Namespaces, Source Code Analysis, and Byte Code Compilation

Luke Tierney
Department of Statistics & Actuarial Science
University of Iowa

Introduction

- R is a powerful, high level language.
- As R is used for larger programs there is a need for tools to
  - help make code more reliable and robust
  - help improve performance
- This talk outlines three approaches:
  - name space management
  - code analysis tools
  - byte code compilation

Why Name Spaces

Two issues:

- static binding of globals
- hiding internal functions

Common solution: name space management tools.

Static Binding of Globals

- R functions usually use other functions and variables:

```
f <- function(z) 1/sqrt(2 * pi) * exp(- z^2 / 2)
```

- Intent: `exp`, `sqrt`, `pi` from base.
- Dynamic global environment: definitions in base can be masked.

.GlobalEnv
package:ctest
⋮
package:base

Hiding Internal Functions

Some useful programming guidelines:

- Build more complex functions from simpler ones.
- Create and (re)use functional building blocks.
- A function too large to fit in an editor window may be too complex.

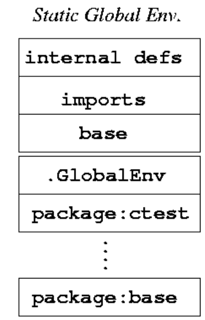
Problem: All package variables are globally visible

- Lots of little functions means clutter for user.
- Lots of functions means name conflicts more likely.
- Consequence: often use big functions with repeated code.

Name Spaces for Packages

Starting with 1.7.0 a package can have a name space:

- Only explicitly exported variables are visible when attached or imported.
- Variables needed from other packages can be imported.
- Imported packages are loaded; may not be attached.



Name Spaces for Packages (cont.)

Adding a name space to a package involves:

- Adding a NAMESPACE file
- Replacing `require` calls by `import` directives.
- Replacing `.First.lib` by `.onLoad` (and maybe `.onAttach`).

NAMESPACE File Directives

- `export`

  ```
  export(as.stepfun, ecdf, is.stepfun, stepfun)
  ```

- `exportPattern`

  ```
  exportPattern("\\.test$")
  ```

- `import`

  ```
  import(mva)
  ```

- `importFrom`

  ```
  importFrom(stepfun, as.stepfun)
  ```

NAMESPACE File Directives (cont.)

- `useDynLib`

  `useDynLib(stats)`

- `S3method`

  `S3method(print, dendrogram)`

NAMESPACE File Directives (cont.)

- `exportClass`, `exportClasses`

  `exportClasses(mle, profile.mle, summary.mle)`

- `exportMethods`

  `exportMethods(AIC, BIC, coef, confint, logLik, ...)`

- `importClassFrom`
- `importMethods`

Name Spaces and Method Dispatch

- S3 dispatch is based on combining generic and class name.
  - no hope of private classes
- Looked up in environment where generic is called.
- Problem: if a package is imported but not attached its methods may not be visible at the call site.

Name Spaces and Method Dispatch (cont.)

- One solution: register S3 methods with the generic.
  - methods are always available to the generic.
  - methods need not be exported
    - * enforces calling methods only via generic.
    - * simplifies author/maintainer's task
- Name space integration is conceptually simpler for S4 classes, methods, and generic functions.
- The current implementation is evolving and may become simpler.

Name Space Odds and Ends

- Name spaces are sealed.
  - cannot add internal variables, imports, exports
  - cannot change values by assignment
  - simplifies implementation
  - helps with byte code compilation
- Exports can be accessed by “fully qualified name”, e.g. `stats::ppr`.
- Internal variables can be accessed using a triple colon, e.g. `stats:::vcov.coxph`

Source Code Analysis

- R provides a powerful infrastructure for managing test code.
- Test code alone cannot cover all possible execution paths.
- Source code analysis provides a complementary approach.
- Source code analysis examines code for possibly erroneous constructs:
  - using variables that are not defined
  - calling functions with the wrong number of arguments
  - calling functions with incorrect argument types
- Name spaces help to make checks more precise.

Some Source Code Analysis Tools

- The package `codetools` provides some experimental tools:
  - `checkUsage` checks individual functions.
  - `checkUsagePackage` checks a loaded package.
- It is likely that these will eventually be merged into the `tools` package.

An Artificial Example

```
g<-function(x, y = TRUE) {
    exp <- y
    w <- x
    y <- x
    if (exp) exp(x+3) + ext(z-3)
    else log(x, bace=2)
}
```

Results of a code analysis:

```
> checkUsage(g,name="g")
g: no visible global function definition for 'ext'
g: no visible binding for global variable 'z'
g: possible error in log(x, bace = 2): unused argument(s)
   (bace ...)
g: local variable 'w' assigned but may not be used
```

An Artificial Example (cont.)

A More Sensitive Analysis:

```
> checkUsage(g,name="g",all=TRUE)
g: local variable 'exp' may shadow global value
g: no visible global function definition for 'ext'
g: no visible binding for global variable 'z'
g: possible error in log(x, bace = 2): unused argument(s)
   (bace ...)
g: local variable 'exp' used as function with no apparent
   local function definition
g: local variable 'w' assigned but may not be used
g: parameter 'y' changed by assignment
```


Issues and Tradeoffs

- Finding the right sensitivity is challenging
  - high sensitivity causes too many false positives
  - low sensitivity misses too many real errors
- Current approach allows tuning by various parameters
- Another option is to attempt to prioritize messages
  - has had some successes on Linux kernel code

Issues and Tradeoffs (cont.)

- More sophisticated checks are needed
- Are user-supplied checks possible?
- Inferred or estimated type information may help
- Intra-procedural analysis may also help
- Partial evaluation may be useful
- May also be able to detect possible inefficiencies
- Source annotation mechanisms may help

Profiling

- `Rprof` takes snapshots of call stack
- `summaryRprof` reports cumulative time in each function.
- Tools to show more detail may help.
- One example: call graph color coded by total time in function.
- Package `proftools` contains some first steps.

Byte Code Compilation

- Compilation can improve efficiency:
  - user code will run faster
  - less native system code needed
- Developing a compiler can clarify the language:
  - features that are hard to compile are hard to understand
- Code analysis is closely related to compilation
  - code analysis for compilation is more conservative
  - code analysis for correctness is more speculative

A Simple Example

- Simplified normal density function:

```
f<-function(x, mu=0,sigma=1)
    (1/sqrt(2 * pi)) *
        exp(-0.5 * ((x - mu)/sigma)^2) / sigma
```

- Compiled with

```
fc<-cmpfun(f)
```


Generated Code

- Byte code and assembly code for a stack machine:

16	1	LDCONST 0.398942	push $1/\sqrt{2\pi}$
16	2	LDCONST -0.5	push constant -0.5
20	3	GETVAR x	get, push x
20	4	GETVAR mu	get, push mu
45		SUB	subtract
20	5	GETVAR sigma	get, push sigma
47		DIV	divide
16	6	LDCONST 2	push 2
48		EXPT	pop x, y, push x^y
46		MUL	multiply
50		EXP	pop x, push e^x
46		MUL	multiply
20	5	GETVAR sigma	get, push sigma
47		DIV	divide
1		RETURN	pop, return value

Generated Code (cont.)

- The compiler
  - folds constant expressions like $1/\sqrt{2\pi}$
  - inlines basic arithmetic functions
- Some timings for 1,000,000 repetitions:

Function	x = 1	x = seq(0,3,len=5)
f	14.62	17.75
fc	3.95	7.67
dnorm	4.59	7.24

- Most improvement comes from constant folding.

The Virtual Machine

- byte code instruction set
- stack architecture
- similar approach to Python, Perl, many Scheme systems
- also related to JVM, .NET
- Alternatives:
  - threaded code (using GCC extensions)
  - generate C code
  - generate JVM, .NET code

Compiler Operation

- Optimizations:
  - constant folding
  - special opcodes for most `SPECIAL`s, many `BUILTIN`s
  - inlines simple `.Internal` calls: `dnorm(y, 2, 3)` is replaced by
    `.Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))`
  - special opcodes for many `.Internal`s
- Compiler currently uses a single recursive pass:
  - fast, but limits optimizations.

Timings and Performance

- well-vectorized code will not improve much
- some examples see substantial speedup:

Context	Speedup
Marching Cubes	2
MCMC	2.5
Dynamic Programming	2.4

- internal version of `lapply` almost not needed anymore
- need to improve variable lookup
- need to improve function calling

Future Directions

- Partial evaluation when some arguments are constants
- Intra-procedural optimizations and inlining
- Run-time specialization
- Vectorized opcodes
- Declarations (sealing, scalars, types, strictness)
- Advice on possible inefficiencies
- C code generation (maybe C−−)
- Compilation technology? (Lisp/ML, Haskell, Self, NESL)

Conclusions

- As R is used for more high level projects, the need for programming support tools increases.
- The highly dynamic nature of R makes creating these tools challenging.
- New language features such as annotations or declarations may help.
- Results so far are quite promising.
- Much more work remains to be done.

Obtaining the Code

- Code is still experimental
- Once it is more stable it will be merged into R
- For now, you can obtain the code at

`http://www.stat.uiowa.edu/~luke/R/`