVLit)}$$
$$(\mathit{syn\text{-}rules}\ \mathit{pat}\ \mathtt{a}\ \mathit{se}_{\mathrm{rhs}}(\!|{}^{ks}x^n/\star|\!))$$
$$\text{where } \langle \mathit{pat}, \mathit{se}_{\mathrm{rhs}} \rangle = \mathit{GenPatStar}(\underline{\mathit{pat}}, \mathtt{a}, \mathit{ses}_{rhs}, n)$$

$\mathit{GenPatStar} : \mathit{NormalizedPatterns} \times \mathit{NormalizedLiteralList} \times \mathit{MixtureTerms} \times \mathbb{N} \to \mathit{Patterns} \times \mathit{MixtureTerms}$

$$\begin{aligned} \mathit{GenPatStar}(\mathit{pat}, \mathit{litl}, \mathit{ses}, n) &= \langle \mathit{pat}, \mathit{ses} \rangle \text{ if } \mathit{pat} \in \{a, \mathtt{a}, \underline{\mathtt{a}}, {}^{ks}x^n\} \\ \mathit{GenPatStar}(\mathtt{x}, (l_1 \ldots \mathtt{x} \ldots l_m), \mathit{ses}, n) &= \langle {}^{\emptyset}x^n, \mathit{ses}(\!|{}^{\emptyset}x^n/\mathtt{x}|\!) \rangle \\ \mathit{GenPatStar}(\mathtt{x}, (l_1 \ldots l_m), \mathit{ses}, n) &= \langle \underline{\mathtt{x}}, \mathit{ses}(\!|\mathtt{x}/\mathtt{x}|\!) \text{ iff } \mathtt{x} \notin \{l_1, \ldots, l_m\} \rangle \\ \mathit{GenPatStar}(\mathtt{x}, \mathtt{a}, \mathit{ses}, n) &= \langle \underline{\mathtt{x}}, \mathit{ses}(\!|\mathtt{x}/\mathtt{x}|\!) \rangle \\ \mathit{GenPatStar}((\mathit{pat}_l\ .\ \mathit{pat}_r), \mathit{litl}, \mathit{ses}, n) &= \langle (\mathit{pat}'_l\ .\ \mathit{pat}'_r), \mathit{ses}_l \rangle \text{ where} \\ & \quad \langle \mathit{pat}'_l, \mathit{ses}_l \rangle = \mathit{GenPatStar}(\mathit{pat}_l, \mathit{litl}, \mathit{ses}_r, n) \\ & \quad \langle \mathit{pat}'_r, \mathit{ses}_r \rangle = \mathit{GenPatStar}(\mathit{pat}_r, \mathit{litl}, \mathit{ses}, n) \end{aligned}$$

$$\begin{aligned} \mathit{in\text{-}pat?} &: \mathit{Symbol} \times \mathit{NormalizedPatterns} \to \mathit{Boolean} \\ \mathit{in\text{-}pat?}(\mathtt{x}, \mathit{pat}) &= \mathit{false} \text{ if } \mathit{pat} \in \{a, \mathtt{a}, \underline{\mathtt{a}}, {}^{ks}x^n\} \\ \mathit{in\text{-}pat?}(\mathtt{x}, \mathtt{x}) &= \mathit{true} \\ \mathit{in\text{-}pat?}(\mathtt{x}, \mathtt{y}) &= \mathit{false} \text{ if } \mathtt{x} \neq \mathtt{y} \\ \mathit{in\text{-}pat?}(\mathtt{x}, (\mathit{pat}_l\,.\,\mathit{pat}_r) &= \mathit{in\text{-}pat?}(\mathtt{x}, \mathit{pat}_l) \vee \mathit{in\text{-}pat?}(\mathtt{x}, \mathit{pat}_r) \end{aligned}$$

Figure 3.34: Elimination of $(\!|\,|\!)$ for `syntax-rules`

`rules` form is a binding construct for meta-variables and therefore the rules for hygienic macro expansion have to be obeyed. The same ideas for maintaining hygiene as for λ-abstractions apply: Whenever the meta-substitution replaces a meta-variable by an identifier in a binding position, the whole meta-substitution has to be updated to reflect the new binding. As Figure 3.35 shows, rule (MetaSubstSR) defers this task to the function *SMP*. This function in turn uses a helper function SMP' to walk along the pattern and search for meta-variables that are affected by the meta-substitution s. If it finds such a meta-variable, it first resolves the meta-substitution and normalizes the pattern using the reduction $\mapsto_{\text{patnorm}}$. Then SMP' uses the function *VarToMVB* to turn identifiers within the result into binding meta-variables and the function *VarToMV* to turn identifiers within the meta-substitution into meta-variables. However, these two functions *VarToMV* and *VarToMVB* must only turn those identifiers into meta-variables that are not literals. As they operate on the result of a meta-substitution, hygiene requires that only the literals created by the same meta-substitution count. As an example for a meta-variable appearing within a pattern, consider this R^{5}RS code:

```
(define-syntax m1
  (syntax-rules ()
     ((m1 p)
      (let-syntax ((m2 (syntax-rules (x)
                         ((m2 p) 13))))
         (m2 42)))))
(m1 x)
```

The macro `m1` is called with argument `x`, which is bound to the pattern variable `p`. The definition of macro `m2` lists `x` as a literal. Now the question is: Is the `x` in the pattern `(m2 p)` a pattern variable or a literal identifier? If it were a literal identifier, the `x` in the literal list would have "captured" the identifier `x` inserted by the macro application `(m1 x)`—a violation of hygiene. Hence, the argument of `m2` is a pattern variable and the above code expands to `13`.

On the other hand, in the following code capturing is intended:

```
(define-syntax m1
  (syntax-rules ()
     ((m1 p q)
      (let-syntax ((m2 (syntax-rules q
                         ((m2 p) 13))))
         (m2 42)))))
(m1 x (x))
```

Here the macro application `(m2 42)` no longer matches the pattern `(m2 p)` because `p` is replaced by `x` and the same macro application also replaces `q` by `(x)` and therefore the argument to `m2` has to be the unbound identifier `x`.

Figure 3.35 defines the list of literals that are changed by the meta-substitution s as $litl_c$. The functions *VarToMVB* and *VarToMV* consult this list to decide whether an identifier is a meta-variable or a literal identifier. The literals that are not affected by s is defined as $litl_u$. *SMP* returns $litl_u$ as the new list of literals.

Having extended the parsing substitutions and the meta-substitutions to *letrec-syn* forms, Figure 3.36 completes the elimination of expansion substitutions for these forms by presenting the elimination of the shift and mark operators and the identifier substitution. The (ShiftSR) rule propagates the shift operator to the pattern, literals and template, the rules (IdSubstSR) and (MarkSR) do the same for the identifier substitution and the mark operator.

The elimination rules from Figures 3.34, 3.35, and 3.36 have moved the various expansion substitutions to the arguments of the `syntax-rules` transformer and thereby parsed this form. Propagation of the expansion substitutions to the transformer means that we have recorded the lexical information of the macro definition to the transformer. The rules in Figure 3.35 did not propagate meta-substitutions to the pattern and the list of literals but the rules for parsing

$$
\begin{aligned}
\mathit{MetaSubstSR} \ni \mathit{tf}_{\mathrm{MS}} &::= (\mathit{syn\text{-}rules}\ \mathit{pat}\ \mathit{litl}\ \mathit{ses})\wr s\int \\
&\to_{\wr\int_{\mathrm{SR}}} \subseteq \mathit{MetaSubstSR} \times \mathit{Transformers} \\
(\mathit{syn\text{-}rules}\ \underline{\mathit{pat}}\ \underline{\mathit{litl}}\ \mathit{ses}_{rhs})\wr s_!\int &\to_{\wr\int_{\mathrm{SR}}} (\mathit{syn\text{-}rules}\ \mathit{pat}'\ \mathit{litl}'\ \mathit{ses}_{rhs}\wr s'\int) \qquad \text{(MetaSubstSR)} \\
&\quad \text{where } \langle \mathit{pat}', \mathit{litl}', s'\rangle = \mathit{SMP}(\underline{\mathit{pat}}, \underline{\mathit{litl}}, s_!)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{SMP} :& \mathit{NormalizedPatterns} \times \mathit{LiteralList} \times \mathit{MetaSubsts} \\
&\to \mathit{Patterns} \times \mathit{LiteralList} \times \mathit{MetaSubsts} \\
\mathit{SMP}(\mathsf{a}, \mathit{litl}, s) &= \langle \mathit{VarToMVB}'(\mathit{pat}_r), \mathit{litl}_u, \mathit{VarToMV}'(\mathit{pat}_r, s)\rangle \\
&\quad \text{iff } \mathsf{a}\wr s\int \mapsto^*_{\text{patnorm}} (\underline{\mathit{pat}}_l \,.\, \underline{\mathit{pat}}_r) \\
\mathit{SMP}(\mathit{pat}, \mathit{litl}, s) &= \langle \mathit{pat}', \mathit{litl}_u, s'\rangle \\
&\quad \text{where} \\
\langle \mathit{pat}', s'\rangle &= \mathit{SMP}'(\mathit{pat}, s) \\
\mathit{litl}_u &= \begin{cases} \{{}^{ks}x^n \in \mathit{litl}\} \cup \{\mathsf{a} \in \mathit{litl} | \mathsf{a}\wr s\int \to^*_{\wr\int} \mathsf{a})\} \text{ iff } \mathit{litl} = (l, \dots) \\ \mathsf{a} \text{ iff } \mathit{litl} = \mathsf{a}, \mathsf{a}\wr s\int \to_{\wr\int} \mathsf{a} \\ \emptyset \text{ iff } \mathit{litl} = \mathsf{a}, \mathsf{a}\wr s\int \mapsto^*_{\text{patnorm}} ({}^{ks}x^n, \dots) \end{cases} \\
\mathit{litl}_c &= \begin{cases} \{{}^{ks}x^n | \mathsf{a}\wr s\int \mapsto^*_{\text{patnorm}} {}^{ks}x^n, \mathsf{a} \in \mathit{litl})\} \text{ iff } \mathit{litl} = (l, \dots) \\ \emptyset \text{ iff } \mathit{litl} = \mathsf{a}, \mathsf{a}\wr s\int \to_{\wr\int} \mathsf{a} \\ ({}^{ks}x^n, \dots) \text{ iff } \mathit{litl} = \mathsf{a}, \mathsf{a}\wr s\int \mapsto^*_{\text{patnorm}} ({}^{ks}x^n, \dots) \end{cases} \\
\mathit{SMP}' :& \mathit{NormalizedPatterns} \times \mathit{MetaSubsts} \to \mathit{Patterns} \times \mathit{MetaSubsts} \\
\mathit{SMP}'(a, s') &= \langle a, s'\rangle \\
\mathit{SMP}'({}^{ks}x^n, s') &= \langle {}^{ks}x^n, s'\rangle \\
\mathit{SMP}'(\underline{\mathsf{a}}, s') &= \langle \underline{\mathsf{a}}, s'\rangle \\
\mathit{SMP}'(\mathsf{a}, s') &= \langle \mathsf{a}, s'\rangle \text{ iff } \mathsf{a}\wr s\int \to_{\wr\int} \mathsf{a} \\
\mathit{SMP}'(\mathsf{a}, s_r) &= \langle \mathit{VarToMVB}(\underline{\mathit{pat}}), \mathit{VarToMV}(\underline{\mathit{pat}}, s)\rangle \\
&\quad \text{iff } \mathsf{a}\wr s\int \mapsto^*_{\text{patnorm}} \underline{\mathit{pat}}, \underline{\mathit{pat}} \neq \mathsf{a} \\
\mathit{SMP}'((\mathit{pat}_l \,.\, \mathit{pat}_r), s') &= \langle (\mathit{pat}'_l \,.\, \mathit{pat}'_r), s_r\rangle \\
&\quad \text{iff } \langle \mathit{pat}'_l, s_l\rangle = \mathit{SMP}'(\mathit{pat}_l, s), \langle \mathit{pat}'_r, s_r\rangle = \mathit{SMP}'(\mathit{pat}_r, s_l) \\
\mathit{VarToMV} :& \mathit{NormalizedPatterns} \times \mathit{MetaSubsts} \to \mathit{MetaSubsts} \\
\mathit{VarToMV}(\mathit{pat}, s') &= s' \text{ if } \mathit{pat} \in \{a, \mathsf{a}, \underline{\mathsf{a}}\} \\
\mathit{VarToMV}({}^{ks}x^n, s') &= s' \text{ iff } {}^{ks}x^n \in \mathit{litl}_c \\
\mathit{VarToMV}({}^{ks}x^n, s') &= s'[\mathsf{x}/{}^{ks}x^n] \text{ iff } {}^{ks}x^n \notin \mathit{litl}_c \\
\mathit{VarToMV}((\mathit{pat}_l \,.\, \mathit{pat}_r), s') &= \mathit{VarToMV}(\mathit{pat}_r, s_l) \text{ iff } s_l = \mathit{VarToMV}(\mathit{pat}_l, s') \\
\mathit{VarToMV} :& \mathit{NormalizedPatterns} \times \mathit{MetaSubsts} \to \mathit{NormalizedPatterns} \\
\mathit{VarToMVB}(\mathit{pat}) &= \mathit{pat} \text{ if } \mathit{pat} \in \{a, \mathsf{a}, \underline{\mathsf{a}}\} \\
\mathit{VarToMVB}({}^{ks}x^n) &= {}^{ks}x^n \text{ iff } {}^{ks}x^n \in \mathit{litl}_c \\
\mathit{VarToMVB}({}^{ks}x^n) &= \underline{\mathsf{x}} \text{ iff } {}^{ks}x^n \notin \mathit{litl}_c \\
\mathit{VarToMVB}((\mathit{pat}_l \,.\, \mathit{pat}_r)) &= (\mathit{VarToMVB}(\mathit{pat}_l) \,.\, \mathit{VarToMVB}(\mathit{pat}_r))
\end{aligned}
$$

Figure 3.35: Reduction $\wr\int$ for `syntax-rules`

$$((\textit{syn-rules}\ \underline{\textit{pat}}\ \underline{\textit{litl}}\ \textit{ses}_{rhs})\uparrow^n) \rightarrow_{\uparrow_{\mathrm{SR}}} (\textit{syn-rules}\ (\underline{\textit{pat}}\uparrow^n)\ (\underline{\textit{litl}}\uparrow^n)\ (\textit{ses}_{rhs}\uparrow^n)) \quad \text{(ShiftSR)}$$

$$(\textit{syn-rules}\ \underline{\textit{pat}}\ \underline{\textit{litl}}\ \textit{ses}_{rhs})\male^n \rightarrow_{\male_{\mathrm{SR}}} (\textit{syn-rules}\ \underline{\textit{pat}}\male^n\ \underline{\textit{litl}}\male^n\ \textit{ses}_{rhs}\male^n) \quad \text{(MarkSR)}$$

$$(\textit{syn-rules}\ \underline{\textit{pat}}\ \underline{\textit{litl}}\ \textit{ses}_{rhs})[u] \rightarrow_{[]_{\mathrm{SR}}} (\textit{syn-rules}\ \underline{\textit{pat}}[u]\ \underline{\textit{litl}}[u]\ \textit{ses}_{rhs}[u]) \quad \text{(IdSubstSR)}$$

Figure 3.36: Elimination of the shift operator, the mark operator, and identifier substitutions for `syntax-rules` forms

substitutions in Figure 3.34 and the rules for the shift and mark operators and the identifier substitutions did attach the substitutions to the patterns and the list of literals. For the rule for macro application of `syntax-rules` (ExpandMappSR) in Figure 3.32 to work, we need to normalize this pattern. The follow rule (SRNorm) from Figure performs this normalization. It again relies on the reduction $\mapsto_{\mathrm{patnorm}}$that eliminates all substitutions within patterns and literals.

$$(\textit{syn-rules}\ \textit{pat}\ \textit{litl}\ \textit{ses}_{rhs}) \rightarrow_{\mathrm{SRNorm}} (\textit{syn-rules}\ \textit{pat}'\ \textit{litl}'\ \textit{ses}_{rhs}) \quad \text{(SRNorm)}$$

iff $\textit{pat} \notin \textit{NormalizedPatterns} \vee \textit{litl} \notin \textit{NormalizedPatterns}$ where $\textit{pat}' \mapsto_{\mathrm{patnorm}} \textit{pat}$ and $\textit{litl}' \mapsto_{\mathrm{patnorm}} \textit{litl}$

Figure 3.37: Normalization of patterns within `syntax-rules`

This finishes the elimination for `syntax-rules` forms which we can now define as:

$$\rightarrow_{\text{El-SR}} = \rightarrow_{\text{⇫}_{\mathrm{SR}}} \cup \rightarrow_{\uparrow_{\mathrm{SR}}} \cup \rightarrow_{\male_{\mathrm{SR}}} \cup \rightarrow_{\wr\int_{\mathrm{SR}}} \cup \rightarrow_{[]_{\mathrm{SR}}} \cup \rightarrow_{\mathrm{SRNorm}}$$

This reduction is the extension of $\rightarrow_{\text{El-Tf}}$ for `syntax-rules` transformers.

3.9.3 Elimination Rules for Patterns

This section defines the reductions $\mapsto_{\mathrm{patout}}$and $\mapsto_{\mathrm{patnorm}}$used several times in the previous sections. These reductions eliminate expansion substitutions within patterns. As we regard the list of literals of a `syntax-rules` form as a pattern as well, these functions also work on literals. Furthermore, lexical s-expressions as found within syntactic objects are also very similar to patterns—they only lack binding meta-variables. We therefore apply $\mapsto_{\mathrm{patout}}$and $\mapsto_{\mathrm{patnorm}}$to lexical s-expressions as well.

Two strategies exist for eliminating substitution operators within patterns:

- Complete elimination of the operators. The normal forms of this strategy are terms of *NormalizedPatterns*.
- Elimination of the outermost operators. The normal forms of this strategy are terms of *OutsideNormalizedPatterns*. Patterns of this form are either basic pattern terms or pairs of patterns with expansions substitutions (*Patterns*).

The elimination for patterns is necessary in the following situations:

- *match* and *gen-subst* use $\mapsto_{\mathrm{patout}}$ on the input to remove the keyword from the call.
- *match*′ uses $\mapsto_{\mathrm{patnorm}}$ on the input if $\textit{pat}_{lhs}$ is an identifier or a constant.
- *match*′ and *gen-subst*′ use $\mapsto_{\mathrm{patout}}$ on the input if $\textit{pat}_{lhs}$ is a pair.
- *gen-subst* uses $\mapsto_{\mathrm{patout}}$ on the input if $\textit{pat}_{lhs}$ is a pair.

- The function *SMP* uses $\mapsto_{\text{patnorm}}$ to eliminate the meta-substitution applied to literals and patterns.
- (SRNorm) uses $\mapsto_{\text{patnorm}}$ to normalize pattern and literals in definitions.
- The definition of `syntax-car` and `syntax-cdr` use $\mapsto_{\text{patout}}$ to reveal the pair structure of a syntactic object.

Patterns contain two sorts of terms not present in the mixture syntax from Figure 3.31: binding meta-variables and pairs of patterns. In addition, the rule (ParseSubstSRIdMVB) generates parsing substitutions that replace symbols by binding meta-variables. We will now define elimination rules for these terms and combine them with rules for elimination rules for the mixture syntax to define the elimination of expansion substitutions for patterns.

Figure 3.38 contains, for all expansion substitutions the elimination rules for binding meta-variables and lists.[15] For the elimination of the new parsing substitutions of the form $pat(\!|\underline{\mathtt{a}}/\mathtt{x}|\!)$, rule (ParseSubstSymMVb) replaces a symbol by a binding meta-variable and rule (ParseSubstSymOtherMVb) drops the substitution for a non-matching symbol. The remaining rules (ParseSubstMVb) and (ParseSubstList) just drop their substitution for binding meta-variables and propagate them to the elements of a list (or pair) pattern.

$$\begin{aligned}
\mathtt{x}(\!|\underline{\mathtt{a}}/\mathtt{x}|\!) &\to_{(\!|)_{\text{Pat}}} \underline{\mathtt{a}} && \text{(ParseSubstSymMVb)}\\
\mathtt{x}(\!|\underline{\mathtt{a}}/\mathtt{y}|\!) &\to_{(\!|)_{\text{Pat}}} \mathtt{x} \text{ iff } \mathtt{x} \neq \mathtt{y} && \text{(ParseSubstSymOtherMVb)}\\
\underline{\mathtt{a}}(\!|r|\!) &\to_{(\!|)_{\text{Pat}}} \underline{\mathtt{a}} && \text{(ParseSubstMVb)}\\
(\underline{pat}\ldots)(\!|r|\!) &\to_{(\!|)_{\text{Pat}}} (\underline{pat}(\!|r|\!)\ldots) && \text{(ParseSubstList)}\\
\underline{\mathtt{a}}[u] &\to_{[]_{\text{Pat}}} \underline{\mathtt{a}} && \text{(IdSubstMVB)}\\
(\underline{pat}\ldots)[u] &\to_{[]_{\text{Pat}}} (\underline{pat}[u]\ldots) && \text{(IdSubstList)}\\
\underline{\mathtt{a}}\wr s_! \int &\to_{\wr\int_{\text{Pat}}} \underline{\mathtt{a}} && \text{(MetaSubstMVB)}\\
(\underline{pat}\ldots)\wr s_! \int &\to_{\wr\int_{\text{Pat}}} (\underline{pat}\wr s_! \int\ldots) && \text{(MetaSubstPat)}\\
(\underline{\mathtt{a}}\uparrow^n) &\to_{\uparrow_{\text{PatO}}} \underline{\mathtt{a}} && \text{(ShiftMVB)}\\
((p\ldots)\uparrow^n) &\to_{\uparrow_{\text{PatO}}} ((p\uparrow^n)\ldots) && \text{(ShiftList)}\\
\underline{\mathtt{a}}\male^n &\to_{\male_{\text{PatO}}} \underline{\mathtt{a}} && \text{(MarkMVB)}\\
(p\ldots)\male^n &\to_{\male_{\text{PatO}}} (p\male^n\ldots) && \text{(MarkList)}
\end{aligned}$$

Figure 3.38: Reduction rules for patterns

The elimination of patterns is the union of the elimination reductions presented above plus all the rules for the syntactic base forms (constants, symbols, identifiers, and meta-variables) as presented in previous sections. We hence omit the rules that deal with other abstract data terms like λ-abstractions or procedure applications and include only the rules for the base forms:

$$\begin{aligned}
\to_{\text{El-Pat}} = {} & \to_{(\!|)_{\text{Pat}}} \cup \to_{\uparrow_{\text{Pat}}} \cup \to_{\male_{\text{Pat}}} \cup \to_{\wr\int_{\text{Pat}}} \cup \to_{[\,]_{\text{Pat}}} \cup\\
& (ParseSubstStar) \cup (ParseSubstSym) \cup (ParseSubstConst) \cup (ParseSubstId) \cup\\
& (ParseSubstSymOther) \cup (ShiftConst) \cup (ShiftId) \cup (ShiftMV) \cup\\
& (MarkConst) \cup (MarkId) \cup (MarkMV) \cup\\
& (NormalizedMetaSubsts) \cup (MetaSubstConst) \cup (MetaSubstId) \cup\\
& (MetaSubstMV) \cup (MetaSubstMVEmpty) \cup (MetaSubstMVOther) \cup\\
& (IdSubstId) \cup (IdSubstOther) \cup (IdSubstMV) \cup (IdSubstConst)
\end{aligned}$$

[15] The rules operate on lists instead of pairs for technical reasons beyond the scope of the current discussion. The rules for pairs can be defined analogously.

The inclusion of the rules to meta-substitutions might come as a surprise as the rule (MetaSubstSR), which handles meta-substitutions for `syntax-rules` forms, does not pass these substitutions to the patterns. However, the very same rule uses the $\mapsto_{\text{patnorm}}$to eliminate meta-substitutions while the rule builds the pattern if a meta-substitution replaces a meta-variable within a pattern or list of literals.

The two elimination functions for patterns build on the $\rightarrow_{\text{El-Pat}}$ reduction to eliminate expansion substitutions within patterns. Two contexts, *PAT* and *PATO*, stipulate where the reduction may take place:

$$\begin{array}{lll} PATO & ::= & [\,] \mid PATO(\!|r|\!) \mid PATO[u] \mid PATO\wr s\int \mid PATO\uparrow^n \mid PATO\male^k \\ PAT & ::= & [\,] \mid (\underline{pat} \ldots PAT\, pat \ldots) \mid PAT(\!|r|\!) \mid PAT[u] \mid PAT\wr s\int \mid PAT\uparrow^n \mid PAT\male^n \end{array}$$

The context *PATO* places the reduction within the substitution operators only. *PATO* places no reduction within composed patterns. In contrast, the first rule for the context *PAT* places the hole within the left-most non-normalized pattern of a pattern list.

The reductions $\mapsto_{\text{patnorm}}$ and $\mapsto_{\text{patout}}$ are the extensions of the $\rightarrow_{\text{El-Pat}}$ reduction to the two contexts:

$$pat \mapsto_{\text{patnorm}} pat' \text{ iff } pat = PAT[pat_1], pat' = PAT[pat_2], pat_1 \rightarrow_{\text{El-Pat}} pat_2$$

$$pat \mapsto_{\text{patout}} pat' \text{ iff } pat = PATO[pat_1], pat' = PATO[pat_2], pat_1 \rightarrow_{\text{El-Pat}} pat_2$$

We let $\mapsto^*_{\text{patnorm}}$ and $\mapsto^*_{\text{patout}}$ denote the reflexive, transitive closures.

3.9.4 Parsing `syntax-rules`

So far, the specification for `syntax-rules` transformers includes the new expansion rules (ExpandMappSR) and (ExpandMappSRFail) and the reduction $\rightarrow_{\text{El-SR}}$ as an extension of $\rightarrow_{\text{El-Tf}}$. As required by the enumeration at the end of Section 3.5, a transformer specification also needs to provide parsing rules that extend the reduction $\rightarrow_{\text{PT}}$, on which the rule (ExpandParseTransformer) from the core macro expander in Figure 3.9 builds. Figure 3.39 contains these rules for `syntax-rules` forms. First, the rule (SyntaxRulesNorm) normalizes the substitutions applied to the `syntax-rules` form using the $\rightarrow_{\text{El}}$ reduction. This allows the other two rules to use parsing contexts that are defined based on the normalized versions of the substitution operators.[16] The rules (SyntaxRulesListPat) and (SyntaxRulesSymPat) generate the abstract syntax for `syntax-rules` clauses. Rule (SyntaxRulesListPat) strips the keyword from the pattern whereas in the rule (SyntaxRulesSymPat) the pattern is a single identifier that must correspond to a meta-variable. Later, the (MetaSubstSR) will replace the meta-variable by a pattern and strip the keyword.

$$\rightarrow_{\text{PT}} \subseteq \textit{Lex-S-Expressions} \rightarrow \textit{Transformers}$$

$$ses \rightarrow_{\text{PT}} ses' \text{ iff } ses' \neq ses \text{ where } ses \mapsto^*_{\text{El}} ses' \qquad \text{(SyntaxRulesNorm)}$$

$$P[(\texttt{syntax-rules}\ (pat_1\ pat_2\ \ldots)\ litl\ se_{\text{rhs}})] \rightarrow_{\text{PT}} P[(\textit{syn-rules}\ (pat_2\ \ldots)\ litl\ se_{\text{rhs}})] \qquad \text{(SyntaxRulesListPat)}$$

$$P[(\texttt{syntax-rules}\ x_{lhs}\ litl\ se_{\text{rhs}})] \rightarrow_{\text{PT}} P[(\textit{syn-rules}\ x_{lhs}\ litl\ se_{\text{rhs}})] \qquad \text{(SyntaxRulesSymPat)}$$

Figure 3.39: Parsing reduction for `syntax-rules`

This completes the semantics of `syntax-rules` transformers. Note that this semantics is independent from the semantics of the computational macros from Section 3.6. The elimination rules

[16] Actually, only meta-substitutions come in normalized and non-normalized variants.

for meta-substitutions and identifier substitutions are the only rules needed from the description of the computational macros, which are also required by the `syntax-rules` forms. On the other hand, as both transformers produce the same kind of expansion substitutions, the transformers are compatible.

3.10 Future Work Towards Full Scheme

The semantics in the previous sections already cover a wide range of the features present in current macro-expander facilities. Unlike existing descriptions, our presentation leaves no room for ambiguity but instead pins down the semantics precisely.

However, a few features are still missing in our description. This section sketches how extensions implementing features would work.

λ and `letrec-syntax` with Multiple Parameters So far, λ-abstractions and `letrec-syntax` forms accept only one parameter. Unfortunately, extending these forms to multiple parameters is not compatible with our current representation of identifiers. While the level of an identifier uniquely determines the binding form, it cannot identify a single parameter out of a list of parameters. At first glance, it looks as if the name of the variable could provide this information as the names of the bound variables of a binding form must be disjoint. However, this is not compatible with hygienic macro expansion. Consider this example, adapted from the paper by Dybvig et al. [DHB92]:

```
(let-syntax ((dolet (syntax-rules ()
                      ((dolet b)
                       ((lambda (a b) (+ a b))
                        3 4)))))
   (dolet a))
```

The macro application `(dolet a)` substitutes `a` for the pattern variable `b` in the parameter of the `lamdba`-form. However, hygiene requires the inserted `a` to be different from the `a` already present in the template. Consequently the name of a variable by itself is not sufficient to determine which parameter binds the variable. One solution is to use the marks of an identifier in combination with the name. In the example above, the two occurrences of `a` have different marks, and this is no coincidence: if they had the same marks, this would correspond to a duplicate variable within a parameter list, which is an error. The drawback of using the marks and the name is that the evaluation of expressions would need to take name and marks into account as well. As before, the level selects the binder but then the name identifies the possible parameter positions. If there are several possible parameter positions, the marks determine the correct position. Another solution is to extend the identifier representation by another index that indicates the position within the parameter list. The parser needs to generate this index once; after that the position only matters for the equality checks on identifiers. The position index approach seems preferable as its semantics is straightforward: the position is another index that describes the binding place of an identifier.

`let-syntax` and `define-syntax` In the definitions so far `letrec-syntax` is the only available binding construct for macros. The Scheme standard defines two other forms: `let-syntax` and `define-syntax`. `Let-syntax` is the non-recursive variant of `letrec-syntax`; `define-syntax` is a top-level definition. Adding rules for `let-syntax` to the semantics is easy: basically the rules for `letrec-syntax` can be adapted to reflect the fact that the keyword is not bound within the transformer. For a semantics of `define-syntax`, top-level definitions need to be included in the language. However, the semantics of top-level definitions in Scheme is not well-defined. The problem is that such top-level definitions include `define-syntax` forms as well as global definition via `define` and ordinary expressions that include macro applications. Such a top-level macro application can expand into a global definition. This definition then extends the set of defined identifiers. However, the Scheme standard prescribes no order for processing the top-level

definitions during macro expansion. This is problematic, as macro expansion is sensitive to the set of defined identifiers, the result depends on the order in which a Scheme implementation chooses to expand the top-level definitions. Consider the following example provided by Richard Kelsey:[17]

```
(define-syntax foo
  (syntax-rules (bar)
    ((foo bar baz) (baz))
    ((foo x baz) (define x baz))))))

(foo bar (lambda () (display 'first)))

(define bar 'second)))
```

Here, if the macro expander completely expands the macro application `(foo bar ...)`, the program prints `first` during evaluation. If, however, the macro expander expands the macro application first to obtain the set of global definitions, then processes the definition `(define bar 'second)`, and then processes the macro application a second time, this time to perform the expansion proper, it will choose the first rule of the macro application and the program prints `second` during evaluation. Consequently, to define a semantics for `define-syntax`, the way the macro expander processes top-level definitions needs to be defined more precisely than currently done in R^{5}RS.

Multiple Rules for `syntax-rules` Our variant of `syntax-rules` allows only a single rule while the original version from R^{5}RS admits an arbitrary number of rules. To extend our semantics to multiple rules, the representation of a `syntax-rules` transformer would contain a list of pattern/template pairs in addition to the list of literals. The rule (ExpandMappSR) would then try to match the patterns from top to bottom against the input from. If the first pattern matches, it would call *gen-subst* with the corresponding template. If no pattern matches, it would fail like our (ExpandMappSRFail) rule.

Patterns with Ellipsis Patterns with ellipsis are an important concept of the `syntax-rules` facility as they enable the programmer to describe the required shape of the input form without commiting to a fixed size. Adding patterns with ellipsis to the semantics of `syntax-rules` from Section 3.9 would require an extended pattern language, extensions of the matching *match* function and the function for generating the meta-substitutions, *gen-subst*, and elimination rules of the expansion substitutions applied to patterns with ellipsis. For pattern with ellipsis, *match* would need to repeatedly match the pattern before the ellipsis against elements of the corresponding input form. The function *gen-subst* would need to bind the meta-variables from the pattern to lists containing the corresponding elements from the input form. New rules of the reduction for eliminating meta-substitutions $\rightarrow_{\wr\wr}$ would then ensure that these meta-variables are used only followed by an ellipsis in the template and then construct the output according to the template form before the ellipsis.

`quote` So far our language does not contain `quote` forms. Figure 3.40 contains the rules to extend the parser, the expander and Macro Scheme. The rules propagate the expansion substitutions to the argument of quote in case there are occurrences of meta-variables inside. At the end of expansion, the *strip* function removes any lexical information by turning identifiers back into symbols. It is defined as the standard reduction function of the $\rightarrow_{\text{strip}}$ reduction, which contains only a single rule (StripId):

$$^{ks}\mathtt{x}^{n} \rightarrow_{strip} \mathtt{x} \qquad \text{(StripId)}$$

and take place in strip contexts *STRIP*:

[17] Personal communication.

$STRIP \;::= [\,] \mid (se \ldots STRIP\ \underline{pat} \ldots)$

Besides the comparison of unbound identifiers in *free-id=?*, the rule (StripId) is the only rule that makes use of the name of an identifier: the name of the identifier denotes the name of the symbol.

Mixture syntax:	
e	$::= \ldots \mid (\text{'}ses)$

$$\underline{d}, k \vdash P[(\texttt{x}\ se)] \to_{\text{Parse}} \underline{d}, k \vdash P[(\text{'}se)] \text{ iff } P[\texttt{x}] \mapsto^*_{\text{El}} \texttt{quote}^n \quad \text{(ParseQuote)}$$

$$\underline{d}, k \vdash (\text{'}ses) \to_{\text{Expand}} (\text{'}strip(ses')) \text{ where } ses \mapsto^*_{\text{patnorm}} ses' \quad \text{(ExpandQuote)}$$

$$(\text{'}ses)(\!|r|\!) \to_{(\!|\,|\!)} (\text{'}ses(\!|r|\!)) \quad \text{(ParseSubstQuote)}$$

$$(\text{'}ses)[u] \to_{[]} (\text{'}ses[u]) \quad \text{(IdSubstQuote)}$$

$$(\text{'}ses)\wr s \int \to_{\wr\int} (\text{'}ses\wr s\int) \quad \text{(MetaSubstQuote)}$$

$$((\text{'}ses)\uparrow^n) \to_{\uparrow} (\text{'}(ses\uparrow^n)) \quad \text{(ShiftQuote)}$$

$$(\text{'}ses)\male^n \to_{\male} (\text{'}ses\male^n) \quad \text{(MarkQuote)}$$

$$((\text{'}ses)\downarrow^n) \to_{\downarrow} (\text{'}ses) \quad \text{(UnshiftQuote)}$$

$$ES[(\text{'}ses)] \to_{\text{ES}} (\text{'}ses) \quad \text{(EvalESQuote)}$$

Figure 3.40: Mixture syntax, parsing, expansion, elimination and stripping for quoted forms

Macro Scheme and State The definition of Macro Scheme does not include state. This ensures that the input of a macro application can only be used within the expansion of the macro application. If a macro could access code from other macro applications from a foreign scope and would include this code in its output, hygiene would break because the foreign identifiers would no longer refer to their original binding places. Consequently, for a version of Macro Scheme that includes state, accessing code from foreign scopes needs to be prohibited. I do not think this would limit the expressiveness of the macro system but I have not yet found a good solution to enforce this restriction.

Full Macro Expansion for Macro Scheme So far, the `syntax-lambda` construct does not support meta-variables as binding variables. This means that macros that expand into macro definitions using `es-transformer` must not use their meta-variables as binding variables of `syntax-lambda` forms. Two extensions are necessary to enable such macros:

- The mixture syntax representation of `syntax-lambda` needs to distinguish binding meta-variable and occurrences of meta-variables. This has already been demonstrated in the `syntax-rules` transformer from Section 3.9.
- As these macro-generating macros may insert references to meta-variables within the scope of a `syntax-lambda`, it is necessary to add levels to meta-variables as well. The techniques demonstrated for ordinary identifiers can be adapted *mutatis mutandis*.

3.11 Comparison with the Work of Bove and Arbilla

The idea of using explicit substitutions to explain hygienic macro expansion is due to Bove and Arbilla [BA92]. Programs in their calculus consist of a set of macro definitions and an expression, which contains macro applications. There is no construct for local macro definitions. A macro definition contains a list of variables and meta-variables as its right-hand side and an expression

with meta-variables as its template. The macro writer has to provide the template as abstract syntax. Likewise the body of a program has to be specified as abstract syntax. Bove and Arbilla use identifiers with levels to represent variables. However, in their calculus the level specifies the distance to the binding place of the variable as the number of λ-abstractions *that bind the same name*. Consequently, identifiers in the source program always have a zero index and can be omitted. This fact and the circumstance that the source language of macro expansion is already abstract syntax makes the use of a parser superfluous. The operators of the calculus are the same as ours but their shift operator also performs the task of our identifier substitution and therefore needs three indices instead of one.

Our work makes the following improvements over the work by Bove and Arbilla:

1. Inclusion of a computational macro transformer, namely `es-transformer`. The Bove and Arbilla only define a rewriting based transformer, which is much simpler as no meta-language (for us this is Macro Scheme) needs to be defined.
2. We have support for local macro definitions. By applying the shift operator to the definition set, we can move the set into local scopes and there extend it by new macro definitions.
3. Conformance to the second condition for hygienic macros [CR91]: References inserted by macros can only be captured by bindings inserted by the macro.
4. True support for literal identifiers. The original work claimed to support literal identifiers (called *keywords* there) but did not consider intermediate bindings for these keywords.
5. Redefinition of special forms (for example `lambda`) and the ability to specify concrete syntax instead of abstract syntax as input terms through an explicit parsing phase.
6. Our pattern language permits arbitrarily nested patterns instead of a flat list of arguments. Thereby, our pattern matcher lazily matches the macro arguments with the pattern to ensure that only the necessary part of the arguments is touched.
7. Levels of identifiers increase with every binder, independent of the name. This corresponds to real-life implementations of static scoping.
8. The use of contexts significantly simplifies the presentation.
9. Support for recursive macros. Recursive macro definitions are commonly used for `syntax-rules` transformers to process inductively defined syntactic extensions.
10. Exact evaluation strategy.

For the forth improvement, an example program that fails to preserve hygiene in the Bove and Arbilla calculus is:[18]

```
(define-syntax curryf
  (syntax-rules ()
    ((curryf a b) ((f a) b))))

(lambda (f) (curryf 1 2))
```

The original calculus expands this example to:

$$(\lambda f.((f^0 1)2))$$

That is, the identifier f, inserted by the macro, gets captured in the expanded program. The technical cause of the problem in the Bove/Arbilla calculus is that it does not modify the level of

[18] To improve readability, the example is written in concrete syntax , even though the Bove/Arbilla calculus can work on abstract syntax only.

identifiers present in the macro definitions as expansion moves into the scope of λ-abstractions. This means that the calculus does not record the lexical information at the macro definition. On the other hand, our semantics correctly expands the example to:

$$(\lambda f.((f^1 1)2))$$

That is, the inserted identifier is not captured but refers to the top-level binding as it did when the macro was defined. Our semantics achieves hygiene by propagating the lexical information into the macro definition. More specifically, during the expansion of the body of an abstraction, the expansion relation applies the shift operator to the macro definition as well. The shift operator applied to a macro definition will shift the right-hand side of the macro definition.

An example program for the third improvement is:[19]

```
(define-syntax mylet
  (syntax-rules (be in)
    ((mylet var be expr in body)
     ((lambda (var) body) expr))))

(lambda (be) (mylet x be 1 in x))
```

Here, the Bove/Arbilla calculus erroneously detects a correct application of the `mylet` macro, even though the identifier `be` does not have the same binding at macro definition as at macro application and must therefore not match the `be` keyword.

In our system the macro application correctly does not match the definition of the `mylet` macro. Again, our semantics improves over the Bove/Arbilla calculus by propagation of lexical information to the macro definition. This time, the shift operator preserves hygiene by shifting the left-hand side of the macro definition: The literals are identifiers within the pattern and the shift operator increments their levels as it processes the pattern.

The only downside of our semantics compared to the original work is that the equational theory in our semantics is weaker: Unlike the Bove/Arbilla calculus, expansion cannot happen under explicit substitutions. This is a consequence of adding a parsing phase, which of cause makes our work substantially more realistic than the calculus of Bove/Arbilla. In addition, our semantics prescribes an exact order for the reductions which is not desirable of a calculus. This can be remedied by altering the contexts so that they no longer uniquely divide terms into context and redex. Of course, then confluence of the rules needs to be proved, which is an important aspect of the work of Bove and Arbilla.

[19] The Bove/Arbilla calculus would specify that `be` and `in` are literals using the abstract syntax.

Chapter 4

Semantics for Modules

Modules appear in all phases of program evaluation: During parsing and macro expansion modules define name spaces. Modules represent the units of compilation. Linking combines modules to form a program. Evaluation manipulates higher-order modules as values. Each of these roles also influences aspects of the whole language. Name spaces are closely related to the representation of identifiers. While the algorithm for hygienic macro expansion from Chapter 3 build on identifiers with labels to track the binding places of variables and keywords through macro expansion, labels are not sufficient to represent binding places located in other modules. Instead, the representation of identifiers needs to be extended to denote the module exporting the identifier as well. Independent compilation and linking require the language to limit the compile-time extent of its constructs to the body of a module or to make it possible to derive the needed information from the interfaces. Finally, the evaluation rules need to handle first-class modules as values with their own scoping and reduction rules.

This chapter describes a language with a higher-order module system, independent compilation, and hygienic macros. The resulting new language is called $\Lambda_n^{\text{Module}}$. The extension builds on a new representation of identifiers that supports modules as binding places. A module body may contain identifiers that are not bound by the module or a λ-abstraction. Instead the module may have imported the identifier from some other module by importing an interface that lists the identifier as exported. Within the abstract syntax, there needs to be a way to describe the binding place for such imported identifiers. $\Lambda_n^{\text{Module}}$ therefore adds an *import path* to the representation of identifiers. An import path describes the series of interfaces that the current module used to import an identifier. Using these new identifiers, we can define parsing and macro expansion for module bodies and module definitions. The linking phase for $\Lambda_n^{\text{Module}}$ is then a set of reductions that use explicit substitutions to replace these identifiers by values defined in imported modules. In the present semantics, the linking phase assumes the imported module to be fully evaluated, which precludes mutually recursive modules. However, alternative formulations that remedy this restriction are conceivable. The evaluation of higher-order modules also uses explicit substitutions to propagate the values bound variables to the bodies of modules that appear within expressions.

4.1 Identifier Representation and Linking for Modules

A module is a new binding construct that contains a set of macro definitions and a set of variable definitions. As argued in Section 3.10, we omit top-level macro applications as their semantics is not well-defined in Scheme. Each definition within the module consists of a name and a right-hand side. For a macro definition, the right-hand side is a transformer, for a variable definition, the right-hand side is an expression. The names of all definitions are bound mutually recursively in all right-hand sides. The macro definitions from the module's export interface extend the set of definitions. In addition, a module imports evaluated definitions and macros from other modules. Importing another module binds the names of the imported definitions in the right-hand sides of

the definitions to the values and transformers from the imported definitions. The export interface of the imported module restricts the set of definitions that other modules can import.

With the addition of modules, variable occurrences in the right-hand side of a definition can now be bound either by a λ-abstraction, or by a definition, or by an imported module. The identifier representation with levels from Chapter 3 is not sufficient in this setting: While the level can select a module as a binding place, it is not enough to describe for an imported variable the module that provides the binding for the variable because the imported modules are all "on the same level".[1] Because of macros, the name of the variable also does not provide sufficient information: While the module system requires the names exported by the imported modules to be disjoint, an exported macro could insert a variable occurrence whose name is also imported from a different module. Hygiene requires that the two names do not interfere. For example, consider this program written in the language from Chapter 2:

```
(define-interface foo-interface
   (export x))

(define-interface bar-interface
   (export (define-syntax insert-x
             (syntax-rules ()
                ((insert-x) x)))))

(define-module foo foo-interface
   (open scheme-interface)
   (begin
      (define x 23)))

(define-module bar bar-interface
   (open scheme-interface)
   (begin
      (define x 42)))

(define-module baz (export)
   (open scheme-interface
         foo-interface
         bar-inteface)
   (begin
      (define y x)
      (define z (insert-x))))

(define-program p
   (modules foo bar baz))
```

The variable occurrence `x` in module `baz` is supposed to refer to the binding in module `foo`; its value will be 23. The right-hand side of the definition of `z` is also a variable occurrence with name `x`, but it has to refer to the binding in module `bar` as the macro `insert-x` from module `bar` inserted this occurrence. Its value will be 42. Therefore, to support modules with macros, we extend the identifier description to contain an additional *import path* that describes the module that contains the binding for the identifier. The import path comprises a sequence of interfaces that corresponds to the path the identifier took while being inserted. The import path of an identifier contains more than one interface if the identifier has been inserted by an imported macro and the interface providing the macro has in turn imported the identifier from another interface. This interface might in turn have imported the identifier through another macro and so on.

[1] Defining some order among the imported modules would remedy this problem but fails in the case of higher-order modules.

The final ingredient necessary to complete the representation of identifiers in the presence of modules is the treatment of unbound identifiers. Unbound identifiers play an important role as literal identifiers in pattern-based macros where macro writers use them to structure syntactic extensions. The `else` literal from the `cond` macro is one such example from R^{5}RS. The parser in Chapter 3 uses the $\star$-substitution to assign a level pointing to the global scope (the scope outside any local binder) to such unbound identifiers. The function *free-id=?*, used by the primitive `free-identifier=?` and the `syntax-rules` pattern matcher, is able to recognize unbound identifiers as equal by comparing their name and the level. As an example for unbound identifiers in the presence of modules, Section 2.2 defined the `if-t-e` macro with the literals `then` and `else` and introduced the `free` clause within an interface:

```
(define-interface if-t-e-interface
  (export
    (open scheme-interface)
    (free then else)
    (define-syntax if-t-e
      (syntax-rules (then else)
         ((if-t-e test then cons else alt)
          (if then else))))))
```

The `free` clause lists identifiers that the exported macros of the interface assume to be unbound. However, within the macros defined in an interface the level is no longer sufficient to select a global scope where unbound identifiers are "bound": The purpose of an interface is to provide the static information of all modules that implement this interface. However, as these modules may appear within the scope of other binders, the level—that is, the number of binders between the identifier occurrence and its binder—differs for modules that appear at different lexical depths. Hence we use the import path of an identifier instead of the level to denote that an imported interface assumes an identifier to be unbound. We initialize the import path of the identifiers from the interface's `free` clause with a special interface, the `free-interface`. In addition, the top-level $\star$-substitution initializes the import path of the identifiers it creates with `free-interface`. Hence the top-level $\star$-substitution of the parser is $se(\!|\emptyset \star^0_{I_{\text{free}}()}/\star|\!)$ and whenever this parsing substitution applies to a symbol, it generates an identifier with the `free-interface` as its import path. As the parser applies the $\star$-substitution as outermost substitution, it only applies to symbols without binders—the unbound identifiers. Two identifiers are equal according to *free-id=?* if they have the same level and the same import path, or if they have the same name and both contain `free-interface` in their import path. The first case covers bound identifiers and the second case corresponds to the comparison of unbound identifiers.

Sections 4.3 and 4.4 contain the parser and the macro expander that expand macros within modules and construct identifiers with levels and import paths on the way. During this process, interfaces play an important role as they contain the macro declarations of the imported macros. However, during linking and evaluation, the actual contents of the interfaces can be (mostly) neglected as will be explained below.

Having a representation for identifiers, the remainder of this section introduces some other concepts necessary for linking and evaluation: evaluation of programs, using explicit substitutions to formalize linking, semantics of definitions, and linking in the presence of macros.

For the evaluation, we start from an expanded program that already contains identifiers with levels and import paths. Evaluation of a program is the evaluation of all modules in the program in the evaluation order derived during configuration. Evaluation of a module is the evaluation of the right-hand sides of the definitions from the module's set of variable definitions. During the evaluation of the definitions the imported identifiers need to be bound—evaluation of the imported modules produces the corresponding values. Therefore, it is either necessary that evaluation processes all imported modules of a module before the module itself or that the imports are placeholders that can be assigned as soon as the corresponding definition has been evaluated. As the latter approach supports mutually recursive definitions, we use placeholders. The details of using placeholders will be explained below when we turn to the semantics of definitions.

Section 2.3 calls the process of binding the imported variables to values from the imported modules *dynamic linking*. Dynamic linking of a module happens before evaluation of the module but after evaluating the imported modules. Thus the dynamic linking of modules is interleaved with the evaluation of a program. To formalize the linking step, the $\Lambda_n^{\text{Module}}$ calculus uses explicit substitutions that replace variables by values. This kind of substitution is not new: the $\rightarrow_{\beta_v^n}$ reduction from Section 3.3 also relies on these substitutions to describe function application. The $\rightarrow_{\beta_v^n}$ reduction replaces the bound variable of a λ-abstraction by the values of the operands using evaluation substitutions (written $\mathfrak{C}\mathfrak{D}$). Hence, $\Lambda_n^{\text{Module}}$ extends evaluation substitutions to work on import paths and uses them to describe linking. However, linking does not directly generate the evaluation substitutions when importing a module. Instead, it first records the imported names along with their values in a special data structure called the *template* (See Section 2.3). As defined in Section 2.3, a module with a template is called a *package*. After linking has resolved all imports, evaluation of the definitions of the package generates evaluation substitutions from the template.

Definitions are a new concept in $\Lambda_n^{\text{Module}}$—$\Lambda_n$ only encompasses expressions. Hence, a closer look at the semantics of definitions is necessary. We model them like top-level definitions in Scheme: the definitions are mutually recursive and support mutation. The denotational semantics for Scheme achieves both features by adding an additional layer of locations to the mapping from identifiers to values. A location is a mutable place in the store. Scheme binds identifiers to locations and dereferences the locations for variable occurrences. The `set!` expression mutates the binding of an identifier by mutating the location the identifier is bound to. Locations also help expressing recursive bindings such as global definitions. The evaluation of the right-hand side of a definition proceeds with an environment where the name of the definition is bound to an uninitialized location. After the evaluation of the right-hand side, evaluation sets the location to the value of the right-hand side.

To support locations with state, we adopt the representation of the denotational semantics of R^5RS [KCR98] and add a store and locations to the abstract syntax. Furthermore, *set-loc!* expressions set the contents of a location and *ref* expressions dereference locations.

In the presence of state, it is also necessary to specify whether a module imports the bindings or the values of the variables from another module. The reason is that if a module mutates a variable that has been exported to other modules, the two cases yield different effects: If the binding of a variable is exported, all modules importing such a mutated variable see the new value of the variable after mutation; if the value is exported, mutation has no effect on the other modules. The description of templates earlier in this section described that importing extends the template by a pair of name and value. Hence, our modules export values. However, the value of a top-level variable is always a location and mutation modifies the value at the location in the store. Consequently, modules perceive the mutation of imported variables.

The presence of macros obscures which variables linking needs to replace by values in order to import a module. This set of variables is not equal to the variables listed in the interface nor to the variables contained in the set of definitions of the exporting module:

1. An imported macro can expand into code containing variables bound by the exporting module but not listed in the interface. We call these variables "hidden exports."

2. A module can export variables it imported from another module. Such an export is called a "re-export;" it is not contained in the set of definitions.

3. Finally, a combination of the two cases above is possible: An imported macro can expand into code with free variables not listed in the interface, and the exporting module imports these variables from another module. We call them "hidden re-exports."

We will now show how linking binds variables of these three classes to values.

Hidden Exports Unfortunately, it is impossible to derive the list of hidden exports from the macro definition. Finding this list amounts to determining the identifiers inserted by the macro:

- For `syntax-rules`-based macros the identifiers used are the identifiers in the right-hand side of the transformation rules that are used outside of a quotation. However, distinguishing quoted from unquoted code needs to track calls to other macros and observe bindings of the `quote` syntax. The latter is impossible to analyze statically. Assume, the following macro is exported:

  ```
  (define-syntax foo
    (syntax-rules ()
      ((foo a) (a b))))
  ```

  For the macro call `(foo quote)`, the `foo` macro has no hidden exports. For macro calls `(foo x)`, where x is bound to a variable, `foo` exports the identifier `b`.

- For computational macros as presented in Section 3.6 another difficulty arises: These macros may compute the output and, via `datum->syntax-object`, also introduce identifiers whose names have been computed (e.g. from a string). In this case, the set of inserted identifiers for these macros is clearly undecidable and may include any identifier.

To conclude, the list of identifiers inserted by a macro is not computable.

There seem to be two ways to accomplish linking of hidden exports despite the impossibility to predict the free variables in the output of macros: By explicitly stating the list of indirectly exported identifiers as in Blume's proposal [Blu97] or by simply trying to link all identifiers defined in the exporting module. As our identifiers have an associated import path, the latter technique is feasible. The linker learns from the import path when a variable refers to the module the linker is just about to import. Unlike Blume's approach this technique also supports computational macros that introduce arbitrary identifiers: such identifiers still have an import path that guides the linker but as argued above it is, in the general case, not practical to list all possible identifiers in advance as required by Blume. A drawback of our approach is that a compiler does not know which variables of a module are exported and must hence be careful with certain optimizations such as inlining these definitions. In addition, it is not statically decidable whether a module implements its export interface because the module might fail to define some of the hidden exports of the interface. On the other hand, for Blume's proposal it is impossible to decide statically whether the list of indirect exports contains all identifiers inserted by the exported macros and hence to check for an interface whether the declarations of indirect exports and the macro definitions are consistent which each other.

Re-Exports Dealing with re-exports is next. Consider the following example:

```
(define-interface a-interface
  (export a))

(define-module mod-a a-interface
   (open scheme-interface)
   (begin
      (define a 23)))

(define-interface b-interface
  (export a))

(define-module mod-b b-interface
  (open scheme-interface a-interface))

(define-module mod-c (export)
  (open scheme-interface b-interface)
  (begin
```

```
    (define in-c a)))

(define-program p
   (modules mod-a mod-b mod-c))
```

Here, module `mod-a` exports identifier `a` via interface `a-interface`, module `mod-b` imports `a-interface` and exports identifier `a` via `b-interface`, and module `mod-c` imports interface `b-Rinterface`, an occurrence of identifier `a` in module `mod-c` will be parsed as a reference to `b-interface`. The reason is that during the (independent) compilation of module `mod-c`, the parser does not know which module will satisfy the import `b-interface`. However, module `mod-b` does not contain a definition of `a`. Consequently, parsing and macro expansion leads to:[2]

```
(define-module mod-a a-interface
  (open scheme-interface)
  (begin
    (define a 23)))

(define-module mod-b b-interface
  (open scheme-interface a-interface))

(define-module mod-c (export)
  (open scheme-interface b-interface)
  (begin
    (define in-c a^0_b-interface)))
```

That is, the variable `a` in module `mod-c` refers to the interface `b-interface`. During linking, module `mod-b` will satisfy this import but does not provide a definition for `a` as it re-exports `a`.

One approach to support re-exports is to add definitions for all imported variables to each module. The right-hand side of these definitions are a reference to the imported variable. Thus, the expansion of `mod-b` becomes:

```
(define-module mod-b b-interface
  (open scheme-interface
        a-interace)
  (begin
    (define a a^0_a-interface)))
```

and the linker can replace the reference in module `mod-c` by the value from module `mod-b`. Adding these definitions is always possible as the imports must be disjoint from the set of definitions. A consequence of this approach is that it introduces a new binding for the variable, which might be undesirable. Alternatively, the linker could add an entry to the template for every variable in the interface. The identifier of the entry would have an empty import path. The value of the entry would correspond to the imported value. As the imported values are the locations of the variables, this approach would preserve the binding of the exported variable. We chose the first approach because it seems easier to implement.

Hidden Re-Exports Hidden re-exports require less work than non-hidden re-exports: a hidden re-export still references the defining module. However a non-hidden re-export is a reference to the re-exporting module, not to the defining module. In the example above, if module `mod-b` would define and export a macro `m` that inserts a reference to identifier `a`, the reference to identifier `a` in the expansion of `m` within `mod-b` would point to module `mod-a` as imported identifiers are bound in the right-hand sides of the macros. If module `mod-b` exports `m`, the reference to `a` in the expansion of `m` within `mod-c` would get as its import path `b-interface,a-interface`:

[2] We always use the names of the interfaces in the configuration language instead of the actual interfaces to represent import paths.

```
(define-interface a-interface
  (export a))

(define-module mod-a a-interface
  (open scheme-interface)
  (begin
    (define a 23)))

(define-interface b-interface
  (export
     (define-syntax m
        (syntax-rules ()
           ((m) a)))))

(define-module mod-b b-interface
  (open scheme-interface
        a-interface)
  (begin
    (define b (m))))

(define-module mod-c (export)
  (open scheme-interface
        b-interface)
  (begin
    (define hurz (m))))

(define-program p
   (modules mod-a mod-b mod-c))
```

Parsing and macro expansion leads to:

```
(define-module mod-a a-interface
  (open scheme-interface)
  (begin
    (define a 23)))

(define-module mod-b b-interface
  (open scheme-interface
        a-interface)
  (begin
    (define b a^0_a-interface)))

(define-module mod-c (export)
  (open scheme-interface b-interface)
  (begin
    (define hurz a^0_b-interface,a-interface)))
```

For linking of hidden re-exports, it is neither necessary nor would it be correct to extend the set of definitions of the re-exporting module: The import path of a hidden re-export already contains a reference of the defining module and, as the export is hidden, there is no guarantee that the re-exporting module does not itself have a definition for this name. Instead, an extension of the "link all" mechanism the linker uses to resolve hidden exports is required: Rather than adding the defined variables of the imported package to the template, the linker simply adds the contents of the template of the imported package to the template—with the import path extended by the interface the linker is about to satisfy. This way, the linker also adds entries for the variables

that the imported module has imported itself. In the example above, the linker adds an entry for the identifier $\mathtt{a}^0_{\mathtt{a\text{-}interface}}$ to the template of `mod-b` while importing `mod-a` and during linking `mod-c`, the linker adds entries for the identifiers $\mathtt{a}^0_{\mathtt{a\text{-}interface}}$ and $\mathtt{b}^0_{\mathtt{b\text{-}interface,a\text{-}interface}}$. Of course, the template can get quite large this way and a realistic linker would add only the identifiers referenced in the module body and remember the identifiers it has omitted (along with the export interface) for future linking steps where this template is imported.

The concept of hidden exports and re-exports extends to keywords. Keywords need to be treated by the macro expander only; the linker does not see keywords. Consequently, the template does not map keywords to macro transformers.

At this point, it is also clear why the contents of interfaces do not matter for linking: The linker may import all variables from the template of the exporting package because it adjusts the import path of the variables in the template of the importing package. It is the task of the parser and the macro expander to introduce only references to imported identifiers which are either listed in imported interfaces or inserted as hidden exports by imported macros.

4.2 The $\Lambda_n^{\text{Module}}$ Calculus

This section defines the syntax and an operational semantics for $\Lambda_n^{\text{Module}}$, a language with modules based on the call-by-value lambda calculus. As motivated above, the mutually recursive definitions require the introduction of a store. The value of a variable bound by a definition is a location in the store, and the store maps the location to the value proper. However, the programmer does not deal with locations directly. Instead, we assume that a pre-processing phase has augmented all references to variables bound by definitions with the new primitive *ref*, which fetches the value of the location from the store. Analogously, mutations of these variables are assumed to be replaced by calls to the primitive *set-loc*, which stores the value in the store at the place denoted by the location bound to the variables. For the sake of simplicity, λ-abstractions still bind variables directly to values, and these variables do not support mutation.

In our calculus with modules, evaluation is not to limited to expressions but must cover the evaluation of a program and its modules as well. Thereby, it needs to incorporate the generation of packages from modules and the dynamic linking of packages. To perform these tasks, the semantics defines the following reductions:

- The reduction $\rightarrow_{\text{module}}$ reduces a module to a package. As introduced in Section 2.3, a package is a module with state, represented by the template that maps identifiers to values. The values in the template are locations in the store that point to the actual values. The reduction $\rightarrow_{\text{module}}$ initializes the template with a mapping from the names of the module's definitions to fresh locations, which point to a default value in the store.

- The reduction $\rightarrow_{\text{link}}$ performs the dynamic linking of packages. Each reduction step satisfies one import of a package: It extends the template of the importing package with the contents of the template of the exporting package, extending the import path of the imported identifiers by the interface of the exporting module.

- The reduction $\rightarrow_{\text{eval-package}}$ accepts a fully-linked package and evaluates the definitions of the packages. It first uses the reduction $\rightarrow_{\text{eval}}$ to evaluate the expression on the right-hand side of the definition. Afterwards, reduction $\rightarrow_{\text{eval-package}}$ records the resulting value in the store where it replaces the default value.

- The reduction $\rightarrow_{\text{eval}}$ is the ordinary expression evaluation reduction for Λ_n from Section 3.2 augmented by reduction rules that deal with locations and hence access the store.

- The reduction $\rightarrow_{\text{prog}}$ drives the overall evaluation of programs. It uses the reduction $\rightarrow_{\text{module}}$ to turn modules into packages, the reduction $\rightarrow_{\text{link}}$ to link packages, and the reduction $\rightarrow_{\text{eval-package}}$ to evaluate fully-linked packages. At the beginning, a program is a set of

modules, at the end, $\rightarrow_{\text{prog}}$ has turned each of these modules into a fully-linked, evaluated package.

This section contains the definitions of these reductions and the accordant abstract syntax. It also extends the $\rightarrow_{\text{eval}}$ reduction to cover higher-order modules. This evaluation of higher-order modules works by augmenting the $\rightarrow_{\text{eval}}$ reduction with the $\rightarrow_{\text{module}}$ reduction to evaluate module expressions to packages and including rules for primitive operations that use the $\rightarrow_{\text{link}}$ and the $\rightarrow_{\text{eval-package}}$ reductions to link and evaluate packages. Hence the semantics employs the three reductions $\rightarrow_{\text{module}}$, $\rightarrow_{\text{link}}$, and $\rightarrow_{\text{eval-package}}$ in two different contexts: once during program evaluation for top-level modules and once during expression evaluation for higher-order modules. Finally, to finish support for higher-order modules, the elimination of evaluation substitutions needs to be extended to cover packages as well.

4.2.1 Abstract Syntax

We first define in Figure 4.1 the abstract syntax for the evaluation of $\Lambda_n^{\text{Module}}$. Its starting basis is the abstract syntax for Λ_n from Figure 3.2. As already mentioned in the examples from Section 4.1, identifiers now contain an import path in addition to the label and are written as $x^n_{i\ldots}$ where n is the level and $i\ldots$ is a sequence of interfaces. An interface, written $I_\iota(x\ldots)$, consists of a unique identifier ι and a list of exported variables, $(x\ldots)$. This representation is sufficient during program evaluation: Linking uses the unique identifier to compare interfaces; the list of exported variables limits access to the contents of a module at run time. During parsing and macro expansion, different representations of interfaces are required as shown in Section 4.3. Applications remain unchanged and evaluation substitutions still have the form $e\langle\!|t|\!\rangle$. However, in $\Lambda_n^{\text{Module}}$ an evaluation substitution carries a sequence of pairs of identifiers and values, which is reflected in the new definition of t. As we will see later, evaluation of a package body generates such an extended evaluation substitution from the contents of the package's template. We extend the set of expressions by applications of primitives, written as $(@@\ x\ e\ldots)$ and assume that the compiler has inserted the @@ after macro expansion. In addition, modules, written as ${}^{i}M_{\mu}^{\texttt{open}\ i\cdots}\mathit{defs}$, extend the set of expressions. For a module, μ is the unique identifier of the module, i is the export interface of the module, $i\ldots$ is the set of interfaces that the module imports, and *defs* are the top-level definitions of the module. A single definition *def*, written $(\Delta\ x\ e)$, is a pair that comprises the name x and the associated expression e. A package starts out with the imported interfaces and the definitions inherited from the corresponding module. As linking proceeds, it resolves the interfaces and replaces them by references in the template. The final result of linking is a package without imports. For each definition, evaluation reduces the right-hand side to a value, mutates the corresponding location in the store to record the value, and removes the definition from the package. At the end, evaluation has removed all definitions and the package is fully evaluated. To represent this process in the abstract syntax, two sorts of packages extend the set of values: *Packages* are packages which have not yet been fully linked and evaluated, and *EvaluatedPackages* are evaluated packages. *EvaluatingPackages* is a subset of *Packages* and contains the packages which have been linked but not yet fully evaluated. The abstract syntax represents members of *Packages* as ${}^{i}Pg_{\mu}^{\texttt{open}\ i\cdots}\mathit{defs}\langle t\rangle$, where μ is unique identity of the module the package represents at run-time, i and $i\ldots$ are the export and the imported interfaces of the package respectively, and t is the contents of the package's template. For these contents, the abstract syntax uses the same representation as for evaluation substitutions: a sequence of pairs $\langle x^n_{i\ldots}, v\rangle$, where $x^n_{i\ldots}$ is an identifier and v a value. We will use this similarity later when we use the bindings from the template for the evaluation of expressions within the package. Linking proceeds by removing interfaces from the list of imported interfaces (and augmenting the template). Hence, for a fully-linked but not evaluated package, the list of imported interfaces is empty. The meta-variable *evp* ranges over these evaluating packages. An evaluated package is written as ${}^{i}Pg_{\mu}\langle t\rangle$ and contains only the unique identity of the corresponding module, the export interface, and the template.

The set of values contains constants and λ-abstractions as before. The new values are packages in all stages, locations *loc*, and quoted s-expressions as presented in Section 3.10.

Syntactic Domains:

a	$\in Const$
x	$\in Vars$
e	$\in \Lambda_n^{\text{Module}}$
im	$\in ImportPaths = RunTimeInterfaces^*$
se	$\in S\text{-}Expressions$
t	$\in (Ids \times Values)^*$
def	$\in TopDefinition$
$defs$	$\in TopDefinition^*$
loc	$\in Locations$
$store$	$\in Stores$
θ	$\in \mathcal{P}(Locations \times Values)$
i	$\in RunTimeInterfaces$
ι	$\in InterfaceIdentifiers$
m	$\in Modules$
μ	$\in ModuleIdentifiers$
p	$\in Packages$
evp	$\in EvaluatingPackages \subset Packages$
ep	$\in EvaluatedPackages$
l	$\in Modules \times RunTimeInterfaces \times Modules$
le	$\in \mathcal{P}(l)$
$prog$	$\in Programs$

Abstract Syntax:

e	$::= x_{im}^n \mid (@\ e\, e \ldots) \mid e\langle\!\langle t \rangle\!\rangle \mid (@@\ x\, e \ldots) \mid m \mid v$
im	$::= i \ldots$
v	$::= a \mid (\lambda\ x.e) \mid p \mid evp \mid ep \mid loc \mid (\text{'}se)$
t	$::= \langle x_{im}^n, v \rangle\ \ldots$
def	$::= (\Delta\ x\ e)$
$defs$	$::= def \ldots$
i	$::= I_x(x \ldots)$
m	$::= {}^{i}M_{\mu}^{\texttt{open}\ i \cdots}\, defs$
p	$::= {}^{i}Pg_{\mu}^{\texttt{open}\ i \cdots}\, defs\langle t \rangle$
evp	$::= {}^{i}Pg_{\mu}^{\texttt{open}}\ defs\langle t \rangle$
ep	$::= {}^{i}Pg_{\mu}\langle t \rangle$
l	$::= (\mu,i,\mu)$
le	$::= \{l \ldots\}$
$prog$	$::= Prog\ le\ p \ldots m \ldots \mid Prog\ le\ ep \ldots evp\ p \ldots$
$store$	$::= \theta \Vdash prog \mid \theta \Vdash m \mid \theta \Vdash p \mid \theta \Vdash e$
θ	$::= \{(loc\ v), \ldots\}$

Figure 4.1: Abstract syntax for programs with modules

The abstract syntax represents programs as *Prog le p ... m ...* or as *Prog le ep ... evp p ...*. In the first case, program evaluation is about to turn all modules into packages. Once this is finished, the second case applies: a program contains a linking environment *le* and a sequence of evaluated packages, *ep...*, a package that is currently subject to evaluation (*evp*), followed by non-evaluated packages (*p...*). The link environment is a set of links (μ_1, i, μ_2), which means that the module with the unique module identifier μ_2 satisfies the interface import i of the module with the unique module identifier μ_1.

Finally, programs, modules, packages, and expressions can be paired up with a store. The configuration of a store, written as θ, is a set of pairs consisting of a location and a value. We write $\Vdash$ to separate the store configuration from the entity that contains the references into the store.

4.2.2 Evaluation of Programs

With the abstract syntax for programs and modules in place, we can turn to the description of the evaluation. Figure 4.2 shows the evaluation of programs as a reduction $\rightarrow_{\text{prog}}$. The purpose of program evaluation is to control the initialization, linking, and evaluation of packages. There are three rules. The first, (ProgramEvalModule), evaluates the top-level modules of the program into packages. The rule uses the $\rightarrow_{\text{EvalModule}}$ reduction to create the template for the package. Figure 4.3 contains the rules for this reduction, it will be described next. The remaining rules work on programs consisting of packages only, hence (ProgramEvalModule) first turns all modules into packages. Initializing all packages before evaluation is necessary for mutually recursive modules, even though we do not cover them here. The second rule, (ProgramLink), selects the first non-evaluated package from the program's package. In the rule, the module identifier of this package is μ_{imp}. For μ_{exp}, the first interface this package imports, the link $(\mu_{\text{imp}}, i_{\text{exp}}, \mu_{\text{exp}})$ in the link environment determines the package with module identifier μ_{exp} that satisfies this import. We assume this package to be fully linked, hence modules cannot be mutually recursive. Section 4.6 sketches the necessary extensions to support mutually recursive modules. The reduction $\rightarrow_{\text{link}}$ performs the actual linking. It takes as redex the triple consisting of the importing package, the interface, and the exporting package and reduces to the importing package with the template extended by the pairs from the exporting package. Figure 4.4 contains the $\rightarrow_{\text{link}}$ reduction, it will be described below. The third rule of the $\rightarrow_{\text{prog}}$ reduction, (ProgramEval) evaluates the first linked but not evaluated package *evp*. To that end, it reduces the package using the standard reduction of $\rightarrow_{\text{eval-package}}$ for evaluating packages from Figure 4.5. Package evaluation affects the store, hence (ProgramEval) evaluates the package with the current store configuration and returns a new store.

The three reductions used by program reduction are presented next. Figure 4.3 contains the reduction for modules. It contains only one rule, (EvalModule), that evaluates a module to a package. The difference between a module and a package is the presence of state in the package. The template represents this state by providing a mapping from identifiers to mutable locations. The rule initializes the template with bindings for the definitions of the module. The value is initially set to 0, but in real life some special value that indicates an non-initialized location would be used. As this rule acts as a reduction rule for higher-order modules as well, it uses the context $E_{(\!|\,|\!),v}$ to select the module to be reduced within an expression. The description of expression evaluation in Section 4.2.3 contains the definition of this context. For now, we can assume it to be defined as the hole [ ], because during reduction of top-level modules in rule (ProgramEvalModule), this context is indeed always empty. Section 4.2.3 integrates this rule into the evaluation of expressions. The rule uses a function *gen-loc* to generate fresh locations in the store.

Next, the linking reduction $\rightarrow_{\text{link}}$ performs the dynamic linking of a package. It extends the template of a package by bindings for the variables from an imported module and also adds bindings for (possibly) re-exported variables to the template. Figure 4.4 contains the rules for dynamic linking. The reduction $\rightarrow_{\text{link}}$ consists of three reductions each containing just one rule. Using this split, the types of the rules can be defined more precisely.

$$\theta \Vdash \mathit{Prog}\ \mathit{le}\ p \ldots\ m_1\ m_2 \ldots \rightarrow_{\text{prog}} \theta' \Vdash \mathit{Prog}\ \mathit{le}\ p \ldots\ p_1\ m_2 \ldots \quad \text{(ProgramEvalModule)}$$
$$\text{iff } \theta \Vdash m_1 \mapsto_{\text{module}} \theta' \Vdash p_1$$

$$\theta \Vdash \mathit{Prog}\ \{l_1 \ldots \langle \mu_{\text{i}}, i_{\text{e}}, \mu_{\text{e}} \rangle l_2 \ldots\}\ \mathit{ep}_1 \ldots\ ({}^{i_{\text{e}}}\mathit{Pg}_{\mu_{\text{e}}}\langle t_{\text{e}}\rangle)\ \mathit{ep}_2 \ldots\ ({}^{i'_{\text{e}}}\mathit{Pg}_{\mu_{\text{i}}}^{\texttt{open}\ i_{\text{e}} i \cdots}\mathit{defs}\langle t_{\text{i}}\rangle)\ p_2 \ldots \rightarrow_{\text{prog}} \quad \text{(ProgramLink)}$$
$$\theta \Vdash \mathit{Prog}\ \{l_1 \ldots l_2 \ldots\}\ \mathit{ep}_1 \ldots\ ({}^{i_{\text{e}}}\mathit{Pg}_{\mu_{\text{e}}}\langle t_{\text{e}}\rangle)\ \mathit{ep}_2 \ldots\ p_1\ p_2 \ldots$$
$$\text{where } \left\langle ({}^{i'_{\text{e}}}\mathit{Pg}_{\mu_{\text{i}}}^{\texttt{open}\ i_{\text{e}} i \cdots}\mathit{defs}\langle t_{\text{i}}\rangle), i_{\text{e}}, ({}^{i_{\text{e}}}\mathit{Pg}_{\mu_{\text{e}}}\langle t_{\text{e}}\rangle) \right\rangle \mapsto^*_{\text{link}} p_1$$

$$\theta \Vdash \mathit{Prog}\ \mathit{le}\ \mathit{ep}_1 \ldots\ \mathit{evp}\ p \ldots \rightarrow_{\text{prog}} \theta' \Vdash \mathit{Prog}\ \mathit{le}\ \mathit{ep}_1 \ldots\ \mathit{ep}\ p \ldots \quad \text{(ProgramEval)}$$
$$\text{iff } \theta \Vdash \mathit{evp} \mapsto^*_{\text{eval-package}} \theta' \Vdash \mathit{ep}$$

Figure 4.2: Program reduction

$$(\mathit{loc}\ v) \ldots \Vdash E_{(\!|\ |\!)}[{}^{i_{\text{e}}}M_{\mu}^{\texttt{open}\ i^*}(\Delta\ x\ e_1) \ldots] \rightarrow_{\text{module}} \quad \text{(EvalModule)}$$
$$(\mathit{loc}\ v) \ldots (\mathit{loc}_{\text{fresh}}\ 0) \ldots \Vdash E_{(\!|\ |\!)}[({}^{i_{\text{e}}}\mathit{Pg}_{\mu}^{\texttt{open}\ i^*}(\Delta\ x\ e_1) \ldots \langle t' \rangle)]$$
$$\text{where } \mathit{loc}_{\text{fresh}} \ldots \text{fresh and } t' = \langle x^0, \mathit{loc}_{\text{fresh}} \rangle \ldots$$

Figure 4.3: Module reduction

First, rule (Link) reduces a triple consisting of the importing package, the name of the interface to be linked, and the exporting package to a new triple consisting of the importing package with the template extended by the imports, the names from the interface, and the exporting package. The figure defines the type of this triple as *LinkingMap*. The linker uses the identifiers from the template of the exporting package and extends their import path by the interface to be linked. That way, the linker builds up the import paths for the variables in the template. Sections 4.3 and 4.4 explain how the parser and the macro expander build the corresponding import paths for variable references. Next, the two remaining rules process the resulting triple. For the first exported variable, rule (LinkReExport) adds a new variable/value pair to the template. The variable receives zero as its level and an empty import path, just like a variable defined within the package. Adding such a variable to the template is always possible as imports must be disjoint from the variables defined in a module. The rule takes the value for the new template entry from the exporting package. As this value is always a location, a mutation of the re-exported variable will affect the original binding as well. The last rule (FinishLinkReExport) matches when the list of exported identifiers is empty and reduces to the importing package whose template now contains all exports of the exporting package.

Evaluation of packages comes next. Evaluation of a package can only proceed after linking has resolved all imports of the package. Then evaluation of a package evaluates the set of definitions of the packages and moves the values of the evaluated definitions to the template.

Figure 4.5 contains the single rule (EvalPacDef) for the evaluation of packages. The rule evaluates the right-hand side of the first top-level definition using the evaluation reduction for expressions. The template of the package provides the variable bindings for the evaluation of the expression. To that end, the rule binds the contents of the template to the meta-variable t and applies the evaluation substitution $(\!|\ t\ |\!)$ to the expression e. The evaluation starts with store θ and produces a new store and a value v_{nd}. To record this value as the value of the definition, the rule (EvalPacDef) modifies the store just obtained. It gets the accordant location *loc* from

$$
\begin{aligned}
\mathit{LinkingMap} &= \mathit{EvaluatingPackages} \times \mathit{Vars}^* \times \mathit{EvaluatedPackages} \\
&\rightarrow_{\mathrm{link}_1} \subseteq (\mathit{EvaluatingPackages} \times \mathit{RunTimeInterfaces} \times \mathit{EvaluatedPackages}) \times \mathit{LinkingMap} \\
&\rightarrow_{\mathrm{link}_2} \subseteq \mathit{LinkingMap} \times \mathit{LinkingMap} \\
&\rightarrow_{\mathrm{link}_3} \subseteq \mathit{LinkingMap} \times \mathit{EvaluatingPackages}
\end{aligned}
$$

$$
\left\langle ({}^{i_\mathrm{e}}Pg_{\mu_\mathrm{i}}^{\texttt{open } i_1 \ldots (I_\iota(x\ldots))i_2\cdots}\, \mathit{defs}\langle\langle x_{i\ldots}^n, v\rangle \ldots\rangle), (I_\iota(x\ldots)), ({}^{(I_\iota(x\ldots))}Pg_{\mu_\mathrm{e}}\langle\langle x_{i_{\mathrm{e}}\ldots}^{n_e}, v_e\rangle \ldots\rangle)\right\rangle \rightarrow_{\mathrm{link}_1} \quad \text{(Link)}
$$

$$
\left\langle ({}^{i_\mathrm{e}}Pg_{\mu_\mathrm{i}}^{\texttt{open } i_1 \ldots i_2\cdots}\, \mathit{defs}\langle\langle x_{i\ldots}^n, v\rangle \ldots \langle x_{e_{(I_\iota(x\ldots))i_{\mathrm{e}}\ldots}}^{n_e}, v_e\rangle \ldots\rangle), (x, \ldots), ({}^{(I_\iota(x\ldots))}Pg_{\mu_\mathrm{e}}\langle\langle x_{i_{\mathrm{e}}\ldots}^{n_e}, v_e\rangle \ldots\rangle)\right\rangle
$$

$$
\left\langle ({}^{i_\mathrm{e}}Pg_{\mu_\mathrm{i}}^{\texttt{open } i^*}\, \mathit{defs}\langle\langle x_{i\ldots}^n, v\rangle \ldots\rangle), (y, w, \ldots), ({}^{i'_\mathrm{e}}Pg_{\mu_\mathrm{e}}\langle t\rangle)\right\rangle \rightarrow_{\mathrm{link}_2} \quad \text{(LinkReExport)}
$$

$$
\left\langle ({}^{i_\mathrm{e}}Pg_{\mu_\mathrm{i}}^{\texttt{open } i^*}\, \mathit{defs}\langle\langle x_{i\ldots}^n, v\rangle \ldots \langle y^0, v_2\rangle\rangle), (w, \ldots), ({}^{i'_\mathrm{e}}Pg_{\mu_\mathrm{e}}\langle t\rangle)\right\rangle \quad \text{where } (y^0, v_2) \in t
$$

$$
\left\langle ({}^{i_\mathrm{e}}Pg_{\mu_\mathrm{i}}^{\texttt{open } i^*}\, \mathit{defs}\langle t_\mathrm{i}\rangle), (), ({}^{i'_\mathrm{e}}Pg_{\mu_\mathrm{e}}\langle t_\mathrm{e}\rangle)\right\rangle \rightarrow_{\mathrm{link}_3} ({}^{i_\mathrm{e}}Pg_{\mu_\mathrm{i}}^{\texttt{open } i^*}\, \mathit{defs}\langle t_\mathrm{i}\rangle) \quad \text{(FinishLinkReExport)}
$$

$$
\rightarrow_{\mathrm{link}} = \rightarrow_{\mathrm{link}_1} \cup \rightarrow_{\mathrm{link}_2} \cup \rightarrow_{\mathrm{link}_3}
$$

Figure 4.4: Linking reduction

the template of the package (where x^0 maps to *loc*) and replaces the old value v by the new value. Finally, the rule discards the definition because it has now been evaluated and its value is accessible to the other definitions and to other packages through the location *loc*.

$$\theta \Vdash ({}^{i_e}Pg_{\mu}^{\texttt{open}}\ (\Delta\ x\ e)\mathit{def}\ldots\langle t\rangle) \rightarrow_{\text{eval-package}} \quad \text{(EvalPacDef)}$$
$$(\mathit{loc}_1\ v_{11})\ldots(\mathit{loc}\ v_{\text{new}})(\mathit{loc}_2 v_{22})\ldots \Vdash ({}^{i_e}Pg_{\mu}^{\texttt{open}}\ \mathit{def}\ldots\langle t\rangle)$$
$$\text{where } (x^0 \mathit{loc}) \in t$$
$$\text{iff } \theta \Vdash e(\!|t|\!) \mapsto^*_{\text{eval}} (\mathit{loc}_1\ v_{11})\ldots(\mathit{loc}\ v_{\text{old}})(\mathit{loc}_2 v_{22})\ldots \Vdash v_{\text{new}}$$

Figure 4.5: Evaluation of packages

4.2.3 Evaluation of Module Expressions

The evaluation of expressions in the package body follows the rules from the call-by-value calculus with explicit substitution and levels from Section 3.3 with the following extensions:

- Evaluation uses a store to record the values of variables bound by definitions. $\mapsto_{\text{eval}}$ needs to be re-defined to reduce β-redexes contained in terms with a store.
- Locations extend the set of values. The semantics includes the reduction $\rightarrow_\delta$ to describe the primitives *ref* and *set-loc*, which dereference and mutate locations.
- Higher-order modules extend the set of expressions. The reduction $\rightarrow_{\text{module}}$ turns them into packages, which in turn extend the set of values.
- The reduction $\rightarrow_\delta$ also describes the primitives `link`, `eval-package`, and `package-binding` from Section 2.2. These primitives manipulate packages and are part of the higher-order modules facility.

As a consequence of these extensions, the elimination reduction for evaluation substitutions $\rightarrow_{(\!|\,|\!)}$ needs to cover locations and packages. Hence we extend the rules from Figure 3.3 accordingly. The evaluation reduction $\mapsto_{\text{eval}}$ can then be defined as

$$\mapsto_{\text{eval}} = \mapsto_{\beta_v^{n,\text{Module}}} \cup \rightarrow_\delta \cup \rightarrow_{\text{module}} \cup \mapsto_{(\!|\,|\!)}$$

To lift the β-reduction into expressions under a store, we use a modified version of evaluation contexts that place the hole under the store:

E' ::= $\theta \Vdash E$

Using E' as evaluation context, the standard reduction function of the reduction $\rightarrow_{\beta_v^n}$ from Section 3.3 is now defined as:

$$e \mapsto_{\beta_v^{n,\text{Module}}} e' \text{ iff } e = E'[e_1], e' = E'[e_2], e_1 \rightarrow_{\beta_v^n} e_2 \text{ for some } E'$$

Next, we add the primitives *ref* and *set-loc* to the calculus. The argument of *ref* must be a location. The application of the *ref* primitive evaluates to the value that the location maps to in the store. The *set-loc* primitive accepts two arguments: a location and an arbitrary value. *set-loc* replaces the value of the location in the store by its second argument. Figure 4.6 contains the rules for primitives. They are part of the reduction $\rightarrow_\delta$ that describes the evaluation of primitives. The rule (EvalPrimRef) dereferences the location loc_d and reduces to the value v_d from the store. The application of the primitive takes place in the evaluation context E under the store. Rule (EvalPrimSetLoc) changes the value of the location loc_d from the old value v_d to its second argument v_{nd}. It also reduces to this new value. Figure 4.3 already contains the reduction

$$(loc_1\ v_1)\dots(loc_d\ v_d)(loc_2\ v_2)\dots \Vdash E[(@@\ ref\ loc_d)] \to_\delta (loc_1\ v_1)\dots(loc_d\ v_d)(loc_2\ v_2)\dots \Vdash E[v_d] \quad \text{(EvalPrimRef)}$$

$$(loc_1\ v_1)\dots(loc\ v_{\text{old}})(loc_2\ v_2)\dots \Vdash E[(@@\ \textit{set-loc!}\ loc\ v_{\text{new}})] \to_\delta (loc_1\ v_1)\dots(loc\ v_{\text{new}})(loc_2\ v_2)\dots \Vdash E[v_{\text{new}}] \quad \text{(EvalPrimSetLoc)}$$

Figure 4.6: Evaluation of store-related primitives

$\to_{\text{module}}$ for evaluating modules. Its single rule (EvalModule) turns the module into a package and uses the context $E_{(\!(\,)\!),v}$ to select the module to be reduced. We now define this context to place its hole in an evaluation context and there under all evaluation substitutions:

$$\begin{array}{ll} E_{(\!(\,)\!),v} & ::= E'_{(\!(\,)\!)} \mid (@\ v\dots E_{(\!(\,)\!),v} e\dots) \mid (@@\ x v\dots E_{(\!(\,)\!),v} e\dots) \\ E'_{(\!(\,)\!)} & ::= [\,] \mid E'_{(\!(\,)\!)}(\!(t)\!) \end{array}$$

This definition still contains the empty context as needed for rule (ProgramEvalModule).

For the final item from the list of extensions, the definition of module-related primitives, Figure 4.7 extends the reduction $\to_\delta$. Three further primitives deal with higher-order modules. The first, called *link!*, performs one linking step on a package. It returns the package with one import replaced by its third argument. To that end, the corresponding rule (EvalPrimLink) uses the reduction $\to_{\text{link}}$ from Figure 4.4. The rule expects the interface to be specified as a quoted s-expression.

The rule (EvalPrimEvalPac) describes the evaluation of the `eval-pac` primitive. This primitive triggers the evaluation of the definitions of a package. Again, the rule resorts to a reduction from the previous section, this time $\to_{\text{eval-package}}$ from Figure 4.5 for evaluation the definitions of packages.

The third module-related primitive, called `package-binding`, references one binding from the package template. The first argument of the primitive is the package and the second argument is a symbol denoting the name of the definition to be referenced. The rule (EvalPrimPacBind) describes the reduction of the primitive. It maps the symbol with name w to an identifier with level 0 and empty import path and looks up the corresponding value in the template of the package. The rule also ensures that the identifier is listed in the export interface of the package. This check is the only place where access to the export interface of the package is necessary. Note that there is no program evaluation rule corresponding to this primitive as the bindings of top-level modules are only accessed by other modules. The program itself is not interested in these bindings.

$$\theta \Vdash E[(@@\ \texttt{link!}\ p\ (\texttt{'(interface}\ \iota\ \texttt{(free) (open) ()}\ (x\dots)))\ ep)] \to_\delta \theta \Vdash E[p'] \text{ iff } \langle p, (I_\iota(x\dots)), ep\rangle \mapsto^*_{\text{link}} p' \quad \text{(EvalPrimLink)}$$

$$\theta \Vdash E[(@@\ \texttt{eval-pac}\ evp)] \to_\delta \theta' \Vdash E[ep] \text{ iff } \theta \Vdash evp \mapsto^*_{\text{eval-package}} \theta' \Vdash ep \quad \text{(EvalPrimEvalPac)}$$

$$\theta \Vdash E[(@@\ \texttt{package-binding}\ ({}^{(I_{\iota_e}(x_4\dots w\ x_5\dots))} Pg_\mu\langle\langle x^n_{i\dots}, v_1\rangle\dots\langle w^0, v_3\rangle\langle y^m_{j\dots}, v_2\rangle\dots\rangle)\ (\texttt{'}w))] \to_\delta \theta \Vdash E[v_3] \quad \text{(EvalPrimPacBind)}$$

Figure 4.7: Evaluation of module-related primitives

Finally, the elimination of evaluation substitutions needs to cover locations and packages as new values to which an evaluation substitution might have been applied. Also, as the representation of identifiers has changed, the rules for identifiers needs to be adapted as well. In addition, elimination needs to reflect the fact that evaluation substitutions now contain sequences of identifier/value pairs instead of a single pair. Fortunately, the rules from Figure 3.3 can be adapted *mutatis mutandis* as shown in Figure 4.8 in the rules from (EvalSubstId) to (EvalSubstLam). Moreover, rule (EvalSubstQuote) covers elimination for quoted s-expressions and rule (EvalSubstPrimApp) primitive applications. Rule (EvalSubstLoc) explains the elimination for locations and drops the substitution. Rule (EvalSubstPac) covers higher-order modules. As an evaluation substitution binds variables and the bindings for the variables of a package are stored in the template of the package, the rule propagates the bindings from the substitution to the template by extending the template of the package with the bindings of the evaluation substitution. Thus it increments the level of the identifiers by one because modules are binding operator and hence the identifiers of the substitution move one level further away from their binding place.

$$y^m_{j\ldots}\langle\!(\langle x^n_{i\ldots}, v_1\rangle \ldots \langle y^m_{j\ldots}, v_2\rangle\langle w^l_{k\ldots}, v_3\rangle \ldots)\!\rangle \rightarrow_{\langle\!(\,)\!\rangle} v_2 \qquad \text{(EvalSubstId)}$$

$$w^l_{k\ldots}\langle\!(\langle x^n_{i\ldots}, v_1\rangle \ldots)\!\rangle \rightarrow_{\langle\!(\,)\!\rangle} w^l_{k\ldots} \text{ iff } w^l_{k\ldots} \notin \{x^n_{i\ldots}, \ldots\} \qquad \text{(EvalSubstIdOther)}$$

$$a\langle\!(t)\!\rangle \rightarrow_{\langle\!(\,)\!\rangle} a \qquad \text{(EvalSubstConst)}$$

$$('se)\langle\!(t)\!\rangle \rightarrow_{\langle\!(\,)\!\rangle} ('se) \qquad \text{(EvalSubstQuote)}$$

$$(@\ e_1 \ldots)\langle\!(t)\!\rangle \rightarrow_{\langle\!(\,)\!\rangle} (@\ e_1\langle\!(t)\!\rangle \ldots) \qquad \text{(EvalSubstApp)}$$

$$(@@\ x\ e \ldots)\langle\!(t)\!\rangle \rightarrow_{\langle\!(\,)\!\rangle} (@@\ x\ e\langle\!(t)\!\rangle \ldots) \qquad \text{(EvalSubstPrimApp)}$$

$$(\lambda y.e)\langle\!(\langle x^n_{im}, v\rangle \ldots)\!\rangle \rightarrow_{\langle\!(\,)\!\rangle} (\lambda y.e\langle\!(\langle x^{n+1}_{im}, v\rangle \ldots)\!\rangle) \qquad \text{(EvalSubstLam)}$$

$$loc\langle\!(t)\!\rangle \rightarrow_{\langle\!(\,)\!\rangle} loc \qquad \text{(EvalSubstLoc)}$$

$$({}^{i_e}Pg^{\texttt{open}\ i^*}_{\mu}\ defs\langle\langle x^n_{i\ldots}, v_m\rangle \ldots\rangle)\langle\!(\langle x^n_{im}, v_s\rangle \ldots)\!\rangle \rightarrow_{\langle\!(\,)\!\rangle} ({}^{i_e}Pg^{\texttt{open}\ i^*}_{\mu}\ defs\langle\langle x^n_{i\ldots}, v_m\rangle \ldots \langle x^{n+1}_{im}, v_s\rangle \ldots\rangle) \qquad \text{(EvalSubstPac)}$$

Figure 4.8: Elimination of evaluation substitutions for packages

Section 3.3 placed the elimination of evaluation substitution in elimination contexts $E_{\langle\!(\,)\!\rangle}$. Now elimination must take place under the store. Therefore we use extended elimination contexts $E_{\langle\!(\,)\!\rangle}$' defined as:

$$E_{\langle\!(\,)\!\rangle}' \quad ::= \theta \Vdash E_{\langle\!(\,)\!\rangle}$$

Now all reductions have been defined and we can proceed to define $\mapsto_{\text{eval}}$, the reduction for evaluation of expressions:

$$\mapsto_{\text{eval}} = \mapsto_{\beta^n_v} \cup \rightarrow_{\delta} \cup \rightarrow_{\text{module}} \cup \mapsto_{\langle\!(\,)\!\rangle}$$

Based on this reduction, we can define the equational theory:

Definition 4.1 ($\lambda_v^{n,\textbf{Module}}$). *$=_v^{n,Module}$ is the smallest equivalence relation generated by $\mapsto_{eval}$. If $e_1 =_v^{n,Module} e_2$, we write $\lambda_v^{n,Module} \vdash e_1 = e_2$.*

□

This completes the description of evaluation for programs containing higher-order modules. The following sections add parsing and macro expansion for modules, hence generating identifiers with non-trivial import paths and levels.

4.3 Parsing and Importing for Modules and Interfaces

Program evaluation, as described in the previous section, must be preceded by elaboration, which parses and macro-expands programs. However, program declarations are part of the configuration language, therefore elaboration does not affect them. Instead the semantics of the configuration language in Section 2.8 already demonstrated the translation of program declarations into Program data types consisting of a link environment and a set of modules. In this chapter, we assume that the translation has instead generated the *prog* terms in the abstract syntax from Figure 4.1, which resemble the Program terms. Elaboration must still parse and macro-expand modules, and, as modules depend on interfaces, also process interfaces. This section presents the rules that extend the parser and process interfaces. Macro expansion for modules is the subject of the next section.

In our system, interface declaration can only occur as `define-interface` definitions within the configuration language. However, they contain macro declarations required for the expansion of module bodies. Hence the semantics contains rules to parse interface declarations and describes the generation of a set of transformer bindings containing the macro definitions within the interface. This process also binds the identifiers from interfaces imported by the interface (via the `open` clause) and incorporates the declarations of free identifiers from the interface's `free` clause. Finally, module declarations occur as `define-module` definitions in the configuration language and as `module` expressions that create higher-order modules during evaluation. Both forms refer to interfaces using the interface name. In the first case, the configuration phase resolves the interface names to interfaces as shown in the semantics of the configuration phase in Figure 2.3. For higher-order modules, the elaboration phase should resolve the interface references. For this to work, interfaces must be elaboration-time values just like transformers, and the parser or the macro expander must contain an interface environment that maps from interface names to actual interfaces just like the set of transformer bindings maps keywords to transformers. At the beginning of elaboration, the mapping can either contain all interfaces declared in the configuration phase, or a new clause `(import-interfaces` *interface-name* ...`)` for top-level modules could import only a subset of the interfaces from the configuration phase. In our semantics, however, we omit this additional environment to simplify matters. Instead, we assume that the `open` clauses of interfaces and modules do not contain names of interfaces—which would then need to be resolved by the interface environment—but the interface declarations proper. This simplification is possible because the `open` clause and the export interface clause of a module must not contain meta-variables but only names of interfaces declared in the configuration phase. Hence, macro expansion does not affect these places and it does not matter, whether they contain names of interfaces or the interface declarations themselves.

We now present the concrete syntax for declaring interfaces and modules and explain briefly, how to translate the configuration phase definitions into this syntax. The syntax has the following two forms:

interface ::= `(interface` *x* `(free` *x* ...`)` `(open` *interface* ...`)`
`((define-syntax` *x* `t)` ...`)` `(`*x* ...`))`
module ::= `(module` *x* *interface* `(open` *interface* ...`)`
`((define-syntax` *x* `t)` ...`)` `((define` *x* *e*`)` ...`))`

For the explanation of the *interface* form, consider the following example template:

`(interface` *uid*
`(free` $name_{free}$ ...`)`
`(open` $interface_{open1}$ ...`)`
`((define-syntax` $keyword_1$ $transformer_1$`)` ...`)`
`(`var_1 ...`))`

In this declaration, *uid* is the unique identifier of the interface, followed by $name_{free}$ ..., the list of free identifiers. The list $interface_{open1}$ ... contains the imported interfaces, which are again *interface* forms. The `define-syntax` definitions declare the macros exported by the interface.

Each macro definition consists of a keyword and a transformer. Finally $var_1 \ldots$ lists the exported variables of the interface. The `define-interface` form Section 2.7 directly translates into this form if we use the name of the interface as its unique identifier. As explained above, we assume that each interface name within an `open` clause has been replaced by an accordant `interface` form.

For the explanation of the `module` form, consider the following example template:

(`module` *uid* $interface_{export}$
  (`open` $interface_{open1} \ldots$)
  ((`define-syntax` $keyword_1$ $transformer_1$) ...)
  ((`define` $variable_1$ $expression_1$) ...))

In this declaration, *uid* is the unique identifier of the module, and $interface_{export}$ is the export interface of the module. The list $interface_{open1} \ldots$ contains the imported interfaces, and the `define-syntax` definitions declare the macros of the module. The last form contains the variable definitions of the module, where each definition consists of a variable and an expression. Again the `define-module` form directly translates into this form using the name as the unique identifier. As for the interfaces, we assume that the name of the export interface and the names of the imported interfaces have been replaced by corresponding interface declarations. In addition, we use the `module` form as the concrete syntax for the expression that creates a higher-order module. Macro expansion for these forms is limited: Meta-variables can only occur within transformers and the right-hand sides of variable definitions but not at the place of the exported module or within `open` clauses. This restriction makes it possible for a human reader to see at one glance imported and exported interfaces of a module. Some more experience will show whether this restriction is reasonable or whether meta-variables at arbitrary positions of a module declaration are required. Furthermore the above form precludes macro applications at the top-level for the reasons stated in the paragraphs "`let-syntax` and `define-syntax`" from Section 3.10.

We will now present a parser that translates the interface and module declarations in the concrete syntax defined above into mixture syntax. The next section defines a macro expander to turn the mixture syntax into the abstract syntax from Figure 4.1. This expander works on the right-hand side of the module definitions and covers module declarations as expressions. For this task, the expander depends on the information provided by the imported interfaces of the modules. We include an additional elaboration phase, called *importing*, which propagates the information from the imported interfaces to the module. Importing binds the names of the imported variables in the right-hand sides of the definitions and adds the imported macro definitions and the macro definitions from the export interface to the module. It uses the import path of the identifiers to record the interface that provided the variables and the macros. As interfaces also include `open` clauses to import other interfaces, importing also operates on interfaces and incorporates the information of imported interfaces for them.

Figure 4.9 presents the mixture syntax that represents interfaces and modules. It is tailored to the multi-phase nature of parsing, importing and expanding for interfaces and modules. This is necessary because interfaces nest to an arbitrary depth through the `open` clause and hence a single elaboration rule cannot encompass all necessary steps. Instead the rules first recognize the top-level structure and corresponding contexts place the rules inside the list of imported interfaces and also iterate through this list.

For interfaces, the parser first generates the term $\overline{I}^{\texttt{open}\ se\ldots}_{\texttt{free}\ R(x_{\text{free}\ldots})} i\ sdefs\ (x_{\text{open}}\ldots)$, a member of the domain *UnparsedInterfaces*, where the forms for the imported interfaces are still plain s-expressions. In this form, i is the run-time interface already known from Figure 4.1, the list $(x_{\text{free}}\ldots)$ contains the free identifiers of the interface, *sdefs* comprises the macro definitions and the $(x_{\text{open}}\ldots)$ list denotes the exported variables. Next, the parser turns the first imported interface into an unparsed interface $\overline{i}$ and afterwards into a member of the domain *UnexpandedInterfaces*. Then it continues analogously with the next imported interface and so on. This way, the parser generates the list of import interfaces from the figure: first $\underline{i}\ldots$, the interfaces that have been expanded but not yet imported, then $\overline{i}$, the interface that has just been parsed, and finally $se\ldots$, the unparsed interfaces. Parsed interfaces have the form $\underline{I}^{\texttt{open}\ \underline{i}\ldots\underline{i}\ldots}_{\texttt{free}\ x\ldots} i\ d\ sdefs\ (x\ldots)$. Here i is again

the corresponding run-time interface, and $x\ldots$ is the list of free identifiers. This interface form includes a set of transformer bindings, d, as introduced for the core macro expander in Figure 3.8. In this set of transformer bindings, importing stores the syntax definitions of the interface and also the contents of the sets of definitions of all imported interfaces. The list of imported interfaces now represents which interfaces have already been imported into the set of transformer bindings $(i\ldots)$ and which have not yet been processed $(\underline{i}\ldots)$. The list of transformers *sdefs* and the list of exported variables $(x\ldots)$ are still included. Analogously, for modules an unparsed and an unexpanded version exist. Both come in two variants because for modules elaboration also needs to take the export interface into account. Hence the first variant of the first form, ${}^{\{se|\bar{i}\}}\overline{M}_{\mu}^{\texttt{open}\ se\ldots}$ *sdefs defs*, describes modules where the export interface is subject to parsing, whereas the second variant ${}^{\underline{i}}\overline{M}_{\mu}^{\texttt{open}\ \underline{i}\ldots\bar{i}^{?}\ se\ldots}$ *sdefs defs*, describes modules where parsing proceeds within the list of imported interfaces. The second form includes a set of transformer bindings where importing stores the syntactic definitions of the module, the contents of the sets of definitions of all imported modules, and the set of transformer bindings of the export interface. Once the set of transformer bindings is complete, the macro expander uses it to expand the right-hand sides of the module's definitions *defs*. The definition for modules m is the same as in the abstract syntax from Figure 4.1. This form is the normal form of parsing and macro expansion for modules. Finally the expansion terms receive a new representation for identifiers, ${}^{ks}x_{im}^{n}$, where im denotes the import path of the identifier, and a variant of the shift operator, written $c{\uparrow}_{i}^{n}$, that contains an interface. Like the normal shift operator, this variant reflects the fact that the identifiers in the term c have been moved one step away from their binding place. However, this time the additional binder that separates the identifier and its binding place is not a λ-abstraction or a `letrec-syntax` binder as in Chapter 3 but instead a module. The i parameter denotes the interface from where the identifier has been imported into the module. Consequently, if the variant of the shift operator arrives at an identifier, it extends the import path of the identifier by the interface i because the import path is the mechanism to represent the binding place of an identifier in the presence of modules. The level argument n of the shift operator still serves the same purpose as in the original shift operator: it protects local variables and must be incremented as the operator moves into the scope of a binder. Section 4.4 contains elimination rules for the shift operator that deal with the new variant.

Figure 4.10 contains the rules for parsing of interface and module declarations. The rules extend the $\rightarrow_{\text{parse}}$ parsing reduction from Section 3.7. To support module declarations within expressions (that is higher-order modules), the expansion contexts from Section 3.5 receives a new rule:

$$EP \quad ::= \ldots \mid {}^{i}M_{\mu}^{\texttt{open}\ i\ldots}(\Delta\ x\ e)\ldots(\Delta\ x\ EP)(\Delta\ x\ c)\ldots$$

The rule places the redex in the right-hand side of the first non-expanded definition.

The single rule for parsing an interface, (ParseInterface), generates parsing substitutions to bind the exported variables and keywords in the transformers of the exported macros. The rule also binds the free identifier from the `free` clause within the transformers using a parsing substitution that replaces the name of the identifier `x` by ${}^{\emptyset}x_{I_{\text{free}()}}^{0}$. That is, the import path of the identifier contains the special interface `free-interface` as outlined in Section 4.1. The rule generates the normal interface from the unique identifier provided as first argument and from the list of exported variables provided as last argument. After parsing, rule (FinishParseInterface) turns an unparsed interface into an interface. The rule also ensures the well-formedness of the interface using the predicate *WellFormedInterf* defined in the same figure. The predicate verifies that the names of the free identifiers, the keywords, and the variables are disjoint and that no interface provides one of these names. To collect the names of the imported interfaces, a helper function *export* returns for an interface the names of the exported macros and variables. The rules (ParseInterface) and (FinishParseInterface) use the interface parsing context *IP* to select the next interface declaration to be parsed. The context *IP* is defined as:

$$IP \quad ::= P[IP']$$

Syntactic Domains:

$\tilde{i} \in \mathit{UnparsedInterfaces}$
$\underline{i} \in \mathit{UnexpandedInterfaces}$
$\imath \in \mathit{RunTimeInterfaces}$
$\iota \in \mathit{InterfaceIdentifiers}$
$\overline{m} \in \mathit{UnparsedModules}$
$\underline{m} \in \mathit{UnexpandedModules}$
$m \in \mathit{Modules}$
$\mu \in \mathit{ModuleIdentifiers}$
$\mathit{def} \in \mathit{TopDefinition}$
$\mathit{sdef} \in \mathit{SyntaxDefinition}$
$\mathit{ses} \in \mathit{Lex\text{-}S\text{-}Expressions}$
$c \in \mathit{MixtureTerms}$

Expansion Syntax:

$\overline{i} ::= \overline{I}^{\texttt{open}\ \underline{i}\ldots\bar{\bar{i}}^{?}\ se\ldots}_{\texttt{free}\ Rx\ldots}\ i\ \mathit{sdefs}\ (x\ldots)$
$\underline{i} ::= \underline{I}^{\texttt{open}\ x\ldots}_{\texttt{free}\ i\ldots\underline{i}\ldots}\ i\ d\ \mathit{sdefs}\ (x\ldots)$
$i ::= I_{\iota}(x\ldots)$
$\overline{m} ::= {}^{\{se|\bar{i}\}}\overline{M}^{\texttt{open}\ se\ldots}_{\mu}\ \mathit{sdefs}\ \mathit{defs} \mid {}^{\underline{i}}\overline{M}^{\texttt{open}\ \underline{i}\ldots\bar{\bar{i}}^{?}\ se\ldots}_{\mu}\ \mathit{sdefs}\ \mathit{defs}$
$\underline{m} ::= {}^{\underline{i}}\underline{M}^{\texttt{open}\ \underline{i}\ldots}_{\mu}\ d\ \mathit{sdefs}\ \mathit{defs} \mid {}^{i}\underline{M}^{\texttt{open}\ i\ldots\underline{i}\ldots}_{\mu}\ d\ \mathit{sdefs}\ \mathit{defs}$
$m ::= {}^{i}M^{\texttt{open}\ i\ldots}_{\mu}\mathit{def}\ldots$
$\mathit{def} ::= (\Delta\ x\ c)$
$\mathit{defs} ::= \mathit{def}\ldots$
$\mathit{sdef} ::= (\Delta_{syn}\ x\ \mathit{ses})$
$\mathit{sdefs} ::= \mathit{sdef}\ldots$
$c ::= \ldots, {}^{ks}x^{n}_{im} \mid c\uparrow^{n}_{i}$
$im ::= i\ldots$

Figure 4.9: Mixture syntax for parsing and expanding modules and interfaces

$$\underline{d}, k \vdash IP[(\texttt{interface}\ x\ (\texttt{free}\ x_{\mathrm{f}}\ldots)\ (\texttt{open}\ se_{\mathrm{imp}}\ldots)\ ((\texttt{define-syntax}\ x_k\ se)\ldots)\ (x_v\ \ldots))] \rightarrow_{\mathrm{Parse}}$$
(ParseInterface)
$$\underline{d}, k \vdash IP[\overline{I}^{\texttt{open}\ se_{\mathrm{imp}}\cdots}_{\texttt{free}\ Rx_{\mathrm{f}}\ldots}(I_x(x_v\ldots))\ sdefs\ (x_v\ldots)]$$
$$\text{where } sdefs = (\Delta_{syn}\ x_k\ se(\!|^{\emptyset}x_k{}^{0}/\texttt{x}_k|\!)\ldots(\!|^{\emptyset}x_v{}^{0}/\texttt{x}_v|\!)\ldots(\!|^{\emptyset}x_{\mathrm{f}}{}^{0}_{(I_{\mathrm{free}}())}/\texttt{x}_{\mathrm{free}}|\!)\ldots)\ldots$$

$$\underline{d}, k \vdash IP[\overline{I}^{\texttt{open}\ \underline{i}\cdots}_{\texttt{free}\ Rx_{\mathrm{f}}\ldots} i\ sdefs\ (x\ldots)] \rightarrow_{\mathrm{Parse}} \underline{d}, k \vdash IP[\underline{I}^{\texttt{open}\ \underline{i}\cdots}_{\texttt{free}\ x_{\mathrm{f}}\ldots} i\ \epsilon\ sdefs\ (x\ldots)] \quad \text{(FinishParseInterface)}$$
$$\text{iff } WellFormedInterf(\overline{I}^{\texttt{open}\ \underline{i}\ldots}_{\texttt{free}\ Rx_{\mathrm{f}}\ldots} i\ sdefs\ (x\ldots))$$

$$WellFormedInterf : \overline{i} \rightarrow Boolean$$
$$WellFormedInterf(\overline{I}^{\texttt{open}\ \underline{i}_{\mathrm{i}}\cdots}_{\texttt{free}\ Rx_{\mathrm{f}}\ldots} i\ ((\Delta_{syn}\ x_k\ ses)\ldots)\ (x_v\ldots)) =$$
$$\forall x, y \text{ in } (x_{\mathrm{f}}, \ldots, x_k, \ldots, x_{k_e}, \ldots, x_v, \ldots) : x \neq y$$
$$\text{and } \bigcup_{\underline{i} \in \{\underline{i}_{\mathrm{i}}\cdots\}} exports(\underline{i}) \cap (\{x_k, \ldots\} \cup \{x_v, \ldots\} \cup \{x_{\mathrm{f}}, \ldots\}) = \emptyset$$

$$exports : \underline{i} \rightarrow Vars$$
$$exports(\overline{I}^{\texttt{open}\ \underline{i}_{\mathrm{i}}\cdots}_{\texttt{free}\ Rx_{\mathrm{f}}\ldots} i\ ((\Delta_{syn}\ x_k\ ses)\ldots)\ (x_v\ldots)) = \{x_k, \ldots, x_v, \ldots\}$$

$$\underline{d}, k \vdash P[(\texttt{x}\ \mu\ se_{\mathrm{exp}}(\texttt{open}\ se_{\mathrm{imp}}\ldots)\ ((\texttt{define-syntax}\ x_k\ se_k)\ldots)\ ((\texttt{define}\ x_v\ se_v)\ldots))] \rightarrow_{\mathrm{Parse}}$$
(ParseModule)
$$\underline{d}, k \vdash P[^{se_{\mathrm{exp}}}\overline{M}^{\texttt{open}\ se_{\mathrm{imp}}\cdots}_{\mu}\ sdefs\ defs] \text{ iff } P[\texttt{x}] \mapsto^{*}_{\mathrm{El}}{}^{ks}\texttt{module}^{n}_{i\ldots\mathrm{scheme}}$$
$$\text{where } sdefs = (\Delta_{syn}\ x_k\ se_k(\!|^{\emptyset}x_k{}^{0}/\texttt{x}_k|\!)\ldots(\!|^{\emptyset}x_v{}^{0}/\texttt{x}_v|\!)\ldots)\ldots \text{ and } defs = (\Delta\ x_v\ se_v(\!|^{\emptyset}x_k{}^{0}/\texttt{x}_k|\!)\ldots(\!|^{\emptyset}x_v{}^{0}/\texttt{x}_v|\!)\ldots)\ldots$$

$$\underline{d}, k \vdash P[^{\underline{i}_{\mathrm{e}}}\overline{M}^{\texttt{open}\ \underline{i}\cdots}_{\mu}\ sdefs\ defs] \rightarrow_{\mathrm{Parse}} \underline{d}, k \vdash P[^{\underline{i}_{\mathrm{e}}}\underline{M}^{\texttt{open}\ \underline{i}\cdots}_{\mu}\ \epsilon\ sdefs\ defs] \quad \text{(FinishParseModule)}$$
$$\text{iff } WellFormedModule(^{\underline{i}_{\mathrm{e}}}\overline{M}^{\texttt{open}\ \underline{i}\cdots}_{\mu}\ sdefs\ defs)$$

$$WellFormedModule : \overline{m} \rightarrow Boolean$$
$$WellFormedModule(\overline{I}^{\texttt{open}\ \underline{i}_{\mathrm{i}}\cdots}_{\texttt{free}\ Rx_{\mathrm{f}}\ldots} i\ ((\Delta_{syn}\ x_{k_e}\ ses)\ldots)\ (x_{v_e}\ldots)\overline{M}^{\texttt{open}\ \underline{i}\cdots}_{\mu}\ ((\Delta_{syn}\ x_k\ ses)\ldots)\ ((\Delta\ x_v\ ses)\ldots)) =$$
$$\forall x, y \text{ in } \{x_{\mathrm{f}}, \ldots, x_k, \ldots, x_{k_e}, \ldots, x_v, \ldots\} : x \neq y$$
$$\text{and } \bigcup_{\underline{i} \in \{\underline{i}_{\mathrm{i}}\cdots\}} exports(\underline{i}) \cap (\{x_k, \ldots\} \cup \{x_v, \ldots\} \cup \{x_{\mathrm{f}}, \ldots\}) = \emptyset$$
$$\text{and } \{x_{v_e}, \ldots\} \subseteq \{x_v, \ldots\}$$
$$\text{and } \{\underline{i}_{\mathrm{i}}, \ldots\} \subseteq \{\underline{i}, \ldots\}$$

Figure 4.10: Parsing for interfaces and modules

$$IP' \quad ::= [\,] \mid ({}^{IP'}\overline{M}_{\mu}^{\texttt{open}\ (se\ldots)}\ sdefs\ defs) \mid ({}^{\underline{i}}\overline{M}_{\mu}^{\texttt{open}\ (\underline{i}\ldots IP'\ se\ldots)}\ sdefs\ defs) \mid \overline{I}^{\texttt{open}\ (\underline{i}\ldots IP'\ se\ldots)}_{\texttt{free}\ Rx\ldots} i\ sdefs\ (x\ldots)$$

It uses the parsing context from Section 3.7 to place the hole under the expansion substitutions. For parsing within modules, the context *IP'* first selects the export interface and afterwards select the left-most unparsed interface within the imported interfaces of the module. For interfaces, it selects the left-most unparsed interface within the list of imported interfaces.

Figure 4.10 also contains rule (ParseModule), which describes parsing for module declarations. The rule uses the parsing context to resolve the first element of the s-expression to an identifier and requires that the name of the identifier is `module` and that its import path ends with the interface `scheme-interface`. This ensures that the identifier is indeed the `module` keyword. The rule then binds the keywords of the macro definitions and the variables of the definitions in the transformers and the right-hand sides of the definitions using parsing substitutions. After parsing, rule (FinishParseModule) turns an unparsed module into an unexpanded module. It applies the predicate *WellFormedModule* to the module to ensure that the module is well-formed. For modules, well-formedness says that the variables and keywords must not contain duplicates and that the module must not define an identifier listed as free in the export interface of the module. Furthermore, all exported variables must be defined in the module, and module must import all the imported interfaces of the export interface.

After parsing of the concrete syntax, the importing phase handles the `open` clauses, which contain the imported interfaces of interfaces and modules. An interface `i1` imports another interface `i2` to bind identifiers occurring in the output of macros defined within `i1` to the interface `i2`. As an example for an interface importing another interface, consider the interface `insert-insert-x-interface` in the following program written in the language from Chapter 2:

```
(define-interface just-x-interface
   (export x))

(define-module just-x-module just-x-interface
   (open scheme-interface)
   (begin
      (define x 23)))

(define-interface insert-x-interface
   (export
      (define-syntax insert-x
          (syntax-rules ()
             ((insert-x) x)))))

(define-module insert-x insert-x-interface
  (open scheme-interface)
  (begin
    (define x 5)))

(define-interface insert-insert-x-interface
  (export
    (open insert-x-interface)
    (define-syntax insert-insert-x
      (syntax-rules ()
        ((insert-insert-x) (insert-x))))))

(define-module insert-insert-x insert-insert-x-interface
  (open scheme-interface
        insert-x-interface)
```

```
  (begin
    (define-syntax insert-insert-x
         (syntax-rules ()
            ((insert-insert-x) (insert-x))))
    (define x 67)
    (define w (insert-x))))

(define-module main no-exports-interface
   (open scheme-interface
         just-x-interface
         insert-insert-x-inteface)
   (begin
      (define y x)
      (define z (insert-insert-x))))

(define-program p
   (modules insert-x just-x-module insert-insert-x main))
```

Here, the right-hand side of the definition of `w` in module `insert-insert-x` will refer to the definition of `x` in module `insert-x`. The import path is `insert-x-interface`. However, the right-hand side of the definition of `z` in module `main` also refers to the binding of `x` in the module `insert-x`. The way to reach this binding starting at module `insert-insert-x`, is first the interface `insert-insert-x-interface`, then the interface `insert-x-interface`. Importing must therefore propagate the binding informations of the imported interface to the transformer expressions of the importing interface. In addition, the macro definitions from the imported interface extend the set of transformer bindings of the importing interface. Figure 4.11 contains the rules that make up the reduction $\rightarrow_{\text{Import}}$ for describing the importing of interfaces into interfaces. Like parsing and expansion, importing takes place within expansion contexts. But as it does not rely on the set of transformer bindings, its standard reduction $\mapsto_{\text{Import}}$ wraps the context EP in $d, k \vdash$ terms:

$$c \mapsto_{\text{Import}} c' \text{ iff } c = \underline{d}, k \vdash EP[c_1], c' = \underline{d}, k \vdash EP[c_2], c_1 \rightarrow_{\text{Import}} c_2$$

The rule (InterfImportVariable) binds an exported variable within the transformer. It also generates a parsing substitution and uses the interface as the import path of the identifier. The next rule, (InterfImportMacro), imports a macro definition. The rule (InterfBindDefSyn) (described below in Section 4.4) has already parsed the macro definitions of the imported interface and stored them in the set of transformer bindings. Rule (InterfImportMacro) generates a parsing substitution that binds the keyword in the transformer and propagates one macro definition from the set of transformer bindings of the imported interface to the set of transformer bindings of the importing interface. Analogously to the previous rule, the import path of the imported identifier is extended by the imported interface both in the parsing substitution and in the set of transformer bindings. In addition, the rule applies the new variant of the shift operator to the imported transformer because the identifiers referenced by the transformer move "one interface" further away from their binding place as importing propagates the transformer. The interface argument of the shift operator is the imported interface. The next section explains the elimination of the new variant of the shift operator. Once the three rules above have imported all information, the rule (FinishInterfImport) turns the importing interface into a normal interface. At the end of importing, an interface has the form $\underline{I}^{\texttt{open}\ (i\ldots)}_{\texttt{free}\ (x\ldots)}\ i\ d\ \mathit{sdefs}\ (x\ldots)$, that is all imported interfaces are run-time interfaces.

For modules, Figure 4.12 contains the rules for importing interfaces. Rule (ModuleImportExport) imports one macro definition contained in the export interface of the module into the module's set of transformer bindings and binds the keyword in the transformers and the right-hand sides of the definitions. No shifting or modification of the keyword's import path is necessary because macro definitions in the export interface extend the macro definitions within the module. Next, rule (FinishModuleImportExport) replaces the export interface by a run-time interface if its set of transformer bindings is empty (ϵ). The remaining rules propagate information from

$$I[\underline{I}^{\texttt{open } i_1 \ldots \underline{I}^{\texttt{open } i_{1_1} \cdots}_{\texttt{free } x_{\mathrm{f}} \ldots} i_{\mathrm{i}}\ d\ ()\ (x x_e \ldots) \underline{i}_2 \cdots}_{\texttt{free } x_{\mathrm{f}_2} \ldots} i\ d_2\ ((\Delta_{syn}\ x_m\ ses) \ldots)\ (x_{e2} \ldots)] \rightarrow_{\text{Import}}$$

(InterfImportVariable)

$$I[\underline{I}^{\texttt{open } i_1 \ldots \underline{I}^{\texttt{open } i_{1_1} \cdots}_{\texttt{free } x_{\mathrm{f}} \ldots} i_{\mathrm{i}}\ d\ ()\ (x_e \ldots) \underline{i}_2 \cdots}_{\texttt{free } x_{\mathrm{f}_2} \ldots} i\ d_2\ ((\Delta_{syn}\ x_m\ ses(\!|^{\emptyset} x^0_{i_i}/\mathtt{x}|\!)) \ldots)\ (x_{e2} \ldots)]$$

$$I[\underline{I}^{\texttt{open } i_1 \ldots \underline{I}^{\texttt{open } i_{1_1} \cdots}_{\texttt{free } x_{\mathrm{f}} \ldots} i_{\mathrm{i}}\ (({}^{ks} x^n_{i_x \ldots} \mapsto tf) :: d_1)\ ()\ () \underline{i}_2 \cdots}_{\texttt{free } x_{\mathrm{f}_2} \ldots} i\ d_2\ ((\Delta_{syn}\ x_m\ ses) \ldots)\ (x_{e2} \ldots)] \rightarrow_{\text{Import}}$$

(InterfImportMacro)

$$I[\underline{I}^{\texttt{open } i_1 \ldots \underline{I}^{\texttt{open } i_{1_1} \cdots}_{\texttt{free } x_{\mathrm{f}} \ldots} i_{\mathrm{i}}\ d_1\ ()\ () \underline{i}_2 \cdots}_{\texttt{free } x_{\mathrm{f}_2} \ldots} i\ (({}^{ks} x^0_{i_{\mathrm{i}} i_x \ldots} \mapsto (tf\ \uparrow^0_{i_{\mathrm{i}}})) :: d_2)\ (\Delta_{syn}\ x_m\ ses(\!|^{ks} x^0_{i_{\mathrm{i}} i_x \ldots}/\mathtt{x}|\!)) \ldots\ (x_{e2} \ldots)]$$

$$I[\underline{I}^{\texttt{open } i_1 \ldots \underline{I}^{\texttt{open } i_{1_1} \cdots}_{\texttt{free } x_{\mathrm{f}} \ldots} i_{\mathrm{i}} \in ()\ () \underline{i}_2 \cdots}_{\texttt{free } x_{\mathrm{f}_2} \ldots} i\ d\ sdefs\ (x_{e2} \ldots)] \rightarrow_{\text{Import}} \quad \text{(FinishInterfImport)}$$

$$I[\underline{I}^{\texttt{open } i_1 \ldots i_{\mathrm{i}} \underline{i}_2 \cdots}_{\texttt{free } x_{\mathrm{f}_2} \ldots} i\ d\ sdefs\ (x_{e2} \ldots)]$$

Figure 4.11: Importing for interfaces

imported interfaces to the module. They are very similar to the rules that handle importing for interfaces from Figure 4.11. The only difference is that modules contain definitions and the rules bind the imported identifiers within the right-hand sides of these definitions, too. At the end of importing, a module has the form $^{i \cdots} \underline{M}^{\texttt{open } d}_{\mu}\ sdefs\ defs$, that is, all imported interfaces have been replaced by run-time interfaces.

4.4 Macro Expansion for Interfaces and Modules

This section extends the macro expander and the elimination reductions for the expansion substitutions by rules that handle modules. The importing phase from the previous section has already propagated the information from the imported interfaces to the module's set of transformer bindings and bound the imported identifiers within the transformers and the definitions. In this section, rules that extend the core macro expander from Figure 3.9 use a module's set of transformer bindings to expand the right-hand sides of the definitions and turn them into expressions.

Before the expansion of the module body, the core macro expander needs to record the macro definitions in the module's set of transformer bindings. For interfaces, it is analogously necessary to record the macro definitions in the interface's set of transformer bindings. Figure 4.13 contains the rule (InterfBindDefSyn) that adds a macro in the interface to the set of transformer bindings. This rule is analogous to the expansion of `letrec-syntax` in rule (ExpandLetSyn) from Figure 3.9: The reduction $\rightarrow_{\mathrm{PT}}$ parses the lexical s-expression representing the transformer, and afterwards (ExpandLetSyn) adds a mapping from the keyword to the transformer to the set of definition. However, there is a major difference between the rules (InterfBindDefSyn) and (ExpandLetSyn) concerning the propagation of the lexical information at the macro definition into the transformer. The recording of lexical information at the macro definition is one of the main characteristics of hygienic macro expansion. For `letrec-syntax`, the elimination rules propagate the expansion substitutions surrounding the macro definition to the transformer. For example, rule (ShiftLetSyn) from Figure 3.11 moves the shift operator to the transformer of a *letrec-syn* form. Such expansion substitutions do not exist for transformers defined within interfaces because an interface is only a specification of a module implementation occurring within a certain lexical context. Instead, the

$$\underline{I}^{\texttt{open}\ i^*}_{\texttt{free}\ x_{\texttt{f}}\ldots}\ i\ ((^{ks}x^n_{im} \mapsto tf) :: d_i)\ ()\ (x_e \ldots)\ \underline{M}^{\texttt{open}\ \underline{i}\ldots}_{\mu}\ d_m\ ((\Delta_{syn}\ x_m\ ses)\ldots)\ ((\Delta\ x_{\text{def}}\ ses_{def})\ldots) \rightarrow_{\text{Import}}$$
(ModuleImportExport)

$$\underline{I}^{\texttt{open}\ i^*}_{\texttt{free}\ x_{\texttt{f}}\ldots}\ i\ d_i\ ()\ (x_e \ldots)\ \underline{M}^{\texttt{open}\ \underline{i}\ldots}_{\mu}\ ((^{ks}x^n_{im} \mapsto tf) :: d_m)\ (\Delta_{syn}\ x_m\ ses(\!|{}^{ks}x^n_{im}/\mathtt{x}|\!))\ldots\ (\Delta\ x_{\text{def}}\ ses_{def}(\!|{}^{ks}x^n_{im}/\mathtt{x}|\!))\ldots$$

$$\underline{I}^{\texttt{open}\ i^*}_{\texttt{free}\ x_{\texttt{f}}\ldots}\ i\ \epsilon\ ()\ (x\ldots)\ \underline{M}^{\texttt{open}\ \underline{i}\ldots}_{\mu}\ d_m\ sdefs\ defs \rightarrow_{\text{Import}}$$
(FinishModuleImportExport)

$${}^{i}\underline{M}^{\texttt{open}\ \underline{i}\ldots}_{\mu}\ d_m\ sdefs\ defs$$

$${}^{i_{\text{i}}}\underline{M}^{\texttt{open}\ i_{\text{i}}\ldots\underline{I}^{\texttt{open}\ i_1\cdots}_{\texttt{free}\ x_{\texttt{f}}\ldots}\ i_{\text{i}_2}\ d\ ()\ (xy\ldots)\underline{i}_2\cdots}_{\mu}\ d_m\ ((\Delta_{syn}\ x_m\ ses)\ldots)\ ((\Delta\ x_{\text{def}}\ ses_{def})\ldots) \rightarrow_{\text{Import}}$$
(ModuleImportVariable)

$${}^{i_{\text{i}}}\underline{M}^{\texttt{open}\ i_{\text{i}}\ldots\underline{I}^{\texttt{open}\ i_1\cdots}_{\texttt{free}\ x_{\texttt{f}}\ldots}\ i_{\text{i}_2}\ d\ ()\ (y\ldots)\underline{i}_2\cdots}_{\mu}\ d_m\ (\Delta_{syn}\ x_m\ ses(\!|t|\!))\ldots\ (\Delta\ x_{\text{def}}\ ses_{def}(\!|t|\!))\ldots$$
where $t = {}^{\emptyset}x^0_{i_{\text{i}_2}}/\mathtt{x}$

$${}^{i_{\text{i}}}\underline{M}^{\texttt{open}\ i_{\text{i}}\ldots\underline{I}^{\texttt{open}\ i_1\cdots}_{\texttt{free}\ x_{\texttt{f}}\ldots}\ i_{\text{i}_2}\ ((^{ks}x^n_{i_x\ldots} \mapsto tf) :: d_i)\ ()\ ()\underline{i}_2\cdots}_{\mu}\ d_m\ ((\Delta_{syn}\ x_m\ ses)\ldots)\ ((\Delta\ x_{\text{def}}\ ses_{def})\ldots) \rightarrow_{\text{Import}}$$
(ModuleImportMacro)

$${}^{i_{\text{i}}}\underline{M}^{\texttt{open}\ i_{\text{i}}\ldots\underline{I}^{\texttt{open}\ i_1\cdots}_{\texttt{free}\ x_{\texttt{f}}\ldots}\ i_{\text{i}_2}\ d_i\ ()\ ()\underline{i}_2\cdots}_{\mu}\ ((^{ks}x^0_{i_{\text{i}_2}i_x\ldots} \mapsto (tf\ \uparrow^0_{i_{\text{i}_2}})) :: d_m)\ (\Delta_{syn}\ x_m\ ses(\!|t|\!))\ldots\ (\Delta\ x_{\text{def}}\ ses_{def}(\!|t|\!))\ldots$$
where $t = {}^{\emptyset}x^0_{i_{\text{i}_2}i_x\ldots}/\mathtt{x}$

$${}^{i_{\text{i}}}\underline{M}^{\texttt{open}\ i_{\text{i}}\ldots\underline{I}^{\texttt{open}\ i_1\cdots}_{\texttt{free}\ x_{\texttt{f}}\ldots}\ i_{\text{i}_2}\ \epsilon\ ()\ ()\underline{i}_2\cdots}_{\mu}\ d_m\ sdefs\ defs \rightarrow_{\text{Import}}$$
(FinishModuleImport)

$${}^{i_{\text{i}}}\underline{M}^{\texttt{open}\ i_{\text{i}}\ldots i_{\text{i}_2}\underline{i}_2\cdots}_{\mu}\ d_m\ sdefs\ defs$$

Figure 4.12: Importing for modules

imported interfaces and the free identifiers of an interface provide lexical information for some of the identifiers referenced by the macro. All other identifiers are assumed to be variables bound by the module. Section 2.2 calls this assumption "defaults to provider." The importing phase already has propagated the lexical information from the imported interfaces to the transformers. Now the rule (InterfBindDefSyn) also records the identifiers of the `free` clause as unbound by generating parsing substitutions that map the symbols from the `free` clause to unbound identifiers. As outlined in Section 4.1, it places the special `free-interface`, here written as $I_{\text{free}}()$, in the import path of the identifiers. All symbols that have been bound neither by the imported interfaces nor by the parsing substitutions for the unbound identifiers have to be replaced by identifiers that refer to the module that implements the interface. The rule achieves this using a $\star$-substitution that maps all symbols to identifiers with level 0. The left-hand side of the rule requires the imported interfaces to be normal interfaces. This ensures that the $\rightarrow_{\text{import}}$ reduction from Figure 4.11 has already processed all imported interfaces. In addition to the well-known parsing context P, the rule uses the context I to place the redex within the import clauses of interfaces and modules and within the export interface of modules:

$$\begin{array}{lll} I & ::= & [\,] \mid ({}^{I}\underline{M}_{\mu}^{\texttt{open}\ (\underline{i}\ldots)}\ d\ sdefs\ defs) \mid ({}^{i}\underline{M}_{\mu}^{\texttt{open}\ (i\ldots I\ \underline{i}\ldots)}\ d\ sdefs\ defs) \\ & & \mid \underline{I}_{\texttt{free}\ (x\ldots)}^{\texttt{open}\ (i\ldots I\ \underline{i}\ldots)}\, i\ d\ sdefs\ (x\ldots) \end{array}$$

$$\underline{d}, k \vdash P[I[\underline{I}_{\texttt{free}\ x_{\text{f}\ldots}}^{\texttt{open}\ i_1\cdots} i\ d\ ((\Delta_{syn}\ x\ ses) sdef_2 \ldots)\ (x_e \ldots)]] \rightarrow_{\text{Expand}} \quad \text{(InterfBindDefSyn)}$$
$$\underline{d}, k \vdash P[I[\underline{I}_{\texttt{free}\ x_{\text{f}\ldots}}^{\texttt{open}\ i_1\cdots} i\ ((^{\emptyset}x^0 \mapsto tf) :: d)\ (sdef_2 \ldots)\ (x_e \ldots)]]$$
$$\text{where } ses(\!|{}^{\emptyset}x^0_{(I_{\text{free}}())}/\texttt{x}_{\texttt{free}}|\!) \ldots (\!|{}^{\emptyset}\star^0_{()}/\star|\!) \mapsto^*_{\text{PT}} tf$$

Figure 4.13: Expansion for interfaces

Expansion for modules is next. Figure 4.14 contains the respective rules. First, rule (Mod-

$$\underline{d}, k \vdash {}^{i_{\text{e}}}\underline{M}_{\mu}^{\texttt{open}\ i_{\text{i}}\cdots}\ d_m\ ((\Delta_{syn}\ x\ ses) sdef_2 \ldots)\ (def \ldots) \rightarrow_{\text{Expand}} \quad \text{(ModuleBindDefSyn)}$$
$$\underline{d}, k \vdash {}^{i_{\text{e}}}\underline{M}_{\mu}^{\texttt{open}\ i_{\text{i}}\cdots}\ ((^{\emptyset}x^0 \mapsto tf) :: d_m)\ (sdef_2 \ldots)\ (def \ldots)$$
$$\text{where } ses \mapsto^*_{\text{PT}} tf$$
$$((^{ks}x^n_{im} \mapsto \underline{tf}) :: \underline{d}), k \vdash {}^{i_{\text{e}}}\underline{M}_{\mu}^{\texttt{open}\ i_{\text{i}}\cdots}\ d_m\ ()\ (def \ldots) \rightarrow_{\text{Expand}} \quad \text{(ModuleExpand)}$$
$$\underline{d}, k \vdash {}^{i_{\text{e}}}\underline{M}_{\mu}^{\texttt{open}\ i_{\text{i}}\cdots}\ ((^{ks}x^{n+1}_{im} \mapsto (\underline{tf} \uparrow^0_{()})) :: d_m)\ ()\ (def \ldots)$$
$$\epsilon, k \vdash {}^{i_{\text{e}}}\underline{M}_{\mu}^{\texttt{open}\ i_{\text{i}}\cdots}\ d_m\ ()\ ((\Delta\ x\ ses)\ldots) \rightarrow_{\text{Expand}} \quad \text{(ModuleExpandDefinitions)}$$
$${}^{i_{\text{e}}}M_{\mu}^{\texttt{open}\ i_{\text{i}}\cdots}(\Delta\ x\ d_m, k \vdash ses)\ldots$$

Figure 4.14: Expansion for modules

uleBindDefSyn) adds a macro definition to the module's set of transformer bindings. Like the corresponding rule for interfaces, (InterfBindDefSyn), it uses the reduction $\rightarrow_{\text{PT}}$ to parse the transformer. However, for a module there is no need to add the $\star$-substitution to bind unbound identifiers because the parser already adds this substitution to the program at the beginning of parsing. For a module, there is also no need to implement the "defaults to provider" strategy or propagate information about unbound identifiers because, unlike an interface, a module is an expression that occurs within a certain lexical context. The task of the "defaults to provider" strategy and of the `free` clause is to provide this information explicitly for an interface, which represents a set of unknown modules. Next, once the module's set of macro definitions is empty, rule (ModuleExpand) adds one definition of the expander's set of transformer bindings into the

module. Thus the definition enters the scope of the module and hence the rule increments the level of the keyword by one and applies the shift operator to the transformer. Once the expander's set of transformer bindings is empty (ϵ), rule (ModuleExpandDefinitions) propagates the module's set of transformer bindings to the right-hand side of the definitions and turns the expanding module into a module. Parsing and expanding these right-hand sides happens through the $\rightarrow_{\text{parse}}$ and $\rightarrow_{\text{expand}}$ reductions because the expansion context will place these reductions within the first non-expanded right-hand side.

To fully support higher-order modules, the elimination of the expansion substitutions must cover modules as well. In addition, the elimination of the $\uparrow^n_{im}$ variant of the shift operator, which adds an interface to the import path of an identifier instead of incrementing the level, needs to be explained. The rules (InterfImportMacro) and (ModuleImportMacro) of the $\rightarrow_{\text{Import}}$ reduction from Figure 4.12 apply this shift operator to an imported transformer to reflect the fact that the transformer just entered the scope of the module via the interface given as argument to the operator. The operator must then add this interface to the import path of an identifier. The two rules (ShiftIdInterf) and (ShiftLocalIdInterf) from Figure 4.15 show the elimination of variant of the shift operator applied to an identifier. As in the case of the normal shift operator (rules (ShiftId) and (ShiftLocalId) from Figure 3.11), the level argument of the shift operator determines whether the operator applies to the identifier or whether the identifier is local with respect to the place where the operator has been introduced. Hence rule (ShiftIdInterf) adds the interface i_2 to the import path of the identifier if the level of the operator (n) is equal or bigger than the level of the identifier (m), and rule (ShiftLocalIdInterf) drops the operator otherwise. The elimination

$$({}^{ks}x^n_{i_{1_1}\ldots}\uparrow^m_{i_2}) \rightarrow_{\uparrow_{\text{PatO}}} {}^{ks}x^n_{i_2 i_{1_1}\ldots} \text{ iff } n \geq m \qquad \text{(ShiftIdInterf)}$$

$$({}^{ks}x^n_{i_{1_1}\ldots}\uparrow^m_{i_2}) \rightarrow_{\uparrow_{\text{PatO}}} {}^{ks}x^n_{i_{1_1}\ldots} \text{ iff } n < m \qquad \text{(ShiftLocalIdInterf)}$$

Figure 4.15: Elimination of the interface variant of the shift operator

of the variant of the shift operator for terms other than identifiers is identical to the normal shift operator. This is because elimination either increments the level of the operator whenever it enters the scope of a binding construct or the operator does not apply to the term at all and can be dropped. Therefore, we do not add another reduction for the elimination of the variant of the shift operator but consider the import path as an optional argument of the operator and specify separate rules for the elimination of the operator applied to identifiers only. These rules are then (ShiftId) and (ShiftLocalId) from Figure 3.11, and (ShiftIdInterf) and (ShiftLocalIdInterf) from 4.15. For modules, a new elimination rule (ShiftModule) defined in Figure 4.16 explains the shift operator applied to modules. As usual, if the shift operator enters the scope of a binding construct, the level of the operator needs to be incremented to prevent the bound variables from being affected by the operator. The rule then applies the operator to the transformers and the variable definition of the module. In the rule we use the notation $\uparrow^n_{i^?}$ to display that the rule concerns both variants of the shift operator.

$$({}^{i_e}\underline{M}^{\texttt{open}\ i_{i\cdots}}_{\mu}\ d\ ((\Delta_{syn}\ x_{s_1}\ ses_{s_1})\ldots)\ ((\Delta\ x\ ses)\ldots)\uparrow^n_{i^?}) \rightarrow_{\uparrow} \qquad \text{(ShiftModule)}$$
$${}^{i_e}\underline{M}^{\texttt{open}\ i_{i\cdots}}_{\mu}\ d\ (\Delta_{syn}\ x_{s_1}\ (ses_{s_1}\uparrow^{1+n}_{i^?}))\ldots\ (\Delta\ x\ (ses\uparrow^{1+n}_{i^?}))\ldots$$

Figure 4.16: Reduction `shift-expr` for modules

We now turn to the elimination rules for the remaining expansion operators applied to modules. For the parsing substitution, Figure 4.17 contains the two rules (ParseSubstIdModule) and (ParseSubstMVModule). The difference between the two rules is that the first rule covers the case where an identifier replaces a symbol while in the second rule a meta-variable replaces a symbol.

Accordingly, the first rule increments the level of the replacing identifier while the second rule leaves the substitution intact. Both rules pass the parsing substitution to the transformer of the module's macro definitions and to the right-hand sides of the variable definitions.

$$
\begin{array}{rl}
{}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_i\cdots}\ d\ ((\Delta_{syn}\ x_{t_1}\ ses_1)\ldots)\ ((\Delta\ x\ ses)\ldots)(\!|^{ks} w_{im}^{n}/\mathsf{y}|\!) \rightarrow_{(\!|\,|\!)} & \text{(ParseSubstIdModule)}\\
\qquad {}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_i\cdots}\ d\ (\Delta_{syn}\ x_{t_1}\ ses_1(\!|^{ks} w_{im}^{1+n}/\mathsf{y}|\!))\ldots\ (\Delta\ x\ ses(\!|^{ks} w_{im}^{1+n}/\mathsf{y}|\!))\ldots & \\
{}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_i\cdots}\ d\ ((\Delta_{syn}\ x_{t_1}\ ses_1)\ldots)\ ((\Delta\ x\ ses)\ldots)(\!|\mathsf{a}/\mathsf{y}|\!) \rightarrow_{(\!|\,|\!)} & \text{(ParseSubstMVModule)}\\
\qquad {}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_i\cdots}\ d\ (\Delta_{syn}\ x_{t_1}\ ses_1(\!|\mathsf{a}/\mathsf{y}|\!))\ldots)\ ((\Delta\ x\ ses(\!|\mathsf{a}/\mathsf{y}|\!))\ldots) &
\end{array}
$$

Figure 4.17: Elimination of parsing substitutions for modules

Recognizing a module as a binding construct is also the key idea for the elimination of meta-substitution applied to modules: as Figure 4.18 shows, rule (MetaSubstModule) shifts the the meta-substitution before passing it to transformers and variable definitions.

$$
\begin{array}{rl}
{}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_i\cdots}\ d\ ((\Delta_{syn}\ x_{s_1}\ ses_{s_1})\ldots)\ ((\Delta\ x\ ses)\ldots)\wr s_!\int \rightarrow_{\wr\int} & \text{(MetaSubstModule)}\\
\qquad {}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_i\cdots}\ d\ ((\Delta_{syn}\ x_{s_1}\ ses_{s_1}\wr(s_!\uparrow)\int)\ldots)\ ((\Delta\ x\ ses\wr(s_!\uparrow)\int)\ldots) &
\end{array}
$$

Figure 4.18: Elimination of meta-substitutions for modules

For identifier substitutions, the rules in Figure 4.19 distinguish whether an identifier or a meta-variable replaces the identifier. In the first case, rule (IdSubstIdModule) increments the levels of both identifiers, the replacing and the replaced, to protect variables introduced by the module. For the meta-variable case, rule (IdSubstMVModule) increments only the level of the replaced identifier. As for λ or `letrec-syntax`, the elimination of the meta-substitution replacing the meta-variable will later shift the levels of the replacing code to prevent free variables from being captured by the `module` form.

$$
\begin{array}{rl}
{}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_i\cdots}\ d\ ((\Delta_{syn}\ x_{s_1}\ ses_{s_1})\ldots)\ ((\Delta\ x\ ses)\ldots)[^{ks_3} w_{im_3}^{n_3}/^{ks_2} y_{im_2}^{m}] \rightarrow_{[]} & \text{(IdSubstIdModule)}\\
\qquad {}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_i\cdots}\ d\ (\Delta_{syn}\ x_{s_1}\ ses_{s_1}[^{ks_3} w_{im_3}^{n_3+1}/^{ks_2} y_{im_2}^{m+1}])\ldots\ (\Delta\ x\ ses[^{ks_3} w_{im_3}^{n_3+1}/^{ks_2} y_{im_2}^{m+1}])\ldots & \\
{}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_i\cdots}\ d\ ((\Delta_{syn}\ x_{s_1}\ ses_{s_1})\ldots)\ ((\Delta\ x\ ses)\ldots)[\mathsf{a}/^{ks_2} y_{im_2}^{m}] \rightarrow_{[]} & \text{(IdSubstMVModule)}\\
\qquad {}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_i\cdots}\ d\ (\Delta_{syn}\ x_{s_1}\ ses_{s_1}[\mathsf{a}/^{ks_2} y_{im_2}^{m+1}])\ldots\ (\Delta\ x\ ses[\mathsf{a}/^{ks_2} y_{im_2}^{m+1}])\ldots &
\end{array}
$$

Figure 4.19: Elimination of identifier substitutions for modules

The unshifting operator $\downarrow$ applies to expanded modules only. Figure 4.20 contains the single rule (UnshiftMod) that increments the level of the unshift operator before passing it to the syntactic definitions and the right-hand sides of the variable definitions.

Finally, Figure 4.21 contains the rule (MarkModule) to eliminate the mark operator applied to a module. As a new scope does not affect the mark operator, the rule passes the operator unchanged to the syntax definitions and the variable definitions.

The above rules, which extend the expansion operators of the parser and macro expander, are sufficient to add first-class modules to our hygienic macro expansion system.

Macro expansion is then defined as the union of parsing, importing, expansion, and unshifting:

$$({}^{i_e}M_{\mu}^{\texttt{open}\ i_{i\cdots}}(\Delta\ x\ e)\ldots\downarrow^n) \rightarrow_{\downarrow} {}^{i_e}M_{\mu}^{\texttt{open}\ i_{i\cdots}}(\Delta\ x\ (e\downarrow^{1+n}))\ldots \qquad \text{(UnshiftMod)}$$

Figure 4.20: Elimination of the unshift operator for modules

$$\begin{aligned}{}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_{i\cdots}}\ d\ ((\Delta_{syn}\ x_{s_1}\ ses_{s_1})\ldots)\ ((\Delta\ x\ ses)\ldots)\male^n \rightarrow_{\male} \\ {}^{i_e}\underline{M}_{\mu}^{\texttt{open}\ i_{i\cdots}}\ d\ (\Delta_{syn}\ x_{s_1}\ ses_{s_1}\male^n)\ldots\ (\Delta\ x\ ses\male^n)\ldots\end{aligned} \qquad \text{(MarkModule)}$$

Figure 4.21: Elimination of the mark operator for modules

$$\begin{aligned}&\mathit{expand}: \mathit{S\text{-}Expressions} \rightarrow \mathit{Expressions}\\ &\mathit{expand}(se) = e \text{ iff } se(\!|{}^{\emptyset}\star^0_{I_{\text{free}}()}/\star|\!) \mapsto^*_{\text{ParseImportExpandUnshift}} e\end{aligned}$$

where

$$\mapsto_{\text{ParseImportExpandUnshift}} = \mapsto_{\text{Parse}} \cup \mapsto_{\text{Import}} \cup \mapsto_{\text{Expand}} \cup \mapsto_{\downarrow}$$

This definition initializes the $\star$-substitution with the identifier ${}^{\emptyset}\star^0_{I_{\text{free}}()}$, which has the `free-interface` as its import path. This way, all unbound identifiers receive this interface in its import path. The *free-id=?* function for comparing free identifiers (defined in Figure 3.18) needs to be re-defined to recognize to identifiers as equal if both identifiers contain the `free-interface` in their import path. Otherwise, the identifiers are equal if they have been bound by the same binder, that is, the name, the level, and the import path must be identical:

$$\mathit{free\text{-}id{=}?}({}^{ks}x^n_{im_1}, {}^{ks'}y^m_{im_2}) = \begin{cases} \mathit{true} \text{ if } x = y \wedge I_{\text{free}}() \in im_1 \wedge I_{\text{free}}() \in im_2 \\ \mathit{true} \text{ if } x = y \wedge n = m \wedge im_1 = im_2 \\ \mathit{false} \text{ otherwise} \end{cases}$$

Now the rules (EvalFreeIdEqTrue) and (EvalFreeIdEqFalse) from Figure 3.18 and the function *match* from Figure 3.33 can use this new definition of *free-id=?* to compare identifiers.

4.5 Independent Compilation and Code Sharing

With the module system presented in this section, independent compilation for a dynamically typed language with macros is feasible. Interfaces contain the full static information about imported identifiers: for macros they contain the transformers, for variables they provide the binding place, and they reveal identifiers assumed unbound. The semantics does not include compilation as a separate phase because compilation is hard to capture formally: compilation is mostly an exchange of the code representation that does *not* change the semantics. Therefore the semantics only includes a description of elaboration, which produces the abstract syntax, skips compilation, and continues with the dynamic linking and the evaluation. However, as elaboration of a module takes place independent from any other module, and as elaboration produces the abstract syntax—including the static semantics of all identifiers—of the module, independent compilation of the module is possible as well. Two (or more) modules can also share the abstract syntax if they import the same identifiers.[3] Consequently, they can also share the compiled code. The `copy-module` clause from the configuration language in Section 2.7 generates such identical modules. Section 2.3 uses the term *unit* for the code and the associated static semantics that can be shared by modules with identical imports. In terms of the abstract syntax, the set of transformer

[3] In addition to the imports, the macro definitions in the export interface have to be identical as well.

bindings and the unexpanded right-hand sides of the definitions comprise the uncompiled unit of the module and the expanded right-hand sides of the definitions correspond to the compiled unit. I have omitted units from the formal presentation because they would complicate matters without adding much insight. The semantics could include units by adding them as an additional layer. A module would then consist of its unique identifier and the unique identifier of the unit, and units would replace the current modules within the semantics. Modules can then share a common unit by referring to it using the unit's identifier. A package would result from a module by generating a fresh template as before and inheriting the code from the unit referred to in the module.

4.6 Future Work

Several extension to the system are conceivable: tags and renaming on import have already been included in Kelsey's original proposal [Kel97], mutually recursive modules and interfaces as elaboration time values were briefly sketched in Section 4.2.2, a new definition of hygiene could build on interfaces, and finally languages with a static type system can be supported. This section explains for each extension the required changes to the semantics.

Renaming on Import One of the purposes of a module system is to manage namespaces. Nevertheless, name collisions occur if a module imports interfaces that provide the same name. Renaming identifiers on import addresses the problem: The import clause receives an extension that allows the programmer to specify new names for the variables and keywords provided by the imported interface. Within the body of the module, only the new names are visible as imports of the respective interface. If chosen properly, the new names do not collide with any other of the imported names.

Including renaming into our semantics is quite simple: The rules (ModuleImportVariable) and (ModuleImportMacro), which import identifiers into the module's body, only need to generate parsing substitutions that replace the symbol by the new name instead of the one provided by the interface. The replacing identifier would remain unchanged. The same change is also necessary for the importing rules of interfaces, (InterfImportVariable) and (InterfImportMacro), which bind names in the transformer of an interface.

Tags on Import As interfaces are only placeholders for modules, importing the same interface several times into the same module makes sense in a parameterized module system: During linking different modules can be used to satisfy the imports. To support this scenario, renaming is of course essential as the imports will all provide the same names. However, it is also necessary to distinguish the interfaces during linking and the current mechanism of using the unique identifier of the module is not sufficient if the same module occurs several times within the same import list. Kelsey [Kel97] proposes to add an optional *tag* to the name of the interface within the `open` clause. The tag provides an additional identity that distinguishes identical interface names.

For the semantics to support tags, it would use the tag in addition to the normal interface within the imported interfaces of a module as well as within the import path of an identifiers. A link within the link environment must then also include the tag of the interface to identify the import to be satisfied.

Mutually Recursive Modules Two modules are mutually recursive if both import each other. The semantics does not support mutually recursive modules as the dynamic linking phase can only link in a package that has already been fully linked, which is not possible if the package depends on the package that linking is currently working on. If linking would link in a package that has not yet been fully linked, the template of the importing package would lack identifiers that macros of the imported package may reference. The linking could be repeated after the imported package has been fully linked, but then also all packages that import this package would require re-linking. This process continues ad infinitum. There is no way to stop the process because macros of mutually recursive modules can be defined mutually recursive as they appear in the interface. In this case

however, the macros can generate identifiers with an arbitrarily long import path. Consequently, the way the semantics represents dynamic linking—by extending the template of the importing package with all identifiers of the imported package—is fundamentally incompatible with mutually recursive modules. On the other hand, a real-life implementation could easily work around this problem: the compiler just needs to determine the identifiers referenced in the module body or needed by other packages and add placeholders for these identifiers to the template. Dynamic linking would then merely fill these placeholders instead of augmenting the template with all identifiers from imported packages. In the case of mutually recursive modules dynamic linking is still required to link a package repeatedly but the process finishes once all templates have been fully linked.

Interfaces as Elaboration-Time Values In the configuration language from Section 2.7 interfaces can only be declared at the top level of a program. This precludes macros that expand into interface definitions. I consider this restriction an important feature because a human reader can always see at one glance the exported variables and macros of an interface. Furthermore the semantics for the macro expander in this chapter also precludes meta-variable occurrences within the `open` clause and the export declaration of modules. This restriction enables the human reader to see easily the imported and exported variables and macros for higher-order modules even if the module declaration is generated by a macro. More experience with real life applications will clarify if this restriction is useful or if meta-variables within module declarations are necessary. The semantics would implement the latter case by having interfaces as elaboration-time values just like transformers.

New Definition of Hygiene Using Interfaces Once we allow meta-variables at arbitrary positions within module definitions, it becomes possible to define derived syntax forms for module definitions that abstract over common patterns. For example, a variant of modules that imports other modules instead of interfaces and directly evaluates its body definitions is useful whenever a higher-order module should be evaluated immediately. The following `package` macro defines such a variant. It expects the imported packages along with their export interfaces within an `open` clause:

```
(define-syntax package
  (syntax-rules ()
    ((package exp (open (imp-i imp-p) ...) body-expr)
     (let ((m (module exp (open imp-i ...) (begin body-expr))))
       (begin
         (link! m imp-i imp-p) ...
         (evaluate-package m))))))
```

Unfortunately, hygiene inhibits the bindings of the imported packages to be visible within the body of the module: The `module` form is a binding construct and the First Hygiene Condition prohibits that it captures references from the input. The problem is that the bound identifiers do not appear within the input of the macro. However, the input of the macro includes the interfaces which in turn contain the names of these variables. As interfaces can only be defined at the top level, it makes sense to perceive them as isomorphic to the set of identifiers they export. Hence the following amendment to the Hygiene Condition could be defined:

> **Amendment of Hygiene Condition**
> With regard to hygiene, a binding interface is interchangeable with the set of identifiers the interface exports.

To implement the amendment, the semantics would need to treat meta-variables that are replaced by interfaces as equal to any identifier exported by this interface. The elimination of meta-substitutions would implement in the case of a meta-substitution applied to a module that lists a meta-variable as an imported interface.

Types So far, the static semantics of a variable has only been its binding place. Macro expansion makes it difficult to derive this information. However, the semantics describes how to obtain the binding place for all identifiers within a module. For statically typed languages, the type of an identifier is also part of the identifier's static semantics. Hence the interface is responsible for providing this information and indeed this topic has been treated extensively in the literature [Org96, Ler94, DCH03]. Kelsey [Kel97] already sketched the basic ideas for a fully-parameterized variant of Standard ML, using parameterized signatures [Jon96]. In addition, even a dynamically typed language like Scheme could benefit from types in the interface [GG01].

Chapter 5

Implementation

Semantics is only useful if it is also possible to implement a programming language that matches this semantics. This chapter describes implementations of the module system and the macro expander. The implementations are not yet integrated with an existing Scheme implementation, but rather serve as prototypes and testing platforms.

- Section 5.1 presents a complete implementation of the configuration phase including an implementation of the configuration language from Section 2.7, and an implementation of the automatic linker. In addition, two backends exist that translate the program resulting from the configuration phase into the module languages of Scheme 48 and PLT Scheme. However, as these systems do not handle macro definitions within interfaces, the backends to not support this aspect of the semantics.

- Section 5.2 presents the macro expander and the module system as implementations within PLT redex, a language for writing rewriting systems. This implementation completely implements the semantics of this dissertation.

- Section 5.3 covers a realistic implementation of the macro expander from Chapter 3. The expander produces abstract syntax for Scheme 48 and also contains an implementation of Macro Scheme.

In addition to these implementations of the semantics, Section 5.4 shows how I generated the LaTeX code for the semantics in Chapters 3 and 4 from the implementation within PLT redex.

5.1 Configuration Phase

Kelsey's description of fully-parameterized modules as a set of abstract data types maps directly into a Scheme implementation. The data types become records [Kel99]. The constructors *make-interface*, *make-unit*, *make-module*, and *make-program* expect the unique identifier as an argument instead of generating it themselves as the constructors in Section 2.5.

Our implementation also supports some extensions not described in Section 2.5. Most notably, modules can add a tag to an imported interface and use this tag instead of the interface name to refer to the interface. This extension described by Kelsey [Kel97] makes it possible for a module to import the same interface several times and link it later to different imports. Furthermore, the interface can declare the type of exported identifiers in a syntax similarly to the Scheme 48 module system.

To determine the initial order of a program, we first generate a directed graph with the modules of the program as nodes. There is an edge from node A to node B if the module corresponding to node A imports the module corresponding to node B. Topological sorting determines the initial order of the program. Scheme 48 comes with an implementation for topological sorting which

assumes that it can write intermediate data into the data structure representing the node. Therefore, the record type for modules receives an additional field, which is mutable and initialized to `#f`.

Two prototype backends for the configuration phase exist. These backends take a fully-linked program and translate it into code that an existing Scheme implementation can process. Thus the backends cover elaboration, compilation, and evaluation. So far, the backends do not cover the full power of our fully-parameterized module system with hygienic macros but they serve as a basis for the design and implementation of the system. The next sections present the two backends and along with their limitations.

5.1.1 The Backend for Scheme 48

The Scheme 48 backend generates a set of module declarations for the Scheme 48 module language [Ree94]. Scheme 48 has a small configuration language separate from the Scheme code proper. As in our system, interfaces are separate entities in Scheme 48. An interface consists of a list of names, possibly decorated with type information. Scheme 48 calls modules "structures" because they are fully linked.[1] A structure consists of an export interface, a list of other structures it imports, and a piece of Scheme code. The `define-structure` form defines a structure and assigns a name to it. A structure definition can rename and hide identifiers during import. Structures can also export macros. In this case, the interface must assign type `:syntax` to the corresponding keyword. During compilation of the structure, the macro expander looks up the definition of imported macros in the exporting structures. This means that Scheme 48 does not support independent compilation. Furthermore, its support for parameterized modules is weak: While there exists a configuration language form to declare parameterized modules, the implementation simply expands parameterized modules into structure definitions as soon as the modules have been linked. Consequently, no out code sharing occurs between different instantiations of the same parameterized module.

As parameterized modules in Scheme 48 do not offer code sharing between the instances, the backend cannot translate modules that share the same unit into Scheme 48 module. Instead, it generates for each module a separate `define-structure` definition. In addition, macro definitions within interfaces are not supported because Scheme 48 does not support this feature. Instead, the macro definitions of the exporting module are used.

Apart form that, the Scheme 48 backend is rather powerful: the complete module definitions of the implementation of the fully-parameterized module system are formulated using the configuration language of Section 2.7 and translated with the Scheme 48 backend into the module language.

5.1.2 The PLT Backend

PLT ships with two module systems: The PLT unit system [FF98] is a fully-parameterized module system with first-class modules but without support for macros, whereas the other module system is a conventional static module system with macros and separate compilation [Fla02]. The PLT backend uses a combination of both systems to implement as much of the required semantics as possible. A plain PLT unit imports and exports variables directly but the unit system comes with an extension called *signed units* that covers interfaces (called *signatures*). The declaration of a signed unit imports a set of signatures and exports a signature. Using the `provide-signature-elements` form makes it even possible to use these interfaces to describe the exports of a static module. Consequently, the PLT backend translates interfaces into signatures. The PLT backend generates static PLT modules for each module from the input program and uses the PLT unit system to express parameterization when necessary. To that end, the backend determines for each unit (in the sense of Section 2.3) of the input program whether several modules use it. If not, the backend translates the corresponding module into a static PLT module. Otherwise

[1] See Section 2.3 for an explanation of this difference.

it generates a static PLT module containing in turn a PLT unit for the unit. The PLT unit is parameterized over all interfaces for which program links the modules using the unit to different providers. For all other imports, the backend generates `require` statements in the surrounding module. Next, the backend generates static PLT modules for all modules using the unit. Each such module imports the module containing the PLT unit and the remaining imported modules as specified by the link environment. The body of the module is then the definition of a compound unit which performs the linking. As a compound unit can only link together units, the backend generates a PLT unit for every imported module as well.

The generated code preserves code sharing as it generates only one PLT unit if a set of modules share the same unit. However, the PLT backend does not handle macros for parameterized modules because a unit cannot export or import a macro. Furthermore, it does not provide independent compilation because PLT does not.

5.1.3 Configuration Language

I have implemented the configuration language from Section 2.7 using a set of `syntax-rules` macros. These macros translate the language into applications of the record constructors from Section 5.1, which model the abstract data types.

The names of the definitions are the names from the definitions in the configuration language. This requires the programmer to sort the definitions for interfaces, modules, and programs according to the order the Scheme implementation evaluates top-level definitions as otherwise identifiers may be referenced before their definition. It would be better to wrap the right-hand sides of the definitions into nullary λ-abstractions and evaluate them on demand. In practice, however, its easy to live with this restriction as most Scheme implementation evaluate the top-level definitions from left to right; one simply has to list interfaces before modules and modules before programs.

5.2 Implementation of the Rewriting Systems

The operational semantics for hygienic macro expansion from Chapter 3 is based on context-sensitive term rewriting: A term reduces to its normal form using a set of binary relations and the context of the term determines which relations apply. PLT redex [MFFF04] is a domain-specific language for context sensitive rewriting systems embedded in Scheme. It enables the designer of a term rewriting system to experiment and test with the rules and also visualizes reduction sequences.

The reduction systems for the parser and the macro expander have been implemented with PLT redex. Section 5.4 shows how a separate program generates the LaTeX source code for this dissertation from the same set of rules. The rest of this section describes PLT redex and the implementation of the rewriting systems from this dissertation using PLT redex.

For the definition of a rewriting system in PLT redex, it is first necessary to define the term language using the macro `language`. A language consists of a list of rules for each non-terminal. Each rule is a list containing the non-terminal as the first element followed by the right-hand sides of the production rules for the non-terminal. Contexts are also non-terminals but should include the terminal `hole` once. There is no way to specify the set of terminals for the language in PLT redex. Instead, any s-expression that is not a non-terminal, becomes a terminal. The following code is a shortened and simplified declaration that roughly corresponds to the definition in Figure 3.8. The language definition is stored in a global variable `lang`:

```
(define lang
  (language
```

The non-terminal `a` corresponds to a constant (a) and is defined as a synonym for `number`, a PLT redex built in non-terminal that covers all numbers:

```
(a number)
```

`Se` specifies s-expressions (*se*in Figure 3.8):

```
(se a
    x
    (se ...))
```

`C` is an expansion term equal to c:

```
(c se xn
   (lam (x) c)
   (@ c c ...)
   (mapp xn ses)
   (letrec-syn x tf ses)
   (subst-sym c r)
   (shift-expr c number)
   (mark c number)
   (unshift c number)
   (with-d c d number))
```

The expansion terms include s-expressions and identifiers with levels (`xn`), λ-abstractions, applications, macro applications, `letrec-syntax` forms, three expansion substitutions, the unshift operator and `with-d`, which corresponds to a $d, n \vdash c$ term.

The expressions are the normal forms of expansion, hence the language includes them with a separate non-terminal `e`:

```
(e a
   xn
   (@ e e ...)
   (lam (x) e))
```

Next, `def` is a definition, `defn` is a normalized definition. Both combine a keyword with a transformer but the definition of transformers and normalized transformers is omitted here. The non-terminal `d` describes a set of transformer bindings (d), which is either empty, or a definition in front of another set, or a definition subject to the shift operator. Normalized set of bindings ($\underline{d}$) do not contain the shift operator and only normalized definitions:

```
(def (mk-def xn tf))

(defn (mk-def xn tfn))

(d epsilon
   (d-cons def d)
   (shift-def d))

(dn epsilon
    (d-cons defn dn))
```

`Ses` is a lexical s-expression and `r` is the argument of the parsing substitution:

```
(ses se
     (subst-sym ses r)
     (shift-expr ses number)
     (mark ses number))

(r (mk-subst-sym x xn))
```

`Xn` denotes an identifier with level and a list of marks:

```
(xn (mk-xn x number (number ...)))
```

The non-terminal `x` covers the set of variables. The special form `variable-expect` defines this set to be any symbol not listed in the arguments of the special form. In our case, these arguments cover all symbols we use to construct the terms:

```
(x (variable-except subst-sym shift-expr mark unshift mk-subst-sym
                    mk-xn @ mapp lam letrec-syn with-d))
```

Finally, the definitions of two contexts `el-ctxt` and `e-ctxt` finish the declaration of the language:

```
(el-ctxt hole
         (subst-sym el-ctxt r)
         (shift-expr el-ctxt n)
         (mark el-ctxt n))

(e-ctxt hole
        (unshift e-ctxt n)
        (@ e ... e-ctxt c ...)
        (lam (x) e-ctxt)))
```

`El-ctxt` corresponds to the elimination context *EL*, `e-ctxt` is the expansion/parsing context *EP* as defined on page 43. Within the definition of the contexts, the terminal `hole` corresponds to the hole of the context ("[ ]").

Reductions in PLT redex are lists of rules. A rule defines a single step of the reduction by describing the left-hand and the right-hand side of the rule. PLT redex applies all the rules for which the left-hand side matches the input term and returns a list of results. A following code is a simple example for a rule declaration. In PLT redex, the following declaration defines the rule (ShiftConst) from Figure 3.11, which explains the elimination of the shift operator applied to a constant:

```
(reduction/context "ShiftConst"
                   lang
                   el-ctxt
                   (shift-expr a_ number_)
                   (term a_))
```

The macro `reduction/context` defines a reduction rule that takes place within a context. The first operand `"ShiftConst"` names the rule.[2] The second operand specifies the language on which the rule operates. Next comes `el-ctxt`, the context for the rule, which must be part of the language. The fourth argument, `(shift-expr a_ number_)`, is an s-expression that describes the left-hand side of the reduction as a pattern. Symbols that end with an underscore (here `a_` and `number_`) refer to non-terminals of the language and match the input term if the input matches the non-terminal derived from the symbol name by stripping the underscore. All other symbols and the parentheses match literally. In the example above, the outer pair of parentheses and `shift-expr` matches literally whereas `a_` matches any constant and `number_` any number. The final argument is a Scheme expression that constructs the right-hand side of the reduction. The value of this expression has to be a term of the language. The macro `term` generates such terms from s-expressions. Within the s-expression, symbols that end with an underscore refer to input terms that matched the same symbol in the left-hand side of the rule. All other parts of the s-expression become part of the term. In the example above, the right-hand side is just the symbol `a_`. It contains an underscore and appears in the left-hand side of the rule, hence PLT redex replaces it by the constant in the input term that matched `a_`.

[2] Naming rules is an extension that I have added for the generation of the LaTeX code, PLT redex does not use it but the latest version of PLT redex contains named rules as an extension.

In the following definition of rule (UnshiftApp) from Figure 3.13, the left-hand side matches `unshift` and `@` literally, whereas `e_1`, `e_2` and `number_` refer to non-terminals in the language definitions.

```
(reduction/context "UnshiftApp"
                   lang
                   e-ctxt
                   (unshift (@ e_1 e_2 ...) number_)
                   (term (@ (unshift e_1 number_)
                            (unshift e_2 number_) ...))))
```

PLT redex distinguishes multiple occurrences of the same non-terminal by additional characters following the underscore. If a left-hand side contains two identical occurrences of the same non-terminal, the corresponding input terms must be identical. The rule also uses another feature of PLT redex: The identifier `e_2` is followed by an ellipsis that has the same meaning as in `syntax-rules`: it matches any number of input terms that match: `e`.[3] As in `syntax-rules`, the ellipsis needs to reappear with the identifier but it is possible to add additional terms that will be repeated as well. Here `unshift` and the number that matched `number_` are copied as many times in the output as the input matched `e_2 ...`.

For an even more involved example, consider the definition of rule (ExpandNormDef) for PLT redex:

```
(reduction/context "ExpandNormDef"
                   lang
                   e-ctxt
                   (side-condition
                    (with-d c_ d_ number_mrk)
                    (not (dn? (term d_))))
                   (let ((dn_ (reduce-star-unique def-reduction (term d_))))
                     (term (with-d c_ ,dn_ number_mrk)))))
```

Here, the left-hand side is not simply a pattern for matching the input term, but the special form `side-condition`. This form receives as first argument a pattern and as second argument a Scheme expression. A left-hand side with `side-condition` only matches the input if the pattern matches and if the Scheme expression returns `#t`. The Scheme expression can access the bindings of the pattern. In this case, the function `dn?` checks whether the term bound to `d_` is a normalized definition. The definition of `dn?` uses the function `language->predicate` from PLT redex to derive a predicate that matches only terms described by the non-terminal `dn`:

```
(define dn?
  (language->predicate lang 'dn))
```

The right-hand side of the above rule is also special because of the comma within the argument of `term`. This form corresponds to `unquote` from R^{5}RS and embeds a Scheme value within a term. The Scheme value itself must also be a term. Here, the rule first normalizes the set of transformer bindings using the reduction $\rightarrow_{\mathrm{Def}}$. The variable `def-reduction` contains the rules of this reduction and the procedure `reduce-star-unique` applies it repeatedly to its second argument until no more reductions are possible. That is, it corresponds to the transitive closure of the reduction. In addition, the procedure checks that in each step only one reduction is applicable, that is, that the reduction is deterministic.

For the evaluation of `es-transformer` procedures, it is necessary to reduce Macro Scheme terms. As the operational semantics for the module calculus contains these terms as a sub-language, and I have translated this semantics to PLT redex as well (see Section 5.2.1), the

[3] This is of course not a coincidence: PLT redex is based on `syntax-rules`/`syntax-case` and the identifiers with underscores become pattern variables of the macro.

expansion of `es-transformer` macros can hand out macro applications to the Macro Scheme evaluator.

About 200 test cases building on the implementation for PLT redex exist. Many of these tests use macros generating other macros as this is the most difficult part of the system. For the automated testing procedures, the graphical front-end would be inappropriate as they require visual interpretation. Instead, I have written a small set of procedures that calls the reduction functions of PLT scheme directly and checks that every step returns just one contractum. This ensures that the reduction is indeed deterministic.

5.2.1 Rewriting System for $\Lambda_n^{\mathbf{Module}}$ and Macro Scheme

For the evaluation of expressions, I have implemented the operational semantics for Macro Scheme within PLT redex and added the rules for the evaluation of modules from Chapter 4. On top of these rules, the rules for program evaluation link and evaluate modules.

The rules for the evaluation of the primitives operating on syntactic objects access the reduction $\mapsto_{\text{patout}}$ and the function *free-id=?* from the implementation of the macro expander. The macro expander uses the evaluation reduction for Macro Scheme expressions for macros defined using `es-transformer`.

The implementation of the function *gen-loc* for the generation of locations uses a counter stored in a global variable to produce fresh locations, which are generated symbols. Before comparing the result of a test case with the expected result, an additional reduction replaces locations within the results by the values from the store. This makes it easier to specify the expected result as it is not necessary to track the location counter during evaluation.

5.2.2 A Short Review of Using PLT redex

This dissertation would not have been possible without PLT redex. The sheer number of rules and the broad coverage of the semantics made it necessary to experiment with the semantics using other means than writing down reductions manually. PLT redex turned out to be an excellent tool for this task. This section briefly summarizes the lessons I learned from using PLT redex.

I developed the semantics incrementally, adding new features only after I was satisfied with the results so far. For every feature, I wrote test cases that cover the feature alone (if possible) and in conjunction with other features. Two major kinds of errors usually occurred: either the test cases did not produce the expected results, or the reduction got stuck. The latter happened far more often than the first because as soon as a rule produces a small flaw, such as a forgotten underscore or an extra pair of parentheses, the pattern on the left-hand side of the next rule no longer matches and reduction stops. For finding such errors, I defined predicates for all non-terminals of the language (analogous to `dn?` above) and used them in the read-eval-print-loop to find out, which of the subterms of the last contractum did not conform to the non-terminal of the next rule.

The reductions in this dissertation are deterministic without exception. The reason is that the semantics is aimed to form the basis of a realistic implementation rather then to serve as a powerful equational theory. The semantics achieves determinism using contexts to uniquely determine the redex of a reduction. Formally, it is necessary to prove for each context a unique-decomposition lemma. Such a proof is tedious and error prone, especially in such a large setting. Algorithms exist to automate this proof [XSA01] and it would be very helpful if PLT redex would include them. I have instead relied on the test cases to identify violations of the lemma. For the test cases, I check after each reduction step that only one contractum exists. To locate the source of the error, the visual frontend of PLT redex is quite useful. The visual frontend displays the reduction sequence as a directed graph where the nodes are the terms that appear during reduction and an edge from one node to another corresponds to a reduction step between the corresponding terms. Alas, the frontend does not annotate the edges of the graph with a hint about the rules that led to the reduction.

PLT redex is a programming language with full access to Scheme and before I started to generate the rules automatically (see Section 5.4) and write descriptions of the rules, I made much more use of Scheme than it is now the case. The problem was that with the access to Scheme, it has been quite tempting to perceive the PLT redex terms as Scheme data and employ Scheme procedures to solve problems instead of trying to formulate appropriate rewriting rules. However, generating LaTeX code and especially describing the rules, and arguing about their correctness is almost impossible for rules that rely on Scheme procedures. A good example for this scenario is the $\rightarrow_{\text{import}}$ reduction from Figure 4.12. At the beginning, I had written a Scheme procedure `bind-ids` that generated the parsing substitutions for all identifiers imported by a module. The procedure used three nested `fold-right` functions to process an imported module. The rule for parsing modules called `bind-ids` before creating the abstract syntax of the module. While writing the description of this rule, I realized that importing is a separate phase after parsing but before expansion. Consequently, I introduced the $\rightarrow_{\text{import}}$ reduction and added for each aspect of importing a separate rule that describes this aspect. I consider the resulting reduction much more descriptive than the original Scheme procedure.

5.3 Direct Implementation of the Macro Expander

In addition to the implementation as a rewriting system, I have also implemented the macro expander directly in Scheme. The implementation is based on the concepts of the operational semantics but uses a recursive functions instead of repeatedly partitioning the term into a context and a redex. This technique is inspired by Felleisen's technique for deriving efficient abstract machines from standard-reduction functions [FF04]. The central idea is to avoid the repeated search for a redex by splitting the machine state into two parts: the *control string* and the *context*. The control string denotes the "current subterm of interest," while the context is the "current evaluation context". Putting the control string into the context yields the state of the original machine. The machine, called *CC machine*, performs actions of three kinds:

- If the control string is a redex, it applies the original reduction rule and builds a new state from the resulting term and the unchanged context.
- Otherwise, it splits the control string into a context and a new control string according to the context grammar and composes the resulting new context with the current evaluation context.
- If the control string is a normal form, it splits the context into the innermost (non-hole) context and the new current evaluation context. The machine obtains the new control string of its resulting state by putting the normal form into the hole of the innermost context. In the next step, the control string is either a redex or the machine splits it again.

For the λ_v-calculus, the resulting machine can be proved to produce the same results as the original machine. While Felleisen proceeds further to produce machines that represent the context in a stack-like fashion, an implementation in a real programming language can be derived directly from the CC machine as a recursive function:

- The redex corresponds to the base cases of the recursive definition. Here the function performs the work proper as described in the reduction rules and returns the result.
- The search for a redex translates into a recursive call on one of the sub-terms guided by the definition of the context. The function repeats this call until a normal form is returned. This is again repeated for all sub-terms specified by the context definition. Finally the function returns the term.
- If the term is in normal form, the function returns it.

Thus a translation of the resulting machine into a real programming language uses continuations to represent the context.

The above technique assumes that a standard-reduction function exists, that is, that it is possible to uniquely decompose each context into a redex and a context with a hole. All contexts in this dissertation have this property as can be easily verified.[4]

Consequently, while it is less obvious than with the PLT-redex implementation that the direct implementation matches the semantics, proving the direct implementation correct would be possible using the techniques described by Felleisen and Flatt [FF04] for the λ_v-calculus. However, such a proof is outside the scope of this dissertation, which deals with the definition of the semantics. The remainder of this section describes the direct implementation that rests upon the ideas just sketched to implement the semantics.

The abstract syntax produced by the expander is implemented as a set of records. The implementation also uses separate record types for the parsing substitution, the identifier substitution, the meta-substitution, and the shift operator. However, the expander constructs these records only when it builds the arguments of a macro application, otherwise it directly calls the according elimination functions. For the unshift operator, no record type exists as this operator always occurs with a `letrec-syntax` form and can be eliminated directly as it operates on expanded terms only. For the abstract syntax of expanded terms—the core expressions of Scheme—a translation to the abstract syntax of the Scheme 48 byte-code compiler exists. This makes it possible to use the expander as a front-end for Scheme 48. This is actually necessary to implement `es-transformer`: The code of the transformer procedure is generated by the macro expander but must be evaluated by Scheme 48, and therefore the Scheme 48 compiler must understand the output of the macro expander.

There are two main procedures; `expand` represents the core of the macro expander, and `parse-form` implements the parser. `Expand` takes as arguments the form to be expanded and the set of definitions. The form is either an s-expression, possibly subject to an expansion substitution, or an abstract syntax term which has not yet been fully expanded. In the first case, the expander calls the parser to produce abstract syntax. Otherwise it dispatches on the type of the abstract syntax and calls itself recursively on the sub-terms. The parser implements the rules from Figure 3.26 and also eliminates expansion substitutions.

The treatment of the parsing context (P in the reduction rules) is worth mentioning. On the left-hand sides of the parsing reduction rules, the parser uses the context to "peek" into the s-expression under the expansion substitutions. On the right-hand sides of the rules the parser applies the context to the parsed terms, and also uses it to resolve symbols to identifiers using the reductions $\rightarrow_{\mathrm{El}}$ and $\rightarrow_{\mathrm{SymRes}}$, and to build the syntactic object of a macro application (see rule (MacApp) in Figure 3.26).

For the `parse-form` procedure, it would be inefficient to repeatedly decompose its input into a parsing context and the s-expression. Still it needs the context to resolve symbols in two different ways and to build syntactic objects. The solution to represent the context as a procedure that immediately performs the task required by the parser and to add this context procedure as an argument to `parse-form`. The procedure receives two arguments: the form to be plugged into the hole and a message (a symbol) that specifies the task. The message either tells the context to reduce the term using the reduction $\rightarrow_{\mathrm{El}}$, or to reduce the term using the reduction $\rightarrow_{\mathrm{SymRes}}$, or to construct a syntactic object. The context procedure then eliminates the expansion substitutions accordingly or generates records to represent the substitutions applied to the term. At the start of the parser, the context procedure represents a parsing context containing the special parsing substitutions for λ, *letrec-syn*, and $\star$. If the parser encounters a expansion substitution, it generates a fresh context procedure that passes the form and the message to the current context procedure and performs the task specified by the message on the result. Note, that representing a context by a procedure that accepts as its argument the term to be substituted into the hole and returns the plugged term establishes a direct analogy with the definition of contexts.

[4] But see the remark about automatically proving this property in Section 5.2.2

For macro applications of `syntax-rules` transformers, the expander calls the same helper procedures as in the operational semantics. The implementation of `es-transformer` is more involved and will be described separately below in Section 5.3.1.

5.3.1 Implementation of Macro Scheme

For the evaluation of the `es-transformer` procedure and for applications of `es-transformer` macros, an implementation of Macro Scheme is required. Macro Scheme is an extension of Scheme with new expressions and primitives. The new expressions are expansion substitutions, `syntax`, and `syntax-lambda`, and the new primitives are `syntax-car`, `syntax-cdr`, `free-identifier=?`, `bound-identifier=?`, and `datum->syntax-object`. The new primitives all deal with syntactic objects as data and hence are straightforward to implement. For the new expressions it is not necessary to modify an existing implementation or even to write a new implementation from scratch. Instead, a set of macros translates these expressions into ordinary Scheme. The remainder of this section presents my implementation of the Macro-Scheme-specific expressions. It is then possible to evaluate the transformer procedure and the application of the resulting transformer closure within the macro expander using the Scheme procedure `eval`.

Figure 3.16 describes the evaluation of expansion substitutions in Macro Scheme. There, the reduction $\rightarrow_{\text{ES}}$ pushes the explicit substitutions inwards until they meet a `syntax` form, where the substitutions extend the syntactic object, or until they meet a λ-abstraction or a syntactic closure, where the substitutions stick. This propagation proceeds exactly in the same order as evaluation. The reason is that it takes place in the context C_{elim} and this context matches the normal evaluation context C^n with the sole addition that it also reduces the innermost explicit substitution. Hence, it is possible to model the propagation by maintaining a *current expansion substitution* during evaluation. The current expansion substitution represents all expansion substitutions that evaluation has generated so far. It is implemented as a procedure that accepts an s-expression as its single argument and augments it by the generated expansion substitutions. The evaluation of the transformer procedure creates the initial current expansion substitution from the expansion substitutions attached to the code of the transformer procedure (see rule (ESTransformer) in Figure 3.24). `Syntax-lambda` extends the current expansion substitution with a new meta-substitution and the evaluation of `syntax` uses the value of the current explicit substitution to turn its s-expression argument into a syntactic object. To model the propagation of the current expansion substitution to happen in parallel to evaluation, the implementation stores the current explicit substitution as a fluid binding [GKSK03]. However, for the evaluation of λ-abstractions, rule (EvalESLam) in Figure 3.16 prescribes that an explicit substitution should propagate to the body of a λ-abstraction. However, the evaluation of a `lambda` expression does not record the fluid bindings in the closure. Instead a procedure application propagates its fluid bindings to the body of the called procedure. I have circumvented this by modifying the binding of the `lambda` keyword to let it record the current substitution while creating the closure and let it reinstall this substitution for the evaluation of the body. This can be achieved easily by a macro that replaces `lambda`:

```
(define-syntax lambda
  (syntax-rules ()
    ((lambda vars body ...)
     (let ((subst (fluid $subst)))
       (r5rs-lambda vars
         (let-fluid $subst subst (r5rs-lambda () body ...)))))))
```

Here, `$subst` is the variable that contains the fluid binding for the current expansion substitution, `fluid` dereferences the fluid binding, and `r5rs-lambda` refers to the original binding for `lambda`. For the evaluation of the `syntax` expression it is also possible to define a small macro, which applies the current expansion substitution to the argument of `syntax`:

```
(define-syntax syntax
```

```
(syntax-rules ()
  ((syntax form)
   ((fluid $subst) 'form))))
```

`Syntax-lambda` is also a macro. It expands into a procedure that accepts the pattern variables as its arguments and uses them to extend the current expansion substitution by a new meta-substitution, which it binds during the evaluation of the body.

5.4 Generation of LaTeX Output

The LaTeX source code of all rewriting rules in this dissertation has been generated from the implementation of the rewriting system for PLT redex. This way, the dissertation is guaranteed to be consistent with the tested implementation of the rewriting system.

For the generation of the LaTeX source code, I have implemented a separate module `texify` for PLT Scheme. Then a separate module uses `texify` as its default language and includes the source code for the rewriting system. This construction is necessary because PLT Scheme does not support renaming of names from the the default language but I needed to redefine most of the standard procedures and special forms. The generation happens during the evaluation of the included code. In the source code, the form `define-reduction` defines a new reduction by listing a set of rules. `Define-reduction` is a macro that expands into code that generates the LaTeX code; I have redefined `define` into a macro that expands into `#t` to prevent any other top-level definitions from being processed.

The first argument to `define-reduction` is the name of the reduction and this name is also the name of a new file `define-reduction` opens (after prepending `.tex`). The `define-reduction` form emits for each rule an `\input` command into this file and puts the actual code into a file with the name of the rule. This makes it possible to choose between including single rules and complete reductions. The PLT redex forms `reduction` and `reduction/context`, which define the actual rules, perform the basic layout of the LaTeX code for single rules. They first emit the code for the left-hand side, then a `\rightarrow`, indexed by the name of the reduction, then the right-hand side of the rule, and finally a `\tag` command to define a name for the rule within a `align` environment of the `amsmath` package.

As a simple example of a generator for the actual terms, consider the representation of the unshift operator. As shown in Section 5.2.1, the form `(unshift e number)` represents the unshift operator within PLT redex. During the generation of the LaTeX code, `texify` binds `unshift` to:[5]

```
(define (unshift expr n)
  (format "(\\unshift{~a}{~a})" expr n))
```

`unshift` is in turn a LaTeX commando defined as

```
\newcommand{\unshift}[2]{\ensuremath{{#1}{\downarrow}^{#2}}}
```

Most of the generators are not this simple but complex macros that dispatch on the layout of the terms. This is necessary to prevent parenthesized forms from corresponding to procedure applications, or to produce context-sensitive output. An example that covers both cases is the `@` macro which procedure applications

```
(define-syntax @
  (syntax-rules (... with-d)
    ((@ (with-d c dn n) (... ...))
     (format "(@\\ (~a) \\dots)" (with-d c dn n)))
    ((@ (s11 s22 (... ...)) s2 (... ...))
```

[5]The double slash is necessary because a single slash creates an escape character within strings. The `format` procedure is the PLT-Scheme equivalent to `printf`, `~a` within the format string inserts the printed representation of the corresponding argument.

```
   (format "(@\\ \\texttt{(}~a ~a \\ldots\\texttt{)} ~a \\ldots)"
           s11 s22 s2))
  ((@ e (... ...))
   (format "(@\\  ~a \\dots)" e))
  ((@ c ...)
   (string-append "(@\\ " (string-join (list c ...) "\\ ") ")"))))
```

This macro definition contains two literal identifiers, `...` and `with-d`, to detect special cases. The first rule adds an additional pair of parentheses around the $\vdash$ form to make the output of the rule (ExpandApp) in Figure 3.9 more readable. The second rule uses the pattern `(... ...)` to detect uses of ellipsis within the definition of the rule. The pattern `(... ...)` matches ellipsis, if `...` is a literal of the macro. The rule matches the case where the first form after the `@` is again a parenthesized form and prevents this form from normal parsing where it would count as a procedure application. The third rule also covers a special case involving ellipsis whereas the ellipsis in the last case are the plain `syntax-rules` tool to match an arbitrary number of arguments, which the template of the rule makes into a list and before applying `string-join` to generate a single string separated by spaces.

For the generation of a complete reduction, consider the following excerpt from the definition of the $\rightarrow_{\downarrow}$ reduction:

```
(define-reduction unshift-reduction
  (reduction/context "UnshiftConst"
                     lang
                     e-ctxt
                     (unshift a_ number_)
                     (term a_))

  ...

  (reduction/context "UnshiftLam"
                     lang
                     e-ctxt
                     (unshift (lam (x_) e_) number_)
                     (term
                      (lam (x_) (unshift e_ ,(+ (term number_) 1)))))

  (reduction/context "UnshiftApp"
                     lang
                     e-ctxt
                     (unshift (@ e_1 e_2 ...) number_)
                     (term (@ (unshift e_1 number_)
                              (unshift e_2 number_) ...))))
```

Evaluation of this form generates a file `unshift-reduction.tex`. The evaluation of each `reduction/context` form emits an `include` command into this file:

```
\input{latex/UnshiftConst}\\
\input{latex/UnshiftLam}\\
\input{latex/UnshiftApp}
```

In addition, each `reduction/context` form generates a file containing the corresponding code. For this to happen, it was also necessary to define the non-terminal patterns because they become variables during evaluation:

```
(define number_ "n")
(define a_ "a")
```

```
(define x_ "x")
(define e_ "e")
(define e_1 "e_1")
(define e_2 "e_2")
```

That is, the characters after the underscore become subscripts in the LaTeX code. The file `latex/UnshiftConst.tex` then contains the following code:

```
(\unshift{a}{n}) \rightarrow_{\unshift{}{}}
   a\tag{UnshiftConst}\index{UnshiftConst}
```

The left-hand side of the rule directly results from evaluating `(unshift a_ number_)`. Each rule generates the arrow but receives the subscript from the current `define-reduction` form via a global variable. The right-hand side of the rule is the result of evaluating `(term a_)`. `Term` is defined as a macro that expands to its sole argument, hence `a` remains. Each rule also emits a `\tag` command from the `amsmath` package to display the name of the rule and a `\index` command to add an entry to the index file of the dissertation. Both commands use the rule name as the argument.

The code for (UnshiftLam) uses the following definition for `lam`:

```
(define-syntax lam
  (syntax-rules ()
    ((lam (var) c)
     (format "(\\lambda ~a.~a)" var c))))
```

and re-defines `unquote` as the identity function. The following LaTeX code results:

```
(\unshift{(\lambda x.e)}{n}) \rightarrow{\unshift{}{}}
  (\lambda x.(\unshift{e}{n + 1}))\tag{UnshiftLam}\index{UnshiftLam}
```

Finally, for the generation of (UnshiftApp), the third rule of the macro definition of `@` transforms the ellipsis into an `\ldots` command. The LaTeX code is then:

```
(\unshift{(@\ e_1\ e_2\ \ldots )}{n}) \rightarrow{\unshift{}{}}
   (@\ (\unshift{e_1}{n})\ (\unshift{e_2}{n})\ \ldots )
   \tag{UnshiftApp}\index{UnshiftApp}
```

The macro expansion system implemented within PLT redex covers modules as well. In particular, this means that the identifier representation contains an import path. To save the reader of Chapter 3 from this alien concept, each rule is actually output twice into two different directories, once for inclusion in Chapter 3 where modules are not discussed, and once with modules for Chapter 4. The generation of the code for Chapter 3 omits any output related to modules.

Chapter 6

Related Work

Several research areas form the basis of this dissertation. This chapter reviews previous work on the research that this dissertation combines: hygienic macro expansion, fully-parameterized module systems, the combination of modules and macros, and meta-languages for macro systems. The chapter starts out with a description of the relationship between this work and first-class macros. Then it presents existing fully-parameterized module systems, hygienic macro expansion algorithms, and module systems that support macros. Finally, it discusses the relationship between Macro Scheme and the meta-languages used in other macro systems.

6.1 First-Class Macros

Bawden describes a macro system where macros are parameterized over the bindings of the variables the macro introduces [Baw00]. In this system, macro transformers are the types of the corresponding keywords and macros are defined along with the variables they introduce within so-called *templates*, which can be seen as lexical environments. From the template, the `type-of` expression derives the type of a macro. The type enables the programmer to bind keywords with `declare`: For each keyword appearing in the list of bound identifiers, the `declare` form assigns a type (that is, a transformer) to the keyword. Thus macros become first-class entities. The macro expander uses the type to expand macro applications in the body of the `lambda` but the expanded code contains references into the template instead of names for the variables of the template. The `instantiate` form creates a run-time representation of the template which the programmer fills with the values of the template's variable and which she passes to the functions that accept keywords of the corresponding type. By passing different instantiations of the template to the function, different bindings are in effect in the macro-expanded code. As applications of these first-class macros, Bawden shows the definition of data structure definitions and a small module system.

The module system from Chapter 4 provides the same power as Bawden does without any additionally machinery. As modules are fully-parameterized, for any module that imports an interface containing macros, linking determines the bindings of the variables inserted by the imported macros. The following code shows the translation of Bawden's example where the `delay` macro from R^{5}RS is parameterized over the implementation of `make-promise` and `force`:

```
(define-interface promises-interface
  (open scheme-interface)
  (define-syntax delay
     (syntax-rules ()
       ((delay form) (make-promise (lambda () form)))))
  force)

(define-module simple-promises promises-interface
```

```
   (open scheme-interface)
   (begin
     (define (make-promise thunk) thunk)
     (define (force promise) (promise))))

(define-module memo-promises promises-interface
   (open scheme-interface)
   (begin
     (define (make-promise thunk) (cons #f thunk))
     (define (force promise)
       (if (car promise)
           (cdr promise)
           (let ((val ((cdr promise))))
              (if (car promise)
                  (cdr promise)
                  (begin (set-car! promise #t)
                         (set-cdr! promise val)
                         val)))))))

(define-module lazy-lists (export lazy-map)
  (open scheme-interface
        promises-interface)
  (define (lazy-map f l)
     (if (null? l)
         '()
         (cons (f (car l))
               (delay (lazy-map f (cdr l)))))))
```

The interface `promises-interface` defines the macro `delay` and the `force` procedure. Two modules implement this interface: `simple-promises` just uses a thunk to represent the delayed computation while `memo-promises` also implements memoization as required by R^{5}RS. The module `lazy-lists` imports `promises-interface` and uses the `delay` macro to implement lazy lists. As the example uses top-level modules, the configuration phase selects the actual implementation of promises used within the program. Relying on the configuration phase is probably sufficient in most situations. However, as first-class modules can export and import macros as well, the equivalent of a function using a first-class macro (which in Bawden's system would use `declare` to reference the macro transformer) is a module importing that macro, and the equivalent of Bawden's template instances are modules exporting the macro. Instead of procedure application, run-time linking resolves the variable references in the macro output.

Even though Bawden's macros can describe a module system, the requirements for a module system are manifold enough to warrant a separate facility for a programming language. As the emulation of Bawden's system shows, first-class modules with macros yield first-class macros.

6.2 Fully-Parameterized Module Systems

Kelsey's unpublished paper "Fully-parameterized Modules or The Missing Link" has been the starting point of my work. In the paper, Kelsey combines `ld`'s automatic inference of the dependency graph with explicit interfaces known from modern languages. Unlike in conventional systems, Kelsey's modules import interfaces only, thus yielding a fully-parameterized module system. His semantics is a set of abstract data types for the module system and he focuses on the automatic linker and the derivation of the *resolve* function, which maps qualified names within a module to the providing module and the name within that module. Section 2.5 revisits the core of Kelsey's description. In the paper, Kelsey also describes additions like renaming and tags on import (covered in Section 4.6), circular dependencies, and copying of modules. Kelsey covers macros

only with a few remarks that are tailored to a module system without independent compilation. In contrast, Chapter 4 defines the semantics of a programming language with fully-parameterized modules and macro definitions within interfaces that enable independent compilation in the presence of macros.

In his Master's thesis [Wie00], Wiesenmaier uses Kelsey's idea to develop an implementation of a fully-parameterized module system. The implementation is essentially a source-to-source transformation that resolves the references to imported identifiers. The transformation has the important property of preserving the sharing of code among modules with the same unit. The transformation works by generating a procedure for each unit, which expects a vector of cells as its only argument. The vector corresponds to the imports of the unit and the cells contain the values of the imported variables. The transformation maintains a mapping from imported names to indices into the vector for each unit. It uses this mapping to convert occurrences of global variables into references in the vector. Applying the unit procedure to a vector filled with cells containing imports creates a module. The transformation covers macros as well but keeps the macro definitions in the module body and does hence not provide independent compilation. The configuration language of Wiesenmaier (which is in turn based on the configuration language of Scheme 48 [Ree94]) has also influenced the configuration language as presented in Section 2.7.

Flatt's Unit system [FF98] is a fully-parameterized module system. In Flatt's system, a *unit* imports and exports variables, contains a set of definitions and an initialization expression. Units can be *invoked*, which means that the imported variables are replaced by values, and the value of the initialization expression is evaluated in the scope of the unit's set of definitions. Furthermore, two units can be assembled to a *compound* unit, where the imports of each of the two units can be satisfied by the other unit respectively. Assembling units roughly corresponds to linking modules. Units are run-time values and linking happens at run-time as well. The semantics of units is specified in terms of meta-substitutions. Due to their run-time nature, units cannot import or export macros. Furthermore, the programmer needs to write down the linking steps manually which gets tedious for large programs and makes it hard to define a compound unit which is not parameterized over an imported unit of one of its sub-units.

Mixin Modules [HLW04] are very similar to units but take place within the typed setting of the ML language. Mixin modules place emphasis on the order the evaluation of module definitions by ordering the definitions automatically, assisted by user-defined annotations. Such annotations also exist in our configuration language but they have no effect on the evaluation as the semantics does not support recursive modules yet. In addition, with mixin modules it is possible to replace bindings of a module which is then called *late binding*. Unlike our system, mixin modules do not contain interfaces and hence linking must happen manually.

For the language Scala, Odersky and Zenger [OZ05] use classes with abstract types to obtain a system for component based programming. In Scala, abstract classes may contain abstract type and value declarations and inheritance using mixins implements these abstract types and provides concrete values. The resulting system is very similar to a fully-parameterized module system because the abstract entities correspond to imported modules and defining an inherited class corresponds to the implementation of an imported module.

6.3 Macro Expansion Algorithms

The work by Bove and Arbilla [BA92] is closest to the semantics for hygienic macros in Chapter 3 and it has also been of great influence for my semantics. Section 3.11 includes a detailed discussion that shows how my semantics improves hygiene and feature coverage upon the calculus of Bove and Arbilla.

"Syntactic Closures"[BR88] are another concept for implementing hygienic macros. There, transformers are procedures in a meta-language (which is ordinary Scheme) that accept, in addition to the s-expression representing the macro application, the *syntactic environment* of the macro application. A syntactic environment maps names to identifiers with binding information. A *syntactic closure* consists of a syntactic environment, a list of names to be left free, and an

expression. The syntactic closure represents a piece of code (the value of the expression) where all free identifiers (besides the ones to be left free) have been bound according to the closure's syntactic environment. The identifiers to be left free can be captured at the invocation site, thus the system can support non-hygienic macros as well. A typical transformer returns a new syntactic closure that contains the syntactic environment from the macro definition, no free names, and an expression representing the output. Within the expression, the programmer may insert sub-expressions that stem from the input but, to ensure hygiene, the programmer must wrap these sub-expressions into syntactic closures whose syntactic environment is the syntactic environment passed to the transformer. While syntactic closures look rather powerful and straightforward to implement, they rely completely on mechanisms of the meta-language to implement hygiene. That is, to understand the semantics of a macro application it is necessary to reconstruct the complete evaluation within the meta-language and trace the values that represent input and output forms, and the syntactic closures themselves. In contrast, the semantics in Chapter 3 contains special means to explain hygiene (mainly the expansion substitutions). To explain the semantics of `syntax-rules`, only a few additional functions that perform matching and generation of substitutions at the meta-level are required. For computational macros, the semantics adds means that deal with hygienic macro expansion to the semantics of the meta-language: syntactic objects, `syntax-lambda`, and the evaluation of expansion substitutions.

Based on the ideas of syntactic closures, Kolbly defines an algorithm for hygienic macro expansion that is interleaved with compilation [Kol02]. The algorithm uses the compile-time environments to preserve the lexical information of identifier occurrences and does not generate the expanded source code but compiles it directly.

The original "Hygienic Macro Expansion" paper [KFFD86] presents the first algorithm for hygienic macros. The algorithm avoids capturing by renaming new identifiers. It consists of four phases: The first phase replaces all identifiers by time-stamped identifiers with time stamp 0. The second phase applies the macro expansion functions. After each transformation, the result is again time-stamped and the current time is increased. After expansion, a stripping phase α renames all bound identifiers to fresh, unstamped identifiers. The final phase renames the remaining stamped identifiers (which are the free identifiers) back to their original name. As an extension, the user can mark identifiers that should be captured by macro expansion by simply applying the initial time-stamping function to them. In the paper, macros are always declared at the top level, but local macro definitions are sketched.

"Macro-by-Example: Deriving Syntactic Transformations from their Specifications"[KW87] introduces the pattern language present in `syntax-rules` but without a list of literals. The paper contains a formal semantics for the language along with the derivation of a practical compiler. However, the semantics directly operates on the pattern language. This rules out macro definitions generated by other macros, which has been an important aspect of the semantics for `syntax-rules` in Section 3.9.

A pattern-based macro system is also the subject of "Macros That Work"[CR91], which combines the algorithm by Kohlbecker et. al. with syntactic closures. The system does not feature a meta-language and administers syntactic environments directly. The expansion algorithm renames bound variables and records the fresh names in the syntactic environment. Dealing with literal identifiers within patterns is simplified in the paper by using a special lexical syntax for pattern variables. However, as noted by the authors, this simplification does not extend to macros defining other macros and indeed much complexity of our semantics for `syntax-rules` in Section 3.9 stems from the proper handling of this case. The paper advocates against the direct usage of syntactic closures from [BR88] for the reason that syntactic closures are too low-level and are not suitable for implementing pattern-based macro systems. Later Hanson showed how to overcome the latter limitation rather easily [Han91].

"Syntactic Abstraction in Scheme"[DHB92] describes the `syntax-case` system. The macro expansion algorithm is an improved version of [KFFD86] with linear complexity. The central contribution of the `syntax-case` system is the representation of identifiers. To support hygiene and the more elaborated operations on syntactic objects, identifiers in the formal system are abstract objects with the following operations: *mark* adds a mark to an identifier and *subst* replaces the

binding name of an identifier by a different symbol. The binding name is used to determine the binding place of an identifier. The symbolic name of an identifier remains unchanged throughout the macro expansion process and is used to maintain the source-object mapping. Before application of a macro transformer, the macro expander applies the *mark* operation with a fresh mark to the input. The same happens to the output of the transformer, but with the same mark. As identical marks cancel, exactly the identifiers introduced by the macro remain marked. The expander uses the *subst* operation to rename identifiers to fresh names if it encounters a binding construct. At the same time, the expander records the "type" of the bound variable in an expansion time environment along with the transformers. A transformer is an expression of the meta-language which is again full Scheme. The transformer receives the macro application form as a *syntax object*, an s-expression subject to *subst* and *mark* operations, and must also return a syntax object. In the formal model, there is a *plambda* special form, which binds a single pattern variable within its body. The `syntax-case` form itself is a pattern matching facility in the spirit of `syntax-rules`, with the change that the right-hand sides of the rules are meta-language expressions which must evaluate to syntax objects. However, the paper contains no formal semantics of `syntax-case` itself. Many aspects of `syntax-case` have inspired the computational macros as presented in Section 3.6 including `syntax`, `free-identifier=?`, `bound-identifier=?`. However the two systems differ heavily in the scoping interaction between the meta-language and the object language as detailed in Section 3.8.1. Our semantics has been designed to perceive syntactic objects as data with `syntax-lambda` as an operation for hygienic insertion into it. In the `syntax-case` model on the other hand the behavior results from the usage of renaming to implement hygiene: for each λ-abstraction, `syntax-case` renames the names of the bound variables within the body to fresh names, regardless of whether the names will later become variables of the same phase or not. The same also applies to λ-abstractions of the meta-language for which the system also renames names appearing within `syntax` expressions. Section 3.8.1 contains a number of examples where the ruthless renaming of `syntax-case` violates the principle of least astonishment.

6.4 Module Systems with Macros

The module systems of most major Scheme implementations allow users to define modules that import and export macros. Among these systems are Chez Scheme [Dyb98], PLT Scheme [Fla04], and Scheme 48 [KR02]. However none of them offers independent compilation and hence they do not meet our definition of a module as an exchangeable component.

Blume's draft paper [Blu97] mentions the idea of adding macros to interface declarations to achieve independent compilation but provides no formal semantics whatsoever. Blume suggests to declare all identifiers inserted by exported macro in the interface. The interface declaration contains an additional form to declare the exported identifiers, hence hidden exports are still possible. As already mentioned in Section 4.1, Blume's proposal does not extend to computational macros that generate identifier names but makes it possible to check whether a module satisfies its exported interface. Checking the internal consistency of an interface is still not possible.

Recently, Culpepper, Owens and Flatt combined Flatt's Unit system (described in Section 6.2) with macros [COF05]. Just as in our system, they put the exported macro definitions into the interface declarations. The main difference to our systems seems to be that Culpepper et. al. use the existing macro expander and the top-level module system of PLT Scheme to implement signatures to embed units with macros whereas our system starts from scratch, based on a formal semantics.

6.5 Meta-Languages for Macros

For computational macros such as our `es-transformer` from Section 3.7, it is necessary to specify a meta-language that computes the result of a macro application. In our case, this language is Macro Scheme. A significant body of research exists that describes the semantics of such *staged*

evaluation languages. We focus on the discussion of macro systems here, as opposed to more general languages that have programs in their domain.

In "Macroexpansion Reflective Tower" [Que96], Queinnec provides a model for understanding macro expansion using a reflective tower. In the model, the meaning of a program is the result of first evaluating a call to a macro expansion function applied to the source code and then obtaining the meaning in the core language. The evaluation of the call to the macro expansion function obtains the meaning of the call one level up in the tower and executes the returned denotation. As the meaning in the upper level also includes macro expansion, this definition builds an infinite tower of language levels. Our systems does not provide direct access to the expansion function, but for `es-transformer` a recursive definition very similar to Queinnec's description emerges: the rule (ESTransformer) in Figure 3.24 first applies the function *expand'* to expand the code of the transformation procedure before passing the result to $eval_v^{MS}$. Hence, the macro expander recursively calls itself and $eval_v^{MS}$ is a special evaluation function in the upper level. Unlike Queinnec's system, our system does not use s-expressions to represent the expanded code, but instead uses the mixture syntax to convey lexical information between the levels and thus to implement hygiene, which Queinnec's model can only do using renaming.

In [Fla02], Flatt deals with separation of evaluation phases that arise in the presence of a full-fledged programming language at the macro level. He introduces a `require-for-syntax` module clauses to import identifiers into the macro phase only and describes the management of the state of the macro language. Flatt's ideas could be applied to our system as well if Macro Scheme were extended to include state.

MacroML [GST01] is an extension of Standard ML that translates into MetaML [Tah99], a language for staged evaluation. MetaML provides constructs for defining and executing code expressions and, within these code expressions, an escape facility that embeds the result of evaluating expressions into the code. Thus, the escape facility allows the programmer to combine computed code with static code. For the code expressions, MetaML can derive a type that indicates the type of the expression itself. MetaML uses renaming to prevent inadvertent capture of variables while combining code, and MacroML builds on this feature to implement hygiene. Macros in MacroML also inherit the typing properties from MetaML, which makes it possible to write type-safe macros. The translation from MacroML to MetaML translates ordinary expressions into code and macro definitions into escaped MetaML function definitions. The syntax of macro definitions in MacroML is restricted in such a way that the system can recognize macros that introduce bindings forms with meta-variables as binding variables the bound meta-variable and the body. For such a body, the translation generates a MetaML function that returns the body abstracted over the bound variable and translates references of the body into applications of the body function with code representing the actual macro argument. For occurrences of macro parameters within the body, the translation generates an escaped occurrence of the parameter. Thus macros binding variables are translated into MetaML functions and MetaML's renaming prevents variables within expression bound to the parameter from being captured by surrounding binders and ensures that the parameter of the body function does not bind variables within the body code. Compared a fully-fledged macro system like `syntax-case` or the system presented in this dissertation, MacroML lacks a lot of features: for example it cannot handle macros that expand into macro definitions or macro applications, macros that inspect their arguments, and macros that take other macros as arguments. It is an open question whether it is possible to extend MacroML to cover these features.

Taha and Johann [TJ03] have extended MacroML with an equational theory that also supports alpha-conversion for bound variables in macro definitions that introduce binding forms. While this revised version is still very restrictive, it indicates a direction of future research on macro systems in Scheme.

Chapter 7

Conclusions

This dissertation shows that it is possible to define a formal semantics for fully-parameterized first-class modules with hygienic macros, independent compilation, and code sharing. The semantics in this dissertation formalize configuration, elaboration, linking, and evaluation of modules. It makes use of explicit substitution to define hygienic expansion, expressive interfaces to provide independent compilation, and explicit substitutions to enable code sharing.

7.1 Review

Chapter 2 defines a configuration language for a fully-parameterized module system with automatic linking. The module system supports macros while providing independent compilation because interfaces contain the definitions of exported macros and the `free` and `open` clauses of interfaces establish the lexical environment for these macros. The semantics of the language translates into abstract data types from which it is possible to derive a link environment using Kelsey's ideas.

For elaboration, the semantics defines a parser and a hygienic macro expander. The semantics for the expander covers almost all features of the popular `syntax-rules` transformer and gives a detailed description for computational macros, including a semantics for the meta-language Macro Scheme. Explicit substitutions are the main means capturing hygiene without relying on non-pure renaming techniques. The semantics for parsing and expansion supports fully-parameterized modules, retrieving macro definitions from interfaces. It supports first-class modules through the identifier representation, which covers module expressions as binding places.

Guided by a link environment, the linking semantics combines fully-parameterized modules into linked packages. The result is a template for each package that serves as an explicit substitution during evaluation. Thus the semantics preserves code sharing as it does not alter the module body for linking. The linker is aware of macros through the identifier representation and collects all identifiers of the imported macros in the template to support hidden exports.

For evaluation $\Lambda_n^{\text{Module}}$, an extension of the λ_v^n calculus with explicit substitutions, describes evaluation of modules where module-bound identifiers receive their values from templates. Higher-order modules extend the set of expressions; the semantics fully incorporates them by defining their evaluation to packages, linking, evaluation, and access to bindings for the package.

Finally, the prototype implementations of the semantics show that the semantics supports a realistic implementation. Thus, the semantics precisely defines a formal framework for the implementation of a powerful module system that should scale to the definition of large applications.

7.2 Insights Gained from the Macro Expansion Semantics

The semantics in Chapter 3 reveals several facts about hygienic macro expansion that are not obvious from previous research. This section summarizes these facts.

For the well-known `syntax-rules` facility, our semantics describes pattern matching with a list of literals. The semantics reveals that the list of literals in a `syntax-rules` declaration is a binder for literals in the pattern. Of course existing implementations are treating the list of literals like this, but no formal semantics covered this aspect so far.

In the semantics, it also becomes apparent, why the name is not enough to distinguish variables bound by the same λ-abstraction: If meta-variables occur as bound variables it may happen that in the expanded code the same variable occurs several times in the list of bound variables (see the macro `dolet` in Section 3.10 for an example). In a semantics based on renaming, this is not observable because renaming puts fresh, unique names into the list of bound variables. Our semantics however uses the bound variables as present in the source term or resolved from meta-variables and the identifier representation selects the proper binding place.

The comparison of our semantics with the semantics of Bove and Arbilla in Section 3.11 makes it obvious that with de Bruijn indices the set of transformer bindings must be adjusted as expansion moves into the scope of local binders. It also shows that including a description of the parser is vital for a realistic semantics.

Section 3.8.1 stresses the main differences between our semantics and the `syntax-case` model. While `syntax-case` uses renaming of meta-language terms to transport lexical information semantics generates explicit substitutions as meta-language expressions and thus meta-language evaluation propagates the lexical information. For most of the differences, the section also describes how to emulate `syntax-case`, which yields a better understanding of the `syntax-case` model. The section also recapitulates our treatment of meta-variables in the source code of the meta-language, another aspect that has not been addressed formally before.

Perceiving marking as a meta-language operation also allows us to use an approach to marking different from `syntax-case`. With `syntax-case`, the expander marks the input of a transformer and applies the same mark to the output; and marks have the property that identical marks cancel. As `syntax-case` does not make marking and renaming meta-level operations, it uses this indirect technique to identify the terms inserted by the transformer. We achieve the same effect directly—without duplicate marking—by applying a mark operator to the transformer closure or the `syntax-rules` template before applying the transformer. Our scheme is reminiscent of the time-stamping algorithm of the original macro expansion paper [KFFD86] (which did however not cover computational macros) but avoids the "very undisciplined" [BR88] marking that the original algorithm performs on source code terms.

7.3 Future Work

Sections 3.10 and 4.6 already summarize possible extensions of the semantics for the macro system and the module system. The most critical items on these lists are probably the addition of top-level macro applications and the treatment of interfaces as elaboration-time values. The first item requires two steps: A definition of the behavior of top-level macro definitions and applications that is not ambiguous and a corresponding extension of the identifier representation for describing the position of a top-level binding. Treating interfaces as elaboration-time values requires that the parser for module definitions recognizes meta-variables within module definitions and that the macro expander implements the Amendment of Hygiene Condition defined in Section 4.6.

The full integration of the implementations of the macro expander and the module system within an existing Scheme implementation is another pending issue. I plan to carry out this implementation within Scheme 48 as this system has a tractable implementation. For the macro expander, a prototype is already available. For a direct implementation, it is necessary that the parser and the macro expander operate directly on the abstract syntax tree of Scheme 48. The main difficulty with this effort is that it requires an exchange of the identifier representation, and thus affects the whole system. The same hold for the module system, which is also used to describe the implementation of the system itself.

A correctness proof for the direct implementation of the macro expander with regard to the semantics is also desirable. Given the semantics defined in this paper, it is possible to formulate

and prove properties about macro expanders; Section 5.3 already discusses how to relate the direct implementation to a control machine. The concrete proof needs to be worked out, though.

Aside from direct extensions of the work presented in this dissertation, other research topics can also benefit from it. Most notably, the literature so far defines the term "hygienic macro expansion" as a set of informal requirements on the macro expansion process. While this definition has so far served its purpose well, a formal definition of hygienic macro expansion is highly desirable. It would, for example, allow the hygiene properties of formal definitions of macro expansion, such as the one in this dissertation, to be validated. I believe that my representation of identifiers using de Bruijn levels could be useful for a formal definition of hygienic macro expansion as it enables reasoning about the binding places of identifiers occurring in the macro input and output in an abstract manner.

The semantics for the `syntax-rules` facility form R^5RS in Section 3.9 suggests that this tool is surprisingly complex and hard to grasp. Designing a different, rewriting-based transformer may therefore be worthwhile. The formal description points to two sources of the complexity in `syntax-rules`: First, as macros may expand into the definition of other macros, the parser must work hard to identify literal identifiers, binding meta-variables and meta-variables within patterns and the literal list. If macro definitions expanding into `syntax-rules` (or a variant thereof) would be prohibited, the semantics would become much simpler. A second complication is the fact that the pattern definition and the template contain no indication about the meta-variables that occur as binding variables in the output and hence require special treatment to maintain hygiene. MacroML [GST01] has worked around this issue by restricting the layout of macros that introduce binding forms to variants of `let` where the bound variables and the body are apparent. Such a form could be defined for Scheme as well and I expect the resulting semantics to be straightforward.

7.4 Closing Words

Explicit substitutions and de Bruijn indices are essential ingredients of the semantics and I would like to close this work with a few remarks about the relationship between these devices and conventional techniques.

De Bruijn Indices vs. Renaming Ever since the invention of hygienic macro expansion, the algorithms have been based on renaming. The explanation by renaming became so popular that nowadays hygienic macros are often perceived as equal to consistent renaming. However, this is not the case. Hygienic macro expansion is about tracking the binding place of identifiers during source code manipulation. Renaming is one *technique* to achieve this tracking but, as it needs state to generate fresh identifiers, it is not purely functional, requires a careful implementation and is still not very descriptive for advanced features such as `syntax-rules` and computational macros. On the other hand, tools for tracking the binding place of identifiers that do not depend on unique names exist since the 1960's , namely the de Bruijn notation. This dissertation has shown that through an extension of this notation into other dimensions such as marks, position, or import path, it is possible to give a precise, purely functional semantics of macro expansion. In the end, renaming is not necessary because each variable occurrence has a unique binder. You just need to know how to find it.

Explicit Substitutions vs. Meta-Substitutions At the start of my dissertation project, I was looking for a means to formalize code sharing for modules with identical units. Denotational semantics is rather inappropriate for this task, as mathematics has no notion of "sharing." Rewriting semantics on the other hand deals with pieces of text, hence copying or sharing of terms is expressible. Hence I decided to choose a rewriting-based semantics. Cardelli mentioned the idea of using explicit substitutions for linking of modules to prevent the code expansion and loss of module identity caused by meta-substitutions [Car97]. My construction of accumulating the linking steps outside the unit in the template of a package and propagating for evaluation the

identifier/value pairs into the unit by an explicit substitution implements this idea and makes the relationship between the semantics and a realistic implementations apparent.

Another important aspect of explicit substitutions is the possibility to intermingle the elimination of substitutions with other reductions. Meta-substitutions do not offer such freedom as they are meta-level operations, which are performed immediately. For macro expansion, however, laziness is important because the expander needs to propagate information about identifiers to source code terms that have not yet been parsed. Without laziness, macro expansion operates on unparsed source code because parsing must happen interwoven with macro expansion and thus cannot have generated the complete abstract syntax yet. This makes it harder to understand the correctness because for renaming it breaks the analogy to α-conversion which is defined on λ-terms only. It is also interesting to note that immediate renaming as performed by the original macro expansion paper [KFFD86] leads to quadratic complexity which has been reduced to linear complexity by the `syntax-case` model by describing a lazy version of renaming—without, however, abandoning the operation on source code terms. In our semantics the usage of explicit substitutions also lead to optimal run-time complexity for hygienic macro expansion.

Appendix A

Notation

$\rightarrow_R$	Reduction R
$A \rightarrow_R B$ $(Name)$	Rule $(Name)$ of reduction R, i.e. (A, B) is $\in \rightarrow_R$
$\rightarrow_R^*$	Reflexive transitive closure of reduction R
$\mapsto_R$	Standard reduction function of reduction R
$\mathcal{P}(M)$	Powerset of the set M
M^*	Sequences over the set M
ϵ	The empty sequence
$x : s$	Sequence concatenation: prepend the element x on the sequence s
s_i	Select ith element of sequence s
$\equiv$	Syntactic equality
$\underline{t}$	Normal form of t (w.r.t. some reduction)

Bibliography

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[BA92] Ana Bove and Laura Arbilla. A confluent calculus of macro expansion and evaluation. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 278–287, San Francisco, California, USA, June 1992. ACM Press.

[Bar84] H. P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.

[Baw00] Alan Bawden. First-class macros have types. In Tom Reps, editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, pages 133–141, Boston, MA, USA, January 2000. ACM Press.

[Blu97] Matthias Blume. Separate compilation for Scheme. `http://ttic.uchicago.edu/~blume/papers/scm-sc.ps.gz`, 1997.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BR88] Alan Bawden and Jonathan Rees. Syntactic closures. In *ACM Conference on Lisp and Functional Programming*, pages 86–95, Snowbird, Utah, 1988. ACM Press.

[Bru72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag Math.*, 34(5):381–392, 1972.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In Neil Jones, editor, *Proceedings of the 1997 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, January 1997. ACM Press.

[Cli91] William Clinger. Hygienic macros through explicit renaming. *Lisp Pointers*, IV(4), 1991.

[COF05] Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In Robert Glück and Michael Lowry, editors, *Generative Programming and Component Engineering, Proceedings*, number 3676 in Lecture Notes in Computer Science, pages 373–388, Tallin, Estonia, September 2005. Springer-Verlag.

[CR90] William Clinger and Jonathan Rees. Macros that work. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, pages 155–162, San Francisco, CA, January 1990. ACM Press.

[CR91] William Clinger and Jonathan Rees. Macros that work. In *Proc. 1991 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 155–162, Orlando, Florida, January 1991. ACM Press.

[Cur96] Pierre-Louis Curien. Categorical combinators. *Information and Control*, 69:288–254, 1996.

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In Greg Morrisett, editor, *Proc. 30th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, LA, USA, January 2003. ACM Press. ACM SIGPLAN Notices (38)1.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, 1990. ACM Press.

[DHB92] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

[Dyb92] R. Kent Dybvig. Writing hygienic macros in Scheme with syntax-case. Technical Report #356, Indiana University Computer Science Department, Bloomington, Indiana, June 1992.

[Dyb96] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 2nd edition, 1996.

[Dyb98] R. Kent Dybvig. *Chez Scheme User's Guide*. Cadence Research Systems, 1998. `http://www.scheme.com/csug/index.html`.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In Keith Cooper, editor, *Proceedings of the 1998 Conference on Programming Language Design and Implementation*, pages 236–248, Montreal, Canada, June 1998. ACM Press. Volume 33(5) of SIGPLAN Notices.

[FF04] Mattias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Utah CS6520 Version, URL: `http://www.cs.utah.edu/plt/publications/pllc.pdf`, 2004.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 10(2):235–271, 1992.

[Fil94] Andrzej Filinski. Representing monads. In POPL 1994 [POP94], pages 446–457.

[Fla02] Matthew Flatt. Composable and compilable macros: You want it when? In Peyton-Jones [PJ02], pages 72–83.

[Fla04] Matthew Flatt. *PLT MzScheme: Language Manual*. Rice University, University of Utah, December 2004. Version 209.

[GG01] Martin Gasbichler and Holger Gast. Soft Interfaces: Typing Scheme at the module level. In Manuel Serrano, editor, *Proceedings of the 2nd Workshop on Scheme and Functional Programming*, Florence, Italy, September 2001.

[GKSK03] Martin Gasbichler, Eric Knauel, Michael Sperber, and Richard Kelsey. How to add threads to a sequential language without getting tangled up. In Matthew Flatt, editor, *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, Boston, November 2003.

[GS02] Martin Gasbichler and Michael Sperber. Final shift for call/cc: Direct implementation of shift and reset. In Peyton-Jones [PJ02].

[GST01] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In Xavier Leroy, editor, *Proceedings of the 2001 International Conference on Functional Programming*, pages 74–85, Florence, Italy, September 2001. ACM Press, New York.

[Han91] Chris Hanson. A syntactic closures macro facility. (from the Scheme repository), November 1991.

[HLW04] Tom Hirschowitz, Xavier Leroy, and J.B. Wells. Call-by-value mixin modules: Reduction semantics, side effects, types. In *Proc. 13th European Symposium on Programming*, Lecture Notes in Computer Science, Barcelona, Spain, April 2004. Springer-Verlag.

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. In *Proceedings of the 1996 ACM SIGPLAN Symposium on Principles of Programming Languages*, St. Petersburg, FL, USA, January 1996. ACM Press.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[Kel97] Richard A. Kelsey. Fully-parameterized modules or the missing link. Technical Report 97-3, NEC Research Institute, Inc., 1997.

[Kel99] Richard Kelsey. SRFI 9: Defining record types. `http://srfi.schemers.org/srfi-9/`, September 1999.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161, 1986.

[Kol02] Donovan Kolbly. *Extensible Language Implementation*. PhD thesis, The University of Texas at Austin, 2002.

[KR95] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[KR02] Richard Kelsey and Jonathan Rees. *Scheme 48 Reference Manual*, 2002. Part of the Scheme 48 distribution at `http://www.s48.org/`.

[KW87] Eugene Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In POPL 1987 [POP87], pages 77–84.

[LAD05] Linux audio developer's simple plugin api. `http://www.ladspa.org`, 2005.

[LB88] B. Lampson and R. Burstall. Pebble, a kernel language for modules and abstract data types. *Information and Computation*, 76(2-3):278–346, 1988.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In Leroy [POP94], pages 109–122.

[MFFF04] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In Vincent van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 301–311. Springer, 2004.

[New42] M. H. A. Newman. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.

[Org96] International Standards Organization. ISO/IEC 10514-1 (Modula-2, base language), 1996.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In Richard P. Gabriel, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume TBA of *ACM SIGPLAN Notices*, page TBA, San Diego, CA, USA, October 2005. ACM Press, New York.

[PJ02] Simon Peyton-Jones, editor. *International Conference on Functional Programming*, Pittsburgh, PA, USA, October 2002. ACM Press, New York.

[POP87] ACM. *Proc. of the 14th Annual ACM Symposium on Principles of Programming Languages*, 1987.

[POP94] *Proceedings of the 1994 ACM SIGPLAN Symposium on Principles of Programming Languages*, Portland, OR, January 1994. ACM Press.

[Que96] Christian Queinnec. Macroexpansion reflective tower. In Gregor Kiczales, editor, *Reflection'96*, pages 93–194, San Francisco, CA, USA, April 1996.

[Ree94] Jonathan A. Rees. Another module system for Scheme. Part of the Scheme 48 distribution, January 1994.

[Ros96] Kristoffer H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, University of Copenhagen, 1996. also DIKU Report 96/1.

[Sto99] John David Stone. SRFI 8: `receive`: Binding to multiple values. `http://srfi.schemers.org/srfi-8/`, August 1999.

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, Portland, Oregon, USA, July 1999.

[TJ03] Walid Taha and Patricia Johann. Staged notational definitions. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering, Proceedings*, number 2830 in Lecture Notes in Computer Science, pages 97–116, Erfurt, Germany, September 2003. Springer-Verlag.

[TWA00] TWAIN specification (version 1.9). `http://www.twain.org/docs/Spec1_9_197.pdf`, 2000.

[Wie00] Hartmut Wiesenmaier. Ein voll parametrisiertes Modulsystem für Scheme. Master's thesis, Universität Tübingen, Wilhelm-Schickard-Institut für Informatik, 2000.

[XSA01] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating the proof of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.

Index