Hygiene for the Unhygienic

Hygiene-Compatible Macros in an Unhygienic Macro System

Pascal Costanza and Theo D'Hondt
(Vrije Universiteit Brussel
Pascal.Costanza@vub.ac.be and Theo.D'Hondt@vub.ac.be)

Abstract: It is known that the essential ingredients of a Lisp-style unhygienic macro system can be expressed in terms of advanced hygienic macro systems. We show that the reverse is also true: We present a model of a core unhygienic macro system, on top of which a hygiene-compatible macro system can be built, without changing the internals of the core macro system and without using a code walker. To achieve this, the internal representation of source code as Lisp s-expressions does not need to be changed. The major discovery is the fact that symbol macros can be used in conjunction with local macro environments to bootstrap a hygiene-compatible macro system. We also discuss a proof-of-concept implementation in Common Lisp and give some historical notes.

1 Introduction

Macros are local program transformations triggered explicitly in the source code of a program. Since their introduction into Lisp in 1963 [Hart 63], they have found their way into many Lisp dialects, since Lisp is especially attractive for macros due to its homoiconic nature: The source code of Lisp programs is constructed from s-expressions, that is lists, symbols and (literal) values, which are all core data types of Lisp itself [Kay 69, McIlroy 60]. Lisp macros can therefore simply be expressed as functions that map s-expressions to s-expressions.

Since the initial macro systems for Lisp have operated on "raw" s-expressions, variable names that are introduced and/or referenced in the result of a macroexpansion are susceptible to inadvertent capture by introductions and/or references in the surrounding code of the macro invocation. There are essentially two kinds of such inadvertent variable capture, and Graham introduced the terms *free symbol capture* and *macro argument capture* to refer to them [Graham 93].

Macro argument capture is straightforward to prevent: A macro just has to make sure that variable names it introduces are unique and cannot be inadvertently captured by other code. For that purpose, most Lisp dialects provide a `gensym` function that generates symbols that are guaranteed to be unique: Such symbols cannot be accidentally typed in as regular source code tokens, and consecutive invocations of `gensym` are guaranteed to yield different symbols.

However, traditionally the kind of macro system sketched so far does not provide principled solutions for the case of free symbol capture. Consider the following code fragment:

```
(let ((x 42))
  (macrolet (((foo) 'x))
    (let ((x 4711))
      (foo))))
```

Here, the local macro `foo` presumably wants to expand into a reference to the outer `x` variable. However, the invocation of `foo` in this code fragment will eventually expand into a reference of the inner `x`, making the overall code fragment evaluate to 4711. This is a very compact example of free symbol capture.

A number of workarounds are suggested in the literature for inadvertent variable capture, like naming conventions, rearranging the results of macroexpansion, and so on. See [Graham 93] for a comprehensive overview. However, especially the solutions for free symbol capture are ad hoc and do not generalize well. To further complicate matters, macros with intentional variable capture are not uncommon, and should therefore be expressible.

These issues have led to extensive research on macro hygiene especially in the Scheme community: *Hygiene-compatible macro systems* provide additional operators to help avoiding free symbol capture manually [Bawden and Rees 88, Clinger 91a]. *Hygienic macro systems*, on the other hand, ensure that locally visible bindings are automatically respected, and add means for intentionally breaking macro hygiene [Dybvig et al. 92]. This eases expressing simple macros compared to traditional macro systems, but complicates more involved macros, because the latter approach differentiates between surface syntax, which is still represented as s-expressions, and internal representation of source code in terms of *syntax objects*. Effectively, the homoiconicity of traditional Lisp macros is lost and, in some cases, code fragments have to be manually mapped between the different representations, for example to intentionally break macro hygiene.

It has indeed been suggested that in order to support macro hygiene, the internal representation of source code has to be changed. For example, Rees discusses a few alternatives for implementing hygienic macro systems, which all rely on introducing new data types for the internal representation of source code that differ from the data types used for the surface syntax [Rees 93]. On the other hand, Clinger claims that "if a macro needs to refer to a global variable or function [...], then it is quite impossible to write that macro reliably using the Common Lisp macro system" [Clinger 91b]. Since Common Lisp's macro system is modelled after the traditional Lisp-style approach sketched above, this seems to suggest, in other words, that an unhygienic macro system cannot support macro hygiene for both macro argument capture and free symbol capture.

In this paper, we make the following contributions.

– It is known that the essential ingredients of an unhygienic macro system can be expressed in terms of advanced hygienic macro systems [Sperber et al. 07]. We show that the reverse is also true: The essential operators of hygiene-

compatible macro systems, as discussed in the literature [Clinger 91a], can be expressed in terms of an advanced unhygienic macro system.

- We show that for this, the internal representation of source code in the form of s-expressions does not need to be changed. The major discovery is the fact that symbol macros can be used in conjunction with local macro environments to bootstrap a hygiene-compatible macro system.
- We present an implementation of our approach in Common Lisp that does not require a code walker and has a fully portable implementation.

2 An Unhygienic Macro System

In this paper, we use an effect-free subset of Scheme for developing both a core unhygienic macro system, as well as the hygiene-compatible macro system built on top in the next section to back our claims. We use Scheme to be able to focus on the essential elements of our approach before discussing a more complete implementation in Common Lisp in Section 4. For most of the constructs used in this paper, the definitions given in any of the recent Scheme reports are sufficient, except for `gensym`, which is not part of any Scheme report, but has been characterized in Section 1 and is provided by many Scheme implementations.

Both our unhygienic and hygiene-compatible macro systems correctly expand the arguments to `set!` in user programs, but themselves do not use side effects in their expansion algorithms. We do not provide syntactic sugar for destructuring macro arguments and constructing resulting s-expressions, since this does not affect the core issues addressed in this paper. We do use (simple forms of) quasiquotation in our macro systems and in example macro definitions, but we consider this part of the metalanguage which is assumed to be manually translated into invocations of `list`, `cons` and `quote`. An integration of full quasiquotation is extensive but straightforward. Finally, the macro system introduced in this section only handles local macro definitions. A generalization to global definitions is straightforward and discussed in Section 4.

2.1 Required Elements

The macro system in this section provides the following elements which are required to build a hygiene-compatible macro system on top in Section 3.

List macros are regular macros, as for example used in Section 1. In the literature, they are typically called just *macros*, but we want to explicitly distinguish them here from symbol macros. *Symbol macros* are macros which define the expansion of symbol forms. They are, for example, part of ANSI Common Lisp [ANSI 94] and R6RS Scheme (there called *identifier macros* [Sperber et al. 07]).

Both list and symbol macros are introduced as *local macros*, which are affected by other local macros of the surrounding scope. To illustrate the latter, consider the following hypothetical code fragment, where a local list macro `foo` is defined:

```
(let ((x 42))
  (macrolet (((foo) (if (< x 50) '(print #t) '(print #f)))) ;; buggy
    (foo)))
```

In lexically scoped Lisp dialects, we would expect that the macro definition sees the variables from the surrounding code (like `x`). However, one important goal of macro systems is that macros can be fully expanded at compile time, before a program is actually executed. In other words, macro definitions cannot see runtime bindings of local variables, so the definition of `foo` above is invalid.

However, things are different for macro definitions in the surrounding code:

```
(macrolet (((x) 42))
  (macrolet (((foo) (if (< (x) 50) '(print #t) '(print #f)))) ;; correct
    (foo)))
```

Since the macro `x` in this version is also available at compile time, `foo` can indeed see and use it. The local invocation of `(foo)` thus expands into `(print #t)`.

Finally, we require low-level functions with which macros can be expanded explicitly, like Common Lisp's `macroexpand`. They are typically used for interactively testing macro definitions, but they also have uses in advanced macro programming. Due to local macro definitions, however, such low-level macroexpansion functions require representations of local macro environments to be passed.

To summarize, we require the following elements:

- List and symbol macros.
- Local macros, which are affected by surrounding local macros.
- Macro expansion functions which operate on local macro environments.

2.2 A Model of Macroexpansion

In our macro system, macro environments are represented as assocation lists that map macro names to expansion functions. A macro name is either a list with one element, the symbol that was given as the name for a list macro, or just a symbol that names a symbol macro. Macro expansion functions take two parameters: the form to be expanded and a macro environment. Macros are introduced using an `expander-let` form. Consider the following local macro definition fragments:

```
(expander-let
  (((foo) (lambda (form expanders) ... 1 ...)))
  (expander-let
    ((bar   (lambda (form expanders) ... 2 ...)))
    (expander-let
      (((baz) (lambda (form expanders) ... 3 ...)))
      ... enclosed code ...)))
```

These definitions create the following local macro environment:

```
(((baz) ... function 3 ...)
 (bar   ... function 2 ...)
 ((foo) ... function 1 ...))
```

Macro environments list inner before outer definitions, to aid `assoc` finding the innermost macro definition for a given name. Note that `foo` and `baz` are list macros, while `bar` is a symbol macro. Based on this data structure, we can now define the core macro system in Figure 1. It consists of three mutually recursive functions `expand-once`, `expand` and `expand-all`, and three helper functions `bind-expander`, `flatten-params` and `remove-expanders`. Each of the three expansion functions takes a representation of a macro environment and a form to be expanded as parameters. For convenience, these functions are curried.

The function `expand-once` performs one step of macro expansion, based on the given macro enviroment, in case it successfully determines that the passed form is indeed a macro invocation. If it is not a macro invocation, the form is simply returned without change. The function `expand` repeatedly invokes `expand-once` on the passed macro environment and form until the consecutive forms yielded by `expand-once` are not changed anymore. This ensures that in the end, the resulting form is not a macro invocation anymore, but represents either a literal value or a core language construct. The function `expand-all` initially calls `expand` on the passed macro environment and form. It then analyzes the form to determine whether any of the subforms of the resulting form require further macroexpansion, in case the resulting form is a list. There are five cases:

- If the form is a quoted form, it is returned unchanged.
- If it is a sequence, conditional or assignment (`begin`, `if`, or `set!`), the remaining elements are further expanded.
- If the form is an `expander-let`, a new local macro environment is created and the corresponding subforms are expanded with that new environment.
- If the form is a lambda form, the parameter list of the lambda form remains unchanged. However, a new local macro environment is created in which all symbol and list macro definitions are removed that have the same names as the parameters of the lambda form. This ensures that such macro definitions are properly shadowed by local variables. All subforms of the body of the lambda form are then expanded in that new macro environment.
- Otherwise, the form is a function application. In that case, each element of the list that represents the function application is further expanded.

Except for the `expander-let` and the `lambda` cases, all local macro expansions are performed with the same environment as initially passed to `expand-all`.

```
(define expand-once
  (lambda (expanders)
    (lambda (form)
      (let ((binding
              (or (and (symbol? form) (assoc form expanders))
                  (and (pair? form) (assoc (list (car form)) expanders)))))
        (cond (binding ((cadr binding) form expanders))
              (else form))))))

(define expand
  (lambda (expanders)
    (letrec ((local-expand
               (lambda (form)
                 (cond ((or (symbol? form) (pair? form))
                        (let ((new-form ((expand-once expanders) form)))
                          (cond ((eq? form new-form) form)
                                (else (local-expand new-form)))))
                       (else form)))))
      local-expand)))

(define bind-expander
  (lambda (expanders)
    (lambda (spec)
      (list (car spec) (eval ((expand-all expanders) (cadr spec)))))))

(define flatten-params
  (lambda (spec)
    (cond ((null? spec) '())
          ((symbol? spec) (list spec))
          (else (cons (car spec) (flatten-params (cdr spec)))))))

(define remove-expanders
  (lambda (specs expanders)
    (cond
      ((null? expanders) '())
      ((or (and (symbol? (car expanders)) (member (car expanders) specs))
           (and (pair? (car expanders)) (member (caar expanders) specs)))
       (remove-expanders specs (cdr expanders)))
      (else (cons (car expanders) (remove-expanders specs (cdr expanders)))))))

(define expand-all
  (lambda (expanders)
    (letrec
      ((local-expand-all
        (lambda (form)
          (let ((form ((expand expanders) form)))
            (cond
              ((pair? form)
               (case (car form)
                 ((quote) form)
                 ((begin if set!)
                  `(,(car form) ,@(map local-expand-all (cdr form))))
                 ((expander-let)
                  (let ((new-expanders
                          (map (bind-expander expanders) (cadr form))))
                    `(begin ,@(map (expand-all (append new-expanders expanders))
                                   (cddr form)))))
                 ((lambda)
                  (let* ((params (flatten-params (cadr form)))
                         (new-expanders (remove-expanders params expanders)))
                    `(lambda ,(cadr form)
                       ,@(map (expand-all new-expanders) (cddr form)))))
                 (else (map local-expand-all form))))
              (else form))))))
      local-expand-all)))
```

Figure 1: The core unhygienic macro system of Section 2.

The function `bind-expander` is used for creating an entry in a macro environment. It is passed an environment of the macros that are considered to be in scope for the macro definition in question, and a specification describing that macro definition. It is either of the form `((name) (lambda ...))` for list macros, or `(name (lambda ...))` for symbol macros. This specification is converted by fully expanding the respective lambda form in the passed macro environment using `expand-all`, and then using `eval` to convert it into a function. Since the macro expansion function does not see local (runtime) variables, it is sufficient to evaluate the lambda form in a predefined global environment.

The `expander-let` case in `expand-all` uses `bind-expander` for creating new local macro environments and creates a new sequence form (with `begin`) that contains the subforms from the `expander-let` form covered by the new macro definitions, fully expanded in the newly created macro environment.

The function `flatten-params` takes a parameter list, as accepted by Scheme lambda expressions, and turns it into a flat list of parameter names. The function `remove-expanders` takes a flattened parameter list and a macro environment as parameters and returns a new macro environment in which all occurrences of symbol and list macro definitions having the same name as any of the names in `specs` are removed. The `lambda` case in `expand-all` uses `flatten-params` and `remove-expanders` for creating new local macro environments and creates a new lambda form that contains the subforms from the original lambda form, fully expanded in the modified macro environment.

3 Bootstrapping Support for Macro Hygiene

The macro system presented in the previous section is still unhygienic. To illustrate the essential idea of how to build a hygiene-compatible macro system on top, recall the example for free symbol capture from the introduction in Section 1. We can actually solve it by simply renaming one of the variables manually to make the code fragment evaluate to 42:

```
; (1) Manual renaming.
(let ((y 42))
  (macrolet (((foo) 'y))
    (let ((x 4711))
      (foo))))
```

This is indeed one of the proposed workarounds for avoiding free symbol capture in unhygienic macro systems. However, this is unsatisfactory because we would like to be able to choose names freely everywhere in the code. What we actually need is an operator `alias` that gives us a reference to a variable in the current lexical scope that cannot be inadvertently captured:

```
; (2) Using aliases.
(let ((x 42))
  (expander-let (((foo) (lambda _ (alias x))))
    (let ((x 4711))
      (foo))))
```

The core idea of the hygiene-compatible macro system introduced in this section is indeed that whenever a variable is introduced by a programmer, this actually leads to the introduction of two variables: One 'external' symbol macro that has the original name chosen by the programmer, and one 'internal' variable that has a unique name, as generated by `gensym`, that carries the actual variable binding. The external symbol macro is defined such that each reference to the original variable name in scope expands into a reference to the correct variable. Additionally, we can introduce the desired `alias` operator which yields internal names to unambiguously refer to correct variable bindings. Effectively, our hygiene-compatible macro system works by automating the renaming shown in code example `(1)`. Code example `(2)` now expands into something like this:[1]

```
; (3) Expanded form of example (2).
(let ((#:sym01 42))
  (expander-let ((x (lambda _ '#:sym01)))
    (expander-let (((foo) (lambda _ '#:sym01)))
      (let ((#:sym02 4711))
        (expander-let ((x (lambda _ '#:sym02)))
          #:sym01)))))
```

3.1 Generating Aliases

Figure 2 shows the additional definitions that are needed on top of the unhygienic macro system from the previous section to make that automatic renaming work. It defines two helper functions: The function `create-alias-formals` takes a parameter list as used in Scheme lambda forms and generates a congruent list, where each occurrence of a variable name is replaced by a unique symbol generated by `gensym`. The function `create-alias-expanders` takes two such parameter lists, one with external variable names and one with corresponding internal names as created by `create-alias-formals`, and creates binding forms suitable for being embedded in an `expander-let` form. Those binding forms map external variable names to lambda forms that ignore their parameters and simply return the quoted internal variable names.

Using these two helper functions, we can define two new macros `alias` and `alambda`. The `alias` macro yields a quoted internal name for an external name by simply performing one step of macro expansion on its parameter. The `alambda` macro expands into a lambda form where the parameter list is replaced by the

[1] `expander-let` introductions are actually removed after they have been 'consumed' in `expand-all` (see Figure 1). However, for clarity we have left them in in this example.

```
(define create-alias-formals
  (lambda (spec)
    (cond ((null? spec) '())
          ((symbol? spec) (gensym))
          (else (cons (gensym) (create-alias-formals (cdr spec)))))))

(define create-alias-expanders
  (lambda (spec alias-spec)
    (cond
      ((null? spec) '())
      ((symbol? spec) (list (list spec `(lambda _ (quote ,alias-spec)))))
      (else (cons (list (car spec) `(lambda _ (quote ,(car alias-spec))))
                  (create-alias-expanders (cdr spec) (cdr alias-spec)))))))

(expander-let
  (((alias)   (lambda (form expanders)
                `(quote ,((expand-once expanders) (cadr form)))))
   ((alambda) (lambda (form expanders)
                (let* ((alias-formals (create-alias-formals (cadr form)))
                       (alias-expanders
                        (create-alias-expanders (cadr form) alias-formals)))
                  `(lambda ,alias-formals
                     (expander-let ,alias-expanders ,@(cddr form))))))
   ((alet)    (lambda (form expanders)
                `((alambda ,(map car (cadr form)) ,@(cddr form))
                  ,@(map cadr (cadr form))))))
  ...)
```

Figure 2: The hygiene-compatible macro system of Section 3.

result of passing it to `create-alias-formals`, and the body is wrapped by an `expander-let` mapping external to internal names. New hygiene-compatible binding forms can now be expressed in terms of `alambda`. As an example, `alet` is defined in terms of `alambda` in Figure 2 in the usual way.

We can now express our example by embedding the following code fragment in these macro definitions. This version indeed evaluates to 42:

```
(alet ((x 42))
  (expander-let (((foo) (lambda _ (alias x))))
    (alet ((x 4711))
      (foo))))
```

The required elements listed in Section 2.1 that are provided in the unhygienic macro system of the previous section are used to build the hygiene-compatible macro system in this section as follows:

- Symbol macros are used to map from external to internal variable names.
- Local macros are affected by (expanded using) outer macros. This allows defining `alias` as a higher-order macro, which is expanded at compile time.
- Local macro environments and low-level macro expansion functions enable `alias` to look up internal variable names for the respective scopes.

The hygiene-compatible macro system presented in this section is fully layered on top of the unhygienic macro system in the previous section. Especially, there is no need for walking the code embedded in a macro definition to ensure that external variable names are correctly mapped to internal ones. Consequently, the hygiene-compatible macro system does not need any knowledge about the core language that is processed by the core unhygienic macro system, but relies on the fact that the core macro system already correctly distinguishes core language constructs from macros. In contrast, traditional algorithms for supporting macro hygiene have to explicitly walk code embedded in macro definitions [Rees 93]. As a consequence, they have to be aware of the core language constructs of the underlying language, so such algorithms have to be intimately tied to the compiler of the core language.

Furthermore, our hygiene-compatible macro system does not require any additional data structures for representing variables and recording their syntactic levels, as is typically done in traditional hygienic and hygiene-compatible macro systems [Rees 93]. Instead, we exclusively use plain symbols for representing variables, which effectively leads to a 'flattening' of all variables in the result of `expand-all`, independent of whether a variable is introduced by the programmer or by a macro, and independent of the stage at which a variable is introduced.

4 Integration into Common Lisp

As a proof of concept, we have implemented a full version of a hygiene-compatible macro system in Common Lisp following the approach in Section 3.[2] To achieve this, all binding forms (`defvar`, `defun` `defmacro`, `let`, `let*`, `flet`, `macrolet`, and so on) have to be reimplemented in a way similar to `alambda` and `alet`, so that they can generate the necessary mappings from external to internal names. The 'internal' names for global definitions cannot be uninterned symbols,[3] so they are symbols with the name of their respective package prepended and interned in a dedicated package: As long as that package is not manipulated by user code, it thus guarantees uniqueness for global names. To keep things manageable, we have not reimplemented all of Common Lisp, but restricted ourselves to ISLISP, which is mostly a small but non-trivial subset of Common Lisp [ISO 97].

On the one hand, this implementation is feasible since Common Lisp provides all of the required elements listed in Section 2, including "list" macros and symbol macros, local macros affected by surrounding macros, and macro expansion functions which operate on local macro environments. Although macros are specified differently from the `expander-let` forms used in this paper (using `macrolet`), it is still also possible to access the local macro environment as part of the macro's parameter list via the `&environment` keyword.

[2] Download available at http://p-cos.net/core-lisp.html
[3] to ensure they can be externalized during file compilation

On the other hand, we are faced with two additional technical challenges: Whereas Scheme uses a single namespace for values, Common Lisp and ISLISP provide different namespaces for variables, functions, block names, and so on. This requires different alias operators for the different namespaces that can potentially be locally rebound, that is, `alias`, `function-alias`, and `block-alias`.[4] However, apart from minor differences, the approach for mapping external to internal names is always essentially the same. Secondly, although providing access to local macro environments, ANSI Common Lisp does not provide any operators for accessing their entries. However, it provides `macroexpand-1` and `macroexpand` (as equivalents to `expand-once` and `expand`) that take such macro environments as parameters. In order to provide mappings from external to internal names, we have to rebuild the macro environments as discussed in this paper on top of these low-level mechanisms.

In spite of these technical challenges, we have been able to preserve the essential characteristics of the hygiene-compatible macro system presented in this paper. Especially, it is a mere layer on top of Common Lisp's unhygienic macro system, does not require a code walker and has a fully portable implementation. Additionally, Common Lisp's package system allows the reuse of the same names for binding forms as the original ones provided by Common Lisp, by defining our own package, shadowing the original Common Lisp binding forms, and reimplementing them as described above.

This esentially means that we have built our own Lisp dialect on top of Common Lisp (*HCL* for *Hygiene-compatible Lisp*). A question that arises is whether and how HCL can use Common Lisp libraries and vice versa. To answer this question constructively, we make the following assumptions: ANSI Common Lisp specifies that redefining or lexically rebinding symbols exported from the `COMMON-LISP` package has undefined consequences (Section 11.1.2.1.2 in [ANSI 94]). We assume that Common Lisp libraries therefore indeed do not redefine or lexically rebind such symbols, which ensures that macros specified in ANSI Common Lisp, and therefore exported from the `COMMON-LISP` package, always see the correct bindings of predefined variables and functions. We furthermore assume that Common Lisp libraries do not redefine or lexically rebind symbols from any other packages either. In other words, we assume that programmers of Common Lisp libraries have indeed used the known workarounds and measures to protect their macros from inadvertent variable capture.

In such a case, exporting definitions from packages implemented in HCL and importing them into Common Lisp code does not pose any problems: Symbols from HCL will not be redefined or rebound in Common Lisp code, so they will not be replaced with bindings without the necessary mappings from external to internal names that are necessary for HCL's aliasing operators to work correctly.

[4] For example, classes and go tags cannot be locally rebound.

Definitions exported from Common Lisp libraries and imported into HCL pose a more serious challenge: A HCL programmer expects to be able to use aliasing to protect against free symbol capture, but aliasing does not work on symbols imported from Common Lisp libraries, because they do not provide the necessary mappings from external to internal names. The solution is that HCL packages *never* import symbols from Common Lisp libraries. Instead, we have provided operators for importing *definitions* from Common Lisp packages, which define new symbols in HCL packages that map to original symbols in Common Lisp packages. Consider the following example:

```
(import-variable pi common-lisp:pi)
```

This example expands into the following code:

```
(progn (define-symbol-macro pi common-lisp:pi)
  ...)
```

The omitted code contains the necessary actions to ensure that macro environments "know" that `common-lisp:pi` is the 'internal' name for `pi`. HCL provides similar operators for importing functions, symbol macros and macros.

Another important case are Common Lisp macros that create new local bindings for some code body. For example, Common Lisp's `defmethod` macro creates the local function `call-next-method` whose name is a symbol exported from the `COMMON-LISP` package. To ensure that local HCL macros can alias such bindings introduced locally by Common Lisp macros, such symbols should not be imported from Common Lisp packages either. Instead, HCL provides operators `with-imported-variables`, `with-imported-functions`, and so on, to provide local mappings from HCL names to Common Lisp names, which are again considered 'internal' names for the purpose of the HCL macro system. So in the following example, `call-next-method` has a corresponding local mapping:

```
(defmethod foo ((x integer) (y integer) (z integer))
  (with-imported-functions ((call-next-method common-lisp:call-next-method))
    ...))
```

These import operators for both global and local definitions from Common Lisp libraries cover the most important cases when interoperating between HCL and Common Lisp. One case that is still not covered are Common Lisp macros that compute new names for automatically generated bindings (like the various functions generated by a `defstruct` macro). HCL does not provide a straightforward solution here. Instead, more effort is necessary in a separate library to define wrappers for such macros that ensure that the new names are interned in an external package, and then imported with the operators discussed above.

Another special case are keywords exported from Common Lisp's *keyword* package that are specified to evaluate to themselves: If used in HCL code, they retain their special status and should not be redefined or rebound. HCL's `nil` is even more special in that it loses its equivalence to `'nil` inside HCL code.

5 History and Related Work

Since the introduction of macros into Lisp in 1963 [Hart 63], macro systems have been continuously improved in various Lisp dialects. Pitman gives a summary of the then state of the art in an overview paper in 1980 [Pitman 80], shortly before the initial specification of Common Lisp was commenced.

Based on the good experiences with lexical scoping in Scheme, one goal for Common Lisp was to define equally powerful lexically scoped constructs, like `let`, `let*`, `flet`, `labels`, `block`, and so on. One of the additions was a lexically scoped `macrolet` which, to the best of this author's knowledge, did not exist in previous Lisp dialects. The introduction of `macrolet` made a representation of local macro environments necessary, as well as a change to `macroexpand` to accept such local macro environments as an additional parameter. The function `macroexpand` itself already existed in previous Lisp dialects [Pitman 80].

Symbol macros, on the other hand, were introduced much later, as part of the Common Lisp Object System: It was considered desirable to enable access to fields in objects in a way similar to that of other object-oriented languages, without the need to mention the (implicit) `this` or `self` reference. In order to avoid code walkers for this purpose, `symbol-macrolet` was introduced in 1988 as part of the CLOS specification [Bobrow et al. 89].

Kohlbecker's seminal work started the research on macro hygiene in 1986 [Kohlbecker et al. 86] - after the introduction of `macrolet` and macro environments in Common Lisp, but before `symbol-macrolet`. Although hygienic macro systems were proposed for Common Lisp, they were not adopted, so research on macro hygiene continued almost exclusively in Scheme. Housel gives an overview of hygienic macro expansion in a series of usenet postings [Housel 93].

The algorithm in [Clinger and Rees 91] is a refinement of Kohlbecker's work. That algorithm performs hygienic macro expansion by way of renaming identifiers when they are newly introduced in macro definitions. This covers both identifiers used for new local variable bindings as well as new free identifiers that are supposed to refer to bindings in the lexical scope of the macro definition. A low-level, hygiene-compatible macro facility is described in [Clinger 91a], and was the basis for an implementation of the hygienic macro system described in [Clinger and Rees 91]. The hygiene-compatible macro system presented in this paper is very similar to the one described in [Clinger 91a]: In our system, 'external' potentially ambiguous identifiers can be turned into 'internal' unique ones by way of `alias`, whereas in their system, `rename` is used for more or less the same purpose. However, in their system, the mapping from potentially ambiguous to guaranteed unique names is not created when variable bindings are introduced (like with `alambda` or derived forms in our system), but is generated by `rename` after the fact as soon as macros introduce new identifiers as part of macroexpansion. Macroexpanded code is further processed in a special lexical environment

that maps the renamed identifiers back to the bindings of the original identifiers in the respective lexical environments of the macro definitions. The algorithm described in [Clinger and Rees 91] recognizes core language constructs of the underlying language in order to work correctly, and thus must be integrated with the compiler or interpreter for that language.

The fact that `symbol-macrolet` did not exist at the time when research on macro hygiene started may be a reason why the potential of using it for resolving macro hygiene issues was not recognized. Symbol macros have been explored in the context of Scheme much later, by Waddell in 1999 [Waddell and Dybvig 99], there called identifier macros, and have been adopted as part of R6RS Scheme only recently [Sperber et al. 07]. The fact that symbol macros can be used in conjunction with local macro environments to bootstrap a hygiene-compatible macro system is the major discovery of this paper.

6 Conclusions and Future Work

Macro argument capture and free symbol capture are sometimes compared to the issues of dynamic scoping. In Lisp dialects where variables are dynamically scoped by default, a new variable binding may inadvertently capture another one with the same name that is needed by a function to be evaluated further down the call chain. Lexical scoping is essential to ensure that closures can close over the variables visible at their definition sites. It is probably impossible to resolve such nameclashes otherwise without manually reimplementing lexical scoping. This paper shows that the case for macro hygiene is a different one, by constructing the essential ingredients of a hygiene-compatible macro system as a mere layer on top of an advanced unhygienic one. The difference is due to the fact that the different syntactic scopes are not needed in the fully macroexpanded code, but that macroexpansion can 'flatten' all identifiers, while separate lexical environments need to be maintained for closures at runtime.

The hygiene-compatible macro system presented in this paper works because macros can expand into definitions of local symbol macros, and in this way control the further expansion of embedded code fragments. Macros implemented in expansion-passing style provide a different approach for controlling such further expansions [Dybvig et al. 88], and it would be interesting to see whether the approach presented here can be reimplemented using expansion-passing style.

Our approach enables implementing high-level operators for inspecting and manipulating macro environments at compile time that are similar to operators as defined in advanced hygienic and hygiene-compatible macro systems like syntactic closures [Bawden and Rees 88], as we will discuss in a follow-up publication. It has been shown that hygienic macro systems can be implemented on top of syntactic closures [Hanson 91], but it remains future work to show that we can do this with our hygiene-compatible macro system as well.

References

[ANSI 94] ANSI/INCITS X3.226-1994. *American National Standard for Information Systems - Programming Language - Common Lisp*, 1994.

[Bawden and Rees 88] Alan Bawden and Jonathan Rees, Syntactic Closures. *Conference on Lisp and Functional Programming*, July 1988, ACM Press.

[Bobrow et al. 89] Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, and David Moon, The Common Lisp Object System Specification. *Lisp and Symbolic Computation*, Vol. 1, No. 3-4, January 1989, Springer Verlag.

[Box 98] Don Box, *Essential COM*. Addison-Wesley, 1998.

[Clinger and Rees 91] William Clinger and Jonathan Rees, Macros That Work. POPL'91, ACM Press.

[Clinger 91a] William Clinger, Hygienic macros through explicit renaming. *Lisp Pointers* IV(4), December 1991, ACM Press.

[Clinger 91b] William Clinger, Macros in Scheme. *Lisp Pointers* IV(4), December 1991.

[Dybvig et al. 88] R. Kent Dybvig, Daniel Friedman, and Christopher Haynes, Expansion passing style: A general macro mechanism. *Lisp and Symbolic Computation*, Vol. 1, No. 1, June 1988, Springer Verlag.

[Dybvig et al. 92] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman, Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4), 1992, Springer Verlag.

[Gosling et al. 05] James Gosling, Bill Joy, Guy Steele Jr., and Gilad Bracha, *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[Graham 93] Paul Graham, *On Lisp*. Prentice-Hall, 1993.

[Hanson 91] Chris Hanson, A Syntactic Closures Macro Facility. *Lisp Pointers* IV(4), 9-16, December 1991, ACM Press.

[Hart 63] Timothy Hart, MACRO Definitions for LISP. AI Memo 57, MIT, 1963.

[Housel 93] Peter Housel, An introduction to macro expansion algorithms, parts 1-4. `http://www.cs.indiana.edu/pub/scheme-repository/doc/misc/macros-01.txt` - macros-04.txt.

[ISO 97] ISO/IEC 13816:1997. *Programming Language ISLISP*, 1997.

[Kay 69] Alan Kay, *The Reactive Engine*, PhD thesis, University of Utah, 1969.

[Kelsey et al. 98] Richard Kelsey, William Clinger, Jonathan Rees (eds.). *Revised^5 Report on the Algorithmic Language Scheme*. Higher-Order and Symbolic Computation, Vol. 11, No. 1, September, 1998.

[Kohlbecker et al. 86] Eugene Kohlbecker, Daniel Friedman, Matthias Felleisen, and Bruce Duba, Hygienic macro expansion. *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, ACM Press.

[McIlroy 60] M. Douglas McIlroy, Macro instruction extensions of compiler languages. *Communications of the ACM*, Vol. 3, No. 4, April 1960.

[Pitman 80] Kent Pitman, Special Forms in Lisp. *LISP Conference 1980*, ACM Press.

[Rees 93] Jonathan Rees, Implementing lexically scoped macros. *Lisp Pointers*, 1993.

[Sperber et al. 07] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). *Revised^6 Report on the Algorithmic Language Scheme*, 2007.

[Waddell and Dybvig 99] Oscar Waddell and R. Kent Dybvig, Extending the Scope of Syntactic Abstraction. POPL'99, ACM Press.