Writing a best-effort portable code walker in Common Lisp

Michael Raskin, raskin@mccme.ru

Aarhus University → LaBRI, Université de Bordeaux

April 3, 2017
Code walking: what and why

Code walker

· a tool for code analysis and transformation
· enumerates all the subforms in the code

Why?

· Code is data is code (so why not)
· Metaprogramming: Write programs that write programs
· Macros
· Write programs that **rewrite** programs!
Is code walking used in Common Lisp?

- iterate

What else?

- CL-Web
- CL-Cont
- Weblocks
- hu.dwim.walker
- local-variable-debug-wrapper
- macroexpand-dammit
- sexml
- fn
- temporal-functions
- trivial-macroexpand-all

- Some implementations include code walking libraries
- Papers with a placeholder for some implementation-specific function

This may be close to a complete list...
A use of code walking: fn

Why use a code walker: a small example

(fn* (+ _ _)) → (lambda (_ _) (+ _ _))
(fn* (+ _ _1)) → (lambda (_ _1) (+ _ _1))
(fn* (subseq _@ 0 2)) → (lambda (&rest _@) (subseq _@ 0 2))

λ(+ _ _1) → (lambda (_ _1) (+ _ _1))

from README of public domain fn library by Chris Bagley (Baggers)

Code walking is used to find argument names (not quoted symbols or function names or local variables)
Are there reusable code walking libraries?

Implementation-supplied libraries
  · Implement exactly what they promise — correctly
  · API differs

```
hu.dwim.walker
  · Reader conditionals
  · Bit rot in the code for some of the implementations
  · Removes macrolet from code
```

```
macroexpand-dammit
  · Portable
  · Correctness problems
  · Some of them avoidable
  · Removes macrolet from code
```
Is a portable code walker possible in Common Lisp?

Functionality we can specify: a macroexpand-all function

Like macroexpand, but also expand subforms
Should take a lexical environment object as the second parameter (for local macros)
Is a portable macroexpand-all function possible in Common Lisp as defined by ANSI?

No
Portable macroexpand-all function is impossible: environments

Macro expansion functions have access to lexical environment

Can use environment to call macroexpand-1 on arbitrary forms

Very powerful feature — even more than it seems
Tricks with environment

(defmacro depth-limit (max &body body &environment env)
  (let*
    ((depth-value (macroexpand-1 (quote (depth-counter)) env))
      (depth (if (numberp depth-value) depth-value 0)))
    (if (> depth max)
      (progn (format *error-output* "Too deep.~%") nil)
        ~(macrolet ((depth-counter () ,(1+ depth))) ,@body)))

(depth-limit 0 (list (depth-limit 1 :test)))

(depth-limit 0 (list (depth-limit 0 :test)))

No code-walking!)
(macrolet
  ((with-gensym ((x) &body body)
    `(macrolet (((x () ',(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))))))
Portable correct environment handling is impossible

macroexpand-all doesn’t see the names of temporary macros

Lexical environment: pass it as is or build a new one from scratch

You cannot create an entry with the name you do not know
Portable correct environment handling:
ANSI CL and «Common Lisp: the Language» (2nd edition)

«Common Lisp: the Language» has functions to inspect and modify lexical environment objects — enough for code walkers.

What the non-portable code walkers actually do: expand in the given environment, add new entries as needed when descending into special forms.

An alternative option: inspect the initial lexical environment, build new lexical environments, put the entries extracted from the original environment there.

In ANSI Common Lisp standard the lexical environment objects are almost completely opaque.
More troubles: expanding standard macros

Let’s expand the `defun` macro...

```
(defun f (x) x)
```

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

Using SBCL as an example — most implementations do that
Special operator function as described in the standard can’t handle this;
portable walker needs to deal with different `named-lambda` symbol names
Unclear if the standard intended to allow this
Agnostic Lizard

Portable code walking is almost possible

Nobody\(^1\) writes such code with expansions
· Apply heuristics to decide what environment to pass
defun (and defmethod) can be hardcoded
· Not a complete solution — user could expand defun and use the result
· · Apply heuristics to guess what style of function extension is used

Agnostic Lizard:

A code walker
No reader conditionals
Works fine unless a combination of bad events happens

\(^1\)I did — for this talk
Agnostic Lizard

Enumerates forms and calls callbacks:
:on-every-form-pre
:on-macroexpanded-form
:on-special-form-pre
:on-function-form-pre
:on-special-form
:on-function-form
:on-every-atom
:on-every-form

Callbacks can replace the form
Accepts hints about the names, the hints are checked

In QuickLisp; also on GitLab.Common-Lisp.net

gitlab.common-lisp.net/mraskin/agnostic-lizard
Agnostic Lizard anatomy

Three main classes: metaenv, walker-metaenv, macro-walker-metaenv

metaenv: basic walking context
  • Used to define metaenv-macroexpand-all

walker-metaenv: the same, plus callbacks
  • Code walking is implemented as macroexpand-all with callbacks and an option to replace forms in the process

macro-walker-metaenv: the same, plus support for recursive macro invocations instead of recursive expansion calls
  • Environment handling fully correct
  • Some limitations on functionality
Agnostic Lizard — possible future

Would be interesting to try applying to random code in QuickLisp
  · Not sure how to check correctness

Callback interface
  · I did use it for some call tracing
  · · Had to expand it in the process...
  · · Feature requests are treated with gratitude as advice
Impact of environment-related extensions: summary

Let’s hope there are no new and creative defun expansion...

macroexpand-all and with-augmented-environment and more generic code-walking
- Can be used to implement each other
- Not much of a performance penalty

Environment inspection
- Enough to implement macroexpand-all etc.
- Transformation uses eval — may be costly
- More useful for other debugging tasks
Using `macroexpand-all` for environment modification

```lisp
(defun 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)
```
(defmacro eval-with-current-environment
  ((var) &body code &environment env)
  `',(funcall (eval `(lambda (,var) ,@code)) env))

(defun with-metaenv-built-env (obj var code)
  (eval
   (metaenv-wrap-form
    obj
    `(eval-with-current-environment
       (,var) ,@code))))
An operator in an environment can have:

- A visible global macro function
- A local macro function (possibly shadowing a global one)
- A local function shadowing a global macro definition
- None of the above — no definition, or a function (local or global)
- Doesn’t matter which

Variables and symbol macros are similar but simpler
Could we please agree on:

- a common name and package name for named-lambda
  (portable alternative using labels is in alexandria, a compiler macro could expand to the current expansion, but failing that...)

- a common package name for macroexpand-all
  - environment parameter handling can be checked

- a common name and package for environment-inspection functionality

- maybe just a common package name for «Common Lisp: the Language» (2nd ed.) functionality not in the standard
  - most implementations provide most of the functionality already, but package names differ

I don’t ask to provide new functionality — just an alias for what exists
Just in case:

common-lisp-extensions:named-lambda
common-lisp-extensions:nfunction

common-lisp-extensions:macroexpand-all

common-lisp-extensions:list-environment-names
common-lisp-extensions:with-augmented-environment
common-lisp-extensions:with-parent-environment

cltl2:parse-macro
cltl2:function-information
cltl2:variable-information
cltl2:declaration-information
Thanks for the attention

Questions?