Tracing and Debugging Mathematica Programs

Bruce Smith, Wolfram Research, 14 Jan 91 (updated 29 May 92)

(some examples by Stephen Wolfram)

Seeing the Steps of an Evaluation

Getting a List of the Intermediate Forms of an Evaluation

Whenever *Mathematica* evaluates an expression, the expression is repeatedly transformed by various rules until no more transformations apply. We can see a list of the intermediate forms using **Trace**.

```
2 3
6
Trace[2 3]
{2 3, 6}
Trace[y + a + y]
{y + a + y, a + y + y, a + 2 y}
```

During the course of an evaluation, subsidiary evaluations may need to be done. These are shown as nested lists.

```
Trace[2 3 + 4 5]
{{2 3, 6}, {4 5, 20}, 6 + 20, 26}
```

All subevaluations are traced, even if they result from separate definitions.

```
fib[0] = 0;
fib[1] = 1;
fib[n_] := fib[n-1] + fib[n-2]
```

```
Trace[fib[3]]
```

```
{fib[3], fib[3 - 1] + fib[3 - 2],
  {{3 - 1, -1 + 3, 2}, fib[2], fib[2 - 1] + fib[2 - 2],
   {{2 - 1, -1 + 2, 1}, fib[1], 1},
   {{2 - 2, -2 + 2, 0}, fib[0], 0}, 1 + 0, 0 + 1, 1},
  {{3 - 2, -2 + 3, 1}, fib[1], 1}, 1 + 1, 2}
```

☐ Limiting the Intermediate Steps that are Recorded

When there are lots of steps, we don't want to see all of them. There are various ways to limit the kind of steps that are recorded. In order for Trace to be useable at all on large evaluations, it is essential to understand how to use these options, singly and in combinations.

The ways to limit what Trace records include:

By specifying a tag to which rules must be attached:

```
Trace[2 3 + 4 5, Plus]
```

```
{6 + 20, 26}
```

```
Trace[2 3 + 4 5, Times]
```

```
{{2 3, 6}, {4 5, 20}}
```

By giving a pattern which expressions must match:

```
Trace[2 3 + 4 5, x_ y_]
```

```
{{2 3}, {4 5}}
```

```
Trace[fib[3], fib[_]]
```

```
{fib[3], {fib[2], {fib[1]}, {fib[0]}}, {fib[1]}}
```

(We can use more than one pattern or tag, by separating them with the **|**, a new pattern construct in Version 2.0:)

```
Trace[fib[3], fib[_] | _ + 1 ]
```

```
{fib[3], {fib[2], {fib[1]}, {fib[0]}, 1 + 0, 0 + 1},
  {fib[1]}, 1 + 1}
```

By specifying a maximum depth:

```
Trace[fib[3], TraceDepth->1]
```

```
{fib[3], fib[3 - 1] + fib[3 - 2], 1 + 1, 2}
```

By turning on tracing only within certain functions:

```
fac[1] = 1; fac[n_] := n fac[n-1]
```

```
Trace[Log[fac[2] x]]
```

```
{{{fac[2], 2 fac[2 - 1],
    {{2 - 1, -1 + 2, 1}, fac[1], 1}, 2 1, 1 2, 2}, 2 x}\
   , Log[2 x]}
```

```
Trace[Log[fac[2] x], TraceOn -> fac]
```

```
{{{fac[2], 2 fac[2 - 1],
    {{2 - 1, -1 + 2, 1}, fac[1], 1}, 2 1, 1 2, 2}}}
```

By turning off tracing within certain functions:

```
Trace[Log[fac[2] x], TraceOff -> fac]
```

```
{{{fac[2], 2}, 2 x}, Log[2 x]}
```

(In the examples above, note that the full trace was split into the parts inside and outside the call to **fac,** by the use of **TraceOn** and **TraceOff** respectively. Thus, one should use **TraceOff** to remove the traces of functions one already understands, and **TraceOn** to limit the trace to the functions whose behavior one wants to investigate.)

If we understand several functions used in the one we are debugging, we can turn tracing off in all of them (again using the **|** pattern construct), to eliminate as much as possible of the irrelevant output:

```
Trace[fib[2] + fac[2]]
```

```
{{fib[2], fib[2 - 1] + fib[2 - 2],
   {{2 - 1, -1 + 2, 1}, fib[1], 1},
   {{2 - 2, -2 + 2, 0}, fib[0], 0}, 1 + 0, 0 + 1, 1},
  {fac[2], 2 fac[2 - 1],
   {{2 - 1, -1 + 2, 1}, fac[1], 1}, 2 1, 1 2, 2},
  1 + 2, 3}
```

```
Trace[fib[2] + fac[2], TraceOff->fib ]
```

```
{{fib[2], 1}, {fac[2], 2 fac[2 - 1],
   {{2 - 1, -1 + 2, 1}, fac[1], 1}, 2 1, 1 2, 2},
  1 + 2, 3}
```

```
Trace[fib[2] + fac[2], TraceOff->fib|fac ]
```

```
{{fib[2], 1}, {fac[2], 2}, 1 + 2, 3}
```

Mathematica 's built-in functions often generate expressions internally, and evaluate them. By default, Trace records a certain subset of these evaluations, including sets and unsets of symbols (e.g. iterator variables), and messages.

We can use the option **`TraceInternal`** to make Trace record all or none of the internal evaluations:

```
Trace[NSum[x, {x,0,1}]]
```

```
{NSum[x, {x, 0, 1}], {x = 1., 1.}, {x =. , Null},
  {x = 0, 0}, {x =. , Null}, 1.}
```

```
Trace[NSum[x, {x,0,1}], TraceInternal->True]
```

```
{NSum[x, {x, 0, 1}], {D[x, {x, 1}], 1}, {x = 1., 1.},
  {{x, 1.}, {1.}}, {x =. , Null}, {x = 0, 0},
  {{x, 0}, {0}}, {x =. , Null}, 1.}
```

When we use **`TraceInternal->True`**,
we see one internal evaluation of **`D[x, {x, 1}]`**, and 2 of **`{x}`**.

```
Trace[NSum[x, {x,0,1}], TraceInternal->False]
```

```
{NSum[x, {x, 0, 1}], 1.}
```

When we use **`TraceInternal->False`**, we see the evaluation of **`NSum`** as an atomic unit. This makes it easier to trace evaluations using **`NSum`** (or many other built-in functions) several times:

```
Trace[NSum[x^2, {x,0,2}] + NSum[1-x^2, {x,0,2}],
    TraceInternal->False ]
```

```
             2
{{NSum[x , {x, 0, 2}], 5.},

                 2
  {NSum[1 - x , {x, 0, 2}], -2.}, 5. - 2., -2. + 5., 3.}
```

☐ Patterns that Select Any Kind of Expression

We can use patterns that match any expression, including "commands":

```
Trace[
    sum = 0; Do[sum += i^2, {i,1,4}],
    i = _
]
```

```
General::spell1:
   Possible spelling error: new symbol name "sum"
     is similar to existing symbol "Sum".
```

```
{{{i = 1}, {i = 2}, {i = 3}, {i = 4}}}
```

```
Trace[
    sum = 0; Do[sum += i^2, {i,1,100}],
    _ += (v_ /; v > 200 && v < 300)
]
```

```
{{{sum += 225}, {sum += 256}, {sum += 289}}}
```

☐ Seeing the Context of an Evaluation

We can use other **Trace** options to see the situation in which an evaluation occurred.

Seeing the final results of certain expressions:

```
Trace[fac[4], fac[_], TraceForward -> True]
```

```
{fac[4], {fac[3], {fac[2], {fac[1], 1}, 2}, 6}, 24}
```

Seeing where certain expressions came from:

```
Trace[fac[7], 120, TraceBackward -> True]
```

```
{{{fac[5], 120}}}
```

Seeing intermediate (as well as initial or final) forms:

```
Trace[fac[7], 120, TraceBackward -> All]
```

```
{{{fac[5], 5 fac[5 - 1], 5 24, 120}}}
```

Seeing forms before arguments are evaluated:

```
Trace[fac[7], 120,
    TraceBackward -> All, TraceOriginal -> True]
```

```
{{{fac[6 - 1], fac[5], 5 fac[5 - 1], 5 24, 120}}}
```

Seeing why an evaluation occurred (*i.e.*, what other evaluations led to it), and/or what resulted from it:

```
Trace[fac[7], 120, TraceAbove -> True]
```

```
{fac[7], {fac[6], {fac[5], 120}, 720}, 5040}
```

```
Trace[fac[7], 120, TraceAbove -> All]
```

```
{fac[7], 7 fac[7 - 1],
  {fac[6], 6 fac[6 - 1],
   {fac[5], 5 fac[5 - 1], 5 24, 120}, 6 120, 720},
  7 720, 5040}
```

Seeing "where from" and "what then" separately:

```
Trace[fac[7], 120, TraceAbove -> {All, False}]
```

```
{fac[7], 7 fac[7 - 1],
  {fac[6], 6 fac[6 - 1],
   {fac[5], 5 fac[5 - 1], 5 24, 120}}}
```

```
Trace[fac[7], 120, TraceAbove -> {False, All}]
```

```
{{{120}, 6 120, 720}, 7 720, 5040}
```

☐ Using **Trace** for Debugging

Here is a program containing a bug.

```
primes[n_] :=
  If[ n = 0,
    (* then *)
        {},
    (* else *)
        If[ PrimeQ[n],
            (* then *) Append[primes[n-1], n],
            (* else *) primes[n-1]
        ]
    ]
(* i.e. Select[ Range[n] , PrimeQ ] *)
```

```
nprimes[n_] := Length[primes[n]]
```

```
General::spell1:
   Possible spelling error: new symbol name "nprimes"
     is similar to existing symbol "primes".
```

Here’s what happens when we try to find out how many primes there are up to 10:

```
nprimes[10]
```

```
Set::setraw: Cannot assign to raw object 10.
3
```

We can trace the use of **Message**. But all we see is the **Message** call that was evaluated, and its result.

```
Trace[nprimes[10], Message]
```

```
Set::setraw: Cannot assign to raw object 10.
{{{{Message[Set::setraw, 10], Null}}}}
```

We need to know why **Message** was called - *i.e.,* what we were evaluating at the time which required the emission of the message.

We use **TraceAbove**, specifying that we should see all of the previous values of the forms being evaluated, but none of the future values (relative to the point at which the message was emitted).

```
Trace[nprimes[10], Message, TraceAbove->{All,False}]
```

```
Set::setraw: Cannot assign to raw object 10.
{nprimes[10], Length[primes[10]],
  {primes[10], If[10 = 0, {},
    If[PrimeQ[10], Append[primes[10 - 1], 10],
     primes[10 - 1]]],
   {10 = 0, {Message[Set::setraw, 10], Null}}}}
```

This result makes the bug fairly easy to find -- just look before the Message to see that it was triggered by evaluating 10 = 0, which came from the If statement.

☐ Printing the Intermediate Forms

We can print out the expressions traced, instead of making them into a list. (But only the forward-in-time options work in this case.)

```
TracePrint[fib[3], fib[_]]
```

```
 fib[3]
  fib[3 - 1]
  fib[2]
   fib[2 - 1]
   fib[1]
   fib[2 - 2]
   fib[0]
  fib[3 - 2]
  fib[1]
2
```

Using **TraceScan**, we can perform any other operation on the expressions when they're intercepted. (See the *Mathematica* book for more details.)

☐ Printing short names of symbols in private contexts

When using Trace to debug functions defined in packages containing private contexts, we often see very long variable names, which makes it hard to read the output. Here is a simple package:

```
BeginPackage["SomeContext`"];
  SomeFunction;
  Begin["`Private`"];
    SomeExponent = 2;
    SomeFunction[xx_] := xx ^ SomeExponent;
  End[];
EndPackage[];
```

Here are the full names of the symbols defined by the package:

```
Names["SomeContext`*"]
```

```
{SomeFunction}
```

```
Names["SomeContext`*`*"]
```

```
{SomeContext`Private`SomeExponent,
  SomeContext`Private`xx}
```

Our **$ContextPath** includes the "public" context, but not the private one:

```
$ContextPath
```

```
{SomeContext`, Graphics`Animation`, Global`, System`}
```

This means that when we use Trace, symbols in the private context are printed using their full names.

```
Trace[SomeFunction[17]]
```

```
                       SomeContext`Private`SomeExponent
{SomeFunction[17], 17                                  ,

                                       2
  {SomeContext`Private`SomeExponent, 2}, 17 , 289}
```

The easiest solution is to add the private context to our ContextPath during debugging, like this:

```
AppendTo[$ContextPath, "SomeContext`Private`"]
```

```
{SomeContext`, Graphics`Animation`, Global`, System`,
  SomeContext`Private`}
```

Now, when we Trace, we get exactly the same output, but it is printed in a way that makes it easier to read:

```
Trace[SomeFunction[17]]
```

```
                       SomeExponent
{SomeFunction[17], 17            , {SomeExponent, 2},

    2
  17 , 289}
```

■ Stopping at Specified Points in an Evaluation

□ Examining Intermediate States with **TraceDialog**

We can use **TraceDialog** to temporarily stop a program just before it evaluates certain expressions, and to examine the state of the system while it's stopped.

When we encounter **fib[4]** during the evaluation of **fib[6]**, we are placed in an interactive "dialog", much like the top-level **In**/**Out** loop. (We could use any pattern in place of **fib[4]**.)

```
TraceDialog[fib[6], fib[4]]
```

```
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
fib[4]
TraceDialog::dgend: Exiting Dialog.
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
fib[4]
TraceDialog::dgend: Exiting Dialog.
8
```

The initial value of **%** in the dialog is set to the expression that is about to be evaluated after the dialog is finished, enclosed in **HoldForm** to prevent it from evaluating within the dialog.

```
InputForm[%]
```

```
HoldForm[fib[4]]
```

(The fact that certain evaluations were done inside a dialog can be recognized from their **In** and **Out** prompts, which have been modified by the Front End, but these prompts are not shown in this document. In a non-FrontEnd version, the prompts are indented. By changing the format of **DialogIndent**, the prompt format used during dialogs can be changed. The following command is an example:)

```
Format[DialogIndent[n_]] :=
    SequenceForm @@ Table["> ", {n}]
```

We can see what expressions the system is in the middle of evaluating by looking at the "stack" ± *Mathematica*'s internal "list of things to do", or more accurately, things to finish doing. The expression whose evaluation started first, and will finish last, is listed first.

```
Stack[_]
```

```
{TraceDialog[fib[6], fib[4]], fib[6 - 1] + fib[6 - 2],
  fib[5 - 1] + fib[5 - 2], fib[5 - 1]}
```

The argument to **`Stack`** is a pattern telling which kinds of expressions to show. Without an argument, only the "tags" ± the symbols to which the active rules or internal machine-code routines are attached ± are seen.

```
Stack[]
```

```
{TraceDialog, Plus, Plus, fib}
```

When we have seen enough in the present (suspended) state, we can continue by typing **`Return`** (or **`Return[]`**) (but *not* by pressing the return key!). We will then stop at the next requested point, if any.

```
Return
```

(In the new version of the Front End, the message output from Dialog is shown in the original cell in which the Dialog began, so we don't see it above. I don't know why we don't see an input prompt above.)

In this case, **`fib[4]`** is used twice in computing **`fib[6]`**, so we have stopped again. The different stack reflects the different point in the algorithm at which we stopped.

```
Stack[_]
```

```
{TraceDialog[fib[6], fib[4]], fib[6 - 1] + fib[6 - 2],
  fib[6 - 2]}
```

This time, after we type **`Return`**, there are no more evaluations of **`fib[4]`**, so we enter no more dialogs, and the original computation finishes.

```
Return
```

(In the Front End, the result is shown after the original TraceDialog above. For this reason, it might be convenient to execute commands within Dialogs in a separate Notebook window attached to the same Kernel. This additionally prevents the Notebook being debugged from becoming cluttered up with debugging commands.)

We could have obtained essentially the same information from **`Trace`** much more easily:

```
Trace[fib[6], fib[4], TraceAbove->{All, False}]
```

```
{fib[6], fib[6 - 1] + fib[6 - 2],
  {fib[5], fib[5 - 1] + fib[5 - 2], {fib[4]}}, {fib[4]}}
```

Thus, **TraceDialog** is only better than **Trace** when you need to examine things other than the stack, you need to make changes to the Trace options during the tracing, or you don't know in advance just what pattern you should trace.

☐ Changing the Search Pattern with **$TracePattern**

We can change the pattern the system is looking for by resetting the value of the global variable **$TracePattern**.

There are three **fib[3]**'s evaluated each time we evaluate **fib[6]**.

```
Trace[fib[6], fib[3]]
```

```
{{{{fib[3]}}, {fib[3]}}, {{fib[3]}}}
```

We can arrange to stop when we encounter each **fib[3]**, much like in the previous example.

```
TraceDialog[fib[6], fib[3]]
```

```
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
```

```
fib[3]
```

```
TraceDialog::dgend: Exiting Dialog.
```

```
8
```

This time, we understand what's happening and don't want to continue stopping. We can examine and change the global variable **$TracePattern**, which contains the current pattern being used by any **Trace**-like function. But we must be careful to avoid accidentally evaluating the value of this variable, since it will typically be a pattern that matches expressions that are not fully evaluated. We can do this with the **??** operator.

```
??$TracePattern
```

```
$TracePattern is the currently active pattern argument
   to Trace and related functions. It can be reset
   during the trace to alter the set of expressions
   recorded or printed.
```

```
$TracePattern = fib[3]
```

By changing **$TracePattern** to something that will not match, we can finish our calculation immediately.

```
$TracePattern = None
```

```
None
```

```
Return
```

(It is also possible for a function being debugged by **`Trace`** to change **`$TracePattern`** while it runs. It can change the trace pattern just during certain evaluations by using **`Block[{$TracePattern = ...}, ...]`**.)

☐ Using the Inspector from the Interrupt Menu

We can get into a dialog any time we want, by using the "Enter Inspector Dialog" button in the Interrupt dialog box (or the **`inspect`** option in the Interrupt menu, in systems that do not have a front end). (Note that the terms "dialog box" and **`Dialog[]`** are unrelated.)

In the current version of the front end, command-comma brings up the interrupt dialog box. (Command-period causes an immediate abort of the calculation.)

(The Notebook window may continue to say "Running... ", due to a bug in the front end. In spite of this, you can evaluate an expression in the Inspector Dialog.)

```
Table[x^2, {x,1,3000}];
```

```
x
```

```
1889
```

```
Stack[]
```

```
{CompoundExpression, Table, Set}
```

```
Return
```

(Another front end bug may cause the original interrupted cell to continue to be marked as if it is evaluating. Scrolling the notebook window so the cell bracket temporarily goes out of view should fix this.)

☐ Using the **`Interrupt[]`** Command

The **`Interrupt[]`** command gives us access to the Interrupt menu or dialog box, which allows us to enter an inspector. (In *Mathematica* Version 1.2, **`Interrupt[]`** caused an abort. This is now done by **`Abort[]`** in Version 2.0.)

Let's make a table of a global constant, but use **`Interrupt[]`** before each use of the constant, in order to reserve the right to change the constant before we use it.

```
a = 1;
```

```
Table[Interrupt[]; a, {3}]
```

```
{1, 2, 2}
```

```
a
```

```
1
```

```
Return
```

Here we change the value of a global variable from within the inspector.

```
a = 2
```

```
2
```

```
Return
```

This time I selected Continue instead of Inspector in the dialog box.

```
{1, 2, 2}
```

Copyright