

Package ‘jit’

February 14, 2012

Version 1.3-1

Title Just-in-time compiler for the R language

Author Stephen Milborrow

Maintainer Stephen Milborrow <milbo@sonic.net>

Description Enable just-in-time (JIT) compilation. The functions in this package are useful only under Ra and have no effect under R. See www.milbo.users.sonic.net/ra/index.html.

License GPL-3

URL <http://www.milbo.users.sonic.net/ra>

Repository CRAN

Date/Publication 2011-08-27 05:08:03

R topics documented:

is.ra	1
jit	2
nojit	7

Index	9
--------------	----------

is.ra	<i>Am I running Ra?</i>
-------	-------------------------

Description

After loading the `jit` library, `is.ra` is TRUE if running Ra, else FALSE.

Note also that `version` will contain "Ra" when running Ra.

Usage

```
is.ra
```

See Also

```
jit
nojit
```

Examples

```
is.ra
```

```
jit
```

Just-in-time compilation

Description

Enable just-in-time (JIT) compilation of a block of R code. Arithmetic in loops in a JIT block is generally faster.

This function is only useful under Ra. It has no effect under standard R. See <http://www.milbo.users.sonic.net/ra>.

This help page is for Ra 1.3.0 and higher.

Usage

```
jit(jit = NA, trace = 0)
```

Arguments

jit	<p>NA Make no changes but return the current JIT state (default)</p> <p>0 End JIT block. Using <code>jit(0)</code> is usually unnecessary because an implicit <code>jit(0)</code> is performed when the function returns.</p> <p>1 Start JIT block.</p> <p>2 Start JIT block with extra optimization of nested loops. Fast but potentially dangerous, see below.</p> <p>3 Start JIT block with sanity checks. Very slow, mainly for development of the jitter code.</p>
trace	<p>0 Silent (default).</p> <p>1 Show which expressions are compiled to JIT code.</p> <p>2 Show compilation details.</p> <p>3 Show code generation details.</p> <p>4 With <code>jit=3</code> show code execution.</p>

Details

Nomenclature

A *JIT block* is the R code between `jit(1)` and `jit(0)` or the end of the function.

Jitted code is R code that has been compiled by the just-in-time compiler (to a form where it will execute more quickly). Not all code in a JIT block is necessarily jitted.

A *jitted variable* is a variable used in jitted code.

What gets jitted?

The following operations are jitted, but only if they are in a `for/while/repeat` loop in a JIT block:

- **Arithmetic** and **comparison** operations on **logical**, **integer**, and **double** scalars and vectors (but not matrices). Here "scalar" means a vector of **length** one. Both arguments of a binary operator must have the same length, or one of them must have length 1 (general R recycling is not jitted).
- Assignment to scalars or vectors: `x <- y`
- **Subscripted** vectors: `x[i] <- y[j]`. Multiple subscripts `x[i, j]` are not jitted on the lhs of assignments.
- **Control** constructs such as `if` and `for` when the condition is **logical**, **integer**, or **double**.
- The following (one argument) functions when the argument is **logical**, **integer**, or **double**: `abs` `acos` `acosh` `asin` `asinh` `atanh` `ceiling` `cos` `cosh` `digamma` `exp` `expm1` `floor` `gamma` `gammaCody` `lgamma` `log` `log2` `log10` `log1p` `sin` `sign` `sinh` `sqrt` `tan` `tanh` `trigamma` `trunc`. Some of these functions can take more than one argument; they are jitted only if given one argument.

Functions called from the JIT block are not jitted. For example, the `for` in the first line of code below is not compiled because it is called from within `system.time`.

```
jit(1); system.time({ for(i in 1:100) x<-x+i }) # is not jitted
system.time({ jit(1); for(i in 1:100) x<-x+i }) # is jitted
```

Use `trace>=1` to see what was jitted (and possibly modify your code to enable more jitting). Code in a JIT block that is not actually jitted will run as normal R code and coexist happily with jitted code.

Summary of the differences between standard R and jitted code

The semantics of jitted code differ from standard R where retaining compatibility would have made the jitted code substantially slower. Bear in mind that not everything in a JIT block is necessarily jitted. Non-jitted code in a JIT block retains R's usual semantics.

The short version of the next few sections is:

- You will get an error message if you change the type or length of a jitted variable. There are a few other semantic restrictions.
- Arithmetic in jitted code is similar to C.
- Avoid NAs in jitted code (NaNs are ok).

Differences between standard R and jitted code

The JIT compiler will not allow you to change the length or type of a jitted variable. (For this reason, the "L" suffix to define an integer constant 123L is useful in JIT blocks, although seldom used in standard R code.) Here "type" means the type returned by `typeof`, such as `logical`, `integer`, or `double`.

The JIT compiler will not allow you to use functions like `attach` in a JIT block or in functions called from the JIT block. The idea is to keep the environment stable so jitted code can be efficient.

NAs of type `integer` and `logical` are not handled in jitted code. `Double` NAs in arithmetic expressions may be converted to NaNs on some architectures. The results of comparison to `NaN` differ between standard R and jitted code. The next section gives details.

Integer arithmetic overflows are detected in standard R but not in jitted code

Most attributes of jitted variables are dropped.

An out-of-range index in a jitted expression will cause an error message (the `check.bounds` options setting is ignored):

```
x <- 1; x[3] # NA in standard R but "Error: out-of-range index" when jitted
```

The value of a loop is the last evaluated loop body in standard R, but `NULL` in jitted code:

```
x <- for (i in 1:3) 99 # x is set to 99 in standard R but NULL when jitting
```

NaNs and NAs

NaNs: In jitted code `NaNs` (always `double`) are handled directly by the machine hardware, as is usually the case in standard R. (We assume IEEE 754 hardware http://en.wikipedia.org/wiki/IEEE_754.) The hardware knows about double NaNs: where necessary it will generate them (zero divided by zero is NaN) and propagate them (NaN + 123 is NaN).

Double NAs: In R, one of the many possible values of NaN provided by the hardware is defined to be the double NA. The hardware therefore makes no distinction between double NAs and other NaNs. Standard R evaluation has some software to handle double NAs over and above the hardware (mostly in type conversions, which don't get jitted anyway); jitted code has no such software. Thus in jitted code, double NAs behave identically to NaNs.

On some architectures, the result of an jitted expression including a double NA may be NaN, not NA i.e. double NAs may be propagated as NaNs.

Integer and logical NAs: Integer and logical NAs are meaningless in jitted code, and will cause incorrect results if used. The following explains why. Integer and logical NAs are represented by `INT_MIN` internally in R, where `INT_MIN` is the minimum integer representable in a machine word (on a 32 bit machine, `INT_MIN` is about $-2e9$). The hardware does not know about integer and logical NAs. In standard R these are therefore handled in software; in jitted code they are treated like any other integer. Thus a logical or integer NA in jitted code is treated as an integer with the value `INT_MIN` (with a logical value of `TRUE`).

Remember that a plain NA in your program text is treated as *logical* by R. In the standard R code below, the software recognizes that the logical NA is a NA, and thus correctly converts it to a double NA (the conversion is necessary to add the NA to 1.2):

```
1.2 + NA # evaluates to NA in standard R
```

In jitted code, the logical NA is not recognized as a NA — it is treated as an integer with the value INT_MIN and is thus converted to a double with the value INT_MIN:

`1.2 + NA` # evaluates to `1.2 + INT_MIN` in jitted code.

Summarizing, avoid NAs in jitted code unless you are sure that your NA is a double.

In jitted code, comparing anything to a NaN results in FALSE, as per IEEE 754:

	standard R	jitted
<code>1.2 == NaN</code>	NA	FALSE
<code>1.2 > NaN</code>	NA	FALSE
<code>NaN == NaN</code>	NA	FALSE (sic)

Other discrepancies:

	standard R	jitted
<code>1L %% 0L</code>	integer NA	runtime error message
<code>1L %/% 0L</code>	integer NA	runtime error message

These discrepancies are justified by the fact that integer NAs are meaningless in jitted code so it does not make sense to generate one. On the other hand, standard division `"/"` of an integer by zero evaluates to double `Inf` in both standard R and jitted code.

Optimizing nested loops with `jit(2)`

Use `jit(2)` for extra optimization of inner nested loops. In the example below the inner `j` loop will undergo this optimization:

```
jit(2)
for (i in 1:n) { # same as jit(1) because is outermost loop
  ...
  for (j in 1:m) # extra optimization with jit(2), 1:m must not change
    ... # any further loops in here also optimized with jit(2)
  ...
}
```

To be optimized this way the entire inner loop body must be jittable and the loop sequence (`1:m` above) must have integer type.

There are some **important conditions** when using `jit(2)`. The danger is that R will usually not warn you if your code doesn't meet a condition — you will just get wrong results. Because of this danger, `jit(2)` is considered experimental at present. The conditions are:

- The inner loop sequence (`1:m` in the example above) is calculated just once and thereafter assumed constant for the entire JIT block. Thus the example code above will give incorrect results without warning if `m` changes between the curly braces (which would require a recalculation of `1:m`). It is fine to change `n` since this affects only the outer loop.
- Do not use `break` or `next` in the inner loop (they will incorrectly break to the outer loop, but please don't rely on this). It is, however, ok to use `return` in the inner loop.

- There is no NAMED handling in the inner loop [TODO elaborate].
- Assigning to the inner loop variable causes incorrect results and an error message (which is issued only at the end of the loop):

```
jit(2)
for (i in 1:3)
  for (j in 1:3)
    j = 9L      # error msg: assignment to loop variable "j"
```

Limits of the current implementation

You can JIT only one function at any time. A call to `jit` in a function that is called from a JIT block is ignored with a warning message.

Jitted code uses more memory than standard R. This is because jitted code does not release the temporary buffers used to evaluate expressions until the end of the JIT block. Standard R allocates and releases temporary buffers as it evaluates each expression.

More types of expression will be jitted in future releases.

Value

This function returns a three element integer vector.

Value under R

All elements are 0.

Value under Ra

[1] The value of the last specified `jit` argument to `jit()`, so is 1, 2, or 3 if in a JIT block, else 0. Is 0 in functions called from the JIT block.

[2] The value of the last specified `trace` argument to `jit()`. Is 0 in functions called from the JIT block.

[3] The value of the last specified `jit` argument to `jit()`. Retains its value in functions called from the JIT block.

Be careful when using `jit()` as an argument to a function. It will be evaluated as if called from that function. Example:

```
jit(1)
jit()[1]      # is 1
print(jit()[1]) # prints 0, because jit() is called from print
jit.flag <- jit()[1]
print(jit.flag) # prints 1
```

See Also

For more information on the jitter see <http://www.milbo.users.sonic.net/ra>

`nojit`
`is.ra`
`enableJIT` in the `compiler` package is not related
`jitter` is not related

Examples

```
foo <- function(N, jit.flag) {
  jit(jit.flag)
  x <- 0
  for (i in 1:N)
    x <- x + 1
  x
}
if(!is.ra)
  cat("\nExpect no speedup because you are running R, not Ra.\n")
N = 1e6
time.nojit <- system.time(foo(N, jit.flag=0))[1]
time.jit <- system.time(foo(N, jit.flag=1))[1]
cat("Time ratio", time.nojit / time.jit, "\n")
jit(0)
```

nojit

Inhibit just-in-time compilation of a variable

Description

Inhibit just-in-time compilation of a variable.

Usage

```
nojit(sym = NULL)
```

Arguments

`sym` the variable

Details

The JIT compiler will not allow you to change the type or length of a jitted variable. Sometimes this can be a nuisance, and if so you can use `nojit(variable)` to inhibit jitting of the specified variable.

Any error message (not necessarily issued by the jitter) disables jitting and thus clears all `nojit` settings — so after an error message you will need to re-`nojit` your variable.

Value

A character vector containing the symbols marked as not jittable, if any, else NULL.

See Also

[jit](#)
[is.ra](#)

Examples

```
## Not run:  
jit(1); x <- 0; y <- 0  
nojit(x)      # allows use of "c" below which changes the length of x  
for (i in 1:3) {  
  x <- c(x,i) # will not be jitted  
  y <- y + 1  # will be jitted  
}  
nojit(0)  
  
## End(Not run)
```

Index

*Topic **programming**

is.ra, 1
jit, 2
nojit, 7

abs, 3
acos, 3
acosh, 3
Arithmetic, 3
asin, 3
asinh, 3
atanh, 3
attach, 4

ceiling, 3
comparison, 3
compiler, 7
Control, 3
cos, 3
cosh, 3

digamma, 3
Double, 4
double, 3, 4

enableJIT, 7
exp, 3
expm1, 3

floor, 3
for, 3

gamma, 3
gammaCody, 3

if, 3
integer, 3, 4
is.ra, 1, 7, 8

jit, 2, 2, 8
jitter, 7

length, 3
lgamma, 3
log, 3
log10, 3
log1p, 3
log2, 3
logical, 3, 4

NaN, 4
NaNs, 4
NAs, 4
nojit, 2, 7, 7

options, 4

repeat, 3

sign, 3
sin, 3
sinh, 3
sqrt, 3
Subscripted, 3

tan, 3
tanh, 3
trigamma, 3
trunc, 3
typeof, 4

version, 1

while, 3