Idempotent I/O for safe time travel

Zoltan Somogyi
zs@cs.mu.oz.au

The University of Melbourne

Why time travel?

- check procedure’s input arguments: all correct
- step over procedure call
- check procedure’s output arguments: some are incorrect

With conventional debuggers, the programmer must start the program again, find the same call, and this time step into it, not over it.

Being able to reset the program to the state it had at the time of the call (as if travelling back in time) is much more convenient.

Restoring an earlier state

Time travel requires restoring the values of all variables, local and global.

In imperative languages, this is very difficult to do efficiently for long running programs.

In single assignment languages, the task is much easier: if the debugger knows which variables are bound at the time of the call, it can simply reset all variables bound since then to unbound.

Prolog can use the trail (required to support backtracking) to record when variables are bound at runtime. In functional languages and moded logic programming languages, the same info is available at compile time.

Internal vs external state

Restoring the values of variables is relatively easy, because they are stored inside the program's address space, over which the debugger can usually exert whatever level of control is necessary.

However, all useful programs also interact with entities outside their own address space, including files, devices, other processes, and network connections.

Restoring the state of these entities is much harder, if it is possible at all. This is why Prolog debuggers, which implement retry operations (jumps backward in time) as a matter of course, typically do not even attempt to do so.

Effects of not restoring external state (1)

```
write_solution(Stream, ProblemDescription) :-
    <compute Solution from ProblemDescription>,
    write(Stream, Solution).
```

Time travel back across a call to `write_solution` will cause output to be duplicated.

Effects of not restoring external state (2)

```
read_problem(Soluion) :-
    read(ProblemDescription),
    <compute Solution from ProblemDescription>.
```

Time travel back across a call to `read_problem` will cause `read_problem` to return the wrong `ProblemDescription`.

Effects of not restoring external state (3)

```
get_stream(Stream) :-
    write("please type filename: "),
    read(Filename),
    open(Filename, read, Stream).
```

Time travel back across a call to `get_stream` will cause a file descriptor leak, as well as duplicated output and skipped input.

Effects of not restoring external state (4)

```
read_next_item(Stream, MaybeItem) :-
    read(Stream, Item),
    ( Item = end_of_file ->
        MaybeItem = no,
        close(Stream)
    ;
        MaybeItem = yes(Item)
    ).
```

Time travel back across a call to `read_next_item` may cause an attempt to read from a closed stream.

Restoring external state

The obvious way to avoid these effects is to restore the external state as well as the internal state. Unfortunately, this is easier said than done.

- Some changes to external state aren't reversible: e.g. sending email, cutting metal.
- Some changes to external state are reversible only in principle: e.g. sending a message to a process whose source code you don't have.
- Some external state is mutated by several processes; even if it can be restored, there is no guarantee that later accesses by the time travelling program will have the same results as on first execution.

I/O primitives

External state can be accessed and modified only through a known set of primitive operations.

The set of operating system calls can be considered to be one set of primitive operations.

In many recent languages, programs cannot issue system calls directly, and must instead go through a library module. The operations of these modules can then be considered to be primitives.

```
read_char_code(Stream, CharCode, S0, S) :-
    <foreign_code, "C", [Stream, CharCode, S0, S],
        "CharCode = getc(Stream); S = S0;">.
```

Idempotent I/O

If we can't restore external state, the next best thing is to arrange for the program to behave *as if* it had been restored. We can do this by modifying the implementations of *all* the primitive operations that can access or modify external state.

- When the program executes a call to a primitive for the first time, we execute the actions called for by the operation and record the results.

- When the program executes a call to a primitive for the second, third etc time, after a retry has warped time from after the call to before the call, we just return the results we recorded the first time without actually performing reading or writing external state.

The idempotency transformation

```
read_char_code(Stream, CharCode, S0, S) :-
    impure allocate_io_action_number(IoActionNum),
    ( semipure io_has_occurred(IoActionNum, Block) ->
        semipure restore_answer(Block, 0, CharCode),
        semipure restore_answer(Block, 1, S)
    ;
        <foreign_code, "C", [Stream, CharCode, S0, S],
            "CharCode = getc(Stream); S = S0;">,
        impure create_answer_block(IoActionNum, 2, Block),
        impure save_answer(Block, 0, CharCode),
        impure save_answer(Block, 1, S)
    ).
```

Time travel with idempotent I/O

At each call, we save the current I/O action number in the new stack frame.

When the programmer wants to retry a call, i.e. to jump back in time to the point when the call was originally made, the debugger resets the global variable holding the current I/O action number to the value saved in the retried call's stack frame.

When forward execution resumes, the program will make the same calls to I/O primitives as it did before. They will return the same answers, but this time without touching external state.

The time warp is thus not visible from outside the process.

Controlling the overhead

When the Mercury compiler is compiling six of its own largest modules (34,000+ lines, 1.2+ Mb of code), it executes over 12 million calls to I/O primitives.

Enabling the tabling of I/O primitives increases runtime in the debugger by about 10%, from 190s to 210s. It also increases the memory size of the process about threefold, from 127 Mb to 389 Mb.

Handling more I/O intensive programs requires a mechanism that turns on I/O tabling for only a part of the program's runtime. Safety can still be ensured for time jumps entirely within the tabled region.

Foreign language interfaces

New languages need foreign language interfaces, because programmers won’t switch to a language that doesn’t let them exploit existing software.

A foreign language interface that can invoke imperative language code effectively takes a part of the internal state of the program, the part controlled by foreign language code, out of the control of the host language implementation.

The solution is to apply the idempotency transformation not just to operations that access external state but also to operations that invoke foreign language code.

Impure declarative languages

Languages such as ML and Prolog have impure constructs such as mutable variables and assert/retract.

The part of the internal state manipulated by such constructs can be restored at time jumps using techniques such as logs, checkpoints and reexecution.

Reexecution is safe only if time travel restores all state, internal and external. If a system uses the idempotency transformation to restore external state, it may also use it to restore the internal state controlled by impure constructs.

Declarative debugging

If the idempotency transformation preserves the identity of the called primitive and its inputs as well as its outputs, then the table can be used not just for time travel but also for declarative debugging.

The semantics of predicate includes not just the values it computes for the output arguments from the given values of the input arguments, but also all its effects:

- all its I/O actions,
- all its foreign language calls, and
- all its impure operations.

Declarative debugging example

```
mdb> dd
prompt_read_int("please input an integer: ", 1123, ...)
5 io actions:
write_string("please input an integer: ")
read_char(’1’)
read_char(’2’)
read_char(’3’)
read_char(’\n’)
Valid? n
write_string("please input an integer: ", ...)
1 io action:
write_string("please input an integer: ")
Valid? y
...
```

Conclusions

Making an operation idempotent requires only a simple transformation. This transformation can ensure the safety of time travel with respect to

- operations on state external to the program
- operations on internal state not under the control of the debugger
- operations on internal state that is not single-assignment

The overheads of the transformation are usually acceptable with respect to both space and time for programs written in a declarative style.

The transformation enables the declarative debugging of programs that perform these kinds of operations.