

A debugger for `rm12c`

Mikael Pettersson
Department of Computer and Information Science
Linköping University, Sweden

March 9, 2004

Abstract

This document describes the functionality, constraints, required mechanisms, design, and implementation of a debugger for the `rm12c` compiler.

1 Functionality

First define the interesting *events* to be procedure entries, procedure calls, unifications, and failures. These events are the non-trivial steps in RML execution. For each event there is a corresponding source-code *site*.

The debugger should at least have the following traditional functionality:

1. Breaking when execution reaches a designated site.
2. Continuing until the next breakpoint is reached.
3. Single-stepping to the next event.
4. Displaying the dynamic call chain.
5. Moving up and down through the activation records, and displaying the values of lexically visible variables.
6. Mixing code modules, some compiled with debugging support, and some not.

2 Constraints

The `rm12c` compiler maps RML programs to machine code by a series of non-trivial transformational steps: the abstract syntax is first mapped to First-Order Logic (FOL), then to Continuation-Passing Style (CPS), then to a low-level imperative Code form, then to ANSI-C code, and finally to machine code. Optimizations are applied at every level of representation. These optimizations tend to *significantly* alter the structure of the program.

Traditional implementations of source-level debuggers operate at the machine level. They require mechanisms for inserting breakpoints, traversing call stacks, and mapping machine state (program counter, registers, and stack) to source-code regions and program variable values. In order to accomplish this, debuggers often require that the compiler's optimizations be severely limited in scope, and sometimes completely disabled. They have also required the compiler to generate additional data structures for the inverse mapping from machine state to program state.

For a language like RML and its `rm12c` compiler, this traditional approach is not viable. The many program transformations applied are essential for acceptable execution speed and memory usage. Keeping track of the correct inverse mapping from intermediate code (at every abstraction level!) to source-code regions and variables would entail significant bookkeeping overhead, and be non-trivial to implement. For example, the FOL optimizations 'join' similar code sequences, making the inverse mapping one-to-many. The CPS optimizations, especially procedure inlining, dead-variable removal, and tailcall optimization, also complicate the inverse mapping.

Therefore, the RML debugger follows the approach taken by the portable SML/NJ and Scheme debuggers [?, ?, ?]. The required mechanisms are implemented by source-code instrumentation (applied at the abstract syntax level) and extensions to the runtime libraries. Then the instrumented program is compiled and optimized using the normal compilation pipeline, and linked with the extended runtime library. This approach is significantly easier to implement than the traditional approach. A disadvantage is that program execution speed is reduced (since the program itself 'polls' the debugger continuously), but this overhead may not be greater than that incurred by traditional debuggers.

Being able to mix instrumented and non-instrumented code modules constrains the instrumentation to be of a purely local nature. In particular, it may not add special parameters to procedure calls.

3 Required Mechanisms

To enable breakpoints, the debugger must be able to:

1. map user-interaction positions to source sites
2. suspend the program at every event for a particular site
3. map events to sites
4. resume a suspended program

To enable single-stepping, the debugger must be able to:

1. force the program to suspend at every event
2. determine the cause of suspension (breakpoint or single-stepping)

To display the dynamic call chain, the debugger must be able to:

1. locate the current activation record (or some equivalent data structure)
2. map an activation record to the current procedure name
3. map an activation record to its caller's activation record

To display the values of variables visible at the site corresponding to an activation record, the debugger must be able to:

1. map an activation record to a source site
2. map a source site to the set of visible variables and their types (*scope descriptor*)
3. map an activation record and scope descriptor to the values of visible variables

Since RML employs ML-style parametric polymorphism, the static types of variables may be partially unknown. To properly display values, some mechanism for recovering the actual runtime types should be present.

4 Design

5 Implementation