Accessing local variables during debugging

Michael Raskin
CS Dept., Aarhus University
raskin@mccme.ru*

Nikita Mamardashvili
Moscow Center for Continuous
Mathematical Education

ABSTRACT

Any reasonably large program has to use local variables. It is quite common in the Lisp language family to also allow functions that exist only in a local scope. Scoping rules often allow compilers to optimize away parts of the local environment; doing that is good for performance, but sometimes inconvenient for debugging.

We present a debugging library for Common Lisp that ensures access to the local variables during debugging. To prevent the optimisations from removing access to these variables, we use code-walking macros to store references to the local variables (and functions) inside global variables.

Keywords

CCS: Software and its engineering... Software testing and debugging; lexical environment, lexical closures

DOI: 10.5281/zenodo.3254726

1. INTRODUCTION

We hope that every program of non-negligible size uses some local variables. Unfortunately, during debugging these variables may be inaccessible because of optimisation. For example, when debugging the following code:

```
(labels ((f (z) (+ z 1))) (let ((x 2))
    (cerror "Continue" "Error invoked") (f x)))
```

in `SBCL`[5] 1.3.4 (the freshest available at the time of writing) neither `x` nor `f` were accessible from the debugger with any combination of safety and debugging declarations.

Lacking access to local variables makes debugging runtime errors significantly less convenient. Also, using a continuable error to get a REPL inside the context of a function is significantly less useful as a debugging and exploration tool if the local variables become inaccessible.

`CLISP`[6] seems to do the right thing from the debugging point of view, but, unfortunately, many libraries (for example, `CLSQL`) do not fully support `CLISP`.

Since searching hasn't revealed a solution to this problem, we implemented a brute-force solution, which became the `local-variable-debug-wrapper` library[1], presented in the current paper.

*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 and from the Center for Research in Foundations of Electronic Markets (CFEM), supported by the Danish Strategic Research Council.

1.1 Feature set

The features of the presented library include:

- Access to local variables and functions from the debugger, including the lexical contexts lower on the stack
- Altering local variables during debugging
- A reader trick that allows wrapping the contents of the entire file by adding one line in the beginning

1.2 Example use

The following code illustrates an example file which has wrapping enabled:

```
(use-package :local-variable-debug-wrapper)
; Wrapping to the end of file
(wrap-rest-of-input)
; Inspecting local variables in a function
(defun test-w-1 ()
  (let ((x 1)) (let ((x 3) (y 2)) (pry) (+ 2 3))))
; Debugging a failure
(defun test-w-2 ()
  (let ((x 1)) (let ((x 3) (y 2)) (error "Oops"))))
```

2. TECHNIQUES USED

Currently, we use `hu.dwim.walker`[2] to annotate the input code (annotations are represented using CLOS objects). The forms whose lexical environment differs from that of their parent form get wrapped in a special call.

At the top level the special call is defined as a local macro by `macrolet`. It uses the `&environment` parameter to access lexical environment of each form and the `hu.dwim.walker` wrapper over implementation-specific lexical environment objects to obtain the names of local bindings.

For example, we get the following expansion:

```
(with-local-wrapper (let ((x 1)) x))
-->
(macrolet
  ((push-lexenv-to-saved-inner (&rest args)
     `(push-lexenv-to-saved ,@args)))
  (progn
    (let ((x (push-lexenv-to-saved-inner 1)))
      (push-lexenv-to-saved-inner x))))
```

We build an alist of local functions and their corresponding names. For variables, we want to allow the user to modify local variables and resume execution. This functionality requires capturing a reference, and we use anonymous functions and lexical closures for that (apparently, there is no safe alternative).

This is performed by `push-lexenv-to-saved`. It is a macro using an `&environment` parameter. For examples, one of its calls expands as follows:

```
(push-lexenv-to-saved-inner x)
-->
(let ((*saved-lexenvs*
       (cons (list :variables (list (cons 'x
        (lambda (&optional (value nil value-given))
            (if value-given (setf x value) x)))))
            *saved-lexenvs*)))    x)
```

To make inspecting a saved lexical environment easier we provide the `pry` macro that creates the dynamic variables with the same names as the lexical local variables in the environment under consideration. The dynamic extent of the created variables is limited to the `pry` call, so they affect the debugging session but not the semantics of the program after continuation.

This function is named after the Pry REPL[3] for Ruby, seeing as how not only is it aimed towards the same use case, but the original inquiry that motivated the development was finding a Common Lisp equivalent for that very library.

We also provide lower-level functions for accessing the saved environments, and some other convenience helpers.

To make wrapping all the forms in a file easier, we provide `(wrap-rest-of-input)` functionality: `wrap-rest-of-input` clones the readtable and makes `(` a macro-character. The corresponding reader function immediately reverts to the previous readtable, calls `unread-char` on the opening parenthesis, and reads a form; afterwards the form is put inside the `with-local-wrapper`.

3. EVALUATION

To test a bad case, we used a very inefficient Fibonacci number calculation:

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

We ran the same code wrapped and unwrapped, recording time and memory. The tests were run on a 4-core i7-4770R.

On SBCL, each function call consed 16 bytes when unwrapped and 128 bytes when wrapped. For large parameter values the wrapped version was approximately 4 times slower than the unwrapped one.

CCL ran the unwrapped test slightly faster than SBCL, but the wrapped version was slower than on SBCL. For large parameter values the slowdown was slightly below 9 times.

CLISP does provide full access to local variables on its own in most cases, but this implementation is not currently supported by `hu.dwim.walker`. The unwrapped version runs 80 times slower than on CCL .

An example run for $n = 40$ gave the following results (values in parenthesis are relative to the unwrapped code on the same implementation):

	time, s	slower than CCL	bytes consed per call
CCL	3.18		16
SBCL	3.50	1.10×	16
CLISP	259.90	81.68×	0
CCL w/w	28.29 (8.89×)	8.89×	160 (+144)
SBCL w/w	13.95 (3.98×)	4.38×	128 (+112)

3.1 Known limitations

To modify the bindings used outside the `pry` session one has to use the low-level `local-variable` macro.

The library currently doesn't provide access to local macros.

The limitations of wrapping a piece of code in `progn` apply (note that `wrap-rest-of-input` wraps each form separately).

Macro definitions can't be wrapped. This is due to limitations of `hu.dwim.walker`. If all the macro definitions are top-level `defmacro` forms `wrap-rest-of-input` will do the right thing.

Portability is limited by the `hu.dwim.walker` package. Currently the library is known to be usable on SBCL and CCL and broken on ECL and CLISP.

3.2 Conclusion

Our testing shows that the wrapper provides reliable access to local variables.

We think that this library can make debugging easier in many cases. Impossibility to enforce local variable availability seems surprising (and confusing) to newcomers; we hope that our work can make Lisp slightly more accessible for programmers coming from other languages.

We plan to fix some of the limitations in future. We will be grateful for pointing out corner cases that we have missed.

4. ACKNOWLEDGEMENTS

We thank an anonymous reviewer who provided a lot of valuable advice for improving the present article.

5. REFERENCES

[1] local-variable-debug-wrapper homepage. Retrieved on 22 April 2016. https://gitlab.common-lisp.net/mraskin/\\local-variable-debug-wrapper

[2] hu.dwim.walker package. Retrieved on 15 April 2016. http://dwim.hu/darcsweb/darcsweb.cgi?\\r=LIVE\%20hu.dwim.walker;a=summary

[3] PRY REPL for Ruby. Retrieved on 15 April 2016. http://pryrepl.org/

[4] ANSI Common Lisp Specification, ANSI/X3.226-1994. American National Standards Institute, 1994.

[5] Steel Bank Common Lisp homepage. Retrieved on 15 April 2016. http://www.sbcl.org/

[6] GNU CLISP homepage. Retrieved on 15 April 2016. http://www.clisp.org/