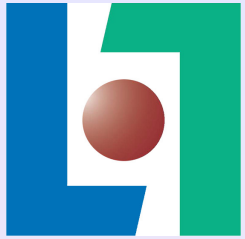


Omnipresent and low-overhead application debugging

Robert Strandh

LaBRI, University of Bordeaux

April, 2020

European Lisp Symposium, Zürich, Switzerland ELS2020

Context: The SICL project

https://github.com/robert-strandh/SICL
Several objectives:

- Create high-quality *modules* for implementors of Common Lisp systems.
- Improve existing techniques with respect to algorithms and data structures where possible.
- Improve readability and maintainability of code.
- Improve documentation.
- Ultimately, create a new implementation based on these modules.
- Provide excellent debugging facilities.
- Keep system *safe*.

Definition of "safety"

For the purpose of this work, we consider a system to be *safe* if and only if, there is no manipulation that a user can do that violates the internal consistency of the system.

We do *not* include in the definition any protection against the program giving the wrong answer as a result of incorrect programming.

Definition of “application debugging”

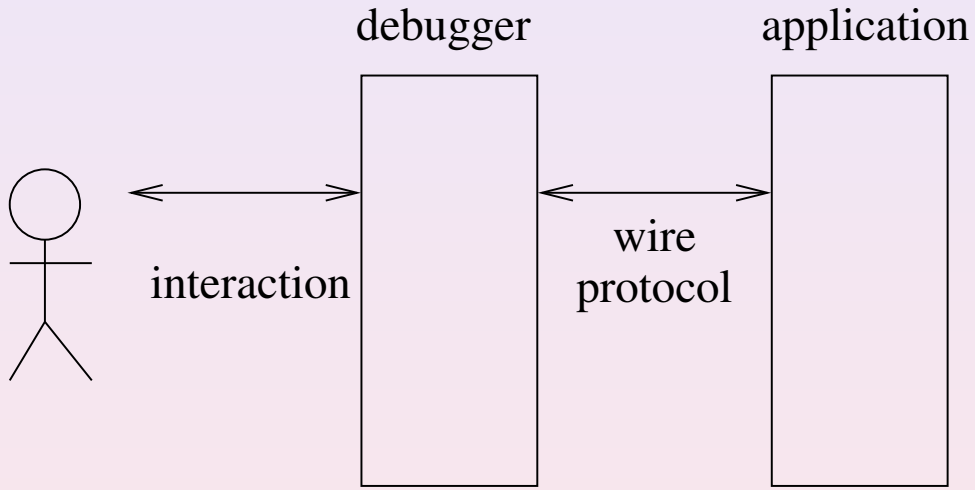
The technique described here is meant for debugging “application code”, as opposed to “system code”.

It is not meant for debugging system components such as the compiler.

It assumes that the compiler generates code that corresponds to the semantics of the source code.

Traditional approach: Process-based debugging

The debugger and the application execute in separate processes with separate address spaces.



Used in UNIX-like systems.
Also recommended for CCL and ECL.

Traditional approach: Process-based debugging

The instructions of the application are modified by the debugger (using copy-on-write pages).

Does not require any collaboration from the application.

Additional debugging information is needed for source-level debugging.

Inherently *unsafe*. The debugger can alter any memory location.

Same-process debugging

Has many advantages in the context of Common Lisp.

Introduces several problems that must be solved.

Setting a breakpoint in a function used by the evaluator or the debugger may render the system useless.

Unsafe operations may crash the system or, worse, silently give the wrong answer.

Debugging in FLOSS Common Lisp implementations

Inferior or non-existing debugging support.

The `trace` facility of most implementations uses *encapsulation*.

Many implementations do not have a working `step` facility.

ECL accomplishes stepping with a special instruction type in the bytecode virtual machine.

SBCL has breakpoints, but they are used only for the `trace` facility. Stepping is accomplished using the condition system, and only when the `debug` quality is sufficiently high.

Debugging in LispWorks

Breakpoints can be set from the editor or the stepper.

The first time a breakpoint is set, the source code of the defining form is re-evaluated with additional annotations for the stepper.

If a breakpoint is encountered, the stepper is automatically invoked if it is not invoked already.

Setting a breakpoint requires source code, so no breakpoints possible in system code.

The `trace` facility uses encapsulation.

Debugging in Allegro

The most complete system of them all.

Breakpoints alter native code, in a way similar to process-based debugging.

To avoid an unusable system, when a breakpoint is encountered, the debugger first *uninstalls* all breakpoints.

This low-level mechanism is used for stepping, source-level debugging, tracing, etc.

The mechanism is inherently unsafe (with our definition of safety).

Our technique

The compiler always includes two versions of each function body:

- One version containing "normal", optimized code.
- One version containing debugging code.

This idea is due to Michael Raskin.

The debugging version contains code for collaboration with the *debugger*.

Our technique

The two different versions are accessible through different *entry points* to the function:

- A function call in the normal version invokes the normal entry point of the callee.
- A function call in the debugging version invokes the debugging entry point of the callee.

Therefore, debugging does not have any overhead in normal code.

This idea is due to Frode Fjeld.

Our technique

The debugger and the application run in the same process, but in different *threads*.

When the application is invoked from the debugger, the debugging entry point is called.

This technique makes it possible to set breakpoints in system code, and even in debugger code.

Our technique, breakpoints

In the debugging version of a function, the compiler inserts a test immediately before and immediately after the evaluation of each form.

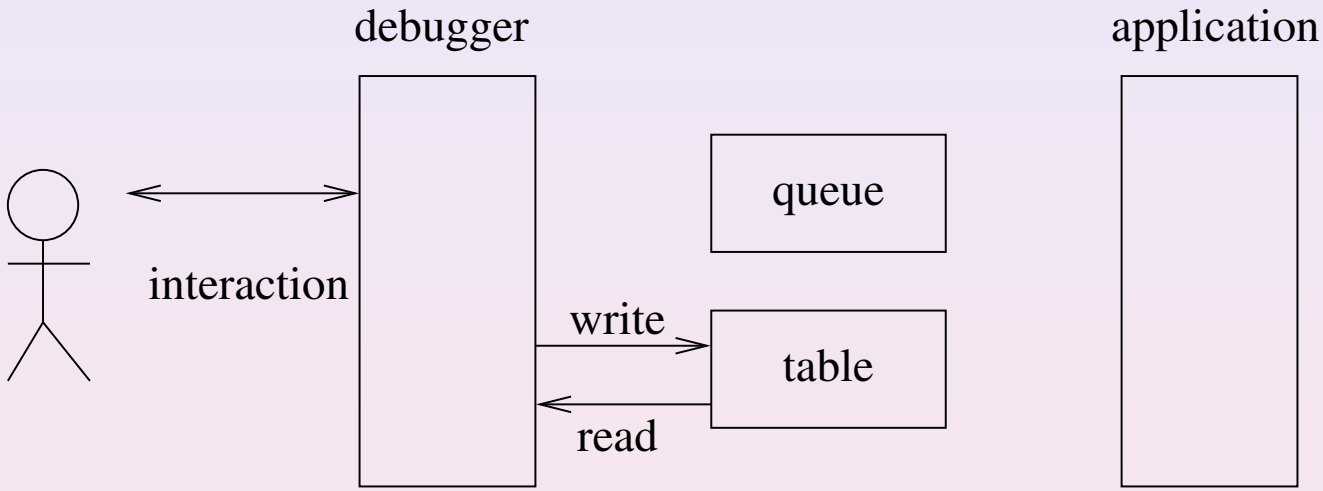
This test consults a table indexed with values of the program counter.

The table indicates whether there is a breakpoint at this location.

This table is managed by the debugger.

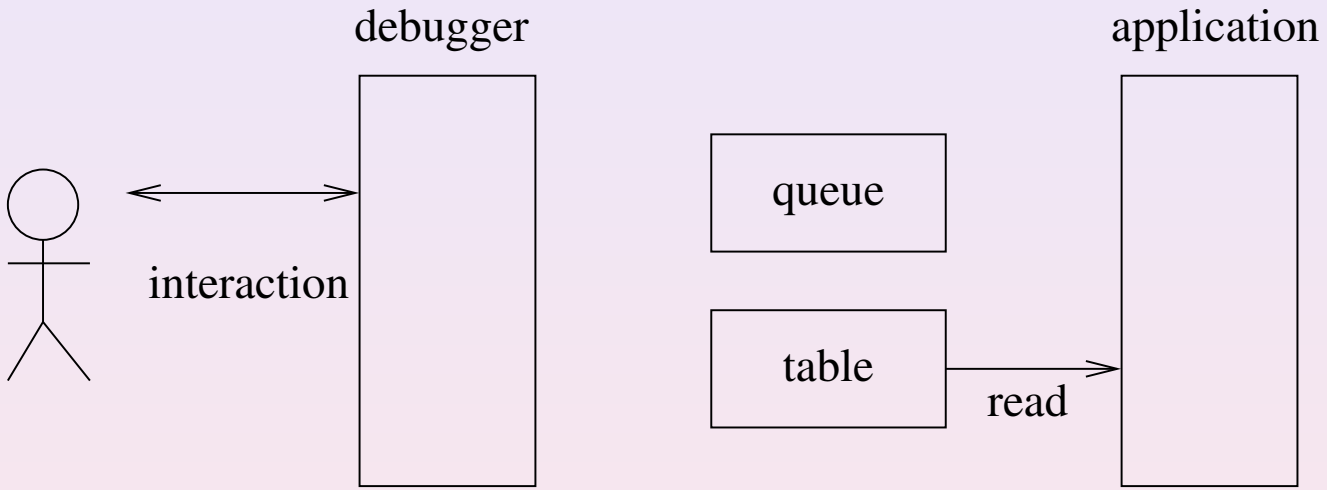
A shared *queue* is provided for communication.

Our technique, breakpoints



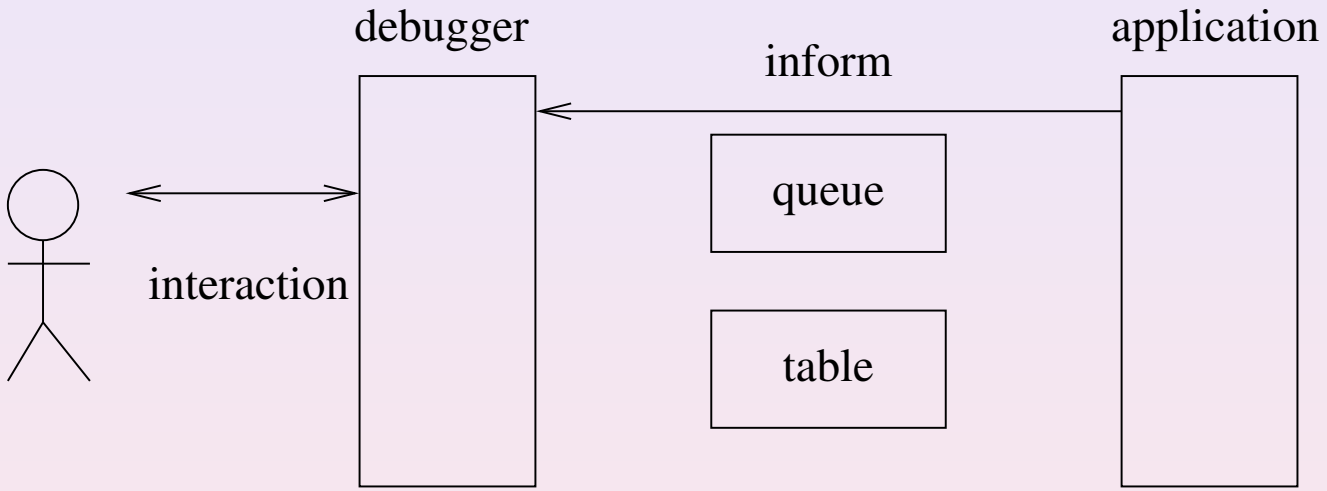
The user sets breakpoints using debugger commands.

Our technique, breakpoints



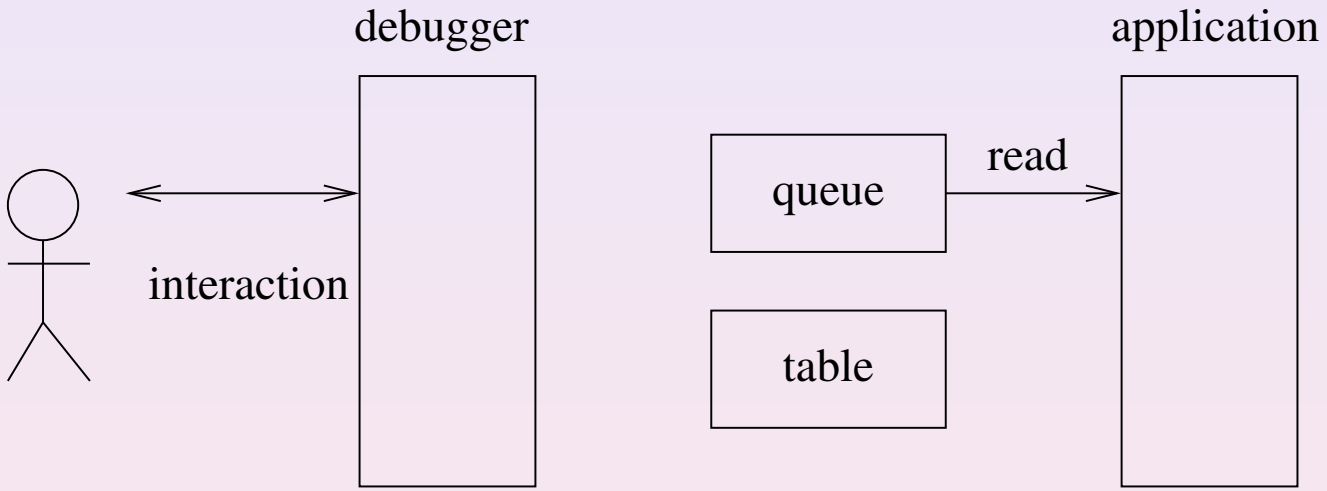
The application consults table before/after each form.

Our technique, breakpoints



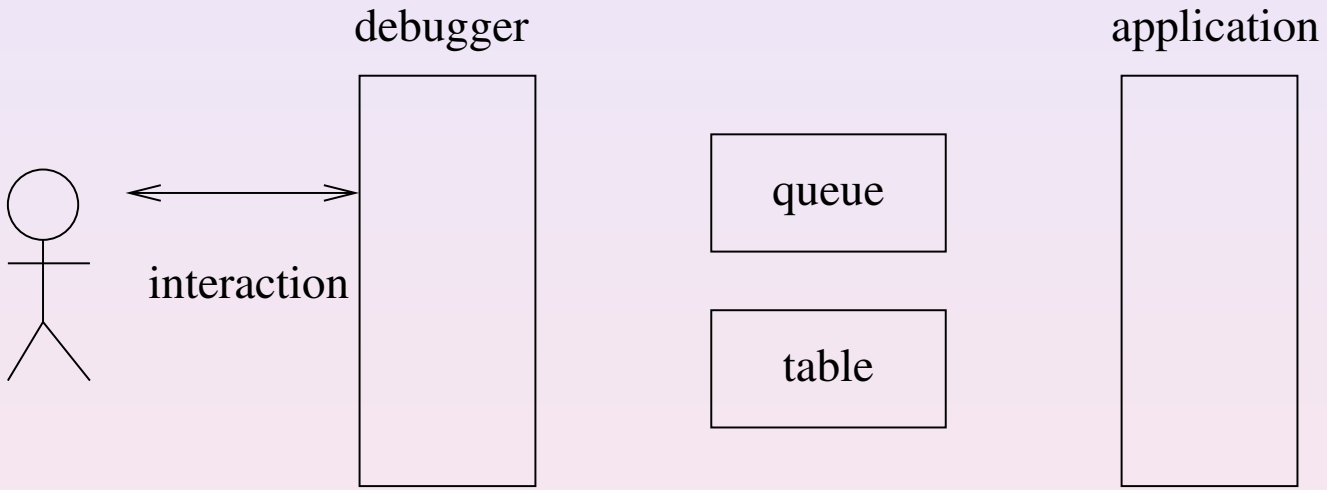
At a breakpoint, the application first informs the debugger.

Our technique, breakpoints



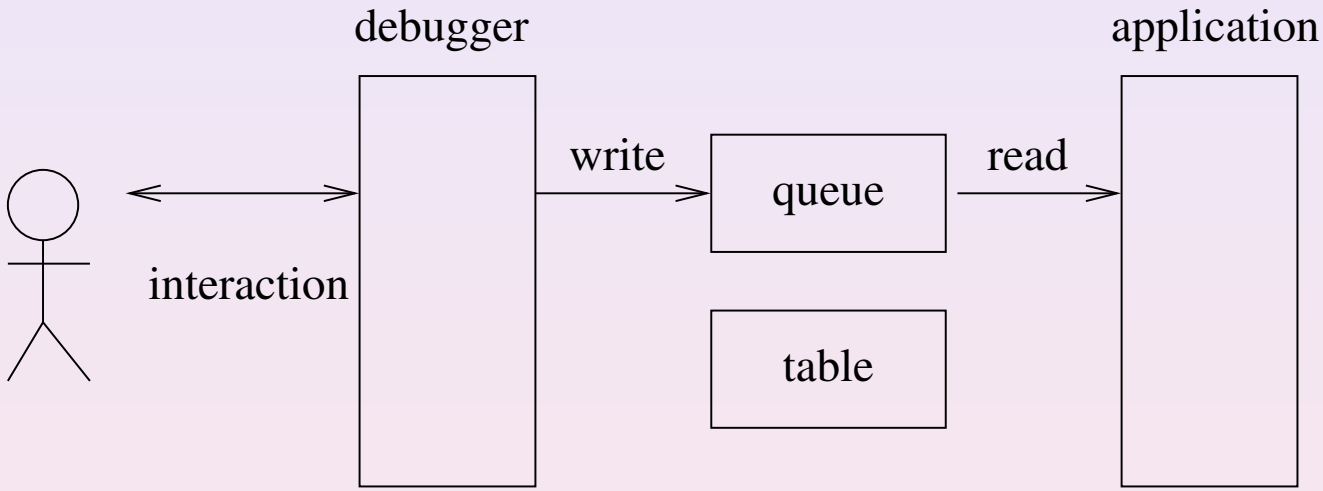
It then blocks from reading from the empty queue.

Our technique, breakpoints



The user examines data and sets/clears breakpoints.

Our technique, breakpoints



The debugger writes to the queue, unblocking the application.

Our technique, stepping

Stepping is accomplished by temporary breakpoints inserted by the debugger.

The user can choose different places to step to

- In to an expression.
- Out of an expression.
- Over an expression.
- Into a function being called.
- Out of the current function being invoked.

The debugger removes the breakpoint from the table, once encountered.

Our technique, tracing

Tracing inserts a breakpoint, marked as a trace point.

When a trace point is encountered, the debugger prints a message, and then immediately unblocks the application.

Our technique, benefits

Safety. The debugger does not alter the application code.

No wire protocol. Simplified communication.

No disadvantages due to encapsulation for tracing.

No need to recompile with a higher `debug` value in order to debug the application.

The full-speed version is used by other threads, so no performance degradation in those threads.

Our technique, disadvantages

Code size will more than double.

Incompatibility with existing Common Lisp implementations.

Our technique, current state

Embryonic implementation of the communication protocol.

Embryonic CLIM-based debugger called Clordane.

Future work

Modify the SICL compiler to generate two versions of every function body.

Try using the portable library for adding debugging annotations to arbitrary code, written by Michael Raskin for implementing/testing communication between the debugger and the application.

Implement the rest of Clordane.

Acknowledgments

We would like to thank the following people for providing information about breakpoints, tracing, and stepping in various Common Lisp implementations: Martin Simmons (LispWorks), Michał “phoe” Herda (CCL), Alex Wood (Clasp), Daniel Kochmański (ECL), Duane Rettig (Allegro).

We would like to thank Frode Fjeld for giving feedback on early versions of this paper, and for suggesting the use of multiple entry points for each function.

We would like to thank Michael Raskin for suggesting that two versions of each function should be provided, thereby making it unnecessary for the programmer to choose the level of debugging.

Thank you