Accessing local variables during debugging

Michael Raskin, raskin@mccme.ru Nikita Mamardashvili (Shviller)

April 2016

Context

- Hopefully, all large programs use local variables
- Some programs even use closures
- Interactive debugging benefits from inspection of all the relevant state
- Common Lisp implementations provide access to local variables during debugging

However...

Local variable availability

Local variables can be optimised away by the optimizer. SBCL manual has a special section (in version 1.3.4 it is section 5.4.1) «Variable Value Availability»

But even avoiding triggering any of the listed optimizations isn't always enough.

For example, constant propagation can also optimize variables away.

Brute-force solution

Copy the lexical environment into global variables using dynamic-scope rebindings.

Code-walk the code, notice where lexical environment changes (`let`, `lambda`, `labels`,...) and push the names and the values into a stack with a global name.

How to support modification of variables from a debugging REPL? Currently we save anonymous functions that can return or modify the values. Not sure if there is any other portable way.

Demonstration

- `with-local-wrapper`
- `wrap-rest-of-input`

- `pry-light`
- `pry`
- `pkg-pry`

- `list-locvars`, `list-locfuncs`
- `locvar`, `locfunc`
- `lexenv-stack-cursor-set`

- `push-lexenv-to-saved`

with-local-wrapper

Code-walk the body to apply the magic

wrap-rest-of-input

Wrap all the forms (except defmacro forms) until the end of file

pry-light

Just a small helper to launch a debugging session using `cerror`.

pry

Launch a debugging session with all lexical variables temporarily copied to dynamic environment

Note: you still need to use the `locvar` function to make any modifications the values of the local variables.

pkg-pry

Create a temporary package, and copy all the local variables and functions to global variables and functions with the same `symbol-name` in the new package. Launch a debugging session with this package as `*package*`;

list-locvars (alias for list-local-variables)

List the local variables in the currently inspected captured lexical environment

list-locfuncs (alias for list-local-functions)

List the local functions in the currently inspected captured lexical environment

lexenv-stack-cursor-set

Choose the captured lexical environment to inspect. The stack position can be specified as an absolute number or an offset with respect to the current position.

locvar (local-variable)

`(locvar x)`: read x

`(locvar x 1)`: set x

locfunc (local-function)

`(locfunc f 1 2)`: call f with arguments 1 and 2

push-lexenv-to-saved

Pushes the current lexical environment to the stack for the dynamic extent of executing body.

The `with-local-wrapper` macro puts a layer of `push-lexenv-to-saved` around the code in a new environment.

The `pry` macro uses `push-lexenv-to-saved` if no saved lexical environments are available.

Portability

Portability is currently limited by `hu.dwim.walker`

Currently: SBCL, CCL work fine; but ECL and CLISP known not to work

Work in progress: a simple portable universal code walker for migration from `hu.dwim.walker`

Performance impact

A really inefficient exponential-time Fibonacci function.

Completely consists of entering lexical environments.

We hope it is the worst case for the wrapper.

Performance impact

The horribly inefficient code

```
(defun fib-uw (n)
  (if (<= n 1)
    (progn (when *pry-on-bottom* (pry)) 1)
    (+ (fib-uw (- n 1)) (fib-uw (- n 2)))))
```

Performance measurements

n=40

	time, s	slower than CCL	bytes consed per call
CCL	3.18		16
SBCL	3.50	$1.10\times$	16
ECL	50.53	$15.88\times$	32
CLISP	259.90	$81.68\times$	0
CCL w/wrap	28.29 ($8.89\times$)	$8.89\times$	160 ($+144$)
SBCL w/wrap	13.95 ($3.98\times$)	$4.38\times$	128 ($+112$)

Limitations

You cannot just `setf` the local variables in a pry session and have the program use the new value.

There is no access to the local macros.

`hu.dwim.walker` doesn't like to walk `defmacro forms`.

Why are portable walkers hard to write?

Because things are complicated...

Things are complicated (CLISP)

```
[1]> (load "clisp-impossible-value.lisp")
;; Loading file clisp-impossible-value.lisp ...
;; Loaded file clisp-impossible-value.lisp
T
[2]> (let ((x (f))) x)
T
[3]> (let ((x (f))) (not x))
NIL
[4]> (let ((x (f))) (setf x nil) x)
*** - SETQ: T is a constant, may not be used as a variable
The following restarts are available:
USE-VALUE      :R1      Input a value to be used instead.
ABORT          :R2      Abort main loop
Break 1 [5]>
```

Things are complicated (CLISP)

```
[1]> (load "clisp-impossible-value.lisp")
;; Loading file clisp-impossible-value.lisp ...
;; Loaded file clisp-impossible-value.lisp
T
[2]> (let ((x (f))) x)
T
[3]> (let ((x (f))) (not x))
NIL
[4]> (let ((x (f))) (setf x nil) x)
*** - SETQ: T is a constant, may not be used as a variable
The following restarts are available:
USE-VALUE      :R1      Input a value to be used instead.
ABORT          :R2      Abort main loop
Break 1 [5]>

[6]> (f)
#<SYMBOL-MACRO T>
```

Things are complicated (macroexpand-dammit)

```
* (load "macroexpand-dammit.lisp")
T
* (defmacro f () 1)
F
* (f)
1
* (macroexpand-dammit:macroexpand-dammit '(let ((x 1)) (f)))
(LET ((X 1)) 1)
```

Things are complicated (macroexpand-dammit)

```
* (load "macroexpand-dammit.lisp")
T
* (defmacro f () 1)
F
* (f)
1
* (macroexpand-dammit:macroexpand-dammit '(let ((x 1)) (f)))
(LET ((X 1)) 1)

*(flet ((f () 2)) (f))
2
```

Things are complicated (macroexpand-dammit)

```
* (load "macroexpand-dammit.lisp")
T
* (defmacro f () 1)
F
* (f)
1
* (macroexpand-dammit:macroexpand-dammit '(let ((x 1)) (f)))
(LET ((X 1)) 1)

*(flet ((f () 2)) (f))
2

* (macroexpand-dammit:macroexpand-dammit '(flet ((f () 2)) (f)))
debugger invoked on a SB-KERNEL::ARG-COUNT-ERROR in thread
…
```

Thanks for your attention

Questions?

OK, now I have a question.

What other annoying (but easy to solve) problems are often overlooked?