A Fast
Abstract Syntax Tree
Interpreter for R

Runtime information can be leveraged to create simple, fast, easy to maintain interpreters for real languages.

“70% of data miners prefer R”

By Rexer Analytics, presented at Predictive Analytics World – Boston 2013
1,259 respondents from 75 countries
An R history

1976 S
Chambers @ Bell Labs, then S-Plus (closed-source owned by Tibco)

1993 R
Ihaka and Gentleman, started R as new language at the U of Auckland, NZ

Today, The R project
Core team ~ 20 people, under GPL.
Continued dev of language & libraries:
namespaces ('11), bytecode ('11), 64-bit indexes ('13)

http://www.r-project.org

Vectors

x <- c(2, 7, 9, NA, 5)
c(1, 2, 3) + x[1:3]
x[is.na(x)] <- 0

Functions

p<-function(x=5,...,y=x+1)
  x + c(...) + y
p()  p(1, 2)  p(y=2)
p(y=2, x=1)
p(1, 2, y=0, 3, 4)

Promises

assert(x==2, report(x))

assert<-function(C,P) if(C)print(P)
Referential transparency

\[
x \leftarrow c(0, 1)
\]

\[
f(x)
\]

\[
assert(x[1] == 1)
\]

\[
f \leftarrow function(a) \{ a[1] <- 0 \}
\]

Reflection

\[
\text{with}(fd, carb*den)
\]

\[
with \leftarrow function(data, exp, \ldots)
\]

\[
eval(substitute(exp), data, parent.frame()))
\]

Attributes

\[
x \leftarrow c(1, 2, 3, 4)
\]

\[
\text{attr}(x, \text{"dim"}) \leftarrow c(2, 2)
\]

Of interpreters and men…
Performance & Complexity

Basic textual interpreter
Abstract syntax tree interpreter
Bytecode interpreter
JIT compiler
Tracing JIT compiler
Optimizing JIT

DLS’12

Self-Optimizing AST Interpreters

Thomas Wurthinger, Andreas Woß, Lukas Stadler, Gilles Duboscq, Doug Simon, Christian Wimmer

Showed how to obtain performance for a simple(r) language
We use Truffle as an inspiration to develop techniques for R

Renjin: R in Java
1.8x slower

Worse is better…
Preparation

- Optimizations:
  - allocate slot for a subset of the local variables
  - return statement elision
- Only “compilation” step in FastR 0.168

Code Specialization

- In-place profile-driven code specialization

$$\text{N} \quad \text{N}_1 \quad \text{N}_2 \quad \text{N}'_1 \quad \text{N}'_2$$

- Use values of variables to choose a better implementation
- Simple to implement in single threaded context, modulo some care around recursive functions

Code Specialization

$$f(12, x+1, a=3)$$

Code Specialization

$$f <- \text{function}(a,b,c)\{a+c\}$$
**Code Specialization**

```java
class If {
    RNode condE, trueB, falseB;
    Object execute(Frame f) {
        try {
            val = condE.executeScalarLogical(frame);
        } catch (UnexpectedResult e) {
            cast = ToLogical.mkNode(condE, e.result());
            replaceChild(condE, cast);
            return execute(frame);
        }
        if (val == TRUE) return trueB.execute(f);
        if (val == FALSE) return falseB.execute(f);
        throw unexpectedNA();
    }
}
```

- Specialized code for simple cases, e.g. arithmetics with scalars; function calls scalar vector indexing arithmetics on vectors of same length
- Bounded self-rewriting:
  - Uninitialized node rewrites itself
  - Initialized node rewrites itself when a guard fails
- Rewriting based on events: e.g. a symbol change can re-writes nodes

**Data Specialization**

- A special representation for common cases of data-types
  - Scalar integer with no names, no dimensions, no custom attributes... just the number

Reduces memory overhead
Merges guards needed in (specialized) code

```java
if (v instanceof RInt && v.size() == 1 && v.names() == null && v.dimensions() == null && v.attributes() == null) { ...
if (v instanceof ScalarIntImpl) { ...
```

- Integer ranges
  - A specialized implementation of an integer vector which happens to be a range \(1, 2, \ldots, k \in \mathbb{N}, k > 0\)
  - Saves memory and enables code specialization
  - Java loop over Java integers \(1\) to \(n\)
  - Column selection using a Java loop from \(1\) to \(n\)
Data Specialization

class RIntSimpleRange implements RInt {
    final int to;
    int getInt(int i) { return i + 1; }
    RInt materialize() {
        int[] c = new int[to];
        for (int i = 0; i < to; i++) c[i] = i + 1;
        return RInt.RIntFactory.getFor(c);
    }
}

A range trivially won’t contain an NA, negative values, zero, and bounds check for whole range is constant time

Much faster vector indexing

Data+Code Specialization

class RIntSimpleRange implements RInt {
    final int to;
    int getInt(int i) { return i + 1; }
    RInt materialize() {
        int[] c = new int[to];
        for (int i = 0; i < to; i++) c[i] = i + 1;
        return RInt.RIntFactory.getFor(c);
    }
}

A range trivially won’t contain an NA, negative values, zero, and bounds check for whole range is constant time

Much faster vector indexing

Data+Code Specialization

a <- b+c
... 
... 
o <- a*2

Views

• …delay construction of large data objects
• …are a data-flow representation of vectors
• …avoid unnecessary work if a subset of the data is required
• …avoid allocation of temporary objects
• …permit fusion of multiple data traversals into one

Data+Code Specialization

a <- b+c
... 
... 
o <- a*2

Profiling view

Counts number of accesses to a view

An executable node, on first invocation, creates a profiling view; second invocation, rewrites based on the profile

Heuristics:

Avoid small views; avoid over-used root views; avoid repeating computation; avoid recursive views

FastR 0.168  (speedup 8.5x)
Conclusion

Runtime information can be leveraged to create simple, fast, easy to maintain interpreters for real languages

http://github.com/allr/fastr