An Improved Prolog Debugger

Peter Schachte
4 October 1989

This note outlines the planned changes to the Quintus Prolog debugger. The changes are needed for several reasons:

- to make the debugger easier to use
- to make the debugger support a windowing interface
- to make mixed compiled/consulted code more debuggable
- to make compiled (efficient) code debuggable
- to make the debugger more memory-efficient
- to make the debugger better reflect Prolog procedural semantics
- to make choicepoints and indexing more understandable
- to make the debugger more flexible

The following sections discuss the major parts of the project: dividing the debugger into two pieces, allowing the debugging of compiled code, source linkage, and the head port, and we finish with a brief discussion of the interaction between the tasks and initial estimates of the effort required to complete the parts of this project.

1. Dividing the Debugger into Two Pieces

Currently, there is no programatic interface to the Prolog debugger. This makes it difficult for us to provide two interfaces: the usual one and a source-linked, window-oriented one. In order to make these interfaces as easy as possible, and to make it convenient for us to supply more interfaces in the future (e.g., ProWINDOWS, ProXL), we will divide the debugger into two pieces. The first, and largest, piece is the debugger proper. This is the part that is responsible for deciding when to interact with the user, and executing most user commands. The second part of the debugger is the interface, which will be replaceable. The interface is responsible for showing users the state of computation, and determining the user's wishes.

2. Debugging Compiled Code

this may be the effect, but perhaps not actually how to do it.

2.1. premises

This scheme is based on the idea that consulted code will be almost as fast as compiled when debugging is turned off, so users will usually consult everything they might want to debug.

Consult means compile and make all procedures in the file debuggable. When the consulted code is assembled into memory, a different instruction table is used, substituting special **debug_** versions of the call instructions (**debug_** versions of the indexing instruction and linkops will be needed for the new HEAD port described below). These will be explained below.

Note that interpreted static code is necessary for runtime systems, though it is not necessary to make interpreted static code debuggable.

is this the only place where it will be used?

apparently!

2.2. The DONE port

We introduce the DONE port in order to achieve greater memory efficiency. The DONE port is a determinate EXIT port. Firstly, this provides a very useful bit of information: it makes it easy to see when choicepoints are being left around, better reflecting Prolog's actual procedural semantics.

good!

Secondly, it allows us to remove the choicepoints usually left behind by the debugger. This will lead to substantial memory savings on largely determinate programs.

The cost of this is that it will be impossible to retry into determinate goals that have exited, that is, not a current ancestor. For consistency, we should probably not allow users to retry into non-determinate non-ancestors, either. Since few customers understand that they can retry to a non-ancestor, this seems an acceptable loss in exchange for much better memory usage.

2.3. procedure record

First, we change the way procedure records are used when a trap is installed. Currently procedure records for compiled procedures look like this:

```
+---------------+
| hash link     |
+---------------+
| module        |
+---------------+
| functor       |
+---------------+
| arity*        |
+---------------+
| first clause  |
+---------------+
| last clause   |
+-------+-------+
| flags | file  |
+-------+-------+
| dead chain    |
+---------------+
```

what does this mean?

who eventually recalculates the DavidArity?

When we trace a procedure, we will change it to this:

```
+---------------+
| hash link     |
+---------------+
| module        |
+---------------+
| functor       |
+---------------+
| first clause  |
+---------------+
| ptr to trap   |
+---------------+
| last clause   |
+-------+-------+
| flags | file  |
+-------+-------+
| dead chain    |
+---------------+
```

$\Rightarrow$ less time, memory turnover to trap predicates

where "ptr to trap" points to a suitable trap instruction. This is actually quite a bit simpler than our current scheme of copying the procedure record of trapped procedures.

2.4. trap instruction

The **trap** instruction would change a bit. It would work like this:

```
load its argument (where to trap to) into a temp
if temp == 0 then
    get pointer to first clause from proc record
    go there
else
    create term and trap as usual
endif
```

This will allow us to switch debugging mode on and off by just smashing the argument to the trap instruction in the (unique) debug and spy trap pseudo-procedures.

2.5. debug_apply_compiled instruction

In order to return from the debugger to the traced procedure, we need to be able to execute a goal, knowing that the procedure is trapped, and actually get to the procedure, ignoring the fact that the usual first clause pointer points to a trap. The **debug_apply_compiled** instruction does exactly as **apply_compiled** does, except that it finds a pointer to the real first clause for the procedure where the arity is usually stored.

2.6. debugger

The **trap** instruction would trap to a procedure that looks like this:

```
compiled_trace_trap(Goal, Proc /*, State */) :-
        FailChoice is '$Quintus: choice_point',
        '$debugger_repeat'(Goal /* & more stuff */),
        RetryChoice is '$Quintus: choice_point',
        % do user interaction at 'CALL' or 'REDO' port
        debug_apply_compiled(Proc, Goal),
        (   determinate ->
                % user interaction at 'DONE' port,
                '$Quintus: cut'(FailChoice)
        ;   RedoChoice is '$Quintus: choice_point',
            repeat,
            % user interaction at 'EXIT' or 'REDO' port
        ).
```

details about depth and invocation counts have been left out.

Note that '$debugger_repeat' is just like repeat, except that it keeps arguments in the choicepoint. We scan backwards to find choicepoints for '$debugger_repeat' (call these "debugger choicepoints") when we want to show ancestors or retry back (debugger choicepoints actually keep invocation number and maybe depth, as well as goal).

Also note that on determinate exit from our goal, we *don't* push a choicepoint. For now, let's call that a DONE port. This means that we won't be able to retry non-ancestors, since they won't leave their choicepoints. But it also means that determinate exit won't leave any choicepoints around to waste space.

We check for determinate exit by checking

```
        RetryChoice =:= '$Quintus: choice_point'.
```

Following are descriptions of how we handle the various debugger actions:

creep	Set debug trap to compiled_debug_trap/2 and succeed.
leap	Set debugger to leap mode, succeed.
skip	Set debug trap to 0, succeed. On return or failure, reset debug trap to what it was before.
nodebug	Set debug trap to 0, succeed.
backup	Temporarily turn off fail and redo leashing, cut to RetryChoice, and fail.
ancestors	Scan choicepoints and environments backwards, printing the goal and invocation arguments from all of the ancestral debugger choicepoints.
retry	Cut to RetryChoice and fail
retry i	Scan choicepoints back to the one for invocation i, then cut back to it and fail.
fail	Cut to FailChoice and fail
fail i	Scan choicepoints back to the one for invocation i, then cut back to the previous choicepoint and fail.
quasi-skip	(Skip, but stop at spypoints and keep debugging information.) Set debugger to leap mode, succeed. On return or failure, reset debug mode to what it was before.
fast-leap	(Skip, but stop at spypoints and don't keep debugging information. This might be useful in expert mode.) Set debug trap to 0, succeed. On return or failure, reset debug mode to what it was before.

all other actions work just the way they do now. The last two items are not currently supported, but wouldn't be hard to add, and seem useful.

2.7. Making Compiled Code Debuggable and Vice Versa

It should be possible to make a compiled procedure debuggable. This will be necessary if we are to allow creeping into compiled code, and will also be a useful feature for experts. A similar approach can be used when consulting files: compile them as usual, and turn the **call** instructions into **debug_call** instructions as that code is assembled into memory. (By **call** instructions, I mean all instructions that can call another procedure.) Then put the appropriate trap on all the procedures.

presumably, this won't be allowed for "frozen" procs

Note that there is a side issue here: should save_qof translate **debug_** versions of instructions to their non-**debug_** versions? I believe it should not, or at least not always, as to do so would mean that save_qof would loose the compiled/consulted destinction of saved code. It would not be possible to compile several files, consult several more, and save_qof a quickly loadable file that is partially debuggable. It might be useful to have an option to save_qof that would do the consulted ⇒ compiled translation.

Another issue is allowing qofs to be "consulted." This would be trivial, and would go no slower than regular qof loading, since it would only require that the instruction table be modified slightly to coerce the above-listed instructions into their **debug_** counterparts. It would be similarly possible to force a qof with some "consulted" procedures to be compiled. Of course~~m~~ these will still require that traps be installed on each debuggable procedure.

2.8. The Interpreter

Given the changes listed above, it will be possible to simplify the interpreter, speeding it up (and making it consume much less memory) in the process. The interpreter will no longer need to pass around terms to keep track of other goals.

yes!

There will, however, have to be a different version of the debugger for debugging dynamic code. It will be much like the current interpreter-based debugger, only it will get information by stack scanning, instead of as arguments.

3. Source Linkage

3.1. debug_call instructions

In order to effect source linkage, we need to know which procedure called a debugged procedure. Since the call instructions don't keep this information (at least not always), we need to introduce a new version of the call instructions, call them **debug_call** instructions. These behave the same as the call instructions, except that they set a **CALLER** pseudo-register to the value of the **P** register at the time of the call. They could even be implemented by having them do the move, and then jump in the emulator to the code for the corresponding non-debug instruction. (In fact, if not for the write-mode hack, they could be placed immediately before the corresponding instruction, and just flow into it.) yes.

3.2. code scanning to find caller

Given that the **CALLER** register points to the last **debug_call** instruction (or thereabouts), we need to be able to show where in the source we are. So first we must figure out which procedure and clause we are in. The method that incurs the least time and space overhead when you're *not* debugging is to scan the source when you need to figure out where you are. The first problem we run into is that the **CALLER** register will point into the middle of a clause somewhere, it's impossible to scan backwards, and currently there is no way to find the end of a clause.

Therefore, we introduce a new **end** instruction which always comes at the end of every clause, and never is executed (because the previous instruction will always jump or return). Since it is never executed, there need be no implementation of this instruction. It could be some odd number, or zero, as long as it's different from all existing instructions. Immediately following the **end** instruction would come an offset back to the linkop that begins the clause. This offset could be viewed as an argument to this instruction or not, whichever is more convenient.

Since there is a relatively small set of instructions that could be the last instruction in a clause, it would be possible to portmanteau the **end** instruction with many, or even all, of these. And since the end instruction doesn't actually do anything, the portmanteau is trivial: it need only be at a different address than the base instruction. It may be possible to make these instructions start immediately before the base instruction, begin with a no-op, and then flow into the base instruction (assuming write mode is not a problem). Actually, if this is done, it may be best to make the base instruction flow into the portmanteaued one, on the theory that these instructions are more often than not last.

This would have some performance penalty in some cases, and might render it undesirable. In that case, it is sufficient to just use the end instruction and pay the extra 2 bytes per clause. In fact, it might not cost this anyway, due to alignment restrictions. In fact, if alignment seems to force 4 bytes to be used in most cases, portmanteauing would be undesirable.

Note that where instructions that have **debug_** versions are portmanteaued, the **debug_** version must also be portmanteaued.

Given this, we could find out which clause of which proc we are in as follows, where Addr0 is the contents of the **CALLER** register and Calleeproc is the proc we're just entering, as returned by the **trap** instr:

why?

```
where_am_i(Addr0, Calleeproc, Proc, Clause) :-
        check_callee(Addr0, Calleeproc),
                        % make sure CALLEE calls me
        find_clause_start(Addr0, Clause_start),
        find_proc(Clause_start, Proc),
                                % find this proc
        scan_for_clause(Proc, Clause_start, 0, Clause).
                        % figure out which clause this is
```

This could be written in C, if that is easier.

Given that you know which clause of which proc you're in, now how do you point to the goal in the source? First, we modify compile and consult to remember the character position in the file where each procedure begins. Starting from there, we find the appropriate clause by skipping over as many clauses as necessary. how?

Then we read this clause using a special read that returns not only the term, but an isomorphic term containing the starting and ending positions for each subterm in this term. Although this level of detail is not strictly necessary, it would be more difficult to collect only the needed information. Also, this information would be useful for a more sophisticated editor interface later.

Next, we scan the compiled code for the clause as we scan the source code term and positions term in parallel. When we find the call we're trying to show, we check to make sure that the source term corresponds to this, and, if so, we have the position of goal in the source. If this operation proves to be too time consuming, we could cache this information and only compute it when needed.

4. The HEAD Port

One innovation that we feel will make it easier to understand the behavior of a Prolog program is the HEAD port. This is a new port in the debugger which comes right after the CALL and REDO ports, but after any indexing instructions. It also appears in backtracking, regardless of whether or not the procedure has reached the exit port. This should make backtracking and unification somewhat less mysterious.

To implement this port, we need to have **debug_** versions of the linkops and indexing instructions. These instructions trap into the debugger, allowing us to show the HEAD port. The debugger then goes *back* to the same instruction that trapped to it, only this time, the instruction does its ~~usual~~ usal job. We also want to have a way to disable the trapping altogether, in order to have consulted code run as quickly as possible when we're not debugging.

4.1. The Trapping Mechanism

The way these instructions trap is somewhat different than the **trap** instruction. When these instructions trap, they want to be able to get back, with the engine in the same state. This is achieved by pushing a choicepoint before trapping. Therefore, the debugger fails in order to restart the instruction and continue execution.

When these **debug_** instructions trap, they do the following:

```
maybe_trap:
        if (!HEAD_TRAP_RESTART && HEAD_TRAP) then
            A1 := build_goal_term
            A2 := P
            A3 := 0
            push choicepoint core
            push A1 into choicepoint
            push any extra regs into choicepoint (B0, PR)
            call proc stored in HEAD_TRAP reg
        endif
        HEAD_TRAP_RESTART := 0
```

Notes:

1. We don't want to trap if we're restarting this instruction, or if debugging is switched off.
2. The address of the procedure record for head_trap_debugger/3 can be found in the **HEAD_TRAP** register. If this register is zero, then we don't want to trap.
3. In order to construct the goal term, these instructions must have the name and arity of the procedure, which are stored in the procedure record. To find this, these instructions must follow the chain of linkops to find the tagged backpointer to the procedure record.
4. The backtrack pointer in the constructed choicepoint should point to the instruction that is trapping, so that failure will re-execute the instruction. When this happens, the emulator will be in a different state, so that trapping will not happen the second time the instruction is executed.
5. In the place where **A1** would be stored in the choicepoint, we store a pointer to the goal term. This holds all the arguments, so we can restore them from there.
6. It's not clear which extra registers will need to be saved. What is unusual here is that the next instruction executed after this choicepoint is failed into might be a **try_me_else** instruction, so we'll need to preserve whatever registers it might need.
7. Each choicepoint will be built by the same instruction that will use it to restore state; therefore, if it becomes necessary, different instructions can build different choicepoints.
8. The **index_on_A1** instruction will actually put a pointer to the next instruction to be executed in **A3**. The debugger can tell this is the case by the fact that the instruction pointed to by **A2** is **index_on_A1**. It can tell whether this is a list case or key hash by the first arg of the goal term in **A1**.

4.2. Restarting an Instruction After a Trap

All the **debug_** instructions begin with the same code:

```
maybe_restart:
    if (HEAD_TRAP_RESTART) then
        restore A regs from goal term
        restore extra registers
        pop choicepoint
    endif
```

Notes:

1. The instruction can tell whether it's restarting by the state of the **HEAD_TRAP_RESTART** register. If this is non-zero, then this instruction is being restarted. This register is set by the debugger just before it fails, restarting the instruction. The register is reset by the **debug_** instructions before dispatching to the next instruction. This is not necessary if they are trapping to the debugger, because the debugger is not expected it be debuggable itself.

2. The choicepoint must always be popped.

4.3. debug_just/try_me_else Instructions

```
debug_just_me_else:
        maybe_restart
        maybe_trap
        goto just_me_else
debug_try_me_else:
        maybe_restart
        maybe_trap
        goto try_me_else
```

4.4. debug_retry/trust_me_else instructions

```
debug_retry_me_else:
        maybe_restart
        restore_args_from_chpt
        maybe_trap
        continue
debug_trust_me_else:
        maybe_restart
        restore_args_from_chpt
        maybe_trap
        pop choicepoint
        continue
```

4.5. debug_index_on_A1 and debug_just_index_on_A1 instructions

Note that the **debug_just_index_on_A1** is only needed because **just_index_on_A1** does not look at the following instruction, and so would not notice whether it was **index_on_A1** or **debug_index_on_A1**.

```
debug_index_on_A1:
        fudge_P
        goto debug_just_index_on_A1
debug_just_index_on_A1:
        maybe_restart
        P1 := do_indexing
        if (!HEAD_TRAP_RESTART && HEAD_TRAP) then
            A1 := build_goal_term
            A2 := P
            A3 := P1            % different from maybe_trap
            push choicepoint core
            push A1 into choicepoint
            push any extra regs into choicepoint (B0, PR)
            call proc stored in HEAD_TRAP reg
        endif
        HEAD_TRAP_RESTART := 0
        continue_to(P1)
```

4.6. debug_try/retry/trust_subitem_else instructions

```
    debug_try_subitem_else:
            maybe_restart
            maybe_trap
            goto try_subitem_else
    debug_retry_subitem_else:
            maybe_restart
            restore_args_from_chpt
            maybe_trap
            continue_to_right_place
    debug_trust_subitem_else:
            maybe_restart
            restore_args_from_chpt
            maybe_trap
            pop choicepoint
            continue_to_right_place
```

4.7. The head_port_debug Procedure

When debugging (and HEAD port interactions are desired), the **HEAD_TRAP** register will hold a pointer to the head_port_debug/3 procedure. This procedure is responsible for determining which clause is about to be tried, and which would be tried next if this clause should fail. It determines this from its arguments, and any existing choicepoint.

```
    head_port_debug(Goal, P, Case) :-
            get_instr(P, This_instr),                                  ?
            find_next(This_instr, P, Case, Continue, This, Next0),
            get_instr(Continue, Next_instr),
            (   \+ debug_instr(Next_instr) ->
                    head_port_user_interaction(Goal, This, Next)
            ;   true
            ),
            set_restart,
            fail.
```

Find_next/6 determines which instruction will be executed next, which clause is being tried now, and which clause will be tried next. It does not worry

```
    find_next(debug_just_me_else, P, 0, Continue, P, 0) :-
            Continue is P + ???.
    find_next(debug_try_me_else, P, 0, Continue, P, Next) :-
            linkop_next(P, Next),
            Continue is P + ???.
    find_next(debug_retry_me_else, P, 0, Continue, P, Next) :-
            linkop_next(P, Next),
            Continue is P + ???.
    find_next(debug_trust_me_else, P, 0, Continue, P, 0) :-
            Continue is P + ???.
    find_next(debug_try_subitem_else, P, 0, Continue, This, Next) :-
            linkop_next(P, Next_subitem),
            subitem_item(P, Continue),
            This is Continue - ???,
            subitem_item(Next_subitem, Next_continue),
            Next is Next_continue - ???.
    find_next(debug_retry_subitem_else, P, 0, Continue, This, Next) :-
            linkop_next(P, Next_subitem),
            subitem_item(P, Continue),
            This is Continue - ???,
            subitem_item(Next_subitem, Next_continue),
            Next is Next_continue - ???.
    find_next(debug_trust_subitem_else, P, 0, Continue, This, Next) :-
            linkop_next(P, Next_subitem),
            subitem_item(P, Continue),
            This is Continue - ???,
            Chpt is '$Quintus: choicepoint',
            (   \+ debugger_choicepoint(Chpt) ->
                    get_chpt_bp(Chpt, Next)
            ;   Next = 0
            ).
    find_next(debug_index_on_A1, P, Continue, Continue, This, Next) :-
            This is Continue - ???,
            Chpt is '$Quintus: choicepoint',
            (   \+ debugger_choicepoint(Chpt) ->
                    get_chpt_bp(Chpt, Next)
            ;   Next = 0
            ).
```

Note there is no find_next clause for **debug_just_index_on_A1**. This should be okay if **debug_just_index_on_A1** is implemented properly, since it should update **P** to point to the **debug_index_on_A1** instruction before trapping.

4.8. Converting Compiled Code to Consulted and Vice Versa

This would only change slightly from what was done for debugging compiled code. The same procedure would apply, only now there are more instructions that need to be transposed to their **debug_** counterparts:

just_me_else	just_index_on_A1	index_on_A1
try_me_else	try_subitem_else	call
retry_me_else	retry_subitem_else	depart
trust_me_else	trust_subitem_else	execute

(and all the variants of these). This change should involve only updating a few tables.

5. Interaction

Note that the designs of these features do interact. In particular, doing the source linkage or HEAD port debugging without debugging compiled code would require a different design. Also, source linkage requires first dividing the debugger into two pieces (since Prolog might not be running under emacs when the user wants to debug).

6. Sizing Estimates

6.1. Split Debugger Into Two Parts (10 days)

divide code into kernel and interface parts	4 days
define bidirectional debugger interface	6 days
documentation	(omitted for 3.0)

6.2. Debugging Compiled Code (34 days)

change to procedure record for traps	3 days
change to trap instruction	1 day
consult, install trap	1 day
debug_apply_compiled instruction	1 day
converting compiled ⇔ consulted in core	(omitted for 3.0)
ancestor scanning	3 days
debugger mods	10 days
simplifying the interpreter	5 days
documentation	10 days

6.3. Source Linkage (68 days)

install debug_call instrs (assemble time)	3 days
install debug_call instrs (load_qof time)	1 day
write debug_call & friends	3 days
end instruction & offset in every clause	5 days
code scanning to find caller proc & clause	5 days
store char positions per procedure	3 days
implement new read that returns char positions	5 days
code scanning to find place in source	5 days
arrange to show place & port in source	15 days
implement breakpoints on goals	5 days
debugger commands from emacs	8 days
documentation	10 days

6.4. The HEAD Port (31 days)

debug_ versions of linkops and index instrs	15 days
install debug_ instrs (much done above)	1 day
HEAD port debugger	10 days
documentation	5 days