Writing a best-effort portable code walker in Common Lisp

Michael Raskin[*]
LaBRI, University of Bordeaux
raskin@mccme.ru

ABSTRACT

One of the powerful features of the Lisp language family is possibility to extend the language using macros. Some of possible extensions would benefit from a code walker, i.e. a library for processing code that keeps track of the status of different part of code, for their implementation. But in practice code walking is generally avoided.

In this paper, we study facilities useful to code walkers provided by "Common Lisp: the Language" (2nd edition) and the Common Lisp standard. We will show that the features described in the standard are not sufficient to write a fully portable code walker.

One of the problems is related to a powerful but rarely discussed feature. The `macrolet` special form allows a macro function to pass information easily to other macro invocations inside the lexical scope of the expansion.

Another problem for code analysis is related to the usage of non-standard special forms in expansions of standard macros. We review the handling of `defun` by popular free software Common Lisp implementations.

We also survey the abilities and limitations of the available code walking and recursive macro expansion libraries. Some examples of apparently-conforming code that exhibit avoidable limitations of the portable code walking tools are provided.

We present a new attempt to implement a portable best-effort code walker for Common Lisp called Agnostic Lizard.

CCS CONCEPTS

•Software and its engineering →Macro languages; Software testing and debugging;

KEYWORDS

code walker, macro expansion, code transformation

[*]The author acknowledges support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61361136003) for the Sino-Danish Center for the Theory of Interactive Computation. This work was supported by the French National Research Agency (ANR project GraphEn / ANR-15-CE40-0009).

ACM Reference format:
Michael Raskin. 2017. Writing a best-effort portable code walker in Common Lisp. In *Proceedings of 10th European Lisp Simposium, Vrije Universiteit Brussel, Belgium, April 2017 (ELS2017),* 8 pages.
DOI: 10.5281/zenodo.3254669

1 INTRODUCTION

Much of the power of Common Lisp comes from its extensibility. Abstractions that cannot be expressed by functions can still be expressed by macros; actually, many of the features described in the standard must be implemented as macros.

Whilst macros are powerful on their own, some ways of extending Lisp would benefit from using higher-level code transformation abstractions. For many such abstractions the most natural way to implement them includes code walking, i.e. enumeration of all the subforms of a piece of code, identification of function calls, variable bindings etc. among these subforms, and application of some code transformations. Unfortunately, code walking is complicated and the libraries to perform it are non-portable. Even portable implementations of recursive macro expansion, also known as `macroexpand-all`, are missing.

Guy Steele writes (in the second edition of the book "Common Lisp: the Language" [3]) that Common Lisp implementations are expected to provide all the functionality needed for code walking code analysis tools. The version of the language described in the book includes support for changing and inspecting the lexical environment objects; the book explicitly recommends to use `macroexpand` on the macros in the language core.

Lexical environment objects are defined by the Common Lisp standard [2] are more opaque. Many Common Lisp implementations do include some functions for environment inspection and manipulation, and even some code walking function. Unfortunately, both the naming and the feature set vary between implementations.

We will show that (unlike CL:tL2) the Common Lisp standard does not allow to implement a portable code walker correctly. We suggest an approach that approximates the desired functionality fairly well and remains implementation-agnostic even when doing the implementation-specific workarounds. We present an implementation of this approach, a library called Agnostic Lizard.

2 RELATED WORK

2.1 Portable tools

The `iterate` library [4] provides an alternative iteration construct also called `iterate`. It is more flexible than the standard `loop` macro, and it uses code walking for implementation. It doesn't work exactly as expected for some code, though. The code snippet

```
(iterate:iterate
  (for x :from 1 :to 3)
    (flet
      ((f (y) (collect y)))
      (f x)))
```

works the same as

```
(iterate:iterate
  (for x :from 1 :to 3)
  (collect x))
```

But the following version with a local macro definition will not work

```
(iterate:iterate
  (for x :from 1 :to 3)
    (macrolet
       ((f (y) `(collect ,y)))
       (f x)))
```

The `macroexpand-dammit` library [7] is an attempt to provide implementation of the full recursive macro expansion functionality (`macroexpand-all`) including an optional lexical environment parameter, but it has some bugs. It is not clear if these bugs can be fixed without major changes. For example, both the original version and the freshest known fork [8] return `1` instead of `2` in the following case:

```
(defmacro f () 1)

(defmacro macroexpand-dammit-here
  (form &environment env)
  `(quote
     ,(macroexpand-dammit:macroexpand-dammit
         form env)))

(macrolet ((f () 2))
  (macroexpand-dammit-here
    (macrolet () (f))))
```

Additionally, the `macroexpand-dammit` library includes the `macroexpand-dammit-as-macro` macro that the environment handling system of the Common Lisp implementation. Both the function and the macro versions of the recursive macro expander remove `macrolet` and `symbol-macrolet` forms from the code (replacing them with `progn` if necessary).

As `SICL` [14] aims to be a modular and portable conforming implementation of Common Lisp in Common Lisp, and the standard requires `compile` to do an equivalent of `macroexpand-all`, there probably will be a usable code walker in `SICL` at some point; to the best of our knowledge, there is currently none.

2.2 Implementation-specific tools

Unfortunately, implementation-specific tools often check the name of the Common Lisp implementation to choose the code path (for example, using `#+` reader conditionals to check for the implementation name). Even if two Common Lisp implementations are closely related and have the same names for the environment processing functions, support for these two implementations has to be added separately. This limitation makes some of the implementation-dependent tools not work on some newer implementations (such as Clasp [15]) even when the tool contains all the code needed for supporting the implementation.

On the other hand, if different versions of the same Common Lisp implementation behave in a different way, such checks can lead to breakage on some of the relatively recent versions of a supported Common Lisp implementation.

Richards Waters has described [5, 6] a clean, almost portable implementation of `macroexpand-all` that needs only a single environment-related function to be implemented separately for each of the Common Lisp implementations.

`CLWEB` [9] by Alex Plotnick follows a similar approach for its custom code walker, and the same approach is described as a "proper code walker" in an essay [10] by Christophe Rhodes.

`hu.dwim.walker` [11] is a comprehensive code walking library that uses a lot of implementation-specific functions for inspecting lexical environments etc. Unfortunately, the current versions of some previously supported Common Lisp implementations are not supported because of relatively recent changes in environment handling. Also `macroexpand-dammit` is implemented in such a way that it removes `macrolet` and `symbol-macrolet` from the code completely after expanding the local macros and the local symbol macros.

The `trivial-macroexpand-all` library [13] provides the `macroexpand-all` functionality by wrapping the best function provided by each Common Lisp implementation. Unfortunately, some implementations don't support the lexical environment argument (for example, CLISP [16]). The same approach is used by SLIME (Common Lisp editing and debugging support for Emacs) [12]. Apparently, no more generic code walking functionality is provided in a consistent way by multiple implementations.

3 PROBLEMS

In this section we present a brief overview of the problems that the portable code walkers face.

3.1 Interpretations and violations of the standard

The Common Lisp standard allows Common Lisp implementations to implement some standard-defined macros as additional special operators. But all special operators added instead of macros must also have a macro definition available. This requirement seems to imply that the standard expects the code walkers to succeed if they implement special handling only for the special operators listed in the standard.

In practice many Common Lisp implementations violate this expectation and implement standard macros using non-standard implementation-specific special operators. For example, the `iterate` library contains workarounds for such macros as `handler-bind` and `cond`.

Fortunately, for most macros the current versions of the major implementations provide usable expansions.

3.1.1 Named lambda expressions. A particular macro that is almost always expanded to code with non-standard special forms is `defun`. We have checked six free software Common Lisp implementations (SBCL, ECL, CCL, ABCL, GCL, CLISP); we have found that only GCL expands the `defun` macro into portable code. SBCL and ABCL pass a form starting with `named-lambda` to `function`, ECL does the same but calls the special symbol `lambda-block`, CLISP passes two arguments (the name and the definition) to `function` and CCL does the same but replaces `function` with special form `nfunction`. For example, the expansion produced by SBCL is as follows.

```
(progn
 (eval-when (:compile-toplevel)
   (sb-c:%compiler-defun 'f nil t))
 (sb-impl::%defun 'f
                  (sb-int:named-lambda f
                      (x y)
                    (block f (* x (1+ y))))
                  (sb-c:source-location)))
```

3.1.2 Theoretical worst-case **defun** *implementation.* It seems that an implementation of the `defun` macro that compiles the code at expansion time and puts a literal compiled function object into the expansion does not violate neither the Common Lisp standard nor the description in the "Common Lisp: the Language". While such an implementation could be compliant, it would make code-walking of entire file meaningless without handling the `defun` macro in a special way.

3.2 Environment handling

The Common Lisp standard provides no functions for inspecting or modifying environments. On the other hand, macro functions can receive environment parameters and request macro expansion of arbitrary code using the environment they receive. Unfortunately, sometimes there is no way to construct an environment that would make the macro expansion would work exactly as desired.

Although `*macroexpand-hook*` is described as an expansion interface hook to `macroexpand-1`, the retrieval of the macro function from the environment is done by the standard rules and cannot be overridden.

4 PASSING INFORMATION VIA `MACROLET` AND `MACROEXPAND-1`, AND TWO KINDS OF THE LEXICAL SCOPE FOR MACROS

In this section we will discuss the available options for passing information between the macro expansion functions, the scopes and extents relevant to the different ways of passing information and implications for portable code walkers. The aim of this section is to provide some context and explanation for the technique used in the next section. This technique will be used to construct an example of impossibility of a correct portable recursive macro expansion function that accepts a lexical environment parameter.

The environment handling issue is related to a useful feature of Common Lisp macros. Let's consider a macro function that needs to pass some information to another macro function or another invocation of the same macro function.

Ordinary functions can pass information via the arguments and the return values, or via global variables. A macro function has an extra option. A macro function can wrap its expansion in a `macrolet` form or a `symbol-macrolet` form that defines a macro (symbol macro) not intended to be used directly. This definition will be accessible to all the macro function invocations corresponding to macro invocations inside the lexical scope of the definition. Such a macro function can use the `macroexpand-1` function to access the definition.

It is a bit awkward to describe the scope of such a definition because there are two kinds of lexical scope relevant for macros. There is the normal lexical scope of the macro function when it is defined and there is the lexical scope of the expansion output in terms of expanded code.

Such approach allows, for example, to define a macro that can be nested but limits the depth of its own nesting without code walking:

```
(defmacro depth-limited-macro
  (n-max &body body &environment env)
  (let*
    ((depth-value
       (macroexpand-1
         (quote (depth-counter-virtual-macro))
           env))
     (depth (if (numberp depth-value)
              depth-value 0)))
    (if
      (> depth n-max)
      (progn
        (format *error-output*
          "Maximum macro depth reached.~%")
        nil)
      (progn
        `(macrolet
           ((depth-counter-virtual-macro ()
              ,(1+ depth)))
           ,@body)))))
```

The following code will expand fine:

```
(depth-limited-macro 0
  (depth-limited-macro 1
    :test))
```

but after a small change it will print a warning and expand to nil:

```
(depth-limited-macro 0
  (depth-limited-macro 0
    :test))
```

This example is probably not very useful on its own except as an illustration and as a test case for code walkers.

5 IMPOSSIBILITY OF A GENERAL SOLUTION

In this chapter we present an example where a portable recursive macro expansion cannot guarantee correct expansion and explain why this code is problematic for macro expansion.

Consider the following code

```
(macrolet
  ((with-gensym ((name) &body body)
    `(macrolet ((,name () '',(gensym))) ,@body)))
  (with-gensym (f1)
    (with-gensym (f2)
      (defmacro set-x1 (value &body body)
        `(macrolet ((,(f1) () ,value))
           ,@body))
      (defmacro set-x2 (value &body body)
        `(macrolet ((,(f2) () ,value))
           ,@body))
      (defmacro read-x1-x2 (&environment env)
        `(list ',(macroexpand-1 `(,(f1)) env)
               ',(macroexpand-1 `(,(f2)) env))))))

(defmacro expand-via-function
  (form &environment e)
  `',(macroexpand-all (quote ,form) ,e))

(set-x1 1
  (set-x2 2
    (expand-via-function
      (set-x2 3
        (read-x1-x2)))))
```

If we replace the `expand-via-function` invocation with an `identity` function call, this code will return `(1 3)`. If `macroexpand-all` worked correctly, the unmodified code snippet would return `(list 1 3)`, because the macros named by the symbols returned by `(f1)` and `(f2)` would expand to `1` as defined by `(set-x1 1 ...)` and `3` as defined by the innermost `(set-x2 3 ...)`. Evaluating this expansion will provide `(1 3)`, as expected. Unfortunately, a portable implementation of a `macroxepand-all` function cannot expand such code correctly.

Note that the symbol naming the local macros defined by `set-x1` is not accessible to the `macroexpand-all` function. The `do-all-symbols` function iterates only on the symbols in the registered packages, and `gensym` produces symbols not accessible in any package; and the scope where the symbol was available does not contain the `macroexpand-all` function or even its call. It is also impossible to observe the internals of execution of the macro expansion function for the `read-x1-x2` macro.

But the `macroexpand-all` function needs to call the macro expansion function for `(read-x1-x2)` and pass it some environment. The Common Lisp standard does not provide any way of obtaining the environment except getting the current environment at the call position. We need the lexical environment to depend on the run-time input of the `macroexpand-all` function, so we need to use `eval`. The `eval` function evaluates in the null lexical environment, so the new environment will contain only the entries that the `macroexpand-all` function can name explicitly while constructing the form to evaluate.

Therefore we can either pass the initial environment (which contains the definition set by `set-x1` but also contains an obsolete definition set by `set-x2`), or construct a new environment, which can take into account the innermost `set-x2` invocation but cannot include any macro definition for the macro name defined by `set-x1`. Both options will cause `(read-x1-x2)` to expand to a wrong result.

6 PARTIAL SOLUTIONS

In this section we describe partial solutions that allow to provide `macroexpand-all` and code walking functionality for the simple cases and many of the most complicated ones.

6.1 Hardwiring specific macros

Given that `defun` is expanded into something non-walkable in most Common Lisp implementations, the walker can treat this macro as a special form and implement special logic to handle it. The same approach can be applied to all the macros as soon as an unwalkable expansion is observed in some Common Lisp implementation.

Note that while hardwiring macros can make some applications of a code walker less convenient, it doesn't sacrifice portability. The resulting code will still be legal even on the implementations where the workaround was not needed.

This workaround doesn't fully solve the problem of implementations expanding standard macros to non-compliant code, because a portable program could ask for an expansion of a standard macro and use it in a macro function for some user-defined macro. The following code fragment illustrates the problem:

```
(defmacro my-defun (&rest args)
  (macroexpand `(defun &rest args)))

(macroexpand-all `(my-defun f (x) x))
```

There is also a risk that an implementation would expand a standard macro to some code including a non-standard invocation of another standard macro; if only the second macro is hardcoded, the code walker will fail.

6.2 Recognizing named lambda by name

The most popular approach to the expansion of `defun` includes passing a non-standard argument to `function`. As a list starting with anything but `lambda` and `setf` is a non-standard argument, trying to interpret the symbol name won't break compatibility with an implementation that implements `function` without extensions. Apparently the forms with the first symbol being `named-lambda` or `lambda-block` are handled by the `function` special form in the same way in all the Common Lisp implementations using them.

This workaround relies on extrapolating behaviour of non-standardized forms based on unwritten traditions. The code walker has no way to know whether

```
(function (named-lambda f (x) y))
```

is a function called `f` returning `y` or a function called `y` returning `f`, but there are reasons to believe that the latter interpretation doesn't occur in practice. Here we have an inherent conflict between detecting the walker's failure and portability.

6.3 Macro-only expansion

In most cases the lexical environment for code walking is the lexical environment in the place of the call to the code walker. This situation allows the code walker to be implemented as a macro that does a single step of the expansion and leaves the recursive calls to itself in the expansion. The lexical environment will be handled by the Common Lisp implementation.

A minor drawback of this approach is a necessity for additional processing when the result of code walking should be put into a run-time variable. In other words, this approach requires additional processing to produce a quoted result.

A more significant limitation is related to the use of code walking with callbacks. Macro-only code walking is done in a top down manner, so the callbacks don't have access to the result of processing the subforms.

If the implementation provides a `macroexpand-all` function with an environment parameter, combining a code transformation implemented by a macro-based code walker and the `macroexpand-all` function yields a code walker that can be called as a function.

This approach fully solves the environment handling problem by asking the Common Lisp implementation to handle the environment, but shares the problems related to non-standard code in the expansions of standard macros.

In the following subsections we describe what can be done in order to construct a portable code walking function.

6.4 Start with the top level

Whilst it is impossible to augment a lexical environment in a portable way, it is easy to construct a lexical environment with given entries. So when a form is walked in the null lexical environment, the code walker can guarantee correct environment arguments for all macro expansions.

The limitations related to the expansions of the standard macros still apply in this case, but the environment handling can be made fully correct.

6.5 Guessing which environment to pass

Despite the fact that it is sometimes impossible to construct a correct environment for the call to `macroexpand-1`, there are cases where it is clear which environment to use. In other cases we can try to make a good guess.

If we have started from the null lexical environment, we can always create the environment from scratch. If we haven't yet collected any lexical environment entries that add or shadow any macro (or symbol macro) definitions, we can use the lexical environment initially provided by the caller of the code walker.

In the remaining case any guess can be wrong. We can try to improve our track record by using the initial environment if and only if we expand a macro defined locally in the initial environment. Otherwise we construct a lexical environment using the entries collected while processing the containing forms.

There is no way to check whether a guess is correct, and in some cases like the example presented in the section 5 both options are wrong. Moreover, there is no way for a code walker to check whether a macro expansion function defined outside of the form being walked uses macroexpand.

7 POSSIBLE APPLICATIONS OF CURRENTLY IMPLEMENTATION-SPECIFIC FUNCTIONALITY

7.1 Environment-augmenting functions

Having a macro `with-augmented-environment` that creates a lexical-scope dynamic-extent variable with an environment with specified additions with respect to an initial one would be enough for implementing a code walker following the standard approach currently taken by the implementation-specific walkers.

7.2 Using a recursive macroexpander to build a code walker

If a code walker has access to the `macroexpand-all` function which accepts an environment argument, there are two natural strategies. The first one is to call `macroexpand-all` before code walking, and code walk the expanded code without needing to expand any macros. The second one is to build `with-augmented-environment` using `macroexpand-all`. It can be done using code similar to the following example.

```
(defmacro with-current-environment
    (f &environment env)
  (funcall f env))

(macroexpand-all
  `(let ((new-x nil))
     (macrolet ((new-f (x) `(1+ ,x)))
       (with-current-environment ,(lambda (e) ...))))
  env)
```

7.3 Using environment inspection for constructing augmented environments

Just having a list of all locally defined and shadowed names is enough to construct an augmented environment in a way that indistinguishable using only the standard facilities. The idea of the construction is to use a macro that obtains its environment, passes it to the function we want to call, quotes the result and returns the quoted result as the result of the macro expansion. An invocation of such a macro can be wrapped in to set up the correct environment, and the wrapped code can be evaluated using `eval`. We do the wrapping related to the desired changes first so that these forms are the innermost forms altering the environment. The evaluated code would be similar to the following example.

```
(defmacro with-current-environment
    (f &environment env)
  (funcall f env))

(eval
  `(let ((y-from-environment nil))
     (let ((new-x nil))
       (macrolet ((new-f (x) `(1+ ,x)))
         (with-current-environment
           ,(lambda (e) ...)))))
```

For each symbol we can query whether it defines a local macro (using `macro-function`) or a local symbol-macro (using `macroexpand` as the macro expansion of a symbol macro does not depend on anything). Having a name and a macro expansion function is enough to construct a wrapping `macrolet` form; having a name and an expansion is enough to construct a wrapping `symbol-macrolet` form. If it doesn't define a local macro or a local symbol macro we can use `let` or `flet` to ensure that global definitions are shadowed. Tags and block names cannot be inspected by macros. The list of names in the lexical environment will also be preserved.

Of course, it would be even better to have access to listing all the variable-like names, all the function-like (operator) names, all the tag names and all the block names. Distinguishing variables and symbol macros, and functions and macros can be done in the same way as before.

8 THE AGNOSTIC LIZARD LIBRARY

We present a new library for code walking, Agnostic Lizard. It implements all of the described workarounds.

For code walking and macro expansion it exports two functions and two macros.

The `macroexpand-all` function accepts two arguments, a form and an optional lexical environment object. It tries to perform recursive macroe xpansions and usually succeeds. The `macro-macroexpand-all` macro accepts a form, and expands it in the current lexical environment. The expansion is quoted, i.e. the runtime value of the generated code is the expansion of the initial form.

The `walk-form` function accepts a form, a lexical environment object (required argument, can be `nil`), and the callbacks as the keywords arguments. The callbacks can be:
`:on-every-form-pre` — called before processing each form in an executable position;
`:on-macroexpanded-form` — called for each form after possibly expanding its top operation, the hardwired macros are passed unexpanded;
`:on-special-form-pre` — called before processing a special form or a hardwired macro;
`:on-function-form-pre` — called before processing a function call;
`:on-special-form` — called after processing a special form or a hardwired macro;
`:on-function-form` — called after processing a function call;
`:on-every-atom` — called after processing a form expanded to an atom;
`:on-every-form` — called after expanding each form in an executable position.

The `macro-walk-form` macro accepts the form as a required argument, and the callbacks as keyword arguments. The callbacks have the same semantics as for `walk-form`. The expansion is the result of walking the form in the current lexical environment with the specified callbacks.

8.1 Implementation details

Agnostic Lizard mostly follows the design of the walkers using the implementation-specific environment inspection and manipulation functions. In other words, it starts at the top and recursively calls itself for the subforms, passing the updated environment information. It keeps track of the lexical environment changes in an object, and uses the initial environment only as a fallback. Agnostic Lizard always identifies macro invocations correctly, but it has to use heuristics for choosing what environment to pass. The code walker also tries to guess how to handle non-standard special forms in the expansions.

Agnostic Lizard defines three classes to handle code walking. The `metaenv` class contains the data directly describing the current walking context. It contains the list of defined functions and macros, variables and symbol-macros, blocks, and tags. It also keeps a reference to the Common Lisp implementation specific lexical environment object which has been initially passed by the caller.

This class stores just enough data to implement the basic recursive macro expansion. The `metaenv-macroexpand-all` generic function is used for dispatching the expansion logic. For the forms that are not `cons` forms with a hardwired operator the function calls `metaenv-macroexpand` first. The `metaenv-macroexpand` generic function contains all the decisions about the environment construction used at that step. Then the `metaenv-macroexpand-all` generic function passes the result of macro expansion or the original form to the `metaenv-macroexpand-all-special-form` generic function with the operator of the form as the first parameter. The methods of this generic function contain all the handling of the special operators and the hardwired macros These methods call `metaenv-macroexpand-all` for recursive processing of the subforms. Actually, the methods do little else: they clone create a new `metaenv` object with extra entries for the child forms (if needed), call `metaenv-macroexpand-all` and build the expanded form out of expansions of the subforms.

The class implementing the callbacks for code walking is `walker-metaenv`. It is a subclass of `metaenv`. Objects of the `walker-metaenv` class additionally store the callbacks to allow replacing the form in various stages of expansion. The only non-trivial method defined for this class is for the `metaenv-macroexpand-all` generic function. The method does the same operations as the method for the `metaenv` class, but optionally invokes the callback functions.

The class for macro-only walking is `macro-walker-metaenv`. It extends `walker-metaenv` with the data needed for macro-based walking: a boolean to alternate between creating a macro wrapper for capturing the updated environment and actually expanding the form using the captured environment; a callback for the macro creation step; and a label for letting the callbacks distinguish the parts of the code wrapped for further walking.

The macro-based recursive macro expansion that returns the code evaluated to the expanded form is done using the callbacks provided by the walkers. The callbacks walk the already walked part of the code and perform a transform similar to quasiquotation; they also make sure that the code that has not yet been expanded will be expanded in the correct environment despite such rewrite of the containing code.

8.2 Portability testing

We have tested Agnostic Lizard by checking that various forms return the same value before and after the expansion. We have written a small test suite to check the handling of local macros, and we have used the `iterate` library test suite to get special form coverage. Agnostic Lizard passed these tests when loaded under SBCL, ECL, CCL, ABCL and CLISP.

8.3 Reimplementing access to local variables

As a demonstration of the code walking interface we provide a partial portable reimplementation of the wrappers ensuring access to local variables during debugging [17]. Currently Agnostic Lizard does not provide an interface that would allow a callback to check if the processed form has the same environment as the parent one, so the wrapper saves the references to the lexical environment entries for every form. On the other hand, the macro code walker in the Agnostic Lizard library allows to get the same code-walking based functionality on a wider range of implementations.

The implementation of the wrapper presented in [17] uses `hu.dwim.walker` [11], which is the only implementation of a generic code walker we could find that was compatible with at least two implementations. Unfortunately, it has limited portability because of changes in some of the previously supported Common Lisp implementations, and it removes macro definitions completely. Agnostic Lizard lacks environment object handling features relying on implementation specific functions, but has better portability and preserves macro definitions.

9 CONCLUSION

We have shown that although a portable code walker for Common Lisp is impossible, it is possible to create an incorrect code walker that is wrong less often than the currently available ones.

9.1 Benefits for portable code walkers from hypothetical consensus changes in implementations

Portable code walkers suffer from two issues: non-standard expansions of standard macros and opaqueness of the lexical environment objects.

For most macros we hope that all the implementations with a release in the last couple of years already expand them to code using only standard invocations of macros and special forms. The only currently exceptional macros are `defun`, `defmacro` and `defmethod` that typically use non-standard `named-lambda` forms as arguments to `function` (or something very similar). There are benefits for storing the name of the function, and adopting Alexandria solution of using `labels` and saving the form definition would probably have some implementation cost. It is possible that the current practice goes against the intent of some parts of the Common Lisp standard, but there are non-trivial costs to changing it. It would be convenient to have at least a consensus symbol name and package name for `named-lambda` and `nfunction` without requiring neither to be present. It would also be nice to have an agreement that the expansions of standard macros should only use the special forms that have consensus names.

Opaqueness of lexical environments can be solved by having a consensus name for either augmenting the lexical environments or listing their entries. Listing the entries seems preferable because of the additional applications of such functionality, but using environment modification functions would probably provide better performance.

The `macroexpand-all` function has a recognizable name (although some Common Lisp implementations use different names, and of course there is no consensus package name), and its interface is clear and natural. It also allows implementing environment augmentation. It would still be useful to support listing the names in the lexical environment, but having portable access to a `macroexpand-all` implementation does allow implementing a portable generic code walker.

We believe that a good interface for a more general code walker requires some period of experimentation and evolution of alternative implementations. Therefore it is too early to promote a consensus name and interface for a generic code walker.

9.2 Further related issues that could benefit from consensus naming

In general it would be nice to have all a consensus package name for CLtL2 functionality not included in the ANSI Common Lisp standard in order to have a portable way to check which parts of this functionality are provided by an implementation. Many implementations de facto provide a large part of the difference, but they use different package names and sometimes also change function names.

Having a consensus name for a type for the lexical environment objects would make using generic functions more comfortable. This would allow writing portable generic functions handling both implementation defined lexical environment objects and library-defined enriched environments. This type can coincide with some other predefined type.

9.3 Future directions

Some interesting uses of code walking require processing large amounts of code. For such applications, performance of a code walker can be important. We haven't benchmarked Agnostic Lizard. It seems likely that some optimisations may be needed.

We have mentioned that most implementations expand `defun` to code containing non-standard special operators or non-standard uses of the `function` special form. In most cases such code is easy to analyze by using a few predetermined heuristics. It may be possible to further reduce the impact of this problem by expanding a `defun` form with the function name and argument names obtained by `gensym`, and analyzing where these names get mentioned.

It would be interesting to see how much of the code available via QuickLisp breaks after expansion by Agnostic Lizard. We haven't tried doing it yet.

Different tasks solved by code walking require different interfaces to be provided by the code walker. The callback interface of Agnostic Lizard is currently pretty minimalistic and should be expanded. Any advice and feedback are very welcome.

10 ACKNOWLEDGEMENTS

We would like to thank Irène Durand and the anonymous reviewers for their comments and advice.

REFERENCES

[1] Agnostic Lizard homepage. Retrieved on 30 January 2017 https://gitlab.common-lisp.net/mraskin/agnostic-lizard
[2] American National Standards Institute, 1994. ANSI Common Lisp Specification, ANSI/X3.226-1994. A hypertext version (converted by Kent Pitman for LispWorks) retrieved on 24 January 2017 from http://www.lispworks.com/documentation/HyperSpec/Front/index.htm
[3] Guy L. Steele. 1990. Common Lisp the Language, 2nd Edition. Also retrieved on 24 January 2017 from https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html
[4] Jonathan Amsterdam. Don't Loop, Iterate. Working Paper 324, MIT AI Lab. Also retrieved on 24 January 2017 from https://common-lisp.net/project/iterate/doc/Don_0027t-Loop-Iterate.html Project homepage: https://common-lisp.net/project/iterate/
[5] Richard C. Waters. 1993. Some Useful Lisp Algorithms: Part 2. Tech. Rep. TR93-17, Mitsubishi Electric Research Laboratories. Also retrieved on 24 January 2017 from http://www.merl.com/publications/TR93-17
[6] Richard C. Waters. 1993. Macroexpand-All: an example of a simple lisp code walker. Newsletter ACM SIGPLAN Lisp Pointers. Volume VI Issue 1, Jan.-March 1993.
[7] John Fremlin. 2009. Macroexpand-dammit. Web Archive copy. Retrieved on 24 January 2017. https://web.archive.org/web/20160309032415/http://john.freml.in/macroexpand-dammit
[8] The freshest macroexpand-dammit fork repository. Retrieved on 24 January 2017. https://github.com/guicho271828/macroexpand-dammit
[9] Alex Plotnick. 2013. CLWEB homepage. Retrieved on 24 January 2017. http://www.cs.brandeis.edu/~plotnick/clweb/
[10] Christophe Rhodes. 2014. Naive vs proper code-walking. Retrieved on 24 January 2017. http://christophe.rhodes.io/notes/blog/posts/2014/naive_vs_proper_code-walking/
[11] hu.dwim.walker repository. Retrieved on 24 January 2017. http://dwim.hu/darcsweb/darcsweb.cgi?r=LIVE%20hu.dwim.walker;a=summary
[12] The Superior Lisp Interaction Mode for Emacs (SLIME) project repository. Retrieved on 24 January 2017. https://github.com/slime/slime
[13] trivial-macroexpand-all repository. Retrieved on 24 January 2017. https://github.com/cbaggers/trivial-macroexpand-all
[14] SICL homepage. Retrieved on 24 January 2017. https://github.com/robert-strandh/SICL
[15] Christian E. Schafmeister. 2015. Clasp - A Common Lisp that Interoperates with C++ and Uses the LLVM Backend. In proceedings of ELS2015. Retrieved on 24 January 2017. http://european-lisp-symposium.org/editions/2015/ELS2015.pdf Project repository: https://github.com/drmeister/clasp
[16] GNU CLISP homepage. Retrieved on 24 January 2017. http://www.clisp.org/
[17] Michael Raskin, Nikita Mamardashvili. 2016. Accessing local variables during debugging. In proceedings of ELS2016. Retrieved on 30 January 2017. http://european-lisp-symposium.org/editions/2016/ELS2016.pdf